# RobotBuilder

# User's Guide

Version 1.0

Steven J. Rodenbaugh
David E. Orin
Department of Electrical Engineering
The Ohio State University


May 30, 2003

## Revision History

| | |
|---|---|
| May 30, 2003 | Updated for Version 1.0 (Lucas Frankart) |

## Background

RobotBuilder and RobotModeler were originally developed as part of Steven Rodenbaugh's M.S. thesis work. Defining and saving a user view and other camera control functions were implemented by Luke Frankart as a part of his Bachelor's Honors thesis. The applications are offered for free to encourage research in the field of robotics. While considerable effort has been dedicated to verifying the accuracy of all calculations, the results are not guaranteed to be valid. For more information about the applications, please consult the User's Guide or the RobotBuilder website at http://eewww.eng.ohio-state.edu/~orin/RobotBuilder/RobotBuilder.html.

# TABLE OF CONTENTS

# 1. Introduction

RobotBuilder is an application that enables the researcher to quickly develop accurate simulations of complex robotic articulations with advanced control implementations. It has an intuitive interface to facilitate efficiency, but also has many options to meet researchers' needs.

RobotBuilder's underlying simulation library is *DynaMechs*, developed by Scott McMillan [1]. The library efficiently simulates the dynamics of robotic articulations, and has the capability to simulate articulations with closed-kinematic loops (developed by Duane Marhefka) [2]. More information about *DynaMechs* can be obtained from http://dynamechs.sourceforge.net/.

A screenshot of RobotBuilder can be seen in Figure 1. At the top of the main window, there is a menu bar and a toolbar. Below that there are two windows which can be resized by the vertical splitter bar found between them. The window on the left is the Link Tree View and shows the hierarchical relationships between the links. The properties of the links can be viewed and edited from context menus created by that window. The window on the right shows a 3D representation of the articulation. The 3D capabilities are created by the *WorldToolKit* graphics library [3].

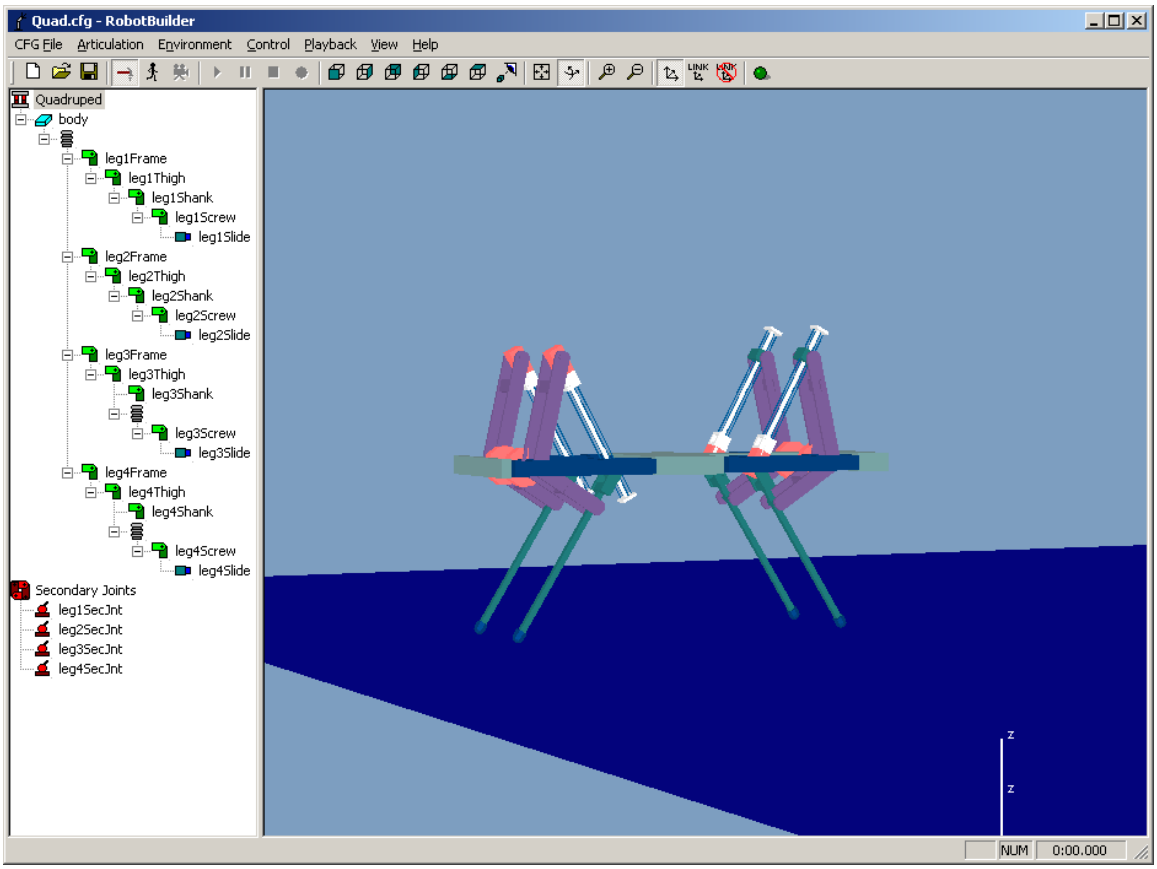



Figure 1: RobotBuilder application screenshot.

RobotBuilder operates in three main modes. The `Build` mode is used to assemble the robotic articulation and define the properties of the links. The `Simulate` mode simulates the robot, and displays a graphical animation of the results in 3D. The `Playback` mode is used to replay previously created simulation results in approximately real time.

## 2. Install

There is no installation program with RobotBuilder. To install, merely unzip the archive to a desired directory.

*Note: A shortcut can be created if desired. Verify that the "Start In" directory is correctly specified as the directory where the RobotBuilder application is located.*

## 3. Data Files

RobotBuilder uses several data files to persistently save components used in simulation projects. The data files' structure was borrowed from *DynaMechs*. This provides several advantages:

1. The data files developed for the latest version *DynaMechs* can be used in RobotBuilder.
2. Data files developed in RobotBuilder can be used in *DynaMechs*
3. The modular design makes it convenient to copy components used in other simulation projects to new projects.

### 3.1 Files

The data files used in RobotBuilder define the various components used in the simulation project. The components, and their file extensions are summarized below:

| *File Type* | *Extension* | *Description* |
|---|---|---|
| Configuration File | `.cfg` | General simulation properties |
| Articulation File | `.dm` | Defines the link-joint pairs that constitute the articulation and their properties |
| Environment File | `.env` | Defines the characteristics of the environment |
| Terrain File | `.dat` | Describes the terrain (when the environment specifies a terrain) |
| Treadmill File | `.dat` | Defines treadmill properties (when the environment specifies a treadmill) |
| Control DLL | `.dll` | Control implementation |

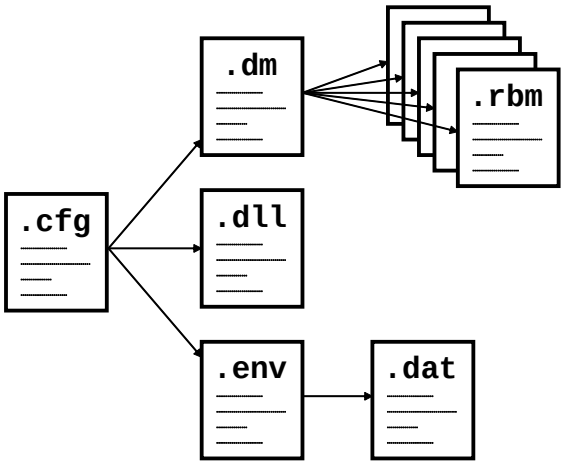
In addition, the graphical representation for a link is stored in a file. RobotBuilder supports many graphics file types:

| *Graphics File Type* | *Extension* | *Description* |
|---|---|---|
| RobotModeler Model | `.rbm` | RobotModeler was specifically developed to enable the researcher to quickly create graphical models for RobotBuilder. Please see the RobotModeler User's Guide |
| *Autodesk* | `.dxf` | |
| *Wavefront* | `.obj` | |
| *Autodesk 3D Studio* | `.3ds` | *3D Studio Max* generates this file type as well |
| *Pro/Engineer* | `.slp` | |
| *MultiGen* | `.flt` | 3D OpenFlight format |
| *VideoScape* | `.geo` | |
| *WorldToolKit* | `.nff` | Neutral File Format |
| *WorldToolKit* | `.bnf` | Binary Neutral File Format |
| VRML 1.0 | `.wrl` | |
| *XAnimate* Model | `.xan` | XAnimate is a graphics library developed at OSU for visualizing robotic articulations |
| *GLAnimate* Model | `.dat` | GLAnimate is a modified implementation of XAnimate that utilizes OpenGL |
| *XAnimate* Combo Model | `.cmb` | A list of XAnimate models that collectively define the graphics model |

## 3.2 File Relationships

The relationships between components are defined in the data files. Data files that possess dependent relationships with other data files, define them in fields inside the file. When opening files, RobotBuilder uses these relationship descriptions to open the correct dependent files, and pass the data onto *DynaMechs* for simulation. As an example, the Articulation File (`.dm`) defines the links in the robotic articulation. Most links have a graphical representation. The graphical model filename for the model used by the link is listed in a field in the `.dm` file.

The relationships between data files that make up a simulation project are shown below:




The file relationships are clear from the diagram. The Configuration File (`.cfg`) defines the project's Articulation File (`.dm`), Environment File (`.env`), and Control DLL (`.dll`). In addition, the environment is further described by the Terrain File (`.dat`) that the Environment File references. The graphical representation of the articulation is

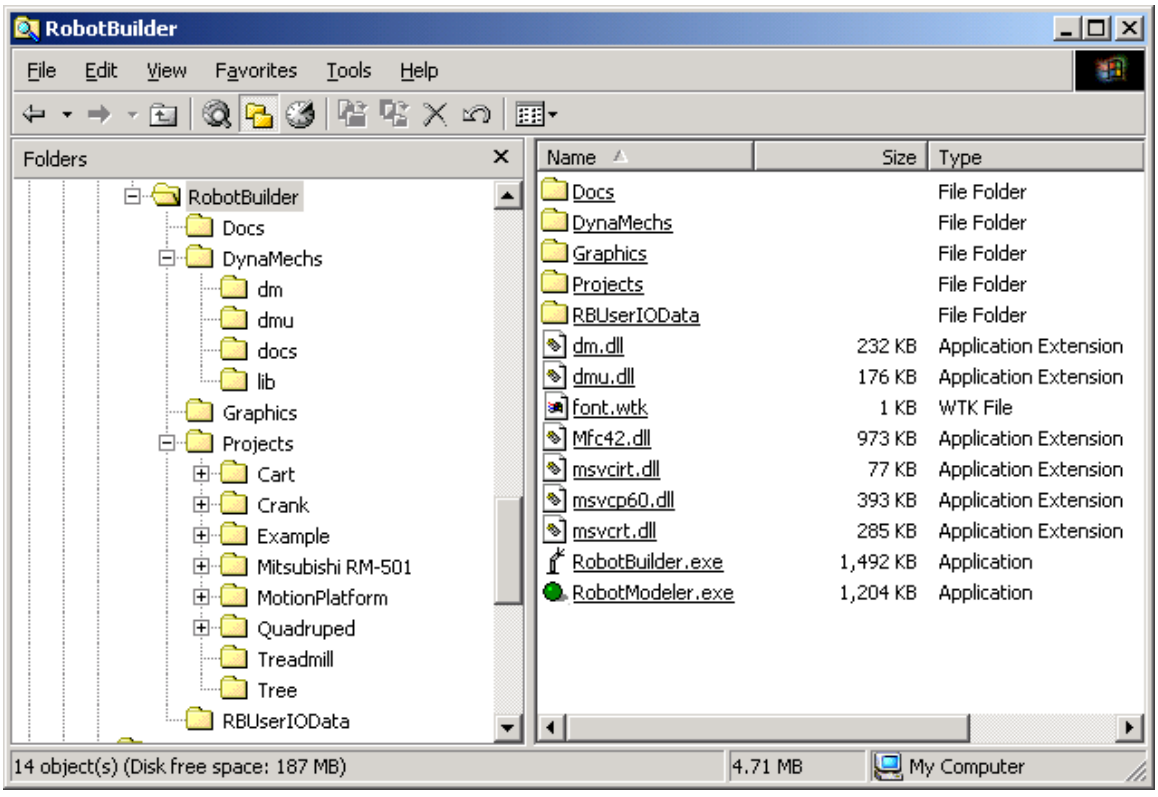
defined by the graphical model for each link. Consequently, the Articulation File (`.dm`) points to a number of graphics files to be used in the 3D View.

*Note: The graphics files can be any supported graphical model format in the Graphics File table above.*

*Note: The `.rbm` file can also point to another file that specifies a texture to apply to the model.*

## 3.3 Directory Structure

The components of RobotBuilder are organized in a directory structure to facilitate usability and productivity. The directory structure will look similar to:



The root directory named `RobotBuilder` can be placed anywhere desired. The files in the RobotBuilder directory are:

- `dm.dll` – *DynaMechs* library
- `dmu.dll` – *DynaMechs* utility library
- `font.wtk` – Sets the font for the axis identifiers in RobotBuilder and RobotModeler.
- `Mfc42.dll` – *Microsoft Foundation Classes* (*MFC*) DLL. RobotBuilder and RobotModeler were developed with *MFC*.
- `msvcirt.dll` – *MS Visual C++* DLL
- `msvcp60.dll` – *MS Visual C++* DLL
- `msvcrt.dll` – *MS Visual C++* DLL
- `RobotBuilder.exe` – RobotBuilder application
- `RobotModeler.exe` – RobotModeler application

*Note: The MFC and MS Visual C++ DLLs may be deleted if you have current versions properly installed on your system.*

The directories used in RobotBuilder are:

- `Docs` – Documentation for RobotBuilder and RobotModeler.
- `DynaMechs` – *DynaMechs* support files
  - `dm` – Definitions for *DynaMechs* classes
  - `dmu` – Definitions for *DynaMechs* utility classes
  - `docs` – *DynaMechs* documentation

    - `lib` – *DynaMechs* import libraries
- `Graphics` – A repository for graphics files.
- `Projects` – Contains folders for each simulation project
    - `Quadruped` – Bounding quadruped project developed by Duane Marhefka [2].
- `RBUserIOData` – Definitions and import library for Control GUI.

It is not required, but in addition to the `Graphics` repository, RobotBuilder will also support repository directories named `Articulation`, `Control`, and `Environment` to hold Articulation Files, Control DLLs, and Environment Files, respectively. The user can also create a subdirectory structure under a project directory to organize different implementations of the same component. For example, if the user wanted to simulate contact with a stiff ground, and then later with a soft ground, the Environment Files to describe both circumstances could be in a subdirectory named `Environment` in the particular project directory. `Articulation`, `Control`, and `Graphics` subdirectories are also supported under any project directory.

## 3.4 File Search Pattern

When RobotBuilder loads a file, it attempts to load the dependent files. For example, when an Articulation File (`.dm`) is opened, each link's graphical model file is opened so the 3D model can be drawn.

The file relationships can be saved in two ways: the full path to the file or just the filename. The first method has the advantage that the file can always be located, but the disadvantage is that the project is not portable. If the file was loaded on another machine with a different configuration, the full path may no longer work. Consequently, using just the filename is encouraged.

When a file needs to be found by just its filename, RobotBuilder performs a sequence of searches in an attempt to find it. At any point during the search sequence, if the file is found, the search process will stop, and RobotBuilder will load the file. If the file is not found by the end of the search process, RobotBuilder will not be able to load the file, and will display an error message indicating the file could not be found. The search process for a file is as follows:

1. Simulation project directory (this is the directory where the `.cfg` file for the simulation exists). If no Configuration File has been specified, this step will be skipped.
2. Subdirectory under the project directory. If no Configuration File has been specified, this step will be skipped.
    a. `Articulation` subdirectory if the file is an Articulation File.
    b. `Graphics` subdirectory if the file is a graphics model, Terrain File, Treadmill File, or texture file.
    c. `Control` subdirectory if the file is a Control DLL.
    d. `Environment` subdirectory if the file is an Environment File.
3. `RobotBuilder` directory

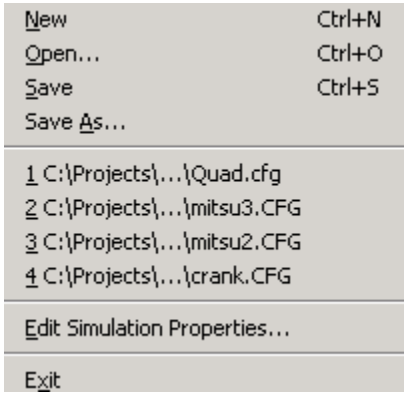
4. Repository directories under the `RobotBuilder` directory
    a. `Articulation` repository directory if the file is an Articulation File.
    b. `Graphics` repository directory if the file is a graphics model, Terrain File, Treadmill File, or texture file.
    c. `Control` repository directory if the file is a Control DLL.
    d. `Environment` repository directory if the file is an Environment File.

*Note: RobotBuilder loads the first file found in the search sequence by the given filename. This implies that if another file exists by the same filename, and can be found earlier in the search sequence than the file desired, the expected file will not be loaded.*

## 4. Menu

The menu bar provides access to many functions offered by RobotBuilder. The menu bar items are:


### 4.1 CFG File Menu




The "CFG File" menu provides access to file related functions for the configuration. The configuration specifies general simulation settings like the integrator to use, the background color, and the Articulation File to use.

**New**

"New" will clear the current configuration and reset all data to default values. This implies the articulation and environment will also be cleared.

**Open...**

"Open..." will open the Open dialog which will let the user open an already existing Configuration File created with RobotBuilder. By default, the file filter will be set to show only `.cfg` files, but it can be changed to show all files.

**Save**

Saves the configuration to a file. If a filename has not been assigned, a Save file dialog will be automatically opened to enable the user to specify the file name and path.

*Note: Although the user is prompted to save changes to the configuration file upon selecting "New" or "Exit", all other data files are NOT monitored for changes. Also, there is no auto save function.*

*Note: When saving the configuration, it does NOT save changes to the articulation or the environment. If it is desired to save changes in those files, select "Save" from the respective menus.*

**Save As...**

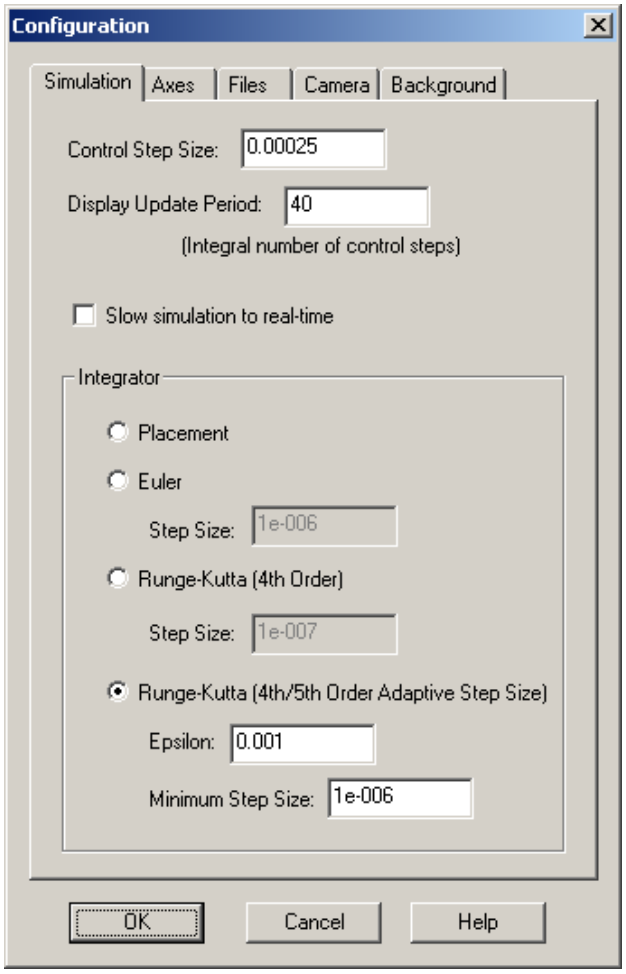
Opens the Save dialog to enable the user to save the configuration to file.

**Recently Used File List**

After the "Save As..." option, there will be up to four items which list recently used files. Clicking on one of these will open the file if it exists.

**Edit Simulation Properties...**

Opens the `Simulation Properties` dialog box. This dialog enables the user to set general simulation properties. The dialog is divided into several tabs of similar properties. All of the data specified in this dialog is saved in the Configuration File (`.cfg`).



- "Control Step Size" – During simulation, RobotBuilder periodically calls user supplied control code. This specifies how frequently it is called (in seconds).

  *Note: RobotBuilder actually calls the control between integration steps, so the actual control period may be slightly greater than the specified control period.*

- "Display Update Period" – The number of control steps to perform before updating the 3D graphics. This number should be set as a compromise between computational efficiency and 3D animation quality.
- "Slow simulation to real-time" – In some simulations, particularly when using kinematic control, the simulation time can progress faster than real-time. Check this setting to slow the simulation so that simulation time progresses at about the same rate as real time.

  *Note: If the simulation time is lagging real time, this setting will have no effect.*

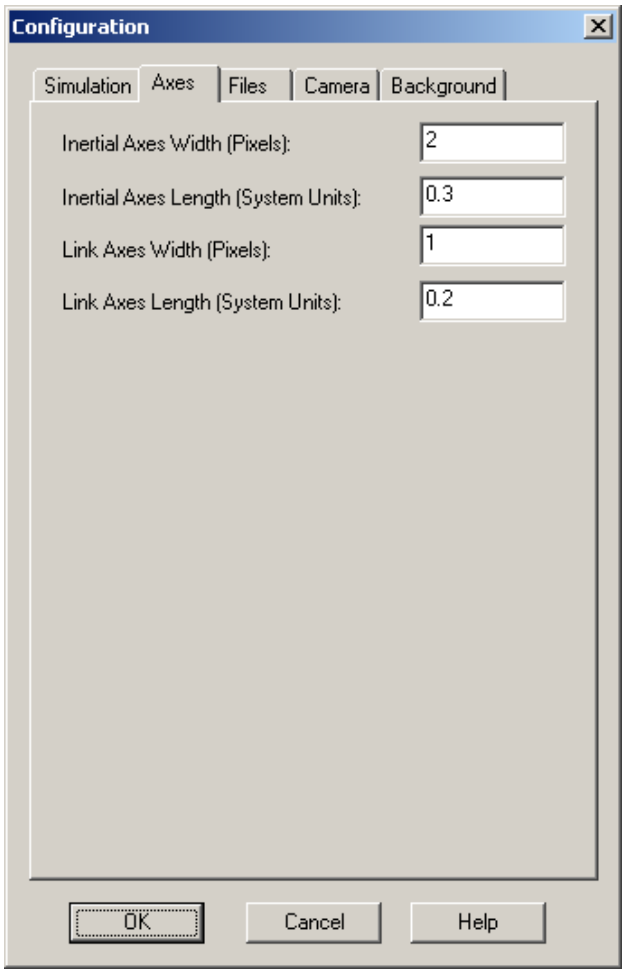
- "Integrator" – Select the integrator type used to advance the system states during simulation. In general, this is trade-off between accuracy and speed, but the best selection depends on many factors.
  - "Placement" – This integrator doesn't actually perform any integration. Instead, the control DLL specifies the desired position of each link.
  - "Euler" – Simple Euler integration of the states. A step size must be specified.

- "Runge-Kutta ($4^{th}$ Order)" – Generally more accurate and a more stable integrator than Euler. A step size must be specified.

  *Note: For more information about Runge-Kutta integration see* *http://www.library.cornell.edu/nr/bookcpdf/c16-1.pdf*.

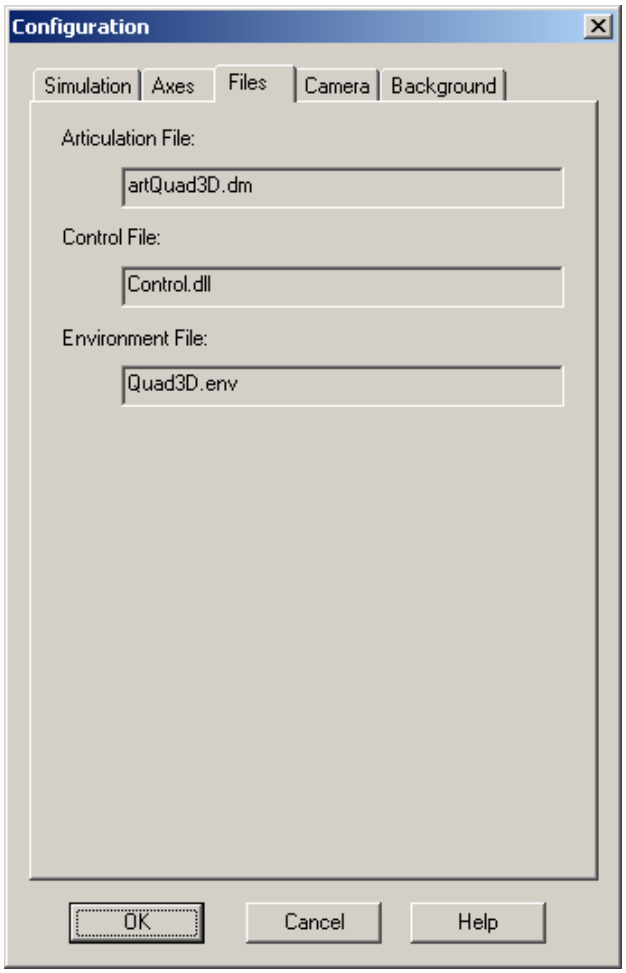
- "Runge-Kutta ($4^{th}$/$5^{th}$ Order Adaptive Step Size)" – Integration method which has a varying step size that remains larger than or equal to a specifiable minimum step size. This integrator ensures that the magnitude of the difference between the fourth-order and fifth-order calculations of the state variables is less than a specifiable epsilon times the magnitude of the current state, by automatically adjusting the step size.

  *Note: For more information about Adaptive Step Size Runge-Kutta integration see* *http://www.library.cornell.edu/nr/bookcpdf/c16-2.pdf*.



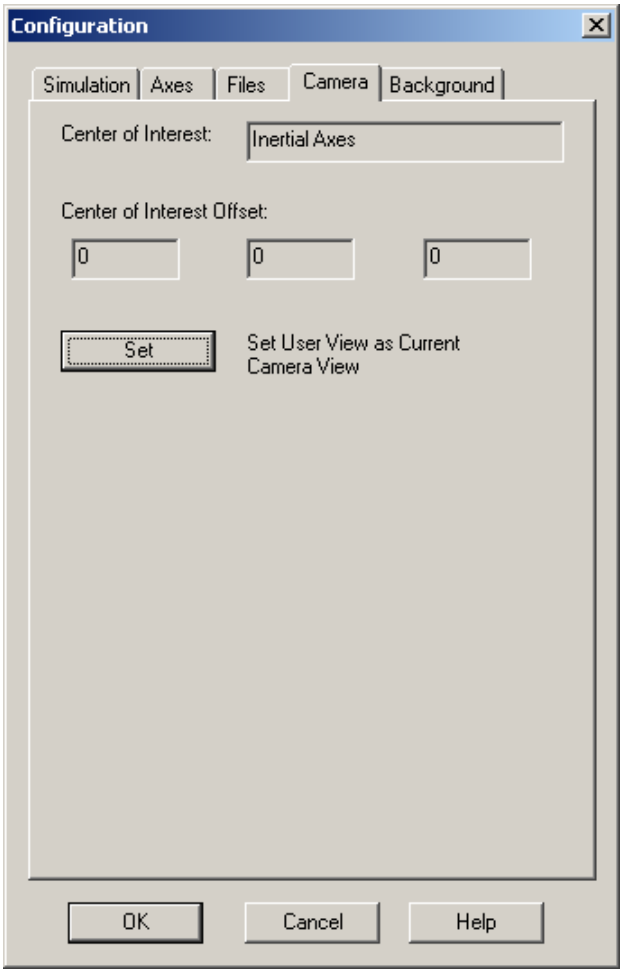

- "Inertial Axes Width (Pixels)" – The number of pixels wide to draw the inertial axes.
- "Inertial Axes Length (System Units)" – The number of units long to draw the inertial axes.
- "Link Axes Width (Pixels)" – The number of pixels wide to draw the axes showing each link's coordinate frame.
- "Link Axes Length (System Units)" – The number of units long to draw the axes showing each link's coordinate frame.

*Note: See Section 9.1 for more information about units.*




- "Articulation File" – This is a read-only field that displays the Articulation File associated with the current configuration. To change this, open or save a new one from the "Articulation" menu. This field will be empty if no Articulation File has been specified yet.
- "Control File" - This is a read-only field that displays the Control DLL associated with the current configuration. To change this, open a different one from the "Control" menu. This field will be empty if no Control DLL has been specified yet.

- "Environment File" - This is a read-only field that displays the Environment File associated with the current configuration. To change this, open or save a new one from the "Environment" menu. This field will be empty if no Environment File has been specified yet.
- "Center of Interest" – This is a read-only field that displays the name of the center of interest link associated with the current user-defined camera view. If the inertial axes are specified (or a user-defined camera view has not been set), "Inertial Axes" will appear in this box.



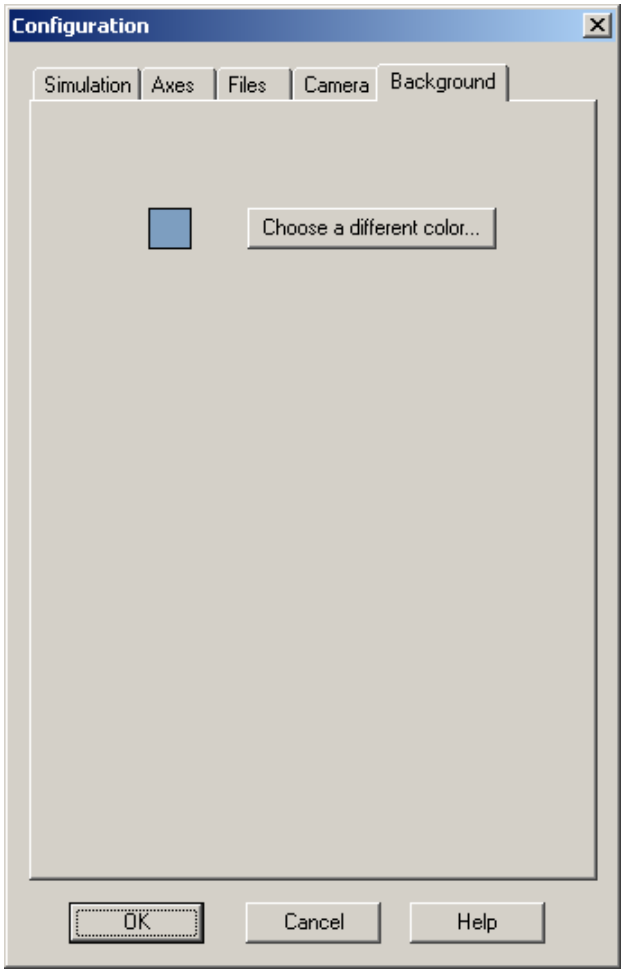

  *Note: For more information about camera control and selecting a center of interest link, please see Section 4.6 and Section 6.3.1, respectively.*
- "Center of Interest Offset" – This is a read-only field that displays the center of interest offset associated with the current user-defined camera view. If a user-defined camera view has not been specified, these boxes will contain zeros.

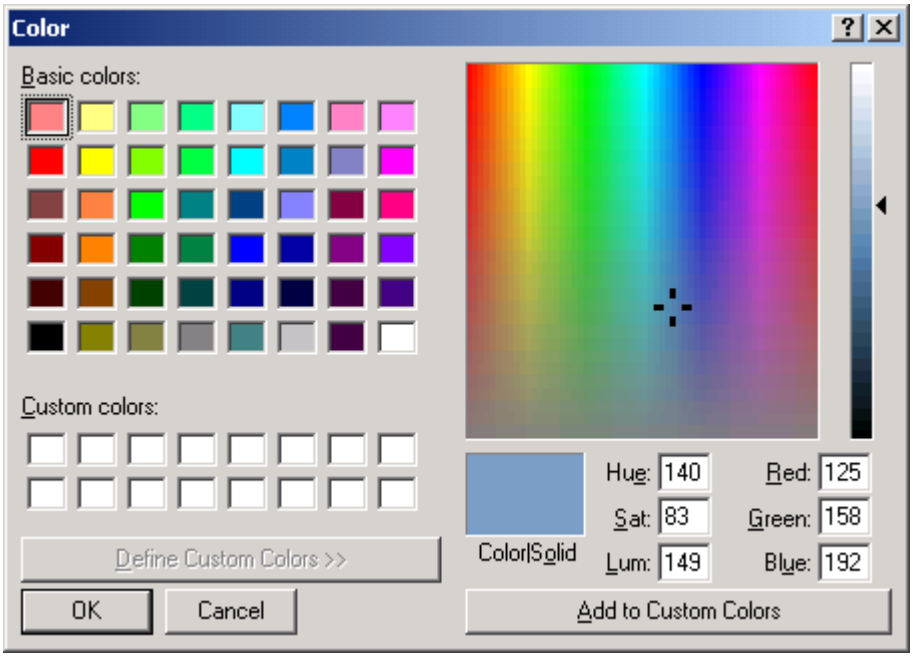
*Note: For more information about camera control, please see Section 4.6.*

- "Set User View as Current Camera View" – Click the "Set" button to specify that the user view should be defined as the current camera view (i.e. the current camera view in the main window). In addition to associating this view with the User View toolbar button, this view becomes the default view when opening the model during subsequent sessions (assuming the configuration file is saved).

  *Note: For more information about the User View toolbar button, please see Section 5.4.*

- "Background" – Click the "Choose a different color..." button to open the color selector dialog box to choose a new background color.






*Note: The color dialog lets the user specify "Custom Colors" to save a color for the current instance of the application. To specify a "Custom Color", choose the custom color square to set (otherwise the default will overwrite the first square), adjust the RGB or HSL as desired, and click the "Add to Custom Colors" button.*

**Exit**

Closes the application.

*Note: Be sure to save your work before you exit. With the exception of the configuration file, the application will NOT prompt you to save your work if it has been modified.*

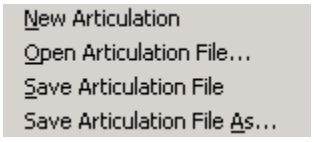
## 4.2 Articulation Menu

The "Articulation" menu enables the user to open and save Articulation Files (`.dm`). When an Articulation File is opened, the current configuration data is updated to point to the opened Articulation File. If the Configuration File is saved, the data will be saved persistently.

**New Articulation File**

Clears the current articulation.

**Open Articulation File...**




"Open Articulation File..." will open the Open dialog which will let the user open an already existing Articulation File. By default, the file filter will be set to show only `.dm` files, but it can be changed to show all files.

**Save Articulation File**

Saves the articulation to a file. If a filename has not been assigned, a Save file dialog will be automatically opened to enable the user to specify the file name and path.

*Note: The application does not warn if the current articulation is not saved when the user opens a new one or exits the application.*

*Note: When the configuration is saved, the articulation is NOT saved. Select "Save Articulation File" from the "Articulation" menu to save it.*

**Save Articulation File As...**

Opens the Save dialog to enable the user to save the articulation to a file.

## 4.3 Environment Menu

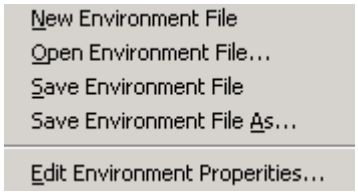
The "Environment" menu enables the user to open and save environment data. The articulation interacts with its environment by contacting it.

**New Environment File**

Clears the current environment and resets it to initial default values.

**Open Environment File...**

"Open Environment File..." will open the Open dialog which will let the user open an already existing Environment File. By default, the file filter will be set to show only .env files, but it can be changed to show all files.




**Save Environment File**

Saves the environment to a file. If a filename has not been assigned, a Save file dialog will be automatically opened to enable the user to specify the file name and path.

*Note: The application does not warn if the current environment is not saved when the user opens a new one or exits the application.*

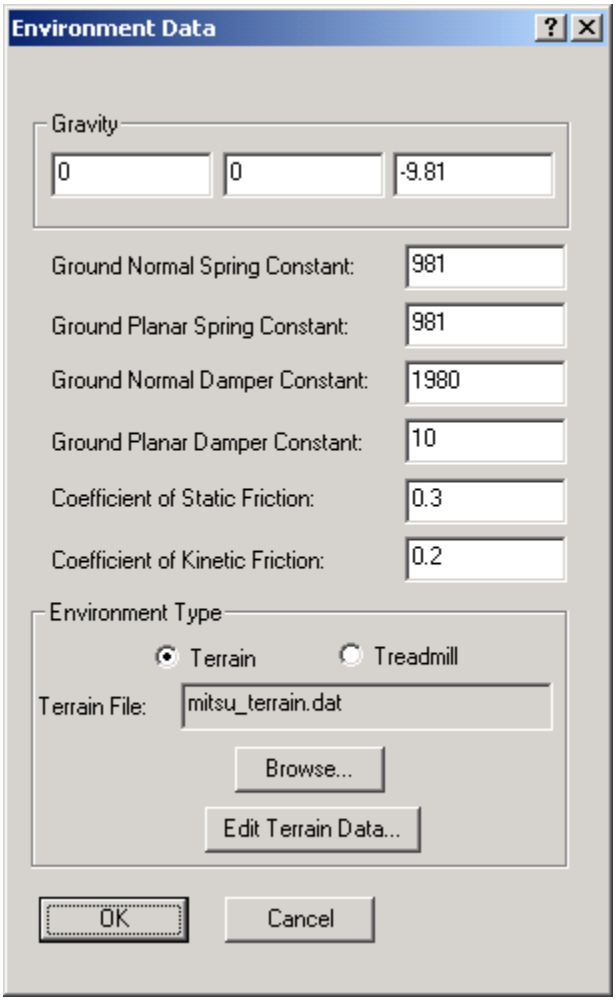
*Note: When the configuration is saved, the environment is NOT saved. Select "Save Environment File" from the "Environment" menu to save it.*

**Save Environment File As...**

Opens the Save dialog to enable the user to save the environment to a file.

**Edit Environment Properties...**

Opens the Environment Data dialog box. The dialog enables the user to define the characteristics of the environment and specify the environment type.




*DynaMechs* models contact with spring and damper effects in the normal and planar directions of the surface, and models sliding across the surface. Choosing the values for the environment parameters varies with the situation. One suggestion is to view the object's interaction with the ground, in

the normal direction, with the equation:

$$ms^2 + bs + k = 0$$

where $s$ is the Laplace variable, $m$ is the mass of the object, $b$ is the damper constant, and $k$ is the spring constant. The $k$ can then be approximated as:

$$k = \frac{mg}{d}$$

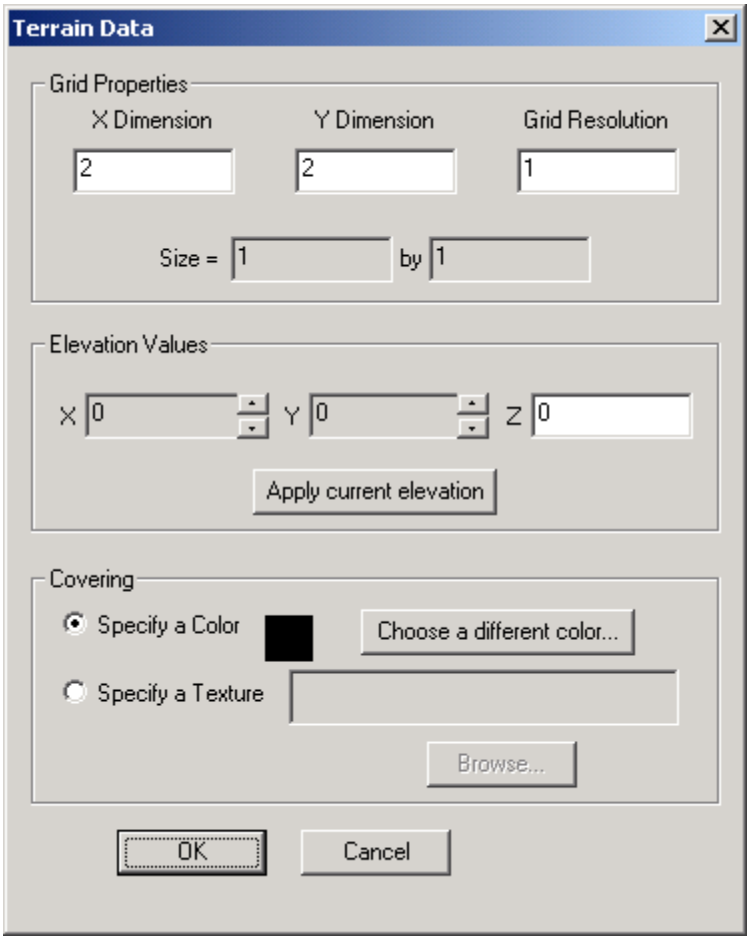
where $g$ is gravity and $d$ is the quiescent deflection of the surface when the object rests on it. If there is more than one contact at the desired quiescent deflection, divide k by the number of contact points. Then calculate $b$ as:

$$b = \sqrt{4k}$$

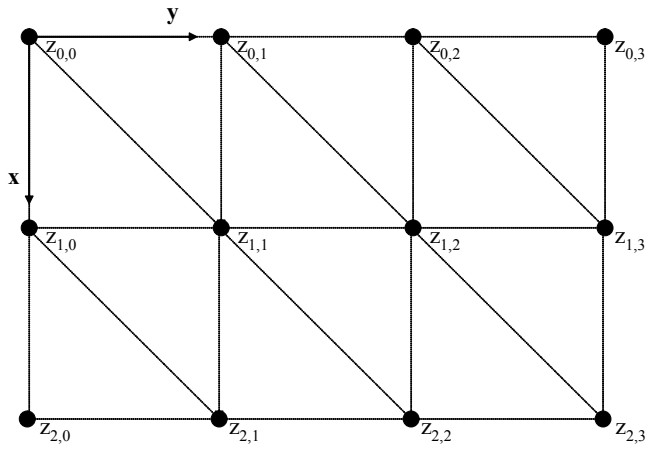
to get approximately critically-damped poles. If $b$ is too small the system may bounce repeatedly on the surface, but if $b$ is too large, the poles will separate widely on the negative real axis and will require smaller integration step size, and thus result in slower simulation. Select the planar spring and damper constants to be equal to the normal values. In addition, specify the static and kinetic coefficients of friction that accurately model the surface.

RobotBuilder supports two types of environments: terrain and treadmill. A terrain environment lets the user specify a matrix of elevations that connect to form triangle strips that define the extents of the contact surface. The treadmill environment provides a flat surface with a velocity that is useful for modeling mobile robots operating on a treadmill.

If the "Environment Type" is "Terrain", clicking the "Edit Terrain Data..." button will open the Terrain Data dialog box to define the terrain. First, the user must specify the size of the terrain by defining the number of elements in the **x** direction and number of elements in the **y** direction. Next the "Grid Resolution" defines the constant spacing in the **x** and **y** directions between elevation points. To clarify, the actual size is shown. Next, the elevation at each point is specified by choosing the **x** index and **y** index, entering the elevation, and clicking the "Apply current elevation" button. The covering for the terrain can be specified as a color or a texture. Clicking "Choose a different color..." will open a color selection dialog box to specify the ambient and diffuse reflectivity properties of the surface.




The surface is generated by connecting the elevation points to form strips of triangles. The result can be seen below:

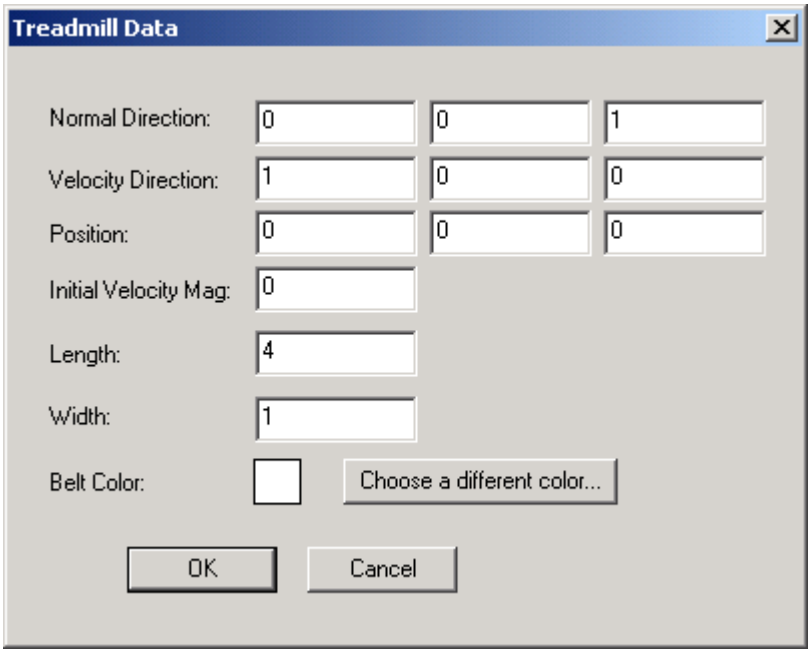



*Note: If a texture file is used, it must be able to be located by the search pattern used in* ***RobotBuilder****. See Section 3.4 for details.*

*Note: The "Apply current elevation" button must be clicked to save an elevation value. Clicking "OK" without clicking that button will NOT save the elevation value.*

If the "Environment Type" on the `Environment Data` dialog box is "Treadmill", clicking the "Edit Treadmill Data" button will open the `Treadmill Data` dialog box (seen below). The dialog box enables the user to define the normal direction, velocity direction, position, initial velocity magnitude, length, width, and belt color of the treadmill. Clicking "Choose a different color..." will open the color selection dialog box to specify the ambient and diffuse reflectivity properties of the treadmill belt. The default values are shown in the screenshot of the `Treadmill Data` dialog box.

*Note: The normal and velocity vectors do not have to be unit length, but they should be orthogonal.* ***RobotBuilder*** *will make them orthogonal if they are not.*




## 4.4 Control Menu

The "Control" menu enables the user to specify the control 

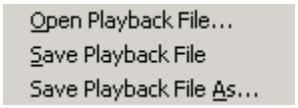
implementation for the simulation. The control is implemented in a user-created control DLL. This flexibility permits the user to develop arbitrary control methods for the simulation.

**Select Control DLL...**

Opens the Open dialog which lets the user select a control DLL. The file filter is set to show only `.dll` files. A control DLL must be specified before RobotBuilder can enter the Simulate mode.

*Note: See Section 7 for more information about the Simulate mode and the control DLL.*

## 4.5 Playback Menu




The "Playback" menu enables the user to open and save playback files. Playback data is generated when the simulation is recorded and can be used to save the resulting animation.

*Note: The Playback mode is discussed in detail in Section 8.*

**Open Playback File...**

"Open Playback File..." will open the Open dialog box which will let the user open an already existing Playback File. The file filter will be set to show only `.rbp` files.

**Save Playback File**

Saves the playback data to a file. If a filename has not been assigned, a Save file dialog will be automatically opened to enable the user to specify the file name and path.

*Note: The application does not warn if the current playback is not saved when the user opens a new one or exits the application.*

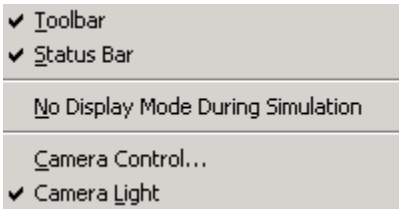
*Note: When the configuration is saved, the playback is NOT saved. Select "Save Playback File" from the "Playback" menu to save it.*

**Save Playback File As...**

Opens the Save dialog to enable the user to save the playback data to a file.

## 4.6 View Menu

The "View" menu enables the user to affect the visibility of some components of the application, and change how the model is viewed in the 3D View.




**Toolbar**

Toggles the visibility of the application's toolbar, located below the menu bar.

*Note: See Section 5 for more information about the toolbar.*

**Status Bar**

Toggles the visibility of the application's status bar. The status bar is located at the bottom of the main window. It shows help information on the left related to the current menu item or toolbar button that the mouse pointer is over and the state of the 'Caps Lock' and 'Num Lock' keys on the right. The current simulation or playback time is shown furthest to the right.


*Note: The simulation time shows 0:00.000 when in the Build mode.*

**No Display Mode During Simulation**


Toggles whether to enable the display of the simulation results by animating the articulation in the 3D View during simulation. Disabling the display update can save some processor time by not requiring the graphics calculations to transform the graphical model. During simulation, the dialog box to the right will be displayed over the RobotBuilder application to indicate why the animation is not updating. This mode can be toggled on or off before entering the Simulate mode, or while the simulation is running. The current simulation time is updated on the dialog box while the simulation is running.

*Note: See Section 7 for more information about the Simulate mode.*

**Camera Control...**

Opens the Camera Control dialog box. The Camera Control dialog box enables the user to change settings that affect how the camera (viewpoint) is positioned and oriented. Before analyzing the camera control settings, the camera coordinate frame must be understood.

The coordinate frame of the camera is defined as the right-hand system with positive **z** pointing in the direction the camera is looking, and **x** pointing to the right, parallel to the window and in the plane of the viewpoint.

The camera is controlled by the mouse in the 3D View window. The application monitors for the left or right mouse button to be pressed down and released. This implies that the mouse can be clicked for an incremental change, or the mouse button can be held down to repeatedly apply the change.

The effect of the change in the camera position is proportional to the distance between the mouse pointer and the center of the window. Consequently, clicking the mouse at the edge of the window will cause the greatest change and clicking near the center of the window will cause a smaller change.

The camera control mode will affect how the application interprets mouse commands made in the 3D View window. There are two main modes available for camera control: `Pan View` and `Examine View`.

In `Pan View`, the camera merely translates in its current plane, or zooms in or out. The mouse affects the camera in the `Pan View` in the following ways:

- Left button in the right half of the window will translate the camera in the camera's **x** direction.

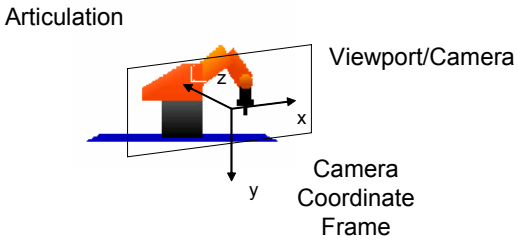



Figure 2: Camera coordinate frame.

- Left button in the left half of the window will translate the camera in the camera's **–x** direction.
- Left button in the top half of the window will translate the camera in the **–y** direction.
- Left button in the bottom half of the window will translate the camera in the **y** direction.
- Right button in the top half of the window will zoom in (translate in **z** direction).
- Right button in the bottom half of the window will zoom out (translate in the **–z** direction).

*Note: The effects will normally be combined. For example, pressing the left mouse button in the top-right part of the window will translate in the **x** and **–y** directions.*

In `Examine View` mode, the camera always points toward a chosen object, referred to as a center of interest, and the camera's movement is constrained to a sphere centered at the center of interest. There are two variations of `Examine View`. In the default mode, the changes are made relative to the inertial coordinate system. When there is a clear default orientation for the object, this mode is sometimes easier to control, because it is easy to rotate the camera to obtain an intuitive view. The other variation makes the changes relative to the camera (viewport) coordinate frame. This mode is convenient to work in because the changes are consistent no matter how the object is viewed.

When in Examine View relative to the inertial coordinate frame, the camera control can be thought of as an angle above or below the **x-y** plane of the inertial coordinate frame translated to the center of interest, and a rotation about the inertial coordinate frame's **z** axis translated to the center of interest.

- Left button in the right half of the window will rotate the camera about the **z** axis. *Note that the apparent direction of the change will switch when the model is upside down.*
- Left button in the left half of the window will rotate the camera about the negative **z** axis.
- Left button in the top half of the window will increase the camera's angle above the **x-y** plane. Increasing to greater than 90º will result in a decrease in the angle with the camera being upside down. This implies that the object can be orbited with the camera in the same plane.
- Left button in the bottom half of the window will decrease the camera's angle above the **x-y** plane (or increase the angle below the **x-y** plane).
- Right button in the top half of the window will zoom in (shrink the sphere centered at the center of interest, that the camera operates on).
- Right button in the bottom half of the window will zoom out (increase the size of the sphere that the camera operates on).

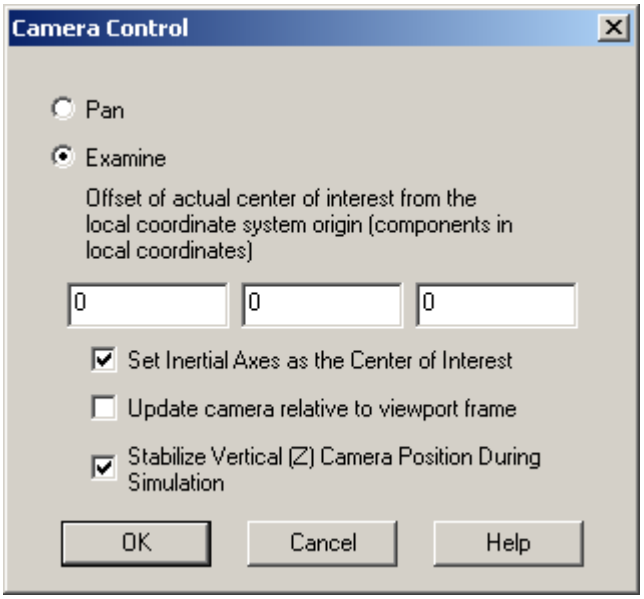
*Note: To be flexible, RobotBuilder allows the user to zoom in within 1 unit of the coordinate frame. If the camera is zoomed inside a model, the model may look invisible because the backfaces of some models may not be rendered.*

When the Examine View is relative to the camera coordinate frame, the camera also operates in a spherical shell centered at the center of interest, but now the changes are relative to the camera coordinate frame:

- Left button in the right half of the window will rotate the camera in a negative angle about the **y** camera frame axis.
- Left button in the left half of the window will rotate the camera in a positive angle about the **y** camera frame axis.
- Left button in the top half of the window will rotate the camera in a negative angle about the **x** axis
- Left button in the bottom half of the window will rotate the camera in a positive angle about the **x** axis.
- Right button in the top half of the window will zoom in (shrink the sphere centered at the center of interest, that the camera operates on).
- Right button in the bottom half of the window will zoom out (increase the size of the sphere that the camera operates on).

The settings on the Camera Control dialog box can now be easily understood. The top two radio buttons switch between the Pan and Examine Views discussed above. The user has the option to specify an offset from the object that is the center of interest for the camera. This offset is in terms of the link coordinate system that is specified as the center of interest. Note that when in Pan View the center of interest offset is continuously adjusted so that the center of interest remains in the center of the viewport frame. Next is a checkbox that enables the user to set the inertial axes as the center of

interest. Following this is a checkbox that sets whether the changes are made relative to the viewport (also referred to as camera) coordinate frame. If that is not checked, the changes will be relative to the inertial coordinate frame. Finally there is a checkbox that indicates whether or not the camera position should be stabilized in the vertical (inertial Z) direction during simulation. When this box is checked, the camera does not track vertical motion of the center of interest that is along the inertial Z axis (perpendicular to the horizontal X-Y ground plane). This prevents the ground from appearing to "bob" up and down during simulation of a mobile vehicle when the center of interest is set on the body.




*Note: Whenever an articulation is opened, the default center of interest is the inertial coordinate frame. If the center of interest is ever changed, the only way to make the inertial coordinate frame the center of interest again is to choose the setting in the* `Camera Control` *dialog box.*

*Note: When RobotBuilder starts, the default camera control is set to Examine View relative to the inertial coordinate frame with the inertial coordinate frame being the center of interest. The vertical camera position is stabilized during simulation by default.*

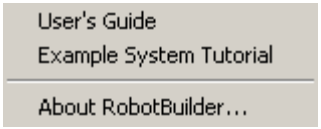
**Camera Light**

Toggles the light that points in the direction of the camera, on or off.

*Note: See Section 6.4's discussion on lighting.*

## 4.7 Help Menu

The "Help" menu enables the user to get help for the application and see general information about the application.




**User's Guide**

Opens the RobotBuilder User's Guide (this document) with the PC's `.pdf` viewer (probably *Adobe Reader*). The document is located in the `Docs` subdirectory.

**Example System Tutorial**

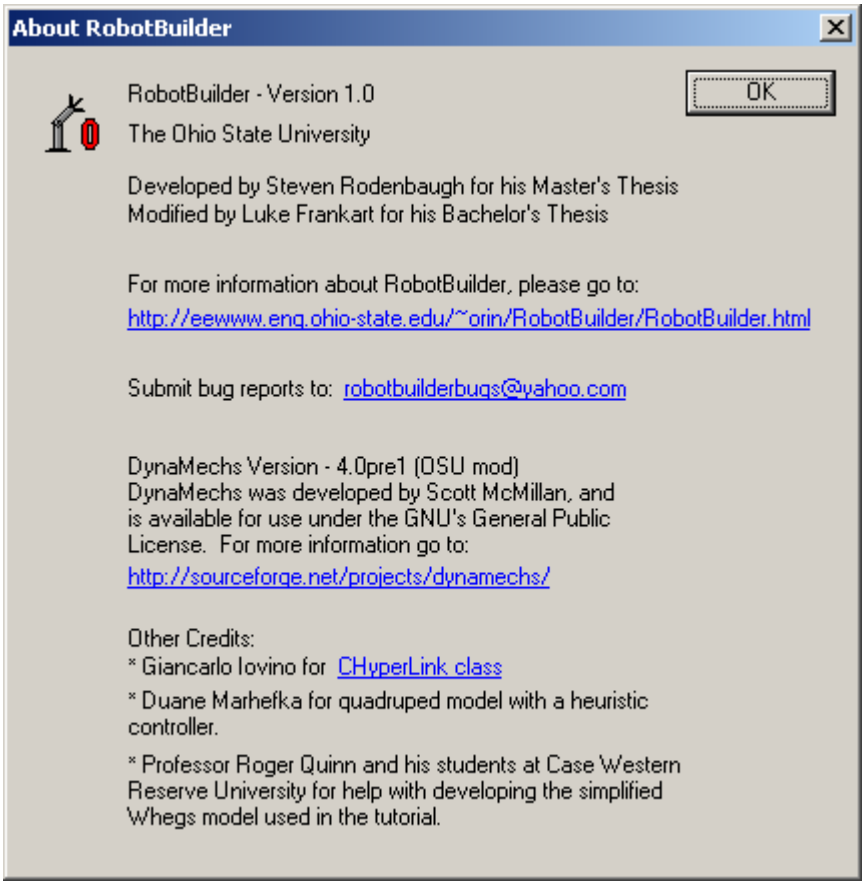
Opens the RobotBuilder Example System Tutorial with the PC's `.pdf` viewer (probably *Adobe Reader*). The document is located in the `Docs` subdirectory.

**About RobotBuilder...**

Opens the About RobotBuilder dialog box. This dialog displays the version number, and a website to get more information.

*Note: Although the version is listed as 1.0, the application has not been extensively tested in different environments. Please send a bug report to the email address in the dialog if a problem is encountered.*

*Note: When submitting a bug report, please give as much information as possible. Please send the application version number, your OS version, your video card and driver version, the problem that is being experienced, and the steps to reproduce that problem.*



# 5. Toolbar

The toolbar provides one-click access to many commonly used functions in RobotBuilder. The toolbar is located below the menu bar, and can be seen below:

## 5.1 File Functions

These three buttons provide shortcuts for menu items on the "CFG File" menu.

**New** 

Clears the current configuration, articulation, and environment data. See Section 4.1 – "New" for details.

**Open** 

Opens a dialog to choose a `.cfg` file to open. See Section 4.1 – "Open" for details.

**Save** 

Saves the current configuration to a file. See Section 4.1 – "Save" for details.

## 5.2 Application Mode Functions

RobotBuilder operates in three main modes: Build, Simulate, and Playback. These buttons enable changing between the three modes.

**Build** 

When pressed down, the application is in the Build mode. Otherwise, pressing it will put the application into the Build mode.

*Note: See Section 6 for more information about the Build mode.*

**Simulate** 

When pressed down, the application is in the Simulate mode. Otherwise, pressing it will put the application into the Simulate mode.

*Note: See Section 7 for more information about the Simulate mode.*

**Playback** 

When pressed down, the application is in the Playback mode. Otherwise, pressing it will put the application into the Playback mode. This button is only enabled when there is valid playback data for the current simulation in the playback buffer.

*Note: See Section 8 for more information about the Playback mode.*

## 5.3 Simulation and Playback Control Functions

When RobotBuilder is in the Simulate or the Playback modes, these buttons are enabled to allow the user to control the progression of the simulation or playback. These buttons are analogous to DVD control buttons.

*Note: See Section 7 for more information about the Simulate mode and Section 8 for more information about the Playback mode.*

**Play** 

Starts the simulation or playback.

**Pause** 

Pauses the simulation or playback. Pressing "Play" will restart the simulation or playback from its current state.

**Stop** 

Stops the simulation or playback and resets to initial values.

**Record** 

Enabled only in the Simulate mode, this button will record the simulation for later playback.

## 5.4 View Functions

These buttons enable the user to easily change the position and orientation of the camera to predefined points. The first six buttons will orient the camera to be perpendicular to a particular plane indicated by the button and position the camera on the side of the model indicated by the button. The camera is then zoomed in or out. The zoom function will place the camera so that the model is wide enough to fit in the window, and the window will be centered on the center of interest (if applicable). The seventh button orients and zooms the camera to a user-defined view.

*Note: To clarify, the model might not completely appear in the window, because the camera will place the center of interest at the center.*

*Note: The zoom setting will likely change to fit the model in the window after pressing one of these buttons.*

**Front View** 

Moves the camera to look at the "front" of the robotic articulation (camera coordinate **x-y** plane is parallel to inertial coordinate **x-z** plane and camera looks toward the inertial positive **y** direction).

**Right View** 

Moves the camera to look at the "right" side of the robotic articulation (camera coordinate **x-y** plane is parallel to inertial coordinate **y-z** plane and camera looks toward the inertial negative **x** direction).

**Back View** 

Moves the camera to look at the "back" side of the robotic articulation (camera coordinate **x-y** plane is parallel to inertial coordinate **x-z** plane and camera looks toward the inertial negative **y** direction).

**Left View** 

Moves the camera to look at the "left" side of the robotic articulation (camera coordinate **x-y** plane is parallel to inertial coordinate **y-z** plane and camera looks toward the inertial positive **x** direction).

**Bottom View** 

Moves the camera to look at the "bottom" side of the robotic articulation (camera coordinate **x-y** plane is parallel to inertial coordinate **x-y** plane and camera looks toward the inertial positive **z** direction).

**Top View** 

Moves the camera to look at the "top" side of the robotic articulation (camera coordinate **x-y** plane is parallel to inertial coordinate **x-y** plane and camera looks toward the inertial negative **z** direction).

**User View** 

Moves the camera to the user-defined view. The particular orientation, position, and zoom are specified by the user through the Simulation Properties dialog box under the "CFG" file menu. If no user-defined view has been specified, this button behaves the same as the Front View button.

*Note: Please see Section 4.1 for information on specifying the user camera view.*

## 5.5 Camera Control Functions

These buttons enable the user to easily switch between the Pan View and Examine View.

**Pan View** 

Switches the camera control to Pan View. Clicking this button is the same as selecting the Pan View radio button in the Camera Control dialog box under the "View" menu.

**Examine View** 

Switches the camera control to Examine View. Clicking this button is the same as selecting the Examine View radio button in the Camera Control dialog box.

## 5.6 Zoom Functions

These buttons enable the user to easily zoom the camera in or out.

**Zoom In** 

Translates the camera in the camera coordinate frame **z** direction.

**Zoom Out** 

Translates the camera in the camera coordinate frame **-z** direction.

## 5.7 Axes Visibility Functions

RobotBuilder has the ability to show the direction of the axes of the inertial and link coordinate frames by using a 2D overlay. This implies that the axes will always appear on top of the graphics models, but this enables the user to easily see the position and orientation of the axes. The axes are also labeled with **x**, **y**, and **z** for clarity. These buttons show or hide the inertial coordinate frame axes and the axes of the individual link coordinate frames.

*Note: The axis labels are by default displayed with an Microsoft Sans Serif (size 9) font, specified in the* `font.wtk` *file found in the same directory as* `RobotBuilder.exe`*. If the labels appear bulky, this file may be missing, the font may not be installed on the system, or the working directory may not be properly specified when the application is started.*

*Note: The axes of the links can be toggled on or off individually by the context menu in the link tree view (see Section 6).*

**Toggle Inertial Axes** 

This button is stateful. When in the down position, the inertial axes will be drawn. When in the up position, the inertial axes will not be drawn.

**Show All Link Axes** 


When pressed, the axes of all links in the robotic articulation will be shown.

*Note: The size of the axes can be set in the `Simulation Properties` dialog box accessed by selecting "CFG File"->"Edit Simulation Properties...". See Section 4.1 for more details.*

**Hide All Link Axes** 

When pressed, the axes of all links in the robotic articulation will be hidden.

## 5.8 RobotModeler

**RobotModeler** 

Starts a new instance of RobotModeler.

# 6. Build Mode

The initial mode RobotBuilder starts with is the Build mode. In this mode, the user defines the components of the simulation and specifies their parameters. The user will mainly interact with the Link Tree View found in the left window of the application. The graphical representation of the current robotic articulation can be seen in the 3D View in the right window.

To define the simulation, the user needs to define the properties of the simulation, the articulation, and the environment. The simulation properties can be defined by opening an existing configuration, or by the Simulation Properties dialog box (See Section 4.1 for more information). The environment properties can be defined by opening an existing environment or by the Environment Data dialog box (See Section 4.3 for more information). The articulation is defined by opening an existing one, or creating one with RobotBuilder by interacting with the Link Tree View.

## 6.1 Link Tree View

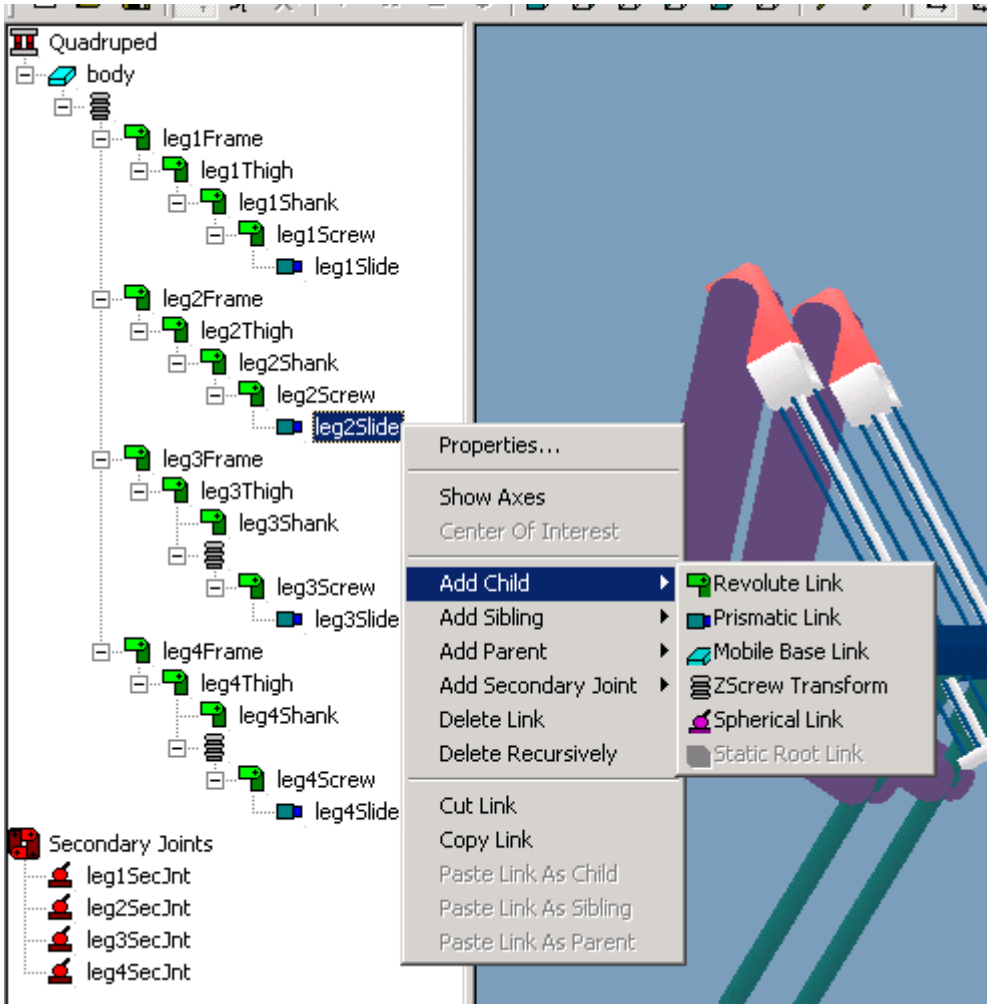
The Link Tree View is in the left splitter window of RobotBuilder during the Build mode. The Link Tree View shows the hierarchical relationships between link-joint pairs. It also provides context menus that enables the user to edit the parameters of the link-joint pairs and change the relationships between the link-joint pairs.

The Link Tree View is implemented with a Windows tree control, similar to what is found in *Explorer* for displaying the directory structure on a computer. The user can

click the plus sign preceding a tree item to expand its children, or click the minus sign in front of a tree item to collapse its children. Each tree item is also preceded by a small icon that indicates the type of tree item. The icons used are:

| | |
|---|---|
|  | Articulation |
|  | Closed Articulation |
|  | Mobile Base Link |
|  | Prismatic Link |
|  | Revolute Link |
|  | Spherical Link |
|  | Static Root Link |
|  | Z Screw Transform |
|  | Secondary Joint List |
|  | Secondary Prismatic Joint |
|  | Secondary Revolute Joint |
|  | Secondary Spherical Joint |

The first tree item in the Link Tree View is always an Articulation or Closed Articulation item (for articulations with closed-kinematic loops). Below that item is the hierarchical representation of the link-joint pair relationships. For a closed-loop articulation, the hierarchical relationships are for the spanning tree. If the articulation is closed, there will be a list of secondary joints following the articulation. See below:




## 6.2 Robotic Articulation Components and Properties

A robotic articulation consists of a coordinate frame and a collection of constituent link-joint pairs. If the articulation has one or more closed loops, it will also possesses a list of secondary joints that close the loops. Each component has data that needs to be specified by the user so *DynaMechs* (the simulation library used in RobotBuilder) can accurately simulate the desired robotic articulation.

The properties of the articulation components will be discussed below, and are accessed in RobotBuilder by selecting the item and choosing "Properties..." from the context menu (see Section 6.3 for more information about the context menu). This will open a property sheet that contains property pages with data about the selected item.

## 6.2.1 Property Sheets

### 6.2.1.1 Articulation Property Sheet

The articulation item is at the root of all open-loop robotic articulations. The articulation property sheet consists of:

- Articulation Data Property Page (See Section 6.2.2.2)
- Object Data Property Page (See Section 6.2.2.1)

### 6.2.1.2 Closed Articulation Property Sheet

The closed articulation item is at the root of all closed-loop robotic articulations. The closed-loop articulation property sheet consists of:

- Articulation Data Property Page (See Section 6.2.2.2)
- Object Data Property Page (See Section 6.2.2.1)

### 6.2.1.3 Mobile Base Link Property Sheet

The mobile base link-joint pair implements a six degree-of-freedom joint that can be used as the base of a mobile robot. Its property sheet consists of:

- Mobile Base Link Data Property Page (See Section 6.2.2.3)
- Rigid Body Data Property Page (See Section 6.2.2.4)
- Link Data Property Page (See Section 6.2.2.5)

  *Note: In the current implementation of DynaMechs, the joint friction does not affect the mobile base link.*
- Object Data Property Page (See Section 6.2.2.1)

#### 6.2.1.4 Prismatic Link Property Sheet

The prismatic link-joint pair implements a prismatic joint. Its property sheet consists of:

- Link Data Property Page (See Section 6.2.2.5)
- MDH Link Data Property Page (See Section 6.2.2.7)
- Rigid Body Data Property Page (See Section 6.2.2.4)
- Object Data Property Page (See Section 6.2.2.1)

#### 6.2.1.5 Revolute Link Property Sheet

The revolute link-joint pair implements a revolute joint. Its property sheet consists of:

- Actuator Data Property Page (See Section 6.2.2.6)
- MDH Link Data Property Page (See Section 6.2.2.7)
- Rigid Body Data Property Page (See Section 6.2.2.4)
- Object Data Property Page (See Section 6.2.2.1)

#### 6.2.1.6 Spherical Link Property Sheet

The spherical link-joint pair implements a spherical joint with 3 rotational degrees of freedom. Its property sheet consists of:

- Spherical Link Data Property Page (See Section 6.2.2.8)
- Link Data Property Page (See Section 6.2.2.5)
- Rigid Body Data Property Page (See Section 6.2.2.4)
- Object Data Property Page (See Section 6.2.2.1)

#### 6.2.1.7 Static Root Link Property Sheet

The static root link implements a link that is used to close a loop that connects through a fixed base. This is only used as the first child of a closed articulation. Its property sheet consists of:

- Object Data Property Page (See Section 6.2.2.1)

#### 6.2.1.8 ZScrew Transform Property Sheet

The **z** screw transform implements a fixed translation and rotation about the **z** axis for efficient modeling of branches in tree structures. Its property sheet consists of:

- ZScrew Transform Data Property Page (See Section 6.2.2.9)
- Object Data Property Page (See Section 6.2.2.1)

*Note: This has no associated graphical model.*

#### 6.2.1.9 Secondary Prismatic Joint Property Sheet

Closes a kinematic loop with a prismatic joint. Its property sheet consists of:

- Secondary Joint Data Property Page (See Section 6.2.2.10)
- Secondary Prismatic Joint Data Page (See Section 6.2.2.11)
- Object Data Property Page (See Section 6.2.2.1)

*Note: This has no associated graphical model.*

#### 6.2.1.10 Secondary Revolute Joint Property Sheet

Closes a kinematic loop with a revolute joint. Its property sheet consists of:

- Secondary Joint Data Property Page (See Section 6.2.2.10)
- Secondary Revolute Joint Data Page (See Section 6.2.2.12)
- Object Data Property Page (See Section 6.2.2.1)

*Note: This has no associated graphical model.*

#### 6.2.1.11 Secondary Spherical Joint Property Sheet

Closes a kinematic loop with a spherical joint. Its property sheet consists of:

- Secondary Joint Data Property Page (See Section 6.2.2.10)
- Secondary Spherical Joint Data Page (See Section 6.2.2.13)
- Object Data Property Page (See Section 6.2.2.1)

*Note: This has no associated graphical model.*

## 6.2.2 Property Pages

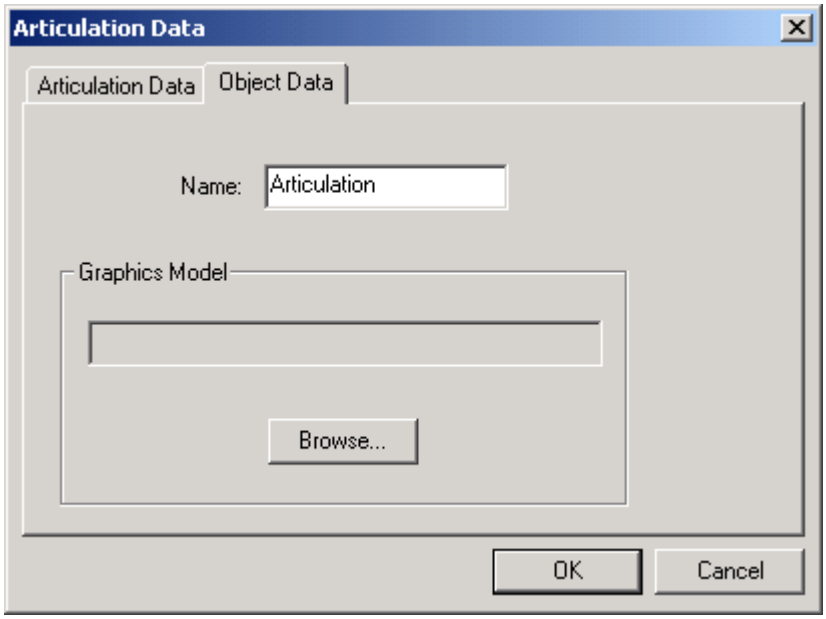
### 6.2.2.1 Object Data Property Page

This property page defines general object parameters. Specifically it enables the user to specify a name for the item and a graphical representation.

*Note: Some items are logical (rather than physical) and have no graphical representation. In this case, the edit box that displays the graphical representation will not be visible.*

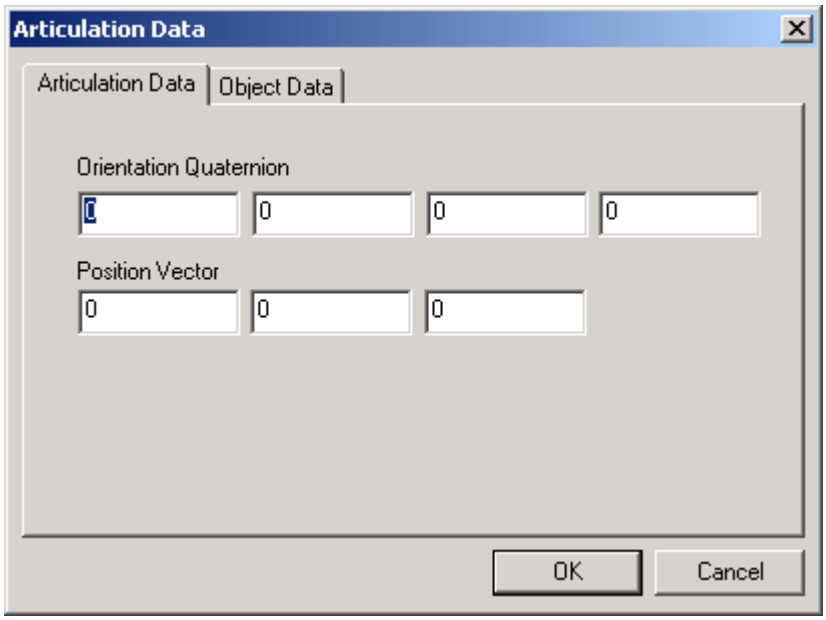
*Note: The name does not have to be unique, but it is recommended (for example, when closing a loop with a secondary joint, the joined links, specified by name, will not be clear if the names are not unique).*

*Note: The default value for the name is a text description of the item. For example, the default name for an articulation item is "Articulation".*



### 6.2.2.2 Articulation Data Property Page

This property page defines the position and orientation of the articulation coordinate frame.



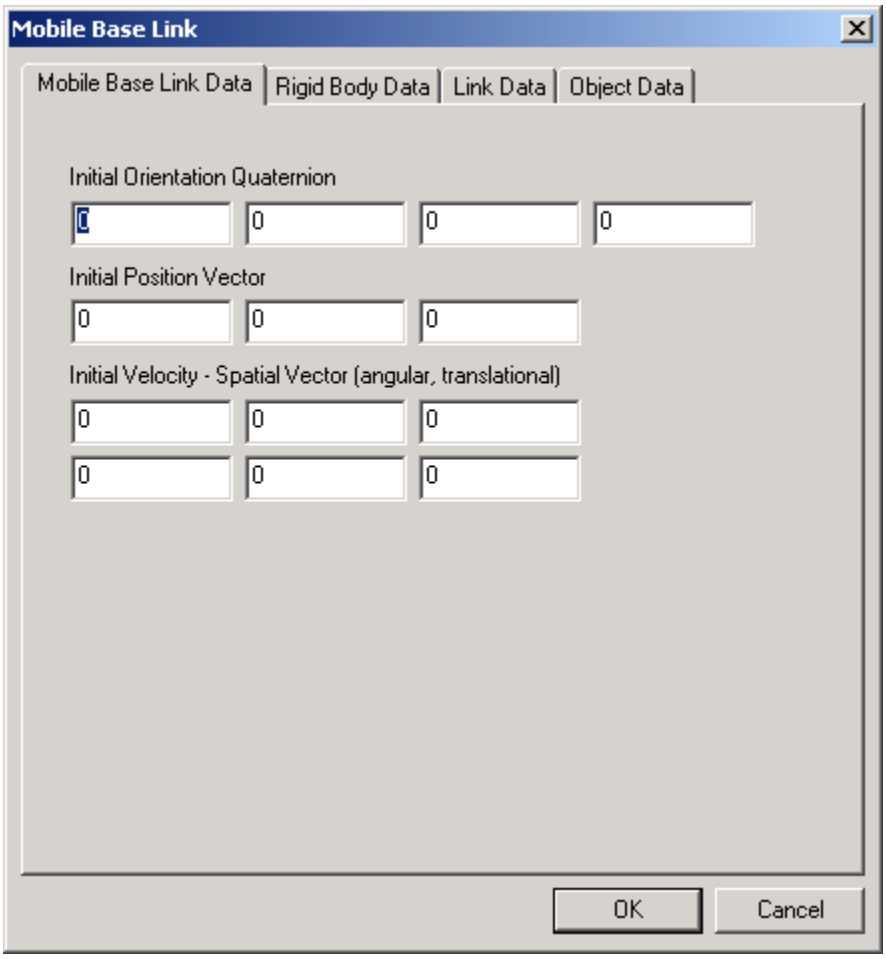

*Note: The default values are shown.*

### 6.2.2.3 Mobile Base Link Data Property Page

Enables the user to view and edit the mobile base parameters:

- "Initial Orientation Quaternion" – the relative orientation of the link's coordinate axes to its parent's coordinate frame.
- "Initial Position Vector" – position from parent coordinate frame to link's coordinate frame in terms of the parent's coordinate frame.

- "Initial Velocity – Spatial Vector" – initial spatial velocity (roll rate, pitch rate, yaw rate, $\dot{\mathbf{x}}$, $\dot{\mathbf{y}}$, and $\dot{\mathbf{z}}$) with respect to parent link, in terms of parent coordinate frame.



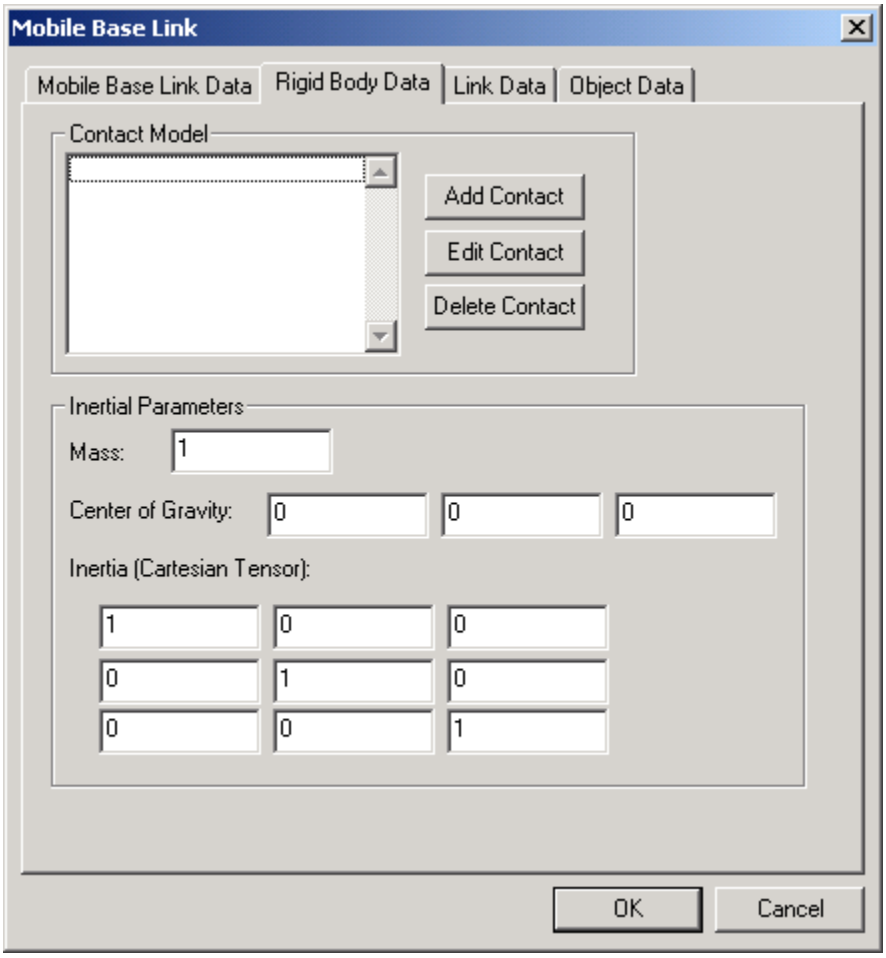

#### 6.2.2.4 Rigid Body Data Property Page

Enables the user to view and edit the parameters for links that have rigid body characteristics. This property page defines the points on the model that can contact the environment. If no contact points are specified, the object will pass through the environment unaffected. Note that contact is only detected with the environment;

therefore, objects are not restricted from passing through each other (the boundary testing is computationally intensive):

- "Add Contact" – opens a dialog box that the user can enter a contact's location in the link coordinates. The user can specify any number of contact points.
- "Edit Contact" – will let the user change the **x**, **y**, and **z** position of the currently selected contact point.
- "Delete Contact" – will delete the currently selected contact from the contact list.
- "Mass" – the mass of the link.

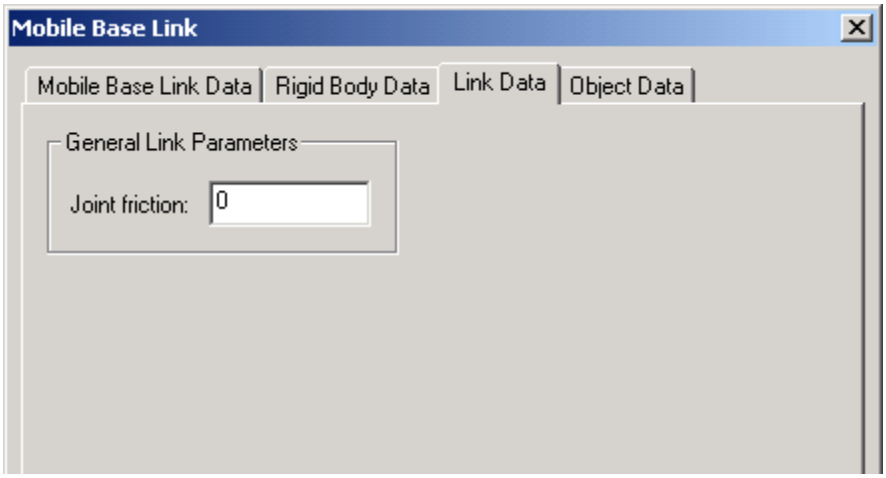
  *Note: See Section 9.1 for a discussion on units*
- "Center of Gravity" – the center of gravity of the link in the link's coordinates.
- "Inertia (Cartesian Tensor)" – the inertia tensor of the link.




### 6.2.2.5 Link Data Property Page

Enables the user to view and edit the link's joint friction.

- "Joint friction" – friction at the joint that joins the link to its parent.

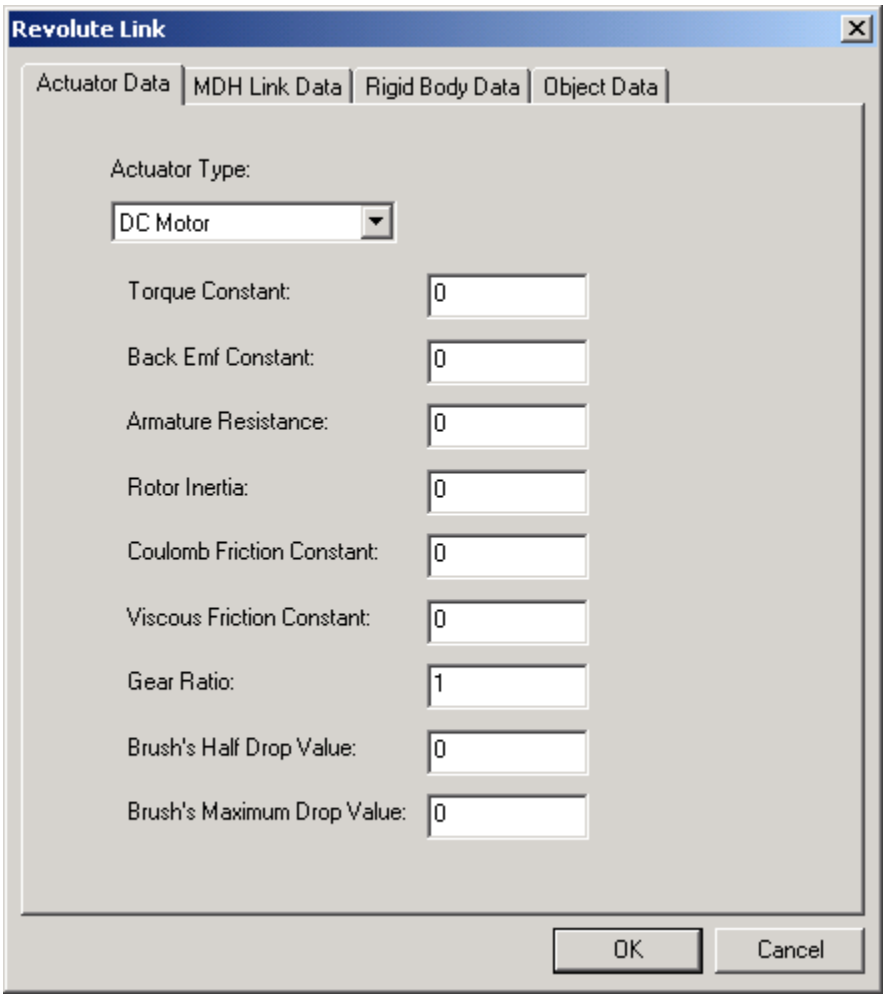



#### 6.2.2.6 Actuator Data Property Page

Enables the user to view and edit the link's actuator. The only supported actuator presently is a DC motor for a revolute joint. The combo box on the property page also lets the user choose "No Motor", and the page changes to just one field to enter the joint friction between the link and its parent. If the "Actuator Type" is "DC Motor", then the following fields are available:

- "Torque Constant"
- "Back Emf Constant"
- "Armature Resistance"
- "Rotor Inertia"
- "Coulomb Friction Constant"
- "Gear Ratio"
- "Brush's Half Drop Value" – for brush motors. Enter 0 for brushless motors.

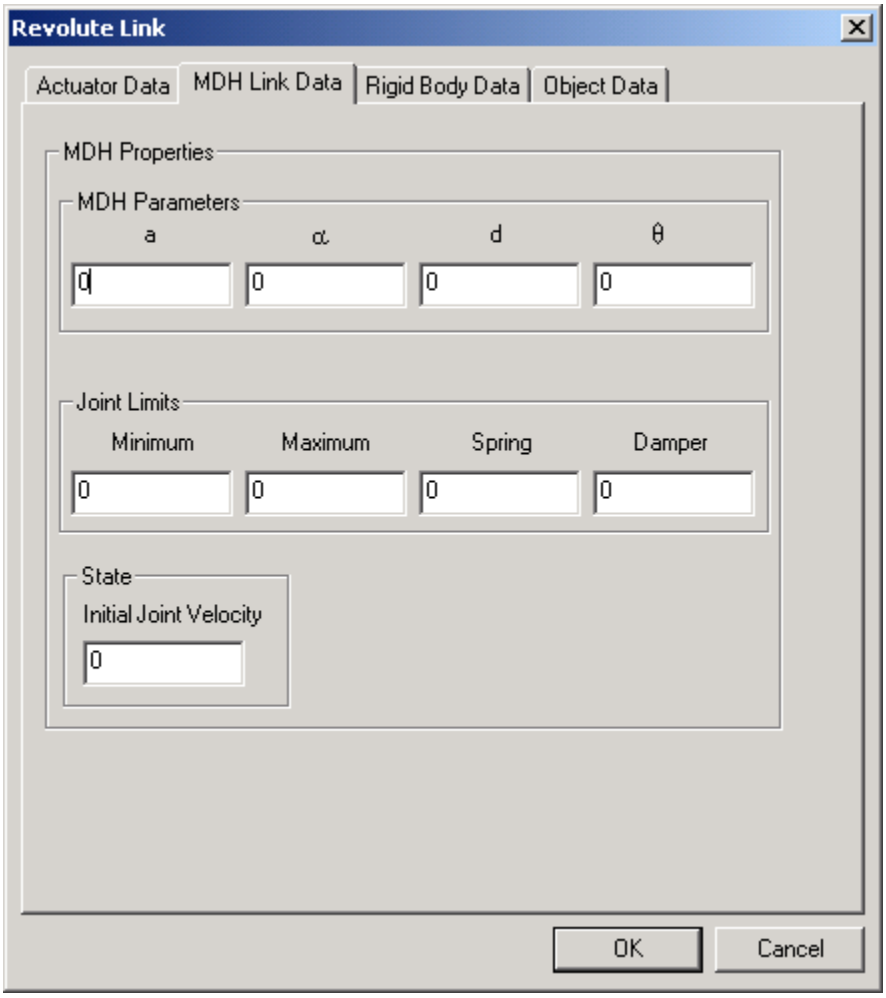
- "Brush's Maximum Drop Value" – for brush motors. Enter 0 for brushless motors.



#### 6.2.2.7 MDH Data Property Page

Enables the user to view and edit links that can be specified using Modified Denavit-Hartenberg notation [4]. The normal convention for links specified by Modified Denavit-Hartenberg notation is for their **z** axis to be coincident with the joint axis and the **x** axis is along the vector that is perpendicular to the current **z** and the next link's **z**.

- "MDH Parameters"
  - "a" – distance between link's **z** and the next link's **z** along current link's **x**.
  - "α" – angle from previous link's **z** to the current link's **z**.
  - "d" – distance along **z** from previous link's **x** axis intersection to the current link's **x** axis intersection with the **z** axis.
  - "θ" – angle from previous link's **x** to current link's **x** about the joint **z** axis.
- "Joint Limits" – specifies a valid range of motion for the joint. If this is exceeded, a compliant limit force is simulated using the spring and damper values specified.
  - "Minimum"
  - "Maximum"
  - "Spring"
  - "Damper"
- "Initial Joint Velocity"

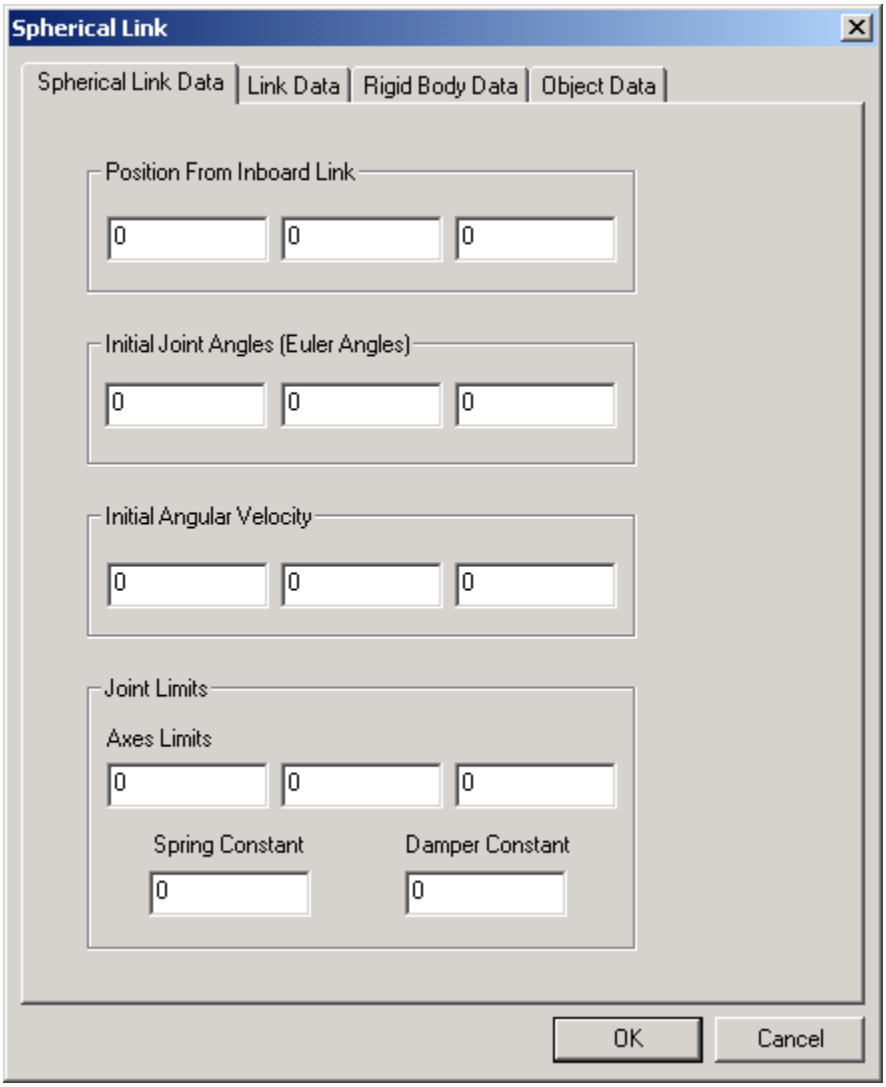



### 6.2.2.8 Spherical Link Data Property Page

Enables the user to view and edit spherical link parameters:

- "Position From Inboard Link" – position in parent link's coordinates.
- "Initial Joint Angle (Euler Angles)" – the angles are in radians. The first is the rotation about the parent's **z** axis. The second is about the once rotated **y** axis. The third is about the twice rotated **x** axis (which is the same as the current link's **x** axis).

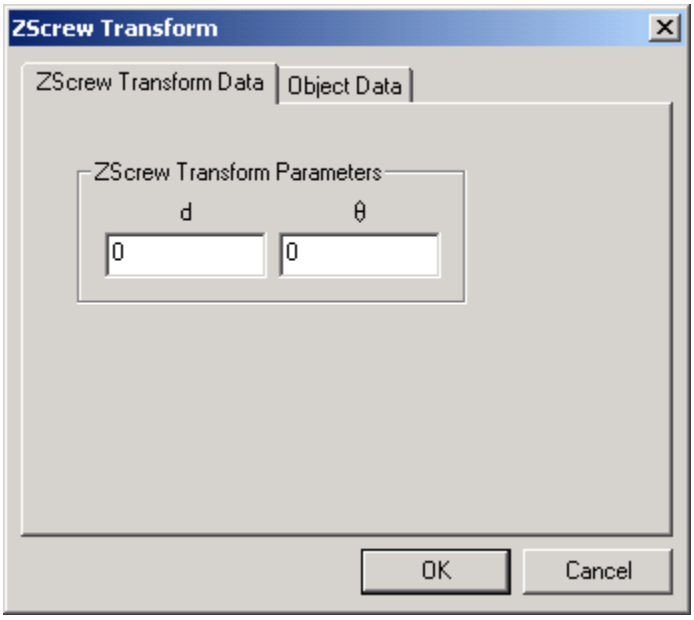
- "Initial Angular Velocity" – angular velocity of the link relative to the parent's coordinate frame in the current coordinates. Note this is not the time derivative of the Euler angles.
- "Joint Limits"
    - "Axes Limit" – the maximum angle between the parent link and the current link's axes. Using a 0 will imply no restrictions, and the order is **x**, **y**, and **z**.
    - "Spring Constant" and "Damper Constant" – if the joint limit is exceeded, a restorative force will be applied using the spring and damper constants provided.




### 6.2.2.9 ZScrew Transform Data Property Page

Enables the user to view and edit **z** screw transform parameters:

- "d" – the distance along the link's **z** from the parent coordinate frame to the current coordinate frame.
- "θ" – the rotation about the link's **z** to the parent's coordinate frame.

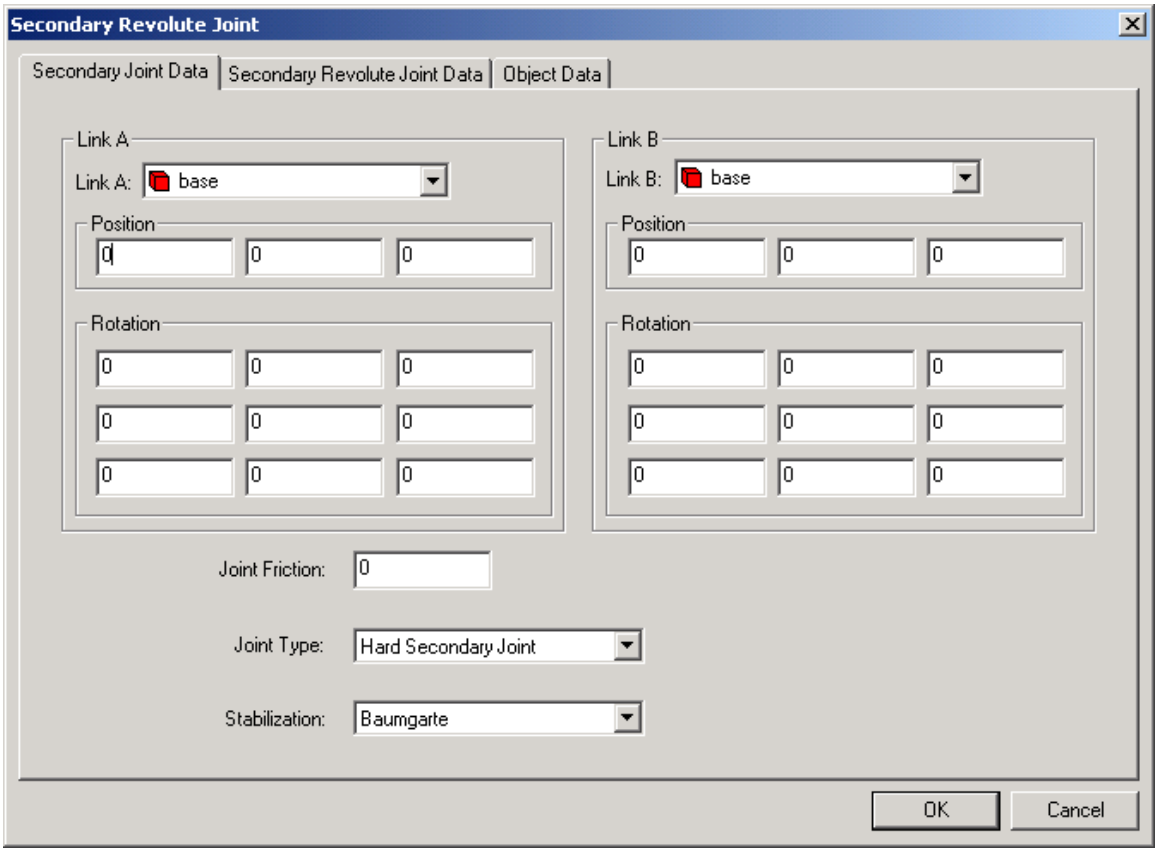



#### 6.2.2.10 Secondary Joint Data Property Page

Enables the user to view and edit secondary joint parameters. The user must specify the links that are joined, and their relationships:

- "Link A" and "Link B" – choose the links to join from the combo boxes. Next specify the position and orientation of the secondary joint's two coordinate frames relative to the joined links' frames. For prismatic secondary joints, the coordinate frames should be parallel and the degree of freedom will be along the **z**. For revolute secondary joints, the origins should be equal, and **z** axes coincident. For spherical secondary joints, the origins of the two coordinate frames should be equal [5].
- "Joint Friction" – friction to apply to joint.

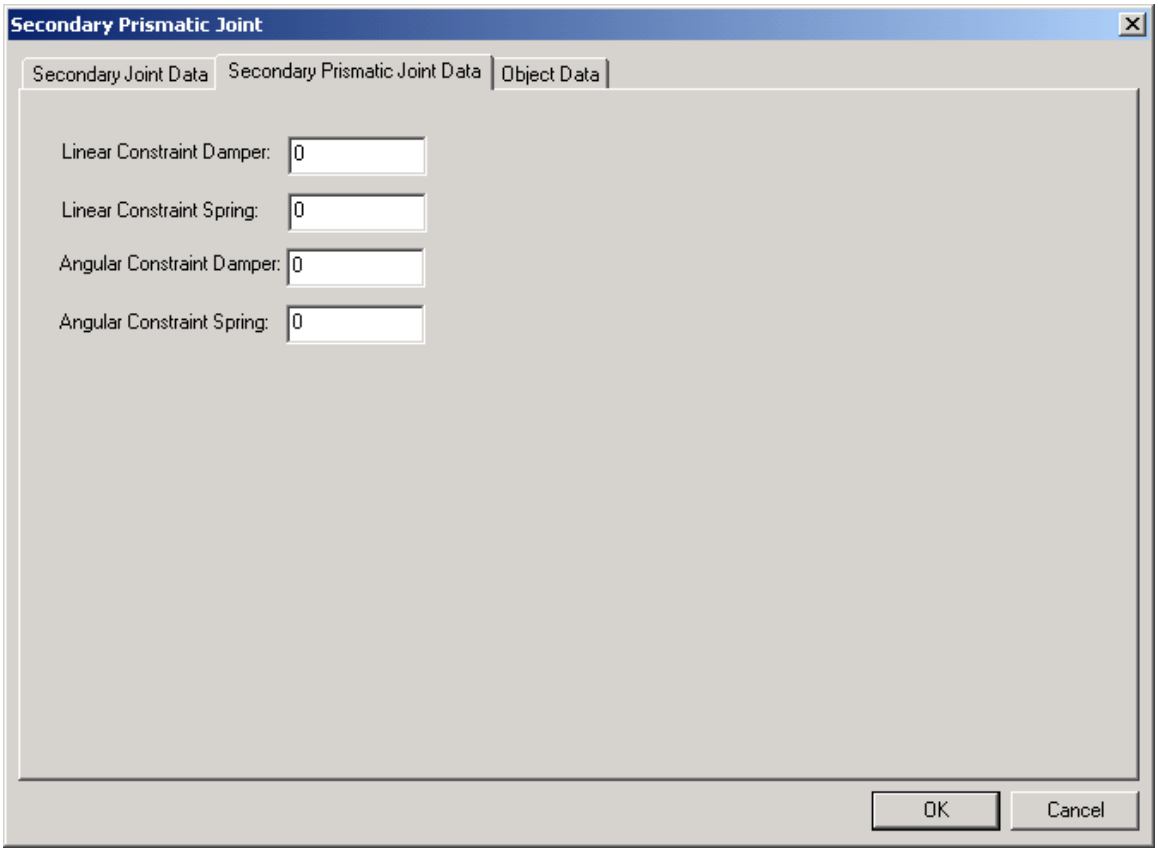
- "Joint Type" – supported types are "Hard Secondary Joint" and "Soft Secondary Joint". A "Hard Secondary Joint" has constraint stabilization applied to the joint.
- "Stabilization" – specifies the constraint stabilization to utilize when the "Joint Type" is "Hard Secondary Joint". The options are "Baumgarte", "Spring and Damper", and "None".




### 6.2.2.11 Secondary Prismatic Joint Data Property Page

Enables the user to view and edit secondary prismatic joint parameters. The property page sets the parameters for implementing the constraints. The meaning and use of the

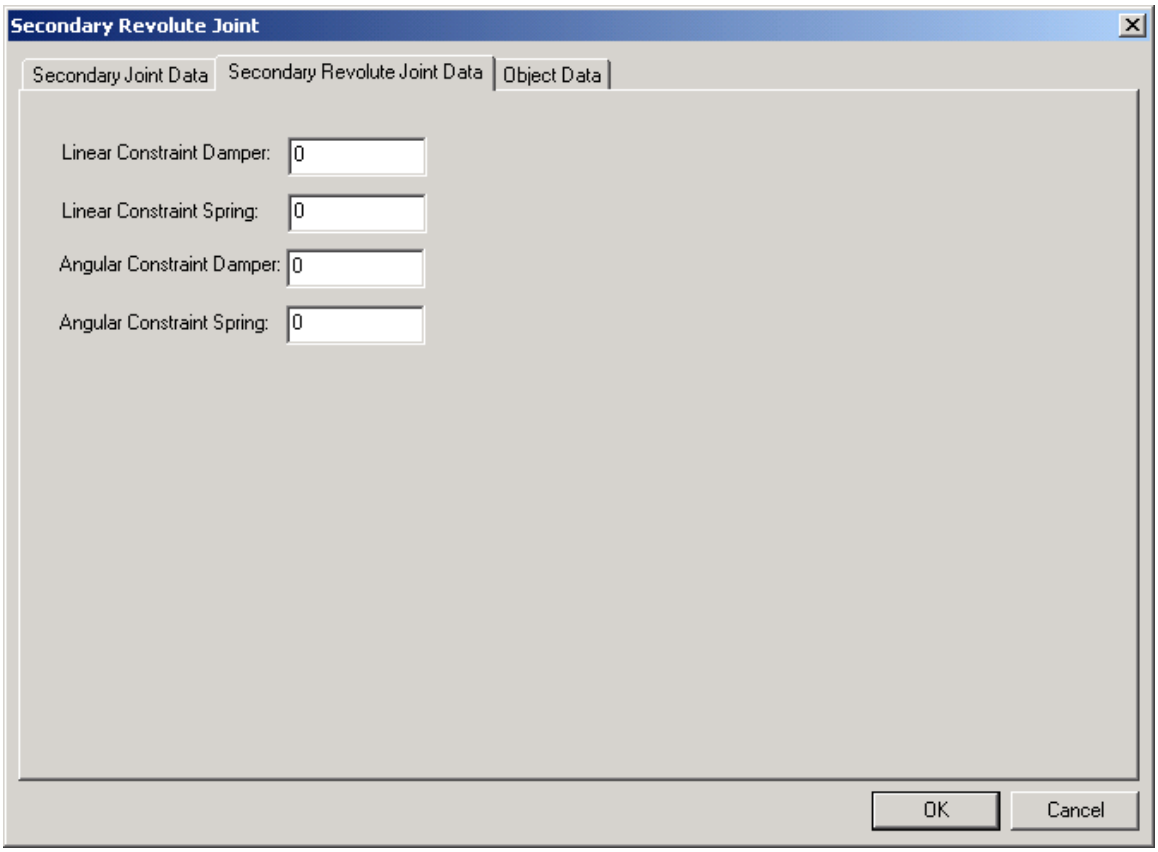
parameters depends on the type of secondary joint. For soft joints, the terms are the linear and angular springs and dampers which implement the joint constraints. For hard joints with spring/damper stabilization, the terms are the springs and dampers which implement the constraint stabilization. For hard joints with Baumgarte stabilization, the terms are interpreted as the Baumgarte coefficients.




#### 6.2.2.12 Secondary Revolute Joint Data Property Page

Enables the user to view and edit secondary revolute joint parameters. The property page sets the parameters for implementing the constraints. The meaning and use of the parameters depends on the type of secondary joint. For soft joints, the terms are the

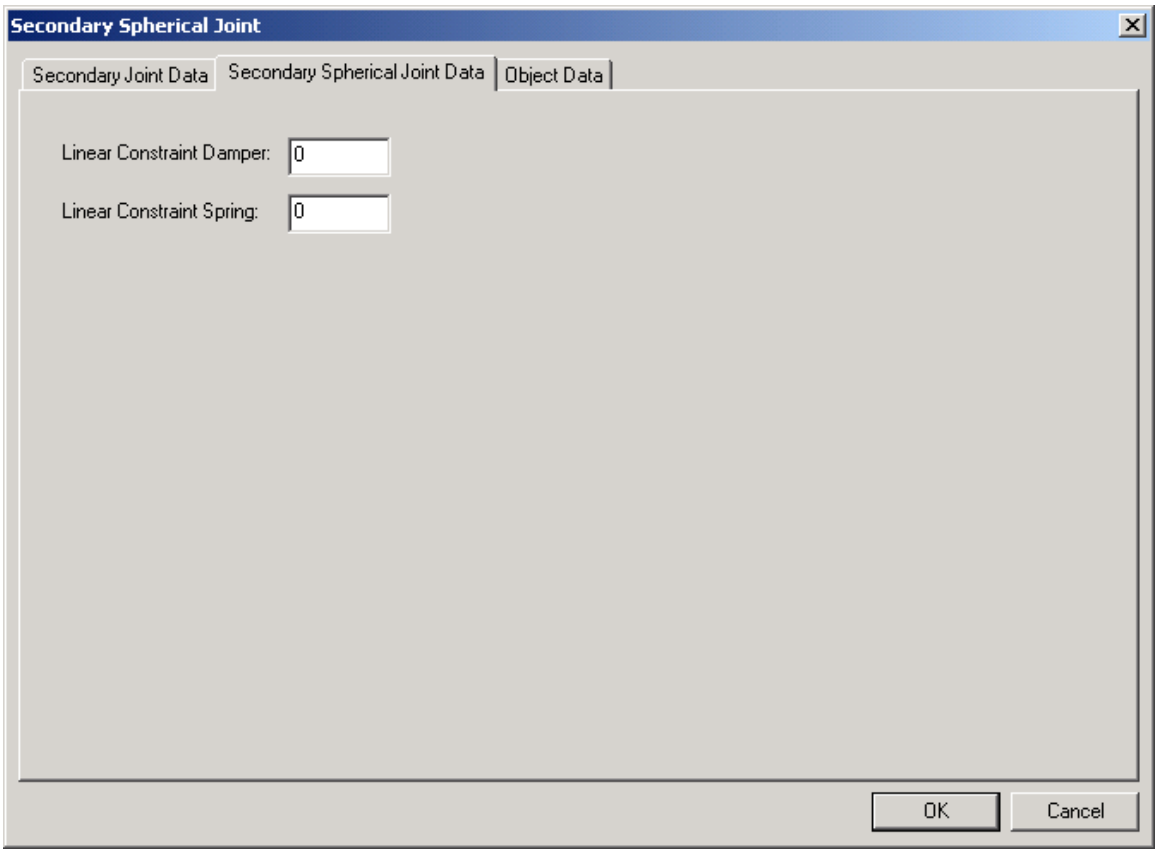
linear and angular springs and dampers which implement the joint constraints. For hard joints with spring/damper stabilization, the terms are the springs and dampers which implement the constraint stabilization. For hard joints with Baumgarte stabilization, the terms are interpreted as the Baumgarte coefficients.




### 6.2.2.13 Secondary Spherical Joint Data Property Page

Enables the user to view and edit secondary spherical joint parameters. The property page sets the parameters for implementing the constraints. The meaning and use of the parameters depends on the type of secondary joint. For soft joints, the terms are the linear spring and damper which implement the joint constraints. For hard joints with

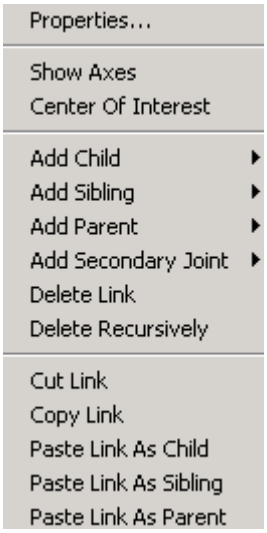
spring/damper stabilization, the terms are the spring and damper which implement the constraint stabilization. For hard joints with Baumgarte stabilization, the terms are interpreted as the Baumgarte coefficients.




## 6.3 Link Tree View Context Menu

The context menu in the Link Tree View provides access to many functions to manipulate the links and their relationships. The context menu is accessed by selecting an item and right clicking it.

### 6.3.1 Context Menu for the Articulation or a Link




If the selected item is an articulation or a link-joint pair, the context menu to the right is used.

**Properties...**

Opens the property sheet for the selected item.

*Note: For more information about the property sheets for various items, see Section 6.2*

**Show Axes**

Toggles the visibility of the selected item's coordinate axes.

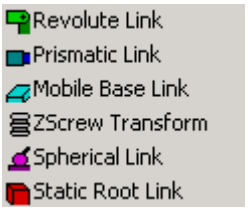
*Note: All of the coordinate axes can be toggled on or off from the toolbar. See Section 5.7 for more information.*

**Center of Interest**

Sets the selected item as the center of interest.

*Note: For more information about camera control, see Section 4.6.*

**Add Child**




Opens the Link Submenu (seen on the right) to let the user select the link-joint pair to add as a child to the selected link. This action will make the current item the parent of the new link, and all of the

currently selected item's children will become children of the new link.

*Note: The static root link is only available as a child of the articulation. See Section 6.2.1.7 for more information.*

**Add Sibling**

Opens the Link Submenu to enable the user to select the link-joint pair to add as a sibling to the currently selected item. This will result in the new link having the same parent as the selected link, and the new link will initially have no children.

*Note: The static root link is only available as a child of the articulation. See Section 6.2.1.7 for more information.*

*Note: This option is not available on the context menu for the articulation item.*

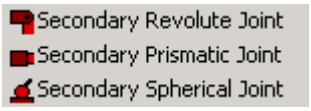
**Add Parent**

Opens the Link Submenu to enable the user to select the link-joint pair to add as the parent of the currently selected item. This will result in the new link having the same parent as the selected link, and the new link will initially have the currently selected link and its descendents as the new links descendents.

*Note: The static root link is only available as a child of the articulation. See Section 6.2.1.7 for more information.*

*Note: This option is not available on the context menu for the articulation item.*

**Add Secondary Joint**




Opens the Secondary Joint Submenu to enable the user to select the secondary joint to add to the system. The secondary joint's links will be initialized with the currently selected link. It is the user's responsibility to change one of the links to the correct link by changing the properties of the new secondary joint. The new secondary joint will be added to the secondary joint list beneath the articulation in the Link Tree View.

*Note: As soon as a secondary joint is added to the system, the articulation item is automatically changed to a closed articulation (signified by the preceding icon). If all of the secondary joints are deleted, it will revert back to an articulation.*

*Note: Both of the links that are joined by the secondary joint can be changed if desired.*

*Note: This option is not available on the context menu for the articulation item.*

**Delete Link**

Deletes the currently selected link. All children of the deleted link will be made children of the deleted link's parent. Deleting a link that has one or more secondary joints will also delete the secondary joints.

*Note: This option is not available on the context menu for the articulation item.*

**Delete Recursively**

Deletes the currently selected link and all of its descendents. Deleting a link that has one or more secondary joints will also delete the secondary joints.

*Note: This option is not available on the context menu for the articulation item.*

**Cut Link**

Removes the currently selected link and places it in the buffer. All of the children of the selected link will become children of the selected link's parent.

*Note: This option is not available on the context menu for the articulation item.*

*Note: This option is not available if the selected item is a static root link*

**Copy Link**

Places a copy of the currently selected link in the buffer. The relationships of the selected link remain unaffected.

*Note: This option is not available on the context menu for the articulation item.*

*Note: This option is not available if the selected item is a static root link*

**Paste as Child**

Adds the link in the buffer to the articulation. This will result in the selected item becoming the parent of the pasted link, and all of the currently selected item's children will become children of the new link.

*Note: This is only enabled when there is a link in the buffer.*

**Paste as Sibling**

Adds the link in the buffer to the articulation. This will result in the pasted link having the same parent as the selected link, and the pasted link will initially have no children.

*Note: This is only enabled when there is a link in the buffer.*

*Note: This option is not available on the context menu for the articulation item.*

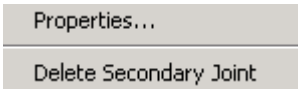
**Paste as Parent**

Adds the link in the buffer to the articulation. This will result in the pasted link having the same parent as the selected link, and the pasted link will initially have the currently selected link and its descendents as the pasted link's only descendents.

*Note: This is only enabled when there is a link in the buffer.*

*Note: This option is not available on the context menu for the articulation item.*

## 6.3.2 Context Menu for Secondary Joints




The context menu for secondary joints is on the right. The secondary joint list only exists if the user has added secondary joints to the system to create a closed articulation.

**Properties...**

Opens the property sheet for the selected item.

*Note: For more information about the property sheets for various items, see Section 6.2*

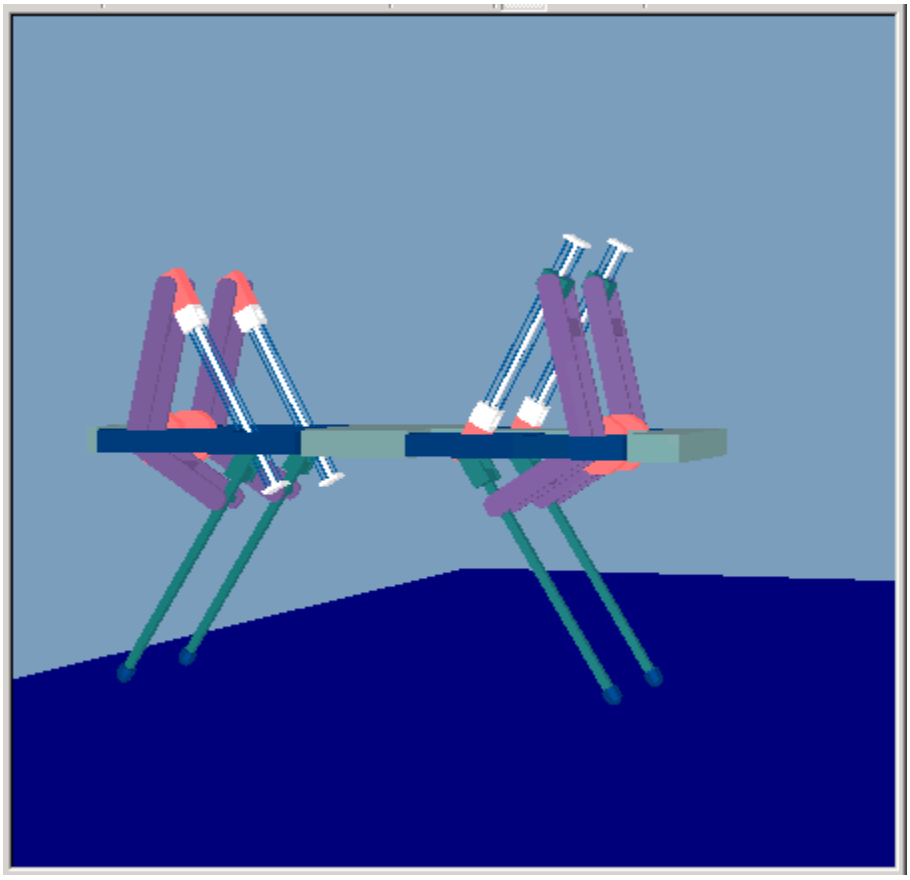
**Delete Secondary Joint**

Removes the selected secondary joint from the secondary joint list. If the deleted joint was the last secondary joint, the closed articulation will automatically change to an articulation (not closed), and the secondary joint list will disappear.

*Note: If there is no secondary joint list, it will reappear as soon as a secondary is added to the system.*

## 6.4 3D View

The 3D View displays a 3D representation of the current graphical model.



The colors that result in the 3D View are due to the material properties and the lighting used. By default, there is a directed light that points where the camera is facing. This light is a white light at 30% intensity. In addition, the scene has an ambient white light at 95% intensity that illuminates all faces evenly.

*Note: For information about navigation in the window, see Section 4.6.*

*Note: For information about toggling the visibility of the inertial and link axes, see Section 5.7.*

## 6.5 Steps to Create a New Simulation

Several steps are taken to create a new simulation from the initial state of RobotBuilder. First the simulation properties should be set in the Simulation Properties dialog box accessed from the "CFG File" menu. This defines general simulation parameters such as the integration method to use and the background color of the scene. While not strictly required, the .cfg file should be saved now so that RobotBuilder will know where the project files will be located. This will prevent warning dialogs from being created when other components of the simulation are specified. Next, the environment should be created by either opening an existing file or creating a new one by setting the parameters in the Environment Data dialog box. From that dialog box, the terrain or treadmill data should be set and saved to a file too. After the environment is saved, the articulation to simulate must be opened, or created. The parameters for the links should be specified from choosing "Properties" on the context menu and editing the property pages with the correct values. After the articulation file is saved, the control DLL should be specified by opening it from the menu bar. Finally, the .cfg file should be saved again so that locations of the component files are properly saved in the configuration.

# 7. Simulate Mode

Once the articulation, environment, and configuration have been completely defined in the `Build` mode, RobotBuilder can be used to simulate the system.

## 7.1 Control DLL

The control DLL provides a way for the user to develop arbitrary control to interface the articulation created in RobotBuilder. The control DLL will calculate forces and torques to apply to the link-joint pairs in the system. It also may interface a control GUI to display information, and obtain information from the user.

The entry points to the DLL must be well defined in order for RobotBuilder to be able to load it. The entry points required by RobotBuilder are `InitializeDMControl`, `ComputeDMControl`, `UninitializeDMControl`, and `HandleKeyPress`. The DM in the names is an abbreviation for *DynaMechs*. The exact prototypes can been seen by looking at the example code. The `InitializeDMControl` function is used to perform initialization work. Normally the code will obtain pointers to the link-joint pairs that the DLL will control and also initialize the control GUI. The `ComputeDMControl` function is called every control step by RobotBuilder. The function should contain calls to *DynaMechs* to apply the desired forces and torques to control the system. `UninitializeDMControl` is called when the simulation is stopped to deallocate any heap memory. The control GUI's `CleanUp` function should also be called. The `HandleKeyPress` function will be

called by RobotBuilder when it detects key presses. This provides another method that control can interact with the user.

*DynaMechs* is a very extensive package. Fortunately, the user will only have to use a few of the functions in the control DLL. Below is a list of functions that might be useful in a control DLL and an explanation of how to use them. Many of the functions use a parameter called `Float`. This abstraction allows *DynaMechs* to be built with single precision or double precision floating point values, depending on how `Float` is defined. While the current version of RobotBuilder uses single precision, the control source code should use the `Float` datatype to be compatible with future versions that might use double precision.

Function: `dmuFindObject`
Class: none (global in `dmu.dll`)
File: `dmu.h`
Declaration:

```
dmObject *dmuFindObject(const string &name, dmArticulation
*system)
```

Comments:

Used to get a pointer to the link named *name* in the articulation *system*. Often, this function is utilized during control initialization to obtain pointers to the link-joint pairs that will be controlled by the control DLL.

Function: `getContactState`
Class: `dmContactModel`
File: `dmContactModel.hpp`
Declaration:

```
bool getContactState(unsigned int index) const
```

Comments:

This function returns true if the *index* contact point is in contact, otherwise false. The order for the contact points can be seen in RobotBuilder's `Rigid Body Property Page`.

Function: `getExternalForce`
Class: `dmRigidBody`
File: `dmRigidBody.hpp`
Declaration:

```
void getExternalForce(SpatialVector force)
```
Comments:
This function returns any external force applied to a rigid body by the user. The output is a vector with three moments and three forces.

Function: `getForce`
Class: `dmRigidBody`
File: `dmRigidBody.hpp`
Declaration:
```
dmForce *getForce(unsigned int index) const
```
Comments:
This function returns a pointer to a force object to be applied to the link. If the link has a contact model defined, the return value for the 0 index will be a pointer to the `dmContactModel` which can be used for contact detection.

Function: `getFreeState`
Class: `dmSecondaryJoint`
File: `dmSecondaryJoint.hpp`
Declaration:
```
virtual void getFreeState(Float q[], Float qd[]) const = 0
```
Comments:
This function is used to obtain the current state of the secondary joint. The parameters are two arrays that have a number of elements equal to the degrees of freedom of the secondary joint. This is the analog of `getState` for links.

Function: `getNumDOFs`
Class: `dmLink`
File: `dmLink.hpp`
Declaration:
```
virtual int getNumDOFs() const = 0
```
Comments:
Returns the number of degrees of freedom for the link.

Function: `getNumDOFs`
Class: `dmSecondaryJoint`
File: `dmSecondaryJoint.hpp`
Declaration:
```
virtual int getNumDOFs() const = 0
```
Comments:
Returns the number of degrees of freedom for the secondary joint.

Function: `getState`
Class: `dmSystem`
File: `dmSystem.hpp`
Declaration:

```
virtual void getState(Float q[], Float qd[]) const = 0
```
Comments:
This function is used to obtain the current state and state derivatives. The parameters are two arrays that have a number of elements equal to the degrees of freedom of the link.

Function: `setExternalForce`
Class: `dmRigidBody`
File: `dmRigidBody.hpp`
Declaration:
```
void setExternalForce(SpatialVector force)
```
Comments:
This function adds an external force to a rigid body. The input is a vector with three moments and three forces.

Function: `setJointInput`
Class: `dmLink`
File: `dmLink.hpp`
Declaration:
```
virtual void setJointInput(Float joint_input[]) = 0
```
Comments:
Use this function to change the natural dynamics at a joint. The input array will have one element for each degree of freedom. The meaning depends on the joint type. For example, with a prismatic link, this would apply a force. For a revolute link, this would apply a torque. If the link has an actuator, this would apply an input to the actuator (a voltage for a DC motor).

Function: `setJointInput`
Class: `dmSecondaryPrismaticJoint, dmSecondaryRevoluteJoint, dmSecondarySphericalJoint`
File: `dmSecondaryPrismaticJoint.hpp, dmSecondaryRevoluteJoint.hpp, dmSecondarySphericalJoint.hpp`
Declaration:
```
void setJointInput(Float joint_input[])
```
Comments:
Use this function to change the natural dynamics at a secondary joint. The input array will have one element for each degree of freedom.

Function: `setState`
Class: `dmSystem`
File: `dmSystem.hpp`
Declaration:
```
virtual void setState(Float q[], Float qd[]) const = 0
```
Comments:

This function sets the state of the object. This can be used when integrating with the Placement Integrator to place the links in the desired location. The array has an element for each degree of freedom.
In some circumstances, this can be used to override the dynamics of the system. In order to do this, set the state as desired, and then call the integrator's `synchronizeState` function so the integrator's internal data structures are updated correctly.
This function is also used to set the treadmill velocity.

Function: `synchronizeState`
Class: `dmIntegrator`
File: `dmIntegrator.hpp`
Declaration:
```
void synchronizeState()
```
Comments:
Updates the integrator's internal data structures with the current state of the system. This must be called if the natural dynamics are overridden by calling `setState`.

## 7.2 Control GUI

The control GUI provides a convenient way for the control code to interact with the user. It supports several GUI components that can display the current state of variables, or accept input from the user.

The components in the GUI are completely configurable by the control source code developer. They are allocated and initialized in the control initialization function. The compute control function then interacts with the control GUI during each control step, either updating components that output information, or checking for input. Finally, when the control code is uninitializing, the components of the control GUI are deallocated by calling `CleanUp`.

The control GUI exists in the left splitter window when RobotBuilder is in the `Simulate` mode (replacing the Link Tree View). The available window is a fixed size at

1000 x 1000 pixels. The origin is at the top left corner of the window, and the **y** value increases going down the window. The **x** value increases as it moves to the right. The viewable amount of the window is set by the vertical splitter window bar, and initially, will be the same size as the Link Tree View window. The user can place the controls as desired in the window. There is nothing to prevent controls from being placed on each other, so some care and trial-and-error time must be taken to create an organized control GUI.

Below is a discussion of the useful functions applicable to the control GUI. They are defined in the `RBUserIOData.h` file. This file contains other functions that are used by RobotBuilder that should not be used in the control DLL. The name `RBUserIOData`, is a more specific name that was originally used during development for the implementation of the control GUI data container.

**Button**

The button enables the control to respond to a button click by the user.


Function:
```
bool SetNumberOfButtons (int nNumber)
```
Parameters:
*nNumber*
Number of buttons on the control GUI.
Return Value:
`true` if successful. Call `GetLastError` to get more information about the error if it fails.
Remarks:
Call first during control initialization to allocate the number of buttons to be used in the control GUI.

Function:
```
bool SetupButton (int nIndex, int nPosX, int nPosY, int
nWidth, int nHeight, string strLabel)
```

Parameters:

*nIndex*

Index of the button that is being set up. The index is in the range of 0 to one less than the number of buttons specified by `SetNumberOfButtons`.

*nPosX*

**x** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.

*nPosY*

**y** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.

*nWidth*

Integer defining width in pixels.

*nHeight*

Integer defining height in pixels.

*strLabel*

String label for the button.

Return Value:

`true` if successful. Call `GetLastError` to get more information about the error if it fails.

Remarks:

Call during control initialization for each button to set up for use in the control GUI.

Function:

```
bool GetButtonClicked (int nIndex, bool &rbClicked)
```

Parameters:

*nIndex*

Index of button whose data is being set. The index is in the range of 0 to one less than the number of buttons specified by `SetNumberOfButtons`.

*rbClicked*

A reference to `bool` that indicates if the button has been clicked since the last time this function was called.

Return Value:

`true` if successful. Call `GetLastError` to get more information about the error if it fails.

Remarks:

Call to determine if the button has been clicked since the last time the function was called. Normally, this should be checked every control step so the button response is quick.

**Check Box**

The check box lets the user toggle a state. The control can interpret the 

value to enable or disable a particular function.

Function:
`bool SetNumberOfCheckBoxes (int nNumber)`
Parameters:
*nNumber*
Number of check boxes on the control GUI.
Return Value:
`true` if successful. Call `GetLastError` to get more information about the error if it fails.
Remarks:
Call first during control initialization to allocate the number of check boxes to be used in the control GUI.

Function:
`bool SetupCheckBox (int nIndex, int nPosX, int nPosY, bool bInitialValue, string strLabel)`
Parameters:
*nIndex*
Index of the check box that is being set up. The index is in the range of 0 to one less than the number of check boxes specified by `SetNumberOfCheckBoxes`.
*nPosX*
**x** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.
*nPosY*
**y** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.
*bInitialValue*
The initial value of the check box. Set to `false` to initialize with the check box unchecked.
*strLabel*
String label for the check box – located to the right of the box.
Return Value:
`true` if successful. Call `GetLastError` to get more information about the error if it fails.
Remarks:
Call during control initialization for each check box to set up for use in the control GUI.

Function:
`bool GetCheckBoxValue (int nIndex, bool &rbValue)`
Parameters:
*nIndex*
Index of check box whose data is being set. The index is in the range of 0 to one less than the number of check boxes specified by `SetNumberOfCheckBoxes`.

*rbValue*
A reference to `bool` that indicates if the check box is currently checked.
Return Value:
`true` if successful. Call `GetLastError` to get more information about the error if it fails.
Remarks:
Call to determine if the check box is currently checked or not. Normally, this should be checked every control step so the response is quick.

**Edit**

The edit box enables the user to enter arbitrary values 

for the control to interpret. The data is stored in a string, but it can easily be converted to a number.

Function:
```
bool SetNumberOfEdits (int nNumber)
```
Parameters:
*nNumber*
Number of edit boxes on the control GUI.
Return Value:
`true` if successful. Call `GetLastError` to get more information about the error if it fails.
Remarks:
Call first during control initialization to allocate the number of edit boxes to be used in the control GUI.

Function:
```
bool SetupEdit (int nIndex, int nPosX, int nPosY, int
nWidth, string strInitialData, string strLabel)
```
Parameters:
*nIndex*
Index of the edit box that is being set up. The index is in the range of 0 to one less than the number of edit boxes specified by `SetNumberOfEdits`.
*nPosX*
**x** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.
*nPosY*
**y** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.
*nWidth*
Integer defining width in pixels.

*nHeight*
Integer defining height in pixels.
*strInitialData*
String that holds the initial data for the edit box.
*strLabel*
String label for the edit box. The label will be to the left of the edit box.
Return Value:
`true` if successful. Call `GetLastError` to get more information about the error if it fails.
Remarks:
Call during control initialization for each edit box to set up for use in the control GUI.

Function:
```
bool GetEditValue (int nIndex, string &rstrValue)
```
Parameters:
*nIndex*
Index of edit box whose data is being set. The index is in the range of 0 to one less than the number of edit boxes specified by `SetNumberOfEdits`.
*rstrValue*
A reference to a string that holds the data currently in the edit box.
Return Value:
`true` if successful. Call `GetLastError` to get more information about the error if it fails.
Remarks:
Call to retrieve the data currently in the edit box. Normally called every control step.

**Label**

The label enables the user to display data directly on the 

control GUI window. This is useful for always displaying the current value of a state (the velocity for example) or for displaying a static text string to clarify other data in the control GUI.

Function:
```
bool SetNumberOfLabels (int nNumber)
```
Parameters:
*nNumber*
Number of labels on the control GUI.
Return Value:
`true` if successful. Call `GetLastError` to get more information about the error if it fails.
Remarks:

Call first during control initialization to allocate the number of labels to be used in the control GUI.

Function:

```
bool SetupLabel (int nIndex, int nPosX, int nPosY, int
nWidthInChars, string strInitialValue)
```

Parameters:

*nIndex*

Index of the label that is being set up. The index is in the range of 0 to one less than the number of labels specified by `SetNumberOfLabels`.

*nPosX*

**x** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.

*nPosY*

**y** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.

*nWidthInChars*

Integer number of characters that is the maximum number of characters to be used in the label.

*strInitialValue*

String that is used as the initial value for the label.

Return Value:

`true` if successful. Call `GetLastError` to get more information about the error if it fails.

Remarks:

Call during control initialization for each label to set up for use in the control GUI.

Function:

```
bool SetLabelValue (int nIndex, string strValue)
```

Parameters:

*nIndex*

Index of label whose data is being set. The index is in the range of 0 to one less than the number of labels specified by `SetNumberOfLabels`.

*strValue*

String to display in the label.

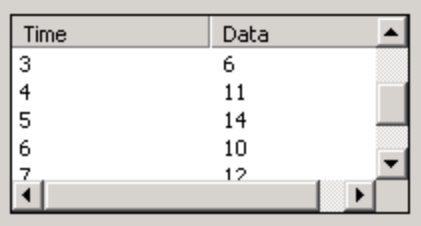
Return Value:

`true` if successful. Call `GetLastError` to get more information about the error if it fails.

Remarks:

Call to set the value of the label. Normally, this is called in the compute control function to update the value held in the label.

**Output List**




Implements a window that can display output data. The window can possess multiple resizable columns and is useful for displaying the current state to the user. If the data exceeds the display limits of the output list, scrollbars will be automatically be added to scroll both horizontally and vertically. This component is useful for viewing how data changes over time.

Function:
```
bool SetNumberOfOutputLists (int nNumber)
```
Parameters:

*nNumber*

Number of output lists on the control GUI.

Return Value:

`true` if successful. Call `GetLastError` to get more information about the error if it fails.

Remarks:

Call first during control initialization to allocate the number of output lists to be used in the control GUI.

Function:
```
bool SetupOutputList (int nIndex, int nPosX, int nPosY,
int nWidth, int nHeight, string strLabels)
```
Parameters:

*nIndex*

Index of the output list that is being set up. The index is in the range of 0 to one less than the number of output lists specified by `SetNumberOfOutputLists`.

*nPosX*

**x** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.

*nPosY*

**y** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.

*nWidth*

Integer defining width in pixels.

*nHeight*

Integer defining height in pixels.

*strLabels*

String that is used contains the labels for the columns. Each column label is separated by a tab. This implicitly defines the number of columns in the output list, because one column will be created for each label.

Return Value:

`true` if successful. Call `GetLastError` to get more information about the error if it fails.

Remarks:

Call during control initialization for each output list to set up for use in the control GUI.

Function:

```
bool SetOutputListData (int nIndex, string strData)
```

Parameters:

*nIndex*

Index of output list whose data is being set. The index is in the range of 0 to one less than the number of output lists specified by `SetNumberOfOutputLists`.

*strData*

String of data to append to the output list. The data for each column is separated by a tab character.

Return Value:

`true` if successful. Call `GetLastError` to get more information about the error if it fails.

Remarks:

Call to append a new line of data to the output list. The call should be made in the compute control function, but it might be desired to only output periodically (rather than every control step). Otherwise the output list will fill very quickly.

**Slider**

Implements a slider control for user input. When the user clicks and holds the mouse on the slider pointer, the current value is displayed above it. The slider only supports integer values, but the source code can scale the value to obtain floating point values if desired.


Function:

```
bool SetNumberOfSliders (int nNumber)
```

Parameters:

*nNumber*

Number of sliders on the control GUI.

Return Value:

`true` if successful. Call `GetLastError` to get more information about the error if it fails.

Remarks:

Call first during control initialization to allocate the number of sliders to be used in the control GUI.

Function:

```
bool SetupSlider (int nIndex, int nPosX, int nPosY, int
nWidth, int nHeight, int nMinValue, int nMaxValue, int
nInitialValue, string strLabel)
```

Parameters:

*nIndex*

Index of slider that is being set up. The index is in the range of 0 to one less than the number of sliders specified by `SetNumberOfSliders`.

*nPosX*

**x** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.

*nPosY*

**y** position in the control GUI in pixels. Number should be between 0 and 1000 inclusive.

*nWidth*

Integer defining width in pixels.

*nHeight*

Integer defining height in pixels.

*nMinValue*

Integer defining the lowest number on the slider scale.

*nMaxValue*

Integer defining the highest number on the slider scale.

*nInitialValue*

Integer defining the initial value for the slider.

*strLabel*

String that is used as the label which is centered and placed above the slider.

Return Value:

`true` if successful. Call `GetLastError` to get more information about the error if it fails.

Remarks:

Call during control initialization for each slider to set up for use in the control GUI.

Function:

```
bool GetSliderValue (int nIndex, int &rnValue)
```

Parameters:

*nIndex*

Index of slider whose data is being retrieved. The index is in the range of 0 to one less than the number of sliders specified by `SetNumberOfSliders`.

*rnValue*

A reference to an integer which will contain the current value of the slider.

Return Value:

`true` if successful. Call `GetLastError` to get more information about the error if it fails.

Remarks:

Call during each control step to find the current value of the slider.

Three other functions are important for the control GUI:

Function:

```
void CleanUp ()
```

Remarks:

This function should be called when the control is uninitializing to deallocate the control GUI components.

Function:

```
string GetProjectDirectory () const
```

Return Value:

String that contains the path to the current project directory (the directory where the `.cfg` file currently open in RobotBuilder exists). Note that if no configuration file is open, the return string will be empty.

Remarks:

Use this function to find the directory where the currently open `.cfg` file is located. This is useful if an output file needs to be generated from the control (because the current directory will change depending on the what the user is doing in RobotBuilder).

Function:

```
int GetLastError () const
```

Return Value:

Integer indicating the last error.

NO_RBUSERIODATA_ERROR – no errors

ERROR_INVALID_POSX - the position should be between 0 and 1000 inclusive

ERROR_INVALID_POSY - the position should be between 0 and 1000 inclusive

ERROR_INVALID_WIDTH - the width should be between 0 and 1000 inclusive

ERROR_INVALID_HEIGHT - the height should be between 0 and 1000 inclusive

ERROR_INVALID_CONTROL_INDEX - the specified index is not valid. It must be from 0 to the number of controls specified

ERROR_INVALID_NUMBER_OF_CONTROLS - the number of controls (slider, edit, etc.) must be between 0 to 1000 inclusive

ERROR_SETUP_LOCKED - the setup was attempted to be changed after it was locked by RobotBuilder. The setup can only be changed during control initialization.

Remarks:

This function will give more information about an error that occurred. This function should only be called immediately after receiving a `false` return value from a function call to a `CRBUserIOData` function.

## 7.3 Advanced Control Development Topics

RobotBuilder can be used in conjunction with *Microsoft Visual Studio's* debugging capabilities to aid in the development of the control DLL. (Other compilers have similar capabilities, but this discussion will use *Visual Studio*.)

*Visual Studio* has many advanced features that the developer can use to debug software. It has the capability to place break points in the code, see the current value of data during execution, and view the call stack. For more information on the capabilities and how to utilize them, please see the Microsoft documentation.

A couple of steps need to be taken to use the debugging features. First specify RobotBuilder as the "Executable for the debug session" on the "Debug" tab in the `Project Settings` property sheet. Next, set the "Working Directory" to point to the directory where RobotBuilder is located. Make sure the active configuration is the debug version and set breakpoints where desired. Finally, press the "Go" ('F5') button. This will start RobotBuilder. Open up your project, and begin simulating. When the breakpoint is hit, *Visual Studio* will be brought to the front of the screen, and the debugging tools will be available.

# 8. Playback Mode

The Playback mode enables the user to review a previously created simulation. It provides the capability to archive the animated results so that qualitative comparisons between different simulations can be easily made. It also enables the user to see the results in approximately real time.

The Playback mode is accessed by pressing the "Playback" button on the toolbar (See Section 5.2). Upon entering the mode, the current articulation will still be shown in the Link Tree View, but the window will be disabled (grayed out). The 3D View shows the results of a previous simulation by animating the articulation. The camera control performs identically in the Playback mode as it does in the Build and Simulate modes. While the playback is playing, the simulation time can be seen in the status bar.

To create playback data, the user needs to press the "Record" button in the Simulate mode. (See Section 5.3 for information about the "Record" button.) If the "Record" button is pressed before the "Play" button is pressed, the playback data will include the initial position of the robotic articulation. The "Record" button can be toggled off to stop recording the data for playback purposes. Anytime the "Record" button is toggle on, any existing playback data will be cleared. This implies that the user cannot concatenate pieces of the simulation in one playback file.

Playback data can be saved persistently by using the "Playback" menu on the menu bar (see Section 4.5 for more information about the "Playback" menu). The menu also enables the user to open previously recorded simulation data for playback.

The playback data that is saved is only the position and orientation of movable items in the simulation. RobotBuilder obtains the other information, like the robotic articulation and environment, from the currently loaded project. Consequently, to see a playback, the project that created the playback data has to be loaded in RobotBuilder.

RobotBuilder attempts to perform some validation that the currently loaded project was used to create the playback data. First, the "Playback" button on the toolbar is only enabled when there is playback data loaded that has the same number of movable items as the current configuration. This implies that if playback data is created for a robotic articulation, and then another link-joint pair is added, the "Playback" button will not be enabled because the number of movable items in the current robotic articulation does not match the number in the playback data.

The playback data file has some additional fields that record the articulation file, environment file, and each file's system time stamp when the playback data is created. When the user attempts to play a playback data file, those fields are checked against the currently loaded articulation and environment. If a difference is detected, a warning message is presented to the user. It is then up to the user to decide to proceed or not. Playing a playback file from another articulation than the currently loaded one will not cause any problems, but the resultant playback will not make sense because the current articulation will be moved in the manner of the recorded articulation. Note that only a warning message is presented because the playback data could still be valid for the current articulation. For example, if the playback data was generated before the articulation was saved to a file, and the playback is entered after the articulation is saved,

the playback is valid for the articulation, but the warning will be generated because there will be a mismatch in the articulation file names. This implementation was chosen to provide more flexibility for the user.

# 9. General Issues

## 9.1 Units

The units of data values used in RobotBuilder are left up to the user. As long as all of the data values have consistent units, *DynaMechs* will be able to accurately simulate the system. The exception is time. For display purposes, and real-time functionality, time needs to be measured in seconds in RobotBuilder.

## 9.2 Application Timeout


Due to licensing restrictions with the graphics library used in RobotBuilder (Sense 8's *WorldToolKit*)., the application will periodically (usually after a year) stop working. It will present the following error message to indicate the license has expired. At that point, just download a new version of RobotBuilder (http://eewww.eng.ohio-state.edu/~orin/RobotBuilder/RobotBuilder.html).

## 9.3 Bugs

RobotBuilder is still under development. While it is believed that the application is useful and functional, it has not been extensively tested in different environments. Please send a bug report to robotbuilderbugs@yahoo.com if a problem is encountered.

When submitting a bug report, please give as much information as possible. Please send the application version number, your OS version, your video card and driver version, the problem that is being experienced, and the steps to reproduce that problem.

# 10. Conclusion

Hopefully, RobotBuilder will be helpful in your research. If you discover any bugs, please send an email to robotbuilderbugs@yahoo.com.

# BIBLIOGRAPHY

[1] Scott McMillan, David E. Orin, and Robert B. McGhee, "Object-Oriented Design of a Dynamic Simulation for Underwater Robotic Vehicles," *IEEE International Conference on Robotics and Automation*, pp. 1886-1893, 1995.

[2] Duane Marhefka, *Fuzzy Control and Dynamic Simulation of a Quadruped Galloping Machine*, PhD dissertation, The Ohio State University, Columbus, Ohio, 2000.

[3] *WorldToolKit Reference Manual Release 9*, software reference manual, Engineering Animation, Inc., April 1999.

[4] John J. Craig, *Introduction to Robotics: Mechanics and Control, 2nd Edition*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[5] Duane W. Marhefka and David E. Orin, "Dynamic Simulation of Running Machines Containing Kinematic Loops," submitted to *IEEE Transactions on Robotics and Automation,* 2001.