Model Editing System

The implementation of model editing through the Artisynth GUI

Tracy Wilkinson
CPSC 399
September 4, 2007
UBC Electrical and Computer Engineering

September 4, 2007

Constance Wun
Room 170 Chemistry/Physics Building
6170 University Blvd.
Vancouver, BC

Dear Ms. Wun,

Enclosed is my work term report entitled Model Editing System: The implementation of model editing through the Artisynth GUI. This report was prepared for John Lloyd, an associate professor and my supervisor in the Artisynth Lab at the University of British Columbia.

This report documents the overall implementation of the Model Editing System in Artisynth and gives some specific examples of editing actions. I have worked on the model editing system in the Artisynth lab during the summer of 2007.

The model editing system was implemented so that editing could occur within the Artisynth viewer after a model was loaded into the viewer. Model editing involves changing a model or its components in some way, or adding new components. Before the model editing system models and components could only be changed in code, and not from within the GUI.

The report is intended to be used as a reference guide on the model editing system so that future developers may refer to it for guidance when implementing new model editing actions.

Yours truly,

Tracy Wilkinson

Model Editing System

The implementation of model editing through the Artisynth GUI

Prepared for:
John Lloyd
Artisynth Lab, UBC
Vancouver, Canada

Prepared by:
Tracy Wilkinson
23303035
September 4, 2007
Artisynth Lab, UBC
Vancouver, Canada

Summary

The Model Editing system is a set of classes within Artisynth designed to allow generic editing of model components from within the Artisynth viewer. Each component that can be edited has an Editor class associated with it. Each editing action that can be performed on a component has an EditWidget class, which is responsible for carrying out the editing and display of GUIs for editing. The editors are all registered with the EditorManager, thus the appropriate editor for a model component can be obtained by querying the EditorManager. If an editing action is to be undoable it must be implemented as a Command and registered with the UndoManager. This set of classes for the model editing system allows for any component to be made editable by creating an editor and an edit widget for that component. This report describes the requirements of model editing within the Artisynth viewer, how the overall system was implemented, and some specific examples of model editing.

Table of Contents

List of Figures

Introduction

The model editing system was implemented in the Artisynth project to facilitate the editing of models through the Artisynth viewer, rather than through code. Prior to the editing system all model editing had to be done in code before a model was loaded into the Artisynth viewer. With the editing system models are still loaded into the viewer, but then they can be modified while being displayed. This provides very useful actions that were not possible before such as adding marker points and editing muscle fibres.

The overall system consists of a set of editing classes. The classes were developed to mimic the model component hierarchy already in place in Artisynth. Thus each component that has editing actions associated with it also has an editor that is responsible for those actions. The editors are the central point of all editing actions that can occur for any given component, and the editor manager knows about each editor and which component it edits. Each editor is responsible for a set of edit widget classes. The edit widgets classes are where the editing actions are carried out.

The purpose of this report is to describe the model editing system, how it can be used, and how further development on model editing should occur within the existing framework. The report first describes the requirements of the model editing system. Then the overall structure of the editing system is explained in detail, topics discussed include the editor manager, editors, widgets and the undo system. Finally some examples of the editing system that have been implemented are described. These include the editing of rigid bodies, muscle fibres, muscle bundles and muscle tissue.

Model Editing Requirements

Before the Model Editing System, models could be viewed within the Artisynth viewer by writing them in code and then loading them in. If you wanted to change a model in any way the code for that model would have to be changed and then the model reloaded. This causes different challenges for certain models and components. Initially the concept of model editing was developed so that marker points could be added to rigid bodies. This developed into a more generic model editing system such that any model or model component could be edited in whatever ways were needed. The idea was that different needs for model editing would arise and there should be a generic system in place to meet those needs.

The model editing system needs to be generic enough that any new requirements for model editing can easily be plugged in. This means that the model editing cannot be dependent on the model or the component that is being edited. For example if a component that was not previously editable has some need for editing, a class can easily be created to carry out that editing. Or if a new way of changing a previously editable component arises that editing capability can easily be created.

One way of addressing this issue was to create a set of model editors that mimic the component hierarchy that already exists in Artisynth. For each model that is editable there would exist an Editor, which is responsible for carrying out the editing actions of that component. In this way the model component class itself is not responsible for the editing actions, instead its editor is.

Discussion

Overall System

The editing system core consists of three major parts, which are shown in Figure 1. These are the editor manager, the editors, and the edit widgets. In addition to these pieces there are the editor utilities and undo system. The editor manager is the central access point for editing model components. The editor manager knows which components can be added to others, so it can be queried to see if one component can be added to another. It also stores a HashMap with model components as keys and editors as values, thus it can be queried to get an editor for a model component. This way if a class wants to perform an editing action on a model component, but it doesn't know which component it is, it can query the editor manager for the correct editor. An editor exists for each component that is editable. An edit widget exists for each specific editing action an editor might carry out. For example an editor exists for rigid bodies, and an edit widget exists for adding marker points to rigid bodies, and one could imagine a different edit widget existing for modifying rigid bodies in some other way. The editor utilities are procedures that editors and edit widgets may need to carry out. The undo system allows for undoing editing actions on model components, as long as they are logged correctly.

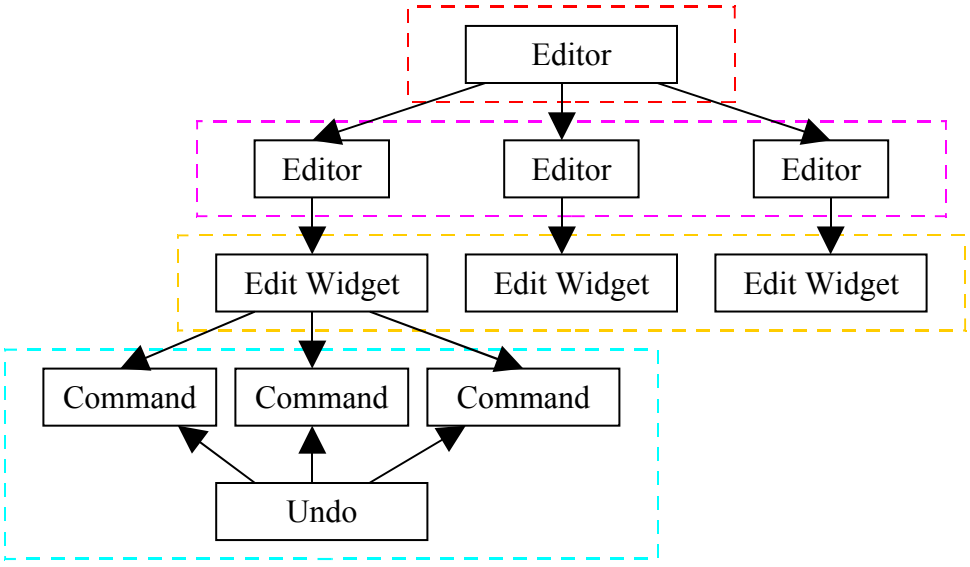


Figure 1 – Editing System Class Structure: The class structure of the editing system. The dashed boxes indicate different parts of the editing system.

Editors

While the editor manager is responsible for determining the correct editor to use for a specific component, editors are responsible for determining which editing actions can occur for a component. Each component that is editable has an editor, for example rigid bodies use the RigidBodyEditor. There are three main actions that editors perform. The first is they provide a list of editing actions (myEditingActions) that can be performed on a component, which is a list of JMenuItems. The second is they can be queried to get the correct edit widget for an editing action through the createEditWidget function. The third is that each editor has a list of model components that can be added to the component it is responsible for editing, which are myAddableComponents. For example the RigidBodyEditor uses this list to show that markers can be added to Rigid Bodies. The following is a set of steps to create a new editor for a model component that does not already have one:

1. Create the editor, which extends BaseEditor, in the EditorManager package.
2. In the constructor add the JMenuItems that specify the different editing actions that can occur on the model, and that the editor can support, examples are add, edit and delete.
3. Add the JMenuItems to the list of myEditingActions and have the editor implement ActionListener to listen for each of the JMenuItems
4. In the constructor add any components that can be added to the one this editor is associated with to myAddableComponents.
5. Register the new editor in the editorsMap in EditorManager.
6. Write the createEditWidget() function to create and return the correct edit widget for whatever the editing action is.

The JMenuItems that specify the editing actions can then be accessed to create a menu of possible editing actions for a model component. This is done when the SelectionPopup is displayed. The JMenuItem editing actions are added to the popup when an editable component is selected in the Artisynth viewer. Then when they are selected the editor creates the appropriate edit widget.

Edit Widgets

Edit widgets are responsible for carrying out the editing actions such as add, edit and delete on model components. An edit widget knows which component it is editing, and about any parent components that may need to be modified as well. For example the MuscleBundleEditWidget knows about the bundle it is editing and about the parent tissue to the bundle. When components are added or deleted the navigation panel must also be updated. This is done by calling notifyStructureChange() with a reference to the components parent. To create an edit widget carry out the following steps:

1. Create the edit widget, which extends BaseEditWidget, in the editorManager package.
2. Add the member variables, which are specific to the model component being edited, that the edit widget needs to know about to perform editing actions.
3. Implement the initDisplay() function to create the widgetDisplay if there is a GUI for performing editing actions in this widget.
4. Implement the functions for editing actions done by the edit widget. Examples include add, edit and delete. If the actions are to have the possibility of being undone then they need to be implemented within the constraints of the undo system. This means that each action needs to be implemented as a command that is carried out and then registered with the undo manager.

Undo System

The undo system is implemented using the Command Design Pattern. Each editing action that can be undone exists as a class that implements the Command interface, which has two functions, execute and undo. Each class implementing the Command interface contains enough extra information about its command that it can undone. For example adding a fibre to a muscle bundle is an editing action that exists as the AddFibreCommand. This command is created specifying a fibre and the bundle to add it to, then it can be executed. After execution it can be registered with the undo manager. Then when commands registered with the undo manager are undone, the AddFibreCommand undo function is called, which removes the fibre from the bundle. The AddFibreCommand class knows about the fibre and the bundle so it can successfully

undo the executed action. The undo system is implemented so that the undo manager does not care what type of command it is undoing, it simply calls the undo function and the command is responsible for actually carrying out the undo. This way the undo system is generic enough that any model editing action can be undone. For a model editing action to be undoable it needs to be created as a class that implements the Command interface. Then that class is created by the edit widget, executed, and registered with the undo manager. The code for creating and undoing an editing action follows these steps:

1.In the edit widget the AddCommand addCmd is created, then addCmd.execute() is called.

2.The addCmd is registered with the undo manager by UndoManager.addCommand(addCmd).

3.UndoManager.undoLastCommand() is called, and so addCmd.undo() is executed.

The undo manager maintains a two dimensional ArrayList of commands to be undone. This way sets of commands that should be undone together can be added to the undo manager. Then when undoLastCommand is called the entire set of commands is undone. One instance where this is useful is the cut and paste fibre action, where both a delete and an add command need to be undone together.

Model Editing Examples

Rigid Bodies

Rigid bodies can be edited by having marker points added to them. This is done by putting the Artisynth viewer into marker adding mode by clicking the add marker button as in shown in Figure 2, or by right clicking and selecting the add marker option. While the viewer is in marker adding mode if a rigid body is clicked on the RigidBodyEditor creates a RigidBodyEditWidget with the parent rigid body, the child marker and the mouse ray. Then the intersection of the ray and the rigid body is calculated and the marker added to the rigid body at the intersection point as is shown in Figure 3.

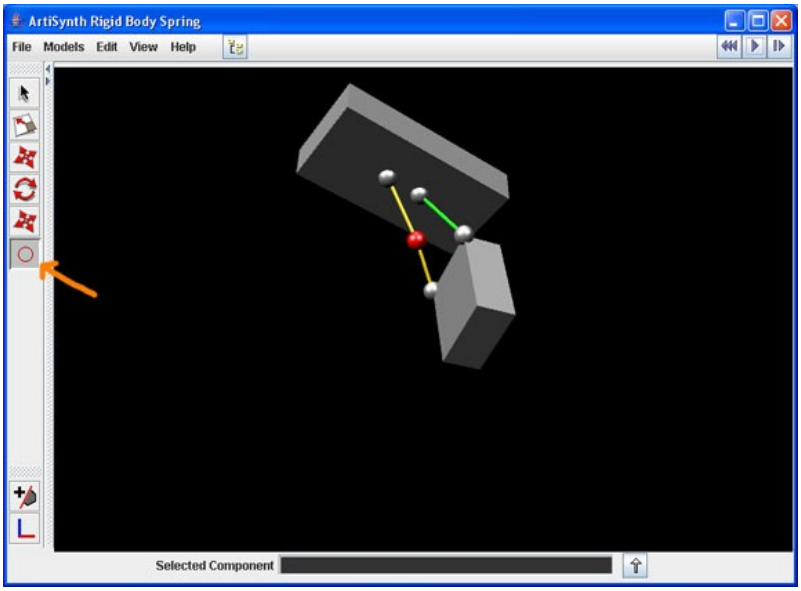


Figure 2 – Marker Adding Mode: Selecting the add marker button puts the Artisynth viewer into add marker mode so that markers can be added to rigid bodies.

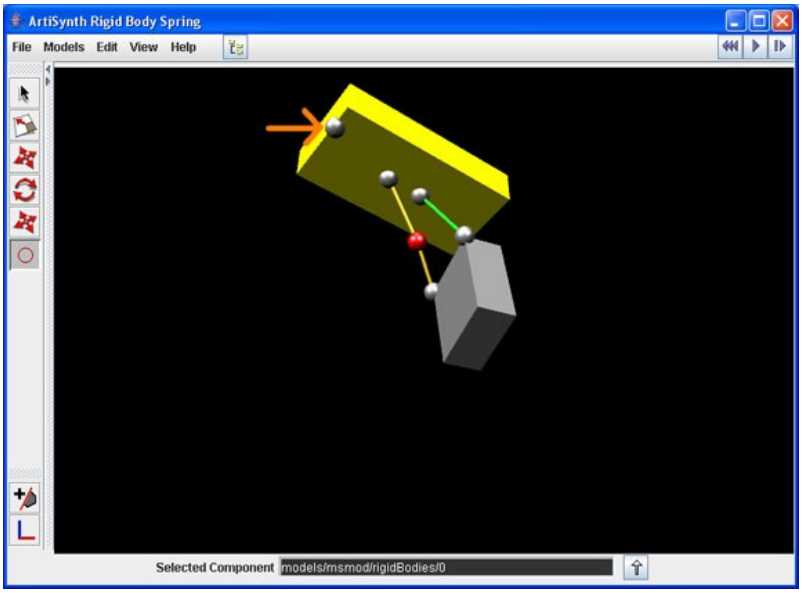


Figure 3 – Adding Markers: A new marker has been added to a rigid body by clicking a point on a rigid body.

Muscle Tissue

Muscle tissue can be edited by having muscle bundles added and deleted from a tissue. When the "Add New Bundle" item is selected the muscle tissue editor creates a muscle tissue edit widget that displays the window for creating a new bundle, which we see in Figure 4. To create a new bundle either nodes or fibres in the current muscle tissue must be selected, in this example we have selected the nodes at the bottom of the tissue to create a zig zag muscle bundle. Then clicking 'Done' causes the edit widget to create a new muscle bundle, which we can see in a fuschia colour in Figure 5. Muscle bundles can also be deleted from a tissue by selecting the "Delete Bundle" menu item.

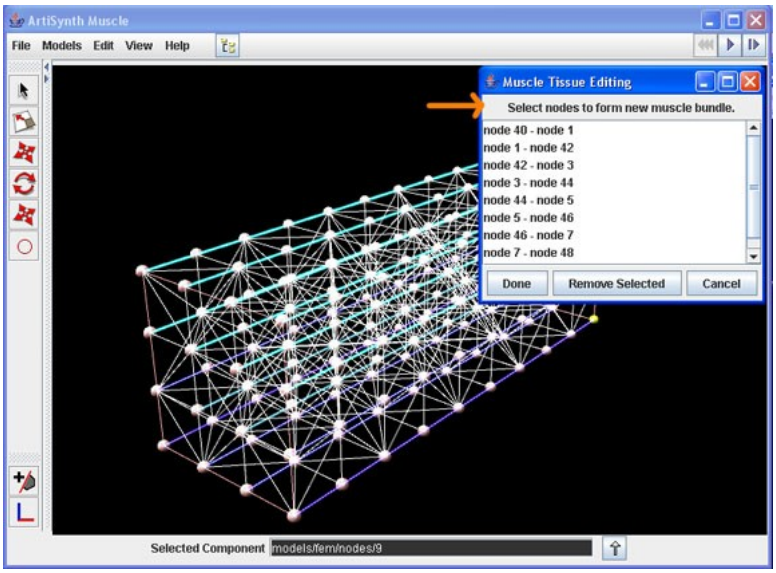


Figure 4 – Adding Muscle Bundles: The edit widget display shows the list of node pairs that have been selected to add fibres between. These fibres will be added to a new muscle bundle when 'Done' is clicked.

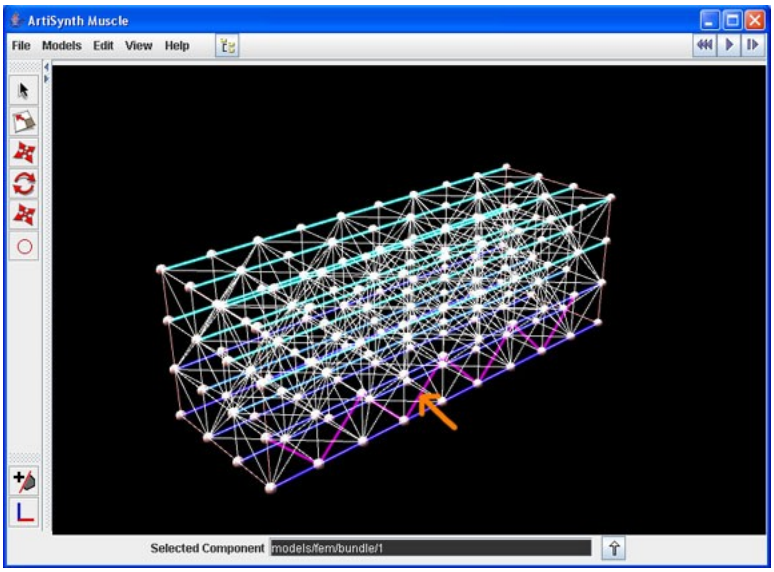


Figure 5 – A New Muscle Bundle: The new muscle bundle, which is a fuschia colour, is created and added to the muscle tissue.

Muscle Bundles

Muscle bundles can be edited by having fibres added or deleted from them. In Figure 6 and Figure 7 we see that selecting the "Add Fibre" item causes the FibreEditor to display an edit widget. From the edit widget nodes or fibres in the muscle tissue the bundle belongs to can be selected, and when they are selected the edit widget adds new fibres to the bundle. To edit fibres select the "Edit Fibres" option, which displays an edit widget with a list of all the fibres in the bundle so they can be selected for deletion as in Figure 8. In Figure 9 we can see that the fibres selected in Figure 8 have been deleted from the bundle. The "Delete" option deletes the bundle from the tissue, and "Undo" undoes the previous editing action carried out.

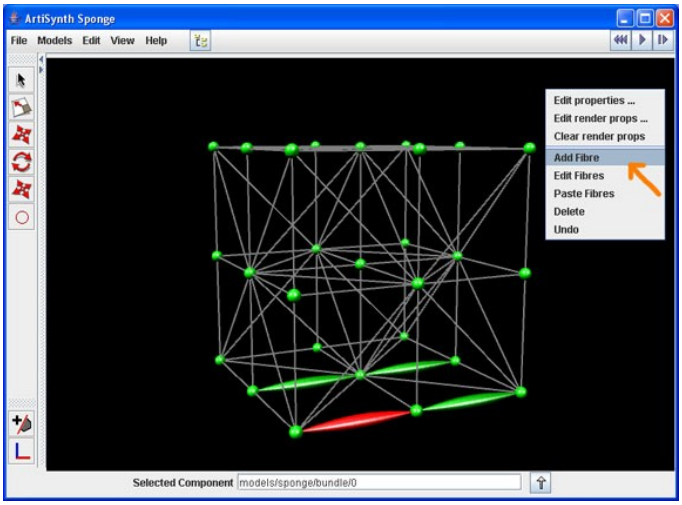


Figure 6 – Adding Fibres: The 'Add Fibre' menu item is selected to display a window from which fibres can be selected to be added to the bundle.

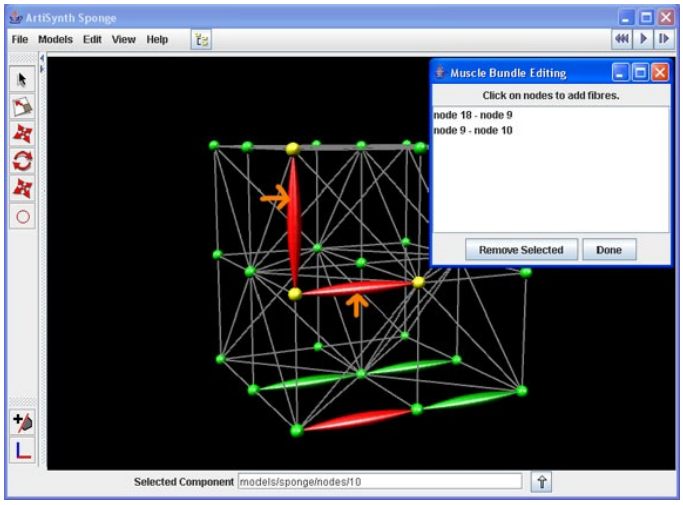


Figure 7 – New Fibres: Two pairs of nodes are selected and fibres are created between those nodes and added to the selected muscle bundle.

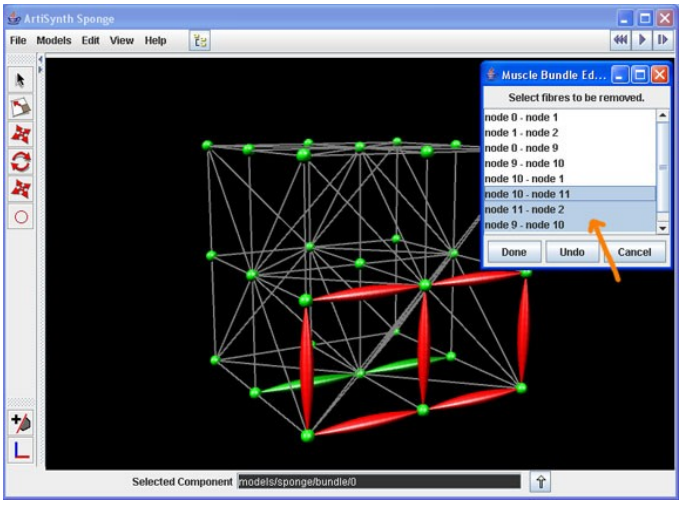


Figure 8 – Deleting Fibres: Three pairs of nodes that specify fibres are selected in the edit fibres widget to be removed from the muscle bundle.

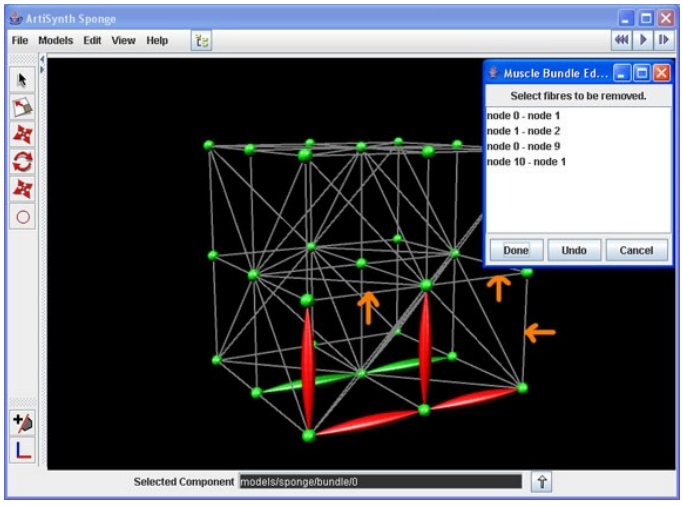
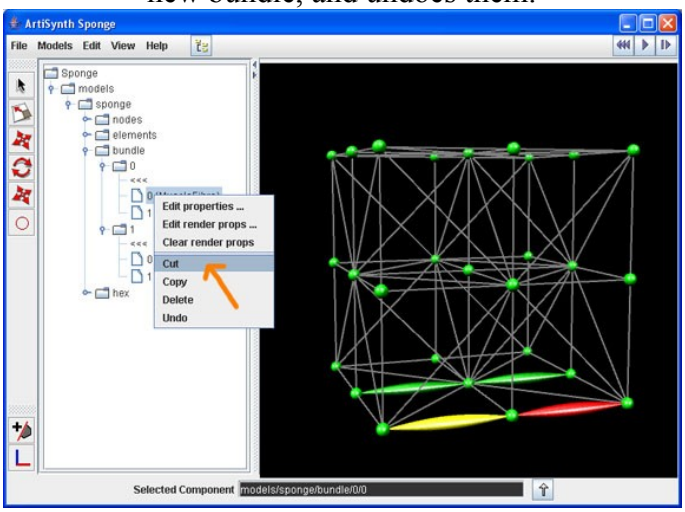
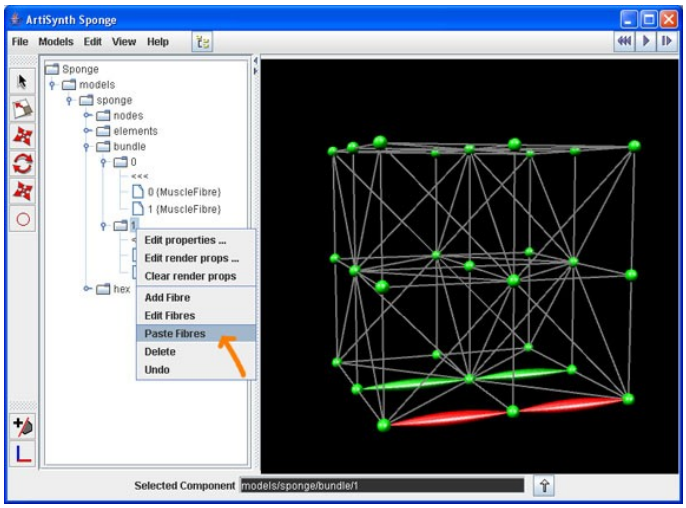


Figure 9 – Fibres Removed: The specified fibres have been removed from the muscle bundle.

Muscle Fibres

Muscle fibres can be edited by moving fibres between bundles. This is done through the cut and copy items we see in Figure 10. We can select a fibre, or a set of fibres to cut or copy from a bundle. Doing this makes a fibre editor that creates a fibre edit widget, which adds the selected fibres to its list of fibres being edited. However, no editing actions are actually carried out until the "Paste Fibres" item is selected as is shown in Figure 11. This causes the selected fibres to be deleted from their current bundle and inserted into the selected new bundle. We can see in Figure 12 that the selected fibre has now been moved to the green bundle from the red bundle. It is possible to undo this action by selecting the "Undo" item, which moves the specified fibre back to the red bundle. This can be done because adding and deleting fibres are implemented as Commands, and are saved to the UndoManager. So when "Undo" is selected the UndoManger gets the last set of commands, which was to delete and add the fibre to a new bundle, and undoes them.





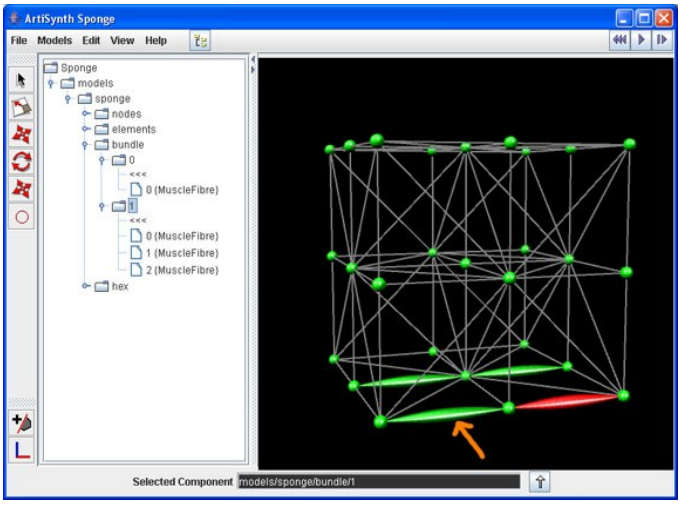


Figure 10 – Cutting Fibres: (top left)
The 'Cut' menu item is selected and the current fibre is added to the list of fibres being edited in the FibreEditWidget.

Figure 11 – Pasting Fibres: (top right)
The 'Paste Fibres' menu item is selected and the fibres that are to be edited in the FibreEditWidget are removed from their current bundle (red) and added to their new bundle (green).

Figure 12 – Finished Cut and Paste: (bottom left)
A fibre has been moved from the red bundle to the green bundle.

Future Recommendations

One major issue in model editing is how to delete components from models. The delete functionality that has been implemented thus far in edit widgets has been very specific to the component that was being deleted. Having a generic ability to delete components would be useful and would have to take into account the issue of cascading deletes. Cascading deletes are an issue because some model components are attached to other components in ways that are dependent and one should not exist without the other. One example is springs and markers. If a marker attached to a spring is deleted the spring cannot be left unattached with nothing at one end. In this case a dialog should appear listing all the dependencies on the component that is being deleted, warning that those components will also be removed if the delete is completed. Then the user can choose whether or not to complete the delete.