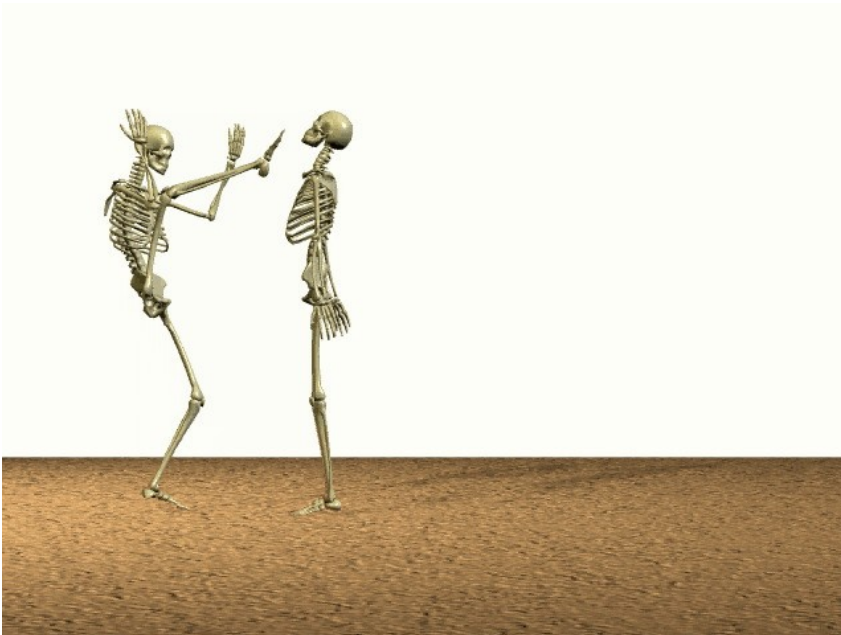


Ari Shapiro  
[shapiroari@yahoo.com](mailto:shapiroari@yahoo.com)  
<http://www.arishapiro.com>

Updated 4/11/09

## I. DANCE MANUAL



### Help

For help, please refer to the DANCE website: <http://www.arishapiro.com/dance/>

To subscribe to the DANCE development email list, send an email to:

[dancedev-subscribe@yahoogroups.com](mailto:dancedev-subscribe@yahoogroups.com)

To post a message to this email list, subscribe first, then send an email to:

[dancedev@yahoogroups.com](mailto:dancedev@yahoogroups.com)

If you have any specific issues you would like to discuss about DANCE, you can email Ari Shapiro directly at:

[shapiroari@yahoo.com](mailto:shapiroari@yahoo.com)

<http://www.arishapiro.com>

For a technical overview of the DANCE platform, please see our research paper at:

[http://www.arishapiro.com/shapiroa\\_DANCE.pdf](http://www.arishapiro.com/shapiroa_DANCE.pdf)

and for our paper on controllers:

[http://www.arishapiro.com/Sandbox07\\_DynamicToolkit.pdf](http://www.arishapiro.com/Sandbox07_DynamicToolkit.pdf)

## 1. Overview

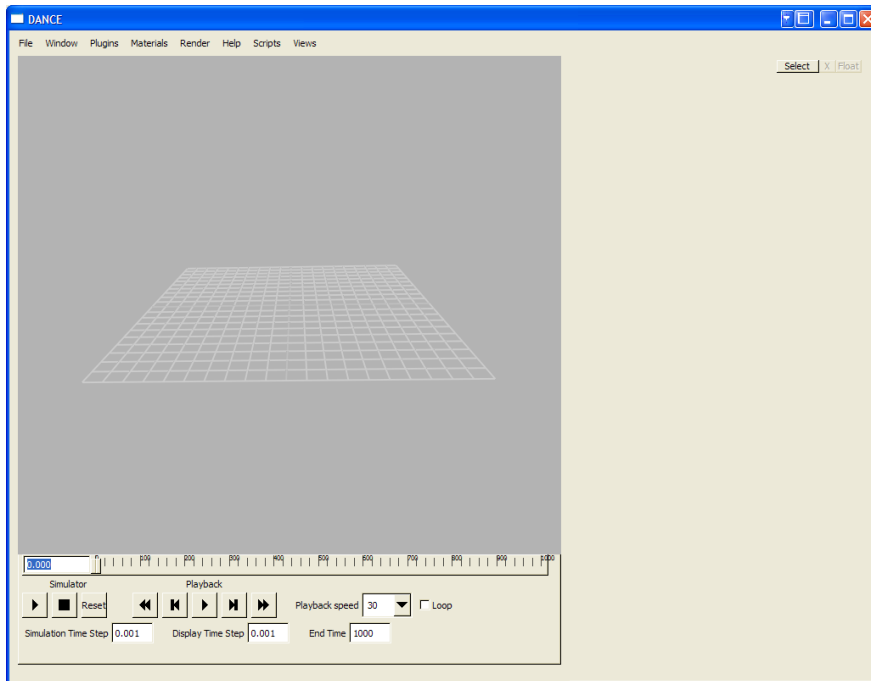
DANCE is a software environment for graphics and animation with an emphasis on rigid body dynamic and character controller development. DANCE allows you to create ragdoll simulations, rigid body collisions, and control characters under physical simulation. DANCE contains the following features:

- Ragdoll simulation
- Physical simulation of characters with dynamic control
- Pose-based dynamic control
- Physically-based controller scripting language via Python
- Import/Export of motion capture (BVH, ASF/AMC)
- Loading/saving of standard 3-D model object types.
- Skinning algorithm to create deformable meshes driven by skeletons/bones.
- Creation of videos from OpenGL or high quality output via POVray  
<http://www.povray.org>
- Integrated with the Open Dynamics Engine (ODE) <http://www.ode.org>
- Multi-platform (Windows, linux, OSX)

The DANCE environment operates using a plugin architecture. Users can create and manipulate plugins that perform various actions in the DANCE environment. For example:

- The Cube plugin will create a 3-dimensional cube that can be manipulated and animated.
- The ODESim plugin will create a rigid body simulator that uses the Open Dynamics Engine ([www.ode.org](http://www.ode.org)).
- The MotionPlayer plugin will allow the user to load, save and manipulate motion capture data.

Each plugin typically has a visual representation that is shown in the DANCE window, a set of GUI widget that allows the user to interactive control various parameters, and a set of commands that can be accessed via the Python scripting language. DANCE comes with a set of plugins that include shapes, physical simulation, motion capture display and so forth. Each of the plugins can be modified if needed. In addition, custom plugins can be built in order to expand functionality as well.



*Initial DANCE screen. The window on the left displays objects in the DANCE environment. The empty panel on the right holds the GUI interface for each object. The timeline on the bottom shows the simulation time. The controls on the lower left control the simulators. The controls in the middle control the playback for simulated objects.*

### 1.1.1 Supported Platforms

DANCE currently runs on both the Windows and Linux platforms and has been adapted to work on OSX. The source code for DANCE is available and may be used to port to other platforms as well. Windows code is compiled using Microsoft Studio .NET (v8.0/2005, v9.0/2008) and using g++ on Linux. The DANCE code is cross-platform compatible and all libraries used by DANCE (FLTK, ImageMagick, ODE) are cross-platform libraries.

### 1.1.2 License

#### Noncommercial License

The DANCE software is distributed for noncommercial use in the hope that it will be useful but WITHOUT ANY WARRANTY. The author(s) do not accept responsibility to anyone for the consequence of using it or for whether it serves any particular purpose or works at all. No warranty is made about the software or its performance.

Any plugin code written for DANCE belongs to the developer of that plugin, who is free to license that code in any manner desired.

Content and code development by third parties (such as FLTK, Python, ImageMagick, ODE) may be governed by different licenses.

You may modify and distribute this software for noncommercial use

as long as you give credit to the original authors by including the following text in every file that is distributed:

```
/******  
  Copyright 2005 by Ari Shapiro and Petros Faloutsos  
  DANCE  
  Dynamic ANimation and Control Environment  
  -----  
  AUTHOR:  
    Ari Shapiro (ashapiro@cs.ucla.edu)  
  ORIGINAL AUTHORS:  
    Victor Ng (victorng@dgp.toronto.edu)  
    Petros Faloutsos (pfal@cs.ucla.edu)  
  CONTRIBUTORS:  
    Yong Cao (abingcao@cs.ucla.edu)  
    Paco Abad (fjabad@dsic.upv.es)  
*****/
```

## Commercial Use

DANCE may be used commercially only for internal purposes under the following conditions:

- Any commercial users of DANCE must notify Ari Shapiro by email at [shapiroari@yahoo.com](mailto:shapiroari@yahoo.com) as to:
  - o The purpose of the use of DANCE, in whole or in part.
- The authors of DANCE reserve the right to advertise that the software is being used commercially by such commercial entity.
- DANCE may not be sold in whole or in part, and can only be used for internal purposes or development of software used internally and not for sale.
- An example of valid commercial use is:
  - o Internal development of applications to generate images or videos – this is allowed by the commercial license indicated here.
  - o Extraction of algorithms or techniques from DANCE for use as a plugin into another software program that is used for internal purposes – this is allowed by the commercial license indicated here.
- An example of invalid use is:

- DANCE being sold in whole or in part – this is NOT allowed by the commercial license indicated here.
- Packaging DANCE independently or extracting algorithms from DANCE and selling them as part of a commercial entity's software products – this is NOT allowed by the commercial license indicated here.

If you are interested in using DANCE in whole or in part for commercial uses not covered by the license, please contact Ari Shapiro at shapiroari@yahoo.com.

### 1.3 DANCE\_DIR Environment Variable

Before running DANCE, it is important to set a system variable called:

DANCE\_DIR

to the directory that contains the main DANCE distribution. Users on all platforms (including Windows) should use forward slashes ('/') and not backslashes ('\'). For example, if the DANCE distribution is located in c:\programs\dance\_v4, the DANCE\_DIR variable should be:

c:/programs/dance\_v4.

## 1.4 Compilation

### 1.4.1 Windows Compilation

DANCE will compile on the Windows platform using Microsoft Studio v8.0 (Visual Studio 2005) and Microsoft Visual Studio 2008.

1) Unzip the DANCE distribution. The DANCE distribution will be `dance_v4_MMddyy_hhmm.zip` where `MMddyy_hhmm` is the month, day, year, hour and minute of the build time.

2) Create the environmental variable: `DANCE_DIR` and make this variable indicate the location of the `dance_v4/` directory. Use forward slashes (/) and not backslashes(\) to separate subdirectories. For example, if your `dance_v4/` directory is located in

`c:\my\subdirectory\dance_v4`, your `DANCE_DIR` variable should be:

`c:/my/subdirectory/dance_v4`

3) Download the 8-bit DLL version of image magick.

<http://www.imagemagick.org/download/binaries/ImageMagick-6.4.5-3-Q8-windows-dll.exe>

(Note that the latest version of the ImageMagick libraries may change. Please choose the latest 8-bit version at <http://www.imagemagick.org/script/binary-releases.php#windows>). Install the executable. During the installation process, make sure that you check 'Install development headers and libraries for C and C++'. The paths to the image magick directory will need to be added to your Visual Studio directories in step 6 below.

4) Download the Python DLL:

<http://www.python.org/ftp/python/2.5.2/python-2.5.2.msi>

Install the executable. Note that the latest version of the Python libraries may change. DANCE is compatible with Python 2.3, Python 2.4 and Python 2.5.

5) Perform the following additional steps in order to link the proper runtime library with both FLTK, ODE and lib3ds:

1. Recompile FLTK dlls:

- Open %DANCE\_DIR%/extensions/fltk-2.0.x-r4773/ide/vcnet/fltk.sln
- Build the fltkdll project in Debug and Release modes.
- Build the fltkdll\_images project in Debug and Release modes.
- The compiled .DLLs will be automatically placed in the \$DANCE\_DIR/bin directory, and the .lib files will be automatically placed in the \$DANCE\_DIR/lib/win directory.
- Note that more recent versions of FLTK ~may~ be substituted for the 4773 version included in the DANCE distribution. However, the more recent versions of FLTK on Windows have much worse performance than the 4773 versions. If you choose to use a more recent version of FLTK, make sure that the .lib files are placed in \$DANCE\_DIR/lib/win and the .DLLs are placed in the \$DANCE\_DIR/bin directories.

2. Recompile ODE dlls:

- Open %DANCE\_DIR%/extensions/ode-0.10.1/build/vs2005/ode.sln
- Build the ode project in debugdoubledll and releasedoubledll modes.

- The compiled .DLLs will be automatically placed in the \$DANCE\_DIR/bin directory, and the .lib files will be automatically placed in the \$DANCE\_DIR/lib/win directory.

### 3. Recompile lib3ds libs:

- Open %DANCE\_DIR%/extensions/lib3ds-1.3.0/msvc8/lib3ds.sln
- Build the lib3ds project in debug and release modes.
- The compiled .DLLs will be automatically placed in the \$DANCE\_DIR/bin directory, and the .lib files will be automatically placed in the \$DANCE\_DIR/lib/win directory.

6) For Visual Studio 2005, open the %DANCE\_DIR%/build/vs2005/dance\_v4\_2005.sln solution in Microsoft's Visual Studio.

For Visual Studio 2008, open the %DANCE\_DIR%/build/vs2008/dance\_v4\_2005.sln solution in Microsoft's Visual Studio and allow Visual Studio to automatically convert the project and solution files automatically.

6) Make sure the proper Python and ImageMagick paths have been properly added to the Visual Studio include and library directories:

Go to: **Tools -> Options**

Choose: **Projects** (left hand side)

Select: **VC++ Directories**

Under "**Show Directories For**" (upper right), select "**Include Files**"

Make sure that the ImageMagick include directory is correct. For example: c:\program files\ImageMagick-6.4.5-Q8\include.

Make sure that the Python include directory. For example: c:\Python25\include.



Under “Show Directories For” (upper right), select “**Library Files**”

Make sure the ImageMagick lib directory is correct. For example: c:\program files\ImageMagic-6.4.5-Q8\lib

Make sure the Python lib directory is correct. For example: c:\Python25\libs

7) Build the entire solution (right click on ‘Solution ‘dance\_v4’/Build Solution). Note that there are two building configurations: Debug and Release. The Debug configuration will build DANCE with debug symbols and will run slower than the Release version. The Debug version of DANCE creates an executable named

```
%DANCE_DIR%/bin/dance_d.exe
```

The Release version of DANCE creates an executable named

```
%DANCE_DIR%/bin/dance.exe
```

### **1.3.2 Linux Compilation**

The Linux version of DANCE requires the g++ compiler as well as the following libraries:

- Python 2.5 (Python 2.3 and 2.4 are compatible)

- ImageMagick (with development libraries)

- Mesa (OpenGL)

1) Unzip the DANCE distribution. The DANCE distribution will be `dance_v4_MMddy_hhmm.zip` where `MMddy_hhmm` is the month, day, year, hour and minute of the build time.

2) Create the environmental variable: `DANCE_DIR` and make this variable indicate the location of the `dance_v4/` directory. For example, if your `dance_v4/` directory is located in

`/home/mydir/dance_v4`, your `DANCE_DIR` variable should be:

```
/home/mydir/dance_v4:
```

Using `bash`, `sh` or `ksh`:

```
export DANCE_DIR=/home/mydir/dance_v4 or
```

Using csh:

```
setenv DANCE_DIR /home/mydir/dance_v4
```

3) Install the latest version of FLTK (you will need to have the X11 development and Mesa/OpenGL development installed already):

- Download from <http://ftp.easysw.com/pub/ftk/snapshots/ftk-2.0.x-r6140.tar.gz>
- Unzip to the \$DANCE\_DIR/extensions directory:
  - cd \$DANCE\_DIR/extensions
  - unzip ftk-2.0.x-r6140.tar.gz
  - Build the FLTK libraries into the DANCE\_DIR with shared library support:
    - ./configure --prefix=\$DANCE\_DIR --enable-shared
    - make
    - make install
  - Note that more recent versions of FLTK may be used from [www.ftk.org](http://www.ftk.org).

4) Build and install the version of ODE in this DANCE distribution:

- cd \$DANCE\_DIR/extensions/ode-0.10.1
- chmod +x configure
- ./configure --prefix=\$DANCE\_DIR --enable-double-precision --enable-shared
- make
- make install
- Note that if you use a different version of ODE than is included in this distribution, make the following changes:
  - In file ode-10.1/src/

5) If Python 2.5 is not installed, install it on the system. It can also be installed locally via:

- Download the Python source from <http://www.python.org/ftp/python/2.5.2/Python-2.5.2.tgz> and place it in the \$DANCE\_DIR/extensions directory.
- Uncompress the file:
  - `tar -xvzf Python-2.5.2.tgz`
- Run the configure script, using the DANCE\_DIR as the installation directory:
  - `cd Python-2.5.2`
  - `./configure --prefix=$DANCE_DIR --enable-shared`
- Make python:
  - `make`
- Install the binary and libraries in the \$DANCE\_DIR
  - `make install`

6) If ImageMagick development libraries are not installed, install it on the system. It can also be installed locally via:

- Download the ImageMagick source from <http://voxel.dl.sourceforge.net/sourceforge/imagemagick/ImageMagick-6.3.9-10.tar.gz> (or more recent version) and place it in the \$DANCE\_DIR/extensions directory.
- Uncompress the file:
  - `tar -xvzf ImageMagick.tar.gz`
- Run the configure script, using the DANCE\_DIR as the installation directory and using 8-bit textures:
  - `cd ImageMagick-6.4.2`
  - `./configure --prefix=$(DANCE_DIR) --enable-shared --without-perl --with-quantum-depth=8`
- Make ImageMagick:

- make
- Install the binary and libraries in the `$(DANCE_DIR)`
  - make install

7) Install lib3ds locally via:

- Download the lib3ds source from
- <http://www.lib3ds.org/> and place it in the `$(DANCE_DIR)/extensions` directory.
- Uncompress the file:
  - Unzip lib3ds.1.3.0.zip
- Run the configure script, using the `DANCE_DIR` as the installation directory:
  - `cd lib3ds-1.3.0`
- `./configure --prefix=$(DANCE_DIR)`
- Make lib3ds:
  - make
- Install the binary and libraries in the `$(DANCE_DIR)`
  - make install

8) Change the location of directories in `$(DANCE_DIR)/Makefile.inc` if necessary. By default, the directories are set up to run properly if the above libraries (FLTK, ImageMagick, Python) are installed as specified above.

9) Modify the include and library directories (if necessary) in the `$(DANCE_DIR)/Makefile.inc`.

10) Build DANCE in release mode with:

```
make
```

or build DANCE in debug mode with:

```
make debug
```

The DANCE binary is located in `$DANCE_DIR/bin/dance` (or `$DANCE_DIR/bin/dance_d` for the debug version). DANCE is built using dynamic libraries, so the following needs to be added to the `LD_LIBRARY_PATH` variable:

Using `bash`, `sh` or `ksh`:

```
export LD_LIBRARY_PATH=$DANCE_DIR/lib
```

Using `csh`:

```
setenv LD_LIBRARY_PATH $DANCE_DIR/lib
```

then run DANCE with:

```
$DANCE_DIR/bin/dance
```

### 1.3.3 Compilation on OSX

1) Follow the instructions for a linux build in order to build FLTK, ODE, Python, ImageMagick and NumPy.

2) Copy `Makefile-osx.inc` to `Makefile.inc`. Modify the locations of includes and libraries if necessary.

To compile DANCE on non-Windows, non-Linux platforms (OSX, for example), you can modify the `Makefile.inc` used for linux. By default, the `Makefile.inc` includes setup for compilation on OSX.

3) Build DANCE in release mode with:

```
make
```

or build DANCE in debug mode with:

```
make debug
```

The DANCE binary is located in `$DANCE_DIR/bin/dance` (or `$DANCE_DIR/bin/dance_d` for the debug version). DANCE is built using dynamic libraries, so the following needs to be added to the `LD_LIBRARY_PATH` variable:

Using `bash`, `sh` or `ksh`:

```
export LD_LIBRARY_PATH=$DANCE_DIR/lib
```

Using `csh`:

```
setenv LD_LIBRARY_PATH $DANCE_DIR/lib
```

then run DANCE with:

```
$DANCE_DIR/bin/dance
```

## 1.4 Directories

The following is a listing and brief description of the DANCE directories:

Directory	Purpose
/	Main dance directory.  Holds profile.py file which runs initial commands on execution of dance.exe.
bin/	Binary and .dll directory
build/	Build scripts for each platform
controllers/	Controller plugin source code
core/src	Core DANCE source code
core/math	Math routines source code
data/	Materials, motions and models
docs/	Documentation
extensions/	Supplementary libraries (FLTK, ODE, etc)
geometries/	Geometry plugin source code
lib/	Libraries
modifiers/	Modifier plugin source code
plugins/	Plugins (.dll in Windows, .so in linux)
renderers/	Renderer plugin source code
run/	User work area
scripts/	DANCE scripts that will appear on the GUI interface
sdfastobjs/	Sd/fast files
simulators/	Simulator plugin source code

system/	System plugin source code
tmp/	Area used to store temporary files when running DANCE

## 2. Getting Started

A number of example scripts that demonstrate the functionality of DANCE are loaded into the application. To access these examples, choose a script from the Scripts/tutorials menu. These scripts demonstrate ragdoll simulation, controller simulation, object collision, motion capture data access and basic object creation. You can examine these examples by looking at the files in the DANCE\_DIR/scripts/tutorials directory. Each script contains a set of Python commands that can be used to program DANCE.

### Tutorial 1 : Skeleton Ragdoll example

Simulates a skeleton as a ragdoll.

### Tutorial 1a : Skeleton Ragdoll example with attachments

Simulates a skeleton as a ragdoll while fixing one of the bones in space. Attachments are used to fix a body to a point in space or on another moving body.

### Tutorial 2 : Skeleton With Pose Control

Skeleton example using poses and PD control. Poses can be set and achieved interactively during physical simulation.

### Tutorial 3 : Skeleton With Pose Control and Collisions

Pose control demonstrating collisions with other objects.

### Tutorial 4 : Many Skeleton Ragdolls

Multiple skeleton ragdolls in an interactive environment.

### Tutorial 5 : Motion Capture

The motion capture plugin is loaded allowing you to load any .bvh or asf/amc mocap file.

### Tutorial 6 : Test Controller

Example of using controllers with the skeleton. The code for the controller is located in the TestController project. The controller is not effective at keeping the skeleton balanced but is included as an example of how to program a controller using C++ in DANCE.

### Tutorial 7 : Controller Scripting



Example of using controllers in DANCE that are created using scripts written in Python. The advantage of this method is that no compilable code needs to be written, and scripts can be quickly changed and tested.

Details of using these tutorials are in the section 5. Quickstart, below.

### **2.1 Example Plugin Development**

Please consult the section 'DANCE Plugin Development' for details on how to create a plugin for DANCE. A sample plugin project is included in DANCE\_DIR/generic/Sample. This sample plugin demonstrates programming the basic capabilities in DANCE.

### **2.2 Example Controller Development**

DANCE controllers can be created using either scripts written in Python or by developing controllers directly written in C++.

- Sample controller scripts are located in the \$DANCE\_DIR/run directory that have a .py extension. In addition, a description of the controller API is given in Section 11, DANCE Controller Scripting.
- A sample skeleton controller written in C++ is given in DANCE\_DIR/controllers/TestController. This controller may be executed by running Scripts/tutorials/Tutorial 6 Skeleton Test Controller. You may modify this code (change the applyPDParams() method) to write a new controller for the skeleton.

### 3. Camera Navigation

The DANCE camera focuses on objects in the scene. The camera can be controlled using the mouse and keyboard. The following mouse and keyboard commands can be used to control the camera:

Camera Command	Action
Left mouse button + drag	Rotate camera
SHIFT + Left mouse button + drag	Rotate camera, constrain to axis
Right mouse button + drag	Zoom camera
Middle mouse button + drag	Pan camera
Spacebar	Restore camera tilt to align with the y-axis

## 4. Keyboard Commands

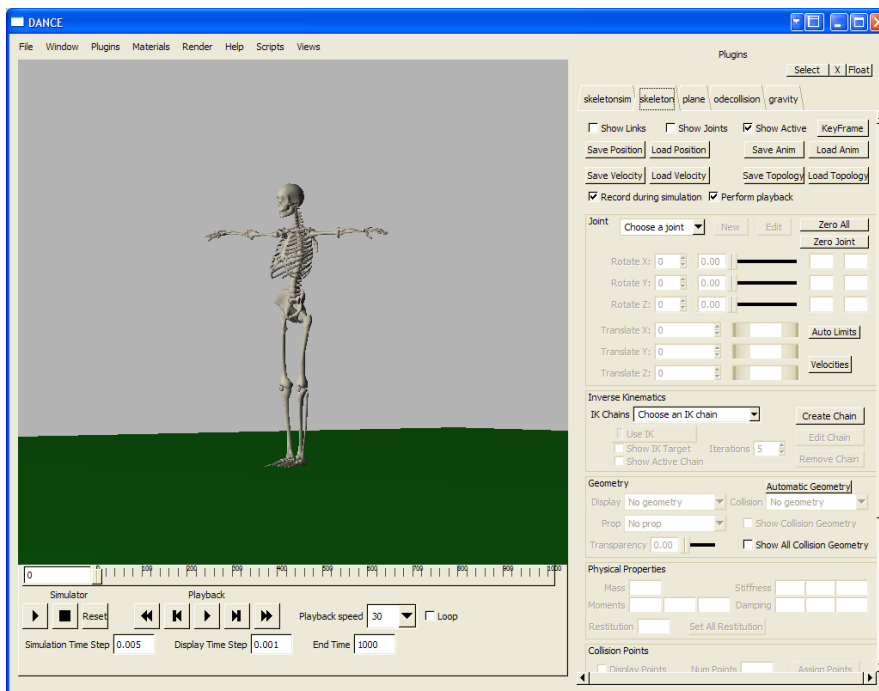
The following keyboard commands operate in the DANCE window:

Keyboard Command	Action
G	Toggle grid on/off
Spacebar	Restore camera tilt to align with the y-axis
H	Display object selection window
F	Place all objects in the scene within focus

## 5. Quickstart

If you want to start using DANCE immediately, then use the following guide and perform the tutorials as follows:

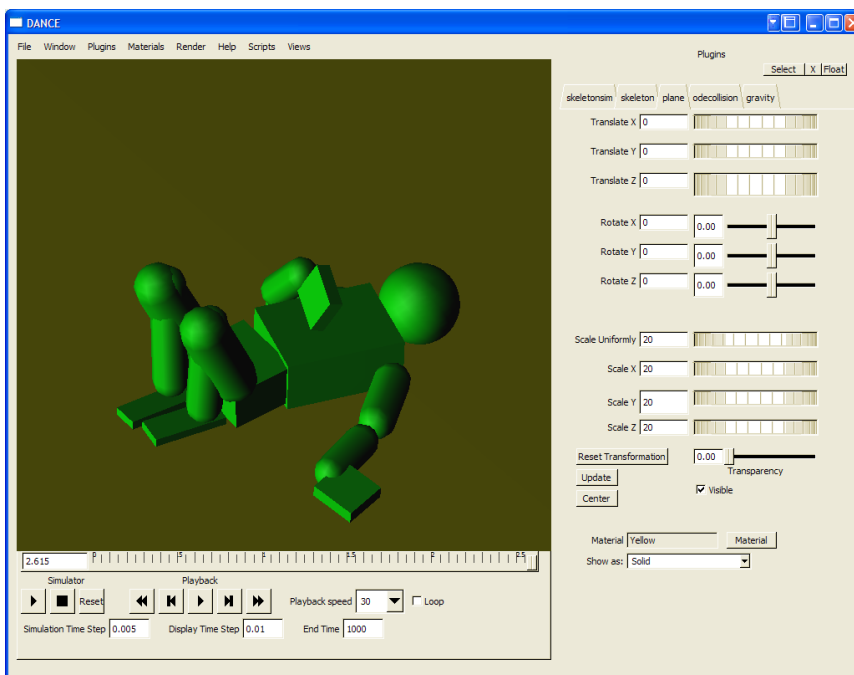
### Tutorial 1: Skeleton Ragdoll Example



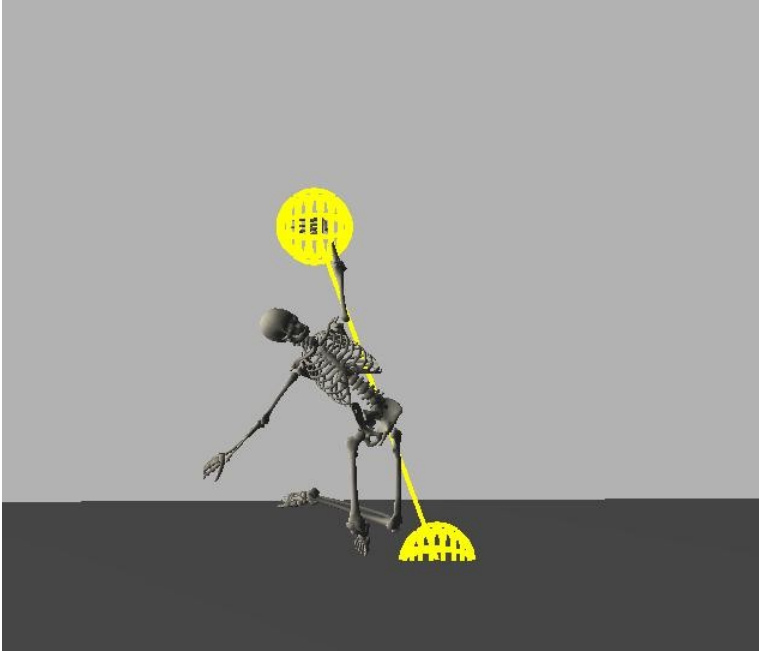
- From the menubar, start the first tutorial by choosing Scripts/Tutorials/Tutorial\_1\_Skeleton\_Ragdoll. You'll see a skeleton with its arms outstretched standing on a green ground.
- Start the simulation by pressing the simulation start button under the word "Simulator" in the lower left part of the screen. The skeleton will fall to the ground.
- If you have instability in the skeleton (the skeleton flies around unexpectedly or disappears entirely), try lowering the Simulator Time Step from .005 to .001. A slower simulation time step will yield greater stability. If your simulation looks choppy, you can also lower the Display Time Step which measures how often the screen is refreshed. A slower display time step will show you the results of the simulation more frequently, although it will also slow down the simulation slightly.

- Stop the simulation by pressing the simulation stop button that is next to the start button.
- Playback this motion by first pressing the rewind button under the word “Playback” (this is next to the set of simulator controls). After pressing rewind, the skeleton will be upright again. You can use your mouse and click on the time slider to see the animation, or you can press the play button under the word “Playback” and see the motion in real time.
- Press the simulator ‘Reset’ button to restore the character to its original, arms out posture.
- To drop the skeleton from a tall point, you can move the skeleton to any position in the world. To do this, choose a joint from the joint dropdown list on the skeleton GUI (this is on the right hand side of the screen under the tab named “skeleton”). Under “choose a joint”, select the Hip. Next, move the “Translate Y” roller to move the skeleton upwards slightly. You can also rotate the character so that it is no longer facing straight upwards using the ‘Rotate’ sliders. Pressing the simulator Play button will again start the simulation and the skeleton will fall to the ground. You can choose any set of joints from the joint dropdown list and rotate them into the proper position. Pressing ‘Zero All’ will return the character to the initial position. Pressing ‘Zero Joint’ will return only that joint to the initial position.
- You can give the skeleton’s body some initial velocity by selecting the appropriate joint and then pressing the ‘Velocities’ button. Choose the Hip, press ‘Velocities’, set Linear Velocity X to 10, and then press ‘Set and Close’. This will give the Hip body an initial velocity of 10. Press simulator Play to see the effect. To remove these velocities, press ‘Reset All Velocities’,
- You can add objects in the environment for the skeleton to collide with. Move the skeleton slight upwards by selecting the Hip joint and then Translate Y. Create a Cube by choosing Scripts/Create\_cube. A cube will appear under the skeleton. At this point, the cube does not participate in collisions. To make the skeleton collide with the cube, you must first select the tab marked ‘odecollision’. At the bottom left, there is a button called ‘Update’. Press it, and the cube will appear in the list above. Click on the checkbox next to the cube, and now the cube will collide with the skeleton. To change the size and location of the cube, select the cube GUI by choosing the tab marked ‘cube’ and then translate, rotate and scale the cube. Other types of objects can be created using the Create\_Sphere and Create\_Capsule scripts.
- To save your skeleton ragdoll experiments, select File/Save Session and choose a file name that ends with “.dpy”. To restore this session at a later time, choose File/Load Session and then select your .dpy file.

- To export the motion of your skeleton to a motion capture .bvh file, choose the skeleton tab, and then click on the Save Anim button. The motion capture can be replayed using the MotionPlayer as described in tutorial 5, or by pressing the Load Anim button on the skeleton tab and choosing that motion capture file.
- Note that the geometry of the skeleton (bones) is not the geometry used in collisions. To see the geometry that is used in collisions, check the 'Show All Collision Geometry' box in the Geometry area of the skeleton tab. This collision geometry will explain why the skeleton occasionally appears to be floating above the ground, since the bones are much thinner than the real collision geometry.

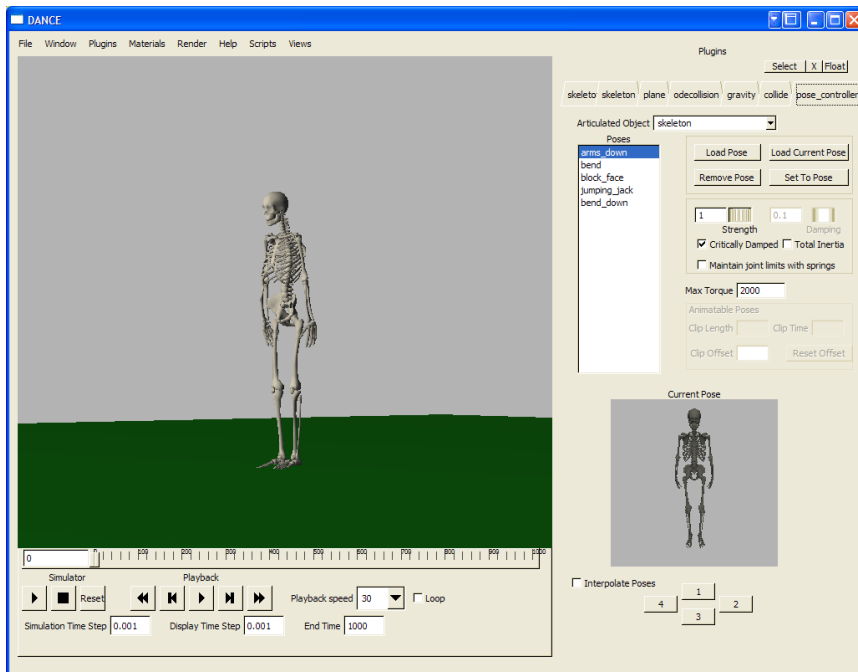


## Tutorial 1a: Skeleton Ragdoll Example With Attachments



- This tutorial is similar to Tutorial 1, except that the skeleton's hand is attached to a fixed point in space.
- On the skeleton GUI, press the button marked Attachments. Choose the bone from the list on the top. The Skeleton Attachments are attachments from that bone to another bone in the scene (which could be either on a different skeleton or on the same skeleton). The Static Attachments are attachments to fixed geometry. The bone will be attached relative to the other bone or geometry in its current position.
- To remove an attachment, press Remove Attachments or Remove All Attachments .
- Check the Show Attachments box to display the attachments.

## Tutorial 2: Skeleton With Pose Control

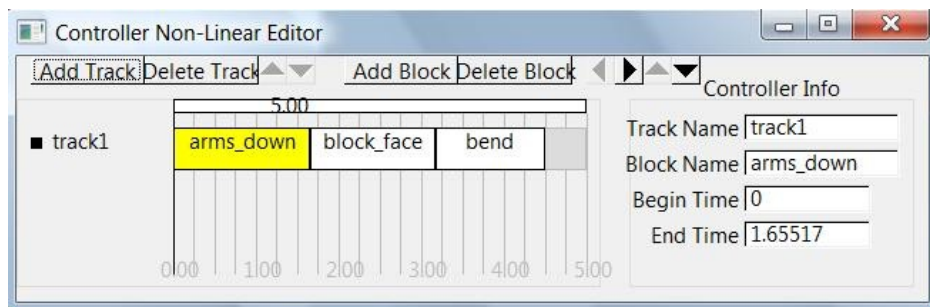


- This tutorial is similar to Tutorial 1 except that the skeleton has a pose control that can allow the skeleton to achieve a particular pose under physical simulation using PD controllers.
- The desired pose is shown on the right hand side. By clicking on the list of poses (such as arms\_down, bend, block\_face), the desired pose will change. You can place your mouse over the pose window and click the left button to rotate the view. The right mouse button will reset the view to the original facing.
- Choose the *bend* pose, then press the simulator start button. The character will achieve a bending pose as gravity takes over and sends the character off balance to the ground. While under simulation, you can change the pose to any pose listed in the pose list. The character will always attempt to achieve that pose, but collisions, gravity and balance might prevent the skeleton from achieving that particular posture.
- You can adjust the strength by which the character achieves these poses by modifying the strength slider. A good value for strength is between 0 and 1.0. Higher strength values will require a lower simulation time step and make the skeleton very stiff. The initial value of .3 is a value that represents a normal speed or strength.
- You can add poses to the pose list by positioning the character. While the simulation is stopped, choose the skeleton tab, choose the joint from the dropdown list, and move the joint to the desired position. Once that pose has been designed, choose the



pose\_controller tab, then press the Load Current Pose button and name the pose. It will now appear in the pose list and can be used during simulation.

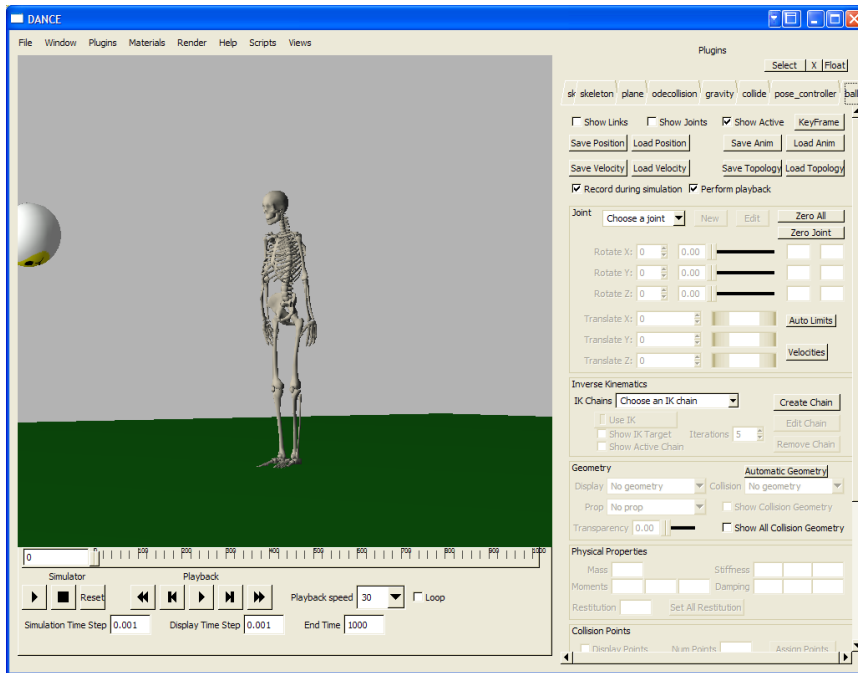
- You can create a pose that is interpolated from two poses by using the pose window. First, select a pose from the pose list, then press any one of the 1, 2, 3, 4 buttons on the pose window. These buttons indicate the pose that will be stored at each of those locations. Once at least two poses have been selected, click on the interpolate poses checkbox, and then move your mouse and click between the two yellow boxes. This will give you an interpolated pose between those two poses. Poses can only be interpolated between adjacent poses.
- Collision obstacles can be added in the same manner as from tutorial 1.
- Note that if the skeleton becomes unstable, you will need to lower the timestep slightly. The default is .001. A more stable timestep would be .0005 or .0001.
- Poses can be timed and run using the Non-Linear Editor:
  - Check the Non-Linear Editor box. This will popup a window that allows you to place and activate controllers at different times:



- Press the Add Track button to create a track.
- Click on the rectangular area to the right of track1.
- Press Add Block and the currently selected pose will appear on the non-linear editor.
- Select the edges of the block, left click the mouse and drag to move the length of time the controller will be active for.
- Left click on the middle of the block and drag to move the entire controller block.

- Controllers may be placed on multiple tracks. This is useful if one controller operates on the upper body, while another operates on the lower body. Another example if one controller maintains a pose, while another controller checks the environment for certain conditions (i.e. check to see if the skeleton is falling, and if so, react to that event). For this simple tutorial, placing controllers on multiple tracks will execute the controllers on the upper tracks first, then override them with the controllers on the lower tracks (this effectively means that only the lower track controllers will be in effect, since simple pose controllers override all movement).
- Scripted controllers can make effective use of multiple tracks and can be programmed to respond to varying environments. Scripted controllers are described in Section 11.

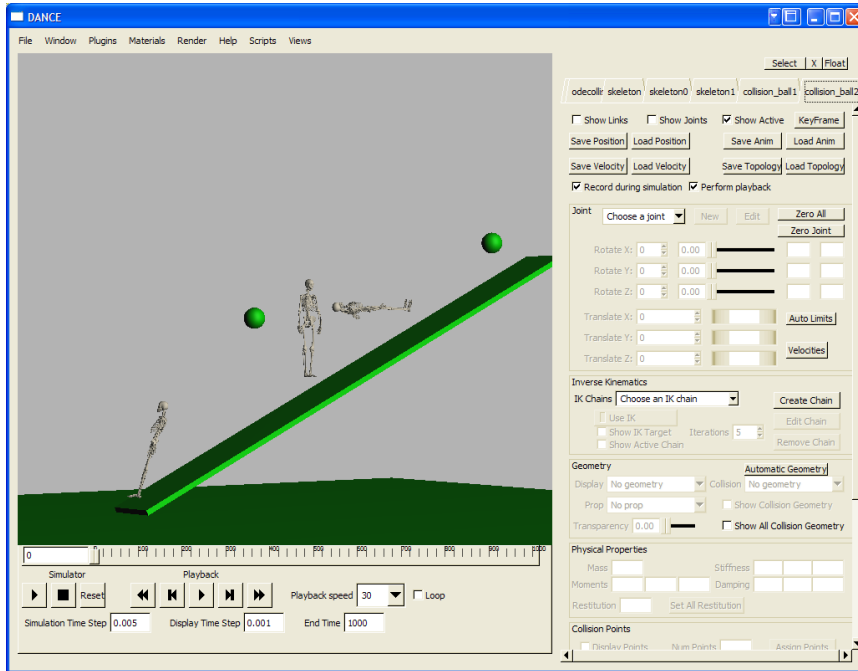
### Tutorial 3: Skeleton With Pose Control and Collisions



- This tutorial is similar to Tutorial 2 except that the environment includes other moving objects.
- Press the simulator play button and the ball will be thrown at the skeleton.
- You may use the pose\_controller to allow the skeleton to achieve different poses as the ball comes towards it.
- To change characteristics of the ball, choose the 'ball' tab, then select the root joint from the joint dropdown. The ball is also a skeleton, but a much simpler one with only one body and one joint. Go to the box marked 'Physical Properties' and change the mass from 30 to 100. This will make the ball much more massive and have a larger effect on the skeleton.
- Press the 'Velocities' button on the root joint for the ball and change the Linear Velocity Y from -1. to 1. This will give the ball a slightly upwards movement.
- To change the initial position of the ball, select the root joint from the joint list and then translate the ball to the proper position.

- You can change the bounciness of the ball by setting the Restitution parameter to a higher value (1 = bouncy, 0 = not bouncy).

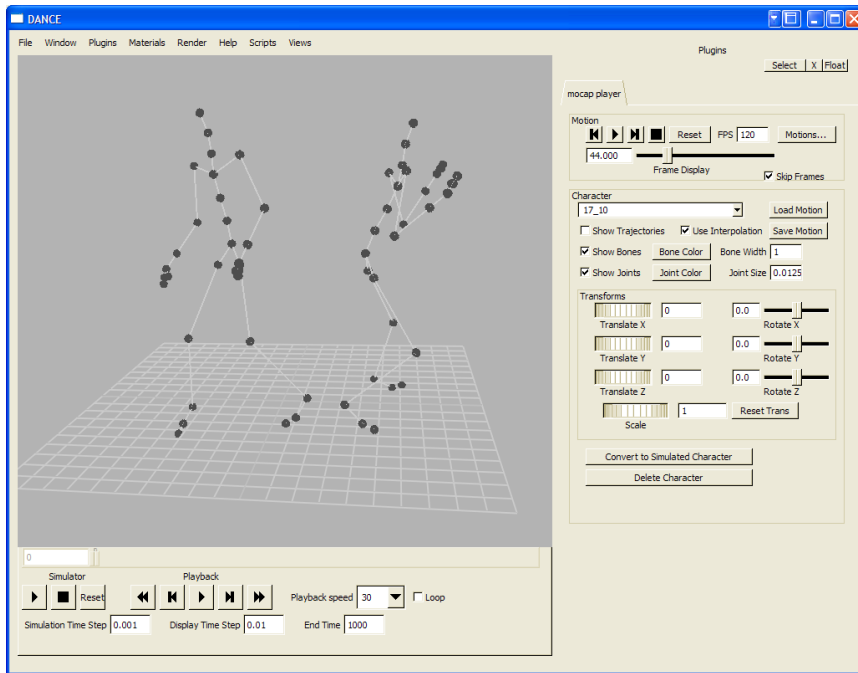
## Tutorial 4: Many Skeleton Ragdolls



- This tutorial is similar to Tutorial 1 and Tutorial 3 except that there are multiple skeletons, none of them use pose control (they are all simple ragdolls).
- To add or remove static collision objects, create the object in the environment (cube, sphere, plane, capsule), select the 'odcollision' tab, press the Update button and then select/deselect the object from the list. You can move the static objects around by clicking on their name tab, then translate, rotate and scale.
- To change the position of the collision balls, press the collision\_ball1 or collision\_ball2 tab, choose the root joint from the joint dropdown list, and then translate or rotate. Press the 'Velocities' button to set their initial velocities.
- To change the size of the collision balls, perform the following:
  - Press 'h' to call up the list of plugins.
  - Choose the 'sphere' or 'sphere0' from the drop down list and press 'Select'. The GUI for the sphere will appear on the right.

- Slide the 'Scale Uniformly' roller to change the size of the ball. Note that only uniform scaling will work properly. Do not translate, rotate or scale non-uniformly. You may change the color/material of the sphere by pressing the material button.

## Tutorial 5: Motion Capture



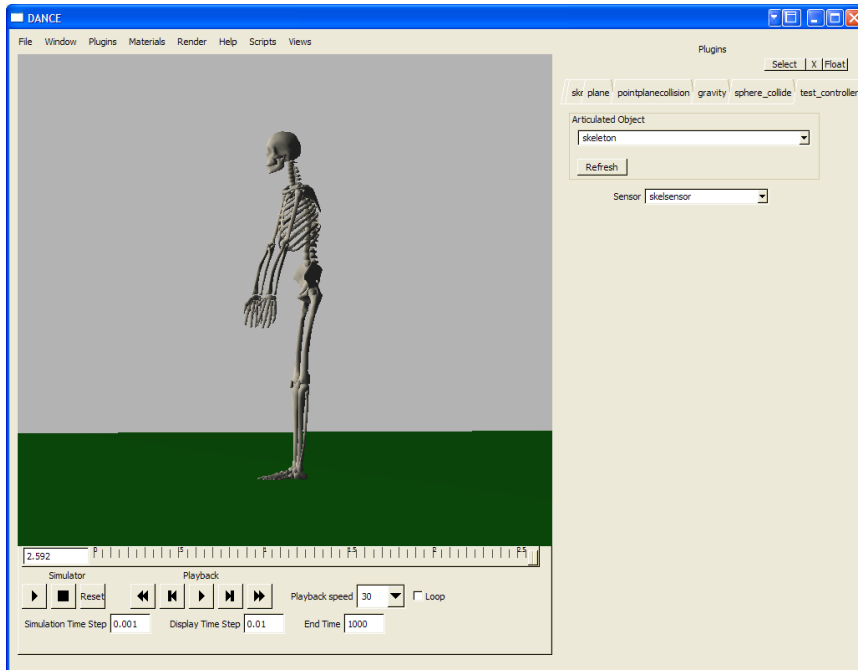
- This tutorial demonstrates how to use motion capture in DANCE.
- On the MotionPlayer GUI, press the Load Motion button, and then navigate to the dance\_v4/data/motions folder. Choose the file 'walk01\_edited.bvh'. After the file is chosen, you will be prompted for a name for the motion capture character. Keep the default name and press ok. The MotionPlayer reads both .bvh and asf/amc format files.
- Press 'f' to bring all the objects in the scene into focus. You will see a character on the left. Press the play button under the word 'Motion' at the top right.
- Check the box marked 'Show Trajectories' to see the path of the character. The yellow line will indicate the path of the root joint. You can modify the path of the character by sliding the 'Rotate Y' slider. You can change the size of the character by moving the 'Scale' roller. You can save this new motion to a .bvh file by pressing the 'Save Motion' button.
- Press the stop button under the word 'Motion'. Load another character by pressing 'Load Motion', navigating to the dance\_ve/data/motion directory and choosing 17.asf then 17\_10.amc. Press the play button again. Notice that the two characters are moving at different speeds. This is because the original .bvh motion was captured at 30 fps, while the other character uses 120 fps. To make the second character move at normal

speed, change the FPS input to 120. Select the first character from the Character dropdown list and then press the button marked Delete Character to remove the first character and keep the second.

- Additional motion can be found at the CMU motion capture web site at:  
<http://mocap.cs.cmu.edu/>



## Tutorial 6: Test Controller

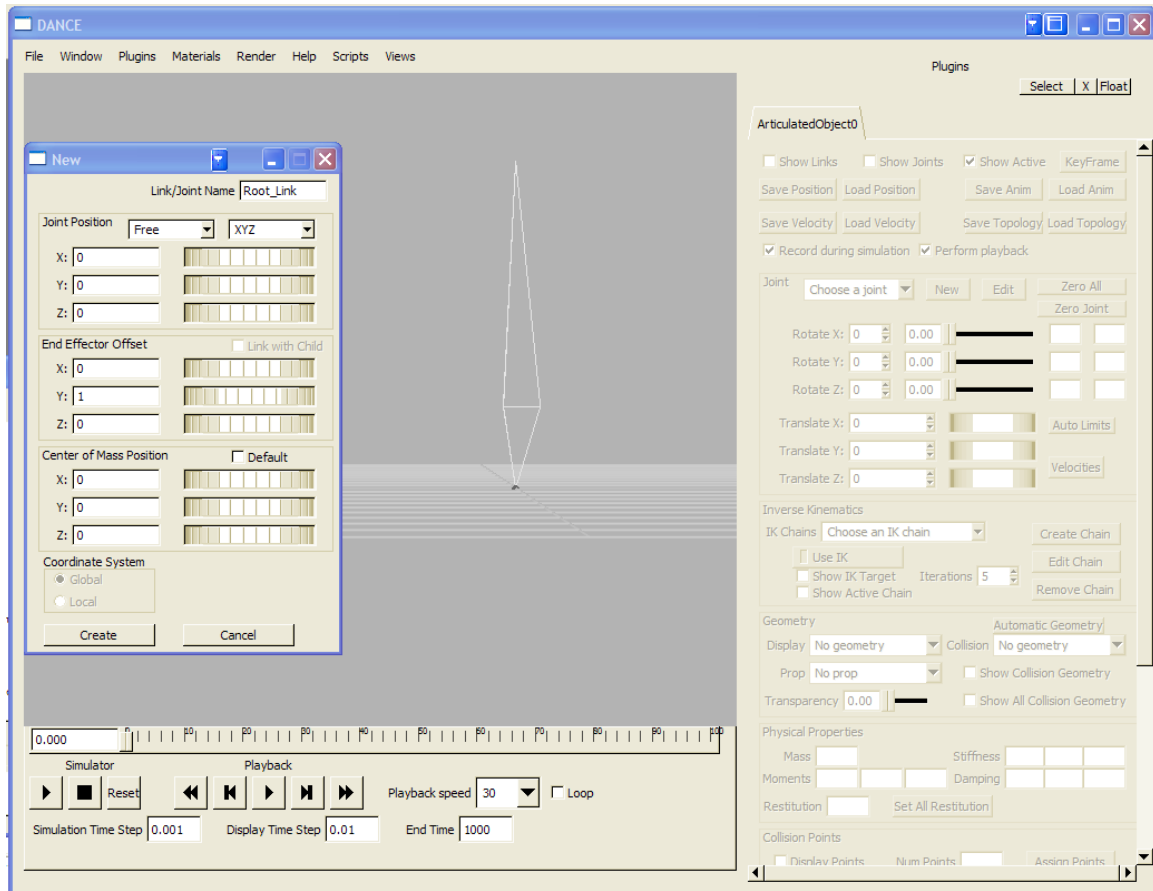


- This tutorial demonstrates the use of a simple balancing controller for use on the skeleton.
- The balancing controller code is located in the TestController project. The strategy is simple – the arms are sent in the opposite direction of the velocity of the skeleton’s center of mass. This strategy ultimately fails, since the swinging arms is insufficient to balance a character. However, the controller code is designed to be simple and easy to follow.

## Tutorial 7: Skeleton With Script Control

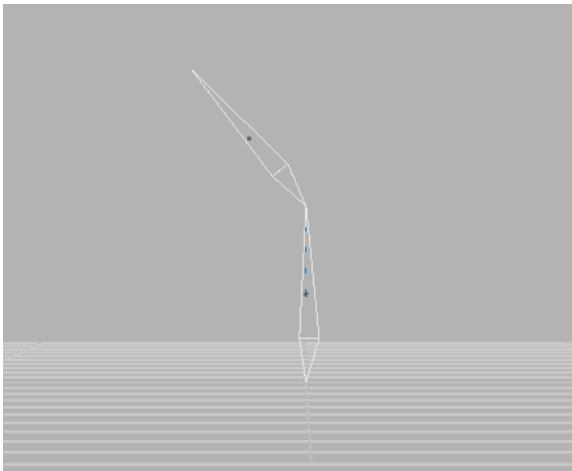
## Tutorial: Simple Pendulum

- This tutorial demonstrates the use of DANCE to build and simulate objects. A pendulum will be build in this example.
- Start a new Session, by choosing the File/Reset Session option.
- The first step is to create an object with joints and bodies. Open the plugins window by either pressing 'h' or by choosing Plugins/Create from the menubar. Choose the 'ArticulatedObject' plugin and press the 'Create Instance' button. A GUI will appear on the right side of the application.
- Under the Joint group box, press the New button. This will open the joint creation window and prompt you for the joint parameters.
- At this point, press zoom in on the joint by pressing the right mouse button in the main DANCE window and sliding the mouse upwards. Press the spacebar to align the camera with the ground plane. The middle mouse button pans the camera up and down, while the right mouse button rotates the view with an arcball. If the view becomes disorienting, press the spacebar again.

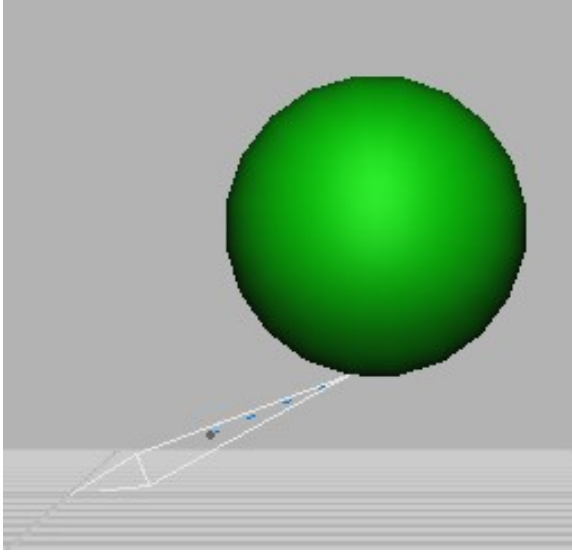


- The black sphere at the base on the joint represents the center of mass of the joint. The long tip represents the end effector. The joint is implicitly indicated at the opposite end of the end effector. In this case, it is exactly at the center of mass.
- Change the joint type from 'Free' to 'Pin'. This will change the joint from a 6-DOF joint to a 1-DOF joint. Note that the center of mass will also change to the middle of the joint. Press the create button to create the joint.
- The next step is to attach a simulator to your ArticulatedObject. Open the plugins window ('h' on the keyboard or Plugins/Create from the menubar) and select an ODEsim plugin then press Create Instance. Under the SimulatedObject, choose ArticulatedObject0. This will attach an ODE simulator to your pendulum so that it may be simulated.
- Access the ArticulatedObject GUI again by selecting the ArticulatedObject0 tab. Choose the Root\_Link from the joint dropdown, and then slide the Rotate Z slider so that the pendulum sits at 45 degrees.

- Load gravity into the DANCE environment, by choosing Scripts/Create\_gravity from the menubar.
- Press the play button under the Simulator on the lower left side of the screen. The pendulum should swing back and forth.
- To make the pendulum swing faster, you can raise the time step to .01.
- To make a multi-link pendulum, go to the ArticulatedObject0 tab. Choose the Root\_Link from the joint dropdown, then press Zero All to return the pendulum to the initial position.
- Press the 'New' button to add a new joint, then Create to make another pin joint. On the ArticulatedObject0 GUI, rotate the second joint along the Z axis so that the structure is slightly bent, then press the simulator play button again. Both links of the pendulum are now simulated.



- To attach geometry to the pendulum, first stop the simulator. Reset the pendulum to the zero position (Zero All button). Create a sphere from the plugins window (Plugins/Create, choose Sphere, then Create Instance). Scale the sphere uniformly to .5. Translate the sphere in the Y direction by 1.5. On the ArticulatedObject0 tab, choose link1 from the joint dropdown. Under the Geometry box, change the Display dropdown from 'No geometry' to 'Sphere0'. Rotate the joint to 45 degrees again and start the simulator.



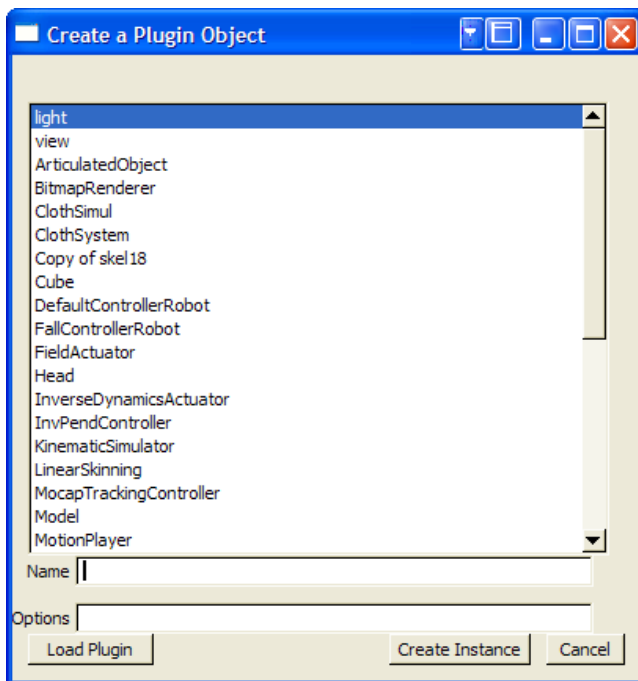
- Multiple pendulums can be created by instantiating new ArticulatedObject plugins and attaching them to new ODESim plugins. Make sure that you change both the center of mass and the end effector positions of your additional pendulums (by default, they all start at 0, 0, 0).

## 6. Plugins

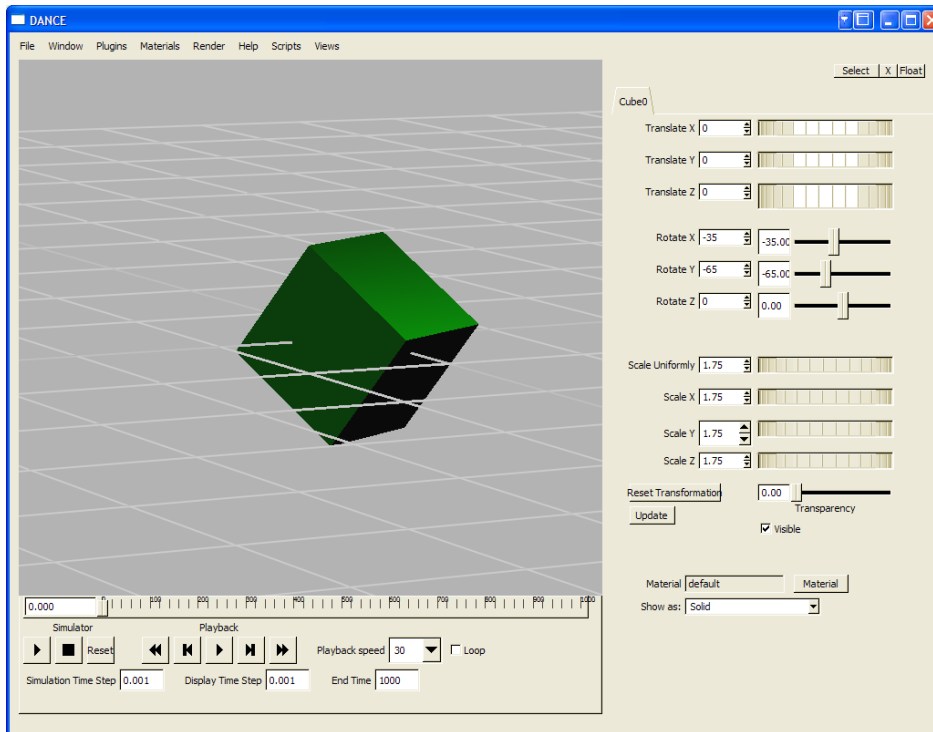
DANCE's plugin architecture allows the user to create instance of plugins, called DANCE objects. All plugins that are available to the DANCE system are located in DANCE\_DIR/plugins/win. Plugins compiled in debug mode have a \_d.dll suffix, while those compiled in release mode have a .dll suffix.

### 6.1 Creating a DANCE Object

To create a DANCE object, select the Plugins/Create menu choice. This will bring up a list of all the plugins available to DANCE.

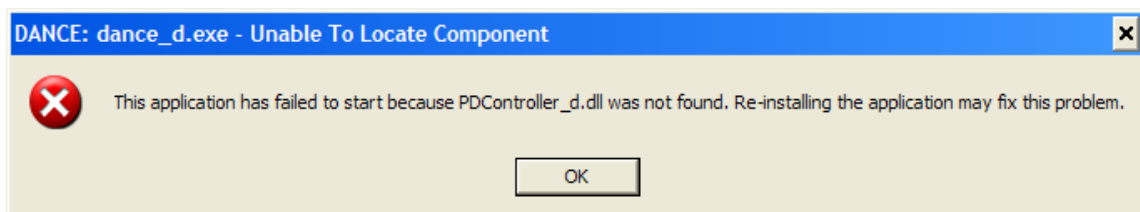


*To create a DANCE object, select the desired plugin, enter the name of the object and press the 'Create Instance' button. Once the instance has been created, the GUI interface for the instance will appear on the interface panel on the right side of the DANCE screen.*



*The above example shows the DANCE environment after creating a Cube instance.*

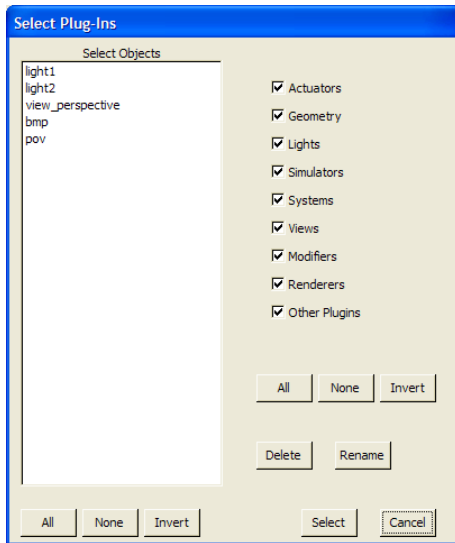
*Please note that some plugins are dependent on other plugins, and the DLL for the dependent plugin must be loaded first. If a plugin instance cannot be created because another plugin needs to be loaded first, a message such as the following will be displayed:*



*To load this dependent plugin, choose the plugin from the plugin list and press the 'Load Plugin' button. This process will not instantiate an object, but rather load the .dll into memory.*

## 6.2 Plugin GUI Interfaces

Each plugin has its own set of GUI interfaces which can be manipulated by the user. GUI interfaces appear on the right side of the DANCE window in a series of tabs. To add an object's interface to the DANCE interface area, choose the Plugins/Select menu item or press 'h' in the main DANCE window area. This will create the following window:

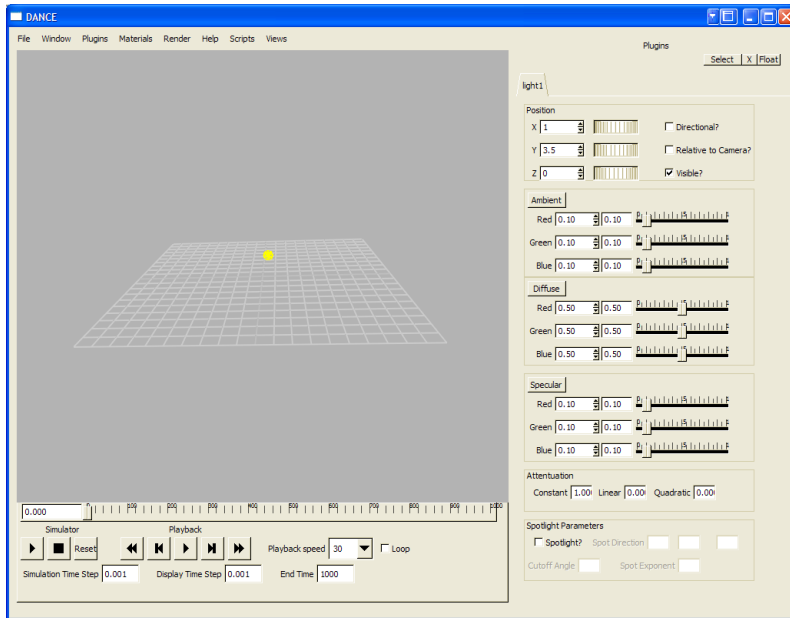


To display the GUI interface for an object, choose that object name and press the Select button.

The DANCE objects available in the scene will appear on the left hand side of the selection window. The checkboxes on the right (Actuators/Geometry/Lights, etc.) filter the list of objects by plugin type. The All/None/Invert buttons immediately below the checkboxes on the right will allow the user to select All the objects types, None of the object types, or Invert the object type list. For example, deselecting the Lights checkbox will eliminate all lights from the object list on the left. Pressing Invert will then show only the lights.

The All/None/Invert buttons on the lower left control which object GUI interfaces will be selected.





Example of GUI interface after choosing object 'light1'. The GUI appears on the right side of the DANCE environment and may be used to manipulate the DANCE object.

### 6.3 Renaming DANCE Objects

DANCE objects may be renamed by opening the Plugin Selection window, selecting a plugin, and then pressing the Rename button.

### 6.4 Deleting DANCE Object

DANCE objects may be deleted by opening the Plugin Selection window, selecting a plugin, and then pressing the Delete button. All objects that are dependent upon the object that is being deleted will be notified of this deletion event.

## 7. Sessions

A DANCE session is the current working environment in DANCE, including the camera position, objects and working set of GUI interfaces. By saving and loading a DANCE session, the entire DANCE workspace can be restored between user sessions.

### 7.1 Saving Sessions

To save your environment to a file, choose the File/Save session menu item. This will prompt you to save your session to a file with a .dpy extension (short for a DANCE Python file). The .dpy file will contain all scripting commands necessary to reconstruct the DANCE environment to its current state. Note that a .dpy file is merely a set of python scripts that will be run in order to restore the DANCE state. There is no difference between a .dpy file and a .py file that contains python commands.

### 7.2 Loading Sessions

To load a session (.dpy file), choose the File/Load session menu item. Please note that a DANCE session often contains references to other files in the file system, and are not necessarily self-supporting. For example, a .dpy file might contain a reference to 3-D model contained in an .obj file. Thus, you cannot move a .dpy file to another computer without moving the referenced files as well. Sessions will be saved to the .dpy file, and all accompanying information will be saved to a subdirectory named dancesession.<sessionname>. To transfer a session to another computer or to share session information with another DANCE user, it is necessary to preserve both the .dpy file as well as the dancesession.<sessionname> directory and its contents.

### 7.3 Resetting Sessions

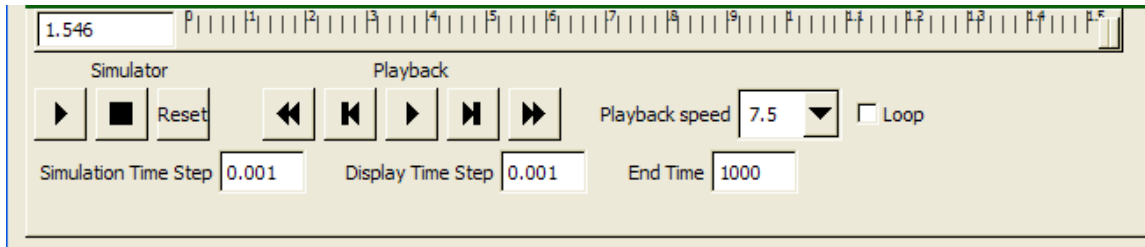
To clear the current session, choose the File/Reset session option. This will load the default set of DANCE commands, described below.

### 7.4 Default Session Commands & profile.py

The DANCE environment on startup loads all Python commands that are contained in the file DANCE\_DIR/profile.py. This file will typically contain default views, lights, camera positions, commonly accessed plugins and the current working directory. Any DANCE command may be placed in this file, including complicated scripting commands. Note that whatever commands are entered in the profile.py file will be run when DANCE session is reset.

## 8. Simulations

DANCE can run many simulators at the same time. Simulator objects generally modify System objects by changing their states. System objects can save their states over time in order to replay the animation.



The main simulator GUI controls can start and stop all simulators using the play, pause and stop buttons on the left. The Reset button restores and rewinds the main simulator to time 0.0 and tells each System to reset its state to the state it has stored at time 0.0.

The Simulation Time Step determines the size of the step of the main simulator (although individual simulators may run at their own time steps). The Display Time Step determines how often the screen is refreshed. The End Time specifies the time when the simulation will automatically stop.

The playback buttons from left to right are: rewind to beginning, go back 1 frame, play/pause, go forward 1 frame, fast forward to end. The playback speed determines how fast the animation will be replayed (30 fps = real time). The loop checkbox can be checked in order to loop the playback animation.

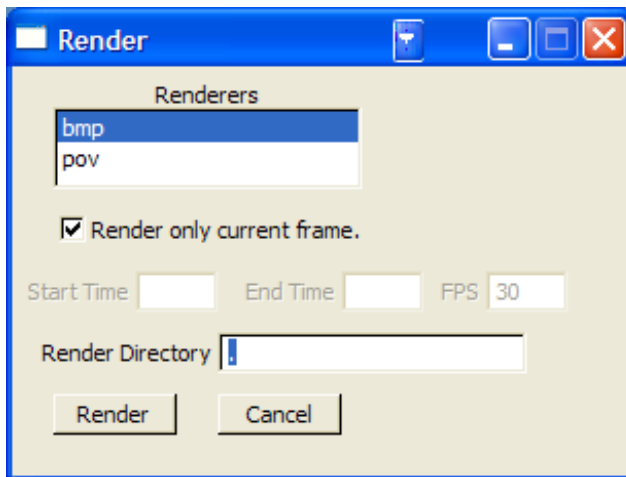
Animations are created from System instances which save their state at various times during a simulation. An animation is played back by restoring the state of the System at different times. Thus, only System plugins and other plugins that save their state over time can be animated.

## 9. Rendering

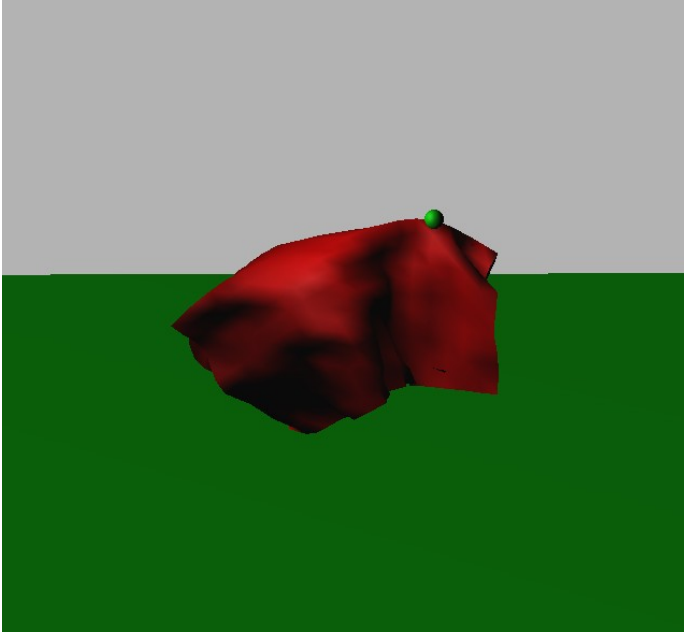
Objects in the DANCE environment can be rendered to a file in two ways (and more if additional Renderer plugins are written): to a file in one of the standard image formats (.png, .bmp, .jpg or .gif), or to a POVRay ([www.povray.org](http://www.povray.org)) scene description file.

### 9.1 Rendering to a Image File

To render a scene to an image file, choose the Render/Render... menu item.



Choose the image\_renderer, and choose the directory you would like the frame rendered to. The '.' directory represents the default working directory which is shown in the command shell. The default image type is .png. A numbered frame will be saved, such as f000001.png. The next frame rendered will be saved as f000002.png, and so forth. Note that the .png renders copies the scene exactly as it appears on screen into a file. To render to an image of a different type (.bmp, .jpg, .gif or .tga) select the GUI interface of the image\_renderer and choose the appropriate type from the drop-down list.



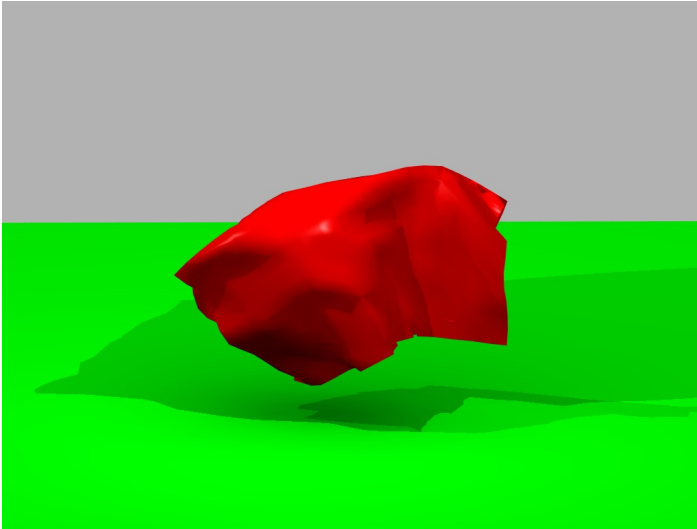
Example .png rendering: Cloth colliding with a large sphere (mostly hidden), suspended by point (green sphere).

## 9.2 Rendering to the Raytracer

Raytraced output is often superior to OpenGL output. To render a scene to the POVRay raytracer, follow the same instructions above and choose the pov renderer. The POVRay renderer produces three files: the scene file (000001.pov, for example), a file that contains lighting information (lights.inc), a file that contains material information (custommaterials.inc) and a file that contains object definitions (objects.inc). This is done so that lighting and material changes can be changed quickly and easily outside of DANCE in order to fine-tune the appearance of the resulting image. For example, the following code can be added to the lights.inc file in order to add radiosity to the rendering:

```
global_settings {  
    radiosity {  
        brightness 2.0  
        count 100  
        error_bound 0.15  
        gray_threshold 0.0  
        low_error_factor 0.2
```

```
    minimum_reuse 0.015
    nearest_count 10
    recursion_limit 5
    adc_bailout 0.01
    max_sample 0.5
    media off
    normal off
    always_sample 1
    pretrace_start 0.08
    pretrace_end 0.01
}
}
```



Example rendering to POVray: this is the same scene from DANCE as shown above. Notice the soft shadows and better lighting.

### 9.3 Rendering an Animation

To render an entire animation, uncheck the 'Render only current frame' box and choose the start time, end time and frames-per-second. An animation stored in the DANCE system will be

rewound and played back at the indicated speed. Note that DANCE does not automatically produce animations from these images. A third-party program (such as Quicktime) can easily assemble the animations into a working video. Future versions of DANCE will accommodate this functionality.

## 10. DANCE Scripting

DANCE allows the user to script their actions using a Python command shell. The scripting environment can access the objects present in the DANCE environment. The entire Python language may be used to run scripts in DANCE. A Python command may be run in DANCE by any of the following ways:

1. Enter Python code in command shell and press the ENTER key. (Multiple lines may be typed by pressing SHIFT+ENTER to add another line without sending the command to the interpreter first).
2. Choose the Scripts/Run script menu item and choose a Python file (.py) that contains Python commands.
3. Place a Python (.py) file in the DANCE\_DIR/scripts directory. The name of the file will appear under the Scripts menu. By choosing that file name from the Scripts menu, the commands in that file will be processed by the DANCE interpreter.

Note that the DANCE Command Window is also used for output from the DANCE environment.

All DANCE objects are accessible via Python. All Python parameters are either string parameters or numeric parameters. Numeric parameters do not need to be quoted as strings, but can be if desired. For example, the command:

```
dance.system("myobject", "size", "2", "3", "5")
```

Is the same as:

```
dance.system("myobject", "size", 2, 3, 5)
```

The following methods are accessible from the dance object:

Method	Description
<code>dance.actuator(objectname, ...)</code>	Sends commands to an Actuator object



<code>dance.system(objectname, ...)</code>	Sends commands to a System object
<code>dance.simulator(objectname, ...)</code>	Sends commands to a Simulator object
<code>dance.geometry(objectname, ...)</code>	Sends commands to a Geometry object
<code>dance.generic(objectname, ...)</code>	Sends commands to a Generic object
<code>dance.modifier(objectname, ...)</code>	Sends commands to a Modifier object
<code>dance.view(objectname, ...)</code>	Sends commands to a View object
<code>dance.light(objectname, ...)</code>	Sends commands to a Light object
<code>dance.renderer(objectname, ...)</code>	Sends commands to a Renderer object
<code>dance.hideinterface(objectname)</code>	Removes the GUI interface for an object from the DANCE interface panel
<code>dance.showinterface(objectname)</code>	Shows the GUI interface for an object on the DANCE interface panel
<code>dance.addinteraction(objectname)</code>	Adds the mouse/keyboard interaction controls for a given object
<code>dance.removeinteraction(objectname)</code>	Removes the mouse/keyboard interaction controls for a given object
<code>dance.reset()</code>	Resets the DANCE session
<code>dance.rename(objectname, newname)</code>	Renames an object
<code>dance.quit()</code>	Quits DANCE
<code>dance.instance(pluginname, objectname, ...)</code>	Creates a new plugin of type <code>pluginname</code> and name <code>objectname</code>
<code>dance.simul(...)</code>	Sends commands to the Simulation Manager
<code>dance.viewmanager(...)</code>	Sends commands to the View Manager
<code>dance.load(filename)</code>	Loads script commands from the file specified
<code>dance.save(filename)</code>	Saves a DANCE session to the file specified
<code>dance.plugin(pluginname)</code>	Loads the DLL for a given plugin type without creating an object.
<code>dance.show(pluginname)</code>	Returns a list of all plugins of that type

<code>dance.exists(objectname)</code>	Returns "yes" if the plugin of that name exists, or "no" otherwise
<code>dance.chdir(newdir)</code>	Changes the current working directory.
<code>dance.ls()</code>	Shows a listing of all files in the current directory.

The commands that can be sent to each instance vary depending on the plugin type. Currently, there is no repository of DANCE commands. Each plugin must be examined to determine which commands it handles (look in the `commandPlugin()` method in the plugin).

However, the DANCE primitive types (Geometry, System, Simulator, Actuator, Modifier, View and Light) all have well-defined commands. Any plugin that inherits from the primitive type classes will also have access to those same commands. The following commands may be sent to primitive plugins and their sub-classes:

Actuators are objects that apply forces to Systems. Actuators can be field forces like gravity, collision forces, or rigid-body controller forces.

<b>Actuator Command</b>	<b>Explanation</b>
<code>dance.actuator("myobject", "apply", "mysystem")</code>	Applies the effects of an actuator to a System object called "mysystem"
<code>dance.actuator("myobject", "apply", "all")</code>	Applies the effects of an actuator to all System objects in the DANCE environment
<code>dance.actuator("myobject", "apply", "none")</code>	Unapplies the effects of an actuator from all System objects.
<code>dance.actuator("myobject", "remove", "mysystem")</code>	Unapplies the effects of an actuator from a the System object called "mysystem"
<code>dance.actuator("myobject", "name", "myname")</code>	Renames the Actuator object to "myname"

Systems are objects that are organized in a semantically meaningful way to model various things such as articulated objects, cloth, particles, and so forth.

<b>System Command</b>	<b>Explanation</b>
<code>dance.system("myobject", "simulators")</code>	Returns a list of Simulator objects affecting this System object.
<code>dance.system("myobject", "simulator", "mysim")</code>	Connects the Simulator object named "mysim" to this System object.
<code>dance.system("myobject", "remove_simulator", "mysim")</code>	Disconnects the Simulator object named "mysim" from this System object.
<code>dance.system("myobject", "record", "on")</code>	Turns the playback recorder on. This will allow the System object to save its state during a simulation and to replay the state during playback.
<code>dance.system("myobject", "record", "off")</code>	Turns the playback recorder off. System objects with their playback recorder's turned off cannot participate in playback.
<code>dance.system("myobject", "recordstep", ".03")</code>	Determines how often the System object will record its state during a simulation (in this case, state will be saved every .03 seconds).
<code>dance.system("myobject", "playback", "on")</code>	Turns the System objects playback flag on. System objects with this function turned on can be animated with the playback buttons in the main simulator bar.
<code>dance.system("myobject", "playback", "off")</code>	Turns the System objects playback flag off. System objects with this function turned off will not be animated with the playback buttons in the main simulator bar.
<code>dance.system("myobject", "name", "myname")</code>	Renames the System object to "myname"

Simulators are objects that change the state of System objects over time.

<b>Simulator Command</b>	<b>Explanation</b>
<code>dance.simulator("myobject", "systems")</code>	Returns a list of System objects affecting this Simulator object.
<code>dance.simulator("myobject", "system", "myobject")</code>	Connects the System object named "myobject" to this Simulator object.
<code>dance.simulator("myobject", "remove", "myobject")</code>	Disconnects the System object named "myobject" from this Simulator object.
<code>dance.simulator("myobject", "timestep", ".001")</code>	Sets the timestep for this Simulator object. The timestep can be different from the timestep of the main simulator.
<code>dance.simulator("myobject", "usetimestep", "true")</code>	Make this Simulator object use the timestep of the main simulator instead of its own timestep value.
<code>dance.system("myobject", "usetimestep", "false")</code>	Make the Simulator object use its own timestep value instead of the main simulator's timestep value.
<code>dance.simulator("myobject", "name", "myname")</code>	Renames the Simulator object to "myname"

Geometries are are objects that represent meshes or primitive shapes, such as cubes, planes, spheres and meshes.

<b>Geometry Command</b>	<b>Explanation</b>
<code>dance.geometry("myobject", "material", "mymaterial")</code>	Sets the material of this Geometry object to the material named "mymaterial" from the Material Manager.
<code>dance.geometry("myobject", "scale", 2, 2, 2)</code>	Scales this Geometry object by the x, y, z values given.
<code>dance.geomtry("myobject", "rotate", "x", 90, yes)</code>	Rotates the Geometry object around the "x" axis by 90 degrees. The final parameter determines if the rotation will occur around the center of the object's bounding box or not.
<code>dance.simulator("myobject", "translate", 10, 5, 8)</code>	Translates this Geometry object by the x, y, z

	values given.
<code>dance.simulator("myobject", "trans_matrix", ...)</code>	Sets this Geometry object's transformation matrix to the 16 parameters that are entered. Values are entered in row-major order.
<code>dance.system("myobject", "use_trans_matrix", "true")</code>	Enables the use of this Geometry object's transformation matrix.
<code>dance.system("myobject", "use_trans_matrix", "false")</code>	Disables the use of this Geometry object's transformation matrix.
<code>dance.simulator("myobject", "apply_transform", "myname')</code>	Applies the transformation matrix to this Geometry object. The effect is dependent on the implement in the Geometry. Typically, this multiples the original vertices, normals and faces by the transformation matrix.
<code>Dance.simulator("myobject", "name", "myname")</code>	Renames the Simulator object to "myname"

Lights are objects that represent lighting in the DANCE environment. They are implemented as OpenGL lights.

<b>Light Command</b>	<b>Explanation</b>
<code>dance.light("myobject", "reset")</code>	Resets the lighting information in OpenGL for this Light object.
<code>dance.light("myobject", "allstatus")</code>	Displays the status of all the OpenGL parameters Light object.
<code>dance.light("myobject", "visible", "on")</code>	Makes this Light object appear as a wireframe sphere in the DANCE scene.
<code>dance.light("myobject", "visible", "off")</code>	Makes this Light object's wireframe sphere in the DANCE scene. The light will still apply its effects, but a representation of the light in the scene will not be seen.
<code>dance.light("myobject", "position", 10, 10, 30, 1)</code>	Sets the position of this Light object. The first three paramers are the x, y, z locations. The fourth corresponds to the fourth parameter in

	OpenGL.
<code>dance.light("myobject", "ambient", .2, 0, .2, 1)</code>	Sets the ambient color values of this Light object.
<code>dance.light("myobject", "diffuse", .2, 0, .2, 1)</code>	Sets the diffuse color values of this Light object.
<code>dance.light("myobject", "specular", .2, 0, .2, 1)</code>	Sets the specular color values of this Light object.
<code>dance.light("myobject", "directional", "on")</code>	Makes this Light object a directional light.
<code>dance.light("myobject", "directional", "off")</code>	Makes this Light object a non-directional light.
<code>dance.light("myobject", "spotlight", "on")</code>	Makes this Light object a spotlight.
<code>dance.light("myobject", "spotlight", "off")</code>	Makes this Light object a normal light.
<code>dance.light("myobject", "spotlight_direction", 1, 0, 0)</code>	Sets the spotlight direction of this Light object to the vector specified.
<code>dance.light("myobject", "spotlight_cutoff_angle", 30)</code>	Specifies the cutoff angle for the spotlight parameters for this Light object.
<code>dance.light("myobject", "spotlight_exponent", 2)</code>	Specifies spotlight exponent for this Light object.
<code>dance.light("myobject", "constant_attenuation", .5)</code>	Specifies the constant rate of attenuation of this Light object.
<code>dance.light("myobject", "linear_attenuation", .5)</code>	Specifies the linear rate of attenuation of this Light object.
<code>dance.light("myobject", "quadratic_attenuation", .5)</code>	Specifies the quadratic rate of attenuation of this Light object.
<code>dance.light("myobject", "name", "myname")</code>	Renames the Light object to "myname"

Views are objects that represent camera-based perspectives of the DANCE scene.

View Command	Explanation
--------------	-------------

<code>dance.view("myobject", "background", .1, .1, .1, .1)</code>	Change the background color to r,g,b,a values.
<code>dance.view("myobject", "projtype", "persp")</code>	Sets the projection type of the view. 2 <sup>nd</sup> parameter can be one of the following: persp, top, right, front
<code>dance.view("myobject", "dump", "start")</code>	Write each screen refresh to a file
<code>dance.view("myobject", "dump", "stop")</code>	Stop writing each screen refresh to a file
<code>dance.view("myobject", "dump", "- d", "/path/to/mydir")</code>	Write files saved from the screen into the directory /path/to/mydir
<code>dance.view("myobject", "dump", "- f", "myfile")</code>	Use myfile as the prefix for dumped files.
<code>dance.view("myobject", "dump", "- n", 100)</code>	Start numbering the saved files from 100
<code>dance.view("myobject", "fitview")</code>	Fit all objects in DANCE into the current view.
<code>dance.view("myobject", "refresh")</code>	Refresh view
<code>dance.view("myobject", "lights", "on")</code>	Turns on lights for the view.
<code>dance.view("myobject", "lights", "off")</code>	Turns off lights for the view.
<code>dance.view("myobject", "shadows", "on")</code>	Turns on shadows for the view.
<code>dance.view("myobject", "shadows", "off")</code>	Turns off shadows for the view.
<code>dance.view("myobject", "orientation", ...)</code>	Sets the orientation 4x4 matrix for the camera. Needs 16 values.
<code>dance.view("myobject", "grid", "on")</code>	Turns on the grid for the view.
<code>dance.view("myobject", "grid", "off")</code>	Turns off the grid for the view.

Renderers represent object that capture the DANCE scene and transform them into image-based representations (saving the scene to an image file or to a raytracer).

Renderer Command	Explanation
dance.renderer("myobject", "render_directory", "c:/dance_v4/run")	Changes the default rendering directory to "c:/ dance_v4/run"
dance.renderer("myobject", "name", "myname")	Renames the Renderer object to "myname"



## 11. Controllers in DANCE

Skeletons in DANCE can use dynamic controllers during physical simulation. There are several different types of controllers that can be created and used:

- Pose controllers
  - These controllers are specified by creating a file that describes the pose of the character. These controllers can be constructed either through the PosePDController UI by copying an existing pose, or by hand creating a .state file.
- Scripted controllers
  - These controllers are created by creating Python classes that utilize a controller API that allows the skeleton to provide joint torques based on sensor information, state of the environment and so forth. Scripted controllers can access DANCE objects through the controller scripting API.
- C++ Controllers
  - These controllers are written using C++ and can utilize any DANCE information or method.

### 11.1 Pose Controllers

#### 11.1.1 Pose controllers

Pose controllers are specified in a .state file as follows:

```
bone1dof1 bone1dof2 bone1dof3 bone4dof4
bone2dof1 bone2dof2 bone2dof3
bone3dof1 bone1dof2
....
....
! bone1dof1strength bone1dof2strength bone1dof3strength bone4dof4strength
bone2dof1strength bone2dof2strength bone2dof3strength
bone3dof1strength bone1dof2strength
```

The first section is a space-separated list of the state of each bone's degree of freedom. The second section, which is optional and begins with an exclamation mark (!), shows the relative strength used to maintain the pose for each degree of freedom. Note that the format of this state file (using the first section only) is the same as what is created when the Save Position button is pressed on the skeleton GUI. Note that controllers will include degree of freedoms that specify translations, and are only meaningful for degrees of freedom that specify rotational values, since the controllers are run via PD (proportional derivative) control, and generate joint torques. As such, only rotational values can be generated.

## 11.1.2 Scripted controllers

### 11.1.2.1 Python Class

Controllers can be designed by writing a controller class in Python that implements the controller methods, specifically:

`start()`

called when the controller is started

`step()`

called every time step

`stop()`

called when the controller is stopped

`success()`

to determine if the controller was successful or not

### 11.1.2.2 Use of Sensors During Controller Scripting

In addition, it is necessary to load a Sensor plugin for the skeleton for the controller scripts to work properly. The Sensor class will allow access to state information of the skeleton (how many bones/joints, where are they located, etc.).

### 11.1.2.3 Script Example

As an example, the following code illustrates how to create a controller that will place the skeleton in a ragdoll state (although it is easy to turn the skeleton into a ragdoll – simply don't use a PosePDController. This example shows how to use the scripting controllers to create the ragdoll).

The following is code that is placed into the file called ragdoll.py:

```
import PoseInterpreter as pi
import os
```

```
ddir = dance.dancedir()
```

```

execfile(ddir + "/bin/control.py")

class Ragdoll:

    def __init__(self, argPose):
        pi.setPythonController(argPose, self)
        pi.printString("Init Ragdoll...")

    def start(self, argPose):
        pi.printString("Start Ragdoll...")

    def step(self, argPose):
        # make loose skeleton
        numJoints = pi.getNumJoints(argPose)
        for i in range(0, numJoints):
            jointSize = pi.getJointStateSize(argPose, i)
            for j in range(0, jointSize):
                pi.setControlParam(argPose, i, j, 2,
0.0);
                pi.setControlParam(argPose, i, j, 3,
0.0);

    def stop(self, argPose):
        pi.printString("Stop Ragdoll...")

    def success(self, argPose):
        pi.printString("Success Ragdoll...")

def newRagdoll(argPose):
    return Ragdoll(argPose)

globals()['Ragdoll'] = Ragdoll
globals()['newRagdoll'] = newRagdoll

```

The `pi.setPythonController()` command is necessary to indicate that the controller will be processed with this code. The `printString()` commands will display text in the command window. Note that the `getJointStateSize()` indicates how many degrees of freedom that each joint has (a hinge/pin joint has one, a universal joint has two, a ball joint has four – one for each value of a quaternion). Note also that the `setControlParam()` method's fourth parameter is to specify which aspect of the control parameter to use where:

- 0 = drive type
- 1 = control position
- 2 = ks value
- 3 = kd value
- 4 = maximum torque

So the `ragdoll.py` file is specifying a 0.0 ks and 0.0 kd value for every degree of freedom on every joint, which will turn the skeleton into a ragdoll.

To see how the ks values affect the skeleton, you can change ks value to 1.0. For example, to make a completely stiff skeleton, set all ks values to 3.0 and kd values to 1.0. Note that any time

you change the .py code, you must reload the script by pressing the Reload Script button in the PosePDController GUI.

Here is another example of using the point and line drawing mechanisms to assist with controller development. The following script displays the center of mass and the center of mass velocity for the skeleton: (from PointLineTest.py)

```
import PoseInterpreter as pi
import os

ddir = dance.dancedir()
execfile(ddir + "/bin/control.py")

class PointLineTest:
    def __init__(self, argPose):
        pi.setPythonController(argPose, self)
        pi.printString("Init...")

    def start(self, argPose):
        pi.printString("Start...")
        pi.clearGraphics(argPose);
        #pi.setControlUpdateFrequency(self, .016)

    def step(self, argPose):
        t = pi.getTime()

        ##### center of mass #####
        # get the center of mass
        com = pi.getCenterMass(argPose)
        # show the center of mass
        pi.addPoint(argPose, "com", com[0], com[1], com[2])
        # show the center of mass projected on flat ground
        pi.addPoint(argPose, "comprojground", com[0], 0.0, com[2])
        # draw a line to com ground projection
        pi.addLine(argPose, "comprojgroundline", com[0], com[1],
com[2], com[0], 0.0, com[2], 1.0, 1.0, 0.0)

        ##### center of mass velocity #####
        comaccel = pi.getCMVelocity(argPose);
        scale = 1.0
        pi.addLine(argPose, "comvelocity", com[0], com[1], com[2],
com[0] + scale * comaccel[0], scale * com[1] + scale * comaccel[1],
com[2] + comaccel[2])

    def stop(self, argPose):
        pi.printString("Stop...")

    def success(self, argPose):
        pi.printString("Success...")

def newPointLineTest(argPose):
    return PointLineTest(argPose)

globals()['PointLineTest'] = PointLineTest
```

```
globals()['newPointLineTest'] = newPointLineTest
```

Other examples of controller scripts can be found in the \$DANCE\_DIR/run directory.

#### 11.1.2.4 How to Run the Scripted Controller

Using the PosePDController UI, press the 'Load Pose' button and find the .py file that contains the scripted controller. Note that it is necessary to have created a Sensor plugin (use SensorSkel18 for the default skeleton). The scripted pose will function the same way as other poses function. They can be run interactively by selecting their name in the Pose List, or they can be run on the Non-Linear Editor.

Note that errors from the script will appear in the command window. Remember that if any changes are made to the .py script after it is initially loaded, you must press the Reload Script button on the PosePDController GUI.

#### 11.1.2.5 Scripting methods

The following is a list of methods used for controller scripting. Note that each method must be run using the PoseInterpreter namespace. The following list can also be generated by using the following command in the command window:

```
help(pi)
```

##### **addLine(...)**

```
(pose, name, fromx, fromy, fromz, tox, toy, toz) or (pose, name,
fromx, fromy, fromz, tox, toy, toz, colorr, colorg, colorb) draws
a line on the screen at from fromx, fromy, fromz to tox, toy,
toz()
```

##### **addPoint(...)**

```
(pose, name, x, y, z) or (pose, name, x, y, z, colorr, colorg,
colorb) draws a point on the screen at location x, y, z ()
```

##### **applyImpulseToLink(...)**

```
(pose, bodyNum, x, y, z) apply an impulse to the skeleton's
link in direction x, y, z ()
```

##### **attachLink(...)**

```
(pose, linkNum, geometryName) attaches a link to static geometry
()
```

##### **attachLinkAO(...)**

```
(pose, linkNum, targetSkeleton, targetLink) attaches a link to
another link on a different skeleton ()
```

##### **attachLinkStatic(...)**

```
(pose, linkNum, geometryName) attaches a link to static geometry
()
```

##### **clearGraphics(...)**

(pose) removes all points, lines and graphical shapes.

**detachLink(...)**

(pose, linkNum, geometryName) detaches a link from geometry ()

**detachLinkAO(...)**

(pose, linkNum, targetSkeleton, targetLink) dettaches a link to another link on a different skeleton ()

**detachLinkStatic(...)**

(pose, linkNum, geometryName) detaches a link from static geometry ()

**getCMRelativeDistance(...)**

(pose) gets the distance between the center of mass and the support polygon (float[3])

**getCMVelocity(...)**

(pose) gets center of mass velocity (float[3])

**getCenterMass(...)**

(pose) returns center of mass (float[3])

**getControlParam(...)**

(pose, linkNum, dofNum, param) get a single control param by linkNum, dof, param (0=driveType,1=v,2=ks,3=kd,4=maxTorque) (float)

**getControlParams(...)**

(pose, linkNum, dof) get the control params by linkNum, dof, returns an array holding values ([DriveType,v,KS,KD,MaxTorque])

**getControlUpdateFrequency(...)**

(pose) gets the number of timesteps per control execution (float)

**getDistToSupportPolygon(...)**

(pose) returns the (signed) distance from the character's ground-projected center of mass to the closest point on the support polygon. The distance is positive if the CM is inside the support polygon. (float[3])

**getEulerFromBall(...)**

(pose, jointNum, XYZorder) gets an euler decomposition of the orientation (float[3])

**getFacing(...)**

(pose) get facing for root/hip (float[3])

**getIndxFirstConDof(...)**

(pose) gets the index of the first controllable degree of freedom (int)

**getJointFromPose(...)**

(pose, poseIndex, jointIndex) gets a joint position from the indicated pose (float[numDof]).

**getJointPosition(...)**  
(pose, jointNum) gets a joint's position (float[3])

**getJointStateSize(...)**  
(pose, jointNum) gets the number of joints/links (int)

**getLerpJointFromStates(...)**  
(pose, jointNum, state1[], state2[], interpolation) interpolate the joint values (float[])

**getNearestPointOnSupportPolygon(...)**  
(pose) returns a 3-element array containing the point on the support polygon closest to the character's ground-projected center of mass (float[3]).

**getNumJoints(...)**  
(pose) gets the number of joints/links (int)

**getOrientation(...)**  
(pose, bodyNum, vecx, vecy, vecz, targetBody) transforms a vector in body coordinates to world coordinates (float[3])

**getPosition(...)**  
(pose, bodyNum) returns position (float[3])

**getPositionLink(...)**  
(objectName, linkNum) gets a link's position (float[3])

**getPositionObject(...)**  
(objectName) returns object position (float[3])

**getSimIndex(...)**  
(pose, linkIndex, dofIndex) given the link and dof index it returns the equivalent simulator index (integer)

**getState(...)**  
(pose, stateIndex) gets the state value of a degree of freedom (float)

**getStateSize(...)**  
(pose) gets the skeleton state size (int)

**getTime(...)**  
( ) returns time

**getUpVector(...)**  
(pose) get up vector for root/hip (float[3])

**getVelocity(...)**  
(pose, bodyNum) returns velocity (float[3])

**getVelocityObj(...)**  
(string) gets velocity of an object by name (float[3])

**getWorldQuaternion(...)**  
(pose, bodyNum) returns the orientation quaternion of link with respect to the world (float[4])

**getWorldZeroXYZAngles(...)**  
(pose, linkNum) returns an array of XYZ euler angles that will make the arg link's world orientation zero.(x, y, z)

**interpolatePoses(...)**  
(pose, poseindex1, poseindex2, interpolation) interpolate the given poses and set the control values ()

**isAttached(...)**  
(pose, linkNum, geometryName) checks to see if given link is attached (int)

**isAttachedAO(...)**  
(pose, linkNum, targetSkeleton, targetLink) checks to see if given link is attached to a particular link on another skeleton ()

**isAttachedStatic(...)**  
(pose, linkNum, geometryName) checks to see if given link is attached to static geometry ()

**isLinkColliding(...)**  
(pose, bodyNum) returns if link is in collision (bool)

**isLinkOnGround(...)**  
(pose, bodyNum) returns if link is on the ground (bool)

**magic(...)**  
(pose, x, y, z) adds force to counteract current velocity in direction indicated by x, y, z ()

**magicBody(...)**  
(pose, bodyNum, x, y, z) adds force to counteract body's velocity in direction indicated by x, y, z ()

**magicBodyToBody(...)**  
(pose, bodyNum, x, y, z, objectName) adds force to counteract body's velocity in direction indicated by x, y, z relative to otherBody ()

**magicTorque(...)**  
(pose, bodyNum, x, y, z, objectName) adds torque to counteract body's angular velocity in direction indicated by x, y, z ()

**noise(...)**  
(key) returns one-dimensional Perlin noise (float)

**printString(...)**  
(string) prints the input ()

**printValueDouble(...)**  
(floatvalue) or (floatvalue1, floatvalue2, floatvalue3) prints the input double or vector ()

**printValueInt(...)**  
(intvalue) prints the input integer ()



**reach(...)**  
(pose, ikName, x, y, z, distance, elbowDirX, elbowDirY, elbowDirZ) sets IK chain to x,y,z location with given elbow direction ()

**reachLink(...)**  
(pose, ikName, objectName, linkName, elbowDirX, elbowDirY, elbowDirZ) sets IK chain to location indicated by linkName with given elbow direction ()

**reachObject(...)**  
(pose, ikName, objectName, distance, x, y, z) sets IK chain to location indicated by objectName with given elbow direction ()

**reachOffset(...)**  
(pose, ikName, x, y, z, distance, elbowDirX, elbowDirY, elbowDirZ, offsetX, offsetY, offsetZ) sets IK chain to x,y,z location with given elbow direction and offset from end effector ()

**removeLine(...)**  
(pose, name) removes a line with given name from the screen ()

**removePoint(...)**  
(pose, name) removes point with given name from the screen ()

**setControlParam(...)**  
(pose, linkNum, dofNum, param, value) set a single control param by linkNum, dof, param (0=driveType,1=v,2=ks,3=kd,4=maxTorque) and value ()

**setControlUpdateFrequency(...)**  
(pose, rate) sets the number of timesteps per control execution ()

**setFootFlat(...)**  
(pose, linkNum, eulerOrder) sets control parameters so that the foot is flat ()

**setJoint1(...)**  
(pose, jointNumber, position) sets the joint position in radians of a pin joint ()

**setJoint2(...)**  
(pose, jointNumber, position1, position2) sets the joint position in radians of a universal joint ()

**setJoint3(...)**  
(pose, jointNumber, position1, position2, position3) sets the joint position of a gimbal joint ()

**setJoint4(...)**  
(pose, jointNumber, q1, q2, q3, q4) sets the joint position of a ball joint as quaternion values ()

**setJoint4Euler(...)**

(pose, jointNumber, x, y, z sets the joint position of a ball joint as Euler values in radians ())

**setJointKsKd1(...)**

(pose, jointNum, ks, kd) sets the ks and kd of a pin joint ()

**setJointKsKd2(...)**

(pose, jointNum, ks1, kd1, ks2, kd2) sets the ks and kd of a universal joint ()

**setJointKsKd3(...)**

(pose, jointNum, ks1, kd1, ks2, kd2, ks3, kd3) sets the ks and kd of a gimbal joint ()

**setJointKsKd4(...)**

(pose, jointNum, ks1, kd1, ks2, kd2, ks3, kd3) sets the ks and kd of a gimbal joint ()

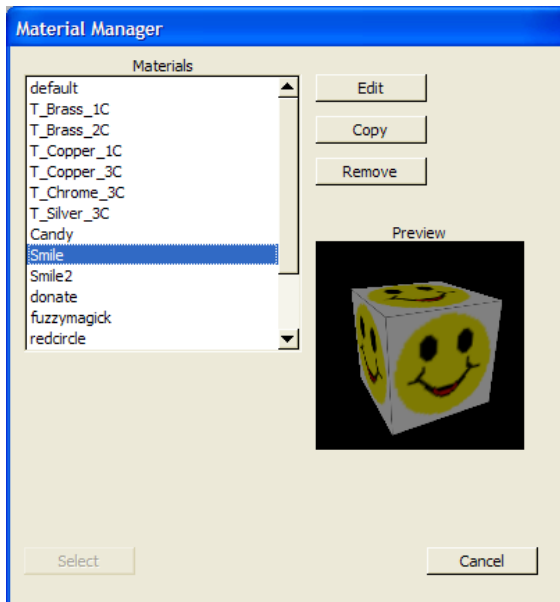
### 11.1.3 C++ Controllers

An example of using C++ to create a controller is given in the project TestController. Please refer to that code and the comments in that code for details. Note that controllers written in C++ have access to all methods and capabilities of the DANCE system by invoking DANCE methods directly.

## 12. Materials

DANCE maintains a material editor that can be used to apply various materials to objects in the DANCE scene. The definitions for these materials are located in the file:

DANCE\_DIR/data/materials/stock.txt. Please consult that file to determine the format for the materials. Textures can be loaded from most common file formats (.png, .bmp, .jpg, etc.).



New materials can be created by pressing the Copy button, which will create a duplicate material with a modified name. You can then choose that new material and press the Edit button. Note that at the time of this writing, materials created in a DANCE session will not be restored in a later session. To add a material permanently, the description must be added to the DANCE\_DIR/data/materials/stock.txt file.

### **13. Plugin Descriptions**

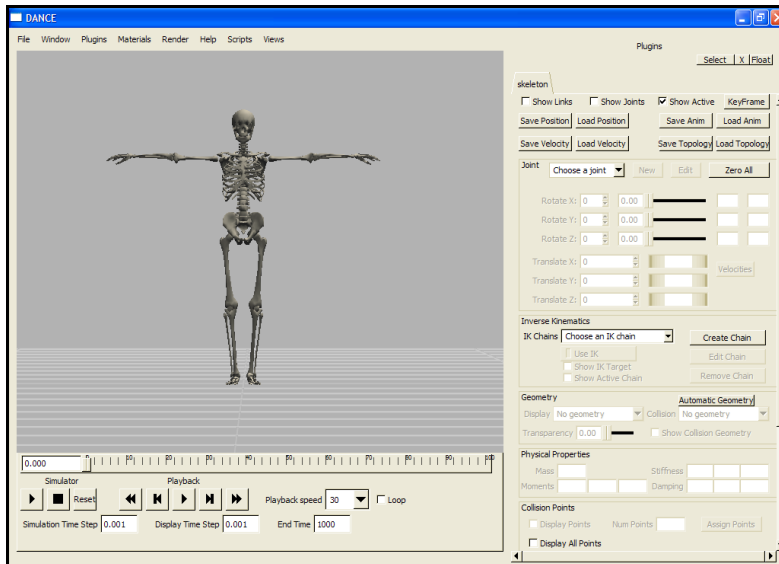
Each DANCE plugin has its own capabilities and interface. Details for these plugins are located in the section titled 'II. DANCE Plugin User Guide'.

## II. DANCE Plugin User Guide

### Plugins

#### 1. ArticulatedObject

The ArticulatedObject plugin is a plugin that manages hierarchically connected rigid bodies. It can be used to model simple rigid bodies (displayed as simple polygons) as well as complex figures (humanoid skeletons). For the purposes of this description, we will interchange the word 'skeleton' with the name of the plugin 'ArticulatedObject'.



An ArticulatedObject instance representing a human skeleton.

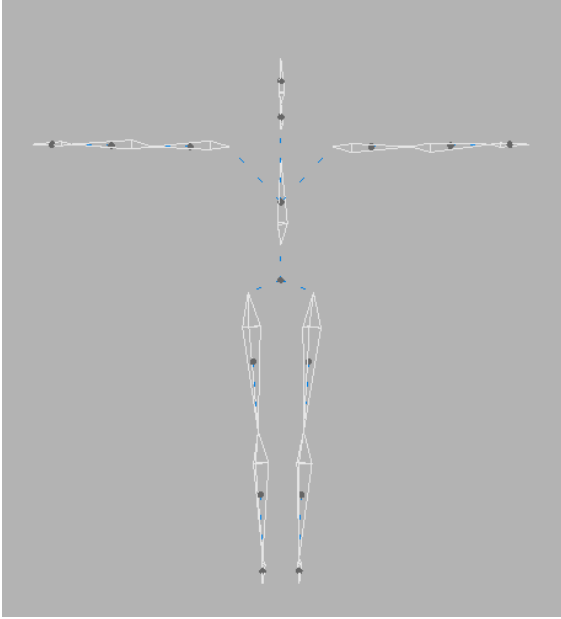
The following features are available in the ArticulatedObject plugin:

- Interactive creation of joint hierarchies (New, Edit)
  - Joints may be created or added by pressing the New button.
  - Joints may be edited by pressing the Edit button.

- Joint limits
- Joint types – pin, universal, gimbal, free, weld
- Explicit assignment of joint velocities
- Variable rotation orders (XYZ, ZXY, etc.)
- Keyframing
- Save/Load position (.state files)
- Save/Load velocities (.sstate files)
- Save/Load animation (.bvh files)
- Save/Load topology (.sd files)
- Inverse kinematics
- Automatic geometry creation (spheres/ellipses or cubes)
- Manipulation of physical properties (mass, inertia, stiffness, damping)
- Point-based collision information (monitor points)
- Create/Save/Load sphere-based collision information (.sphere files)
- Props

### **1.1 Creating a new skeleton**

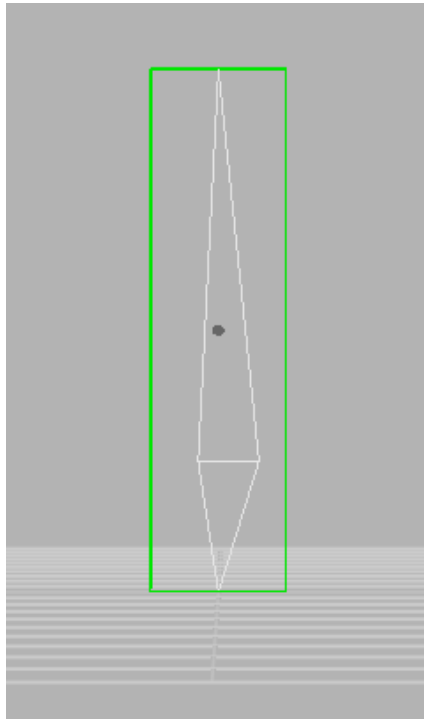
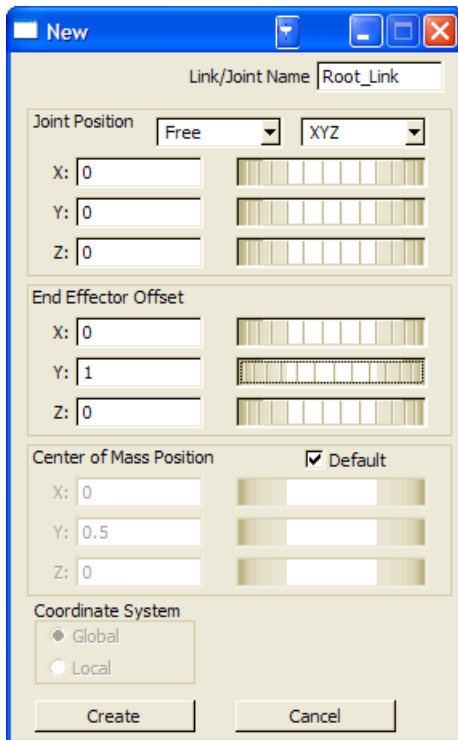
An ArticulatedObject consists of links and joints attached to each other hierarchically. The links represent bones or rigid bodies, while joints connect links together and constrain their movement. Every ArticulatedObject has an equal number of links and joints.



Above is shown the default visualization of links and joints. The links are represented by white tetrahedrons. The black dots indicate the center of mass of the links. The dotted blue lines indicate connections between the center of mass of a link and its joints. The joints are shown implicitly at the end of the dotted blue line. In order to enhance the appearance of the skeleton, geometry objects may be attached to the links.

### **1.1.1 Interactive Skeleton Topology**

Skeleton topology may be created interactively by clicking on the New button in the ArticulatedObject GUI interface.



Each link/joint pair is created together. By default, the center of mass (COM) will be placed halfway between the joint position and the end effector position. The COM may be set manually by unchecking the Default checkbox in the Center of Mass Position area. A green bounding box will be placed around the border of the active link. To create the link with an accompanying joint, press the Create button. The Cancel button will remove this link.

The joint may be set to any of the following types of joints:

Joint Type	Comments
Free	6-DOF (3 translation, 3 rotation)
Gimbal	3-DOF (3 rotation)
Universal	2-DOF (2 rotation)
Pin	1-DOF (1 rotation)
Weld	0-DOF (fixed joint)

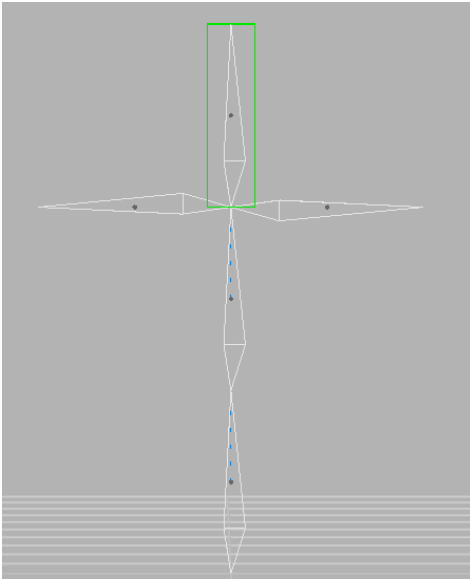


Note that the rotation order can be set for each joint (XYZ or ZYX, for example).

Note that some simulators may support more joints or place restrictions on other joints. For example, ODE does not allow you to place a universal joint as the root joint. In addition, ODE requires that the joint location of a free rigid body be its COM. When possible, DANCE will warn the user of these situations.

Note that an ArticulatedObject can only have 1 root link/joint. All ArticulatedObjects use a tree hierarchy and must be attached. Thus, free joints may only exist at the root joint.

By default, the newly created link/joint pair becomes the new active link. Additional child link/joint pairs may be created by choosing the appropriate parent joint from the Joint dropdown list, then pressing the New button.



A slightly more complex ArticulatedObject is shown above. The second link from the bottom has 3 children: one that points left, one that points right and one that points upwards. This object was created by creating one of the children then selecting the same parent joint from the dropdown list.

Link/joint pairs may be edited by selecting the appropriate joint and pressing the Edit button.

### 1.1.2 Loading/Saving Skeleton Topologies from files

A skeleton topology may be saved to a skeleton topology file. The format of the skeleton topology file is that of the sd/fast .sd file. Please refer to the sd/fast documentation for details. A skeleton topology may also be loaded from a motion capture file (.bvh). The skeleton topology will be automatically gleaned from the structure of the motion capture file. When loading skeleton topologies from .bvh files, DANCE will assign a mass of 1 kg to each link and a default moment of inertia. It is recommended that new masses and moments of inertia be assigned when loading topologies from motion capture files.

A skeleton topology may be saved and loaded by pressing the Save Topology and Load Topology buttons.

### 1.1.3 Converting Motion Capture to Skeletons

Animations generated from motion capture data can be transformed into ArticulatedObjects. Please see the MotionPlayer plugin for details. Currently, the conversion works only from .bvh files. Conversions from asf/amc files will produce improper rotations for those joints that included non-zero axis information.

## 1.2 Displaying geometry on skeletons

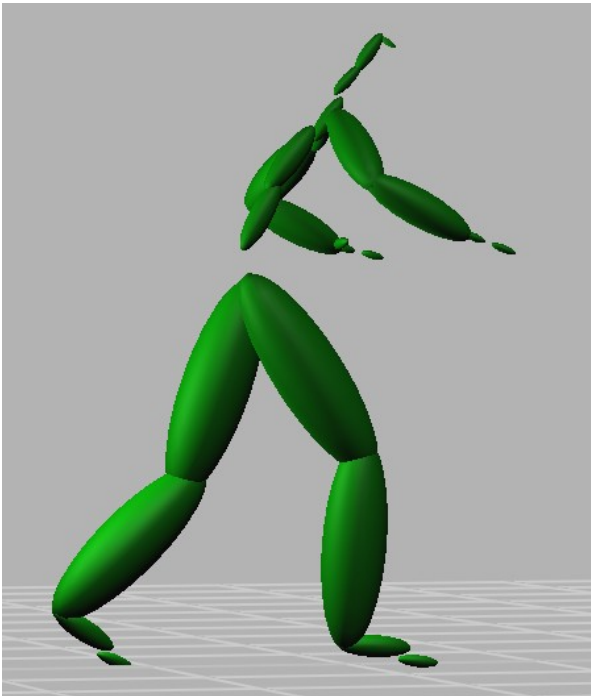
Geometry of any type (Cubes, Spheres, Models, or others) may be applied to each link of the skeleton. Once attached, the Geometry will be displayed instead of the default skeleton tetrahedron. In addition, the Geometry will be moved as the skeleton moves.

To attach Geometry to a skeleton:

1. instantiate a Geometry instance
2. position the instance to the relative location on the skeleton
3. choose the joint/link name from the joint dropdown list that will be attached
4. in the geometry area of the ArticulatedObject GUI interface, choose the Geometry from the Display dropdown. This will attach the Geometry in the link's local coordinates.
5. To reposition the Geometry, select No Geometry in the Display dropdown, move the Geometry, then select the Geometry again.

Note that any instance derived from a Geometry class may be used. This includes the primitive types (Cubes, Spheres, etc.) as well as instances of the Model class which can be used to load .3ds, .obj and .wrl files.

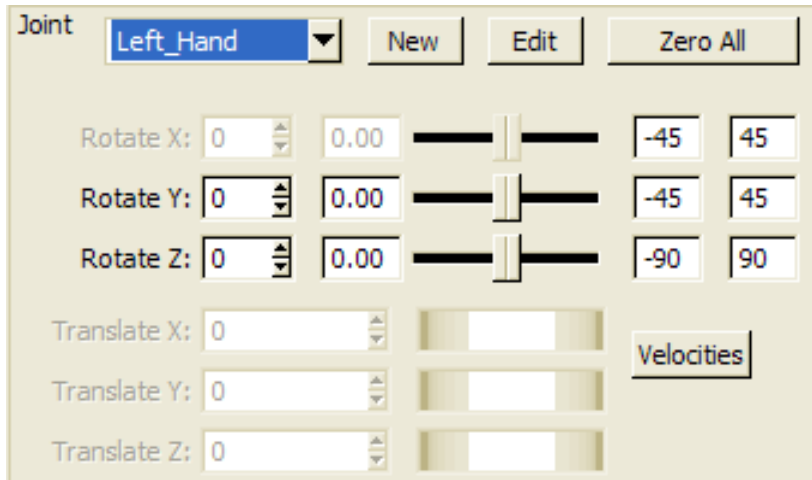
Spheres or cubes can be automatically assigned to each link by pressing the 'Automatic Geometry' button.



The picture above shows an ArticulatedObject with spheres automatically connected to each link after pressing the 'Automatic Geometry' button. The spheres have been automatically stretched into ellipses in order to fit the approximate bounding box of each link. Note that this will also create a default set of collision points for each ellipse. Note that the user may instead choose to automatically assign Cubes rather than Spheres.

## 1.2 Positioning the skeleton

Skeletons can be positioned by changing the value of each link/joint pair. Choose the joint from the joint dropdown list.



The available rotation and translation channels will be shown and can be manipulated through direct input or by using the sliders. Note that only the root joint will have translation channels. The numbers to the right of the joint sliders indicate the joint limits.

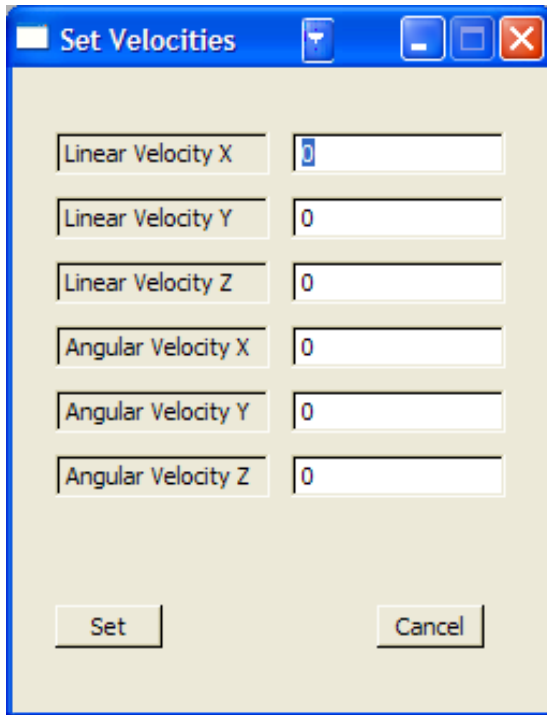
To reset the skeleton to the zero position, press the Zero All button. This will set the value all translation and rotation channels to 0.

To save the position to a file, press the Save Position button. This file format (.state) saves the value of each channel to the file, each line separating the joint. To subsequently load a saved position from a file, press the Load Position button.

In order to position many joints and DOF at the same time using inverse kinematics, see the Section 1.7 below.

### 1.2.1 Setting velocities

You can set velocities on the joints in addition to setting the position. Select the proper joint from the joint dropdown list and press the 'Velocities' button.

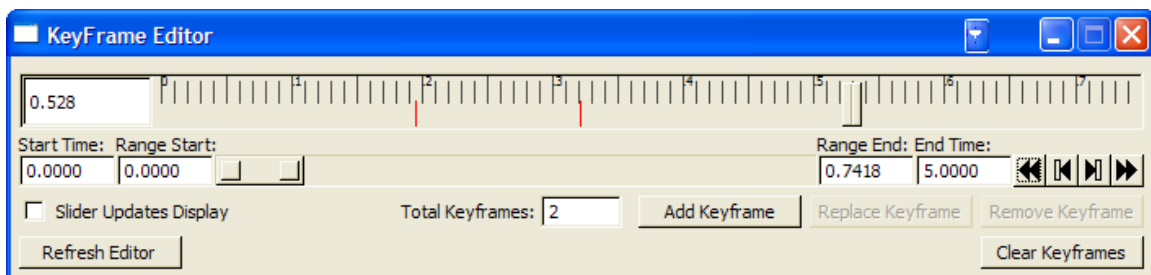


Each velocity channel is displayed and may be entered manually. To save the velocity settings for the entire skeleton, press the 'Save Velocity' button. To restore velocities stored to a file, press 'Load Velocity'.

Note that velocities will only be used to seed simulations.

### 1.3 Keyframing

The ArticulatedObject can be keyframed. To keyframe a skeleton, press the 'Keyframe' button.



The red lines indicate keyframes. The start and end times indicate the span of the frames. The range start and range end indicate the range of times that are currently visible in the keyframe editor.

To save a keyframe, move the slider to the appropriate time, position the ArticulatedObject and press the 'Add Keyframe' button. The keyframe will be indicated by a red tick mark. To animate the ArticulatedObject, use the playback controls on the main simulator or select the 'Slider Updates Display' checkbox and move the slider back and forth.

Note that if a simulation is run that animates the ArticulatedObject, the keyframe editor will show the saved animation as keyframes.

## **1.4 Animation**

All ArticulatedObjects store their state for later playback. This animation can be loaded and saved as a .bvh file. To play an animation that is stored in an ArticulatedObject, use the main Playback controls.

### **1.4.1 Loading an Animation**

To load an animation to apply to the ArticulatedObject, press the 'Load Anim' button. This will prompt you for a .bvh file whose hierarchy matches the hierarchy and channel setup of the ArticulatedObject. If the hierarchy does not match, or if there is a discrepancy between the .bvh file and the ArticulatedObject, a warning will be posted and the animation will not be loaded.

### **1.4.2 Saving an Animation**

To save an animation to a .bvh file, press the 'Save Animation' button. Animations will be saved at 30 frames per second.

### **1.4.3 Saving an Animation**

To save an animation to a .bvh file, press the 'Save Animation' button. Animations will be saved at 30 frames per second.

## **1.5 Simulating the skeleton**

An ArticulatedObject may be simulated by an DSimulator object that understands the ArticulatedObject structure. To physically simulate a skeleton, instantiate a Simulator plugin (ODESim, for example), attach it to the ArticulatedObject, then press the simulator play button on the main simulator. (It might be necessary to add a FieldActuator to simulate the effects of gravity).

Currently, there are two physical simulators included in DANCE: ODESim and SdfastSimul. ODESim uses the Open Dynamics Engine simulator, while SdFastSimul uses the Sd/Fast

simulator. ODESim can simulate arbitrarily designed skeletons, while SdFastSimul requires that the skeleton hierarchy be compiled and loaded as a .DLL. The DANCE distribution includes two SdFastSimul skeletons: a 38-DOF, 16 link three-dimensional skeleton called 'skel18', and an 18-DOF 16-link two-dimensional skeleton called 'robot'.

Other Simulator plugins can be built that do not simulate the ArticulatedObject through physical means. For example, a Simulator plugin can be built that performs kinematic animation on the ArticulatedObject, rather than physical simulation.

### 1.5.1 Physical properties

The physical properties area of the ArticulatedObject GUI interface allows you to set the physical properties of each link, including mass, moments of inertia, stiffness and damping.

Physical Properties								
Mass	16.61				Stiffness	10	10	10
Moments	0.18	0.16	0.23	Damping	10	10	10	

Note that physical properties cannot be changed while simulating an ArticulatedObject using the SdFastSimul simulator, since the physical properties are typically compiled into the simulation source code. ArticulatedObjects that use the ODESim simulator will accept changes in the physical properties.

### 1.5.2 Initial Velocities

Upon starting a simulation, initial velocities of an ArticulatedObject are automatically inferred by examining past keyframes. For example, if a simulation is started during second 2.0, keyframes stored at times 1.97 and 1.94 will be consulted to determine if any initial velocity exists. If no keyframes or stored animation exists, the initial velocity will be zero for all bodies.

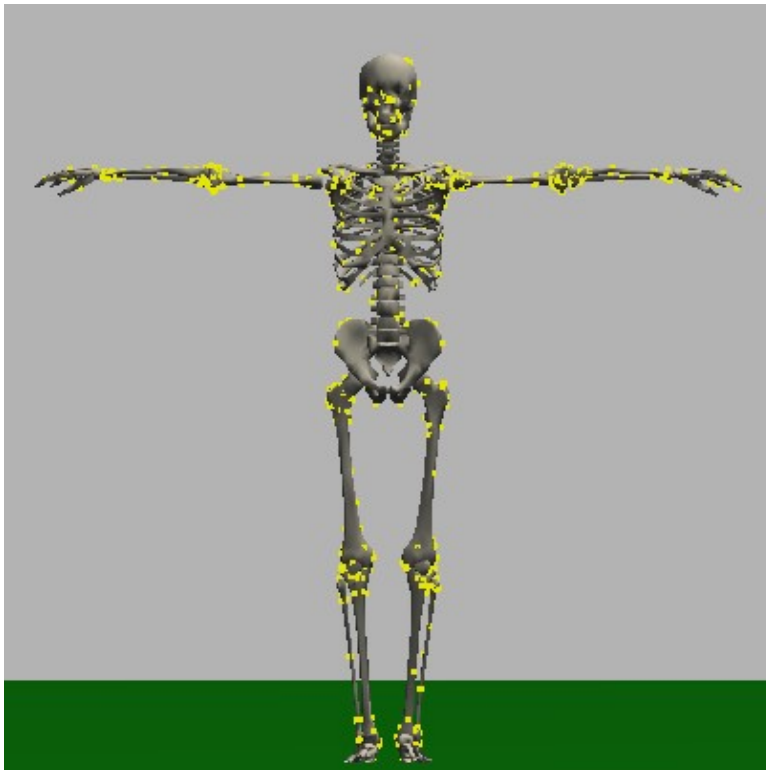
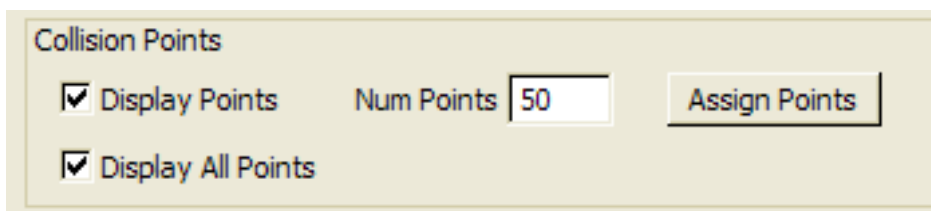
### 1.6 Collisions

Every ArticulatedObject has two built-in collision types: points and spheres. Points can be used to monitor various locations on an ArticulatedObject for use in detecting collisions with other shapes, such as planes, spheres and so forth. Spheres can be used to detect collisions between other spheres, planes, points and so forth. In the basic DANCE distribution, points are used under physical simulation to determine ground penetration, while spheres are used to detect collisions between skeletons as well as collisions between body parts.

Currently, DANCE does not have built-in support for polygonal collisions. However, this can be easily added as a plugin that could incorporate, for example, RAPID, Swift, or ODE's built-in collision detection system.

### 1.6.1 Point Collisions

Each link on the ArticulatedObject can be assigned a number of monitor points used in collision detection. To show the locations of the currently assigned points for a particular link, check Display Points. To see the currently assigned points for all the links, check Display All Points. By entering a number in the Num Points input and pressing the 'Assign Points' button, that number of points will be randomly assigned to the link.



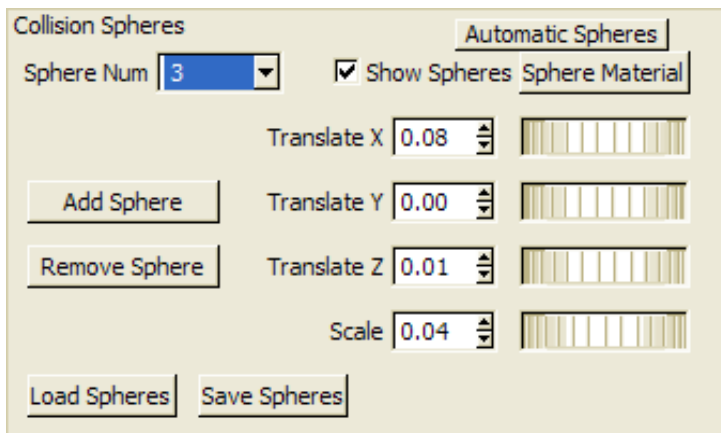


The above is an example of the display of collision points on an ArticulatedObject. The yellow dots indicate the location of the collision points.

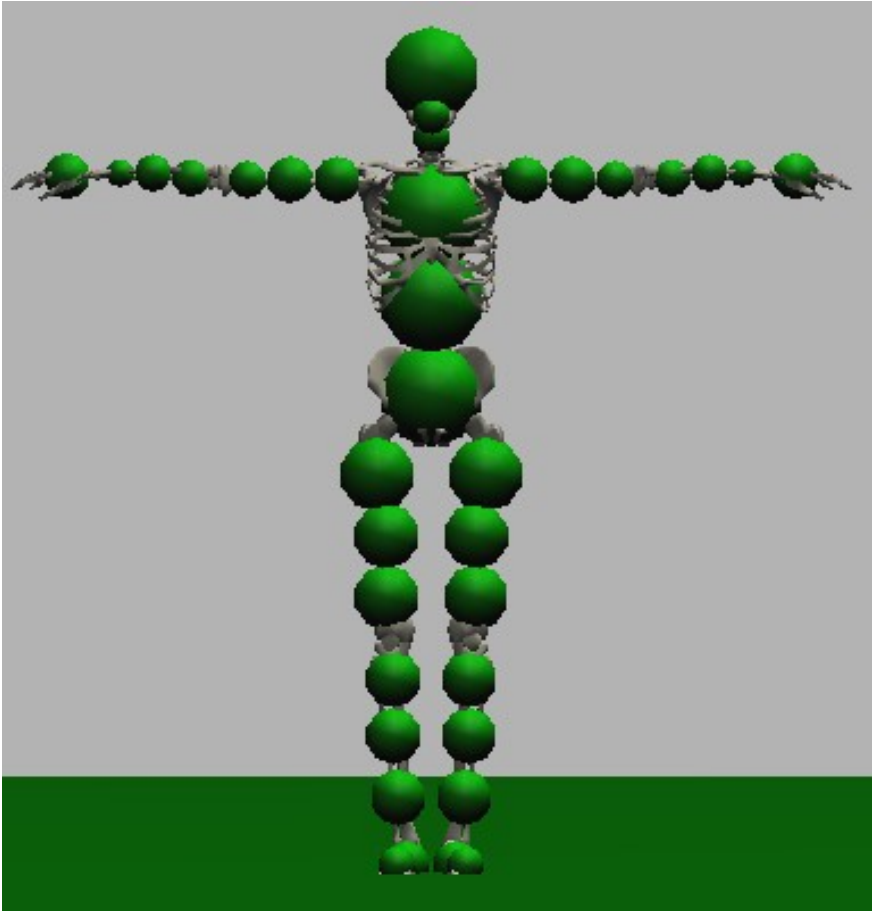
The PointPlaneCollision plugin shows an example of how to use the point collision information while developing with DANCE. This plugin manages point collisions between an ArticulatedObject and a Plane object (that represents the ground, for instance).

### 1.6.2 Sphere Collisions

Each link on an ArticulatedObject can have up to three collision spheres (this maximum may also be changed by altering the `MAXNUMSPHERES` variable in the source code). Each collision sphere may be translated and scaled relative to the center of the link.



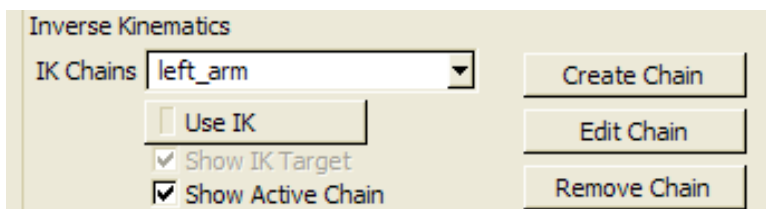
To see the collision spheres currently placed on the skeleton, check Show Spheres. To place a sphere on a skeleton, press 'Add Sphere', then translate and scale the sphere until it is in the proper position. Only one sphere may be manipulated at a time, so to move a different sphere on the same link, choose a different sphere from the Sphere Num dropdown. To remove a sphere, press 'Remove Sphere'. The spheres may be rendered with different materials by clicking the 'Sphere Material' button. To save the spheres to a .sphere file, press 'Save Sphere'. To restore spheres from a file, press 'Load Spheres'. The 'Automatic Spheres' button allows you to automatically place spheres on the ArticulatedObject. Pressing this button will remove all spheres currently assigned to each link and place one sphere in the middle of each link scaled according to the largest length of each link.



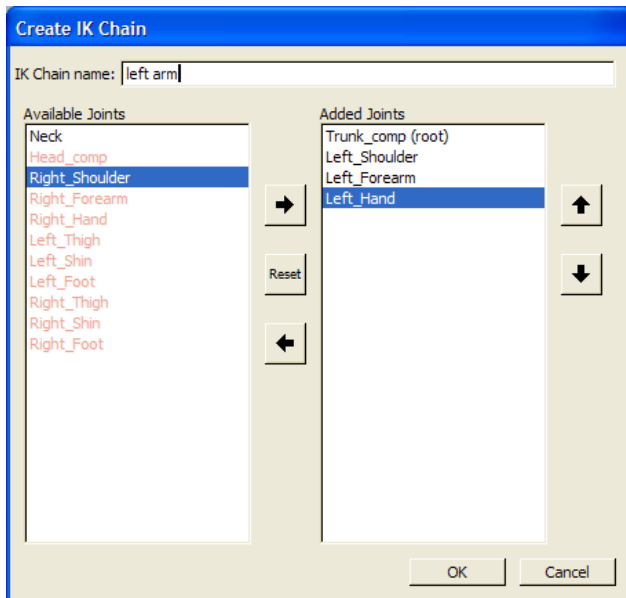
The SphereCollision plugin manages collision detection and resolution between ArticulatedObjects using their sphere collision information.

### 1.7 Using Inverse Kinematics

An ArticulatedObject can use inverse kinematics (IK) in order to position links of the skeleton. IK allows the user to manipulate many links at the same time, in contrast to the joint manipulation which only allows you to manipulate one DOF of one joint at a time.

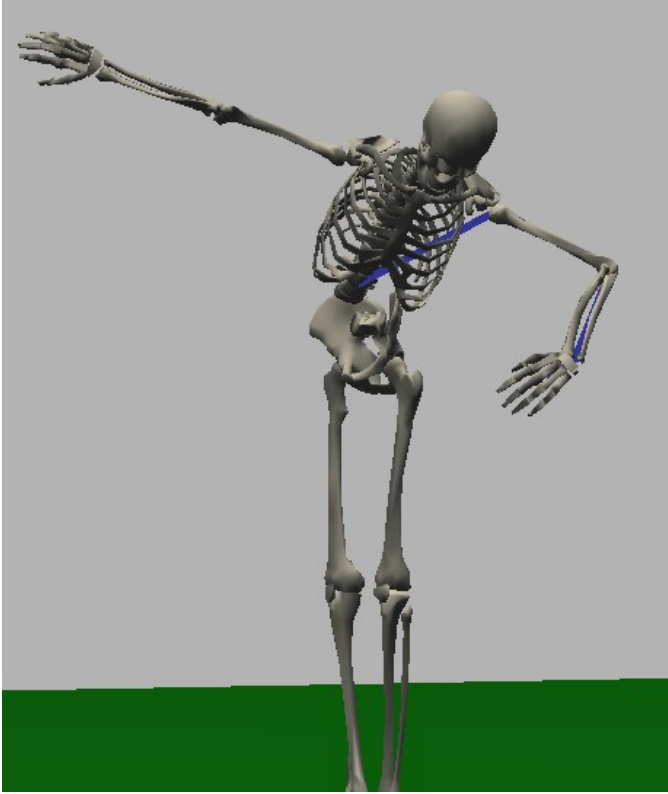


To use inverse kinematics, you must first create a chain of joints that will be used by the inverse kinematics algorithm. To do this, press the 'Create Chain' button. This will display a dialog that shows all the joints of the ArticulatedObject on the left, with the joints selected for this IK chain on the right. Once an initial joint is selected as the root of the chain, only joints that are directly connected to that joint will be shown on the left. Joints that cannot be selected as part of the chain will be listed in red.



After you name the IK chain, press the 'OK' button. To use that IK chain, select it in the IK Chains dropdown, then check Use IK. This will turn your mouse into the IK target when you place it into the main display window and press the left mouse button. As long as the Use IK checkbox is selected, your mouse will no longer work as a camera, but only as an IK target.

The mouse uses the screen X-Y coordinates and maps them to the 3-D coordinates in the virtual world. To move the IK chain in a direction that cannot be mapped from the current view, uncheck Use IK, reposition the camera so that the X-Y screen coordinates can be mapped to the desired 3-D world coordinate of the ArticulatedObject, then check Use IK once again and move the IK chain.



The picture above shows an example of IK in use. The blue line indicates the active chain.

### **1.6 Skinning**

An ArticulatedObject can be fit inside a mesh and deform that mesh according to the movement of the joints and links. This process is called 'skinning' and is implemented with the LinearSkinning plugin. Please see the description of that plugin for more details.

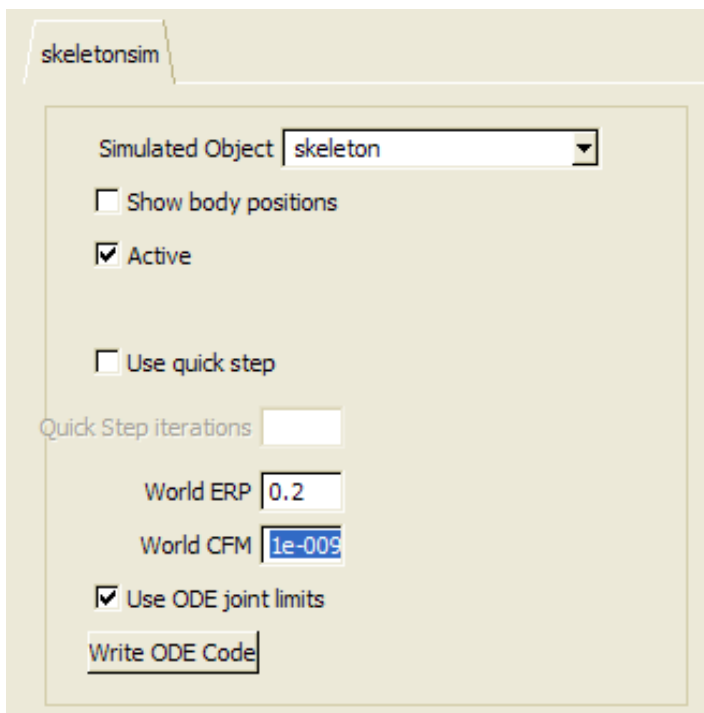
### **1.7 Props**

Props are geometries that are attached to an ArticulatedObject and move relative to the movement of the particular link that they are attached to. Any geometry type (Cubes, Spheres, Models, etc.) can be attached as a prop. To attach a geometry as a prop, position the geometry in the desired position, then select the appropriate joint/link from the joint drop down list. Under the Geometry parameters, select the geometry from the Prop drop down. To remove a prop, select 'No Prop' from the geometry drop down. Note that when a geometry is removed as a prop, it will return to the position it was when the prop was first attached as a prop.

## 2. ODESim

The ODESim plugin implements a rigid body simulator using the Open Dynamics Engine (<http://www.ode.org>). DANCE currently uses the 0.6 version of ODE. Each ODESim instance simulates one ArticulatedObject instance. Any ArticulatedObject can be simulated in an ODESim instance.

Note that ODE allows you to set joint limits through the use of joint stop parameters. This feature may be turned on or off. Multiple instances of ODESim will all simulate within the same ODE world.

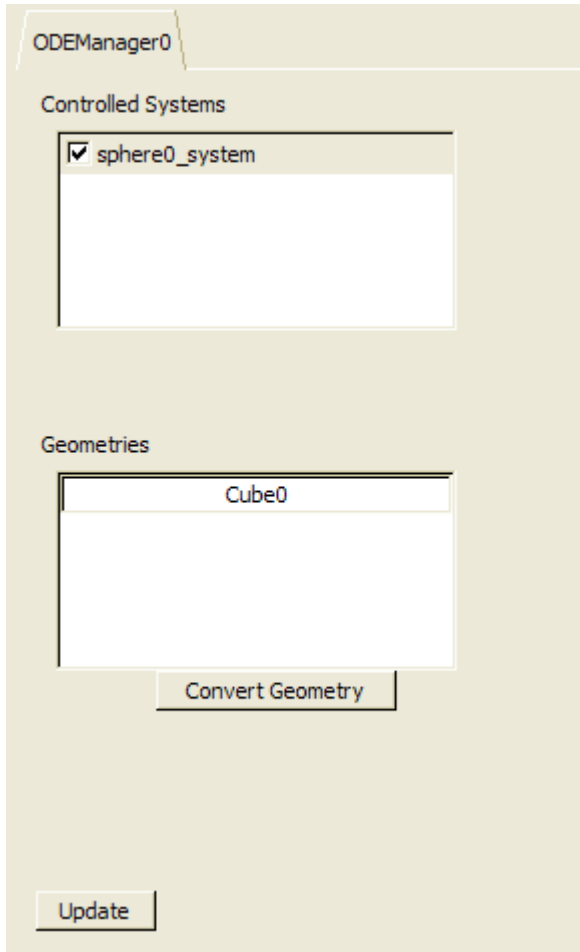


- **Simulated object** - choose the articulated object from the Simulated Object dropdown.
- **Show body positions** - display the position of the bodies as specified by the ODE simulators.
- **Active** - checkbox turns the simulator on or off. It can be used when multiple simulators are used on the same articulated object. For example, you might use a kinematic simulator to move the articulated object according to a specified path, and also use the ODESim on the articulated object to simulate it using dynamics.

- **Use quick step** - sets ODE's quick step intergrator.
- **Quick Step iterations** – set sets the ODE parameter of the same name.
- **World ERP** – ODE's error reduction parameter
- **World CFM** – ODE's constraint force mixing parameter
- **Use ODE joint limits** – sets lo and hi stops in ODE according to the joint limit parameters in Articulated Object.
- **Write ODE Code** – writes the Articulated Object topology and physical parameters to a .cpp file that contains code to create the same object using ODE without DANCE.

### 3. ODEManager

The ODEManager plugin allows you to quickly attach and create an ODESim simulator to any ArticulatedObject in the system or to simulate any geometry object that you have instantiated (cubes, spheres, models, etc.).



- **Controlled Systems** – shows a list of all ArticulatedObjects in DANCE. A checkbox next to the entry indicates that an ODESim instance is currently simulating that object. You can check and uncheck the value to add or remove an ODE simulator.
- **Convert Geometry** – shows a list of all DGeometry objects that are not simulated. To simulate that DGeometry instance, click on the name. An ArticulatedObject will be created for that object as well as an ODESim instance. This can be used to quickly create models in DANCE and assign them physical properties. To change the physical properties such as mass, inertial properties and so forth, select the <name>\_system object.



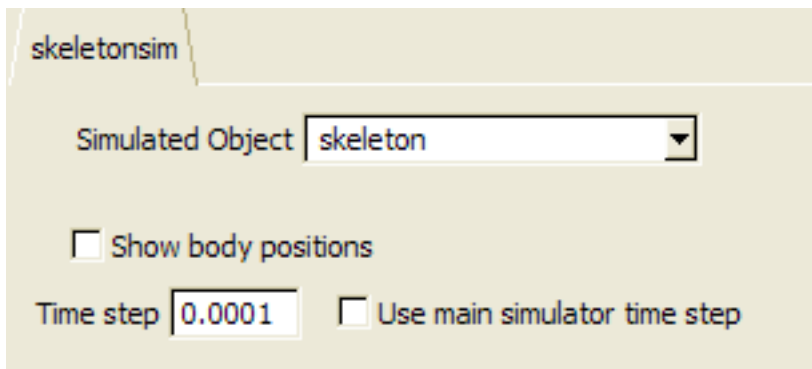


#### 4. SdfastSimul

The SdfastSimul plugin implements a rigid body simulator using the SD/Fast simulator (<http://www.sdfast.com>). Each SdfastSimul instance simulates one ArticulatedObject instance.

Unlike ODE, SD/Fast uses compiled code to simulate their rigid bodies. Thus, any skeleton that uses an SdfastSimul instance must be connected with another plugin that contains the SD/Fast simulation code. For example, the distribution includes the skel18 and robot skeletons which may be simulated with SdfastSimul. Any change in topology requires new SD/Fast code to be compiled, so a user would need an SD/Fast license to create an ArticulatedObject with a different topology. However, if you have an Sd/Fast license, the DANCE code will support any configuration.

SdfastSimul instances can either use the main simulator's time step, or their own.

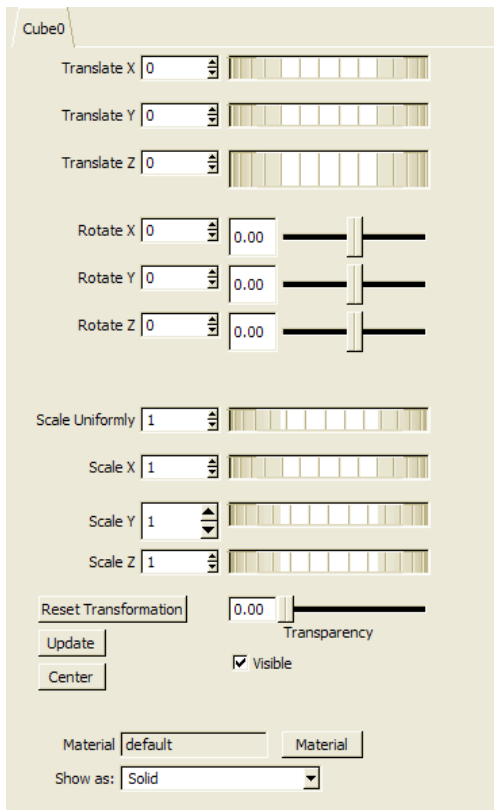


- **Simulated object** - choose the articulated object from the Simulated Object dropdown.
- **Show body positions** - display the position of the bodies as specified by the ODE simulators.
- **Use step fast** - sets ODE's step fast simulator.
- **Time step** – Time step for this simulator. This can be different from DANCE's main simulator time step for accuracy purposes.
- **Use main simulator time step** – use the time step from DANCE's main simulator instead of the one specified here.



## 5. Cube, Sphere, Plane, Capsule

The Cube, Sphere, Plane and Capsule plugins represent geometry instances that have visual representations. Each of these instances utilizes a transformation matrix.



- **Translate X, Y, Z** – translate the geometry by the given value.
- **Rotate X, Y, Z** – rotate the geometry around the object’s center by the given value.
- **Scale Uniformly** – scale the geometry in the X,Y & Z directions by the given value.
- **Scale X, Y, Z** – scale the geometry by the given value.
- **Reset Transformation** – resets the transformation matrix to the identity matrix.
- **Update** – refreshes this GUI.
- **Center** – places the center of the geometry’s vertices at the (0, 0, 0). By default, the object will be translated by an amount needed to place the center at (0,0,0). A popup dialog will ask if you wish to change the geometry’s vertices rather than to merely

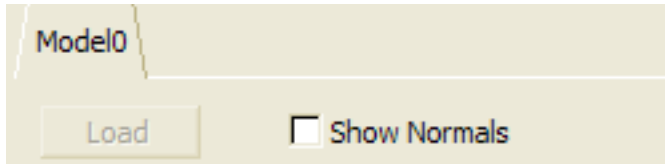
translate the object. If the geometry's vertices are changed, the object will be centered at (0, 0, 0) when the transformation matrix is reset to the identity matrix.

- **Material** – choose a material for the object.
- **Show As** – show the object as a solid (normal mode), as a mesh or as points.

Note that the Capsule plugin does not have scaling parameters. Instead, the Capule plugin has a length and a radius.

## 6. Model

The Model plugin represents a polygonal mesh and can be used to load 3D models from .3ds, .obj and .wrl files. Models are similar to other DGeometry types like spheres, cubes and planes, above.

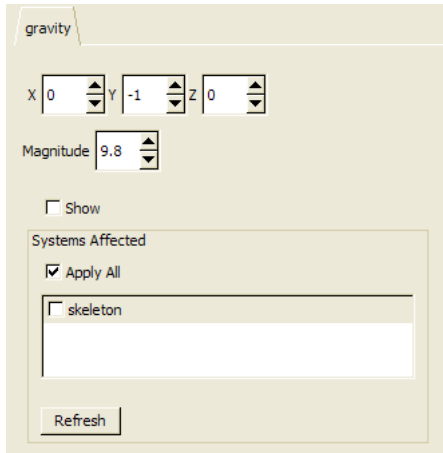


- **Load** – load the model from a .3ds, .obj or .vrmf file.
- **Show normals** – display the face normals for the model.

Please see the description for Cube, Sphere and Plane for details of the other aspects of the Model's GUI.

## 7. FieldActuator

The FieldActuator plugin represents forces that are applied to all Systems in the DANCE environment. A FieldActuator instance could represent gravity, wind, or viscous forces. Like all Actuators, the FieldActuator can be applied to all Systems or only to one specific system.



The screenshot shows a configuration window for a 'gravity' FieldActuator. It features three spinners for X, Y, and Z directions with values 0, -1, and 0 respectively. A 'Magnitude' spinner is set to 9.8. There is a 'Show' checkbox which is currently unchecked. Below this is a 'Systems Affected' section with a checked 'Apply All' option and a list box containing 'skeleton' with an unchecked checkbox. A 'Refresh' button is located at the bottom.

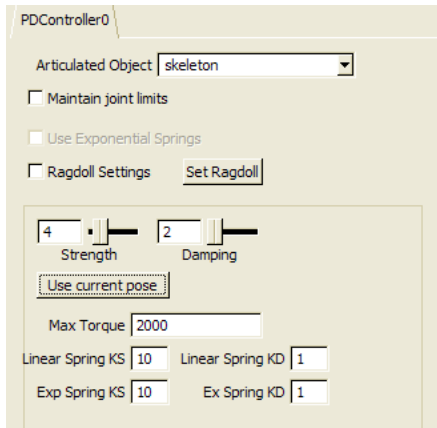
- **X, Y, Z** – direction of field force.
- **Magnitude** – magnitude of field force. Will be multiplied by direction to determine total force vector.
- **Show** – shows the field force direction as an arrow.
- **Systems Affected** – select Apply All to affect all systems. Alternatively, uncheck Apply All and choose the individual systems that will be affected by the field actuator's forces.

## **8. Linear Skinning**

The LinearSkinning plugin allows you to deform a mesh (Model instance) around a skeleton (ArticulatedObject) using linear blending. Up to five links of an ArticulatedObject can influence each vertex of a model. Press see Section IV of the DANCE manual for a complete description.

## 9. PDController

The PDController plugin is a proportional-derivative controller that can be used to maneuver ArticulatedObject instances under physical simulation. The PDController can also set enforce joint limits by correcting forces to the joints once they extend beyond their limits.



- **Articulated Object** – choose the articulated object to use PD control.
- **Maintain joint limits** – use torques to prevent joints from exceeding joint limits.
- **Use Exponential Springs** – uses exponential springs to prevent joints from exceeding joint limits.
- **Ragdoll settings** – set the ArticulatedObject into ragdoll mode, which uses the last set strength and damping value.
- **Set ragdoll** – sets the ragdoll settings to the strength and damping values currently displayed.
- **Strength** – strength settings. Corresponds to spring constant.
- **Damping** – damping settings. Corresponds to damping constant.
- **Use current pose** – sets the current pose of the articulated object as the pose that will be targeted via the PD controller.
- **Max torque** – maximum torque to be used.
- **Linear Spring KS/KD**– spring and damper constant for linear springs used for maintaining joint limits.

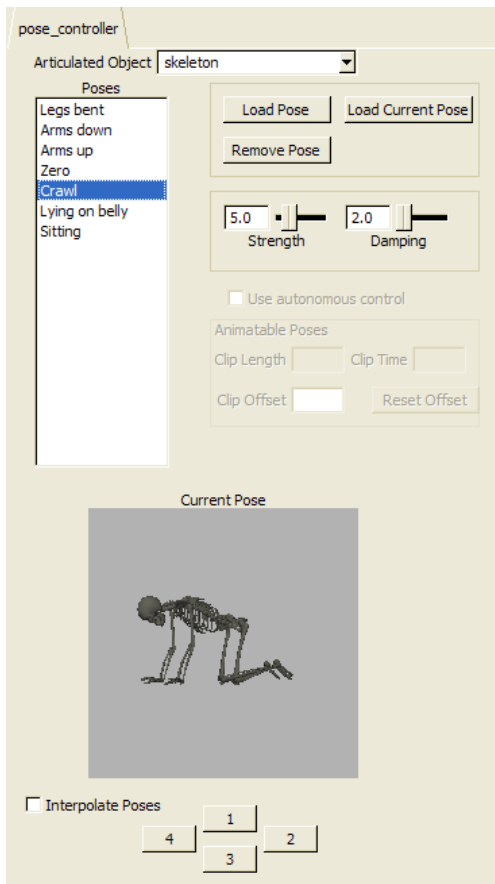


- **Exponential Spring KS/KD** – spring and damper constant for exponential springs used for maintaining joint limits.

## 10. PosePDController

The PosePDController plugin allows you to pose an ArticulatedObject then capture that pose as a target for a PD controller. This allows you to pose ArticulatedObjects under physical simulation. Note that the global translation and orientation are ignored for the purposes of PD control.

In addition, poses set in the PosePDController can be interpolated by using the pose dots. This is done by assigning a pose in the pose list to one of four positions in the pose view. After selecting 'Use Pose Dots', the user can drag the dots between two poses, interpolating between them.



- **Articulated Object** – choose the articulated object to use PD control.

- **Poses** – A list of poses used for PD control. Once a pose is selected under physical simulation, the ArticulatedObject will attempt to achieve that pose using the strength and damping values.
- **Load Pose** – Loads a pose to the pose list from a pose file. Poses can be loaded from a .state file (static poses) or a .bvh file that matches the topology of the ArticulatedObject (animatable poses). Animatable poses will not use PD control, but rather will be moved kinematically.
- **Load Current Pose** – Loads a pose to the pose list from the current pose of the ArticulatedObject. Typically, this is used by positioning the character via IK or other means and then loading that pose into the PosePDController with this button.
- **Remove Pose** – removes the currently selected pose from the pose list.

The format of the static pose file (.state) is the same as is generated from the ArticulatedObject/ Save Position feature as follows:

Joint1dof1 joint1dof 2 joint1dof3 ...

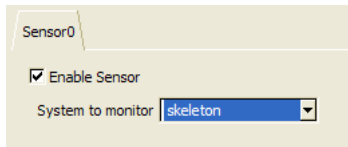
Joint2dof1 joint2dof2 joint2dof3 ...

Thus, the .state file will have as many lines as the ArticulatedObject has joints and as many total entries as the number of states in the ArticulatedObject. Each entry is a number (in radians) that represents the value of the state of that particular DOF.

Dynamic pose files are .bvh files with the same topology as the ArticulatedObject and are animated kinematically.

## 11. Sensor

This is the plugin that implements basic Sensor functioning for the use of controllers.



- **Enable Sensor** – turns the sensor on or off. Typically, sensors will perform some data checking during every time step. If unchecked, sensor checking will not be performed.
- **System to monitor** – a list of systems that can be monitored by this sensor. Once a system is chosen and the sensor is set to active, the sensor will be able to collect information from the environment.

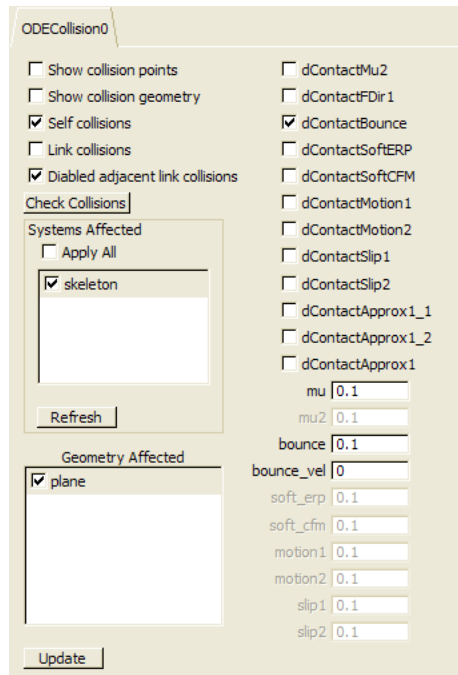
Please note that this plugin should be derived for each specific system.

## 12. **SensorSkel18**

This is a Sensor-derived plugin that implements sensors specifically for the Skel18 plugin. The interface is the same as for the Sensor plugin.

### 13. ODECollision

The ODECollision plugin manages contact between Systems that are simulated with ODE. It uses ODE's collision detection and resolution for the following geometries: spheres and cubes. Plane instances created in DANCE will be simulated as flat cubes.

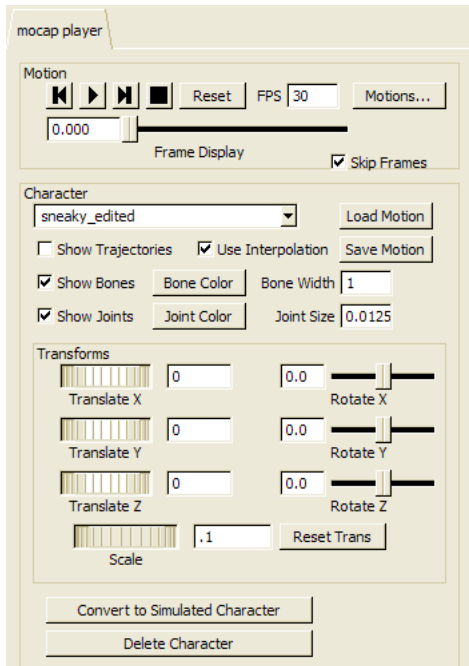


- **Show collision points** – displays the contact points of collision during simulation.
- **Show collision geometry** – displays geometry that ODE is using for collisions. Please note that the plugin will use the geometry of the ArticulatedObject in the following order: Collision geometry, collision spheres, geometry. If a sphere or cube is not found, then no geometry will be used for collisions. Trimesh is currently not supported..
- **Self collisions** – determines if Systems are allowed to collide with themselves.
- **Link collisions** – determines if geometry on the same link will collide with itself.
- **Disable adjacent link collisions** – determines if geometry on adjacent links (parent or child) will collide with each other.
- **Check collisions** – determines if objects are currently in collision.
- **Systems Affected** – choose which systems will collide with each other.

- **Geometry Affected** – choose which static geometry will be used as environmental collision objects.
- **dContactMu, dContactFDir, ...** - ODE parameters of the same name.

## 14. MotionPlayer

The MotionPlayer plugin allows you to load and save motion capture data (.bvh and .asf/.amc formats). In addition, the MotionPlayer allows you to convert motion capture characters to ArticulatedObjects so that they can be physically simulated. The motion capture character topology is copied, although physical parameters still need to be set manually (bone masses must be explicitly set, for example).



- **Load Motion** – loads a .bvh or .asf/.amc mocap file.
- **Save Motion** – saves the currently selected character to a .bvh file.
- **FPS** – speed of motion capture playback.
- **Motions...** – choose which motions of those already loaded will be displayed.
- **FPS** – speed of motion capture playback.
- **Reset** – sets time to zero for all motions.
- **Step back, play, step forward, stop buttons** – mocap playback.
- **Character** – determines which motion capture character is currently selected.
- **Translate** – translate the currently selected character's motion path.
- **Rotate** – rotate the currently selected character's motion path.

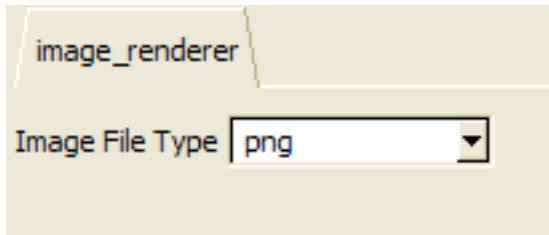


- **Reset Trans** – resets the transformation matrix that affects the path to the identity matrix (sets translation and rotation parameters to zero).
- **Show Trajectories** – shows the path of the root for the currently selected character.
- **Show Bones** – shows the currently selected character’s bones as lines.
- **Show Joints** – shows the currently selected character’s joints as spheres.
- **Bone Color** – sets the bone color of the currently selected character.
- **Joint Color** -sets the bone color of the currently selected character.
- **Bone Width** – sets the width of the bone as a percentage of the currently selected character’s height.
- **Joint Size** - sets the size of the joints as a percentage of the currently selected character’s height.
- **Convert To Simulated Character** – converts the currently selected mocap character into an ArticulatedObject with the same topology and proportions. The ArticulatedObject may then be simulated by first attaching a valid simulator (ODESim, for example).
- **Delete Character** – removes the currently selected character.

**Geometry Affected** – choose which static geometry will be used as

## 15. BitmapRenderer

The BitmapRenderer plugin allows the user to render the current screen to an image file. Please see the section on Rendering, above.



- **Image File Type** – choose a file type to render. Valid types are .png, .bmp, .jpg, .gif.

For more rendering options, select the Render menu item.

## **16. POVRayRenderer**

The POVRayRenderer plugin allows the user to render the current screen to a POVRay .pov file. Please see the section on Rendering, above.

## III. DANCE Plugin Development

### 1. Plugin Overview

DANCE is a software environment for graphics and animation. The DANCE platform allows you to build plugins that leverage the core functionality of DANCE. A plugin written for DANCE can leverage the following capabilities of the core platform:

1. **On screen rendering** – each plugin can have a visual representation in the DANCE environment using OpenGL.
2. **Scripting interface** – each plugin can be controlled through scripting commands. DANCE uses Python ([www.python.org](http://www.python.org)) as a scripting language. The plugin commands are embedded into the Python language.
3. **Off screen rendering** – each plugin can be rendered to an image file (.bmp) or using a raytracer (POVRay, [www.povray.org](http://www.povray.org)).
4. **Session persistence** – each plugin can be saved and loaded to a file, saving its state between different sessions.
5. **GUI Interface** – each plugin can have its own widget interface that allows the user to interactively control various parameters of the plugin. There is no limit to the complexity of the interface. DANCE uses FLTK ([www.fltk.org](http://www.fltk.org)) as a widget set.
6. **Interaction Controls** – each plugin can map its own set of keyboard and mouse controls.
7. **Dependency Management** – DANCE manages each plugin's dependencies and insures that plugins use valid references. Plugins can be created and destroyed without causing other plugins dependent upon those instances from crashing the system, since each connection between plugins is carefully controlled.
8. **Interoperability** – DANCE plugins can interoperate with other DANCE plugins. For example, Geometry plugins written in DANCE will function correctly with any other plugin that operates with a DANCE Geometry type. Objects can be added and removed from the DANCE system.

9. **Event Management** – DANCE plugins can respond to events in the system, such as simulator events (simulator begins, simulator takes a step, simulator ends), plugin events (object is created, object is destroyed (NOT YET IMPLEMENTED)) and interaction events (object is in collision (NOT YET IMPLEMENTED)).
  
10. **Plugin Primitives** – DANCE provides an infrastructure for creating primitive objects in a graphical environment: Geometries (spheres, cubes, meshes, etc) have graphical representations and topologies; Systems (articulated bodies, cloth, particle sets, etc.) allow objects to be organized semantically meaningful way, maintain state and can be animated; Simulators (rigid body simulator, cloth simulator, particle simulator) process and animate Systems over time. Each plugin primitive contains built-in functionality that aids the developer in creating graphical animations.

## **2. Software Requirements**

DANCE plugins can be developed on any platform that DANCE supports, including Windows, linux and OSX.

## 3. Plugin Creation

### 3.1 Environment Variables

You must set the following environment variables before compiling your plugins:

```
DANCE_DIR = /path/to/DANCE/installation
```

Make sure that you use forward slashes (/) and not backslashes (\) For example, if your DANCE installation is located in c:\dance\_v4, the value of your DANCE\_DIR variable will be:

```
c:/dance_v4
```

### 3.2 Project Setup

#### Linux

Please refer to the Makefile that accompanies the Sample plugin.

#### Windows

DANCE plugins require particular project settings. Open the dance\_v4.sln solution file. This contains all the DANCE plugins. Add a new project of type “Empty Project (.NET)”. For the purposes of illustration, we will name it Sample, but you may name it anything you want. Substitute “Sample” in the remainder of this section for the name of your plugin. Add a Sample.h and a Sample.cpp file to the project. Make the following changes to the project under Project-Properties:

#### General

**Configuration type:** Dynamic Library (.dll)

**Use Managed Extensions:** No

#### C/C++

**General/Additional Include Directories:** \$(DANCE\_DIR)/core/src;\$(DANCE\_DIR)/core/math;\$(DANCE\_DIR)/libinclude;\$(FLTK\_DIR)

**Preprocessor/Preprocessor Definitions:**add 'FL\_SHARED'

**Code Generation/Runtime Library:** Multi-threaded Debug DLL (/MDd)

**Language/Enable Run-Time Type Info:** Yes (/GR)

**Advanced/Compile As:** Default

Linker

**General/Output File:** \$(DANCE\_DIR)/plugins/win/ Sample\_d.dll

**Additional Library Directories:** \$(DANCE\_DIR)/lib/win

**Linker/Input:** fltkdll.lib dance\_d.lib dancemath\_d.lib opengl32.lib

**Advanced/Import Library:** \$(DANCE\_DIR)/lib/win/ Sample d.lib

The above configuration is to compile plugins in Debug mode. If the plugin is compiled in Release mode, make the above project changes except for:

C/C++

**Code Generation/Runtime Library:** Multi-threaded DLL (/MD)

Linker

**General/Output File:** \$(DANCE\_DIR)/plugins/win/ Sample.dll

**Advanced/Import Library:** \$(DANCE\_DIR)/lib/win/ Sample.lib

Notice that the 'd' extensions on the .dll and import library have been dropped.

### 3.3 Basic PlugIn Code

Place the following code in your Sample.h file:

```
// Sample.h  
  
#ifndef _SAMPLE_H  
  
#define _SAMPLE_H  
  
#include "PlugIn.h"
```



```
class Sample : public PlugIn
{
    public:
        PlugIn* create(int argc, char **argv);

        Sample();
};

#endif
```

Add the following to the Sample.cpp file:

```
// Sample.cpp

#include "Sample.h"
#include "dance.h"
#include "danceInterp.h"

PlugIn* Proxy()
{
    return new Sample();
}

PlugIn* Sample::create(int argc, char **argv)
{
    Sample* s = new Sample() ;

    return s;
}
```

```
Sample::Sample() : PlugIn()  
  
{  
  
    danceInterp::OutputMessage("Sample has been created!");  
  
}
```

The above code is the minimal amount of code needed to create a plugin. The following sections will outline how to run the plugin in DANCE and how to add each of the plugin features listed in the first section.

### 3.4 Running the Plugin

Build the project. The project should generate a file 'Sample\_d.dll' (or Sample.so in Linux) in DANCE\_DIR/plugins/win (DANCE\_DIR/plugins/linux for Linux) and an import library in DANCE\_DIR/lib/win (DANCE\_DIR/lib/linux for Linux).

#### Windows

Run the DANCE program (make sure that the **dance\_v4** project has been 'Set as Start-up Project').

#### Linux

Run the DANCE program by:

```
$(DANCE_DIR)/bin/dance
```

Note that you might have to set the LD\_LIBRARY\_PATH if it has not already been done:

Using bash, sh or ksh:

```
export LD_LIBRARY_PATH=$DANCE_DIR/lib
```

Using csh:

```
setenv LD_LIBRARY_PATH $DANCE_DIR/lib
```

Under the 'Plugins' menu, choose: Create. Scroll down until you find the plugin named 'Sample'. Click on this plugin, type in a name for the instance (Sample0, by default) and press 'Create Instance'.

The plugin will be created. The text "Sample has been created!" will appear in the command shell window.

To delete the plugin, click on the 'Select' button in the main window on the upper right. Select 'Sample0' and press the 'Delete' button.

## 4. Plugin Functionality

The following section describes the functionality that can be added to the plugins. Each function (such as onscreen rendering, persistence, etc.) is implemented by overriding methods that have been implemented in the parent class. Those methods and a summary of how to override them is listed below.

### 4.1 On Screen Rendering

DANCE plugins can be rendered on screen using OpenGL. There is no restriction as to what OpenGL functions can be called. Functions that can be called are only limited by your graphics card and OpenGL implementation on your system. To produce an onscreen rendering, override the following method:

```
void output(int mode);
```

Here is an example implementation:

In Sample.h:

```
public:
    void setColor(float r, float g, float b);
    float* getColor();

private:
    float color[3];
```

In Sample.cpp:

```
#include <FL/gl.h>
```

```
Sample::Sample() : PlugIn()
```

```
{  
  
    danceInterp::OutputMessage("Sample has been created!");  
  
    setColor(1.0, 0.0, 0.0); // set the initial color to red  
  
}  
  
void Sample::setColor(float r, float g, float b)  
{  
  
    color[0] = r;  
  
    color[1] = g;  
  
    color[2] = b;  
  
}  
  
float* Sample::getColor()  
{  
  
    return &color;  
  
}  
  
void Sample::output(int mode)  
{  
  
    glPushAttrib(GL_ENABLE_BIT);  
  
  
    glDisable(GL_LIGHTING);  
  
    glColor3f(color[0], color[1], color[2]);  
  
    glBegin(GL_LINE_LOOP);  
  
    glVertex3f(0.0, 0.0, 0.0);  
  
    glVertex3f(0.0, 1.0, 0.0);  
  
    glVertex3f(1.0, 1.0, 0.0);  
  
    glVertex3f(1.0, 0.0, 0.0);  
  
}
```

```
glEnd();

glPopAttrib();
}
```

Note that FLTK includes its own OpenGL wrappers and you should specify `<FL/gl.h>` and not `<GL/gl.h>` in the header. Also, if you wish to include GLUT functionality, include `<FL/glut.h>` instead of `<GL/glut.h>` and include `glut32.lib` in the Linker after the `fltkdll.lib` library.

The above code will draw a red rectangle in the DANCE window. Note that care must be taken when enabling or disabling OpenGL parameters, such as colors and lights. Each plugin is responsible for maintaining its own parameters, but DANCE does not ensure that each plugin properly sets and resets its parameters. It is possible, for example, for a plugin to change the OpenGL drawing color without resetting the color, and another plugin uses that newly set color.

## 4.2 Scripting Interface

DANCE plugins can be scripted through the command shell. Commands can be saved in text files. DANCE's scripting language is Python. DANCE commands and plugin access is connected to the Python by embedding DANCE objects. Typically, a command can be sent to a DANCE plugin by entering the following Python command in the command shell:

```
dance.generic("Sample0", "color", "1", "0", ".5")
```

This sends to the generic plugin named 'Sample0' the command 'color' with three parameters: '1', '0' and '.5'.

To give scripting ability to a plugin, override the following method:

```
int commandPlugIn(int argc, char **argv);
```

This method takes in a number of command arguments (specified by argc) and stored in argv. Using the above example, argc will be 4, with argv[0] = "color", argv[1] = "1", argv[2] = "0" and argv[3] = ".5".

This method returns one of the following:

DANCE\_OK if the command was handled properly.

DANCE\_ERROR if there is a problem with the command.

DANCE\_CONTINUE if the command is not handled by this plugin.

Since each plugin can leverage the abilities of its parent classes, it is good form to allow the parent classes to handle the commands first. If the commands are handled in the parent class, then the current plugin will not have to handle the command.

Here is an example implementation in Sample.cpp:

```
int Sample::commandPlugIn(int argc, char **argv)
{
    // let the parent class (PlugIn) handle the commands first
    int ret = PlugIn::commandPlugIn(argc, argv);
    if (ret == DANCE_OK || ret == DANCE_ERROR)
        return ret;

    if (strcmp(argv[0], "color") == 0)
    {
        if (argc < 4)
        {
            danceInterp::OutputMessage("%s 'color' parameter
requires 3 parameters.", this->getName());
        }
    }
}
```

```

        return DANCE_ERROR;
    }
    else
    {
        setColor(atof(argv[1]), atof(argv[2]), atof(argv[3]));

        danceInterp::OutputMessage("%s 'color' has been set
to %4.2f, %4.2f, %4.2f.", this->getName(), color[0], color[1],
color[2]);

        dance::Refresh();

        return DANCE_OK;
    }
}

return DANCE_CONTINUE; // command was not found
}

```

Note that numeric values do not need to be quoted when passing commands to the scripting interface. They will be automatically transformed into strings before being sent to the `commandPlugin()` method. For example:

```
dance.generic("Sample0", "color", "1", "0", ".5")
```

is identical to:

```
dance.generic("Sample0", "color", 1, 0, .5)
```

#### 4.21 Calling DANCE Commands From the Source Code

DANCE commands can be called from source code through the interpreter by calling:



```
int danceInterp::ExecuteCommand(char *command);
```

The function will return DANCE\_OK, DANCE\_CONTINUE or DANCE\_ERROR. For example:

```
int ret =  
danceInterp::ExecuteCommand("dance.geometry(\"cube\", \"translate\", 0,  
1, 0)");
```

will load translate a geometry object named 'cube' by one unit in the y-direction. Note that calling Python scripting commands is slower than accessing the methods directly from the plugin instance, since the commands must be first sent to the Python interpreter before being executed in DANCE. For faster access, use the methods directly from the plugin object (see section 4.8, below). Using the same example as given above:

```
Cube* cube = (Cube*) dance::AllGeometry->get("cube");  
cube->Translate(0, 1, 0);
```

### 4.3 Off Screen Rendering

DANCE plugins can be rendered automatically to an image file (.bmp) using the BitmapRenderer. However, plugins can also be rendered to a POV-Ray file. To do so, each plugin must override the following method:

```
void render(int argc, char** argv, std::ofstream& file);
```

This method gets passes a set of parameters indicating rendering options (argc and argv) and an open file where POV-Ray commands can be written. To write the POV-Ray commands, simply write to the stream called file:

```
void Sample::render(int argc, char** argv, std::ofstream& file)  
{  
    file << "polygon {\n";
```

```

file << " 4\n";

file << " <0.0, 0.0, 0.0>\n";

file << " <0.0, 1.0, 0.0>\n";

file << " <1.0, 1.0, 0.0>\n";

file << " <1.0, 0.0, 0.0>\n";

file << " pigment { color rgb <" << color[0] << " " << color[1]
<< " " << color[2] << ">}\n";

file << "}\n";

}

```

To render a scene in DANCE, select the Render menu item, Render... and then choose either the bmp renderer or the pov renderer. The bmp renderer will capture the screen and place it into an image file. the pov renderer will create the following files:

**custommaterials.inc** - contains all materials used in DANCE

**lights.inc** – lighting information

**000001.pov** – povray frame (frame number will change)

The scene can be rendered by running the 000001.pov file in POVray. Either custommaterials.inc or lights.inc can be used to add global parameters (such as radiosity).

#### 4.4 Session Persistence

Each DANCE session, a user can persist (save to disk) their entire environment for later recreation. DANCE plugins are saved in a session and loaded through the use of the scripting interface. Each plugin state should implement its own state setting commands through its scripting interface (commandPlugIn()). All of the object's state is written to a file containing Python scripting commands. When that file is loaded, a new plugin object is created, the scripting commands are run to load the old state into the new object. The sessions are stored in a file with a .dpy extension (DANCE Python).

The session saving procedure occurs in three steps, called modes. The first mode instantiates the plugin. The second mode runs plugin scripting commands that are not dependent upon any other plugins. The third mode runs scripting commands that are dependent on the existence of other plugins.

The following methods must be overridden to implement session persistence:

```
int getNumPluginDependents();  
char* getPluginDependent(int num);  
void save(int mode, std::ofstream& file);
```

The first two methods indicate which Plugins this plugin depends upon. The first returns the number of dependent plugins, the second returns a string containing the name of the dependent plugins. Plugins that do not rely on any other plugins can ignore these methods.

The following is an example save() method:

```
void Sample::save(int mode, std::ofstream& file)  
{  
    char buff[512];  
  
    if (mode == 0)  
    {  
        file << "dance.instance(\"Sample\", \"" << this->getName()  
<< "\")" << std::endl;  
    }  
  
    else if (mode == 1)  
    {  
        Plugin::save(mode, file); // let the parent class handle  
        saving in this mode  
    }  
}
```

```

        // set the color

        sprintf(buff, "\"color\\", %4.2f, %4.2f, %4.3f", color[0],
color[1], color[2]);

        pythonSave(file, buff);

    }

    else if (mode == 2)

    {

        // add any commands that rely upon other plugins

        PlugIn::save(mode, file); // let the parent class handle
saving in this mode

    }

}

```

If this plugin was dependent on both the Cube and Sphere plugins, for example, the other methods would be implemented as follows:

```

int getNumPluginDependents()

{

    return 2;

}

char* getPluginDependent(int num)

{

    if (num == 0)

        return "Cube";

    else if (num == 1)

        return "Sphere";

    else

        return NULL;

}

```

```
}
```

These methods ensure that plugins are created in their proper order.

#### 4.5 GUI Interface

Each plugin can be created with its own set of GUI widgets. DANCE uses the FLTK ([www.fltk.org](http://www.fltk.org)) toolset for GUI interfaces. FLTK has widgets for buttons, checkboxes, sliders, dropdown lists, dials, file browsers and many other widgets. In order to create a GUI for your plugin, the following method needs to be overridden:

```
fltk::Widget* getInterface();
```

This method returns an FLTK Widget of any type. Typically, this widget is a container (of type `fltk::Group`) which holds many widgets. Any widget that is returned The following is an example implementation of a GUI for our Sample plugin:

Add to Sample.h:

```
#include <fltk/Widget.h>

class SampleWindow;

public:
    ~Sample();

private:
    SampleWindow* gui;
```

Add to Sample.cpp:

```
#include "SampleWindow.h"

Sample::Sample() : PlugIn()
{
    danceInterp::OutputMessage("Sample has been created!");
    color[0] = 1.0; // set the initial color to red
    color[1] = 0.0;
    color[2] = 0.0;

    gui = NULL;
}

Sample::~Sample()
{
    if (gui != NULL)
        delete gui;
}

fltk::Widget* Sample::getInterface()
{
    if (gui == NULL)
    {
        gui = new SampleWindow(0, 0, 300, 400, this->getName());
    }

    return gui;
}
```

```
}
```

Add the file SampleWindow.h:

```
#ifndef _SAMPLEWINDOW_  
  
#define _SAMPLEWINDOW_  
  
#include <fltk/Group.h>  
#include <fltk/ColorChooser.h>  
#include "Sample.h"  
  
class SampleWindow : public fltk::Group  
{  
    public:  
        SampleWindow(Sample* s, int x, int y, int w, int h, char*  
name);  
  
        void show();  
        void updateGUI();  
  
    private:  
        static void ChangeColorCB(fltk::Widget* widget, void* data);  
  
        fltk::ColorChooser* chooserColor;  
        Sample* sample;  
};  
  
#endif
```

Add the file SampleWindow.cpp:

```
#include "SampleWindow.h"

#include <fltk/Color.h>

#include "dance.h"

using namespace fltk;

SampleWindow::SampleWindow(Sample* s, int x, int y, int w, int h, char*
name) : Group(x, y, w, h, name)
{
    sample = s;

    this->begin();

    chooserColor = new ColorChooser(10, 20, 200, 200, "Polygon
Color");

    chooserColor->callback(ChangeColorCB, this);

    this->end();

    this->updateGUI();
}

void SampleWindow::show()
{
    this->updateGUI();

    Group::show();
}
```



```

void SampleWindow::updateGUI()
{
    float* color = this->sample->getColor();
    this->chooserColor->rgb(color[0], color[1], color[2]);
}

void SampleWindow::ChangeColorCB(fltk::Widget* widget, void* data)
{
    SampleWindow* win = (SampleWindow*) data;
    win->sample->setColor(win->chooserColor->r(), win->chooserColor-
>g(), win->chooserColor->b());
    win->updateGUI();
    dance::Refresh();
}

```

As a coding guideline, the authors recommend that a MVC (Model/View/Controller) format is followed for GUI widgets. There is no restriction on how the widgets may be created, however MVC architecture works well here. The plugin (Sample) is the model and contains all methods needed to manipulate its state or perform actions. The view (output() method of Sample) provides a GUI showing the choices. The controller (SampleWindow) allows the user to modify the plugin.

#### 4.6 Interaction Controls

Each plugin can have its own set of keyboard and mouse controls which can be turned on and off as desired. DANCE uses FLTK's event system to handle mouse and keyboard events. The following methods should be overridden:

```

int interact(Event* event);

int interactStart(Event* event);

```

```
int interactEnd(Event* event);
```

The first method is called every time an FLTK event occurs (such as a mouse move, mouse click, keyboard click, and so forth). The method should return -1 if the event is handled, or anything else if the event is not handled. If the event is not handled, it will be sent to the next object in the interaction stack. If there are no objects in the interaction stack, the event will be handled by the main DANCE system. The methods `interactStart()` and `interactEnd()` are called when the interaction with that plugin instance starts and ends.

DANCE maintains an interaction stack that keeps track of which objects require interaction. To add and remove objects from the interaction stack, the following method must be called:

```
void dance::addInteraction(DObject* obj)
void dance::removeInteraction(DObject* obj)
```

Objects may also be added to or removed from the interaction stack through the Python scripting commands:

```
dance.addInteraction("Sample0")
dance.removeInteraction("Sample0")
```

Where "Sample0" is the name of the object instance. An object can be placed on the interaction stack through the DANCE window by selecting the menu item 'Plugins' and choosing 'Interactions...'.

The following is an example implementation of an object's interaction with the keyboard and mouse:

In `Sample.h`:

```
int interact(Event* event);  
int interactStart(Event* event);  
int interactEnd(Event* event);
```

In Sample.cpp:

```
int Sample::interact(Event* event)  
{  
    double x, y;  
    int button;  
  
    switch (event->getEventType())  
    {  
        case fltk::KEY:  
            switch (fltk::event_key())  
            {  
                case 'r':  
                    this->setColor(1.0, 0.0, 0.0);  
                    danceInterp::OutputMessage("Color set to  
red.");  
                    dance::Refresh();  
                    if (this->gui != NULL)  
                        this->gui->updateGUI();  
                    break;  
                case 'g':  
                    this->setColor(0.0, 1.0, 0.0);  
                    danceInterp::OutputMessage("Color set to  
green.");  
                    dance::Refresh();
```

```

        if (this->gui != NULL)
            this->gui->updateGUI();

        break;

    case 'b':

        this->setColor(0.0, 0.0, 1.0);

        danceInterp::OutputMessage("Color set to
blue.");

        dance::Refresh();

        if (this->gui != NULL)
            this->gui->updateGUI();

        break;

    default:

        danceInterp::OutputMessage("Keyboard
button %c was pushed.", fltk::event_key());

        break;

    }

    return -1;

    break;

case fltk::PUSH:

    button = fltk::event_button();

    x = (float) fltk::event_x();

    y = (float) fltk::event_x();

    danceInterp::OutputMessage("Mouse button %d was
pushed at (%f, %f)", button, x, y);

    return -1;

    break;

case fltk::RELEASE:

    danceInterp::OutputMessage("Mouse button was
released...");

```

```

        return -1;

        break;

    case fltk::DRAG:

        danceInterp::OutputMessage("Mouse was dragged...");

        return -1;

        break;

    default:

        break;

}

return 0;
}

```

#### 4.7 Dependency Management

DANCE plugin objects may use other DANCE plugin objects. However, it is possible to remove and change DANCE objects via code or the GUI interface. By registering their dependents in the DANCE system, DANCE notifies the instances when their dependent objects have been removed from the DANCE system. This is critical when, for example, a rigid body plugin is dependent upon a Cube which represents its geometric boundaries. If the Cube is removed from DANCE, the rigid body plugin will be notified of this event.

To add the dependency to the object instance call:

```
void addDependency(DObject* object);
```

The DObject parameter is the object that this instance will depend upon. If that object is ever removed from DANCE, the following method on the first instance will be called, which should be overridden:

```
void onDependencyRemoval(DObject* object);
```

Within that method, code should undo any connections to the dependent instance.

#### 4.8 Interoperability

DANCE plugin instances can access other DANCE plugin instances. To access another object present in the DANCE system, use the following objects that are accessible from the 'dance' static instance:

```
static DObjectList *AllGenericPlugins;  
static DObjectList *AllActuators;  
static DObjectList *AllLights;  
static DObjectList *AllSystems;  
static DObjectList *AllGeometry;  
static DObjectList *AllModifiers;  
static DObjectList *AllRenderers;  
static ViewManager *AllViews;  
static PlugInManager *AllPlugIns;
```

For example, to access a light named 'mylight':

```
DObject* object = dance::AllLights->get("mylight");  
DLight* light = (DLight*) object;
```

Now, all methods of the DLight class can be used. Instances all derive from the base type DObject, and must be cast to their actual object types. Dynamic casting can also be used to determine what kind of class the object is:

```
if (dynamic_cast<Sample*>(object) != NULL)
```

```
{  
  
    // object is of type 'Sample'  
  
}
```

All objects that have been instantiated in plugin code should be added to one of the DANCE lists of the appropriate category (lights, views, generic, etc.) with the following method:

```
bool DObjectList::add(DObject *obj);
```

#### 4.9 Event Management

DANCE manages events and notifies plugin instances that are interested in these events. The events that may be monitored are: Simulator Starts, Simulator Ends, Before Simulator Step, After Simulator Step, On Simulator Step, Record State event (after each simulator step). These methods can be accessed through the `dance::AllSimulators` static instance:

```
void addSimStartCB(DObject* data, int priority, void  
(*stepcb)(DObject* data, double time));  
  
void addSimStopCB(DObject* data, int priority, void  
(*stepcb)(DObject* data, double time));  
  
void addBeforeSimStepCB(DObject* data, int priority, void  
(*stepcb)(DObject* data, double time));  
  
void addAfterSimStepCB(DObject* data, int priority, void  
(*stepcb)(DObject* data, double time));  
  
void addSimStepCB(DObject* data, int priority, void  
(*stepcb)(DObject* data, double time));  
  
void addSimEndCB(DObject* data, int priority, void  
(*stepcb)(DObject* data, double time));  
  
void addRecordStateCB(DObject* data, int priority, void  
(*stepcb)(DObject* data, double time));  
  
void removeAllCallbacks();  
  
void removeAllCallbacks(DObject* object);
```

```
void removeSimStartCB(DObject* object);  
void removeSimStopCB(DObject* object);  
void removeBeforeSimStepCB(DObject* object);  
void removeAfterSimStepCB(DObject* object);  
void removeSimStepCB(DObject* object);  
void removeSimEndCB(DObject* object);  
void removeRecordStateCB(DObject* object);
```

The priority represents the priority of the callback (lower priorities go first). The caller must implement a static function that handles the callback. This static function can in turn call a method on the specific instance.

For example, to add a simulator callback:

In Sample.h:

```
private:  
    void step(double time);  
    static void simstep(DObject* obj, double time);
```

In Sample.cpp:

```
Sample::Sample() : PlugIn()  
{  
    danceInterp::OutputMessage("Sample has been created!");  
    setColor(1.0, 0.0, 0.0); // set the initial color to red  
    gui = NULL;
```



```

        dance::AllSimulators->addSimStepCB(this, 100, simstep);
    }

void Sample::step(double time)
{
    color[0] = sin(time);
    color[1] = cos(time);
}

void Sample::simstep(DObject* obj, double time)
{
    Sample* sample = (Sample*) obj;
    sample->step(time);
}

```

Run the simulator by pressing the simulator play button. The polygon should slowly change colors as the simulator runs.

Plugins that wish to run during idle time (and not during the simulator's time) can use FLTK's idle callback routines:

```

    fltk::add_idle(TimeoutHandler, void* = 0);
    fltk::remove_idle(TimeoutHandler, void* = 0);

```

#### 4.10 Plugin Primitives

DANCE plugins can be written by extending the basic plugin primitives. Each primitive implements the basic DObject and PlugIn classes, but also includes specialized functionality. For example, plugins that use the DGeometry primitive have access to a transformation matrix, collision points to monitor on the geometry, affine transformation method and material settings. DSimulators contain the basic structure for creating dynamic simulators. The primitives

can be used by creating a class that extends the primitive instead of `PlugIn`. For more details, please consult the dance source code located in `dance/core/src`.

## 5.0 Accessing DANCE Objects

In order to access DANCE objects and classes already defined in the system, your plugin project must include the appropriate class and library definitions. In addition, your project must include Windows export information, since the plugins are accessed via Windows DLLs.

### 5.1 Project Settings To Access DANCE Classes

#### Linux

The .h files and .so plugins must be included in the compiler includes and linker dependencies respectively.

#### Windows

In order to connect to other objects in the DANCE system, your plugin project must compile in the .h headers (in the C/C++ / Additional Include Directories) for the plugin as well as include the .lib library (in the Linker / Additional Dependencies, also add \$(DANCE\_DIR)/lib/win to the Linker / General / Additional Library Directories).

For example, to use the Sphere plugin, add the following to the project properties:

C/C++

**General/Additional Include Directories:** \$(DANCE\_DIR)/geometries/Sphere

Linke

**General / Additional Library Directories:** \$(DANCE\_DIR)/lib/win

**General / Input:** Sphered.lib

Note that debug libraries are named <plugin>d.lib, while release mode libraries are named <plugin>.lib.

### 5.2 Accessing DANCE Classes through Windows DLLS

#### Windows

Since the plugin must be accessed through dynamic libraries (DLLs), the class definitions and methods in those classes must be exported by the compiler so that they may be visible through the DLL barrier. Thus, class definitions must have the following definitions when accessed from another plugin (assuming the Sphere plugin is being accessed from your plugin):

```
class __declspec(dllexport) Sphere
```

However, when compiling the Sphere plugin, the class must have the following definition:

```
class __declspec(dllimport) Sphere
```

This is accomplished in DANCE by adding the following code to the class definition:

```
#ifndef SPHERE_EXPORTS
#define DLLSPHERE __declspec(dllexport)
#else
#define DLLSPHERE __declspec(dllimport)
#endif

class DLLSPHERE Sphere
```

Any plugin that uses the Sphere class needs to include the preprocessor definition `SPHERE_EXPORTS` in the project properties under: `C/C++ / Preprocessor/Preprocessor Definitions`.

### 5.3 Making your plugin accessible through DANCE

#### Windows

To make your plugin accessible to other objects in the DANCE system, you must remember to include the DLL definitions described in the section above to your class. For example, to make a plugin called Sample available to the DANCE system, use the following class definition:

```
#ifndef SAMPLE_EXPORTS

#define DLLSAMPLE __declspec(dllexport)

#else

#define DLLSAMPLE __declspec(dllimport)

#endif
```

```
class DLLSAMPLE Sample
```

Your plugin must produce a .lib file (sample.lib for release mode, sampled.lib for debug mode), and place it in the \$(DANCE\_DIR)/lib/win directory. In addition, your plugin must have a .dll file (Sample.dll for release mode, Sample\_d.dll for debug mode) located in the \$(DANCE\_DIR)/plugins/win directory.