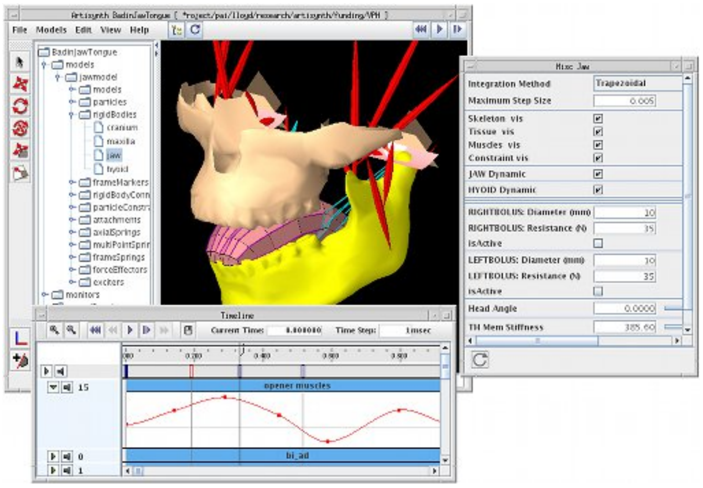
ArtiSynth Description and Design Overview

John E. Lloyd, Ian Stavness

Updated: September 1, 2014

1 Introduction

ArtiSynth (`www.artisynth.org`) 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.



Integration Method: Trapezoidal
Maximum Step Size: 0.005
Skeleton vis ✓
Tissue vis ✓
Muscles vis ✓
Constraint vis ✓
JAW Dynamic ✓
HYOID Dynamic ✓
RIGHTBOLUS: Diameter (mm): 10
RIGHTBOLUS: Resistance (N): 35
isActive ☐
LEFTBOLUS: Diameter (mm): 10
LEFTBOLUS: Resistance (N): 35
isActive ☐
Head Angle: 0.0000
TH Mem Stiffness: 385.60
Current Time: 0.000000
Time Step: 1msec

Figure 1: ArtiSynth screen shot showing a combined jaw-tongue-hyoid model, with the main viewing panel (center), component navigation panel (left), probes arranged on the timeline (bottom), and a panel for adjusting properties (right).

Models can be built from a rich set of components, including particles, rigid bodies, finite elements with both linear and nonlinear materials, point-to-point muscles, and various bilateral and unilateral constraints including contact. These components are arranged in a hierarchy that can be easily navigated and edited through a graphical interface (Figure 1).

Component parameters and state variables are exposed as properties that can be interactively read and adjusted as the simulation proceeds. A state-of-the-art physics engine provides both forward and inverse dynamic simulation, with the latter allowing the computation of the muscle activations required to achieve target motions.

2 General System Design

The ArtiSynth component hierarchy may be connected to various *agent* components, such as control panels, controllers and monitors, and input and output data streams (known as *probes*), which have the ability to control and record the simulation as it advances in time (Figure 2). 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.

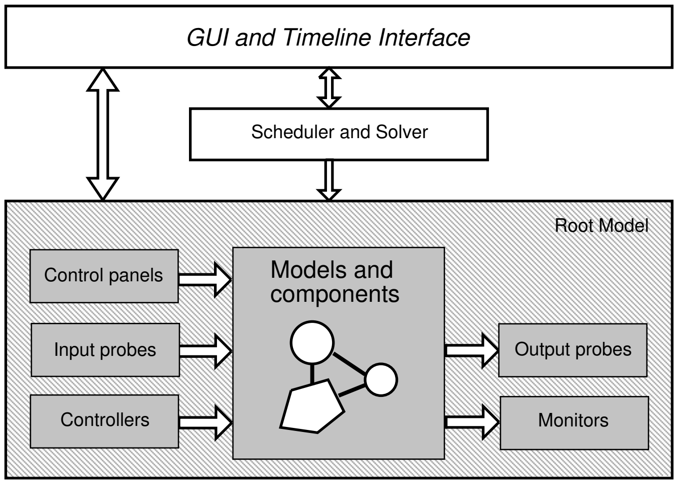


Figure 2: General organization of the ArtiSynth system.

A primary design goal of ArtiSynth is to provide the user with comprehensive interactive simulation control, which is achieved using a rich graphical user interface (GUI). This 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.

2.1 Basic component classes

ArtiSynth is implemented in Java, and every model is formed from a hierarchy of components, each of which is a Java class that is an instance of `ModelComponent` (see Figure 3). Each component has a number (assigned by its parent and returned by `getNumber()`), as well as an optional name returned by `getName()`. A component's parent is returned by `getParent()`, and the methods `connectToHierarchy()` and `disconnectFromParent()` are called whenever the component is added to or removed from the component hierarchy.

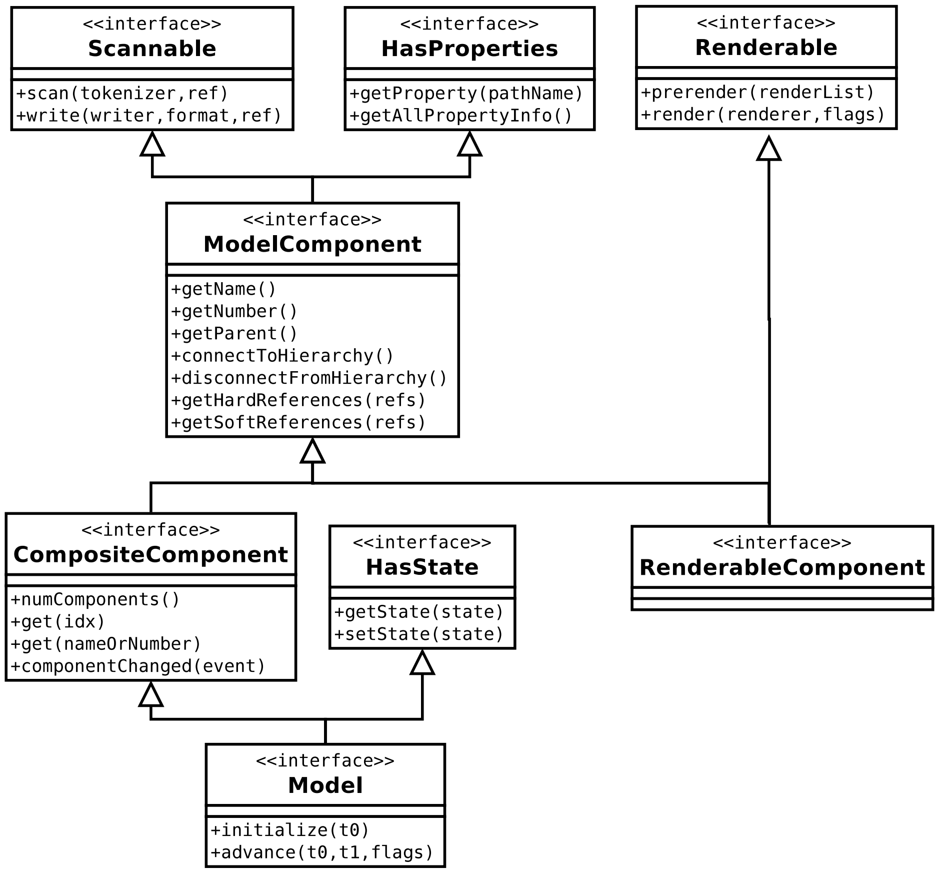


Figure 3: Basic ArtiSynth component classes.

A sub-interface of `ModelComponent` includes `CompositeComponent`, which contains child components. Components which contain state information (such as position and velocity) should extend `HasState`.

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)`. The most common instance of `Model` used in ArtiSynth is `MechModel`, which implements mechanical models consisting of a large variety of components and which advances itself using the physics simulation described in Section 4.

As described in Section 2.4, model components have properties and therefore implement the `HasProperties` interface. They are also responsible for reading and writing their own text file representation, and so also implement the `Scannable` interface that supports serialization to and from a text stream. Components that are capable of rendering themselves to a graphic display must also implement the `Renderable` interface, as described in more detail in Section 3.1.

The base classes for the ArtiSynth components are defined in the package `artisynth.core.modelbase`.

2.2 The component hierarchy

Biomechanical components in an Artisynth model include *dynamic components* such as particles, FEM nodes, or rigid bodies, *force effectors* such as point-to-point muscles (including Hill and other types), linear or nonlinear finite elements, and *constraints* such as joints or collision specifiers. FEM capabilities include support for tetrahedral, hexahedral, and some higher-order elements, along with both linear and large deformation behaviors, including corotated linear [15] and hyperelastic materials. The classes for rigid and particle models are generally defined in the package `artisynth.core.mechmodels`, while the classes for the FEM models are defined in `artisynth.core.femmodels`.

A partial view of a typical component hierarchy (for a jaw-hyoid model) is shown in Figure 4. At the top is a special instance of `Model` known as a `RootModel`, which contains all other models, in addition to special components for interacting with the simulation, such as control panels and probes (Section 3). The jaw-hyoid model itself is an instance of `MechModel` named `JawHyoidModel`, which contains rigid bodies, axial springs (for the point-to-point muscles), and frame markers (which are attached to the rigid bodies and serve as muscle anchor points).

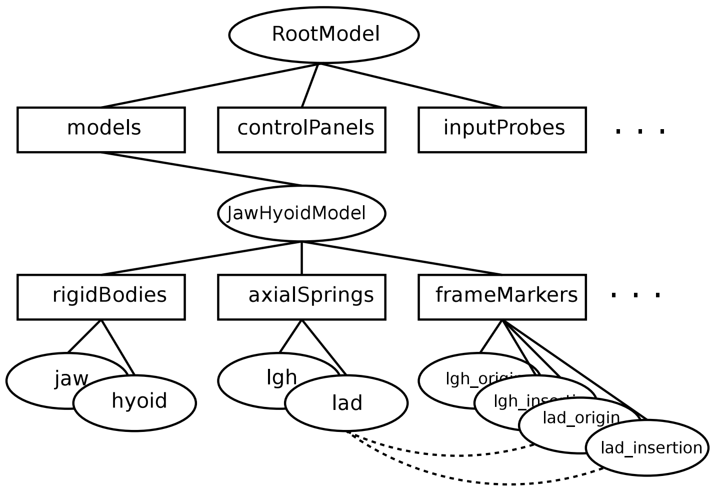


Figure 4: A partial component hierarchy for a Jaw-Hyoid model. The dashed lines show the references from the muscle `lad` to its two attachment markers (other references are omitted for clarity).

Certain components are simply composite components containing lists of other components (indicated by boxes in Figure 4 and including `rigidBodies`, `axialSprings`, and `frameMarkers`). Such components are known as `ComponentLists`, and act as simple containers for sub-components that may be added or removed as required by the modeling application. For convenience, `MechModel` contains a number of predefined lists for different component types, including:

particles	3 DOF particles
rigidBodies	6 DOF rigid bodies
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 (visualization only)

Special methods are provided for adding and removing items from these lists. However, applications are not required to them, and may instead create any component containment structure that is appropriate. If not used, the lists will simply remain empty.

Many components contain references to other components. For example, a point-to-point muscle references its two attachment points, and a rigid body marker references the rigid body to which it is attached. References can be *hard* or *soft*, with a *hard* reference being one which the referring component *must* have, and which if deleted, implies that the referring component should be deleted too. Model components must report all their hard and soft references, via the methods

```
getHardReferences (List<ModelComponent> refs);
getSoftReferences (List<ModelComponent> refs);
```

These methods are used by the structural editing software described in Section 3.6.

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 '/' and begin above the root model. Relative paths omit the leading '/' and can begin lower down in the hierarchy. The absolute path name of the axial spring `lad` in Figure 4 would be

```
/RootModel/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 axial spring `lad` could also be addressed by the path name

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

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

2.3 Model creation

At present, ArtiSynth models are usually created in code, typically by declaring a subclass of the top level `RootModel` (described above) and then using its `build()` method to create the remainder of the component hierarchy. In this sense, the Java code takes the role of a script that creates the various model components and assembles them. This idea is used in other systems, such as the `.mac` file format of ANSYS, which is really a scripting language.

There are serval reasons for creating models in code. First, it tends to be more repeatable, more precise, and (for complex models) easier that using a GUI. Second, models often require specialized code and classes, which cannot be specified easily in a file format.

Biomechanical models will usually contain at least one instance of `MechModel`, which itself provides methods for adding sub-components in a straightforward fashion. A code fragment to construct the portion of the model hierarchy shown in Figure 4 is

```
MechModel jawHyoid = new MechModel ("JawHyoidModel");
RigidBody jaw = RigidBody.createFromMesh (
   "jaw", JawHyoidDemo.class, "jawMesh.obj", 1000, 1);
RigidBody hyoid = RigidBody.createFromMesh (
   "hyoid", JawHyoidDemo.class, "hyoidMesh.obj", 1000, 1);

jawHyoid.addRigidBody (jaw);
jawHyoid.addRigidBody (hyoid);

Muscle lgh = Muscle.createPeck ("lgh", 40, 35, 45, 0.0);
Muscle lad = Muscle.createPeck ("lad", 40, 35, 45, 0.0);

FrameMarker lghOrigin = new FrameMarker("lgh_origin");
FrameMarker lghInsertion = new FrameMarker("lgh_insertion");
FrameMarker ladOrigin = new FrameMarker("lad_origin");
FrameMarker ladInsertion = new FrameMarker("lad_insertion");

jawHyoid.addFrameMarker (
   ladInsertion, hyoid, new Point3d (14.72, 4.47, 8.06));
jawHyoid.addFrameMarker (
   ladOrigin, jaw, new Point3d (7.73, -34.89, 11.66));
jawHyoid.addFrameMarker (
   lghInsertion, hyoid, new Point3d ( 0.99, 1.69, 7.12));
jawHyoid.addFrameMarker (
   lghOrigin, jaw, new Point3d ( 1.99, -33.16, 14.74));

jawHyoid.attachAxialSpring (lghOrigin, lghInsertion, lgh);
jawHyoid.attachAxialSpring (ladOrigin, ladInsertion, lad);

addModel (jawHyoid);
```

First, an instance of `MechModel`, named `JawHyoidModel`, is created. The jaw and hyoid rigid bodies are then generated using the convenience routine `createFromMesh()` that creates rigid bodies given a name, a mesh file located relative to the source for a specified class, a density, and a scale factor. These are then added to the jaw-hyoid model using `addRigidBody()`, which inserts them into the `rigidBodies` component list. The muscles `lgh` and `lad` are created next, using the convenience routine `createPeck()` that assigns names and parameters. Some `FrameMarker` components are then generated to act as the origin and insertion points for these muscles on the jaw and hyoid, and are added to the model using `addFrameMarker()`, which attaches a marker to a particular rigid body at a particular location. Finally, the muscles are added to the model using `attachAxialSpring()`, which connects them to the specified attachment points and inserts them into the `axialSpring` list, and the model itself is added to the RootModel using `addModel()`.

As suggested in the above example, the model construction code can make extensive use of geometric file formats. Surface meshes are often used to describe rigid bodies and can be read in from Alias Wavefront `.obj` files. Similarly, finite element models can be specified using volumetric meshes in either Tetgen or ANSYS file formats.

Once the model generation code has been written and compiled, it can be loaded into ArtiSynth by specifying the name of the RootModel subclass in the GUI. A direct way to do this is to choose `"Load from class"` from the `File` menu, which invokes a dialog allowing the class to be specified. Alternatively, it is possible to configure the system so that the model appears beneath the `Models` menu; selecting the model's menu item will then cause the model to be loaded. Finally, it is also possible to load models using the Jython console (Section 3.3).

Models can also be written to (and read from) files. Artisynth files are given the extension `.art` and use a lightweight text format similar to JSON. This was chosen over XML as it is more compact, faster to parse, and easier to read. Each component is responsible for its own serialization through its implementation of the `Scannable` interface described in Section 2.1. A section of the file representation for the model of the above example looks like:

```
[ name="JawHyoidDemo"
  viewerCenter=[ -0.006 -5.3270499 24.842517 ]
  viewerEye=[ -0.006 -275.82869 24.842517 ]
  ComponentList<artisynth.core.modelbase.Model> [
    name="models"
    MechModel [
      name="JawHyoidModel"
      gravity:Inherited
      stabilization=GlobalMass
      penetrationTol=0.00070010976
      ComponentList<artisynth.core.mechmodels.MechSystemModel> [
        name="models"
      ]
      PointList<Particle> [
      [ name="particles"
        pointDamping:Inherited
      ]
      ComponentList<RigidBody> [
        name="rigidBodies"
        [ mesh="src/artisynth/models/mechdemos/jaw.obj"
          name="jaw"
          axisLength=0
      ...
 ]
```

While it is possible to create a model by directly producing a file, this is generally too tedious to do manually; usage of model files is generally restricted to saving (and later reloading) versions of models that have been changed in some way using the GUI editing methods described in Section 3.6.

The GUI itself can be used to create models directly, but this tends to not be practical for larger models. Instead, the GUI is used more to tweak and adjust existing models, rather than creating them from scratch.

2.4 Properties

ArtiSynth components expose *properties*, which provide a uniform interface for accessing their internal parameters and state. Properties vary from component to component; those for `RigidBody` include `position`, `orientation`, `mass`, and `density`, while those for `Muscle` include `maxForce`, `excitation`, and `damping`. Properties are extremely useful for automatically creating GUI widgets and input and output probes (Section 3.7). They are also useful in automating component serialization.

Each ArtiSynth component implements `HasProperties`, which is defined as

```
interface HasProperties {
   Property getProperty (String name);
   PropertyInfoList getAllPropertyInfo ();
}
```

The method `getProperty()` returns a `Property` handle for the named property, while `getAllPropertyInfo()` returns information for all properties exposed by the class.

A Property handle, in turn, is defined as

```
interface Property {
   Object get();
   void set (Object value);
   Range getRange();
   HasProperties getHost();
   PropertyInfo getInfo();
}
```

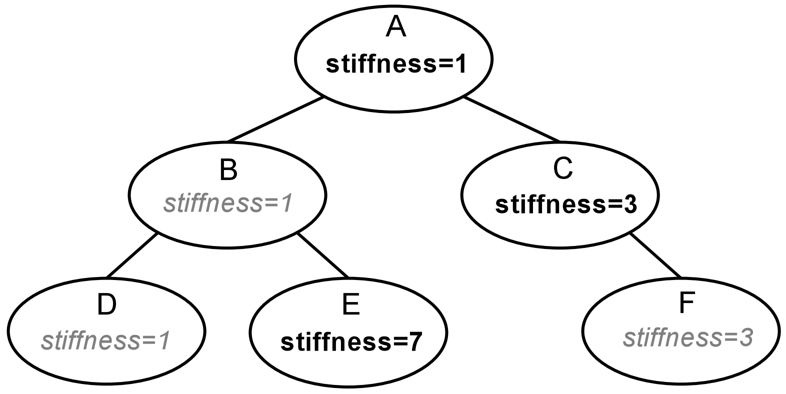


Figure 5: Inheritance of a property named *stiffness* among a component hierarchy. Explicit settings are in bold; inherited settings are in gray italic.

where `get()` and `set()` access the property's value, `getRange()` returns a `Range` object that can be used to determine if a specific value is valid, `getHost()` returns the object to which the property belongs, and `getInfo()` returns detailed information about the property.

The code fragment below shows how to use the property interface to obtain the current excitation value for a `Muscle`:

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

Note, however, that properties are mostly used by generic system code, and the above would more be likely be written directly as

```
double excitation = muscle.getExcitation();
```

Properties are exposed by a class through code contained in the class definition. This includes (a) creating a list of property descriptors within a static code block, and (b) declaring `getXXX()` and `setXXX()` methods for each property's value. The code to expose the property `excitation` within `Muscle` could be as simple as this:

```
double myExcitation;
...
static {
   myProps.add ("excitation", "muscle excitation", 0.0, "[0,1]");
}

public double getExcitation () {
   return myExcitation;
}

public void setExcitation (double e) {
   myExcitation = e;
}
```

Here, `myProps` is the list of property descriptors for the class, and its `add()` method creates and adds an entry with a given name, descriptive comment, default value, and optional range. Given the property's descriptor, the access methods are found automatically using Java reflection.

Properties can be located within the component hierarchy by a path name that consists of a component path name, followed by a colon ':' and the property name. For example, to obtain a property handle for a muscle excitation from a sub-component of a `MechModel`, one could use the fragment

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

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 (5), 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).

2.5 Basic system operation

The typical usage sequence for ArtiSynth is as follows (Figure 6):

1. Create a model as described in Section 2.3.

2. Start the ArtiSynth environment by running `artisynth.core.driver.Launcher`. A convenience command `artisynth` allows this to be done from a shell; alternatively, this can be done from an IDE such as eclipse.

3. Load the desired model into ArtiSynth, as described in Section 2.3.

4. Modify or instrument the model as needed. The GUI (or Jython console, Section 3.3) can be used to set model properties, perform structural edits, assign probes and control panels, etc.

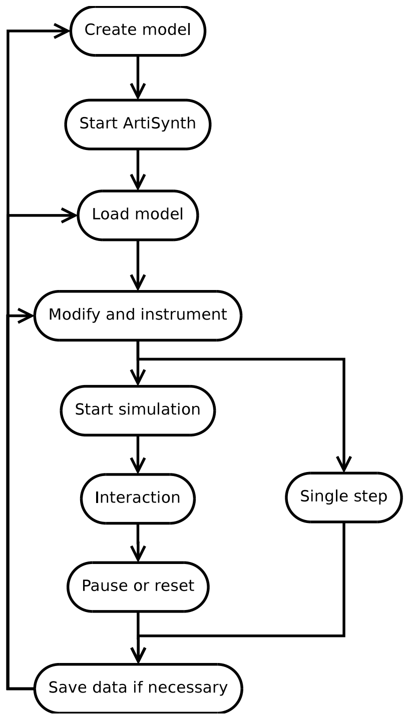


Figure 6: Typical ArtiSynth usage sequence.

5. Start the simulation, using either the play control buttons or a console command. A single step control is also available.

6. Interact with the simulation. Under control of the scheduler, the root model advances itself forward in time, using the physics simulation methods described in Section 4. The viewer is updated at regular intervals, and the time progress of the simulation can be viewed on the timeline. Interaction is possible through property panels or transformers (Section 3).

7. Pause or reset the simulation.

8. Save output data if necessary. If desired, return to steps 1, 3, or 4.

2.6 Packages

ArtiSynth is implemented using a variety of Java packages, shown in Table 1. These are grouped into `maspack`, which provides general classes useful to modeling and simulation, and `artisynth.core`, which contains classes specific to the ArtiSynth environment. The complete Javadocs are available online [4].

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.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

Table 1: The main ArtiSynth packages.

3 Interacting with Models and Simulations

ArtiSynth provides numerous ways for interacting with models and their simulations. More details on the material described here can be found in the ArtiSynth User Interface Guide [5].

3.1 Viewers and rendering

Interaction with an ArtiSynth model is centered around one or more viewing panels, generally known as *viewers*. A main viewer is provided in the center of the main display (Figure 1), and other viewers can be opened in separate windows.

Viewers are based on the `GLViewer` class located in the package `maspack.render`. Graphic rendering is done using OpenGL via the JOGL bindings. As mentioned in Section 2.1, components which are renderable must implement the interface `Renderable` (Figure 3). The two most important methods of this interface are

```
  prerender (RenderList list);

  render (GLRenderer renderer, int flags);
```

`render()` is responsible for the actual 3D rendering of the component to the GL canvas, using resources provided by the renderer (which include interfaces to GL and GLU, along with a large number of generic drawing routines). Because graphic rendering takes place in a separate thread from the simulation, there arises a problem of data consistency, since state information used in rendering (position, in particular) may be modified simultaneously by the simulation. To avoid inconsistent results, components must create a copy of the relevant state information for use inside `render()`. This copying is done in the method `prerender()`, which is called in advance of the render step and in sync with the simulation. More details are given in [6].

Viewers provide the usual interactive ability to adjust the viewpoint and choose between orthogonal and perspective viewing. They also provide the ability to create one or more reference grids which can be turned into clipping planes or clipping slices (Figure 7).

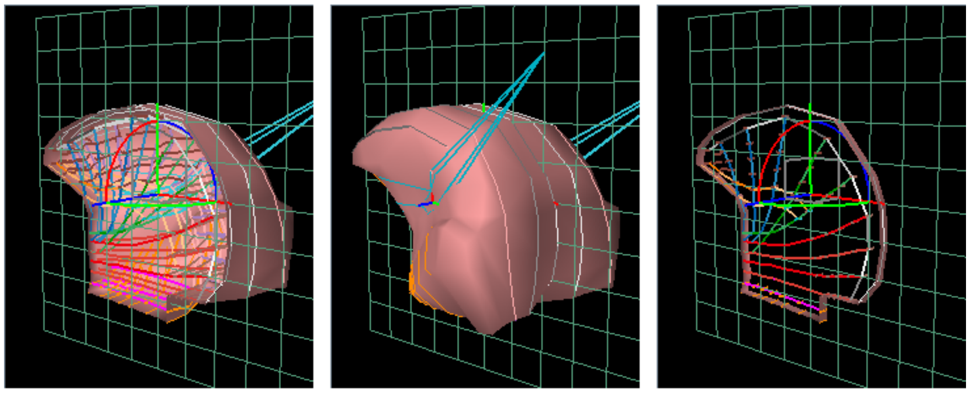


Figure 7: A viewer grid (center), turned into a clipping plane (left) and a clipping slice (right).

3.2 Navigation and selection

Model components in ArtiSynth may be inspected and selected in a variety of ways. A navigation panel (Figure 1, left), exposes the entire component hierarchy and allows the selection of one or more components. Components that are rendered in the viewer may be selected with a left mouse click, and large numbers of components may be selected with a drag selection. Drag selections may be restricted to specific class instances by means of a filter. Another widget, the *selection display*, is located below the main viewer and shows the path name of the most recently selected component. The display also enables component selection, either through typing a path name into it, or by using a *parent* button that successively selects a component's parent.

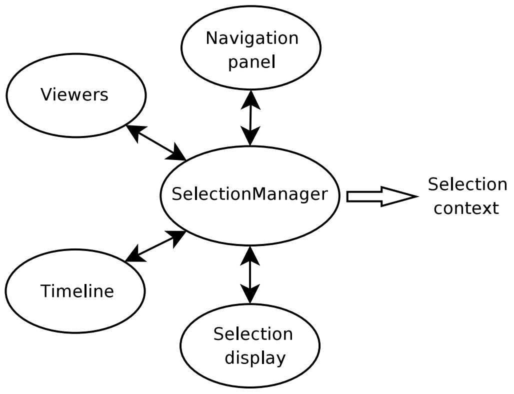
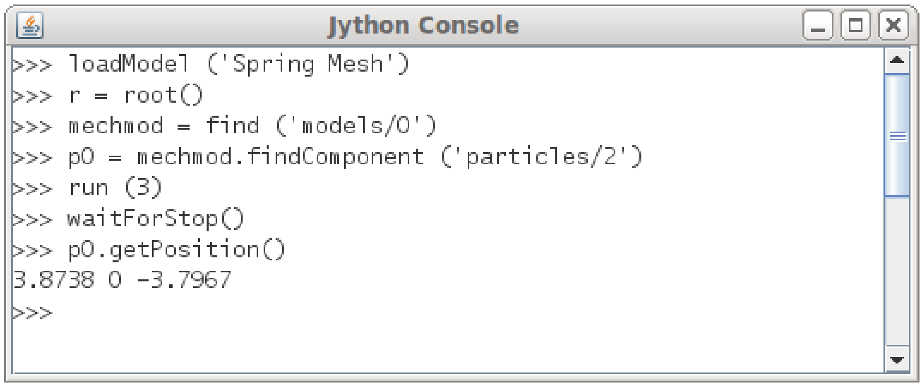


Figure 8: The ArtiSynth selection manager.

Selection is managed by means of a `SelectionManager` (Figure 8), which notifies all selecting agents of a change in selection by any one of them, and maintains the current selection context (i.e., the set of all selected components).

3.3 The Jython console

ArtiSynth has a Jython console that allows access to its operational and component classes through a Jython interface. To start the console, choose `"Show Jython console"` from the `View` menu.



```
>>> loadModel ('Spring Mesh')
>>> r = root()
>>> mechmod = find ('models/0')
>>> p0 = mechmod.findComponent ('particles/2')
>>> run (3)
>>> waitForStop()
>>> p0.getPosition()
3.8738 0 -3.7967
>>>
```

Figure 9: Jython console with sample command sequence.

The Jython console has a number of built-in commands and variables to help load models and run a simulation. Models can be loaded using `loadModel()`. The variable `main` refers to the coordinating object (which is an instance of `Main`) and the variable `sel` is an array containing the current selection context. `root()` returns the currently loaded root model. Components within the root model can be located using `find()`. Simulation can be controlled using `run()`, `pause()`, `waitForStop()`, `reset()`, and `step()`. Waypoints and breakpoints (Section 3.7) can be added using `addWayPoint()` and `addBreakPoint()`.

The built-in `script()` executes a script file within the console, as in the following example:

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

This is similar to the standard Python 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`.

3.4 Transforming geometry

A variety of graphical fixtures, similar to those used in 3D geometric modeling applications, are available to move, rotate, and scale the geometry of selected ArtiSynth components (Figure 10).

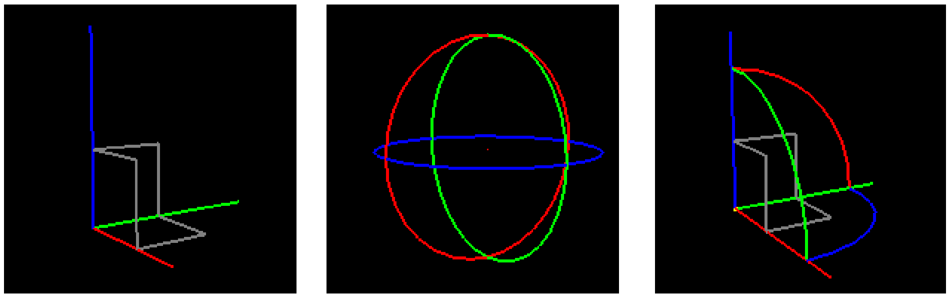


Figure 10: Graphical fixtures for translation, rotation, and combined translation/rotation.

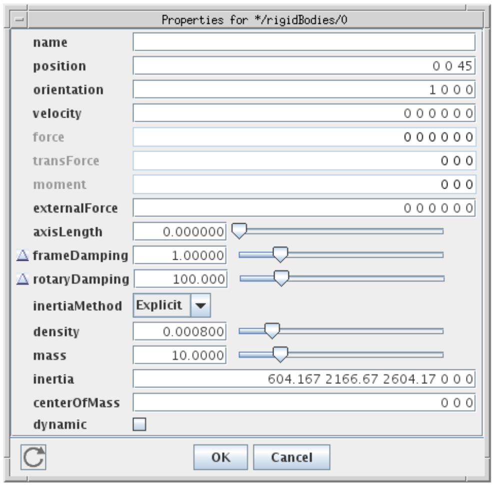
These tools can act on any component that implements `TransformableGeometry`, which provides the method

```
public void transformGeometry (AffineTransform3dBase X);
```

that applies an arbitrary affine transformation to a component's geometry. When applied to a composite component, the transformation is recursively applied to its sub-components.

3.5 Editing properties

A user can edit the properties of one of more components by selecting them and then choosing `"Edit properties ..."` from the context menu (invoked by a right mouse click). This will bring up a property panel such as that shown in Figure 11, which provides a set of widgets for editing individual properties. Render properties are set through a separate panel invoked by



Properties for */rigidBodies/0

name	
position	0 0 45
orientation	1 0 0 0
velocity	0 0 0 0 0 0
force	0 0 0 0 0 0
transForce	0 0 0
moment	0 0 0
externalForce	0 0 0 0 0 0
axisLength	0.000000
frameDamping	1.00000
rotaryDamping	100.000
inertiaMethod	Explicit
density	0.000800
mass	10.0000
inertia	604.167 2166.67 2604.17 0 0 0
centerOfMass	0 0 0
dynamic	☐

Figure 11: Property editing panel for a rigid body.

choosing `"Edit render props ..."`. If more than one component is selected, the property panel presents properties which are common to all components.

An application may also create its own custom property panels, known as *control panels*. Control panels may be created from the GUI by choosing `"Add control panel"` in the `Edit` menu. Property-editing widgets may then be added by selecting properties in specific components. Alternatively, control panels may be created in code, as exemplified by the following fragment (located in the initialization code for a root model):

```
ControlPanel panel = new ControlPanel ("options", "");
panel.addWidget (this, "attachment");
panel.addWidget (this, "collision");
panel.addWidget (this, "models/msmod:integrator");
panel.addWidget (this, "models/msmod:maxStepSize");
addControlPanel (panel);
```

This creates a `ControlPanel` named `"options"` and then populates it with widgets using `addWidget()`. Each widget is specified by simply giving the path name of the property relative to the root model. The first two properties, `attachment` and `collision`, are properties of the root model itself, while the next two belong to descendant components. The appropriate widgets are created automatically using information about the properties' type. When finished, the panel is added to the root model using `addControlPanel()`.

A wide variety of widgets for graphically setting different quantities are defined in the package `maspack.widgets`.

3.6 Structural editing

The graphical interface supports other editing capabilities, including structural edits involving the addition, deletion, and duplication of model components. These capabilities are organized around the current selection context.

Selected components can be deleted by choosing `"Delete"` from the context menu. Applied directly, this could cause an infeasible component structure by removing components that are refereed to by other components. To prevent this, `getHardReferences()` and {`getSoftReferences()` (Section 2.1) are used to determine which components depend on the components being deleted. Dependent components with hard references are deleted along with the other components. For example, deleting a marker to which a point-to-point muscle is attached will cause the point-to-point muscle to also be deleted. Dependent components with soft references are left in place but their special method `updateReferences()` is called to notify them of a structural change in their referenced components.

Selected components can also be duplicated if they implement the `Copyable` interface, which contains methods to ensure that duplication also results in the duplication of additional components needed to preserve a feasible component structure. For example, when duplicating a point-to-point muscle the points to which the muscle is attached are also duplicated.

Other editing operations, particularly those involving the addition of components, operate under the control of an `EditorManager`, which coordinates the actions of various `Editor` objects which serve to perform different editing tasks. When the user invokes a context menu (via a right mouse click), the editor manager creates an *action map* that lists the editing actions and associated editor objects that are appropriate for the current selection context (Figure 12). If the user selects one these actions, the editor is asked to perform the action, which may (optionally) involve creating a persistent `EditingAgent` (which is usually a dialog panel). Editing agents are typically used for operations, such as adding components, that require parameters to be set or items or locations to be selected in the viewer. An editing agent for adding rigid bodies is shown in Figure 13.

Structural changes to the component hierarchy may result in the invalidation of component data, particular cached data that has been precomputed for computational efficiency. Hence a mechanism is provided to notify ancestor objects of changes below them in the component hierarchy. In particular, methods which effect such changes are implemented so as to create a `ComponentChangeEvent` and propagate it up the hierarchy. Ancestor components will then have their `componentChanged()` method called, which clears cached data and propagates the change event upward.

3.7 Probes and the timeline

As mentioned above, it is possible to attach streams of input and output data, called `probes`, to a simulation for purposes of controlling it or recording its results. Input probes may include quantities such as muscle activation levels or forces acting on a body. Output probes may include items such as positions, velocities, or reaction forces. Most probes commonly used

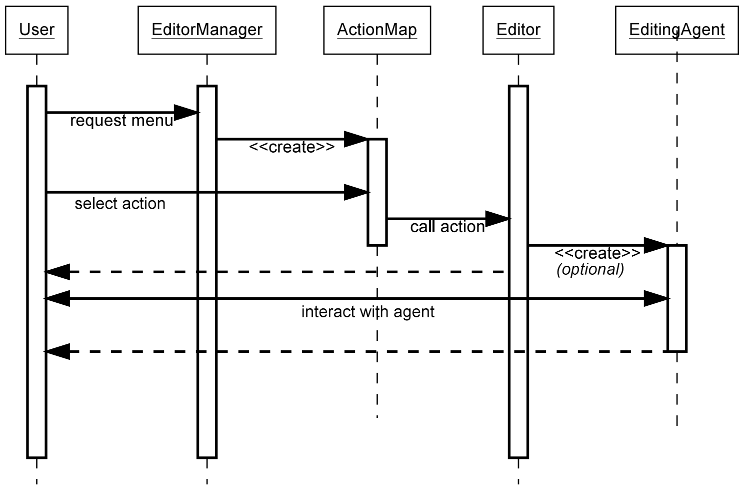


Figure 12: Sequence of operations involving the editor manager.

are instances of `NumericInputProbe` or `NumericOutputProbe`, where the data stream takes the form of a vector of numbers interpolated over time.

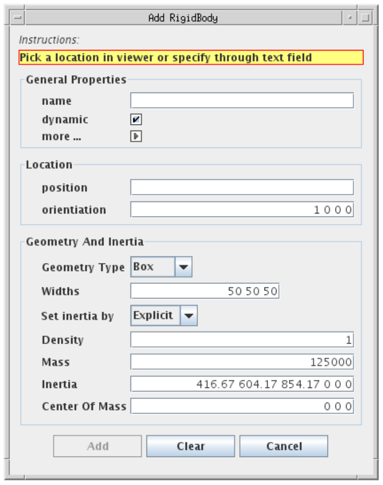
A GUI object known as the *timeline* allows for temporal control of the simulation, through a set of *play control* buttons and a time cursor. It also allows the temporal arrangement of probes to be displayed and adjusted graphically. Figure 14 shows a timeline with one input and two output probes, with two of the probes expanded to show their numeric data. Options exist for adjusting a probe's time interval, editing the numeric data, changing how it is interpolated, creating a large data display, or saving or reading the data from files. Via the timeline, probe data can be examined or adjusted on the "fly".

Probes can be created either graphically or in code. To create a probe graphically, the user chooses `"Add input probe"` or `"Add output probe"` from the `Edit` menu. This will invoke a dialog that allows the user to indicate numeric-type properties in various components to which the probe should be connected. Property information is used to automate much of the process, such as determining the required size for the probe's data vector.

Similarly, probes can be created in code. Within the constructor for a root model, probes can be declared and then added to the root model using either `addInputProbe` or `addOutputProbe`:

```
NumericOutputProbe probe =
   new NumericOutputProbe (
      model, "particles/7:position", "springMeshOut.txt", 0.01);
probe.setStartStopTimes (1, 10);
addOutputProbe (probe);
```

The above snippet creates an output probe which collects the position of particle 7 once every 0.01 seconds, attaches the probe to a file called `springMeshOut.txt`, and sets its start and stop times to 1 and 10 seconds.


Add RigidBody

Instructions:

Pick a location in viewer or specify through text field

General Properties

name	
dynamic	✓
more ...	

Location

position	
orientiation	1 0 0 0

Geometry And Inertia

Geometry Type	Box
Widths	50 50 50
Set inertia by	Explicit
Density	1
Mass	125000
Inertia	416.67 604.17 854.17 0 0 0
Center Of Mass	0 0 0

Figure 13: Editing agent dialog for adding rigid bodies.

The timeline can also be used to set simulation *waypoints* and *breakpoints*. A waypoint is a time location where state is saved when the simulation passes through it, enabling the system to reset itself to that time later using the play control buttons (one cannot normally set the system to an arbitrary time since this requires physical simulation from a known state). A number of uniformly spaced waypoints can be used to create an animation of the simulation. A breakpoint is simply a waypoint at which simulation is also halted.

4 Physical Simulation

Physical simulation is required to advance ArtiSynth models forward in time. In particular, the `advance` method for the top-most mechanical model in the hierarchy needs to solve the second-order ordinary differential equation (ODE) that results from the physics of the mechanical system. How that is done is the focus of this section. Much of the material is taken from [21].

As mentioned in Section 2.2, biomechanical components in Artisynth can be roughly divided into *dynamic components*, *force effectors*, and *constraints*. At present, there are only two types of dynamic component: a six DOF `RigidBody`, and a three DOF `Particle` (the nodes of finite element models are subclasses of `Particle`). Other types of dynamic components, such as reduced coordinate FEM models, may be added in the future.

4.1 The mechanical system ODE

Here we formulate the ODE associated with the mechanical system[1]. Let $\mathbf{q}$ and $\mathbf{u}$ be the generalized positions and velocities of all the dynamical components in the model hierarchy,

[1]Since the ODE contains algebraic constraints, it is technically a *differential algebraic equation*, or DAE

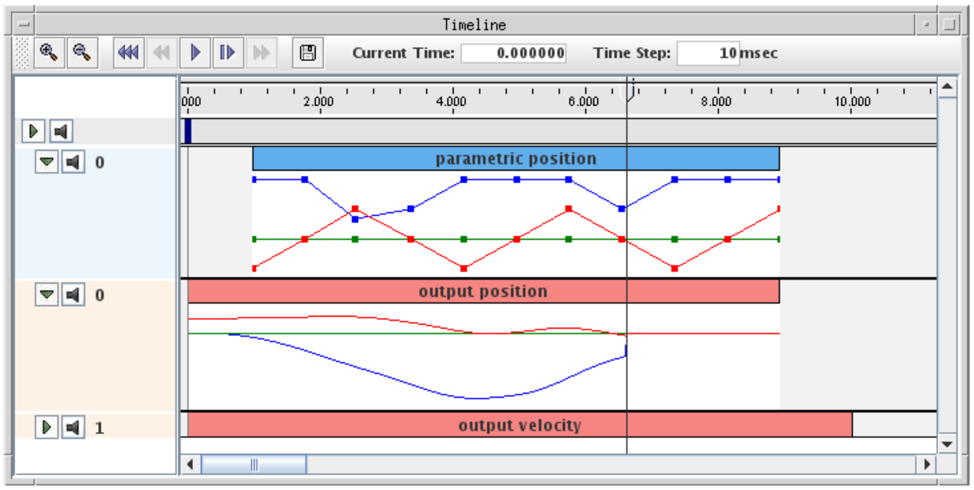


Figure 14: Timeline with probes expanded to show data.

with $\dot{\mathbf{q}}$ related to $\mathbf{u}$ by $\dot{\mathbf{q}} = \mathbf{Q}\mathbf{u}$ ($\mathbf{Q}$ generally equals the identity, except for components such as rigid bodies, where it maps angular velocity onto the derivative of a unit quaternion). Let $\mathbf{f}(\mathbf{q}, \mathbf{u}, t)$ be the force produced by all the force effector components (including the finite elements), and let $\mathbf{M}$ be the (block-diagonal) composite mass matrix. By representing rigid body velocity and acceleration in body coordinates we can ensure that $\mathbf{M}$ is constant. Newton's second law then gives

$$\mathbf{M}\dot{\mathbf{u}} = \mathbf{f}(\mathbf{q}, \mathbf{u}, t). \tag{1}$$

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

$$\mathbf{G}(\mathbf{q})\mathbf{u} = 0, \qquad \mathbf{N}(\mathbf{q})\mathbf{u} \geq 0. \tag{2}$$

Bilateral constraints include rigid body joints, FEM incompressibility associated with the mixed u-P formulation [11], 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) 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.

4.2 Solving the ODE by trapezoidal integration

Solving the equations of motion requires integrating (1) together with (2). ArtiSynth provides a number of integrators, both explicit and implicit, for doing this. When deformable bodies are present, the mechanical system is usually *stiff*, implying the need for an implicit integrator to obtain efficient performance. One of the more commonly used implicit integrators supplied by ArtiSynth is a semi-implicit second-order Newmark integrator [13], with $\gamma = 1/2$ and $\beta = 1/4$, known more generally as the *trapezoidal rule*.

Letting k denote the index of values at a particular time step, and h denote the time step size, this leads to the update rules

$$\mathbf{u}^{k+1} = \mathbf{u}^k + \frac{h}{2}(\dot{\mathbf{u}}^k + \dot{\mathbf{u}}^{k+1}), \quad \mathbf{q}^{k+1} = \mathbf{q}^k + \frac{h}{2}(\mathbf{Q}^k\mathbf{u}^k + \mathbf{Q}^{k+1}\mathbf{u}^{k+1}), \tag{3}$$

subject to

$$\mathbf{G}^{k+1}\mathbf{u}^{k+1} = 0, \qquad \mathbf{N}^{k+1}\mathbf{u}^{k+1} \geq 0. \tag{4}$$

Since $\mathbf{G}$ and $\mathbf{N}$ tend to vary slowly between time steps we can approximate (4) using

$$\mathbf{G}^k\mathbf{u}^{k+1} = \mathbf{g}^k, \qquad \mathbf{N}^k\mathbf{u}^{k+1} \geq \mathbf{n}^k, \tag{5}$$

where $\mathbf{g}^k \equiv -h\dot{\mathbf{G}}^k\mathbf{u}^k$ and $\mathbf{n}^k \equiv -h\dot{\mathbf{N}}^k\mathbf{u}^k$. Likewise, we use the approximation $\mathbf{Q}^{k+1} \approx \mathbf{Q}^k + h\dot{\mathbf{Q}}^k$. For $\dot{\mathbf{u}}^{k+1}$, recalling that $\mathbf{M}$ is constant, an estimate of the (unconstrained) value of $\dot{\mathbf{u}}^{k+1}$ can be obtained from $\dot{\mathbf{u}}^{k+1} \approx \mathbf{M}^{-1}\mathbf{f}^{k+1}$, with $\mathbf{f}^{k+1}$ approximated by the first-order Taylor series

$$\mathbf{f}^{k+1} \approx \mathbf{f}^k + \frac{\partial \mathbf{f}^k}{\partial \mathbf{u}}\Delta\mathbf{u} + \frac{\partial \mathbf{f}^k}{\partial \mathbf{q}}\Delta\mathbf{q}.$$

Placing this into the expression for $\mathbf{u}^{k+1}$ in (3), multiplying by $\mathbf{M}$, noting that

$$\Delta\mathbf{q} = h/2(\mathbf{Q}^k\mathbf{u}^k + \mathbf{Q}^{k+1}\mathbf{u}^{k+1}) \quad \text{and} \quad \Delta\mathbf{u} = \mathbf{u}^{k+1} - \mathbf{u}^k,$$

and incorporating the constraints (5), we obtain the system

$$\begin{pmatrix} \hat{\mathbf{M}}^k & -\mathbf{G}^{kT} & -\mathbf{N}^{kT} \\ \mathbf{G}^k & 0 & 0 \\ \mathbf{N}^k & 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u}^{k+1} \\ \boldsymbol{\lambda} \\ \mathbf{z} \end{pmatrix} + \begin{pmatrix} -\mathbf{M}\mathbf{u}^k - h\hat{\mathbf{f}}^k \\ -\mathbf{g}^k \\ -\mathbf{n}^k \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \mathbf{w} \end{pmatrix},$$
$$0 \leq \mathbf{z} \perp \mathbf{w} \geq 0, \tag{6}$$

where $\mathbf{w}$ is a slack variable, $\boldsymbol{\lambda}$ and $\mathbf{z}$ give the average constraint impulses over the time step, and

$$\hat{\mathbf{M}}^k \equiv \mathbf{M} - \frac{h}{2}\frac{\partial \mathbf{f}^k}{\partial \mathbf{u}} - \frac{h^2}{4}\frac{\partial \mathbf{f}^k}{\partial \mathbf{q}}\mathbf{Q}^{k+1} \quad \text{and} \quad \hat{\mathbf{f}}^k \equiv \mathbf{f}^k - \frac{1}{2}\frac{\partial \mathbf{f}^k}{\partial \mathbf{u}}\mathbf{u}^k + \frac{h}{4}\frac{\partial \mathbf{f}^k}{\partial \mathbf{q}}\mathbf{Q}^k\mathbf{u}^k.$$

The complementarity condition for unilateral constraints is enforced by $0 \leq \mathbf{z} \perp \mathbf{w} \geq 0$. A more detailed explanation of this formulation can be found in [16].

System (6) is a mixed linear complementarity problem, a single solve of which is required to determine $\mathbf{u}^{k+1}$ for each semi-implicit integration step. A fully implicit integrator (not currently implemented in ArtiSynth) would require (6) to be applied iteratively at each time step.

It should be noted that other integration schemes can give rise to same system as (6), only with different values for $\hat{\mathbf{M}}^k$ and $\hat{f}^k$. For example, for the first order semi-implicit Euler scheme, we have

$$\hat{\mathbf{M}}^k \equiv \mathbf{M} - h\frac{\partial \mathbf{f}^k}{\partial \mathbf{u}} - h^2\frac{\partial \mathbf{f}^k}{\partial \mathbf{q}}\mathbf{Q}^{k+1} \quad \text{and} \quad \hat{\mathbf{f}}^k \equiv \mathbf{f}^k - h\frac{\partial \mathbf{f}^k}{\partial \mathbf{u}}\mathbf{u}^k.$$

For finite element models, the localized stiffness and damping matrices are embedded within $\partial \mathbf{f}^k/\partial \mathbf{q}$ and $\partial \mathbf{f}^k/\partial \mathbf{u}$, which means that for models dominated by FEM components $\hat{\mathbf{M}}$ will have an FEM sparsity structure. ArtiSynth FEMs also use a lumped mass model, which ensures that $\mathbf{M}$ is block diagonal and makes it easier to interconnect FEMs with mass-spring and rigid body components.

4.3 Friction, damping, and stabilization

Coulomb (dry) friction can be included by extending (6) to include either a linearized friction cone [3, 16] or a (more approximate but easier to solve) box friction [12]. ArtiSynth currently implements box friction, and since the friction in our system tends to be quite small, we apply this as a post-hoc correction to $\mathbf{u}^{k+1}$ (in the manner of [19]), using a simplified version of (6), with $\mathbf{M}$ instead of $\hat{\mathbf{M}}$ and extra constraints added in the tangential directions at contact points.

Different forms of viscous damping are available, including translational and rotary damping applied directly to particles and rigid bodies, and damping terms embedded in point-to-point springs and muscle actuators. For FEM models, Rayleigh damping is available, which takes the form

$$\mathbf{D}_F = \alpha \mathbf{M}_F + \beta \mathbf{K}_F,$$

where $\mathbf{M}_F$ is the portion of the (lumped) mass matrix associated with the FEM nodes and $\mathbf{K}_F$ is the (instantaneous) FEM stiffness matrix. $\mathbf{D}_F$ is then embedded within the overall system matrix $\partial \mathbf{f}/\partial \mathbf{u}$.

In addition to solving for velocities, it is also necessary to correct positions to account for drift from the constraints, including interpenetrations arising from contact. This can be done at each time step using a modified form of (6) which computes an impulse $\delta \mathbf{q}$ that corrects the positions while honoring the constraints:

$$\begin{pmatrix} \hat{\mathbf{M}}^k & -\mathbf{G}^{kT} & -\mathbf{N}^{kT} \\ \mathbf{G}^k & 0 & 0 \\ \mathbf{N}^k & 0 & 0 \end{pmatrix} \begin{pmatrix} \delta \mathbf{q} \\ \boldsymbol{\lambda} \\ \mathbf{z} \end{pmatrix} + \begin{pmatrix} 0 \\ \boldsymbol{\delta}_g \\ \boldsymbol{\delta}_n \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \mathbf{w} \end{pmatrix},$$
$$0 \leq \mathbf{z} \perp \mathbf{w} \geq 0, \tag{7}$$

where $\boldsymbol{\delta}_g$ and $\boldsymbol{\delta}_n$ are the constraint displacements that must be corrected. If the corrections are sufficiently small, it is often permissible to use $\mathbf{M}$ in place of $\hat{\mathbf{M}}^k$, which improves solution efficiency since $\mathbf{M}$ is constant and block-diagonal.

While such stabilization can sometimes be incorporated directly into (6) [2], we prefer to perform the position correction separately as this (a) allows for the possibility of an iterative correction in the case of larger errors, and (b) explicitly separates the computed velocities from the impulses used to correct errors.

4.4 System solution and complexity

For notational convenience, in this section we will drop the k superscripts from $\hat{\mathbf{M}}$, $\mathbf{G}$, $\mathbf{N}$, $\mathbf{g}$, $\mathbf{n}$, and $\hat{\mathbf{f}}$ in (6) and assume that these quantities are all evaluated at time step k.

System (6) is a large, sparse mixed linear complementarity problem [17] that is not particularly easy to solve, given the unilateral constraints and the fact that $\hat{\mathbf{M}}$ is not block diagonal. If $\hat{\mathbf{M}}$ is symmetric positive definite (SPD), it is equivalent to a convex quadratic program. If there are no unilateral constraints ($\mathbf{N} = \emptyset$), then it reduces to a linear Karush-Kuhn-Tucker (KKT) system.

Generally, $\hat{\mathbf{M}}$ is symmetric (unsymmetric terms sometimes arise from rotational effects but these are usually small enough to ignore) and hence will also be SPD for small enough h (since $\mathbf{M}$ is SPD). However, the resulting system is still harder to solve than non-stiff multibody systems where $\hat{\mathbf{M}} = \mathbf{M}$. This is because $\hat{\mathbf{M}}$, while still sparse, is not block-diagonal. Multibody systems are often solved using the projected Gauss-Seidel method [12]. However, this involves a sequence of iterations, each requiring the computation of $\mathbf{G}_i\hat{\mathbf{M}}^{-1}\mathbf{G}_i^T$ or $\mathbf{N}_i\hat{\mathbf{M}}^{-1}\mathbf{N}_i^T$, which is easy to do for a block-diagonal $\mathbf{M}$ but much more costly for $\hat{\mathbf{M}}$.

At present, ArtiSynth solves (6) by using a Schur complement to turn it into a dense regular linear complementarity problem

$$\begin{gathered}\bar{\mathbf{N}}\mathbf{A}^{-1}\bar{\mathbf{N}}^T\mathbf{z} + \bar{\mathbf{N}}\mathbf{A}^{-1}\mathbf{b} - \mathbf{n} = \mathbf{w} \\ 0 \le \mathbf{z} \perp \mathbf{w} \ge 0\end{gathered} \tag{8}$$

where

$$\mathbf{A} \equiv \begin{pmatrix}\hat{\mathbf{M}} & -\mathbf{G}^T \\ \mathbf{G} & 0\end{pmatrix}, \qquad \bar{\mathbf{N}} \equiv \begin{pmatrix}\mathbf{N} & 0\end{pmatrix}, \quad \mathbf{b} \equiv \begin{pmatrix}\mathbf{M}\mathbf{u}^k + h\hat{\mathbf{f}} \\ \mathbf{g}\end{pmatrix}$$

which is solved using Keller's algorithm [12]. $\mathbf{u}^{k+1}$ and $\boldsymbol{\lambda}$ can then be obtained using back-substitution:

$$\begin{pmatrix}\mathbf{u}^{k+1} \\ \boldsymbol{\lambda}\end{pmatrix} = \mathbf{A}^{-1}\left(\mathbf{b} + \bar{\mathbf{N}}^T\mathbf{z}\right). \tag{9}$$

Keller's algorithm is a pivoting method with an expected complexity of $O(m^3)$, where m is the number of unilateral constraints. In addition, forming (8) and back-solving (9) requires $m + 1$ solves of a system involving $\mathbf{A}$. This is done using the Pardiso sparse direct solver [18], and entails a once-per-step factoring of $\mathbf{A}$, plus $m+1$ solve operations. Experimentally, we have determined that the complexity of factoring $\mathbf{A}$ (using Pardiso) for 3D FEM type problems is roughly $O(n^{1.7})$, where n is the size of $\mathbf{A}$. Similarly, we have also determined that the complexity of solving a factored $\mathbf{A}$ is roughly $O(n^{1.3})$. Hence we can expect the overall complexity for solving (6) to be

$$O(m^3) + mO(n^{1.3}) + O(n^{1.7}).$$

This works well provided that the number of unilateral constraints m is small. To help achieve this, we can sometimes treat the unilateral constraints arising from contact as bilateral constraints (i.e., entries in $\mathbf{G}$) on a per-step basis, as described further in Section 4.7.

4.5 Attachments between bodies

In creating comprehensive anatomical models, it is often necessary to attach various bodies together. Most typically, this is done by connecting points of one body to specific locations on another body. For example, FEM nodes may be attached to particular spots on a rigid body, or to other nodes of a different FEM model.

To facilitate this, ArtiSynth provides the ability to *attach* a dynamic component to one or more *master* components. Let the set of attached components be denoted by β, and the remaining set of unattached *active* components be denoted by α. In general, the velocity $\mathbf{u}_j$ of an attached component is related to the velocities $\mathbf{u}_\alpha$ of the active components by a locally linear velocity constraint of the form

$$\mathbf{u}_j + \mathbf{G}_{j\alpha}\mathbf{u}_\alpha = 0.$$

$\mathbf{G}_{j\alpha}$ will be sparse except for entries corresponding to the master components to which j is attached. Letting $\mathbf{G}_{\beta\alpha}$ denote the matrix formed from $\mathbf{G}_{j\alpha}$ for all attached components, we have

$$\mathbf{I}\mathbf{u}_\beta + \mathbf{G}_{\beta\alpha}\mathbf{u}_\alpha = 0$$

for the constraints that enforce all attachments.

We could simply add these constraints to (6) and solve the resulting system, but this would increase both the system size and solution time. Instead, we use the attachments to actually reduce the size of (6). Consider first the subsystem involving only bilateral constraints. As in Section 4.4, we drop the k superscripts from $\hat{\mathbf{M}}$, $\mathbf{G}$, $\mathbf{g}$, and $\hat{\mathbf{f}}$ in (6) and assume that these quantities are all evaluated at time step k. Letting $\mathbf{b} \equiv \mathbf{M}\mathbf{u}^k + h\hat{\mathbf{f}}$ and partitioning the system into active and attached components yields

$$\begin{pmatrix} \hat{\mathbf{M}}_{\alpha\alpha} & \hat{\mathbf{M}}_{\alpha\beta} & \mathbf{G}_{\alpha\alpha}^T & \mathbf{G}_{\beta\alpha}^T \\ \hat{\mathbf{M}}_{\beta\alpha} & \hat{\mathbf{M}}_{\beta\beta} & \mathbf{G}_{\alpha\beta}^T & \mathbf{I} \\ \mathbf{G}_{\alpha\alpha} & \mathbf{G}_{\alpha\beta} & 0 & 0 \\ \mathbf{G}_{\beta\alpha} & \mathbf{I} & 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u}_\alpha^{k+1} \\ \mathbf{u}_\beta^{k+1} \\ \boldsymbol{\lambda}_\alpha \\ \boldsymbol{\lambda}_\beta \end{pmatrix} = \begin{pmatrix} \mathbf{b}_\alpha \\ \mathbf{b}_\beta \\ \mathbf{g}_\alpha \\ 0 \end{pmatrix}.$$

The identity submatrices make it easy to solve for $\mathbf{u}_\beta^{k+1}$ and $\boldsymbol{\lambda}_\beta$:

$$\mathbf{u}_\beta^{k+1} = -\mathbf{G}_{\beta\alpha}\mathbf{u}_\alpha^{k+1}, \qquad \boldsymbol{\lambda}_\beta = \mathbf{b}_\beta - \hat{\mathbf{M}}_{\beta\alpha}\mathbf{u}_\alpha^{k+1} + \hat{\mathbf{M}}_{\beta\beta}\mathbf{G}_{\beta\alpha}\mathbf{u}_\alpha^{k+1} - \mathbf{G}_{\alpha\beta}^T\boldsymbol{\lambda}_\alpha$$

and hence reduce the system to

$$\begin{pmatrix} \hat{\mathbf{M}}' & \mathbf{G}'^T \\ \mathbf{G}' & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u}_\alpha^{k+1} \\ \boldsymbol{\lambda}_\alpha \end{pmatrix} = \begin{pmatrix} \mathbf{b}' \\ \mathbf{g}_\alpha \end{pmatrix} \tag{10}$$

where

$$\hat{\mathbf{M}}' \equiv \mathbf{P}\hat{\mathbf{M}}\mathbf{P}^T, \quad \mathbf{G}' \equiv \mathbf{G}\mathbf{P}^T, \quad \mathbf{b}' \equiv \mathbf{P}\mathbf{b}, \quad \text{with} \quad \mathbf{P} \equiv \begin{pmatrix} \mathbf{I} & -\mathbf{G}_{\beta\alpha}^T \end{pmatrix}.$$

Similarly, unilateral constraints can be reduced via $\mathbf{N}' = \mathbf{N}\mathbf{P}^T$. The reduction operation can be performed in $O(n)$ time and results in a system that is less sparse but generally faster to solve than the original.

Most attachments in ArtiSynth are point-based, with the most common kind being the attachment of an FEM node to a rigid body or other FEM node. It is also possible to attach FEM nodes to the faces and edges of an FEM element, allowing us to handle the so-called "T-junction" problem and create FEM models with non-conforming element faces. This is quite useful for creating localized subdivisions of particular elements, particularly hexahedrons.

4.6 Parametric control

It is possible to control selected dynamic components parametrically, so that their velocities are explicitly specified by an external source (such as a probe). The forces acting on parametrically controlled bodies then become the unknowns that are solved for. This is useful in situations where certain parts of a system's movement are known a priori and we wish to determine the response of the rest of the system. A dynamic component can be made parametric by setting its `dynamic` property to `false`.

The solution for a system containing parametric components is arranged as follows. Let the set of active components be denoted by α, and the let the parametric components be ρ. As in Section 4.5, we consider first only bilateral constraints, and partition (6) between α and ρ to obtain:

$$\begin{pmatrix} \mathbf{M}_{\alpha\alpha} & \mathbf{M}_{\alpha\rho} & \mathbf{G}_\alpha^T \\ \mathbf{M}_{\rho\alpha} & \mathbf{M}_{\rho\rho} & \mathbf{G}_\rho^T \\ \mathbf{G}_\alpha & \mathbf{G}_\rho & 0 \end{pmatrix} \begin{pmatrix} \mathbf{v}_\alpha \\ \mathbf{v}_\rho \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_\alpha \\ \mathbf{f}_\rho \\ \boldsymbol{\gamma} \end{pmatrix}.$$

Since $\mathbf{v}_\rho$ is given, we can reduce the system to

$$\begin{pmatrix} \mathbf{M}_{\alpha\alpha} & \mathbf{G}_\alpha^T \\ \mathbf{G}_\alpha & 0 \end{pmatrix} \begin{pmatrix} \mathbf{v}_\alpha \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_\alpha - \mathbf{M}_{\alpha\rho}\mathbf{v}_\rho \\ \boldsymbol{\gamma} - \mathbf{G}_\rho \mathbf{v}_\rho \end{pmatrix}.$$

and then solve for $\mathbf{f}_\rho$ as

$$\mathbf{f}_\rho = \mathbf{M}_{\rho\alpha}\mathbf{v}_\alpha + \mathbf{M}_{\rho\rho}\mathbf{v}_\rho + \mathbf{G}_\rho^T \boldsymbol{\lambda}$$

A similar reduction can be applied to unilateral constraints, and an analogous, though more complex, formulation works in the presence of attachments.

4.7 Contact handling

Collision detection can be enabled between any combination of rigid or deformable bodies. It is assumed that the bodies in question contain a triangular surface mesh. A bounding-box hierarchy is used to determine if any two surfaces meshes intersect. If they do, then a tracing algorithm (similar to [14]) is used to compute all the intersection contours between the two meshes as shown in Figure 15. Such contour tracing can be done relatively quickly but does require the use of robust geometry predicates similar to those in [8]; this is particularly

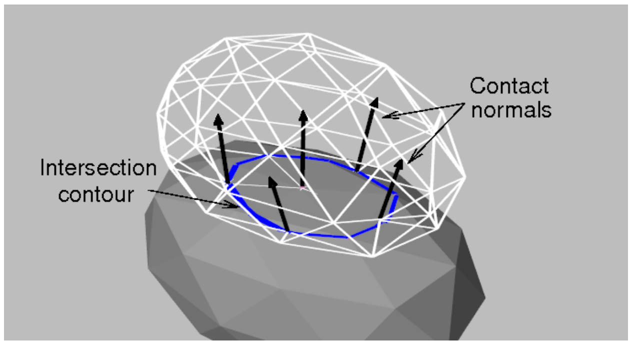


Figure 15: Contact handling between two deformable models (with the topmost rendered as a wireframe), showing the intersection contour (blue) and the contact normals (black lines) of interpenetrating vertices from the upper mesh.

true because collision conditions tend to drive the contacting surfaces into degenerate mesh configurations.

Determining the intersection contour allows us to easily create a set of constraints for correcting the interpenetration and preventing interpenetrating velocities. The way in which is this is done for deformable bodies is described in more detail in [21]. The intersection contour also provides a good estimate of the contact area, which can be used for determining contact pressure.

As mentioned in Section 4.4, the solution time of (6) can be greatly improved if some contact constraints can be temporarily treated as bilateral constraints within a particular time step. By default, ArtiSynth does this for contact involving deformable bodies, since such bodies have many degrees of freedom and their contact constraints tend to be somewhat decoupled. To prevent sticking, each contact's vertex-face pair is stored between time steps, and if it reappears in the next step, it is used as a contact constraint only if its corresponding $\boldsymbol{\lambda}$ value computed in (6) is ≥ 0, implying that there is no force trying to make it separate. This is in effect an active set method, with the active set used to solve (6) being updated between steps.

4.8 Physics engine summary

The ArtiSynth physics engine, using the trapezoidal integrator, is summarized below. It is applicable to most second-order mechanical systems which use a Lagrangian representation of component state. For other ArtiSynth integrators, the structure is similar.

1. Compute contacts (as per Section 4.7) and the bilateral and unilateral constraint matrices $\mathbf{G}^k$ and $\mathbf{N}^k$.
2. Correct positions $\mathbf{q}^k$ to remove interpenetration and drift errors, using (7).
3. If necessary, adjust $\mathbf{G}^k$ and $\mathbf{N}^k$ to reflect changes in $\mathbf{q}$.
4. Solve for $\mathbf{u}^{k+1}$ using (6).
5. Adjust velocities $\mathbf{u}^{k+1}$ for dry friction, as described in Section 4.3.
6. Compute new positions: $\mathbf{q}^{k+1} = \mathbf{q}^k + h/2(\mathbf{Q}^{k+1}\mathbf{u}^{k+1} + \mathbf{Q}^k\mathbf{u}^k)$.

4.9 Interfacing to ArtiSynth

This physics engine is implemented by the class `MechSystemSolver`, which is invoked by the top-level mechanical models to advance themselves forward in time. To communicate with the solver, these models must implement `MechSystem` (Figure 16), which provides the quantities needed for computing the simulation, including those found (6), and then setting the resulting state. Simulation quantities include state, mass, forces, force Jacobians, and constraint information.

The `MechSystem` interface provides a clean separation between the physics simulation and the ArtiSynth component structure, allowing the possibility for new and different simulation mechanisms to be used in the future.

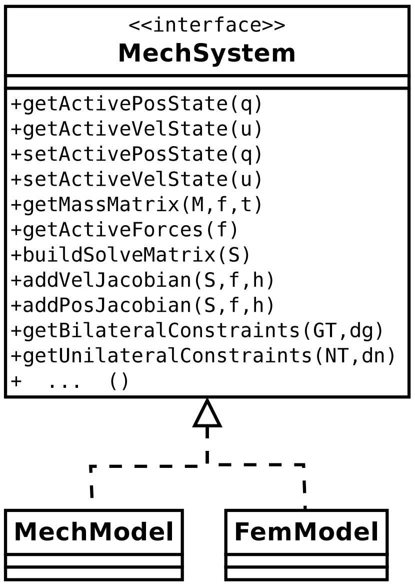


Figure 16: Top-level ArtiSynth mechanical models must implement `MechSystem`, shown partially here.

4.10 Inverse modeling

ArtiSynth provides inverse modeling capabilities that allow the computation of the muscle activations needed to perform a prescribed task. This is a useful feature as it can often be quite difficult to either measure such activations experimentally, or to estimate them by hand for a particular model. Activations are determined using a quadratic program that minimizes the errors for a desired task trajectory while resolving redundancies. The description here is based on material in [20], Chapter 4.

Inverse modeling is currently only supported for systems with bilateral constraints; the addition of unilateral constraints leads to a more difficult mathematical programming problem with complementarity constraints (MPCC).

For inverse modeling purpose, the mechanical system forces are divided into *passive* and *active* components, so that

$$\mathbf{f} = \mathbf{f}_p(\mathbf{q}, \mathbf{u}, t) + \mathbf{f}_a(\mathbf{q}, \mathbf{u}, \mathbf{a}(t))$$

where $\mathbf{a}$ is a vector of muscle activation levels. By convention, the activation levels themselves are assumed to be bounded between 0 and 1. A key assumption is that $\mathbf{f}_a$ is locally linear with respect to $\mathbf{a}$, so that

$$\mathbf{f}_a = \Lambda(\mathbf{q}, \mathbf{u})\mathbf{a},$$

where Λ is a matrix.

For implicit integration, we need to estimate

$$\mathbf{f}^{k+1} = \mathbf{f}_p^{k+1} + \mathbf{f}_a^{k+1} = \mathbf{f}_p^{k+1} + \Lambda^{k+1}\mathbf{a}^{k+1}.$$

The actuations $\mathbf{a}^{k+1}$ are what we are computing, and we employ the approximation

$$\Lambda^{k+1}\mathbf{a}^{k+1} \approx \Lambda^k\mathbf{a}^{k+1} + h\mathbf{c}^k$$

where $\mathbf{c}^k \equiv \dot{\Lambda}^k\mathbf{a}^k$. For notational simplicity, we shall let $\mathbf{a} \equiv \mathbf{a}^{k+1}$ in the remainder of this section.

The velocities $\mathbf{u}^{k+1}$ and constraint impulses $\boldsymbol{\lambda}$ can be determined from (6), which becomes a linear system since we are not considering unilateral constraints:

$$\begin{pmatrix} \hat{\mathbf{M}}^k & -\mathbf{G}^{kT} \\ \mathbf{G}^k & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u}^{k+1} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{M}\mathbf{u}^k + h\hat{\mathbf{f}}^k + \mathbf{c}^k + \Lambda^k\mathbf{a} \\ \mathbf{g}^k \end{pmatrix} \tag{11}$$

Now assume that we wish to compute the activations to best track a movement specified by a target velocity $\mathbf{v}_*$, with respect to some target velocity space $\mathbf{v}$ that is related to the system velocities $\mathbf{u}$ via a Jacobian matrix $\mathbf{J}_m$:

$$\mathbf{v} = \mathbf{J}_m\mathbf{u}.$$

For time step $k+1$, it is easy to see from (11) that $\mathbf{u}^{k+1}$ is linear with respect to $\mathbf{a}$, so that

$$\mathbf{u}^{k+1} = \mathbf{u}_0 + \mathbf{H}_u\mathbf{a}$$

where $\mathbf{u}_0$ is the solution of $\mathbf{u}^{k+1}$ for (11) with $\mathbf{a}$ set to zero, and each column j of $\mathbf{H}_u$ is the solution of $\mathbf{u}^{k+1}$ for (11) with a right hand side of

$$\begin{pmatrix} \Lambda^k \mathbf{e}_j \\ 0 \end{pmatrix}, \tag{12}$$

with $\mathbf{e}_j$ denoting the elementary unit vector. The velocity tracking error $\mathbf{v}_* - \mathbf{J}_m \mathbf{u}^{k+1}$ then becomes

$$\bar{\mathbf{v}} - \mathbf{H}_m \mathbf{a},$$

with

$$\bar{\mathbf{v}} \equiv \mathbf{v}_* - \mathbf{J}_m \mathbf{u}_0 \quad \text{and} \quad \mathbf{H}_m \equiv \mathbf{J}_m \mathbf{H}_u,$$

which can be minimized by minimizing the quadratic form

$$\phi_m(\mathbf{a}) \equiv \frac{1}{2} \|\bar{\mathbf{v}} - \mathbf{H}_m \mathbf{a}\|^2. \tag{13}$$

For some applications, we may also wish to specify a constraint force target $\boldsymbol{\xi}$ that is related to the constraint impulses $\boldsymbol{\lambda}$ by

$$h\boldsymbol{\xi} = \mathbf{J}_c \boldsymbol{\lambda},$$

with $\mathbf{J}_c$ being an appropriate Jacobian. Again, from (11), it is easy to see that $\boldsymbol{\lambda}$ is linear with respect to $\mathbf{a}$, so that

$$\boldsymbol{\lambda} = \boldsymbol{\lambda}_0 + \mathbf{H}_\lambda \mathbf{a}$$

where $\boldsymbol{\lambda}_0$ is the solution of $\boldsymbol{\lambda}$ for (11) with $\mathbf{a}$ set to zero, and each column j of $\mathbf{H}_\lambda$ is the solution of $\boldsymbol{\lambda}$ for (11) with a right hand side of (12). The constraint force tracking error $h\boldsymbol{\xi} - \mathbf{J}_c \boldsymbol{\lambda}$ then becomes

$$\bar{\boldsymbol{\lambda}} - \mathbf{H}_c \mathbf{a},$$

with

$$\bar{\boldsymbol{\lambda}} \equiv h\boldsymbol{\xi} - \mathbf{J}_m \boldsymbol{\lambda}_0 \quad \text{and} \quad \mathbf{H}_c \equiv \mathbf{J}_c \mathbf{H}_\lambda,$$

which can be minimized by minimizing the quadratic form

$$\phi_c(\mathbf{a}) \equiv \frac{1}{2} \|\bar{\boldsymbol{\lambda}} - \mathbf{H}_c \mathbf{a}\|^2. \tag{14}$$

In order to resolve situations where the number of actuators is redundant, we also include a weighted l^2-norm regularization term, $\frac{1}{2}\mathbf{a}^T \mathbf{W}^{-1} \mathbf{a}$, where $\mathbf{W}$ is a diagonal matrix of muscle cross-sectional areas (CSA), in order to select the most efficient set of activations [1].

Combining the movement and constraint force goals, regularization, and muscle activations bounds, along with appropriate weighting terms, we arrive at the following quadratic program:

$$\begin{aligned} \min_{\mathbf{a}} \quad & w_m \phi_m(\mathbf{a}) + w_c \phi_c(\mathbf{a}) + \frac{w_a}{2} \mathbf{a}^T \mathbf{W}^{-1} \mathbf{a} \\ \text{subject to} \quad & 0 \le \mathbf{a} \le 1 \end{aligned} \tag{15}$$

where w_m, w_c, and w_a are weights used to trade-off between cost terms.

The optimization program (15) is solved at the beginning of each time step in order to determine the activations to be used in the forward dynamics simulation. The ArtiSynth system solver is used to compute $\bar{\mathbf{v}}$, $\mathbf{H}_m$, $\bar{\boldsymbol{\lambda}}$, and $\mathbf{H}_c$. The resulting quadratic program is dense but tends to be small since its dimension is the size of $\mathbf{a}$, i.e. the number of activations being solved for. The quadratic program is also convex, which means it can be solved as a linear complementarity problem, which is done using the ArtiSynth implementation of Keller's algorithm [12].

5 Existing Biomechanical Models

ArtiSynth has been used to produce a number of biomechanical models and associated studies, mostly focused on the oral and upper airway region.

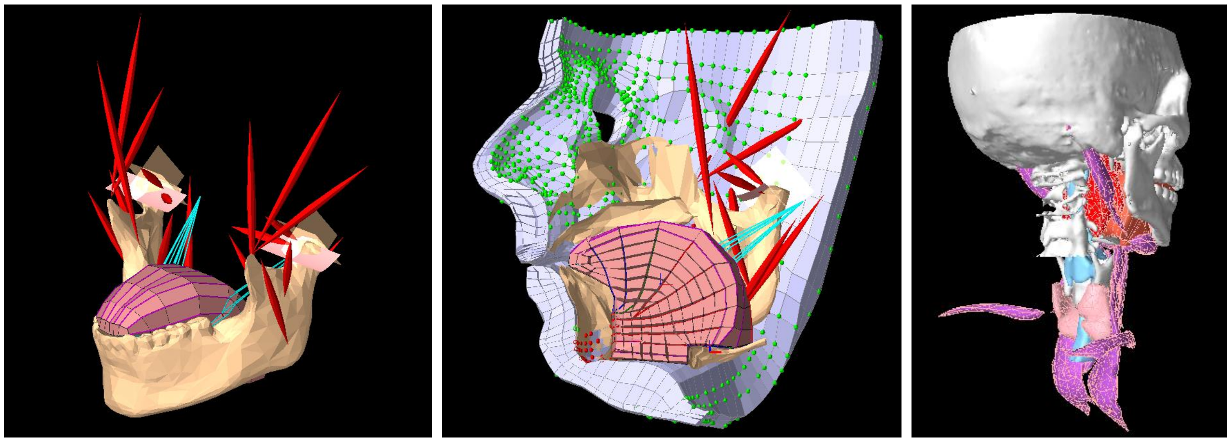


Figure 17: Some example models developed with ArtiSynth. Left: an integrated jaw-tongue-hyoid model. Center: the extension of this to include face and lips. Right: a full airway model currently under development.

One of the first models to be produced was one of the jaw and hyoid structures used to simulate chewing [9]. This was then used to study the modeling possibilities for surgical resection and reconstruction of the jaw [10]. The jaw-hyoid model was combined with a 3D finite element tongue model [7] to create an fully integrated jaw-tongue-hyoid model [21] (Figure 17). An extension of this to include the face and lips is currently in progress [22], and a video of the work can be seen in [23].

Efforts are continuing to create a comprehensive model of upper airway anatomy, including the soft palate, pharyngeal wall, and laryngeal structures, for a variety of medical and research purposes. Details can be found at `www.artisynth.org/opal`.

References

[1] R. Ait-Haddou, A. Jinha, W. Herzog, and P. Binding. Analysis of the force-sharing problem using an optimization model. *Mathematical biosciences*, 191(2):111–122, 2004.

[2] Mihai Anitescu and Gary D. Hart. A constraint-stabilized time-stepping approach for rigid multibody dynamics with joints, contact and friction. *International Journal for Numerical Methods in Engineering*, 60(14):2335–2371, 2004.

[3] 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.

[4] ArtiSynth Project. Artisynth Java API. `http://www.artisynth.org/doc/javadocs/index.html`.

[5] ArtiSynth Project. Artisynth User Interface Guide. `http://www.artisynth.org/doc/html/uiguide/uiguide.html`.

[6] ArtiSynth Project. Maspack Reference Manual. `http://www.artisynth.org/doc/html/maspack/maspack.html`.

[7] Stephanie Buchaillard, Pascal Perrier, and Yohan Payan. A biomechanical model of cardinal vowel production: Muscle activations and the impact of gravity on tongue positioning. *Journal of the Acoustical Society of America*, 126(4):2033–2051, 2009.

[8] Herbert Edelsbrunner and Ernst Peter Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, 1990.

[9] Alan G. Hannam, Ian Stavness, John E. Lloyd, and Sidney Fels. A dynamic model of jaw and hyoid biomechanics during chewing. *J Biomechanics*, 41(5):1069–1076, 2008.

[10] Alan G. Hannam, Ian Stavness, John E. Lloyd, Sidney Fels, Art Miller, and Don Curtis. A comparison of simulated jaw dynamics in models of segmental mandibular resection versus resection with alloplastic reconstruction. *Journal of Prosthetic Dentistry*, 104(3):191–198, 2010.

[11] T. J. R. Hughes. *The finite element method: linear static and dynamic finite element analysis*. Dover Publications, New York, 2000.

[12] 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.

[13] Christoph Lunk and Benrd Simeon. Solving constrained mechanical systems by the family of newmark and α-methods. *Journal of Applied Mathematics and Mechanics (ZAMM)*, 86(10):772–784, 2006.

[14] M.J. Berger M. J. Aftosmis and J.E. Melton. Robust and efficient cartesian mesh generation for component-based geometry. *AIAA Journal*, 36(6):952–960, 1998.

[15] Matthias Müller and Markus Gross. Interactive virtual materials. In *GI '04: Proceedings of Graphics Interface*, pages 239–246, 2004.

[16] 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.

[17] Richard E. Stone Richard W. Cottle, Jong-Shi Pang. *The Linear Complementarity Problem*. Academic Press, 1992.

[18] Olaf Schenk and Klaus Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Gener. Comput. Syst.*, 20(3):475–487, 2004.

[19] Tamar Shinar, Craig Schroeder, and Ron Fedkiw. Two-way coupling of rigid and deformable bodies. In *SCA '08: Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 95–103, 2008.

[20] Ian Stavness. *Byte your tongue : a computational model of human mandibular-lingual biomechanics for biomedical applications*. PhD thesis, University of British Columbia, Dept. of Electrical and Computer Engineering, 2010.

[21] Ian Stavness, John E. Lloyd, Yohan Payan, and Sidney Fels. Coupled hard-soft tissue simulation with contact and constraints applied to jaw-tongue-hyoid dynamics. *International Journal of Numerical Methods in Biomedical Engineering*, in press, 2010.

[22] Ian Stavness, John E. Lloyd, Yohan Payan, and Sidney Fels. Dynamic hard-soft tissue models for orofacial biomechanics. In *ACM SIGGRAPH Talks*, page 1, 2010.

[23] Ian Stavness, John E. Lloyd, Yohan Payan, and Sidney Fels. Dynamic hard-soft tissue models for orofacial biomechanics. 2010.