

# ArtiSynth: A Fast Interactive Biomechanical Modeling Toolkit Combining Multibody and Finite Element Simulation

John E. Lloyd, Ian Stavness, and Sidney Fels

**Abstract** ArtiSynth ([www.artisynth.org](http://www.artisynth.org)) is an open source, Java-based biomechanical simulation environment for modeling complex anatomical systems composed of both rigid and deformable structures. 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. A state-of-the-art physics simulator provides forward simulation capabilities that combine multibody and finite element models. Inverse simulation capabilities allow the computation of the muscle activations needed to achieve prescribed target motions. ArtiSynth is highly interactive, with component parameters and state variables exposed as properties that can be interactively read and adjusted as the simulation proceeds. Streams of input and output data, used for controlling or observing the simulation, can be viewed, arranged, and edited on an interactive timeline display, and support is provided for the graphical editing of model structures.

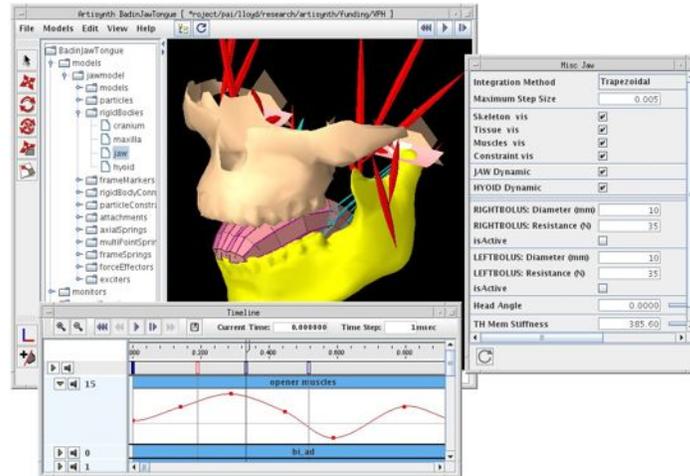
## 1 Introduction

The ArtiSynth modeling platform has been developed at the University of British Columbia for the past several years. Originally created for modeling the mechanics of speech production, the system has evolved into a general biomechanical simulation environment, with most usage to date focussed on modeling the head and neck region for research in physiology, medicine, dentistry, and linguistics (Figure 1). Many applications are directed toward understanding and treating pathologies of the oral cavity and upper airway, including obstructive sleep apnea [19], swallow-

---

Lloyd and Fels: Department of Electrical and Computer Engineering  
University of British Columbia, e-mail: [lloyd@cs.ubc.ca](mailto:lloyd@cs.ubc.ca), [ssfels@ece.ubc.ca](mailto:ssfels@ece.ubc.ca)

Stavness: Department of Bioengineering  
Stanford University, e-mail: [stavness@stanford.edu](mailto:stavness@stanford.edu)



**Fig. 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).

ing disorders [22], and speech pathologies [9], and evaluating the consequences of surgical interventions [12].

Modeling the head and neck region is challenging because it is a highly complex area combining both deformable tissue (e.g., tongue, soft palate) and rigid bony structures (e.g., mandible, hyoid, hard palate). Traditionally, deformable structures are modeled using finite element method (FEM) techniques [7, 8], while rigid structures are modeled using multibody approaches [29]. Most commercially available simulation packages tend to be oriented toward either one approach or the other (e.g., ANSYS and SIMULA for FEM, SolidWorks and ADAMS for multibody) and creating hybrid models within either is generally not easy.

In developing ArtiSynth, our goal has been to create a highly interactive simulation environment, tailored to the needs of biomedical researchers, that combines FEM and multibody methods in a convenient fashion. This contrasts with commercial FEM simulation software that often uses a non-interactive pre-process—simulate—post-process framework. Writing our own software has allowed us to implement novel mechanisms for interactivity (Section 3), create custom modeling components (such as specialized muscle models or constraints), and incorporate state-of-the-art methods for simulation (such as fast solvers and new collision handling techniques) that are not yet available in commercial packages. Finally, by making ArtiSynth available as an open source project, we are able to provide the research community a cost effective platform for collaborative development.

Key features of ArtiSynth include:

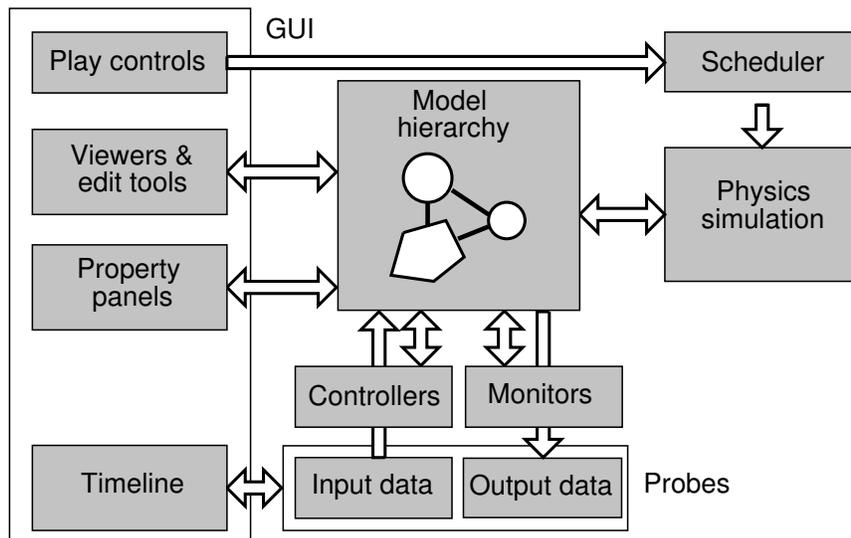
- A comprehensive API for creating and interconnecting anatomical models from rigid bodies, joints, finite element models, point-to-point muscles (including Hill and other types), particles, etc.
- Tetrahedral, hexahedral, and higher-order FEM elements, along with materials including co-rotated linear, hyperelastic, viscoelastic, and transverse isotropic.
- Fast simulation using state-of-the-art physics simulation that handles bilateral and unilateral constraints, contact and friction.
- Inverse simulation capabilities to determine the muscle activations required to achieve particular tasks.
- A Jython console for dynamic interaction and scripting.
- A highly interactive environment, with GUI support for setting properties, transforming and editing structure, and viewing and editing input and output data on a timeline.
- Extensibility, with the ability to easily add custom and special purpose components or override aspects of the simulation behavior.

Other open source systems for biomechanics have been developed in recent years. These include: OpenSim [15], which provides a highly accurate multibody simulator and lumped line-based muscle models for performing musculoskeletal analysis; FEBio [26], a finite element package that uses a traditional pre-process—simulate—post-process framework with special support for tissue modeling and some support for rigid bodies, contact and constraints. Systems geared toward surgical training include Gipsi [11] and Spring [24]. Sofa [2] provides a general software architecture in which models can be partitioned into different sub-models for simulating appearance, behavior, and/or haptic response. ArtiSynth and Sofa both provide the ability to combine multibody and FEM models, with Sofa targeted more towards realtime applications such as surgical simulation and ArtiSynth directed more towards precise physiological modeling. Sofa uses a very general scene-graph arrangement for creating models, while ArtiSynth employs a more traditional component-based approach. ArtiSynth also supplies a comprehensive simulation engine (Section 4) that can solve any Lagrangian based mechanical system containing both bilateral and unilateral constraints.

The remainder of this chapter is organized as follows. An overview of ArtiSynth's design is given in Section 2, followed by a description of the user interactivity features (Section 3), the forward-dynamics physics simulation (Section 4), and inverse simulation capabilities (Section 5). Section 6 gives a concise overview of some of the models that have been created to date, and concluding remarks and a discussion of future work is given in Section 7.

## 2 General System Design

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). Figure 2 provides an overview schematic of the system's organization. At the center are the models, composed of a hierarchy of components. The GUI allows a user to view and edit the model hierarchy, using one or more OpenGL-based *viewers*, along with selection, navigation, and editing tools that allow components to be added, modified, or deleted. The properties of selected components can be modified using *property panels*. Play controls allow simulation to be started, paused, single stepped, or reset. Simulation proceeds under the control of a *scheduler*, which coordinates the actions of the a *physics simulator* that advances the models through time. Streams of input and output data (known as *probes*) can be attached to the model to control or observe the simulation as it proceeds. Typically, input probe data consists of external forces, muscle activation levels, or kinematically specified motions, while output probe data contains simulation results such as positions, velocities, muscle forces, or reaction forces. Probes and their data can be observed, edited, and temporally arranged using a GUI *timeline* component. Applications can also add *controllers* to preprocess and provide control inputs at the beginning of each time step, as well as *monitors* to collect and post-process observed data at the end of each time step.



**Fig. 2** General organization of the ArtiSynth system.

### 2.1 Java implementation and basic component classes

ArtiSynth is implemented in Java. This decision was made to facilitate portability, particularly for the graphical interface, across various system platforms. While Java is slower than other languages (typically by a factor of two for optimized code after just-in-time compilation), this is not too problematic since the major computational bottleneck is usually the linear solves required by the physics simulator (Section 4). These linear solves, in turn, are done using direct solvers compiled in native code.

Every ArtiSynth 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 `connectToParent(parent)` and `disconnectFromParent()` are called when the component is added to or removed from the hierarchy.

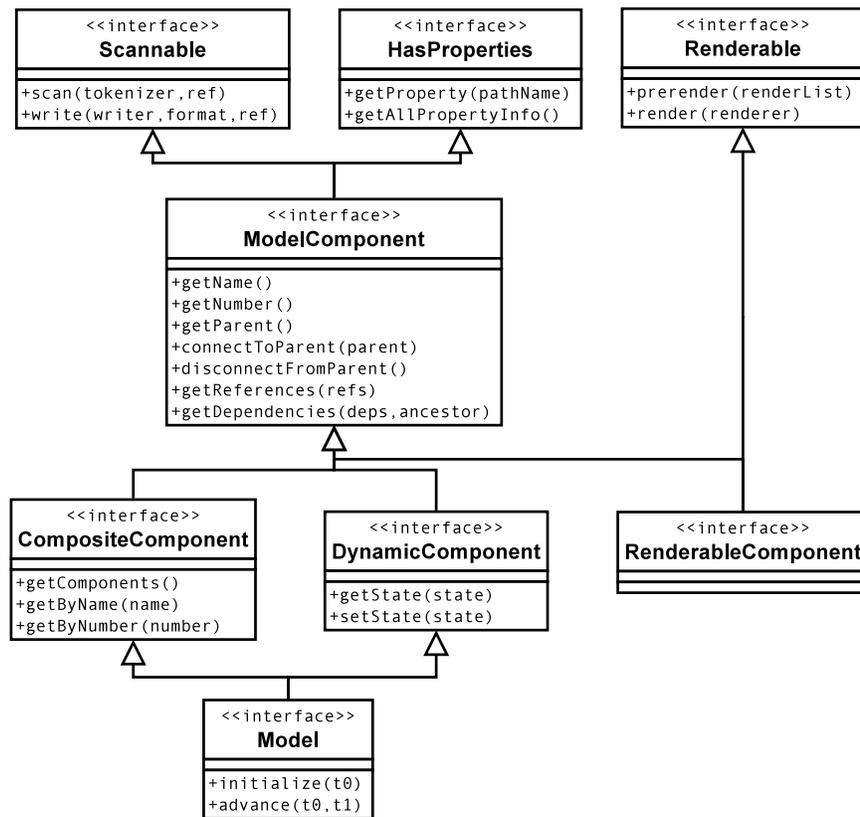


Fig. 3 Basic ArtiSynth component classes.

Sub-interfaces of `ModelComponent` include `CompositeComponent`, which contains child components, and `DynamicComponent`, which contains state information such as position and velocity.

A `Model` is a sub-interface of `CompositeComponent` and `DynamicComponent` that contains the notion of advancing through time and which implements this with the methods `initialize(t0)`, which initializes a model to time `t0`, and `advance(t0, t1)`, which advances a model from time `t0` to `t1`. 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.

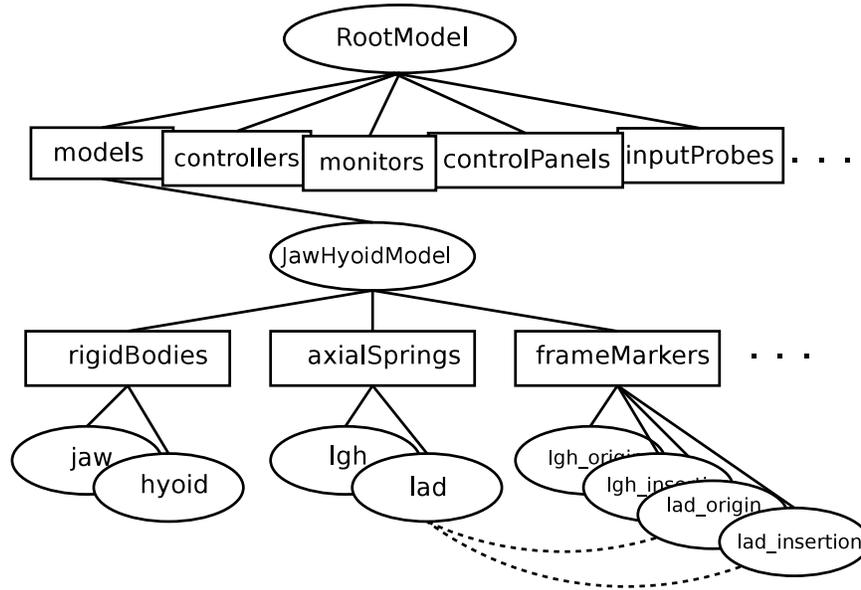
The base classes for the `ArtiSynth` components are defined in the package `artisynth.core.modelbase`.

## 2.2 The component hierarchy

Broadly speaking, the `ArtiSynth` model components include *dynamic components*, which contain dynamic state, and *force effectors* and *constraint* components which represent interactions between the dynamic components. This provides a general framework for modeling any kind of mechanical system using a Lagrangian representation. Dynamic components include particles, FEM nodes, and rigid bodies. Force effectors include point-to-point muscles (including Hill and other types) and linear and nonlinear finite elements. Constraint components include sources of both bilateral constraints (such as joints and FEM incompressibility), and unilateral constraints (such as contact and joint limits).

FEM capabilities include support for tetrahedral, hexahedral, and some higher-order elements, along with both linear and large deformation behaviors, including corotated linear [25] and hyperelastic materials. For tissue modeling, transverse isotropic and viscoelastic materials are also available. The classes for rigid and particle models are 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 controllers, monitors, 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).



**Fig. 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).

Some components are simply composite components containing lists of components of identical type (indicated by boxes in Figure 4 and including `rigidBodies`, `axialSprings`, and `frameMarkers`). These are known as `ComponentLists`, and act as simple containers for sub-components that may be added or removed as required by the modeling application. Component lists themselves are generally fixed with respect to their parents and cannot be changed or removed by the application. This is done primarily for organizational convenience, but leads to a deeper hierarchical structure in which another level is imposed between a component and what one might normally consider its “parent”. For example, rigid bodies belonging to a `MechModel` are not children of the `MechModel` itself, but of the `ComponentList rigidBodies`, which is in turn a child of the `MechModel`.

Many components contain references to other components. For example, a point-to-point muscle references its two attachment points, and a rigid body references the rigid body to which it is attached. Model components must be able to report both the components that they refer to, via the method

```
getReferences (List<ModelComponent> refs);
```

as well as the components that refer to them, via the method

```
getDependencies (List<ModelComponent> deps);
```

The method `getReferences()` should append referred components to the list `refs`, while `getDependencies()` should append referring components (i.e., de-

dependencies) to `deps`. Both `getReferences()` and `getDependencies()` are used by the structural editing software described in Section 3.6. A component's internal structures which keep track of references and dependencies should be updated in `connectToParent()` and `disconnectFromParent()` when components are connected and disconnected from the hierarchy.

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
```

A component can also be addressed by its number (returned by `getNumber()`), so that even nameless components always have a valid path name. For example, 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. A component's number is assigned when it is added to its parent, and that number persists until the component is removed, ensuring that path names remain valid as long as a component is connected to the hierarchy.

### 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 constructor 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 essentially a scripting language.

There are several reasons for creating models in code. First, it tends to be more repeatable, more precise, and (for complex models) easier than 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", "jawMesh.obj", 1000, 1);
RigidBody hyoid = RigidBody.createFromMesh (
```

```

    "hyoid", "hyoidMesh.obj", 1000, 1);

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

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

    FrameMarker lghOrigin = new FrameMarker("lgh_origin");
    FrameMarker lghInsertion = new FrameMarker("lgh_insertion");

    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);

    addModel (jawHyoid);

```

First, an instance of `MechModel`, named `JawHyoidModel`, is created. The jaw and hyoid rigid bodies are then generated using `createFromMesh()`, which is a convenience routine that creates rigid bodies given a name, a mesh file, 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 muscle `lgh` is created next, using the convenience routine `createPeck()` that assigns names and parameters. Two `FrameMarker` components are then generated to act as the origin and insertion points for the `lgh` muscle 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 muscle is added to the model using `attachAxialSpring()`, which connects it to the specified attachment points and inserts it into the `axialSpring` list, and the model itself is added to the `RootModel` using `addModel()`.

As suggested in the above example, model construction code often makes 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 read in from either Tetgen or ANSYS `.node` and `.elem` files.

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, a specific set of `RootModel` classes may be assigned "demo" names in the configuration file `.demoModels` (located in any directory specified the user's `ARTISYNTH_PATH` environment variable), using entries that look like:

```
"Spring Mesh" artisynth.models.mechdemos.SpringMeshDemo
```

```
JawLarynx artisynth.models.dynjaw.JawLarynxDemo
```

The names on the left then appear as entries in `ArtiSynth Models` menu, allowing the corresponding models to be loaded with a simple menu select. Models can also be loaded 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 [14]. 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 ]
  models=
  [ name="models"
    artisynth.core.mechmodels.MechModel
    [ name="JawHyoidModel"
      gravity:Inherited
      stabilization=Local
      collisionPointTol=0.70010976
      penetrationTol=0.00070010976
      particles=
      [ name="particles"
        pointDamping:Inherited
      ]
      rigidBodies=
      [ 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, complex 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 2.5). 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);
    Object validate (Object value, StringHolder errMsg);
    HasProperties getHost ();
    PropertyInfo getInfo ();
}
```

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

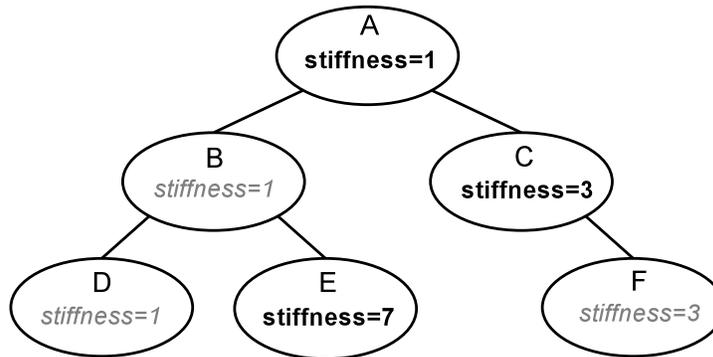
The code fragment below shows how to use the property interface to obtain the current excitation value for a `Muscle`:

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

Note, however, that properties are mostly used by generic system code; in an application, the above would more 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:



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

```

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 ":" 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 =
    mechModel.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 Probes, controllers, monitors and model advancement

As mentioned at the beginning of this section, 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 are instances of `NumericInputProbe` or `NumericOutputProbe`, where the data stream takes the form of a vector of numbers interpolated over time, and this numeric data is then mapped onto property values within selected model components.

Every probe is an instance of a `Probe` object, which implements a method

```
apply (t)
```

that is called repeatedly by the system at time `t` as simulation progresses. For a `NumericInputProbe`, `apply` will set its associated properties to the values of its data stream at time `t`. For a `NumericOutputProbe`, `apply` will collect the values of its associated properties and write them to its data stream. Applications can also define their own probes for special purposes.

In general, probes are associated with a model, and input and output probes being called, respectively, before and after the model's `advance` method. In addition, an application can define and associate with a model special purpose `Controller` and `Monitor` objects, each of which implements a method

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

that is called, respectively, before and after the model's `advance` method. Controllers are generally used to control model inputs, while monitors are used to process outputs. While most controllers and monitors are defined by the application, the inverse controller of Section 5 is implemented using a special built-in controller.

The calling sequence for model advancement is summarized as follows:

```
for (each input probe p) {
  p.apply (t1);
}
```

```

for (each controller c) {
    c.apply (t0, t1);
}
model.advance (t0, t1); // advance from time t0 to t1:
for (each monitor m) {
    m.apply (t0, t1);
}
for (each output probe p) {
    p.apply (t1);
}

```

If the model is a mechanical model (such as `MechModel`), then its `advance` method will call the physics engine described in Section 4. If the model is a `RootModel`, then its `advance` method will invoke the above advancement sequence for each of its sub-models. Probes, controllers and monitors that are not explicitly associated with a model are assumed to be associated with the `RootModel` and are invoked before and after the advance of the `RootModel`. Advancement of the `RootModel` itself is controlled by the scheduler, with the size of the advance step (i.e.,  $t_1 - t_0$ ) determined from the application as well as maximum step size information returned by the sub-models.

### 3 Interacting with Models and Simulations

ArtiSynth provides numerous ways for interacting with models and their simulations. A typical usage workflow is shown in Figure 6. More details on the user interface 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 rendering utility 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

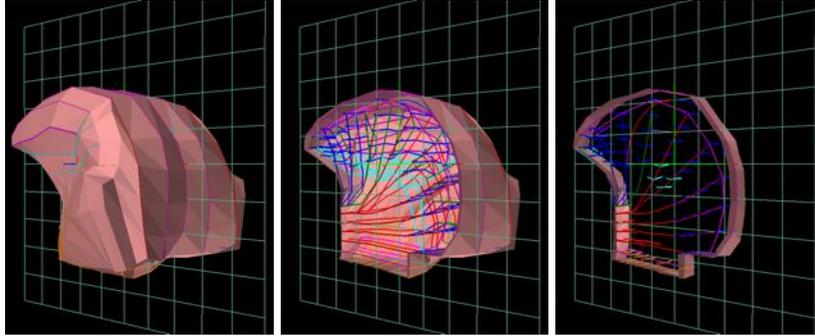
```

prerender (RenderList list);
render (GLRenderer renderer);

```

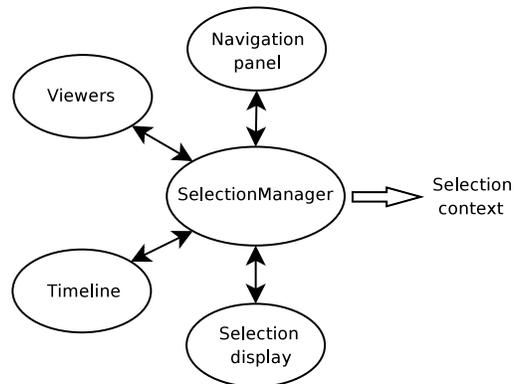
`render()` is responsible for the actual 3D rendering of the component to the GL canvas, using resources provided by the renderer (which include interfaces to GL





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

### 3.2 Navigation and selection



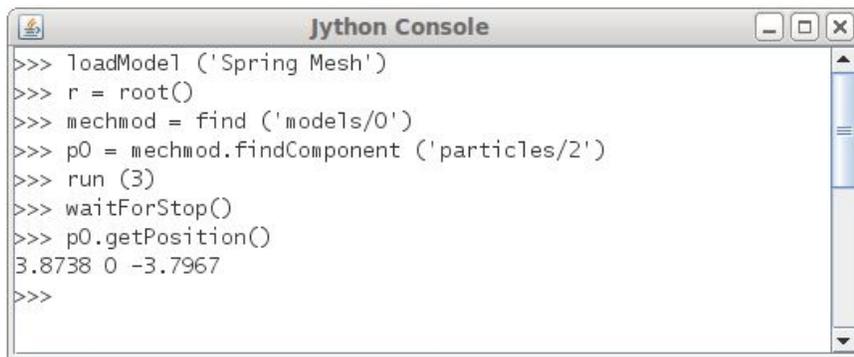
**Fig. 8** The ArtiSynth selection manager.

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.

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 Jython and MATLAB interfaces

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.

A screenshot of a window titled "Jython Console". The window contains a text area with the following text:

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

Fig. 9 Jython console with sample command sequence.

The Jython console has a number of built-in functions and variables to help load models and run a simulation. Models can be loaded using `loadModel()`. The variable `main` refers to the central ArtiSynth coordinating object, which is an instance of the class `Main` and contains references to software components such as the selection manager, viewers, and timeline. The variable `sel` is an array containing the current selection context. The function `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")
```

Using a MATLAB interface to ArtiSynth is straightforward because ArtiSynth is written in Java. Any Java object can be created in the MATLAB workspace by

calling the constructor for that class<sup>1</sup>. ArtiSynth can be launched from MATLAB by first adding ArtiSynth classes to MATLAB's classpath. The ArtiSynth Main class can then be instantiated as follows:

```
>> main = artisynth.core.driver.Main;
```

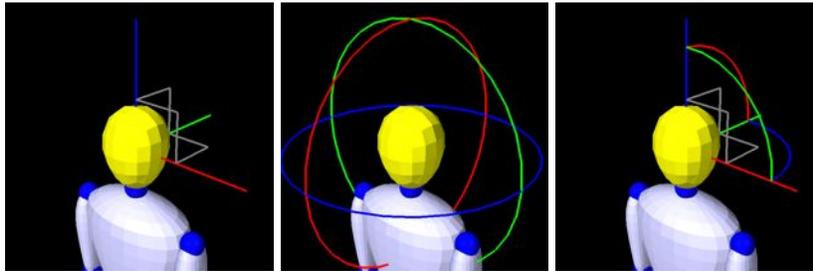
and other ArtiSynth objects can then be accessed through main:

```
>> main.loadModel('Spring Mesh');
>> rootmodel = main.getRootModel();
```

As with the Jython console, the MATLAB interface is a convenient way to script multiple simulations. It is also a powerful way to interface with user defined MATLAB scripts and functions for pre-processing input probe data and for plotting and analyzing output data from simulations.

### 3.4 Transforming geometry

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



**Fig. 10** Graphical manipulators for translation, rotation, and combined translation/rotation.

These tools can act on any component that implements the interface `TransformableGeometry`, which declares 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.

<sup>1</sup> For further details refer to: [http://www.mathworks.com/help/techdoc/matlab\\_external/](http://www.mathworks.com/help/techdoc/matlab_external/)

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

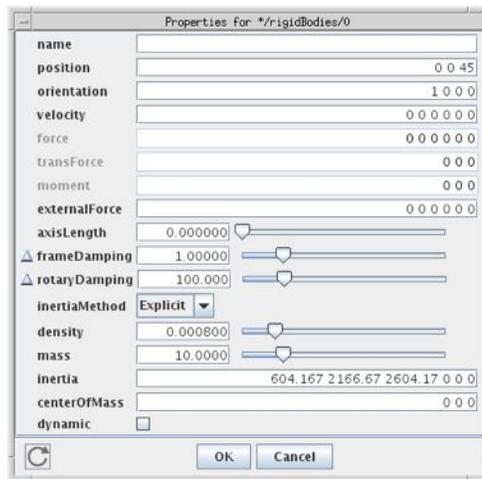


Fig. 11 Property editing panel for a rigid body.

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 (possibly 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 `artisynt.core.gui.widgets`.

### 3.6 Structural editing

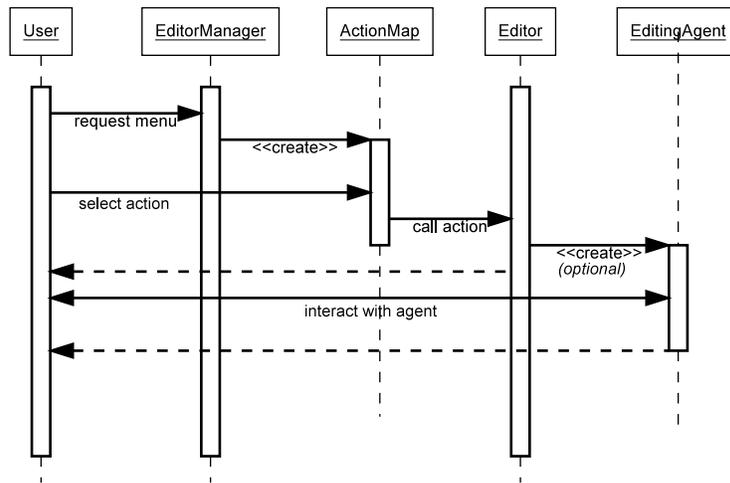


Fig. 12 Sequence of operations involving the editor manager.

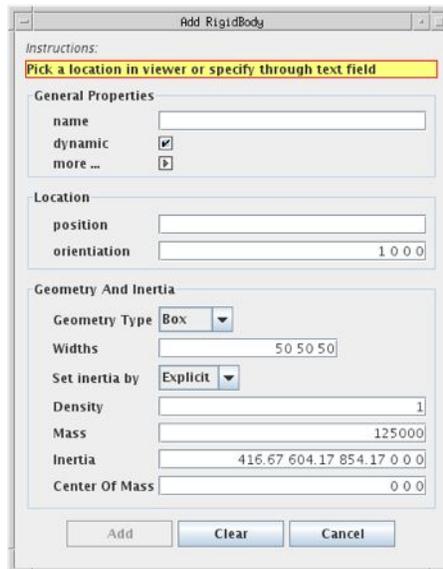
The graphical interface supports other editing capabilities, including structural edits involving the addition, deletion, and duplication of model components. These capabilities are organized around the current selection context.

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 referred to by other components. To prevent this, the system uses the components' `getDependencies()` method (Section 2.1) to expand the deletion list so that all dependent components are removed as well. 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.

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 struc-

ture. 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 of 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.



**Fig. 13** Editing agent dialog for adding rigid bodies.

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 can create a `ComponentChangeEvent` and propagate it up the hierarchy. Ancestor components will then have their `componentChanged()` method called, which can clear any cached data and propagates the change event upward.

### 3.7 The timeline

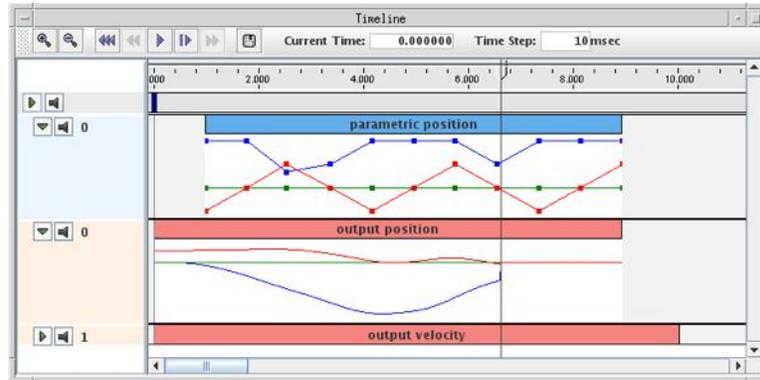


Fig. 14 Timeline with probes expanded to show data.

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 (Section 2.5) 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.

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 [37].

As mentioned in Section 2.2, ArtiSynth components 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

In this section we present the ODE associated with the mechanical system<sup>2</sup>. Let  $\mathbf{q}$  and  $\mathbf{u}$  be the generalized positions and velocities of all the dynamical components in the model hierarchy, 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 composite mass matrix. For FEM models we currently 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. By representing rigid body velocity and acceleration in body coordinates we can also ensure that  $\mathbf{M}$  is constant. Newton's second law then gives

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

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

---

<sup>2</sup> Since the ODE contains algebraic constraints, it is technically a *differential algebraic equation*, or DAE.

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

Bilateral constraints include rigid body joints, FEM incompressibility associated with the mixed u-P formulation [20], 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 [23], 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

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

subject to

$$\mathbf{G}^{k+1}\mathbf{u}^{k+1} = 0, \quad \mathbf{N}^{k+1}\mathbf{u}^{k+1} \geq 0. \quad (4)$$

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

$$\mathbf{G}^k\mathbf{u}^{k+1} = \mathbf{g}^k, \quad \mathbf{N}^k\mathbf{u}^{k+1} \geq \mathbf{n}^k, \quad (5)$$

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 mixed linear complementarity problem

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

$$0 \leq \mathbf{z} \perp \mathbf{w} \geq 0, \quad (6)$$

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 [27].

A single solve of (6) 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 result in same system as (6), only with different values for  $\hat{\mathbf{M}}^k$  and  $\hat{\mathbf{f}}^k$ . For example, for the first order semi-implicit Euler scheme, we have

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

while for the explicit forward Euler scheme we have  $\hat{\mathbf{M}}^k \equiv \mathbf{M}$  and  $\hat{\mathbf{f}}^k \equiv \mathbf{f}^k$ .

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.

### 4.3 Friction, damping, and stabilization

Coulomb (dry) friction can be added to system (6) by including extra constraints that create frictional forces along directions tangent to the contact points. A linearized friction cone [4, 27] can be created that provides an arbitrarily accurate friction approximation (depending on the number of facets in the cone), but results in a system of equations that is no longer positive-semidefinite and must be solved using techniques such as Lemke's algorithm [13] which are difficult to implement in a numerically robust way. Box friction [21] is a more approximate model that assumes the magnitudes of the contact normal forces are known a-priori (typically from the previous solve step), but adds at most two extra constraints per contact point to (6) and can be solved using relatively robust pivoting methods such as Keller's algo-

rithm [21]. Since Coulomb friction effects in our models tend to be small, ArtiSynth currently implements the numerically simpler box friction, applying it as a post-hoc correction to  $\mathbf{u}^{k+1}$  (in a manner similar to [30]), using a simplified version of (6), with  $\mathbf{M}$  instead of  $\hat{\mathbf{M}}$ .

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:

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

$$0 \leq \mathbf{z} \perp \mathbf{w} \geq 0, \quad (7)$$

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) [3], 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 [13] 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 [21]. 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

$$\begin{aligned} \tilde{\mathbf{N}} \mathbf{A}^{-1} \tilde{\mathbf{N}}^T \mathbf{z} + \tilde{\mathbf{N}} \mathbf{A}^{-1} \mathbf{b} - \mathbf{n} &= \mathbf{w} \\ 0 \leq \mathbf{z} \perp \mathbf{w} &\geq 0 \end{aligned} \quad (8)$$

where

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

which is solved using Keller's algorithm [21].  $\mathbf{u}^{k+1}$  and  $\boldsymbol{\lambda}$  can then be obtained using back-substitution:

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

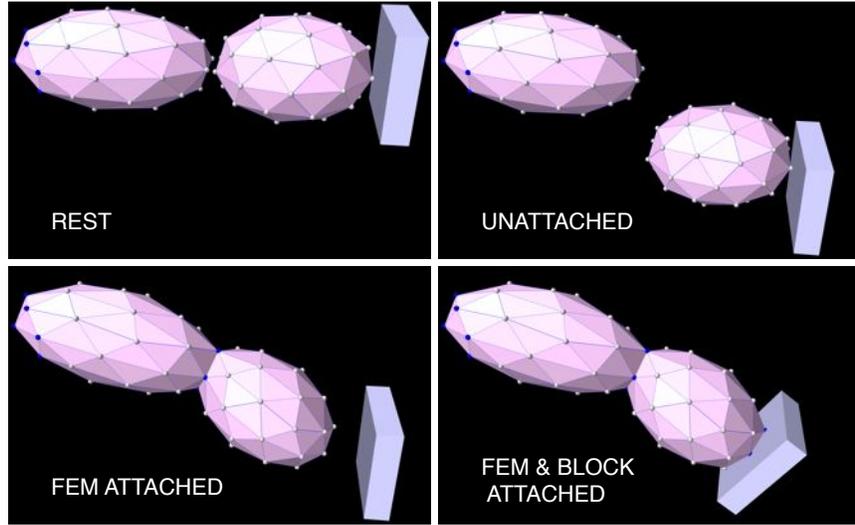
Keller's algorithm is a pivoting method with an expected complexity of  $O(m^3)$ , where  $m$  is the number of unilateral constraints. In addition, forming (8) and back-solving (9) requires  $m+1$  solves of a system involving  $\mathbf{A}$ . This is done using the Pardiso sparse direct solver [28], 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



**Fig. 15** Simulation results of bodies falling under gravity illustrating the effect of node-based dynamic attachments between bodies.

specific locations on another body. For example, particles or FEM nodes may be attached to particular spots on a rigid body, or to other nodes or elements of a different FEM model (Figure 15).

To facilitate this, ArtiSynth provides the ability to *attach* a dynamic component to one or more *master* components<sup>3</sup>. Let the set of attached components be denoted by  $\beta$ , and the remaining set of unattached *active* components be denoted by  $\alpha$ . 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

$$\mathbf{u}_j + \mathbf{G}_{j\alpha}\mathbf{u}_\alpha = 0. \quad (10)$$

$\mathbf{G}_{j\alpha}$  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 can exploit the special form of (10) 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

<sup>3</sup> The case of multiple masters arises if we connect a particle to an FEM element, in which case each of the elements' nodes acts as a master of the particle.

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  $\boldsymbol{\lambda}_{\beta}$ :

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

and hence reduce the system to

$$\begin{pmatrix} \hat{\mathbf{M}}' & \mathbf{G}'^T \\ \mathbf{G}' & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u}_{\alpha}^{k+1} \\ \boldsymbol{\lambda}_{\alpha} \end{pmatrix} = \begin{pmatrix} \mathbf{b}' \\ \mathbf{g}_{\alpha} \end{pmatrix} \quad (11)$$

where

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

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.

## 4.6 Kinematic control

It is possible to control selected dynamic components kinematically, so that their velocities are explicitly specified by an external source (such as a probe). The forces acting on kinematically controlled components 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. In particular, it provides an easy way to include experimentally recorded kinematic data in a simulation. A dynamic component can be made kinematic by setting its dynamic property to `false`.

The solution for a system containing kinematic components is arranged as follows. Let the set of active components be denoted by  $\alpha$ , and let the kinematic components be  $\rho$ . As in Section 4.5, we consider first only bilateral constraints, and partition (6) between  $\alpha$  and  $\rho$  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

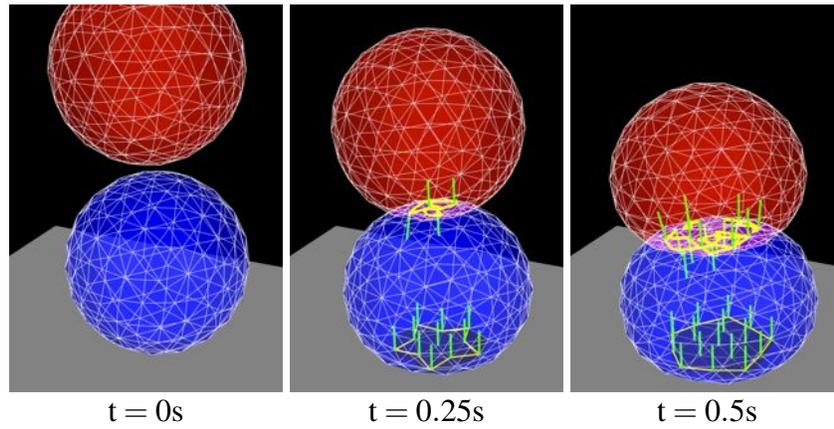
$$\begin{pmatrix} \mathbf{M}_{\alpha\alpha} & \mathbf{G}_\alpha^T \\ \mathbf{G}_\alpha & 0 \end{pmatrix} \begin{pmatrix} \mathbf{v}_\alpha \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_\alpha - \mathbf{M}_{\alpha\rho} \mathbf{v}_\rho \\ \boldsymbol{\gamma} - \mathbf{G}_\rho \mathbf{v}_\rho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

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



**Fig. 16** Time sequence of contact handling between two deformable models falling under gravity, showing the intersection contours (yellow) and the contact normals (green lines).

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 that is both closed and manifold. A bounding-box hierarchy is used to determine if any two surfaces meshes intersect. If they do, then a tracing algorithm (similar to [1]) is used to compute all the intersection contours between the two meshes as shown in Figure 16. Such contour tracing can be done relatively quickly but does require the use of robust geometry predicates similar to those in [16]; this is particularly true because collision conditions tend to drive the contacting surfaces into degenerate mesh configurations.

Determining the intersection contour allows us to create a set of constraints for correcting the interpenetration and preventing interpenetrating velocities. For rigid bodies, this is done by fitting a plane to each contour, projecting the contour onto this plane, and then sampling the vertices of the projection's 2D convex hull to create individual contact points, using the planar normal as the contact normal. For deformable bodies, contact constraints are generated for each interpenetrating node as described in [37]. The intersection contour can also provide an 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  $\lambda$  value computed in (6) is  $\geq 0$ , implying that there is no force trying to make it separate. This is effectively 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.

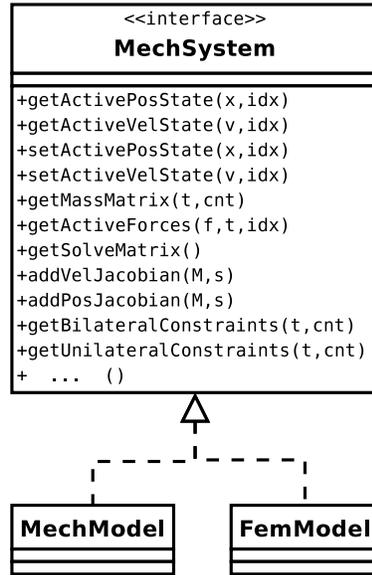
1. Compute contacts (as per Section 4.7) and the bilateral and unilateral constraint matrices  $\mathbf{G}^k$  and  $\mathbf{N}^k$ .
2. Correct positions  $\mathbf{q}^k$  to remove interpenetration and drift errors, using (7).
3. If necessary, adjust  $\mathbf{G}^k$  and  $\mathbf{N}^k$  to reflect changes 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 the physics engine to ArtiSynth

The physics engine is implemented by the class `MechSystemSolver`, which is invoked by the `advance` method of mechanical models (Section 2.5) to advance themselves forward in time.

To communicate with the solver, these models must implement the interface `MechSystem` (Figure 17), 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.



**Fig. 17** ArtiSynth mechanical models, such as `MechModel` and the basic FEM model `FemModel`, must implement `MechSystem`, shown partially here.

## 5 Inverse simulation

ArtiSynth provides inverse simulation capabilities that compute muscle activations required for a forward-dynamics model to track a prescribed kinematic task. This is a useful feature as it is often difficult to reliably measure muscle activations experimentally, or to estimate activations by hand for a particular model. In our trajectory-tracking formulation, muscle activations are determined using a quadratic program that minimizes the errors for a desired movement goal while resolving motor redundancy at each integration time step. The description here is based on material in [31], Chapter 4.

For inverse simulation, 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  $\mathbf{q}$  and  $\mathbf{u}$  are the position and velocity state vectors and  $\mathbf{a}$  is a vector of muscle activation levels (bounded between 0 and 1). We assume that  $\mathbf{f}_a$  is locally linear with respect to  $\mathbf{a}$  (which is true for a standard Hill-type muscle model), so that

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

where  $\Lambda$  is a matrix.

At present, inverse simulation in ArtiSynth is currently only supported for systems with bilateral constraints<sup>4</sup>. The velocities  $\mathbf{u}^{k+1}$  and constraint impulses  $\boldsymbol{\lambda}$  can be determined from (6), which reduces to a linear system since we are not considering unilateral constraints:

$$\begin{pmatrix} \hat{\mathbf{M}}^k & -\mathbf{G}^{kT} \\ \mathbf{G}^k & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u}^{k+1} \\ \boldsymbol{\lambda} \end{pmatrix} = \begin{pmatrix} \mathbf{M}\mathbf{u}^k + h\hat{\mathbf{f}}^k + h\Lambda^k\mathbf{a} \\ \mathbf{g}^k \end{pmatrix}. \quad (12)$$

We wish to determine  $\mathbf{a}$  at the beginning of each forward-dynamics integration time step so as to track a movement goal. The movement goal is specified by a target velocity  $\mathbf{v}_*$  in a target velocity space  $\mathbf{v}$  that is related to the system velocities  $\mathbf{u}$  via a Jacobian matrix  $\mathbf{J}_m$ , so that  $\mathbf{v} = \mathbf{J}_m\mathbf{u}$ . For time step  $k+1$ , it is easy to see from (12) 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 (12) with  $\mathbf{a}$  set to zero, and each column  $j$  of  $\mathbf{H}_u$  is the solution of  $\mathbf{u}^{k+1}$  for (12) with a right hand side of

$$\begin{pmatrix} \Lambda^k \mathbf{e}_j \\ 0 \end{pmatrix}, \quad \mathbf{e}_j \equiv \text{elementary unit vector}. \quad (13)$$

We minimize the velocity tracking error  $\|\mathbf{v}_* - \mathbf{J}_m\mathbf{u}^{k+1}\|$ , which can be expressed in quadratic form as

$$\phi_m(\mathbf{a}) \equiv \frac{1}{2} \|\bar{\mathbf{v}} - \mathbf{H}_m\mathbf{a}\|^2, \quad (14)$$

with

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

For some applications, such as computing muscle activations to generate a prescribed bite force with a dynamic jaw model (as done in [33]), we may also wish to specify a constraint force target  $\boldsymbol{\xi}$  that is related to the constraint impulses  $\boldsymbol{\lambda}$  via a Jacobian matrix,  $\mathbf{J}_c$ , so that  $h\boldsymbol{\xi} = \mathbf{J}_c\boldsymbol{\lambda}$ . We minimize the constraint force tracking error  $\|h\boldsymbol{\xi} - \mathbf{J}_c\boldsymbol{\lambda}\|$ , which can be expressed as

$$\phi_c(\mathbf{a}) \equiv \frac{1}{2} \|\bar{\boldsymbol{\lambda}} - \mathbf{H}_c\mathbf{a}\|^2, \quad (15)$$

<sup>4</sup> The addition of unilateral constraints leads to a more complex mathematical programming problem with complementarity constraints (MPCC).

with

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

where  $\boldsymbol{\lambda}_0$  and  $\mathbf{H}_\lambda$  are computed in a similar manner as described for  $\mathbf{u}_0$  and  $\mathbf{H}_u$ .

To resolve activation redundancies, we also include a weighted  $l^2$ -norm regularization term,  $\frac{1}{2} \mathbf{a}^T \mathbf{W} \mathbf{a}$ , where  $\mathbf{W}$  is a diagonal weighting matrix.

Combining the movement and constraint force goals, regularization, and muscle activations bounds, we arrive at the following quadratic program:

$$\begin{aligned} \min_{\mathbf{a}} \quad & w_m \phi_m(\mathbf{a}) + w_c \phi_c(\mathbf{a}) + \frac{w_a}{2} \mathbf{a}^T \mathbf{W}^{-1} \mathbf{a} \\ \text{subject to} \quad & 0 \leq \mathbf{a} \leq 1, \end{aligned} \tag{16}$$

where  $w_m$ ,  $w_c$ , and  $w_a$  are weights used to trade off between cost terms. This formulation can be extended to include optimizations over other biologically relevant variables, such as stiffness or metabolic energy.

The optimization program (16) is solved at the beginning of each time step, using a special built-in `Controller` (Section 2.5), 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 [21].

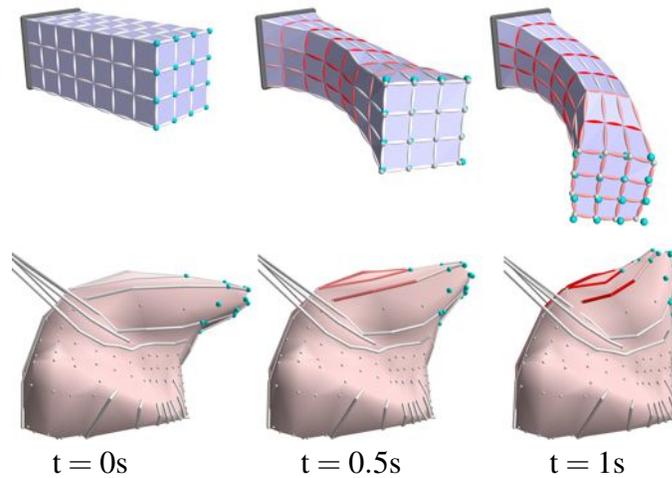
The inverse simulation tools in ArtiSynth have been used to investigate muscle driven muscular-hydrostat motions in 3D models of an idealized tentacle and a human tongue [34], as illustrated in Figure 18.

## 6 Biomechanical Models in ArtiSynth

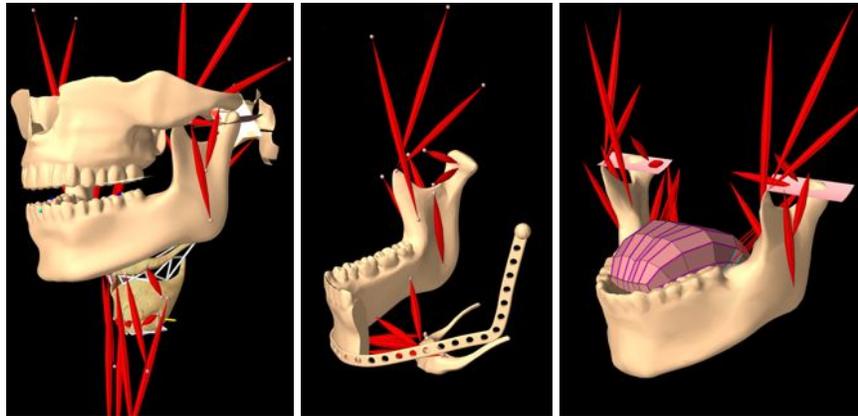
ArtiSynth has been used to produce a number of biomechanical models and associated studies, mostly focused on the oral and upper airway region.

### 6.1 Coupled Face-Jaw-Tongue-Hyoid models

One of the first models to be created in ArtiSynth was a 3D multibody jaw-hyoid model used to study dynamics and joint loading during chewing [17] (Figure 19, left). The model was also used to simulate surgical resection and reconstruction of the jaw and investigate post-operative deficits [18] and rehabilitation strategies [33] (Figure 19, center). The jaw-hyoid model was adapted with medical imaging data and combined with a 3D FEM tongue model [10] to create a dynamically coupled jaw-tongue-hyoid model representing the anatomy of a specific subject [37] (Fig-

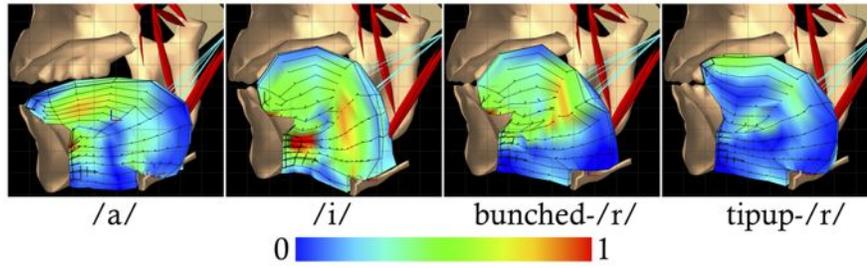


**Fig. 18** Time sequence of forward-dynamics simulation results for bending motion of a tentacle (upper panels) and elevation of the anterior tongue (lower panels) created using the trajectory-tracking controller. Cyan spheres indicate the target motion of nodes at the model's tip. Output muscle fiber activation levels are indicated by line color (white = 0%, red == 100%).



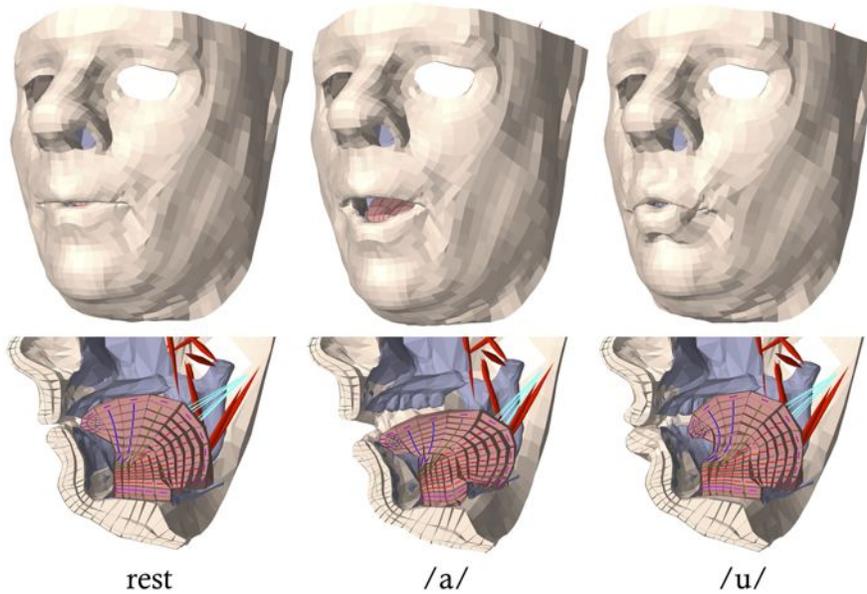
**Fig. 19** Craniomandibular models in ArtiSynth: multibody jaw-hyoid model (left), reconstructed hemimandibulectomy model (center), coupled jaw and FEM tongue model (right).

ure 19, right). This model was used to simulate stress and strain within the tongue during speech postures (Figure 20), in order to investigate the biomechanical basis for categorical speech articulation variability in the context of English /r/ postures [32].



**Fig. 20** Strain plots of tongue deformation during simulation of speech postures with the coupled jaw-tongue-hyoid model.

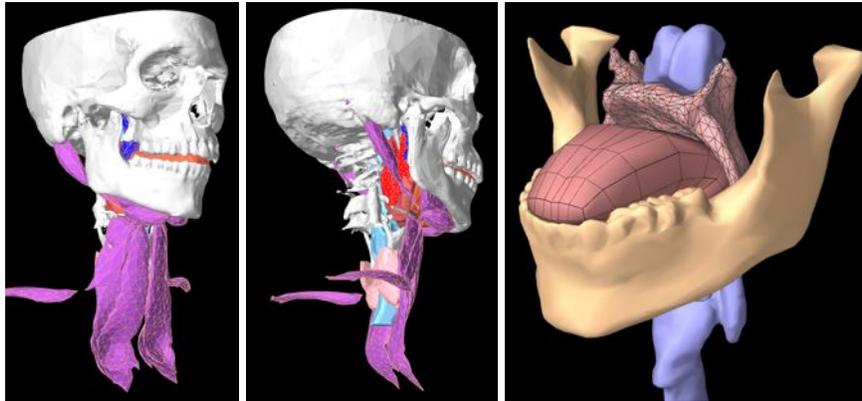
An extension of the jaw-tongue model to include the face and lips is currently in progress [36] and is being used to create forward-dynamics simulations of speech articulation [35] (Figure 21, see also [www.artisynth.org/orofacial/](http://www.artisynth.org/orofacial/)).



**Fig. 21** Oblique (upper panels) and sagittal cut-away (lower panels) views of dynamic simulation of face-jaw-tongue movements for speech vowel postures.

## 6.2 *Comprehensive upper airway model*

Efforts are continuing to create a comprehensive, integrated model of upper airway anatomy, including the soft palate, pharynx, and larynx, for a variety of medical and research purposes. Preliminary versions of these models are depicted in Figure 22 and further details can be found at [www.artisynth.org/opal](http://www.artisynth.org/opal).



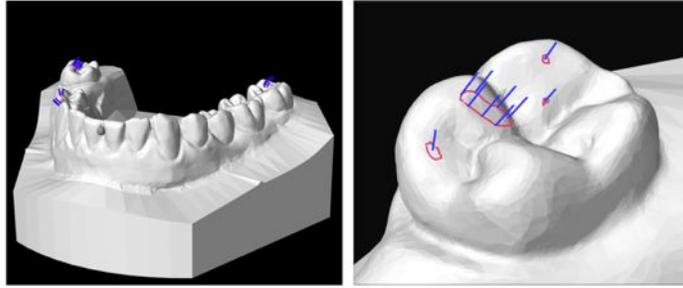
**Fig. 22** Comprehensive upper-airway models under development with ArtiSynth: front and back views of a hyolaryngeal model with FEM models of the extrinsic laryngeal muscles (left, center), an FEM model of the soft-palate coupled to the tongue and skull (right).

## 6.3 *Dental contact models*

Our original multibody jaw-hyoid model used simple planar tooth contact [17]. We are working on more detailed models of tooth contact (Figure 23) using the mesh-based contact handling in ArtiSynth, as discussed in Section 4.7.

## 6.4 *Whole-body and limb musculoskeletal models*

While our modeling efforts to date have primarily targeted head and neck anatomy, where the need for tight coupling of rigid and deformable tissues is readily apparent, the simulation techniques in ArtiSynth are generally applicable to a wide range of biomechanical systems. Figure 24 shows a simple lower limb musculoskeletal model from the OpenSim platform [15] that has been loaded in ArtiSynth. One of our primary research directions is to use the ArtiSynth toolkit to expand the state-



**Fig. 23** Contact detection and handling in ArtiSynth for dynamic simulation of tooth contact with digital dental casts (left); close up view of tooth contact (right). Magenta lines show the contours of contact regions and blue lines show reaction forces.

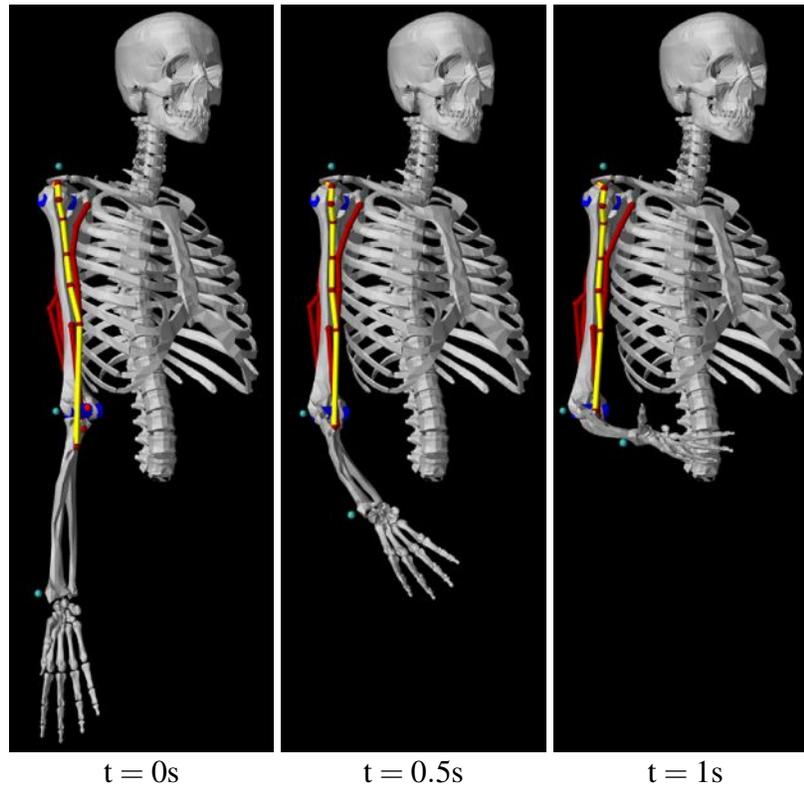
of-the-art in whole-body musculoskeletal models with rigid (or small deformation) skeletal structures coupled with large deformation FEM muscle and tendon models.

## 7 Future Directions and Conclusion

Development of ArtiSynth is ongoing in several areas. We continue to work on the physical simulation, with a focus on finding faster solution methods for (6); one possible approach involves using a hybrid direct-iterative solver. We also intend to expand the set of available components to include shell elements, new materials, and Lagrangian fluid simulation based on smoothed particle hydrodynamics. Further work is also planned for the inverse simulation, extending it to include more general targets such as desired stiffnesses and FEM surfaces.

ArtiSynth provides the biomedical research community with a highly interactive platform for creating models of anatomical structures that combine multibody and FEM models, together with contact and constraints. The open source framework allows for the easy introduction of novel custom components and the integration of cutting edge simulation algorithms. The system's effectiveness has been demonstrated for a variety of applications centered on the functional anatomy of the oral region and upper airway. Expected future applications include the development of general musculoskeletal models and a comprehensive model of swallowing behavior.

The system is freely available for research purposes from [www.artisynth.org](http://www.artisynth.org), and new collaborations with other parties are welcome.



**Fig. 24** Arm26 model from OpenSim [15] implemented in ArtiSynth, showing flexion simulation with activation of biceps brachii long head (denoted in yellow).

## References

1. Michael J. Aftosmis, Marsha J. Berger, and John E. Melton. Robust and efficient cartesian mesh generation for component-based geometry. *AIAA Journal*, 36(6):952–960, 1998.
2. Jérémie Allard, Stéphane Cotin, Francois Faure, Pierre-Jean Bensoussan, Francois Poyer, Christian Duriez, Hervé Delingette, and Laurent Grisoni. Sofa: an open source framework for medical simulation. In *Medicine Meets Virtual Reality (MMVR)*, pages 13–18, February 2007.
3. 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.
4. 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.
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. Ted Belytschko, Wing Kam Liu, and Brian Moran. *Nonlinear Finite Elements for Continua and Structures*. Wiley, 2000.
8. Javier Bonet and Richard D Wood. *Nonlinear Continuum Mechanics for Finite Element Analysis*. Cambridge University Press, 2008.
9. Stephanie Buchaillard, Muriel Brix, Pascal Perrier, and Yohan Payan. Simulations of the consequences of tongue surgery on tongue mobility: Implications for speech production in post-surgery conditions. *International Journal of Medical Robotics and Computer Assisted Surgery*, 3(3):252, 2007.
10. 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.
11. Murat Cenk Cavusoglu, Tolga Goktekin, and Frank Tendick. Gipsi: A framework for open source/open architecture software development for organ-level surgical simulation. *IEEE Transactions on Information Technology in Biomedicine*, 10(2):312–322, 2006.
12. Matthieu Chabanas, Vincent Luboz, and Yohan Payan. Patient specific finite element model of the face soft tissues for computer-assisted maxillofacial surgery. *Medical Image Analysis*, 7(2):131–151, 2003.
13. Richard W. Cottle, Jong-Shi Pang, and Richard E. Stone. *The Linear Complementarity Problem*. Academic Press, 1992.
14. Douglas Crockford. The application/json media type for javascript object notation (json). 2006.
15. Scott Delp, Frank Anderson, Allison Arnold, Peter Loan, Ayman Habib, Chand John, Eran Guendelman, and Darryl Thelen. OpenSim: Open-source software to create and analyze dynamic simulations of movement. *IEEE Transactions on Biomedical Engineering*, 54(11):1940–1950, 2007.
16. 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.
17. 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.
18. 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.
19. Yaqi Huang, David White, and Atul Malhotra. Use of computational modeling to predict responses to upper airway surgery in obstructive sleep apnea. *Laryngoscope*, 117:648–653, 2007.
20. Thomas J.R. Hughes. *The finite element method: linear static and dynamic finite element analysis*. Dover Publications, New York, 2000.
21. 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.
22. Donna S. Lundy, Christine Smith, Laura Colangelo, Paula A. Sullivan, Jerilyn A. Logemann, Cathy L. Lazarus, Lisa A. Newman, Tom Murry, Lori Lombard, and Joy Gaziano. Aspiration: cause and implications. *Otolaryngology-Head and Neck Surgery*, 120(4):474–478, 1999.
23. Christoph Lunk and Bernd Simeon. Solving constrained mechanical systems by the family of newmark and  $\alpha$ -methods. *Journal of Applied Mathematics and Mechanics (ZAMM)*, 86(10):772–784, 2006.
24. Kevin Montgomery, Cynthia Bruyns, Joel Brown, Stephen Sorkin, Frederic Mazzella, Guillaume Thonier, Arnaud Tellier, Benjamin Lerman, and Anil Menon. Spring: A general framework for collaborative, real-time surgical simulation. In *Medicine Meets Virtual Reality (MMVR)*, pages 23–26, 2002.
25. Matthias Müller and Markus Gross. Interactive virtual materials. In *GI '04: Proceedings of Graphics Interface*, pages 239–246, 2004.
26. Musculoskeletal Research Laboratories. FEBio: finite elements for biomechanics. <http://mrl.sci.utah.edu/software/febio>.

27. 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.
28. Olaf Schenk and Klaus Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Gener. Comput. Syst.*, 20(3):475–487, 2004.
29. Ahmed A. Shabana. *Dynamics of Multibody Systems*. Cambridge University Press, 1998.
30. 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.
31. 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.
32. Ian Stavness, Bryan Gick, Donald Derrick, and Sidney Fels. Biomechanical modeling of English /r/ variants. In *International Seminar of Speech Production*, 2011.
33. Ian Stavness, Alan Hannam, John E. Lloyd, and Sidney Fels. Predicting muscle patterns for hemimandibulectomy models. *Computer Methods in Biomechanics & Biomedical Engineering*, 13(4):483–491, 2010.
34. Ian Stavness, John Lloyd, and Sidney Fels. Inverse-dynamics simulation of muscular-hydrostat finite-element models. In *23rd International Society of Biomechanics Congress (ISB)*, number 933, July 2011.
35. Ian Stavness, John Lloyd, Yohan Payan, and Sidney Fels. Towards speech articulation simulation with a dynamic coupled face-jaw-tongue model. In *International Seminar of Speech Production*, 2011.
36. 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.
37. 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*, 27:367–390, 2011.