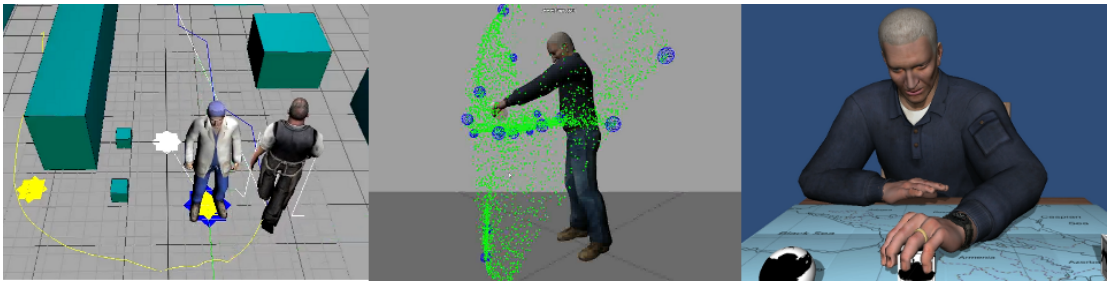
SmartBody



SmartBody is a character animation platform originally developed at the University of Southern California Institute for Creative Technologies. SmartBody provides locomotion, steering, object manipulation, lip syncing, gazing, physics, nonverbal behavior and other types of character movement in real time.

The following manual describes how to download, build and use the SmartBody system. For additional information, please refer to the SmartBody website at: smartbody.ict.usc.edu

SmartBody Manual

Updated 10/13/12

Ari Shapiro, Ph.D.

shapiro@ict.usc.edu

Overview

SmartBody is a character animation platform developed at the Institute for Creative Technologies, a part of the University of Southern California.

SmartBody provides the following capabilities for a digital character in real time:

- Locomotion (walk, jog, run, turn, strafe, jump, etc.)
- Steering - avoiding obstacles and moving objects
- Object manipulation - reach, grasp, touch, pick up objects
- Lip Synchronization and speech - characters can speak with lip-syncing using text-to-speech or prerecorded audio
- Gazing - robust gazing behavior that incorporates various parts of the body
- Nonverbal behavior - gesturing, head nodding and shaking, eye saccades

SmartBody is written in C++ and can be run as a standalone system, or incorporated into many game and simulation engines. We currently have interfaces for the following engines:

- Unity
- Ogre
- Panda3D
- GameBryo
- Unreal

It is straightforward to adopt SmartBody to other game engines as well using the API. The platform is distributed under the LGPL license.

SmartBody is a Behavioral Markup Language (BML) realization engine that transforms BML behavior descriptions into realtime animations.

SmartBody runs on Windows, Linux, OSX as well as the iOS and Android devices.

The SmartBody website is located at:

http://smartbody.ict.usc.edu

Questions, answers and comments about SmartBody can be posted on the SmartBody forum:

http://smartbody.ict.usc.edu/forum/

In addition, the SmartBody email group is located here:

smartbody-developer@lists.sourceforge.net

You can subscribe to it here:

https://lists.sourceforge.net/lists/listinfo/smartbody-developer

If you have any other questions about SmartBody, please contact:

Ari Shapiro, Ph.D.

shapiro@ict.usc.edu

Contributors

SmartBody Team Lead	**Ari Shapiro, Ph.D.**
SmartBody Team	Andrew W. Feng, Ph.D. Yuyu Xu
Inspiration & Guru	Stacy Marsella, Ph.D.
Past SmartBody Team Members	Marcus Thiebaux Jingqiao Fu Andrew Marshall Marcelo Kallmann, Ph.D.
Major SmartBody Contributors	Ed Fast Arno Hartholt Shridhar Ravikumar Apar Suri Adam Reilly Matt Liewer

Downloading SmartBody

The SmartBody source code can be downloaded using subversion (SVN) on SourceForge. You will need to have a Subversion client installed on your computer.

Use the SVN command:

```
svn co https://smartbody.svn.sourceforge.net/svnroot/smartbody/trunk smartbody
```

The entire repository is fairly large, several gigabytes in size. Note that only the trunk/ needs to be downloaded. Other branches represent older projects that have been since been incorporated in the trunk.

Windows

We recommend using Tortoise (http://tortoisesvn.net/). Once installed, right click on the folder where you want SmartBody installed then choose 'SVN Checkout', then put https://smartbody.svn.sourceforge.net/svnroot/smartbody/trunk into the 'URL of Repository' field, then click 'Ok' - this will start the download process.

Updates to SmartBody can then be retrieved by right-clicking in the smartbody folder and choosing the 'SVN Update' option.

Linux

If you don't have SVN installed on your system, Ubuntu variants can run the following command as superuser:

```
apt-get install subversion
```

Then, run the following in the location where you wish to place SmartBody:

```
svn co https://smartbody.svn.sourceforge.net/svnroot/smartbody/trunk smartbody
```

Updates to SmartBody can then be retrieved by going into the smartbody directory and running:

```
svn update
```

Mac/OsX

Subversion is already installed on OsX platforms. Run the following in the location where you wish to place SmartBody:

```
svn co https://smartbody.svn.sourceforge.net/svnroot/smartbody/trunk smartbody
```

Updates to SmartBody can then be retrieved by going into the smartbody directory and running:

```
svn update
```

Running SmartBody

SmartBody can be run in a number of different ways:

1. As a standalone application
2. Embedded within a game engine such as Ogre or Unity
3. On an Android or iPhone mobile device
4. Embedded as a Python library

Running SmartBody as a Standalone Application

The SmartBody distribution comes with its own renderer and suite of tools, called sbm-fltk. This file is located in the smartbody/core/sbm/bin directory. To run SmartBody within an existing game engine or other framework, please refer to the section on Integrating SmartBody With a Game Engine.

How to Use sbm-fltk

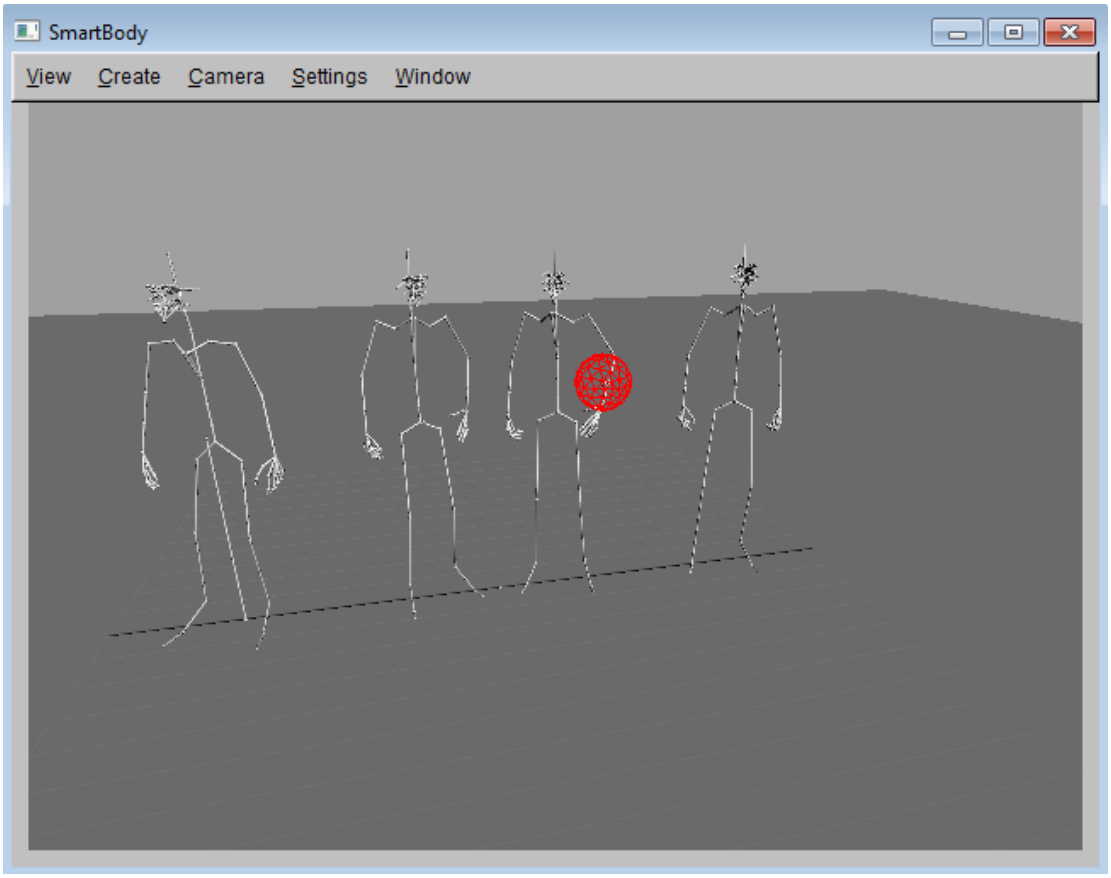
By default, the sbm-fltk application will search for a file called default.py in the same directory as the executable and will execute all commands contained in that script. The default.py file that comes with SmartBody contains commands to run another script called default-init.py which resides in the smartbody/data/sbm-common/scripts directory, and will load several SmartBody characters and position the camera so that it faces these characters.The sbm-fltk application will also accept the following command line arguments:

Argument	Description
-scriptpath	Path containing scripts to be initially loaded
-script	Initial script to run. If not specified, runs default.py.
-mediapath=	Media path for the application. The media path determines the prefix by which all subsequents paths are added. For example, if the media path is: /my/application, then specifying a relative directory for some file parameter, such as 'myfiles', results in the mediapath being added to it. In other words, the final path will be: /my/application/myfiles
-host=	Name of host to connect to using the BoneBus interface. If not specified, the BoneBus will not be launched on startup.
-fps=	Throttle used to limit the fps of the application. If this parameter is not specified, the application will run as fast as possible.
-procid=	identitifier of SmartBody instance. When running multiple SmartBody instances simultaneously, this distinguishes one from another.

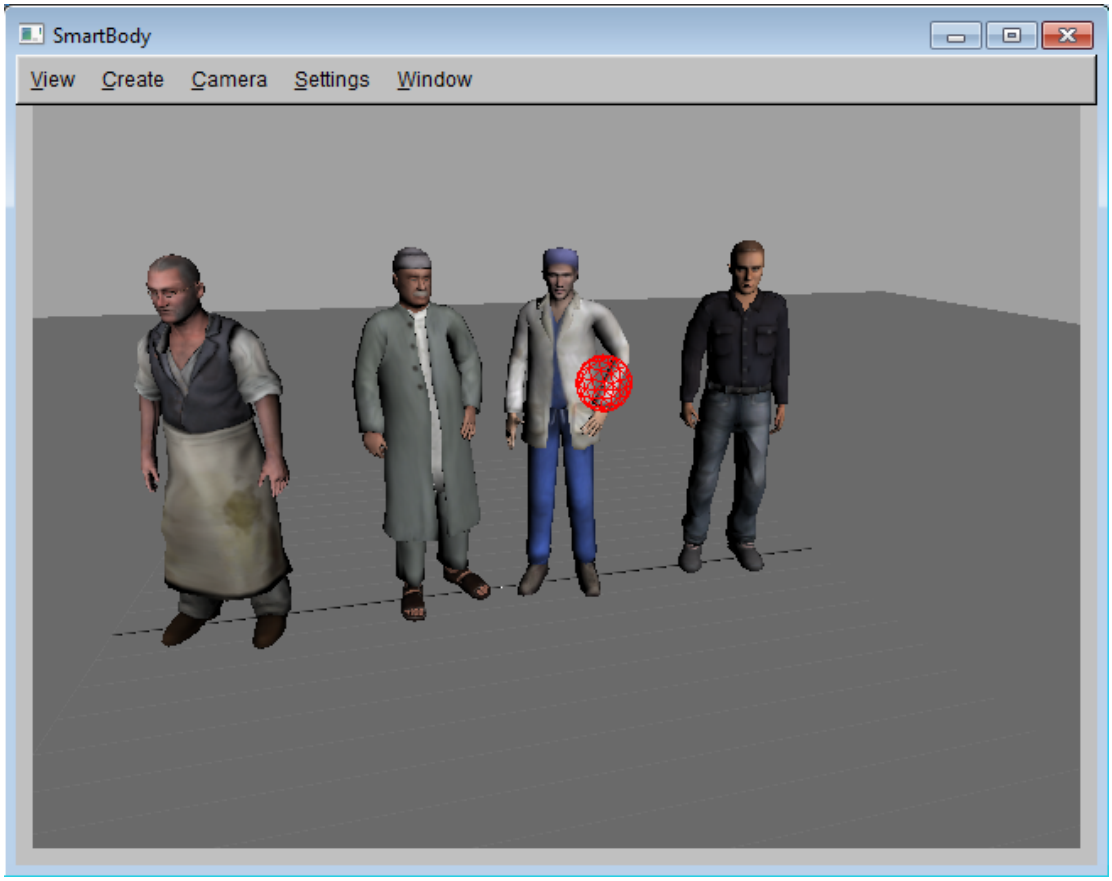
If no arguments are given to sbm-fltk, it will look in the current directory for a file named default.py and run the commands contained within it. Here is an example of some commands to put inside of the default.py file:

```
scene.addAssetPath("script","../../../../data/sbm-common/scripts")
scene.run("default-init.py")
```

the first command tells SmartBody where to find a folder/directory that contains scripts to be run, while the second command tells SmartBody to run the default-init.py script. Note that the default.py file can contain any set and any number of commands. The default-init.py file contains commands to load four characters into the scene, which should look like the image below:



By default, SmartBody will show the characters as skeletons. To see the skinned meshes with textures, select View -> Character -> GPU Deformable Geometry. The results should look like the following:



In some cases, GPU skinning will not be available, in which case you can use the CPU-based software skinning, which results in slower performance via View-> Character->Deformable Geometry. Note that pawns are represented as red spheres.

Adding Models to the Scene

Models can be added to the scene by attaching geometry to a pawn. You can set the 'mesh' attribute on a pawn via the Resource Viewer (Window->Resource View, click on pawns, select the pawn, then on the right side enter the path to the COLLADA/.dae or .obj file). Additionally, the 'meshScale' attribute can be used to scale the geometry. This can also be accomplished through Python:

```
mypawn = scene.getPawn(pawnName)
mypawn.setDoubleAttribute("meshScale", .2)
mypawn.setStringAttribute("mesh", "/path/to/geometry/file.dae")
```

Camera Control

Control	Behavior
ALT + Right Click	dolly
ALT + Middle Click	pan
ALT + Left Click	rotate

Also, the following menu choices are available:

Menu	Action
Camera->Reset	Restore the camera to its default position
Camera->Frame All	Adjust the camera so that all cameras and pawns are visible to the camera
Camera->Face Camera	Place the camera so that the selected character's face appears. If no character is selected, this option will have no effect.
Camera->Track Character	If a character is selected, maintain the relative position between the camera and the character, and follow the character when it moves. This can be used to keep a character in the cameras view while it is moving, without having to manually adjust the camera. To remove the tracking, select this option again. If no character is selected, this option will have no effect.

Selecting and Positioning Characters

A character or pawn can be selected by left-clicking on the character. This will cause a manipulator to appear at the character's root joint, which may vary in location on the character (sometimes the root joint is on the ground, other times it is in the middle of the character).

A character or pawn can be manually positioned by selecting one of the manipulator handles via a CTRL+Left Click and then dragging the character to the new location. To rotate the character or pawn, SHIFT + Left Click on the character or pawn, then SHIFT + Left Click on one of the manipulator handles.

Note that only the root/base joint can be altered interactively.

Controlling Characters

Characters can be controlled either by sending Python commands, or by using the BML Creator. Commands can be sent interactively through the Command Window, or through the VHMessage system using an 'sb' command.

In addition, characters can interactively move by first selecting the character, then right-clicking on the desired location.

Changing Lights

The default scene automatically uses two lights, one at (100, 250, 400), the other at (100, 500, -1000).

If you want to create and position your own lights in the scene, the SmartBody standalone application will use any pawns named 'light0', 'light1', 'light2', etc. as light sources. Thus, to customize the lighting:

1. Create an pawn named 'light0', either by creating the pawn via scripting:
    a. scene.createPawn("light0")
    b. or through the menu by choosing Create->Light
2. Position the light by changing the position of the pawn.
3. Change the lighting attributes as follows:

Attribute	Description
lightIsDirectional	
	boolean. Is the light a directional light? Default is True
lightDiffuseColor	vec3. Diffuse light color. Default is (1, .95, .8)
lightAmbientColor	vec3. Ambient light color. Default is (0, 0, 0)
lightSpecularColor	vec3. Specular light color. Default is (0, 0, 0)
lightSpotExponent	double. Spotlight exponent. Default is 0
lightSpotDirection	vec3. Spotlight direction. Default is (0,0,-1)
lightSpotCutoff	double. Spotlight cutoff angle. Default is 180
lightConstantAttenuation	double. Constant attenuation. Default is 1
lightLinearAttenuation	double. Linear attenuation. Default is 1
lightQuadraticAttenuation	double. Quadratic attenuation. Default is 0

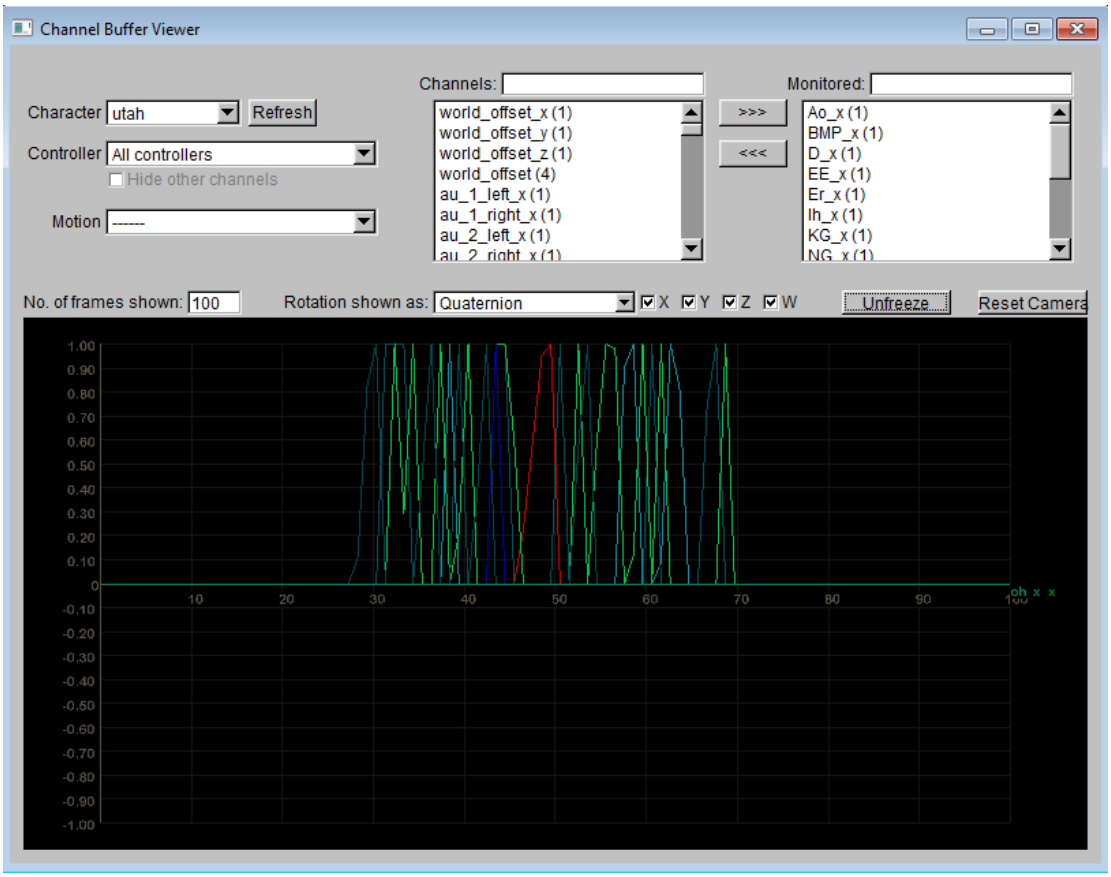
Other Controls

Fle->New	Resets SmartBody to run a new simulation. All objects will be removed.
File->Save	Saves the SmartBody configuration to a .py file. Note that the state of the simulation (including behaviors) will not be saved, but the characters, pawns, postiions, orientations, asset paths, blends, face definitions, joint maps, diphones, camera positions will be saved.
File->Load	Loads a SmartBody scene from a Python (.py) file. The existing scene will be reset.
File->Connect	Connect to another running instance of SmartBody. The objects in the scene in the remote SmartBody instance will be displayed in the window.
File->Disconnect	Disconnect from a remote running instance of SmartBody.
File->Quit	Exit SmartBody
View->Character->Bones	Show the character bones and joints
View->Character->Geometry	Show any rigid geometry bound to the character
View->Character->Collision Geometry	Show any rigid geometry bound to the character used as a collision surface
View->Character->Deformable Geometry	Show character mesh using software-based skinning
View->Character->GPU Deformable Geometry	Show character mesh using GPU-based skinning much faster than software-based skinning, but may not work on older video cards)
View->Character->Axis	Show the orientation of the character's joints.
View->Character->Eyebeams	Show a line segment extending from the character's eyes in the direction of the gaze
View->Character->Bounding Volumes	Displays the bounding volumes for the characters.

View->Character->Show Trajectory	Shows a yellow line for the position of the character in the recent path.
View->Pawns	Toggles pawn display on and off
View->Shadows	Toggles shadows on and off
View->Grid	Toggles the grid on and off
View->Steer->Characters and Goals	Shows debugging information about the steering: characters and their current goals
View->Terrain->Shaded	Shows any terrain as set of polygons
View->Terrain->Wireframe	Shows any terrain as a wireframe
View->Terrain->No Terrain	Hides any terrain data
View->Steer->All Steering	Shows debugging information about the steering: characters, goals, obstacles, facing direction
View->Steer->No Steering	Hides all debugging information about the steering
Create->Character	Creates a character of a specified name. You have to choose the skeleton to associate with the character.
Create->Pawn	Creates a pawn with a given name.
Create->Light	Creates a light. Lights are pawns with the name 'lightN' where 'N' is the light number.
Create->Terrain	Loads a terrain as a heightfield from a .ppm file
Camera->...	See descriptions under Camera int the section above
Settings->Internal Audio	Toggle SmartBody's ability to respond to sound playing commands.
Window->....	Launch the various windows, detailed below
Help->Create Python API	Saves the Python API to an HTML file.

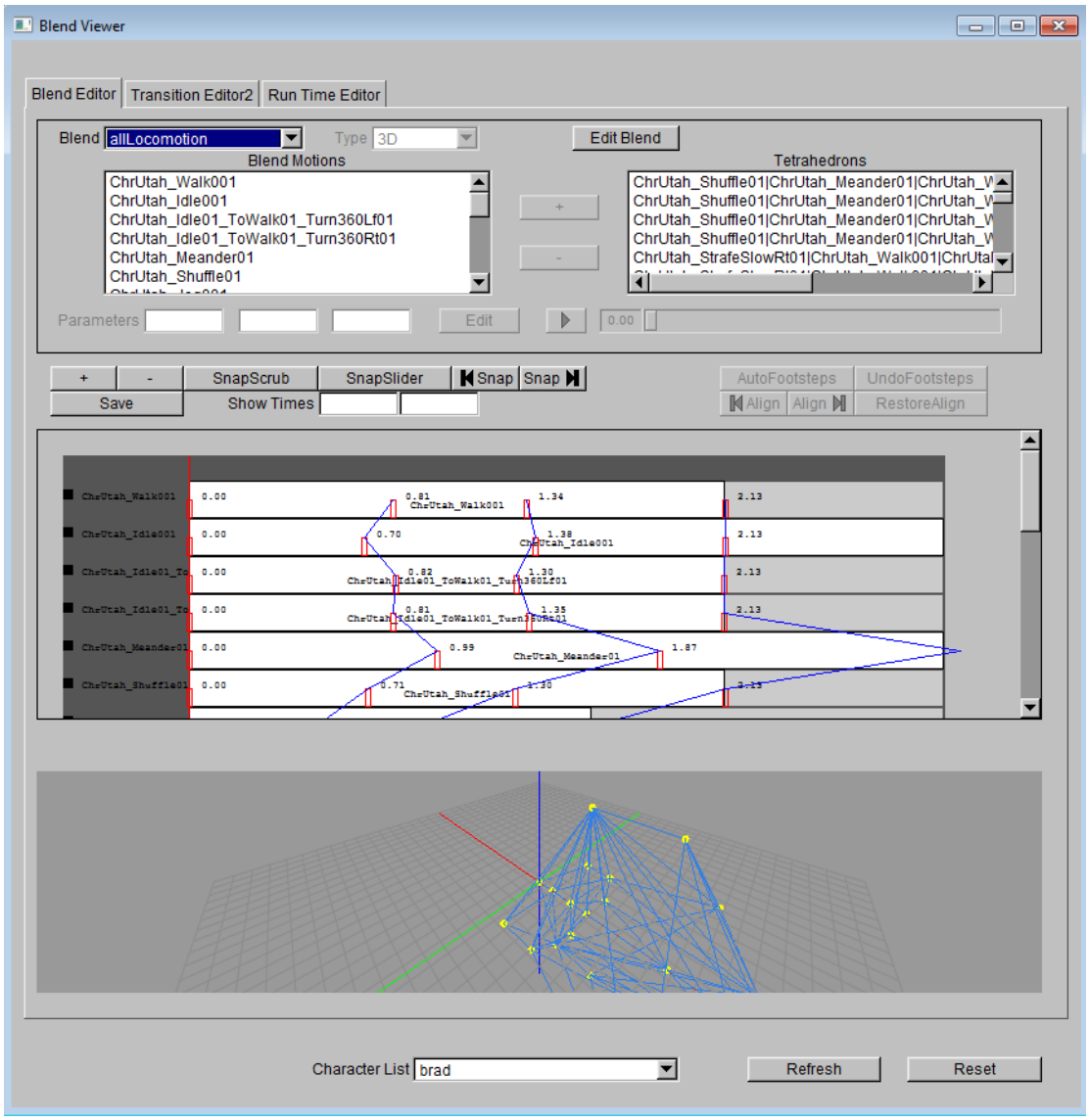
Data Viewer

The Data Viewer allows you to examine joint and channel data on any character in real time.



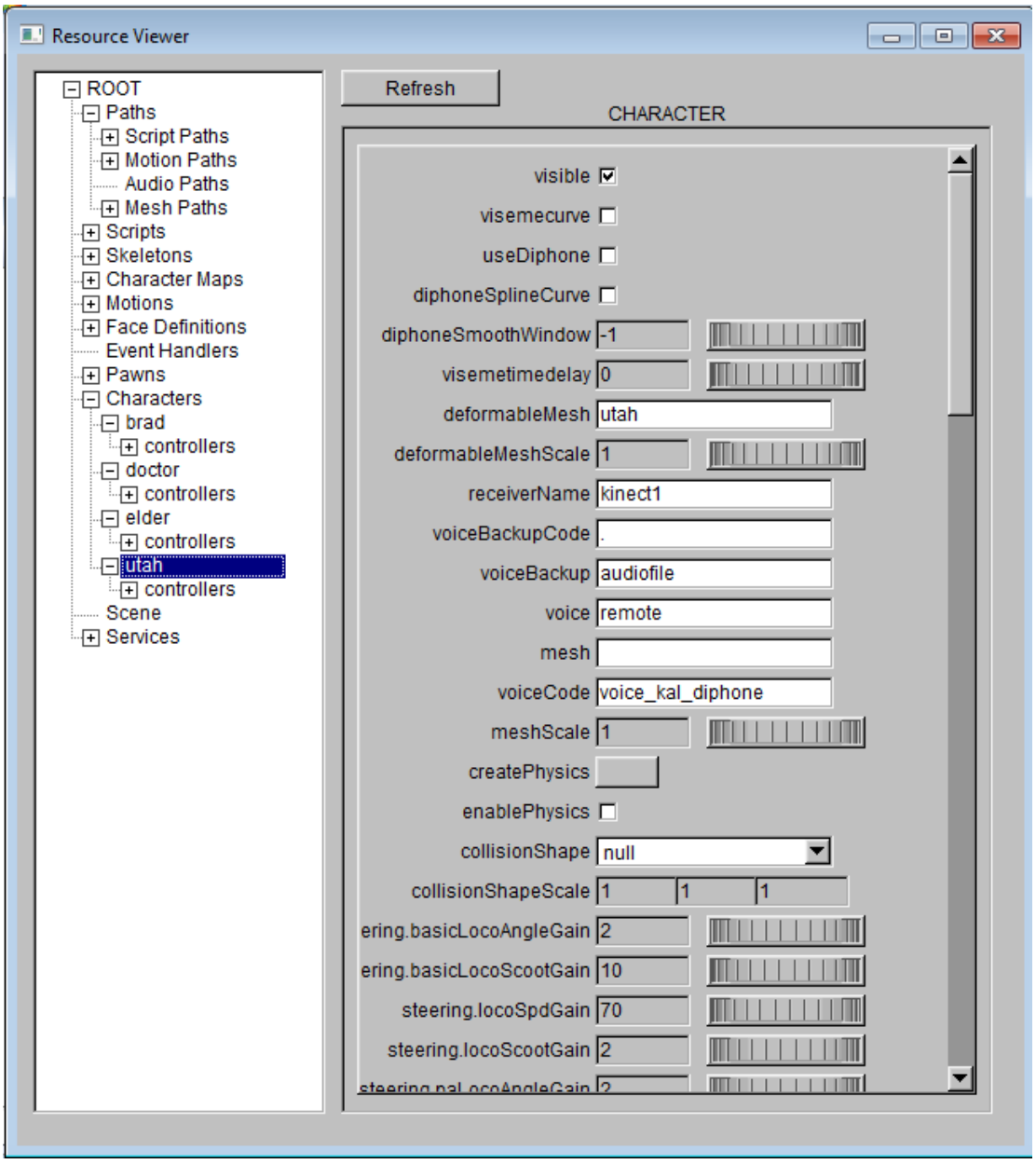
Blend Viewer

The Blend Viewer allows you to create, edit and test blends and transitions between blends.



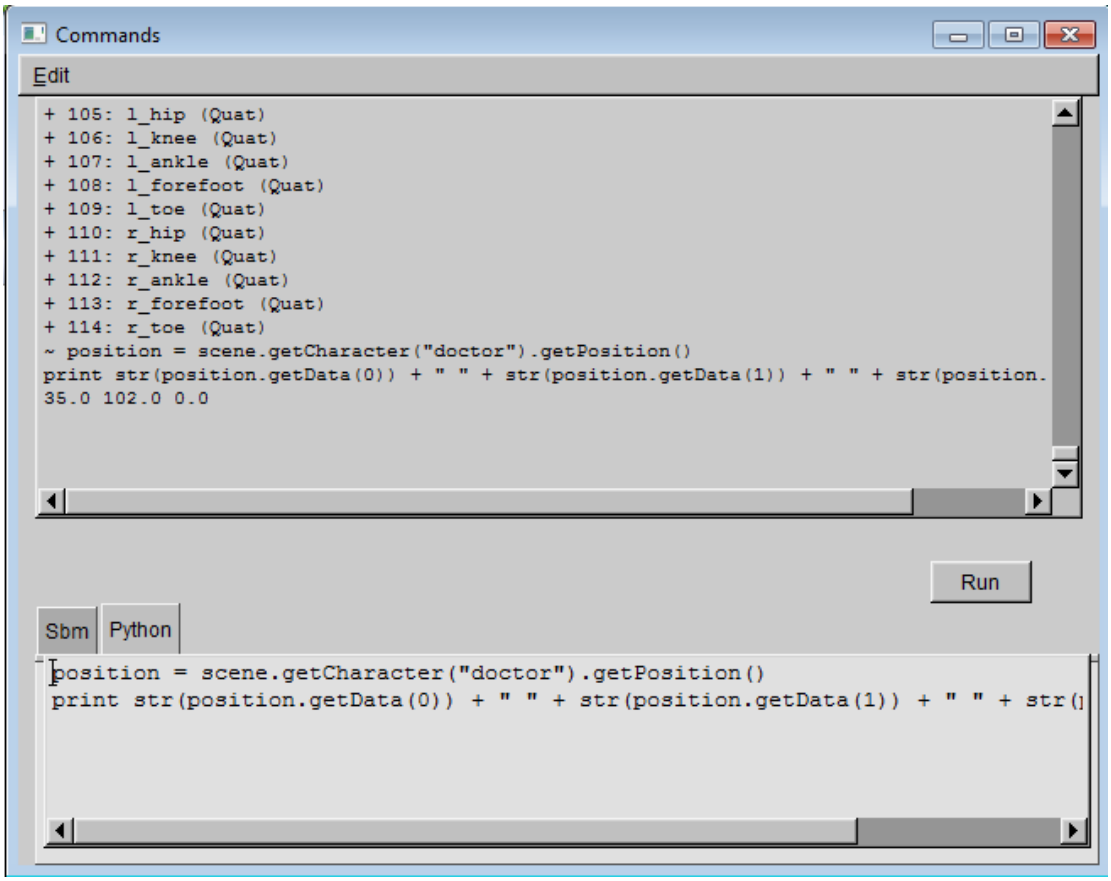
Resource Viewer

The Resource Viewer allows you to examine the assets loaded into the system (skeletons, animations) as well as the characters, pawns, and services being run. All run time structures, such as joint mappings, face definitions, controllers and scripts can be examined or triggered from this window. In addition, there is an interactive interface for examining and setting attributes.



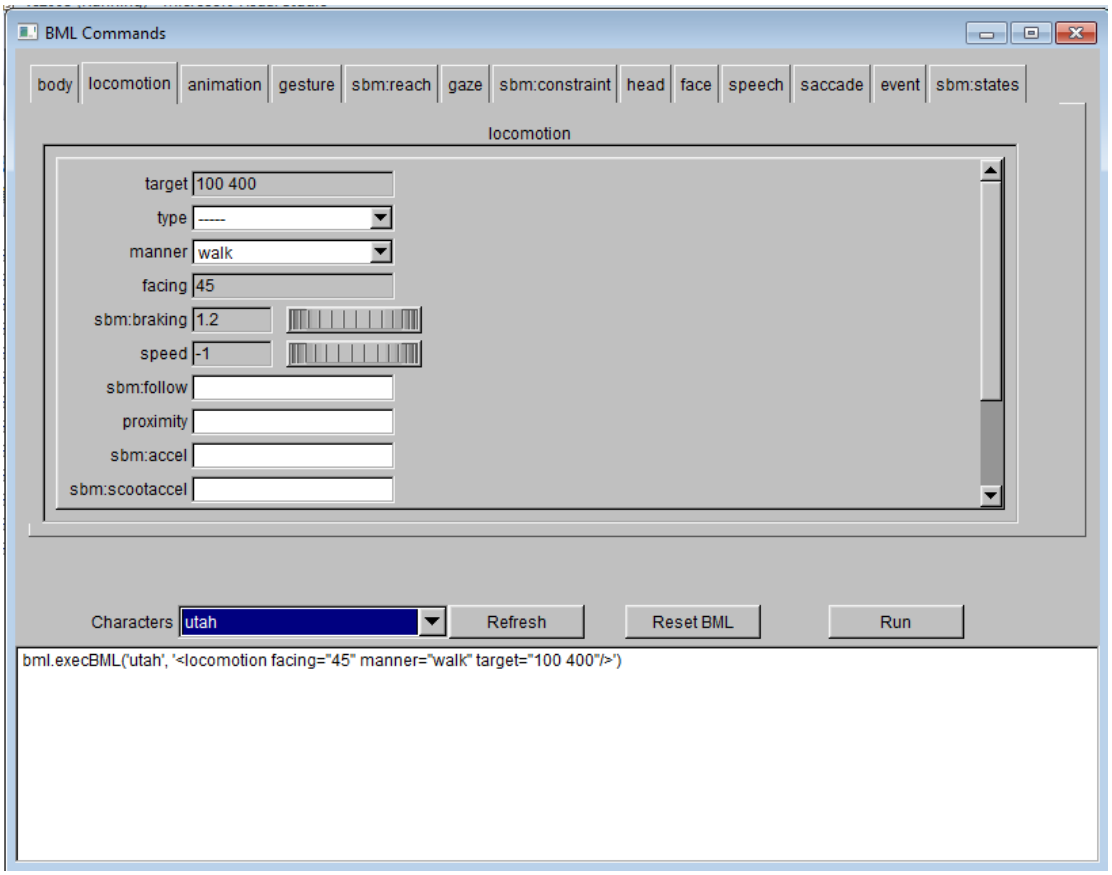
Command Window

The Command Window allows you to write commands, as well as examine messages output from SmartBody.



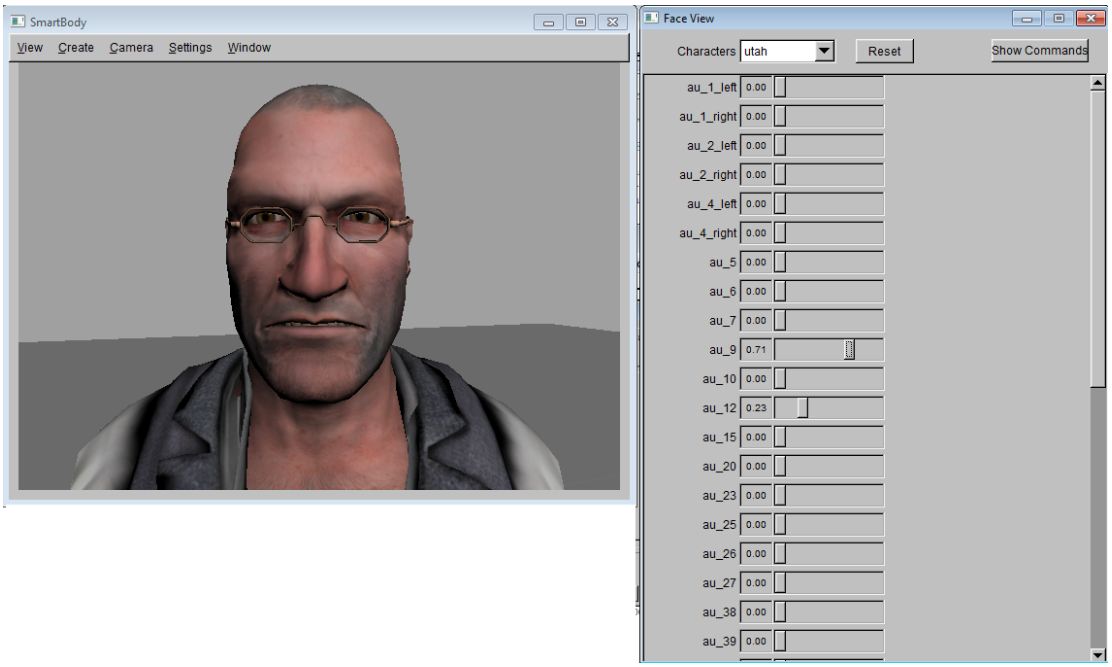
BML Creator

The BML Creator Window allows you to create Behavior Markup Language commands easily. The interface shows the different BML commands available, as well as all the options that can be set for them. For each option, a description will appear if you mouse over the input box.



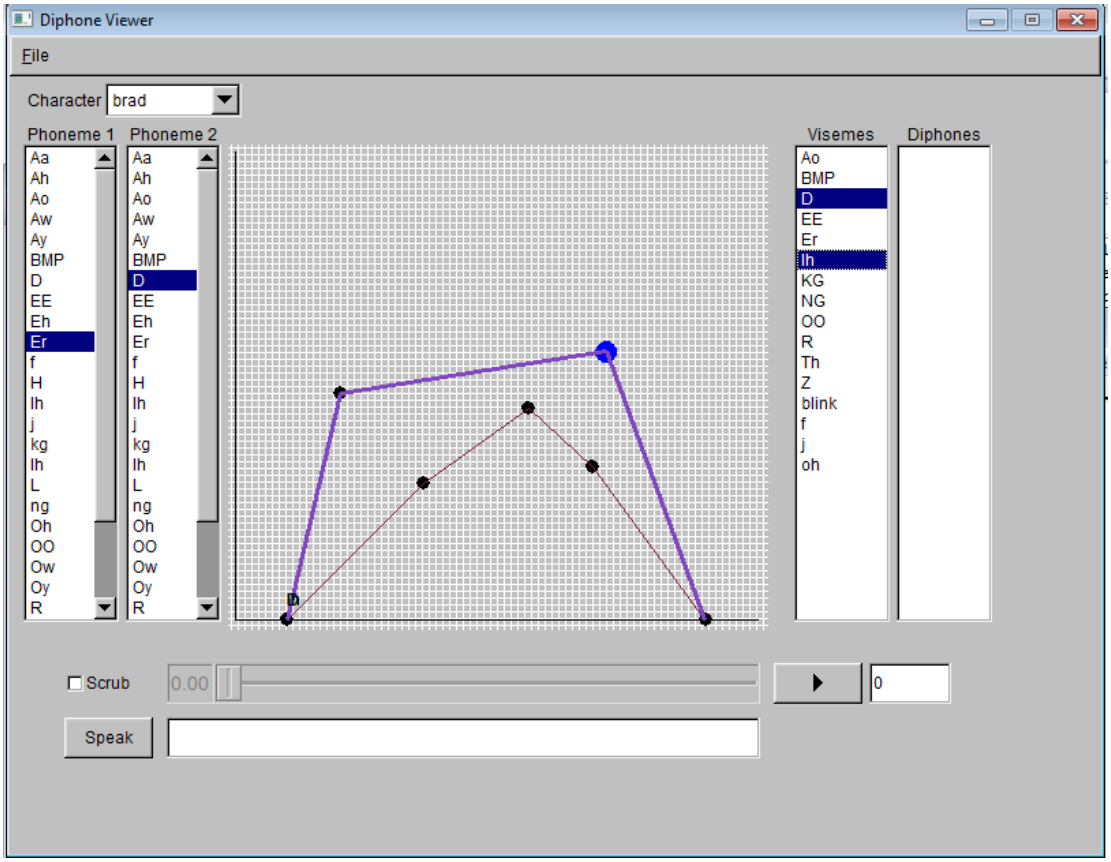
Face Viewer

The Face Viewer allows you to interactively control the facial animation of a SmartBody character. The controls that appear are based on a character's Face Definition. The controls consist of Action Units, which control various aspects of the face, and visemes, which are associated with lip syncing.



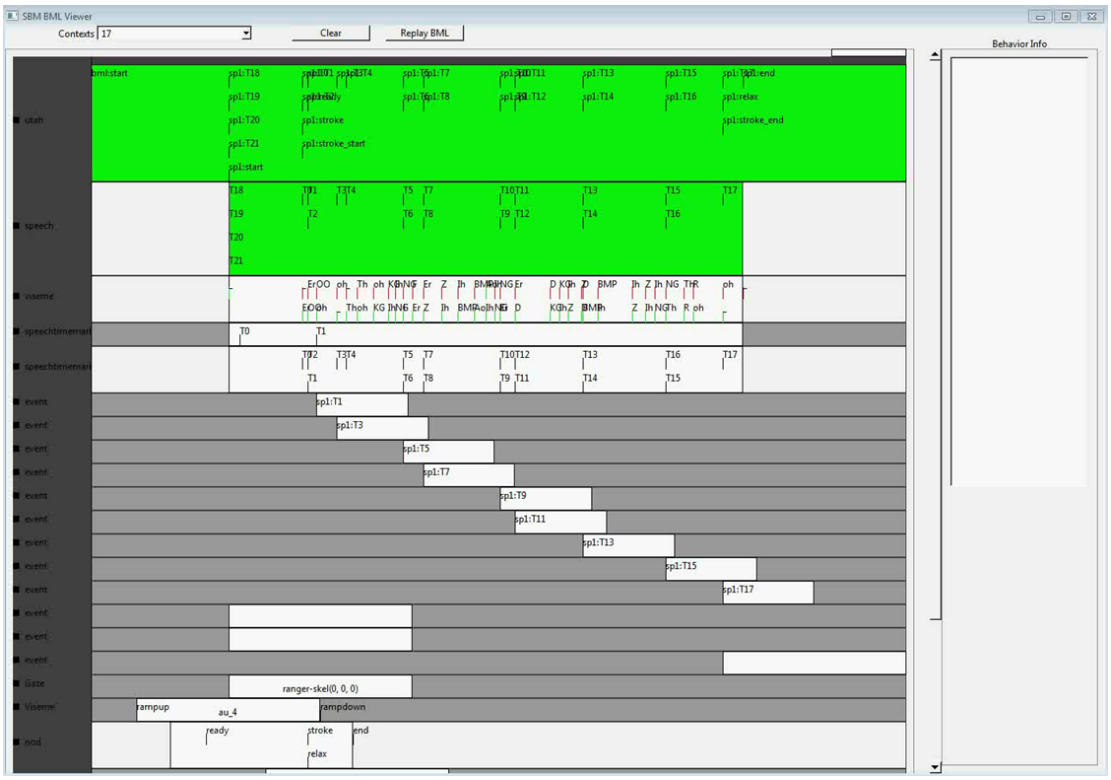
Diphone Viewer

The Diphone Viewer allows you to examine the sets of facial animation associated with various diphones that are generated from either a text-to-speech or prerecorded audio file.



BML Viewer

The BML Viewer shows a visualization of various parts of a BML command how they are timed together.



Running SmartBody in the Ogre Game Engine

The SmartBody distribution includes an application called OgreViewer which can be used to run SmartBody either in-process or out-of-process via the BoneBus protocol.

For SmartBody characters to appear in OgreViewer, an Ogre mesh and Ogre skeleton needs to be present that matches the character type when the character is created:

```
mycharacter = scene.createCharacter("myname", type)
myskeleton = scene.createSkeleton(skeletonname)
mycharacter.setSkeleton(myskeleton)
```

here 'type' is the name of an Ogre mesh file that exists in smartbody/core/ogre-viewer/media. For example, if the 'type' is 'brad', then the file brad.mesh and brad.skeleton will be retrieved from the Ogre media directory. Note that the skeleton used for the SmartBody character ('skeletonname') must match the skeleton used by Ogre. If no Ogre mesh is available for the particular character type, then the default mesh (Brad.mesh) will be used, although doing so might result in mismatches between the SmartBody skeleton and the Ogre skeleton.

The OgreViewer will run in BoneBus mode automatically if no command line parameters are entered. In BoneBus mode, the OgreViewer application will connect to a running SmartBody process and render characters if a matching Ogre-compatible mesh is available.

If command line options are added to OgreViewer, then SmartBody will be run in-process by setting the following command line options:

Argument	Description

-config	Use a configuration file with the following format and options:
	
	`[GENERAL]`
	`UseBoneBus=False`
	`ScriptPath=../../../../data/sbm-common/scripts`
	`DefaultPyFile=default-init.py`
	`MediaPath=.`
	`Scene=vh_basic_level.mesh`

-mediapath	Media path for the application. The media path determines the prefix by which all subsequents paths are added. For example, if the media path is: /my/application, then specifying a relative directory for some file parameter, such as 'myfiles', results in the mediapath being added to it. In other words, the final path will be: /my/application/myfiles
-scriptpath	Path containing scripts to be initially loaded
-script	Initial script to run. If not specified, runs default.py.
-scene	The default mesh that will be loaded into the scene. The mesh must be placed in the Ogre media path.
-help	Prints the help message then exits the application.

Controlling the Camera in Ogre

Key	Behavior
1,2,3,4	Toggles between the default scenes: 1 = default scene as specified on the command line or via configutation file 2 = world scene in feet 3 = world scene in centimenters 4 = scene with rectangular plane and shadows
J	Toggle the camera rotation via mouse off and on
P	Display the camera details
A,D	Move the camera left orright
W,S	Move the camera forward orbackward
Q,E	Move the camera up or down
N	Toggle solid, wireframe or points mode

Controlling Characters in Ogre

If the OgreViewer is being run in BoneBus mode (out-of-process), then commands can be sent directly to the SmartBody process. Please see the documentation on running the SmartBody standalone program. The OgreViewer itself has no interactive controls, and thus any interactive commands need to be sent through via a VHMessage through ActiveMQ. This can be done, for example, via the elsender application. Please see the section on sending VH Messages for more details.

Creating Models and Skeletons for SmartBody and Ogre

We recommend exporting an Ogre mesh and skeleton from your modeling tool in a T- or A- pose. We have had success using the OgreMax exporter (www.ogremax.com/downloads). Additionally, you will need to export the Ogre skeleton into a format that SmartBody understands, such as a COLLADA, bvh, asf or fbx. This skeleton file must be places in one of SmartBody's asset paths (for example, /path/to/smartbody/data/sbm-common/common-sk).

Additionally, any skeleton that doesn't use the standard SmartBody naming convention would need to use a joint mapping (please see the section on Using Custom Skeletons and Animations) before use. Additionally, to use any existing animations in the SmartBody distribution, motions for that skeleton would need to be retargeted (please see the section on Retargetting).

Examples

To run SmartBody using a configuration file, make sure a config.ini file is located in the same directory as the OgreViewer executable containing parameters such as the following:

```
[GENERAL]
UseBoneBus=False
ScriptPath=../../../../data/sbm-common/scripts
DefaultPyFile=default-init.py
MediaPath=.
Scene=vh_basic_level.mesh
```

To run SmartBody in the OgreViewer process, enter a script path, script and optionally a media path:

```
OgreViewer -mediapath /path/to/smartbody/trunk/data -scriptpath sbmcommon/scripts -script default-init.py
```

To run SmartBody in a separate process that connects to the OgreViewer (BoneBus mode), first run the OgreViewer:

```
OgreViewer
```

then run sbm-fltk with the appropriate parameters:

```
sbm-fltk -mediapath /path/to/smartbody/trunk/data -scriptpath sbmcommon/scripts -script default-init.py
```

Running SmartBody with the Unity Game Engine

(details to follow)

Running the SmartBody Desktop Application

The SmartBody distribution includes an application called sbdesktop that renders an entire SmartBody scene as a transparent window on a Windows desktop. The application is a fully functional SmartBody application, and can respond to commands controlled either automatically through a script, or interactively via the ActiveMQ network.

If not given any command line parameters, the sbdesktop application will look for a file called default.py in the same directory as the executable

and run any commands found inside. The default.py that comes with the distribution contains the following commands to set the script path and then

run a setup file that creates the Utah character and positions the camera in a way appropriate for showing him on the screen:

```
scene.addAssetPath("script", "../../../../data/sbm-common/scripts")
scene.run("default-init-desktop.py")
```

The following command line options can be sent to sbdesktop:

Argument	Description
-scriptpath	Path containing scripts to be initially loaded
-script	Initial script to run. If not specified, runs default.py.

-mediapath=	Media path for the application. The media path determines the prefix by which all subsequents paths are added. For example, if the media path is: /my/application, then specifying a relative directory for some file parameter, such as 'myfiles', results in the mediapath being added to it. In other words, the final path will be: /my/application/myfiles
-x	X location of the transparent window, defaults to 200
-y	Y location of the transparent window, defaults to 150
-w	width of the transparent window, defaults to 320
-h	height of the transparent window, defaults to 240

Running SmartBody as an Embedded Python Library

SmartBody can be run as a Python library and thus embedded within any Python process.

(details to follow)

Building SmartBody

SmartBody can be built on Windows, OSx, Linux, Android and iOS platforms.

There are several SmartBody applications that can be built:

Application	Comment
sbm-fltk	a standalone SmartBody application and scene renderer
smartbody-dll	a dynamic library that can be incorporated into a game engine or other application
sbmonitor	(*Windows only*) A separate application that can be used to inspect, monitor and control a running SmartBody instance.
sbm-batch	a standalone SmartBody application that connects to smartbody-dll without a renderer
TtsRelayGui	(*Windows only*) incorporates any text-to-speech engine that uses the TtsRelay interface
FestivalRelay	Text-to-speech engine that uses Festival
MsSpeechRelay	(*Windows only*) Text-to-speech engine that uses Microsoft's built-in speech engine
OgreViewer	The Ogre rendering engine connected to SmartBody
sbdesktop	(*Windows only*) A fully-functional SmartBody environment rendered on the desktop.

In addition, there are several mobile applications that can be built:

Mobile Application	Comment
sbmjni	(*Android only*) Simple OpenGL ES front end for SmartBody for Android devices
sbm-ogre	(*Android only*) Ogre3D engine connected with SmartBody for Android devices
vh-wrapper	(*Android only*) SmartBody interface to Unity3D.
smartbody-openglES	(*iOS only*) Simple OpenGL ES front end for SmartBody for iOS devices
smartbody-ogre	(*iOS only*) Ogre3D engine connected with SmartBody for iOS devices
smartbody-unity	(*iOS only*) Unity3D project for SmartBody

Please see Appendix 1: Building SmartBody for details about how to build each application on the various platforms.

SmartBody Architecture

A SmartBody simulation consists of a scene, one or more characters, one or more pawns, and a number of assets, including animations, skeleton hierarchies, sounds, geometric meshes and textures, among others. Characters are controlled by a set of specialized controllers, which control various aspects of movement, such as idle posturing, facial animation, locomotion, gesturing, movement under physics and so forth. Every SmartBody object, such as a character, pawn, scene, motion or skeleton, has an associated set of dynamic attributes, which can be queried and changed in order to calibrate various features, or to toggle them on or off. SmartBody can be controlled directly through the underlying C++ interface, or through a high level scripting interface in Python. SmartBody can respond to Behavior Markup Language (BML) commands as a means to control and coordinate the behaviors of one or more characters.

A SmartBody simulation can either use a real-time clock, or can be set explicitly by an outside source.

SmartBody can connect to and receive messages from other cooperating components through the use of the Virtual Human Message system (VHMessage). The VHMessage system is an asynchronous messaging system that is built atop the ActiveMQ framework.

Integrating SmartBody With a Game Engine

SmartBody can be integrated with any game engine that uses a C or C++ interface, or by those that utilize a Python interface. Additionally, SmartBody can connect to a game engine using the Bone Bus interface, which is a network-based API, which would require no direct contact with SmartBody except via that network protocol.

Suggested Integration Technique

In order to integrate SmartBody with a game engine using C or C++:

a. In your game engine, create a character that corresponds to a SmartBody character
b. Use the SmartBodyListener interface to respond to character creation and deletion events.
c. Send SmartBody commands to control the character and change the scene every frame as needed.
d. Query SmartBody every frame to obtain the character state, and change the game engine character's state to match it

There are two different C/C++ APIs that can be used: a simplified interface where instructions to SmartBody are sent as Python commands via the VHMessage system (smarbody-dll) and a comprehensive API that allows you to access, query, and change all of SmartBody's internal structures (smarbody-lib). In addition, there is way to integrate SmartBody into a game engine using a network protocol called BoneBus which is language-independent.

The following pages describe the smartbody-dll, smartbody-lib and BoneBus interfaces.

Using smartbody-dll

smartbody-dll provides a simplified interface to SmartBody in a dynamically linked library.

To integrate SmartBody into a game engine using smartbody-dll, you will need to extend the SmartBodyListener class:

```
#include "smartbody-dll.h"

class MyListener : public SmartBodyListener
{
virtual void OnCharacterCreate( const std::string& name, const std::string& objectClass );
virtual void OnCharacterDelete( const std::string& name );
virtual void OnCharacterChanged( const std::string& name );
virtual void OnViseme( const std::string& name, const std::string& visemeName, const float weight, const float
blendTime );
virtual void OnChannel( const std::string& name, const std::string& channelName, const float value );
}
```

The OnCharacterCreate() function gets called whenever a SmartBody character is instantiated, where the object class is a string that can be used by a renderer to determine what mesh to use as deformable geometry, or other character-specific configuration that is handled by the renderer. The OnCharacterDelete() function whenever a character is removed. The

OnCharacterChanged() method is called whenever that character's skeleton is changed, such as by adding joints or other channels. The OnViseme() method is called every frame and sends the values of the visemes (for renderers that use blend shapes). The OnChannel() function is called every frame whenever non-joint information is send from SmartBody. The integration should create characters in the game engine in response to those callbacks, for example:

```
void MyListener::OnCharacterCreate( const std::string& name, const std::string& objectClass )
{
if (name does not exist)
{
   // create game engine character here
}
}
```

To initialize SmartBody, instance the class Smartbody_dll,set the media path, and the initialize it:

```
sb = new Smartbody_dll();
sb->SetMediaPath("/path/to/my/data/");
sb->Init("/path/to/Python/", true);
```

Then call the Update() function by setting the time every simulation frame:

```
sb->Update(currenTimeInSeconds);
```

After the Update(), the character state can be retrieved and used the update the state of the game engine character as follows:

```
int num = sb->GetNumberOfCharacters()
for (size_t n = 0; n < num; n++)
{
SmartBodyCharacter& character = sb->GetCharacter(n);
int numJoints = character.m_joints.size();
for (size_t j = 0; j < numJoints; j++)
{
SmartbodyJoint& joint = character.m_joints[j];
// get the position from the joint
float xpos = joint.x;
float ypos = joint.y;
float zpos = joint.z;
// get the orientation from the joint
float quatw = joint.rw
float quatx = joint.rx;
float quaty = joint.ry
float quatz = joint.rz;
// update the game engine character
//  ...
// ...

}
```

To access other functionality, any function allowed by the SmartBody Python API can be accessed using the following functions:

```
bool PythonCommandVoid( const std::string & command );
bool PythonCommandBool( const std::string & command );
int PythonCommandInt( const std::string & command );
float PythonCommandFloat( const std::string & command );
std::string PythonCommandString( const std::string & command );
```

For example, to run a Python command that has no return value:

```
PythonCommandVoid("scene.setSceneScale(1.0)");
```

Alternatively, if you want to retrieve data from a Python function, the Bool, Int, Float and String functions can be called, as long as the return value is then assigned to a Python variable called *ret*. For example:

```
int val = PythonCommandBool("ret = scene.getNumCharacters()");
std::string val = PythonCommandString("ret = scene.getMotion(\"LHandOnHip_RArm_GestureOffer\").getName()");
```

Note that the Python context persists throughout the simulation, so you can assign a Python variable during one call:

```
PythonCommandVoid("mycharacter = scene.getCharacter(\"utah\")");
```

and subsequently use it during a later call:

```
float xpos = PythonCommandFloat("mycharacter.getPosition().getData(0)");
```

At any time, SmartBody commands can be sent by using the ProcessVHMsgs() function, which sends a message to the ActiveMQ server, which is then picked up by SmartBody. Note that since this interface requires the VHMessage interface, it is necessary to have an ActiveMQ server running.The message should be of the following format:

```
ProcessVHMsgs( "sb", "python command goes here");
```

where 'command' is the Python command. For details on the Python interface, please consult Using Python with SmartBody and Appendix 2: Python API for SmartBody.

Note that game engines that require a C-style interface (for example, an engine that only uses C#, which can access C-style but not C++-style functions), as opposed to a C++-style interface can use the smartbody-c-dll.h interface instead of smartbody-dll.h, with similarly named functions:

C++ Function	C Function	Comments
`OnCharacterCreate`	SBM_IsCharacterCreated	Equivalent of callback function. Must be queried every frame.
`OnCharacterDelete`	SBM_IsCharacterDeleted	Equivalent of callback function. Must be queried every frame.
`OnCharacterChanged`	SBM_IsCharacterChanged	Equivalent of callback function. Must be queried every frame.
`OnViseme`	SBM_IsVisemeSet	Equivalent of callback function. Must be queried every frame.
`OnChannel`	SBM_IsChannelSet	Equivalent of callback function. Must be queried every frame.
SetMediaPath	SBM_SetMediaPath	Sets the media path
Init	SBM_Init	Initializes a SmartBody instance
Update	SBM_Update	Updates the SmartBody simulation step
ProcessVHMsgs	SBM_ProcessVHMsgs	Processes any pending Virtual Human messages

Using smartbody-lib

smartbody-lib provides full access to the underlying SmartBody structures.All SmartBody components including the scene, characters, motions, commands, and other configuration structures can be queried, accessed and changed through this interface.

To set up a SmartBody scene, first obtain the scene:

```
SmartBody::SBScene* scene = SmartBody::getScene();
```

The scene object allows you to create characters, pawns, configure settings, and send commands to SmartBody characters. For example, to create a character:

```
SmartBody::SBCharacter* character = scene->createCharacter(characterName, objectClass);
```

In addition, the SBScene object has access to the various managers in SmartBody. For example. you can access the physics system via:

```
SmartBody::SBPhysicsManager* physicsManager = scene->getPhysicsManager();
```

To run the simulation, obtain the simulation manager:

```
SmartBody::SBSimulationManager* sim = scene->getSimulationManager()
```

Then either update the time to match the clock of the other components:

```
sim->updateTime(timeInSeconds);
```

by contrast you can use the real-time clock, which will automatically update when the simulation is advanced. To advance the simulation, call the step() function:

```
sim->step();
```

which will advance the SmartBody simulation one step at a time. At any time, the state of the system such as characters can be queried like so:

```
std::vector<std::string> characterNames = scene->getCharacterNames();
for (size_t c = 0; c < characterNames.size(); c++)
{
SmartBody::SBCharacter* character = scene->getCharacter(characterNames[c]);
SmartBody::SBSkeleton* skeleton = character->getSkeleton();
int numJoints = skeleton->getNumJoints();
for (int j = 0; j < numJoints; j++)
{
SmartBody::SBJoint* joint = skeleton->getJoint(j);
// retrieve joint information here...
}
}
```

To respond to changes in SmartBody triggered by Python commands, override the SBMListener interface:

```
#include <sbm/mcontrol_util.h>

class MyListener : public SBMCharacterListener
{
public:
virtual void OnCharacterCreate( const std::string & name, const std::string & objectClass );
virtual void OnCharacterDelete( const std::string & name );
virtual void OnCharacterUpdate( const std::string & name, const std::string & objectClass );
virtual void OnCharacterChanged( const std::string& name );
virtual void OnPawnCreate( const std::string & name );
virtual void OnPawnDelete( const std::string & name );
virtual void OnViseme( const std::string & name, const std::string & visemeName, const float weight, const
float blendTime );
virtual void OnChannel( const std::string & name, const std::string & channelName, const float value);
};
```

OnCharacterCreate() and OnCharacterDelete() are called whenever a character is created or deleted. OnPawnCreate() and OnPawnDelete() are called whenever a pawn is created or deleted. OnCharacterUpdate() and OnCharacterChanged() are called whenever the character's skeleton or channel information is changed. OnViseme() is called to indicate the state of the visemes, which is useful for renderers that use blend shapes. OnChannel() will be called whenever non-joint and non-viseme channels are modified.

```
class MyListener : public SmartBodyListener
{
virtual void OnCharacterCreate( const std::string& name, const std::string& objectClass );
virtual void OnCharacterDelete( const std::string& name );
virtual void OnCharacterChanged( const std::string& name );
virtual void OnViseme( const std::string& name, const std::string& visemeName, const float weight, const float
blendTime );
virtual void OnChannel( const std::string& name, const std::string& channelName, const float value );
}
```

Using BoneBus

The BoneBus is a TCP and UDP based network protocol that sends information about the SmartBody character state to a renderer on a remote machine, or in a different process than the originating SmartBody process. SmartBody provides a C++ and C implementation of this protocol, but any coding method could be used provided that it interprets the BoneBus protocol properly.

To connect a renderer to SmartBody via BoneBus, the game engine code should use the bonebus::BoneBusServer interface:

(more details to follow)

Using Python with SmartBody

A SmartBody simulation can be controlled through the use of a Python API. The Python API can be used to start and stop the simulation, create or remove a character, configure characters, place objects in the scene and so forth. A full reference of the Python API can be found in Appendix 2: Python API for SmartBody.

How to Send Python Commands to SmartBody

There are several ways to send Python commands, depending on how you are interfacing with SmartBody.

Sending Python Commands via VHMessage

If your SmartBody instance is connected to an ActiveMQ server via the VHMessage system, you can send a message formatted as follows:

```
sb <python command goes here>
```

Where the sbm commands tell SmartBody to interpret the message, and the command python tells SmartBody to interpret everything after the space as a Python command.

Similarly, you can send:

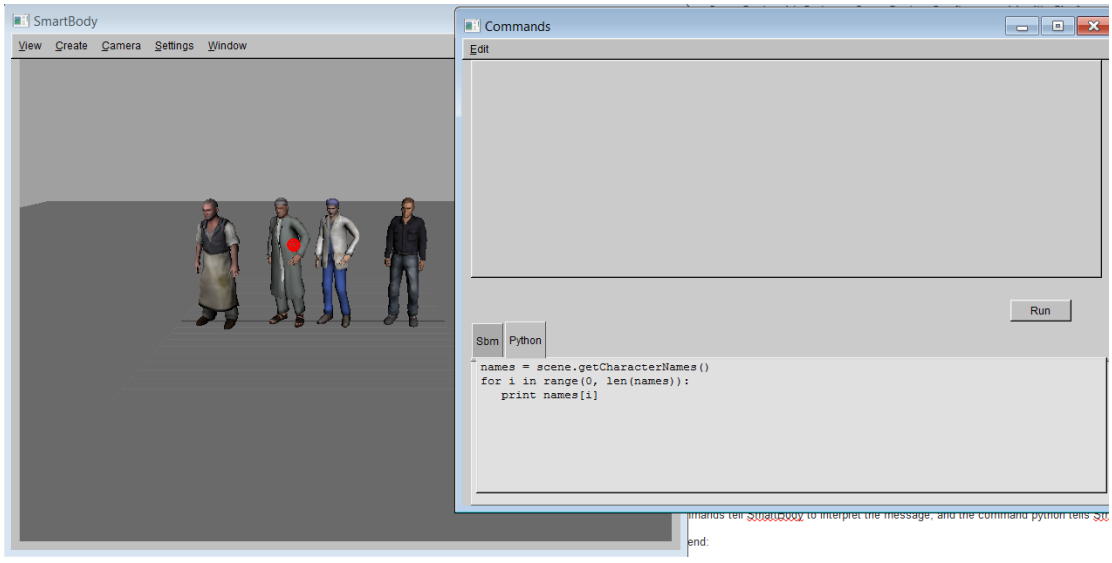
```
sb scene.run(scriptName)
```

where the 'scriptName' file is an existing .py file that is located in SmartBody's asset path.

Sending Python Commands via the SmartBody renderer sbm-fltk

If you are using the default SmartBody renderer (sbm-fltk), you can send a Python command by doing the folllowing:

Choose Window->Command Window then enter the Python command under the tab named 'Python'. To execute the block of Python code, press the 'Run' button.



Sending Python Commands via C++

If you are using a separate renderer (such as Ogre, Unity, etc), Python commands can be sent to SmartBody through the C++ interface.If you are using the smartbody-dll interface, you can send Python commands via the VHMessage interface by using the following functoin:

```
bool ProcessVHMsgs( const char * op, const char * args );
```

where 'op' is "sb" and the second parameter is the Python command. For example:

```
ret = ProcessVHMsgs( "sb", "n = scene.getNumCharacters()");
```

Python API Overview

The Python API uses a module called SmartBody, which includes the following classes:

Class Name	Description
SBObject	Base SmartBody object. Most classes derive from this object.
SBAttribute	Dynamic attributes that can be queried, modified, created and deleted on any SmartBody object.
SBScene	Describes the entire simulation scene, and start point for accessing simulation, BML, resources and other objects needed.
SBPawn	An object that could be a character or other non-living object in the scene.
SBCharacter	A character, subclass of SBPawn.
SBSkeleton	The joint hierarchy of a character
SBJoint	Joint, part of the SBSkeleton
SBService	Services that exist in the scene. Services are non-tangible scene objects that manage various aspects of SmartBody. Examples of services include: physics, steering, bonebus, vhmessages, etc.
SBMotion	An animation for a character or pawn.
SBController	Controller used to manage different behaviors of a character. Examples of controllers are: gaze controller, reaching/grabbing controller, eyelid controller, facial animation controller, etc.
SBAnimationState	Parameterized animation states. Could be a simple animation, or a complex combination of individual motions.
SBAnimationTransition	Transitions between animation states.

SBFaceDefinition	Describe a set of animations or blendshapes used to control facial animation and lip syncing.
SBEventManager	Maintains and controls events that occur within SmartBody
SBSimulationManager	Maintains and controls the simulation flow, including starting, stopping, pausing and changing the simulation frame rate.
SBBmlProcessor	BML (Behavior Markup Language) processor, for sending BML commands.
SBSteerManager	Controls character steering around obstacles.
SBPhysicsManager	Controls and maintains character and pawn physics states.
SBReachManager	Maintains and controls reaching/grabbing and touching behaviors.
SBStateManager	Manages animation states and transitions.
SBGestureManager	Manages gestures for each character.
SBProfiler	Inline performance profiler.

By default, the SmartBody module is imported into the scene, as well as the following objects:

Object	**Description**
scene	instance of SBScene. The scene contains most of the functions needed to access, create and remove all SmartBody-related object. Can also be retrieved via: getScene()
bml	The BML processor, which can be used to send BML commands to various character. Can also be retrieved via: scene.getBMLProcessor()
sim	The simulation manager, which can be used to start, stop and pause the simulation. Can also be retrieved via: scene.getSimulationManager()

Setting Up a SmartBody Scene

A typical SmartBody simulation involves the following steps:

1. Importing assets
2. Configuring the scene
3. Running the simulation

The following is an overview of setting up a SmartBody scene with a few examples of a typical setup.

By default, SmartBody imports the packaged called 'SmartBody" and places an instance of SBScene called 'scene' into the Python context. In order to retrieve the scene object directly, you can call getScene():

```
scene = SmartBody.getScene()
```

The scene object contains many important functions for querying. changing, creating and deleting objects and running the simulation.

Importing Assets

SmartBody can utilize various simulation assets such as skeletons, motions, scripts, model and texture files. In order to use those assets, they must be made known to SmartBody, Skeletons and motions are loaded into memory, whereas many other assets, such as scripts and geometry are loaded as needed.

To indicate where to find the assets, run the following command:

```
scene.addAssetPath(type, path)
```

where type is one of the following types:

Asset Type	Description
motion	Location of animation and skeleton assets.
script	Location of Python scripts.
audio	Location of audio files used for text-to-speech or prerecorded audio.
mesh	Location of geometry, smooth binding/skinning information, and textures

The path can either be an absolute path, or a relative path. Relative paths will be interpreted as extending from SmartBody's media path, which by default is the same as the working directory of the SmartBody executable (set to '.' by default).To change the media path, run:

```
scene.setMediaPath("/path/to/some/where/")
```

For example, if the following motion path was specified:

```
scene.addAssetPath("motion", "foo/mymotions")
```

and the media path was set as above, the the final path for those motions would be: /path/to/some/where/foo/mymotions

Once all the paths are set, the following will load motion and skeleton assets into memory:

```
scene.loadAssets()
```

All motion and skeleton paths Note that skeletons and motions are loaded into memory and will not change after being loaded even if they are modified on disk after the simulation has been started. Scripts can be changed at any time, and are read from the filesystem every time a command to use them is performed.

To load assets from specific directories or folders without changing the asset paths, run:

```
scene.loadAssetsFromPath("/path/goes/here")
```

This would allow partial loading of assets when needed, rather than all at once.

Configuring the Scene

A SmartBody scene can be populated with pawns (generally rigid objects or structures) and characters. In addition, scene configuration includes any non-default settings that SmartBody uses, including responding to events, adding Python scripts that are run during the simulation, turning on the physical simulation, setting up the locomotion/steering options and so forth.

Configuring Pawns

To add pawns to the scene, use:

```
mypawn = scene.createPawn(pawnName)
```

where 'pawnName' is a name unique to both pawns and characters. A pawn can then be positioned in the world by using:

```
pos = SrVec(x, y, z)
mypawn.setPosition(pos)
```

where 'x', 'y', and 'z' are positions in world space. In addition, the orientation of the pawn can be set using:

```
orientation = SrVec(h, p, r)
mypawn.setHPR(orientation)
```

where 'h', 'p', and 'r' are the heading, pitch and roll in degrees of the pawn. A geometry or mesh can also be associated with a pawn, by setting the 'mesh' attribute:

```
mypawn.setStringAttribute("mesh", meshfile)
```

where meshfile is a file or partial path indicating the location of the geometry or mesh file in COLLADA, .obj, or FBX format (if the FBX libraries have been built with SmartBody). To change the size of the mesh, set the meshScale attribute:

```
mypawn.setDoubleAttribute("meshScale", scale)
```

where 'scale' is the scaling factor of the mesh. Note that some SmartBody renderers might choose not to display this mesh in the scene.

Configuring Characters

SmartBody characters have an enormous number of configuration options, as detailed in the following section Configuring Characters. Briefly, you can create a character by using:

```
mycharacter = scene.createCharacter(characterName)
```

where 'characterName' is a name unique to all other characters and pawns in the scene. Characters by default are initialized with a skeleton that has a single joint. To attach a more complex joint hierarchy, create the skeleton and attach it to the character:

```
myskeleton = scene.createSkeleton(skeletonfile)
mycharacter.setSkeleton(myskeleton)
```

where 'skeletonfile' is the name of the file containing a description of the joint hierarchy. The file can be in any of the following formats: .bvh, .asf, COLLADA (.dae or .xml), .fbx, or .sk (SmartBody's proprietary format).

By default, characters do not contain any controllers that allow it to perform complex actions such as lip synching, locomotion, gesturing or head nodding. To add the default set of controllers that allow the character to respond to BML and other commands, do the following:

```
mycharacter.createStandardControllers()
```

Note that additional configuration is also needed to activate some controllers (such as a set of locomotion animations or reaching animations). In addition, many controllers have default settings that can be changed by modifying various parameters on the Character. Please see the section on Configuring Characters for more details.

Character positions and orientations can be set in the same way as for pawns:

```
pos = SrVec(x, y, z)
mypawn.setPosition(pos)
orientation = SrVec(h, p, r)
mypawn.setHPR(orientation)
```

Characters can be attached to geometry using smooth binding/skinning by specifying the directory that contains the skinning mesh and textures then setting the deformableMesh attribute:

```
mycharacter.setStringAttribute("deformableMesh", path)
```

where 'path' is the top level directory under one of the mesh directories that contains a COLLADA (.dae or .xml), FBX or .obj files. Note that the COLLADA file must contain the vertex-joint mappings, and can optionally contain the mesh and locations of textures. If the mesh isn't present in the COLLADA file, then the directory will be searched for .obj files that specify the mesh. The FBX file must contain both the mesh as well as the vertex-joint mappings. Note that some SmartBody renderers may choose not to display the deformable mesh. If the mesh needs to be unifromly scaled, for example to convert a mesh stored in centimeters into meters, the 'deformableMeshScale' attribute can be set to a non-1.0 value:

```
mycharacter.setDoubleAttribute("deformableMeshScale", scaleFactor)
```

where 'scaleFactor' is a multiplier by which all vertices will be multiplied, and '1' is the default.

Running the Simulation

The scene contains a simulation manager called 'sim' (instance of SBSimulationManager) that gets placed into the scene automatically by SmartBody. This can be automatically retrieved from the scene automatically by calling:

```
sim = scene.getSimulationManager()
```

The simulation can be started by calling the start() function:

```
sim.start()
```

and paused, resumed or stepped one frame using the following:

```
sim.pause()
sim.resume()
sim.step(num)
```

where 'num' is the number of steps to run. Note that by default, SmartBody runs using a real-time clock. In order to control the time of the simulation explicitly, you can call the setTime() function using the current time in seconds:

```
sim.setTime(currentTime)
```

Alternatively, SmartBody can be run using simulated time instead of the real-time clock by specifying a simulation

stepping rate:

```
sim.setSimFps(fps)
```

where fps is the number of simulation frames per second (i.e. 60 = 60 frames per second). Setting this value to zero will set SmartBody back into real-time clock mode.

Scene and Simulation Options

There are several options that can be modified in a scene. The following is a list of attributes that can be changed to modify the general behavior of the simulation:

Attribute	Default Value	Description
internalAudio	False	Turns on or off SmartBody's handing of PlaySound messages. If set to True, SmartBody will capture and play the sounds indicated. If set to False, it will ignore PlaySound messages. When using SmartBody as a standalone application (such as via sbm-fltk), this should be set to True. When using SmartBody in combination with a game engine, such as Unity, this should be set to False and the game engine should handle such messages.
speechRelaySoundCacheDir	../../../..	Directory where sound files from speech relays will be placed. The external speech relays transform text into an audio file, and place the resultant sound file in a particular directory. By changing this attribute, the directory can be set to any location.
scale	.01	The scale of the scene in meters. By default, the scene scale is set to centimeters. This attribute is queried during locomotion, steering, camera movement and so forth.

colladaTrimFrames	0	The number of frames to be trimmed when loading a COLLADA motion. This allows you to place the T-pose of the character into the first frame of a motion in order to properly match a motion to a skeleton. SmartBody will ignore the first such frames, and read the remaining data as the animation.
useFastXMLParsing	False	Use faster parsing when reading XML from a file. SmartBody generally uses Xerces for XML parsing. By setting this value to True, SmartBody will use RapidXML for parsing audiofiles for prerecorded speech.
delaySpeechIfNeeded	True	Delays any speech until other behaviors specified in the same BML need to execute beforehand. This can occur when a gesture is synchronized to a word early in the utterance, and the gesture motion needs to be played for awhile before the synch point.

Scripting

Python scripts can be added to SmartBody that are run at various stages of the simulation. A Python script can access any of SmartBody's internal structures through the Python API. Scripts can be used, for example, to query or trigger certain events on a frame-by-frame basis.

Simple Script Example

```
class SimpleScript(SBScript):

def start(self):
print "Starting MyScript..."

def stop(self):
print "Stopping MyScript..."

def beforeUpdate(self, time):
print "Before simulation step..."

def afterUpdate(self, time):
print "After simulation step..."

def update(self, time):
print "During simulation step at time " + str(time)



myscript = SimpleScript()
scene.addScript("simple", myscript)
```

Note that changing the file containing a script will not automatically change the behavior of a script. If a change is made, the script will have to be first removed from the scene:

```
scene.removeScript("simple")
```

then added again:

```
scene.addScript("simple")
```

Motion Editing

SmartBody allows you to change animations that have been loaded as assets. For example, motions can be translated, retimed, retargeted and segments of the motion can be rearranged. Changes made to these animations will be made in memory and will not be saved to disk unless the animation is then explicitly saved. This motion editing functionality can be accessed by calling various methods on the SBMotion class. Motions can be retrieved by name from the scene, as in the following example:

```
motion = scene.getMotion(motionname)
```

Editing Function	Description	Example
translate	Translates the specific joint by x/y/z units	motion.translate(100, 0, 20, "base")
rotate	rotates a specific joint by x/y/z degrees	motion.rotate(45, 0, -30, "base")
scale	scales the translation channels of all joints by a factor	motion.scale(10)
smoothCycle	smooths the cycle of looped animation by fitting intervals in seconds at the beginning and the end together. The mirrored motion will be called 'motionname_smoothcycle'.	motion.smoothCycle(motionname, .1)
mirror	mirrors a motion b y switching right and left joints. Assumes that left joints start with 'l_' and right joints start with 'r_'	motion.mirror(newmotionname, skeletonname)
duplicateCycle	assuming that a motion comprises a single cycle, duplicates the cycle within the motion	newmotion = motion.duplicateCycle(numDupCycles)
alignToBegin	for cycled animation, aligns a motion by moving frames from the end of the motion to the beginning of the motion.	motion.alignToBegin(numFrames)
alignToEnd	for cycled animation, aligns a motion by moving frames from thebeginning of the motion to the end of the motion.	motion.alignToEnd(numFrames)

retime	speeds up or slows down a motion by a scaling factor, where 1 = normal speed, .5 = half speed, 2 = twice speed	motion.retime(factor)
retarget	retargets an animation onto a new skeleton (work in progress). Currently, the t-pose (or a-pose) between each motion will be aligned, and data will be fit onto the new skeleton accordingly.	motion.retarget(motionname, sourceSkeletonName, destSkeletonName, joints)
trim	Tims the starting and ending frames of a motion	motion.trim(5, 23)

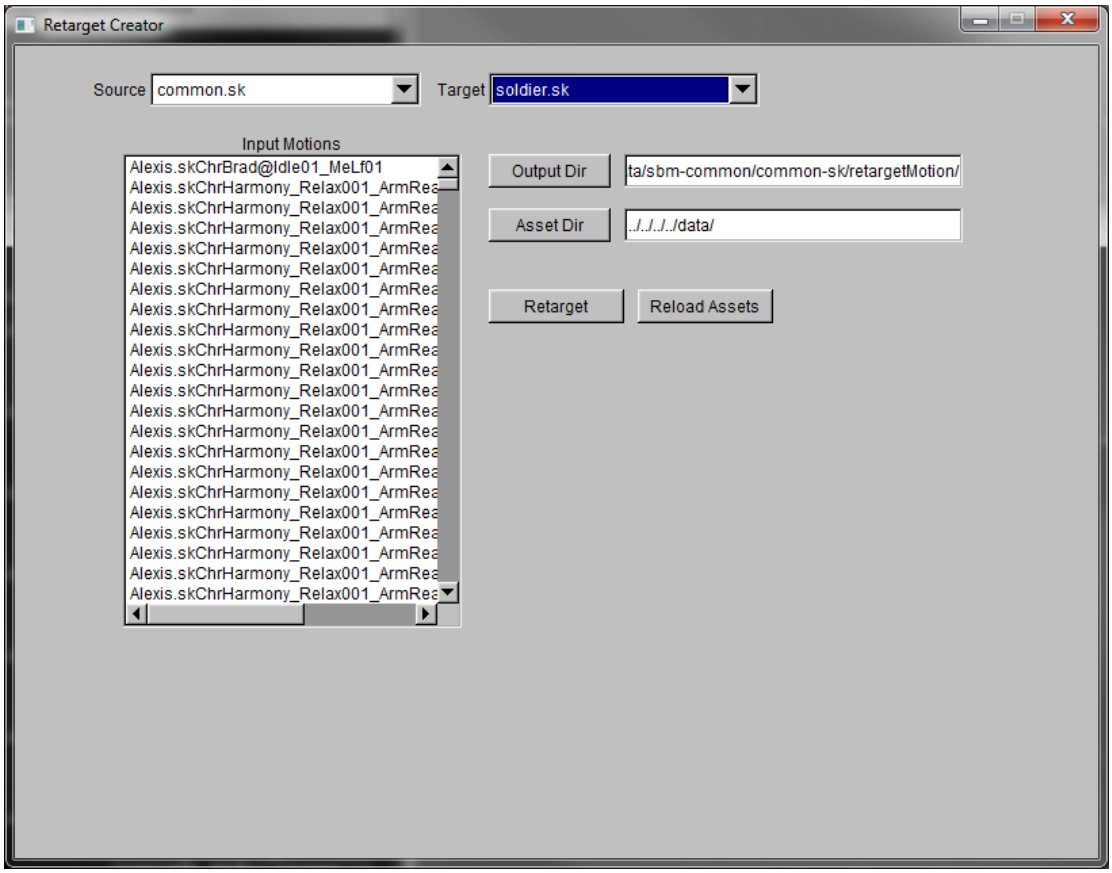
Retargetting

SmartBody includes functionality to retarget or adapt motion to from one character to another. This is an important feature to allow various characters to share animation originally constructed for one character with another.

There are two different interfaces to perform retarget task. One is to use a "Retarget Creator" window in sbm-fltk application. The other is to use SmartBody Python API.

Retarget Creator

Retarget Creator window provides a simple user interface to quickly select the source and target skeleton, and the motions that need to be retargeted. This tool uses heuristics to guess the best mapping between source and target skeletons, and then transfer the joint angles appropriately from source motions to produce new motions for the target skeleton.



As shown from the above image, you need to select the **Source** (skeleton of original motions) and **Target** skeleton (skeleton you want to generate motions for) from the list, and then select the motions corresponding to the source skeleton from the **Input Motions** list. Once they are all selected, press **Retarget** button. This should produce the retargeted motions at ***/data/sbm-common/common-sk/retargetMotion/***.

Perform Retarget Using Python API

User can also utilize the Python API to perform retarget. Since our retarget algorithm require matching pairs between joints from both source and target skeletons, we need to define the joint name mappings for both skeletons. A joint name map can be defined manually, or we have developed a set of heuristic to find these name mapping automatically.

Perform Retarget with Automatic Joint Name Mapping

The following are python commands for retargeting :

scene.run('motion-retarget.py') # need to run the retarget script first before calling the retarget functions in the script file

retargetMotionWithGuessMap(motionName, srcSkelName, tgtSkelName, outDir)

Here ***'motionName'*** indicate the motion corresponds to the source skeleton that needs to be retarget to target skeleton. ***src SkelName*** and ***tgtSkelName*** means the name of source/target skeletons. ***outDir*** is the output directory after retarget is finished.

Perform Retarget with User Defined Joint Name Mapping :

Sometimes the heuristic guess map may not produce desired results. Thus user may also define the joint name mapping themselves. The following are python commands for retargeting :

scene.run('motion-retarget.py') # need to run the retarget script first before calling the retarget functions in the script file

retargetMotionWithMap(motionName, srcSkelName, tgtSkelName, outDir, srcMapName, tgtMapName)

Here ***motionName, srcSkelName, tgtSkelName, outDir*** are defined similarly as in function ***retargetMotionWithGuessM ap. srcMapName*** and ***tgtMapName*** are the joint mapping for source and target skeletons.

Advanced Retarget Command :

The above Python functions provide simplified interface to access the retarget functionality in SmartBody.

(Todo : better explanation of retargeting process in SmartBody. How to use motion.retarget and motion.constrain API to perform advanced retargeting)

Events

SmartBody can be customized to respond to events that occur during the simulation through its own event system. Events can be triggered via a BML message, from markers associated with animations, or through scripting.

An event can be called at any time by creating one from the Event Manager:

```
eventManager = scene.getEventManager()
event = eventManager.createEvent(type, parameters)
```

where 'type' is the event type, and 'parameters' is a string describing the parameters associated with the event. Once an event has been triggered in SmartBody, it will look for an event handler that can process that type of event. An event handler can be created by extending the EventHandler class as follows:

```
class MyEventHandler(EventHandler):

   def executeAction(self, event):
str = event.getParameters()
print "Now executing event with parameters " + str
```

then add the handler to the Event Manager:

```
myHandler = MyEventHandler()
eventManager = scene.getEventManager()
eventManager .addEventHandler("myevents", myHandler)
```

In this example, a handler called 'MyEventHandler' is created, and will respond to 'myevents' events. The handler will extract the parameters from the event, and then print out to the screen some text. The event processing can be more complex, of course, and could perform actions such as play a sound, send a BML request to a character, change the position of the camera, and so forth.

Triggering Events Automatically From Animations

Animations can automatically trigger events when they are played by marking the time, type and parameters on the motion as follows:

```
motion = scene.getMotion(motionName)
motion.addEvent(time, type, parameters, onlyOnce)
```

where 'time' is the local time when the event will occur, 'type' is the event type, 'parameters' is a string describing the event parameters, and 'onlyOnce' is a boolean that determines whether this event will be triggered only once, or every time the motion passes the local time mark. Such animation event markers can be used to play a footstepping sound when the character's foot hits the ground, or to trigger an external event when a motion reaches a certain point, and so forth.

Configuring Characters

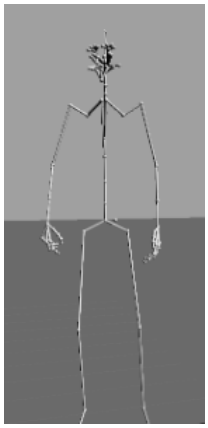
SmartBody characters can perform various skills and tasks if they are configured to do so. Some capabilities require a particular skeleton topology or joint names, while others require a set of configuration data, or motion files. For example, characters can nod, shake or move their heads sideways, but require three joints in the neck and spine named *spine4*, *spine5* and *skullbase* to do so. If the skeleton does not possess such named joints, a joint mapping can be applied to map one joint name to another and thus enable the default capability (see section on Using Custom Skeletons and Animations for more details.) The following is a brief list of capabilities and requirements for them to work:

Capability	Requirements
Head movements: nodding, shaking, tossing	Skeleton with 3 joints in spine and neck named *spine4*, *spine5* and *skullbase*
Gazing	Skeleton with the following joints: *spine1*, *spine2*, *spine3*, *spine4*, *spine5*, *skullbase*, *face_top_parent*, *eyeball_left*, *eyeball_right*
Lip syncing	Face Definition that includes viseme definitions. For joint-driven faces, one motion per viseme and one neutral face motion. For shape-driven faces, only configuration of Face Definition is necessary.
Face movements	Face Definition that includes Action Unit (AU) definitions. For joint-driven faces, one motion per Action Unit and one neutral face motion. For shape-driven faces, only configuration of Face Definition is necessary.
Locomotion	Motion examples of movement that match the character's skeleton.
Idling	Motion files that match the topology of the character's skeleton.
Animations	Motion files that match the topology of the character's skeleton.
Gestures	A gesture mapping and motions files that match the topology of the character's skeleton.
Text-to-speech (TTS)	Either built-in Festival TTS relay, or an external relay using Festival, Microsoft Speech, or any other TTS engine.
Eye saccades	Skeleton with the following joints: *eyeball_left*, *eyeball_right*
Grabbing, touching, reaching	Skeleton topology where shoulder names are: *l_sternoclavicular* and *r_sternoclavicular*. Also, a Reach Configuration
Constraints	None
Physics	Skeleton with root joint called *base*.

Softeyes	Skeleton with the following joints: *eyeball_left*, *eyeball_right*
Blinking	Either a Face Definition that includes a viseme called *blink*, or a Face Definition that includes Action Unit 45 definition. For joint-driven faces, one blink motion and one neutral face motion. For shape-driven faces, only configuration of Face Definition is necessary.
Motion blends/Parameterized animation	Single or multiple motions files that match the topology of the character's skeleton

The Standard SmartBody Skeleton

SmartBody characters can use any joint topology, but certain controllers require particular joint names and relationships. A standard SmartBody character uses the following topology:



- **base**
  - **spine1**
    - **spine2**
      - **spine3**
        - **spine4**
          - **spine5**
            - **skullbase**
              - face_top_parent
                - brow_parent_left
                  - brow01_left
                  - brow02_left
                  - brow03_left
                  - brow04_left
                - ear_left
                - **eyeball_left**
                - upper_nose_left
                  - lower_nose_left
                - lower_eyelid_left
                - upper_eyelid_left
                - brow_parent_right
                  - brow01_right
                  - brow02_right

- brow03_right
- brow04_right
- upper_eyelid_right
- **eyeball_right**
- lower_eyelid_right
- upper_nose_right
- lower_nose_right
- ear_right
- joint18
- face_bottom_parent
- Jaw
- Jaw_back
- Jaw_front
- Lip_bttm_mid
- Lip_bttm_left
- Lip_bttm_right
- Tongue_back
- Tongue_mid
- Tongue_front
- Lip_top_left
- cheek_low_left
- Cheek_up_left
- Lip_out_left
- Lip_top_mid
- Cheek_up_right
- cheek_low_right
- Lip_top_right
- Lip_out_right'

- **l_sternoclavicular**
- l_acromioclavicular
- l_shoulder
- l_elbow
- l_forearm
- l_wrist
- l_pinky1
- l_pinky2
- l_pinky3
- l_pinky4
- l_ring1
- l_ring2
- l_ring3
- l_ring4
- l_middle1
- l_middle2
- l_middle3
- l_middle4
- l_index1
- l_index2
- l_index3
- l_index4
- l_thumb1
- l_thumb2
- l_thumb3

- l_thumb4
- **r_sternoclavicular**
  - r_acromioclavicular
    - r_shoulder
      - r_forearm
        - r_wrist
          - r_pinky1
            - r_pinky2
              - r_pinky3
                - r_pinky4
          - r_ring1
            - r_ring2
              - r_ring3
                - r_ring4
          - r_middle1
            - r_middle2
              - r_middle3
                - r_middle4
          - r_index1
            - r_index2
              - r_index3
                - r_index4
          - r_thumb1
            - r_thumb2
              - r_thumb3
                - r_thumb4
- l_hip
  - l_knee
    - l_ankle
      - l_forefoot
        - l_toe
- r_hip
  - r_knee
    - r_ankle
      - r_forefoot
        - r_toe

Note that only the joints that are in bold are required for all SmartBody controllers to function properly. In addition, skeletons with different joint names can be adapted to SmartBody by creating a skeleton mapping in the section Using Custom Skeletons and Animations.

Using Custom Skeletons and Animations

Any custom skeleton can be read and simulated by SmartBody as long as the skeleton conforms to one of the skeleton asset formats: .bvh, .asf/amc. COLLADA (.dae or .xml) or .fbx. Some SmartBody controllers need to know which joints are involved in a particular action in order to work properly. Thus, the controllers first verity whether or not the skeleton has those needed joints by looking for particular joint names. Thus, it is possible to create a mapping between the custom skeleton that doesn't use the standard SmartBody joint names, and those standard names by creating a joint map. Any number of joint mappings can exists and are managed by the SBJointMapManager:

```
jointMapManager = scene.getJointMapManager()
```

The jointMapManager holds all joint mapping sets, each of which can be dynamically created as follows:

```
mymap = jointMapManager.createJointMap("mymap")
```

Then, a mapping from the custom skeleton to the standard SmartBody skeleton can be done as follows:

```
mymap.setMapping("Hips", "base")
```

Wher 'Hips' is the name of a joint on the custom skeleton, and 'base' is the standard SmartBody joint name (see the section on The Standard SmartBody Skeleton for a list of joint names). The setMapping() method can then be called for each joint that needs to be mapped, for example:

```
mymap.setMapping("Back", "spine1")
mymap.setMapping("Chest", "spine2")
mymap.setMapping("Neck", "spine4")
```

...

Once a mapping has been created, it can be applied to either a skeleton or a motion by using the following commands:

```
mymap.setSkeleton(skeleton)
```

where 'skeleton' is the skeleton object that will be converted using the joint mapping. Note that after the setSkeleton() method is called, all references to joint names in the skeleton will assume the mapped names, and not the original names. Additionally, motions can be converted as follows:

```
mymap.setMotion(motion)
```

where 'motion' is the motion object whose joints matching the joint names in the joint map. Again, after the setMotion() method is called, all joint names in the motion will now be known to SmartBody as the mapped names, and not the original names.

Note that the setSkeleton() and setMotion() methods should be called before those assets are used, preferably before the simulation has been started. Also note that all the SmartBody controllers combined use only a subset of the joints to operate, and thus a minimal mapping would only mapping this subset as described in Configuring Characters and replicated here:

Joint	Required by:
base	SmartBody, all characters need this joint.
spine1	gaze
spine2	gaze
spine3	gaze
spine4	gaze, head movements
spine5	gaze, head movements

skullbase	gaze, head movements
face_top_parent	gaze
eyeball_left	gaze, saccades, softeyes
eyeball_right	gaze, saccades, softeyes
l_sternoclavicular	reaching
r_sternoclavicular	reaching

Notwithstanding controller functionality, any skeletons and motions that have similar joint names can be automatically retargeted during runtime by copying motion curves over the joints. In other words, if your skeleton has the same joint names a motion, that motion can be applied to your skeleton automatically.

How To Export Skeleton/Animation From 3dsMax or Maya

Skeletons and motions can be exported from Maya or 3dsMax using COLLADA format. We recommend using OpenCollada plugin (http://www.opencollada.org/download.html) to export a Maya/3dsMax scene to COLLADA. Note that the current version of OpenCollada plugin does not support Maya 2012 yet, so you need to use the DAE_FBX plugin that came with Maya 2012.

The overall process is similar when exporting from either Maya and 3dsMax :

> go to **Export** Menu, and select output file type as **OpenCOLLADA**, be sure to select **Enable export** for **Animation**. This will output a .dae file that contains both the skeleton and the corresponding motion.

There are also a few *requirements and caveats* that need to be considered when exporting the animations :

1. The first frame (frame 0) needs to be the default **bind pose** (usually called T-pose, depending on your skeleton setup). This ensures that the output bone transformations would be correct. Otherwise, the bone transformations would be relative to the first frame of the animation and the motion would look strange when applied on the T-pose skeleton in SmartBody.

2. 3dsMax allows you to name your joints with spaces, however it is preferable that the joint name does **NOT** contai n any spaces. Some COLLADA exporter may not process this correctly (i.e. The default Autodesk COLLADA exporter) and may cause incorrect results when loading the file into SmartBody. If your skeleton contains joint names with spaces, you need to use OpenCOLLADA plugin for exporting. As it would take care of this by replace space (" ") with underline ("_").

3. If the first frames of your motion is the T-pose, they need to be skipped when loading the motions in SmartBody. This can be done by either :
    a. Post-processing the loaded motion with Python command :

    ```
    testMotion.trim(n,0)
    ```

    This command will trim the first n frames of testMotion so the resulting motions will not contain T-pose.

    b. Set a global attributes to trim the frames when loading a COLLADA motion :

    ```
    scene.setIntAttribute("colladaTrimFrames", n)
    ```

This will automatically trim the first n frames when loading a COLLADA motion. Note that this attribute needs to be set **before** calling scene.loadAssets()

Use Skeleton/Character From Mixamo

Mixamo (http://www.mixamo.com) let you quickly create 3D characters/animations and download it into various format. SmartBody provides the necessary script to quickly create a standard SmartBody character from mixamo character. This enable our users to create a custom character from mixamo, and then use SmartBody to produce various behaviors like locomotion/reach/gaze for the character.

The following is the step-by-step instructions to download a specific '**Football Player**' character from mixamo website and get it working in SmartBody. The same procedure should work for other characters on their website.

1. On the mixamo website, select the free character **'Football Player'** in T-pose.
2. Download the character T-pose into **Collada (.dae)** as well as **FBX** format. Here, we save the file name as **player.dae** and **player.fbx**
3. If you just want to use **SmartBody fltk-viewer** :
    a. Put the Collada file into ***$SmartBodyDir /data/sbm-common/common-sk/*** so SmartBody can load it as a *skeleton*. To use *deformable mesh*, you also need to put both the .dae file and the texture files into ***$SmartBodyDir/data/mesh/player*** directory.
    b. Modify the **default-init-mixamo.py** accordingly to accommodate for your character names. (If you follow the instructions to download the **Football Player** and name is **'player.dae'**, the script should work out of box.)
    c. Run fltk-viewer with argument *"-scriptpath ../../../../data/sbm-common/scripts -script default-init-mixamo.py"*

1. If you want to use OgreViewer :
    a. Be sure to do step 3 correctly so SmartBody can load the skeleton.
    b. install **Ogre Exporter** for Maya. (http://www.ogre3d.org/tikiwiki/Maya+Exporter)
    c. Open the FBX file in Maya, and use Ogre Exporter to save out **player.mesh**, **player.material**, and **player.skeleton**.
    d. Put the ogre files in corresponding directory ***$SmartBodyDir/core/ogre-viewer/media***.
    e. Modify the **default-init-mixamo.py** script. Specifically, make sure when you are creating a character, **the character name is the same as your mesh name** (ex. *scene.createCharacter('player')*) This will enable the OgreViewer to use **'player.mesh'** instead of the default **'brad.mesh'**.
    f. Run OgreViewer with the argument *"-scriptpath ../../../../data/sbm-common/scripts -script default-init-mixamo.py"*

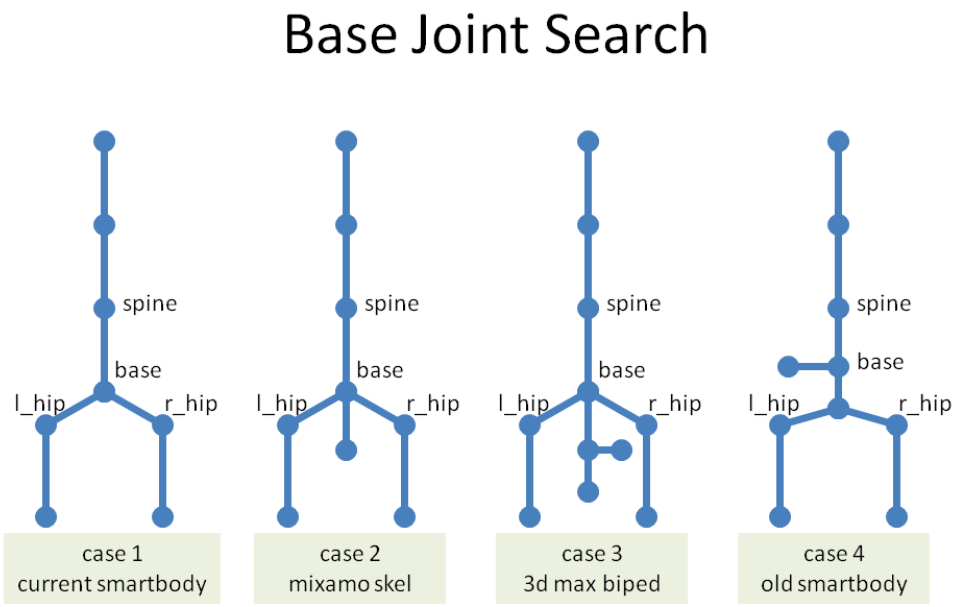
There is one pitfall when downloading the mixamo character. Sometimes they add a prefix to the joint name ("Beast_LeftUpLeg" instead of "LeftUpLeg" for example). This will cause incorrect mapping for the bone names and thus the retarget will fail. For now the solution is to remove those joint name prefix manually from the file for this kind of characters, or modify the joint name mapping in '**mixamo-map.py**' ***to accommodate for the prefix.***

Automatic Skeleton Mapping

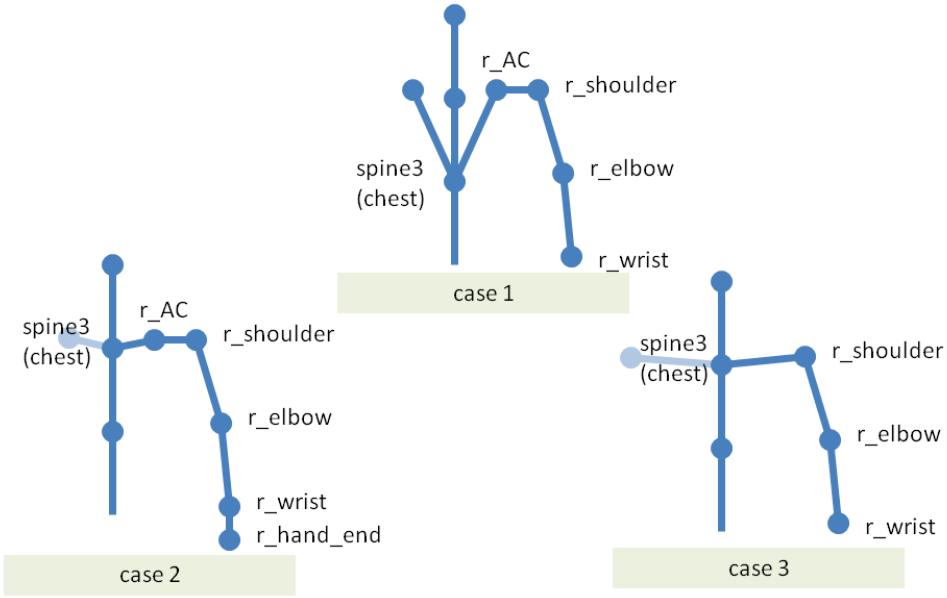
SmartBody can automatically adapt a character to the standard SmartBody skeleton, and thus allow a character to use the SmartBody controllers (such as gazing, head nodding) as well as the default SmartBody animations if the character were to be retargeted. This is done by calling the guessMapping() function on a joint map. For example:

```
mycharacter = scene.getCharacter(characterName)
myskeleton = mycharacter.getSkeleton()
jointMapManager = scene.getJointMapManager()
sbJointGuessMap = jointMapManager.createJointMap(characterName+"-GuessMapping", True) # True to display
mapping, False to hide this output
sbJointGuessMap.guessMapping(myskeleton)
sbJointGuessMap.applySkeleton(myskeleton)
```

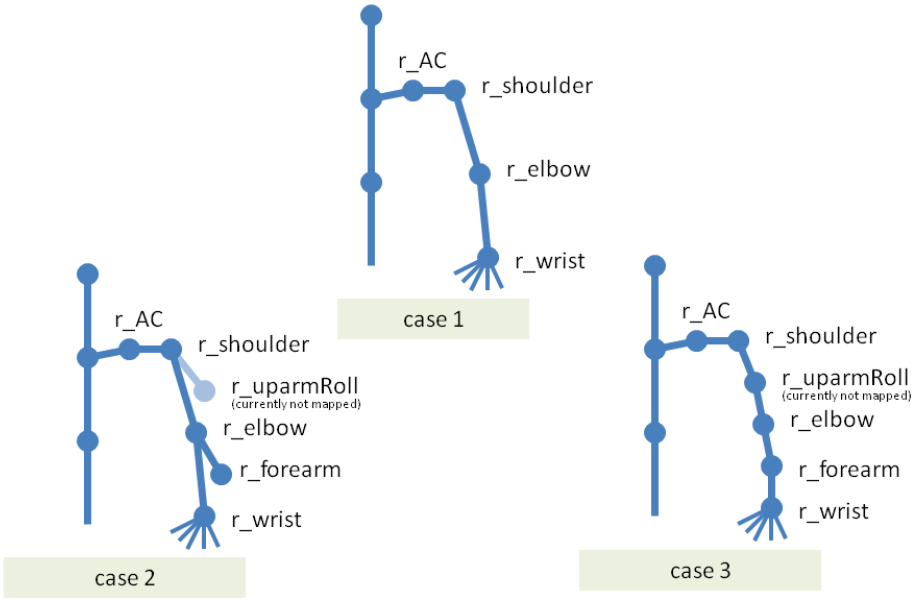
The automatic mapping is done by searching the topology and examining the bone/joint names. Note that the automatic mapping will not work for skeletons that aren't similar to a bipedal, since their topology and joint names would be different.. The images below show how the algorithm finds matches among some commonly used skeletons:



Spine/Arm Joint Search



Arm Joint Search



Keywords Used In Searches

Joint to search	Search keyword	Alternative keyword
skullbase	head, skull	
l_wrist	hand, wrist	
l_ankle	ankle, foot	
l_thumb	thumb, finger0	pollex
l_index	index, finger1	pointer, forefinger
l_middle	middle, mid, finger2	medius
l_ring	ring, finger3	fourth
l_pinky	pinky, finger4	little
eyeball_left	eye (but not lid, brow, lash)	

Configuring Facial Animations and Lip Syncing

SmartBody characters are able to change the expression on their faces and perform lip syncing with prerecorded utterances or utterances generated from a text-to-speech engine.SmartBody uses Facial Action Units (FACS) to generate facial expressions, and uses a procedurally-driven lip syncing algorithm to generate lip synching. In addition, lip synching can be customized for prerecorded audio based on the results, for example, of 3rd party facial animation software.

The ability of a character to perform different facial expressions is determined by pose examples that are are supplied for each character. Poses can be used for FACS units in order to generate facial expressions, or for visemes which can be used to generate parts of a lip syncing sequence.

SmartBody can generate facial expressions for either joint-driven faces, or shape-driven faces.SmartBody will generate the motion for joint-driven face configurations. For shape-driven face configurations, SmartBody will determine the activation values that will be interpreted by a renderer that manages the face shapes.

SmartBody uses a hierarchical scheme for applying animation on characters. Thus, more general motions, such as full-body animation, are applied first. Then, more specific animation, such as facial animation, is applied to the character and overrides any motion that previously controlled the face. Thus, SmartBody's facial controller will dictate the motion that appears on the character's face, regardless of any previously-applied body motion.

Please note that there is no specific requirements for the topology or connectivity of the face; it can contain as many or as few as desired. In addition, complex facial animations can be used in place of the individual FACS units in order to simplify the facial animation definition. For example, a typical smile expression might comprise several FACS units simultaneously, but you might want to create a single animation pose to express a particular kind of smile that would be difficult to generate by using several component FACS units. To do so, you could define such a facial expression with a motion, then attach that motion to a FACS unit, which could then be triggered by a BML command. For example, a BML command to create a smile via individual FACS units might look like this:

```
<face type="FACS" au="6" amount=".5"/><face type="FACS" au="12" amount=".5"/>
```

whereas a BML command to trigger the complex facial expression might look like this, assuming that FACS unit 800 has been defined by such an expression:

```
<face type="FACS" au="800" amount=".5"/>
```

Details

Each character requires a Face Definition, which includes both FACS units as well as visemes. To create a Face Definition for a character, use the following commands in Python:

```
face = scene.createFaceDefinition("myface")
```

Next, a Face Definition requires that a neutral expression is defined, as follows:

```
face.setFaceNeutral(motion)
```

where motion is the name of the animation that describes a neutral expression. Note that the neutral expression is a motion file that contains a single pose. In order to add FACS units to the Face Definition, use the following:

```
face.setAU(num, side, motion)
```

where *num* is the number of the FACS unit, *side* is "LEFT", "RIGHT" or "BOTH", and *motion* is the name of the animation to be used for that FACS unit. Note that the FACS motion is a motion file that contains a single pose. For shape-driven faces, use an empty string for the *motion* (""). Note that LEFT or RIGHT side animations are not required, but will be used if a BML command is issued that requires this. Please note that any number of FACS can be used. The only FACS that are required for other purposes are FACS unit 45 (for blinking and softeyes). If you wanted to define an arbitrary facial expression, you could use an previously unused num (say, num = 800) which could then be triggered via BML.

To define a set of visemes which are used to drive the lip animation during character speech, use the following:

```
face.setViseme(viseme, motion)
```

where viseme is the name of the *viseme* and *motion* is the name of the animation that defines that viseme. Note that the viseme motion is a motion file that contains a single pose. For shape-drive faces, use an empty string for the *motion* (""). Please note that the name of visemes will vary according to the text-to-speech or prerecorded audio component that is connected to SmartBody.

Lip Syncing text-to-speech or prerecorded audio

SmartBody can use two different lip syncing models that require different sets of visemes a high quality, diphone-based method, and a low-quality pose-based method.

High Quality Lip Syncing Using the Diphone Method

The recommended method is to use SmartBody's diphone-based lip syncing scheme. By default, the Speech Relays (Microsoft, Festival, CereProc, etc.) use this method. In this method, lip syncing is accomplished by mapping phonemes (word sounds) to a common set of phonemes, and then combined with a set of artist-created animations for each pair of phonemes.The results are well synchronized with the audio.

The recommended use is to use the following set of visemes (which are compatible with the FaceFx software) and can be used both for text-to-speech and for prerecorded audio:

open, W, ShCh, PBM, fv, wide, tBack, tRoof, tTeeth

Pose	Description
open	

W	
ShCh	
PBM	

Fv	
wide	
tBack	

tRoof	
tTeeth	

To enable the diphone-based lip syncing, load the included diphone animations once:

```
scene.run("init-diphoneDefault.py")
```

this will create the default diphone set that can be used by any character. Then set the following attributes on your character:

```
mycharacter = scene.getCharacter(name)
mycharacter.setBoolAttribute("useDiphone", True)
mycharacter.setBoolAttribute("diphoneSplineCurve", True)
mycharacter.setDoubleAttribute("diphoneSmoothWindow", .2)
mycharacter.diphoneSetName("default")
```

Low Quality Lip Syncing Using Static Facial Poses

SmartBody can also use a lower-quality lip animation method that uses a small set of visemes. In this method, phonemes are mapped directly to a limited set of facial shapes, then the face poses are interpolated according to the timing of the phonemes.By default, the TTS relays (Microsoft, Festival, CereProc, etc.) do not use this method, and need to be launched with the parameter:

```
-mapping sbmold
```

in order to interpret phonemes to this set.

This method uses the following visemes:

Ao, D, EE, Er, f, j, KG, Ih, NG, oh, OO, R, Th, Z

Please note that visemes are not triggered via BML the way that FACS units are triggered. Visemes are typically only triggered in response to TTS or prerecorded audio.

Face Definition

Once the FACS units and visemes have been added to a Face Definition, the Face Definition should be attached to a character as follows:

```
mycharacter = scene.getCharacter(name)
mycharacter.setFaceDefinition(face)
```

where *name* is the name of the character. Note that the same Face Definition can be used for multiple characters, but the final animation will not give the same results if the faces are modeled differently, or if the faces contain a different number of joints.

The following is an example of the instructions in Python that are used to define a joint-driven face:

```
face = scene.createFaceDefinition("myface")
```

```
face.setFaceNeutral("face_neutral")
face.setAU(1, "LEFT", "fac_1L_inner_brow_raiser")
face.setAU(1, "RIGHT", "fac_1R_inner_brow_raiser")
face.setAU(2, "LEFT", "fac_2L_outer_brow_raiser")
face.setAU(2, "RIGHT", "fac_2R_outer_brow_raiser")
face.setAU(4, "LEFT", "fac_4L_brow_lowerer")
face.setAU(4, "RIGHT", "fac_4R_brow_lowerer")
face.setAU(5, "BOTH", "fac_5_upper_lid_raiser")
face.setAU(6, "BOTH", "fac_6_cheek_raiser")
face.setAU(7, "BOTH", "fac_7_lid_tightener")
face.setAU(9, "BOTH", "fac_9_nose_wrinkler")
face.setAU(10, "BOTH", "fac_10_upper_lip_raiser")
face.setAU(12, "BOTH", "fac_12_lip_corner_puller")
face.setAU(15, "BOTH", "fac_15_lip_corner_depressor")
face.setAU(20, "BOTH", "fac_20_lip_stretcher")
face.setAU(23, "BOTH", "fac_23_lip_tightener")
face.setAU(25, "BOTH", "fac_25_lips_part")
face.setAU(26, "BOTH", "fac_26_jaw_drop")
face.setAU(27, "BOTH", "fac_27_mouth_stretch")
face.setAU(38, "BOTH", "fac_38_nostril_dilator")
face.setAU(39, "BOTH", "fac_39_nostril_compressor")
face.setAU(45, "LEFT", "fac_45L_blink")
face.setAU(45, "RIGHT", "fac_45R_blink")


# higher quality, diphone-based lip syncing
face.setViseme("open", "viseme_open")
face.setViseme("W", "viseme_W")
face.setViseme("ShCh", "viseme_ShCh")
face.setViseme("PBM", "viseme_PBM")
face.setViseme("fv", "viseme_fv")
face.setViseme("wide", "viseme_wide")
face.setViseme("tBack", "viseme_tBack")
face.setViseme("tRoof", "viseme_tRoof")
face.setViseme("tTeeth", "viseme_tTeeth")

# lower quality, simple viseme lip syncing
'''
face.setViseme("Ao", "viseme_ao")
face.setViseme("D", "viseme_d")
face.setViseme("EE", "viseme_ee")
face.setViseme("Er", "viseme_er")
face.setViseme("f", "viseme_f")
face.setViseme("j", "viseme_j")
face.setViseme("KG", "viseme_kg")
face.setViseme("Ih", "viseme_ih")
face.setViseme("NG", "viseme_ng")
face.setViseme("oh", "viseme_oh")
face.setViseme("OO", "viseme_oo")
face.setViseme("R", "viseme_r")
face.setViseme("Th", "viseme_th")
face.setViseme("Z", "viseme_z")
face.setViseme("BMP", "viseme_bmp")
face.setViseme("blink", "fac_45_blink")

  '''

mycharacter = scene.getCharacter(name)

mycharacter.setFaceDefinition(face)
```

Configuring Blinking

SmartBody characters will automatically blink at an interval between 4 and 8 seconds. The blinking is triggered by activating FACS units 45 LEFT and 45 RIGHT. Thus, in order for characters to blink, they must have those FACS units defined. Please consult the section on Configuring Visemes and Facial Animations for more details on how to set up a Face Definition that contains those FACS units.

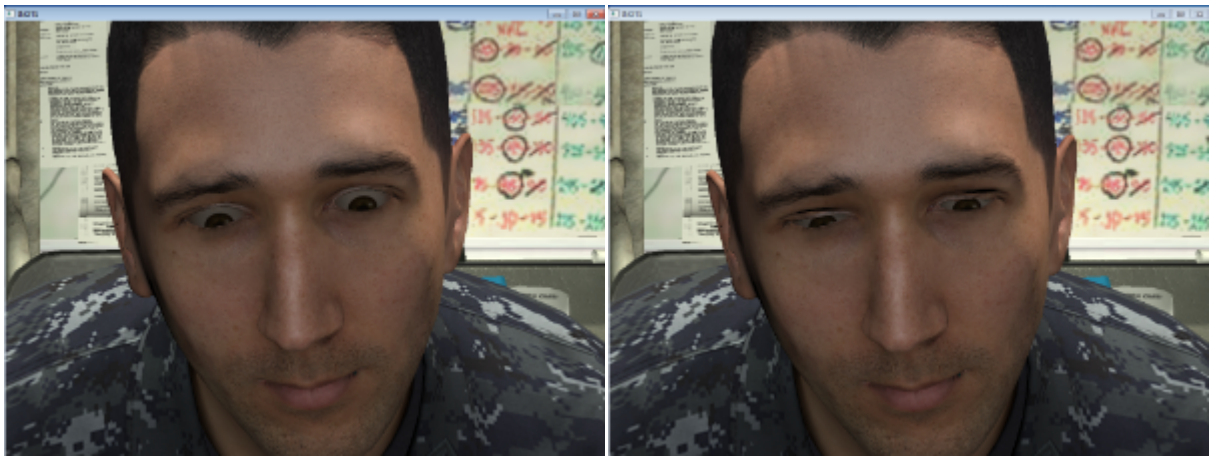
In order to change the blinking interval, the attributes eyelid.blinkPeriodMin and eyelid.blinkPeriodMax can be set on the

character. For example:

```
mycharacter = scene.getCharacter(name)
mycharacter.setDoubleAttribute("eyelid.blinkPeriodMin", 5)
mycharacter.setDoubleAttribute("eyelid.blinkPeriodMax", 9)
```

Configuring Soft Eyes

When people move their eyes, the upper and lower lids tend to follow the eye such that the upper lid goes upwards when the eye moves upwards, and the lower lid moves downwards when the eye moves downwards.



The above images illustrate the 'softeye' effect. On the left, the character is looking downward. Without softeyes, the character's lids remain at their original position. On the right is a character using the softeye correction - the upper eyelid is lowered to just above the iris.

SmartBody leverages the blinking mechanism in order to adjust the eyelids. The softeye effect is produced by activating the blink expression until the upper lid falls to the proper eye level. Thus, characters that use softeyes must have a Face Definition and FACS unit 45 left and right define. Please see the section on Configuring Visemes and Face Animations for more detail. Since SmartBody regulates the FACS unit 45 for both blinking and softeyes, blinks will work appropriately when the softeye feature is enabled. For example, eyes that appear half-closed will only perform half-blinks, etc.

In the future, SmartBody will enhance the softeyes feature to include movement of the lower eyelid. This is important when the eyes are looking downward, where ordinarily the lower lid would drop slightly to accomodate such a movement. Since SmartBody can only modulate the amount by which an eye is fully or partially closed with a blink, a character's eyes might appear to be closed when they are looking downward, since the lower lid would be instructed to move upwards (when blinking) instead of moving downwards to make room for the eye.

The following attributes on the character can be set to adjust the softeyes:

Attribute	Description
`eyelid.softeyes`	Determines whether to use the softeyes feature or not. Default is set to True
`eyelid.rangeUpperMin`	The minimum pitch of the eyelids in degrees. Default is -30.

`eyelid.rangeUpperMax`	The maximum pitch of the eyelids in degrees. Default is 30.
`eyelid.tightWeightUpper`	Lid tightening. How far down the eyeball to to drop the pupil, from 0 to 1. Setting this value to > 0 will result in a 'sleepy' or 'drunk' look to the character. A value of 1 means that the eyelid will drop to the level of the pupil. Default is 0.
`eyelid.delayUpper`	Delay between the movement of the eyes and the movement of the eyelids, from 0 to 1. A zero value indicates that the lids will respond immediately to the movement of the eyes. A one value indicates significant delay before the lids adjust to the eye location. Default is .3.
`eyelid.closeAngle`	The angle at which the eyelid is considered to be closed in degrees. Default is 30.

Configuring Gazing

SmartBody characters that use gazing need to have the following joints:

Joint Name

Joint Name	Gaze part
spine1	Back
spine2	Back
spine3	Chest
spine4	Chest
spine5	Chest
skullbase	Neck
eyeball_left	Eyes
eyeball_right	Eyes

If a character lacks those joint names (or joints that are mapped to those names), then the gaze controller will be unable to properly function. For details on mapping skeletons to match the SmartBody format, see the section on Mapping Skeletons to SmartBody.

The gaze controller sets limits on the amount of movement for each joint that can be directed by the gaze. These limits can be adjusted by setting the following attributes on the character:

Attribute	Default Value	Description
gaze.speedEyes	1000	Relative speed of the eyes during gazing. The default value of 1000 is set to approximate the speed of normal human eye movement. Thus, to move the eyes half as fast, set this to 500, or twice as fast,set this to 2000.
gaze.speedNeck	1000	Relative speed of the neck during gazing. The default value of 1000 is set to approximate the speed of normal neck movement. Thus, to move the neck half as fast, set this to 500, or twice as fast,set this to 2000.
gaze.limitPitchUpEyes	-35	The upper pitch limit of the eyes (X-axis)

gaze.limitPitchDownEyes	35	The lower pitch limit of the eyes (X-axis)
gaze.limitHeadingEyes	40	The heading limits of the eyes (Y-axis)
gaze.limitRollEyes	0	The roll limits of the eyes (Z-axis)
gaze.limitPitchUpNeck	-45	The upper pitch limit of the neck (X-axis)
gaze.limitPitchDownNeck	45	The lower pitch limit of the neck (X-axis)
gaze.limitHeadingNeck	90	The heading limits of the neck (Y-axis)
gaze.limitRollNeck	35	The roll limits of the neck (Z-axis)
gaze.limitPitchUpChest	-6	The upper pitch limit of the chest (X-axis)
gaze.limitPitchDownChest	6	The lower pitch limit of the chest (X-axis)
gaze.limitHeadingChest	15	The heading limits of the chest (Y-axis)
gaze.limitRollChest	5	The roll limits of the chest (Z-axis)
gaze.limitPitchUpBack	-15	The upper pitch limit of the back (X-axis)
gaze.limitPitchDownBack	15	The lower pitch limit of the back (X-axis)
gaze.limitHeadingBack	30	The heading limits of the back (Y-axis)
gaze.limitRollBack	10	The roll limits of the back (Z-axis)

For example, to make the speed of the eye movement 20% faster, do:

```
mycharacter = scene.getCharacter(name)
mycharacter.setDoubleAttribute("gaze.speedEyes", 1200)
```

Configuring Breathing

SmartBody characters can modulate their breathing. Breathing is implemented either by the use of shapes controlled by the rendering engine, or by using a joint-based motion that is overlayed on top of a character. Characters can control the breathing rate, as well as the minimum and maximum respiratory volumes.

SmartBody controls character breathing by creating a breathing cycle and modulating the rate at which it is played.

There are several attributes that can be set on a character that control breathing:

Attribute	Comment
`breathing.motion`	The name of the motion to be used as an animation overlay to control breathing. Typically, this animation would deform a set of joints that define the stomach and possibly the rib cage. The motion should contain a single breathing cycle that can be looped. SmartBody will modulate the effect of the breathing overlay on the character. Default is ""
`breathing.useBlendChannels`	Determines whether to add two channels, breath_x and breath_y to the character's channel array, rather than using a breathing motion overlay. The breathX channel indicates the respiratory volume. The breathY channel indicates the normalized breathing phase value (from 0 to 1). Default is False
`breathing.bpm`	Number of breaths per minute. Default is 15

Configuring Gestures

SmartBody characters can use gesturing as well as speech to communicate. Gestures are animations associated with an underlying body posture. Gestures differs from playing animations directly in that a BML command for a gesture includes the gesture type, handedness and style. These characteristics are then mapped to the appropriate underlying animation automatically through gesture maps, without the user needing to know which animation is appropriate for the character at that time. For example, using the BML command:

<gesture type="YOU" hand="RIGHT_HAND"/>

would trigger the 'YOU' gesture for a character using the right hand. The gesture map would find the animation associated with a right-handed 'YOU' gesture, determine which posture the character is in, then play that animation. The purpose of the gesture map is to simplify the gesturing commands for the user, since otherwise they would have to know which animation is associated with what posture.

Note that the gesture mapping places the burden of mapping animations to gestures on the frees the simulation user to specify gestures, and places the burden of matching gestures to animation on the simulation designer.

Setting Up a Gesture Map

To create a new gesture map:

```
mymap = scene.createGestureMap(characterName)
```

where 'characterName' is the name of the character for which the gesture map will be created. Then to add gestures:

```
mymap.addGestureMapping(animationName, type, lexeme, posture, hand)
```

where 'animationName' is the name of the animation to be played, 'type' is a variation of the gesture (can be blank), 'lexeme' is the kind of gesture(say, 'POINT', or 'WAVE', or 'YOU' or 'ME', an arbitrary string describing the gesture), 'posture' is the name of the underlying idle pose, and 'hand' is either 'RIGHT_HAND', 'LEFT_HAND', or 'BOTH_HANDS'.

For example, the following will create a gesture map for every character in the scene:

```
gMapManager = getScene().getGestureMapManager()
numCharacters = scene.getNumCharacters()
charNames = scene.getCharacterNames()
for i in range(0, numCharacters):
gMap = gMapManager.createGestureMap(charNames[i])
gMap.addGestureMapping("HandsAtSide_Arms_Sweep", "", "SWEEP", "HandsAtSide_Motex", "BOTH_HANDS", "")
gMap.addGestureMapping("HandsAtSide_RArm_GestureYou", "", "YOU", "HandsAtSide_Motex", "RIGHT_HAND", "")
gMap.addGestureMapping("LHandOnHip_Arms_GestureWhy", "", "WHY", "LHandOnHip_Motex", "BOTH_HANDS", "")
gMap.addGestureMapping("LHandOnHip_RArm_GestureOffer", "", "OFFER", "LHandOnHip_Motex", "RIGHT_HAND", "")
gMap.addGestureMapping("LHandOnHip_RArm_SweepRight", "", "SWEEP", "LHandOnHip_Motex", "RIGHT_HAND", "")
gMap.addGestureMapping("LHandOnHip_RArm_SweepRight", "", "YOU", "ChrUtah_Idle003", "RIGHT_HAND", "")
```

Configuring Locomotion, Steering and Path Finding

Locomotion

Steering

The steering system uses a grid-based approach to find collision-free paths to the target goal. The steering engine uses the SteerSuite framework (http://www.magix.ucla.edu/steersuite/) and the PPR (Plan, Predict, React) agent to accommodate steering for both static (unmoving) and dynamic (moving) obstacles. A number of steering options can be set on the SBSteeringManager object as follows:

Attribute	Description
gridDatabaseOptions.gridSizeX	Size of the grid in the X-direction.
gridDatabaseOptions.gridSizeZ	Size of the grid in the Z-direction.
gridDatabaseOptions.numGridCellsX	Number of grid cells in the X-direction.
gridDatabaseOptions.numGridCellsZ	Number of grid cells in the Z-direction.
gridDatabaseOptions.maxItemsPerGridCell	The maximum number of pawns or objects that can occupy a grid cell. Default is 7.

initialConditions.radius	Radius of the characters when performing steering. This can be set to a very small number (say, .001) if you do not wish to consider characters as obstacles during the steering calculation.

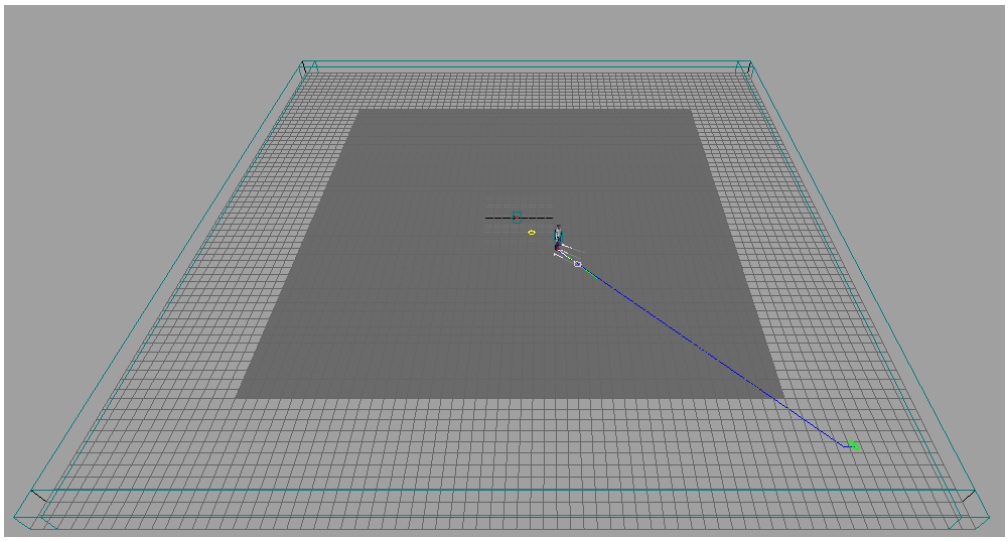
Once the parameters have been set, the steering manager must be restarted, for example:

```
steer = scene.getSteerManager()
steer.setDoubleAttribute("gridDatabaseOptions.gridSizeX", 200)
steer.setDoubleAttribute("gridDatabaseOptions.gridSizeZ", 200)
steer.stop()
steer.start()
```

Note that characters who are instructed to move to spaces outside of the steering grid will not do so, and any characters who reach the boundary of the steering grid will immediately stop movement. There is a set of walls that are automatically constructed on the edges of the steering grid to help guide the locomotion. To remove these walls:

```
steer.setBoolAttribute("useBoundaryWalls", False)
```

The image below shows the default grid size, cell size and border boundaries.



Path Finding

SmartBody can utilize a simple path finding without obstacle avoidance by setting the 'steering.pathFollowingMode' attribute on a character:

```
character = scene.getCharacter(characterName)
character.setBoolAttibute("steering.pathFollowingMode", True)
```

In the path following mode, the character will follow a path as described by a BML locomotion command as closely as possible. There are several attributes on the character that determine how fast the character will move along that path: 'steering.pathMinSpeed', 'steering.pathMaxSpeed', 'steering.pathAngleAcc':

```
character.setDoubleAttribute("steering.pathMinSpeed", minSpeed)
character.setDoubleAttribute("steering.pathMaxSpeed", maxSpeed)
character.setDoubleAttribute("steering.pathAngleAcc", angularAcceleration)
```

where 'minSpeed' is the minimum speed at which the character will move, regardless if the path cannot be followed very closerly. Likewise, 'maxSpeed' is the desired speed of the that the character will move except when it is necessary to move more slowly in order to follow the path more accurately. 'angularAcceleration' is a gain describing how quickly the character will turn.

A BML command specifying locomotion will look something like this:

```
<locomotion target="waypoint1 waypoint2 waypoint3 waypoint4 waypoint5"/>
```

where 'waypointx' are pawns or characters. Likewise, global coordinates could be used instead like so:

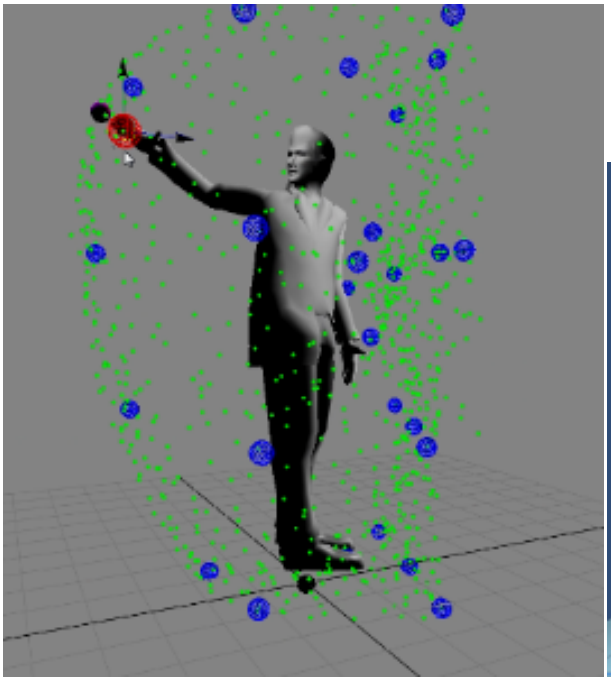
```
<locomotion target="0 100 20 35 200 -400 67 100"/>
```

where each pair of numbers represent an (x,z) waypoint. Likewise, a combination of numbers and pawns/characters can be used:

```
<locomotion target="0 100 20 35 waypoint1 67 100"/>
```

Configuring Reaching, Grabbing,Touching and Pointing

SmartBody characters have the ability to interact with objects in the environment by using a set of example-based motions for reaching, and a set of hand poses for grabbing and touching.



In the first image above, the blue spheres represent a set of reaching examples, while the green dots represent interpolated examples synthesized from the original examples. The character can reach, grab or touch anywhere in that space. In the second image on the right, the character uses a set of reaching examples while in a sitting posture, which allows the character to grab any object on the table and place it anywhere else.

BML is used to instruct the character to reach, grab or touch an object or point in the world. For more details on how to instruct characters to do this, see the description of commands for Reach, Grab, Touch, Point.

Setting Up Reaching/Grabbing/Touching

A reaching setup requires:

- A set of reaching motions for each idle pose (standing vs sitting vs running)
- A set of hand configurations for reaching, grasping and releasing, pointing
- A shape configured for a pawn that is the target of the reaching/grabbing/touching.

For each character, do the following:

Get the Reach Manager:

```
reachManager = scene.getReachManager()
```

Create a reaching set for a particular character:

```
reach = reachManager.createReach(characterName)
```

then add the motions for that particular reach:

```
reach.addMotion(hand, motion)
```

where 'hand' is "left" or "right" describing which hand is performing the motion, and 'motion' is the motion object that can be obtained from the scene as follows:

```
motion1 = scene.getMotion(motionName)
```

The handedness will be used by SmartBody when determining which side of the body to use based on proximity. Next, configure the hands for three separate modes: (1) reaching, (2) grasping, and (3) releasing as follows:

```
reach.setReachHandMotion(hand, reachingMotion)
```

where 'hand' is "left" or "right" and 'reachingMotion' is a pose describing the hand during a reach. Grasping and releasing need to be configured as:

```
reach.setGrabHandMotion(hand, grabbingMotion)
reach.setReleaseHandMotion(hand, releasingMotion)
```

In addition, a pointing hand pose needs to be configured:

```
reach.setPointHandMotion(hand, pointingMotion)
```

When all the motions have been added, run the following to build up the pseudo-examples:

```
character = scene.getCharacter(characterName)
reach.build(character)
```

To interact with a pawn, that pawn must first have a collision geometry associated with it, which can be created by setting the 'collisionShape' attribute on the pawn:

```
mypawn = scene.getPawn(pawnName)
mypawn.setStringAttribute("collisionShape", type)
```

where 'type' is one of: 'null', 'sphere', 'box', or 'capsule. The relative size of the collision shape can be adjusted by setting the 'collisionShapeScale' attribute:

```
mypawn.setVec3Attribute("collisionShape", size1, size2, size3)
```

where 'size1', 'size2' and 'size3' are up to three dimensions of size: for a sphere, 'size1' is the radius. For a box, all three sizes are used to specify the x/y/z dimensions, and for a capsule, 'size1' is the length and 'size2' is the radius.

'

Configuring Speech
Configuring Motion Blends/Parameterized Animation

Characters can utilize sets of parameterized animations, called blends, which are represented by a single or group of animations. Blends can be connected to each other via transitions, and an a blend-transition graph can be constructed to guide a character through valid blends. All characters start in a 'PseudoIdle' blend which represents the hierarchical set of controllersand can transition to any other blend that has been constructed in the scene. For example, a character can be idling (PseudoIdle blend) then transition to a walk (Walk blend) then transition to a jump (Walking-Jump blend) then transition to a falling down (Falling Down blend), and so forth.

The advantage of using blends, rather than simply playing animations are:

1. Blendscan contain multiple animations parameterized along a number of different dimensions.
2. Animation offsets are automatically applied to the characters.
3. Transitions to and from blends can be customized to ensure smooth animation when moving between blends.

A blend can be used to, for example, parameterize a pointing animation as to allow your character to point at anythin in front of them. Likewise, a blend can be used to parameterize a jumping motion given two jumps of differing heights or distances, and thus generate a set of jumps of any height or distance.

Creating Blends

Note that blends can be constructed with SmartBody's Blend tools.

Blends can be parameterized along one, two or three different parameters, denoted as a one-dimensional, two-dimensional or three-dimensional blend. Each parameter represents a different aspect of the animation (such as 'turning angle', or 'speed', or 'jump height', etc.) In addition, a simple blend has no parameterization, and consists only of one animation (a zero-dimensional blend). To set up a blend:

1. Create the blend
2. Add motions to the blend
3. Specify the parameters associated with the blend
4. Set up correspondences between the animations

To create a blend, specify it's name and dimensionality:

```
blendManager = scene.getBlendManager()
blend1 = blendManager.createBlend1D("myblend")
```

where 'myblend' is the name of the blend. In this case, we are specifying a one-dimensional blend. Then we add motions to the blend. Note that when adding motions to a 1-, 2- or 3- dimensional state, you need to add the parameter associated with that motion:

```
blend1.addMotion("motion1", 10)
```

this indicates that the motion 'motion1' will be added to the blend and activated when the parameter desired is '10'. Likewise:

```
blend1.addMotion("motion2", 20)
```

this indicates that the motion 'motion2' will be added to the blend and activated when the parameter desired is '20'. SmartBody will parameterize the blend such that if parameter value '15' is requested, an animation with equal

contributions from 'motion1' and 'motion2' will be generated and played back on the character.

In addition to specifying the parameters, motions that participate in a blend often will have differing lengths, which necessitates shrinking and stretching one or both motions such that the motions are aligned properly before being interpolated. Motions can be aligned in a blend by using a set of correspondence points. Each correspondence point indicates places along each motion which should be aligned to each other. For example, when constructing a blend of motions involving locomotion, a correpondence point can be assigned to each motion during the heel or toe strike. To set correspondence points:

```
m1 = scene.getMotion("motion1")
m2 = scene.getMotion("motion2")

motions = StringVec()
motions.append("motion1")
motions.append("motion2")

points = DoubleVec()
points.append(0)
points.append(0)
points.append(.5)
points.append(.7)
points.append(m1.getDuration())
points.append(m2.getDuration())

blend1.addCorrespondencePoints(motions, points)
```

This will construct threesets of correspondence points: the first connects time 0 of motion1 to time 0 of motion2, the second connects time .5 of motion1, to time .7 of motion2, the third connects the end of motion1 to the end of motion2.

Once the correspondence points have been added to the 1-dimensional blend, the blendcan be activated via the <blend> BML command, such as:

<blend name="myblend" x="15"/>

Please see the section on Specifying Blend Behaviors for more details.

Constructing 0D Blends

A zero-dimensional (0D) blend will have no parameterization, and is represented by a single animation. Such a blend can be activated using the following BML: <blend name="myblend0D"/>

```
blendManager = scene.getBlendManager()
blend0 = blendManager.createBlend0D("myblend0D")

blend0.addMotion("motion1")
```

Constructing 1D Blends

A one-dimensional (1D) blend can have any number of animations, each with a single parameter. Since multiple animations are specified, correspondence points are necessary in order to properly timewarp the animations. Such a blend is activated by specifying the blend parameter, which will blend between the two animations with parameters closest to the one specified.If the parameter specified is the same as a single animation, then the blend will playback the contents of that animation only without blending. Such a blend can be activated using the following BML: <blend name="myblend1D" x="15"/>

```
blendManager = scene.getBlendManager()
blend1 = blendManager.createBlend0D("myblend1D")

blend1.addMotion("motion1")

blend1.addMotion("motion2")
```

```
motions = StringVec()
motions.append("motion1")
motions.append("motion2")

points = DoubleVec()
points.append(0)
points.append(0)
points.append(.5)
points.append(.7)
points.append(m1.getDuration())
points.append(m2.getDuration())

blend1.addCorrespondencePoints(motions, points)
```

Constructing 2D Blends

A two-dimensional (2D) blend can have any number of animations, each with two parameters. Since multiple animations are specified, correspondence points are necessary in order to properly timewarp the animations. Such a blend is activated by specifying the two blend parameters, which will blend between three animations closest to the two parameters specified in a two-dimensional plane. Thus, such a blend requires a set of triangles to be specified that indicate how the blend will be activated. If the two parameters specified are the same parameter values of a single animation, then the blend will playback the contents of that animation only without blending. Such a blend can be activated using the following BML: <blend name="myblend1D" x=".4" y=".7"/>

blendManager = scene.getBlendManager()
blend2 = blendManager.createBlend0D("myblend2D")

```
blend2.addMotion("motion1", 0, 0)
blend2.addMotion("motion2", 1, 0)
blend2.addMotion("motion3", 0, 1)
blend2.addMotion("motion4", 1, 1)

motions = StringVec()
motions.append("motion1")
motions.append("motion2")
motions.append("motion3")
motions.append("motion4")

points = DoubleVec()
points.append(0)
points.append(0)
points.append(0)
points.append(0)
points.append(m1.getDuration())
points.append(m2.getDuration())
points.append(m3.getDuration())
points.append(m4.getDuration())

blend2.addCorrespondencePoints(motions, points)
```

A triangular parameterization must be specified to indicate how to blend the parameters together, such as:

```
blend2.addTriangle("motion1", "motion2", "motion3")
blend2.addTriangle("motion3", "motion2", "motion4")
```

Which separates the parameter space into two triangles: one containing motions 1,2 and 3, the other containing motions 2,3 and 4.

Constructing 3D Blends

A three-dimensional (2D) blend can have any number of animations, each with three parameters. Since multiple animations are specified, correspondence points are necessary in order to properly timewarp the animations. Such a blend is activated by specifying the threeblend parameters, which will blend between four animations closest to the three parameters specified in a three-dimensional space. Thus, such a blend requires a set of tetrahedrons to be specified that indicate how the blend will be activated. If the three parameters specified are the same parameter values of a single animation, then the blend will playback the contents of that animation only without blending. Such a blend can be activated using the following BML: <blend name="myblend1D" x=".4" y=".7" z=".8"/>

blendManager = scene.getBlendManager()
blend3 = blendManager.createBlend0D("myblend2D")

```
blend3 .addMotion("motion1", 0, 0, 0)
blend3 .addMotion("motion2", 0, 0, 1)
blend3 .addMotion("motion3", 1, 0, 1)
blend2.addMotion("motion4", 0, 1, 0)

motions = StringVec()
motions.append("motion1")
motions.append("motion2")
motions.append("motion3")
motions.append("motion4")

points = DoubleVec()
points.append(0)
points.append(0)
points.append(0)
points.append(0)
points.append(m1.getDuration())
points.append(m2.getDuration())
points.append(m3.getDuration())
points.append(m4.getDuration())

blend2.addCorrespondencePoints(motions, points)
```

A tetrahedral parameterization must be specified to indicate how to blend the parameters together, such as:

```
blend2.addTetrahedron("motion1", "motion2", "motion3", "motion4")
```

Note that a greater number of animations would require the specification of more than one tetrahedron.

Specifying Blend Parameters Automatically

Blend parameters can be set by using the Python API available for the motion assets. For example, if the parameter desired is linear velocity, the getJointSpeed() method can be used to extract it automatically from the motion data. For example:

```
skeleton = scene.getSkeleton("common.sk")
baseJoint = skeleton.getJoint("base")
motion = scene.getMotion("motion1")
motion.getJointSpeed(baseJoint, 0, motion.getDuration())
```

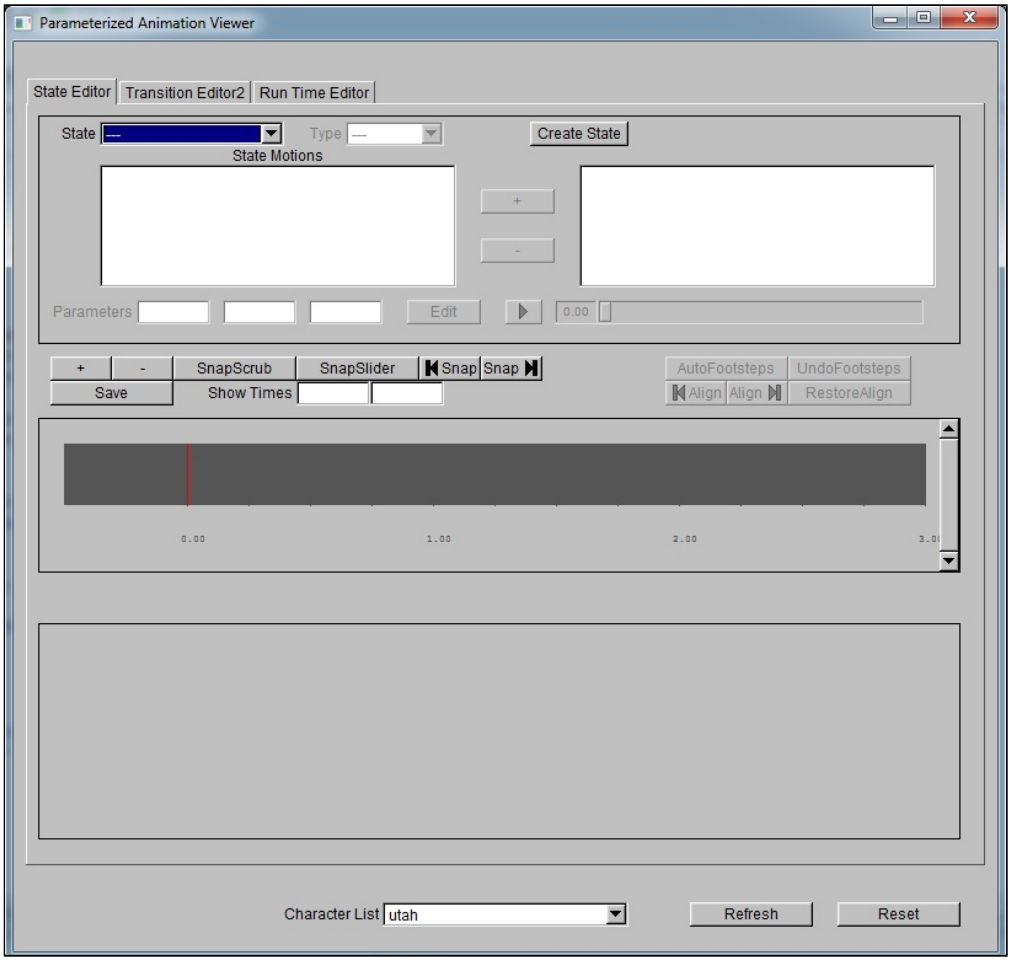
which will determine the average linear velocity of the joint called 'base' for the entire motion.

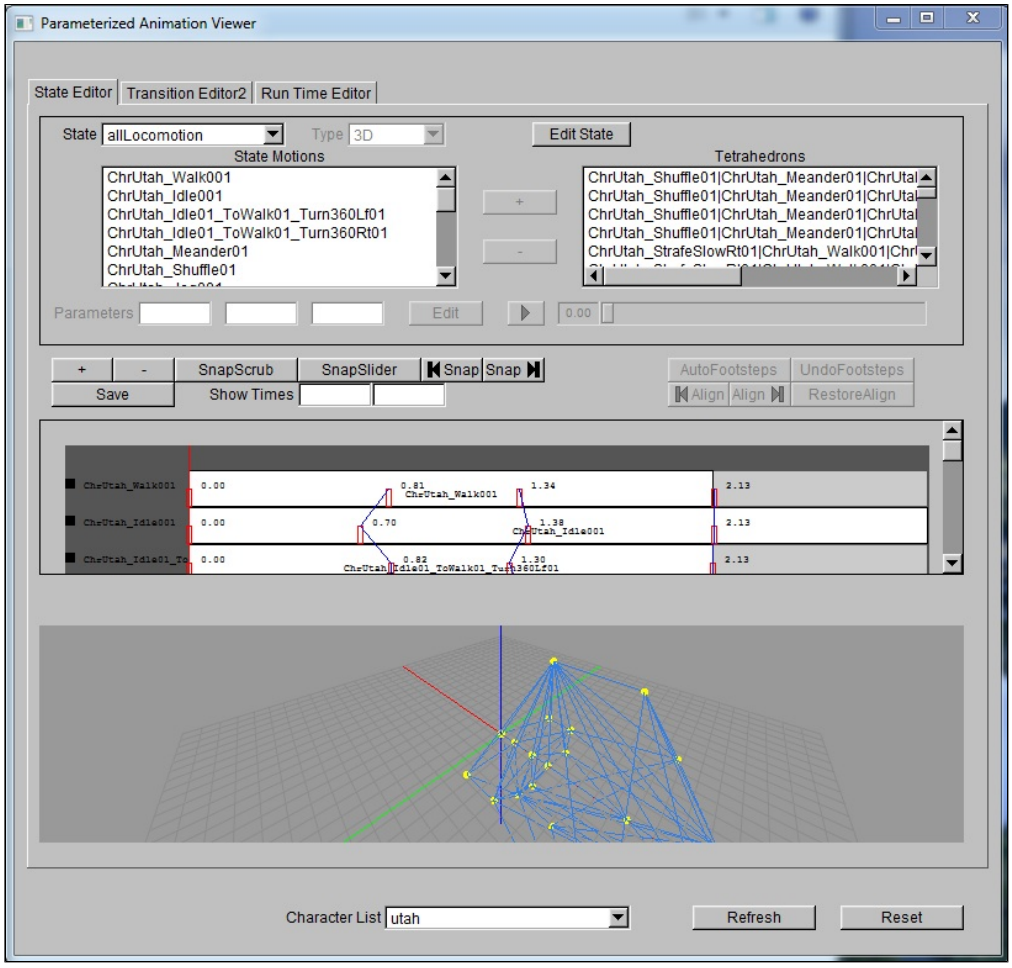
Parameter	Description	
getJointAngularSpeed()	Determines the average angular velocity of a joint	

getJointAngularSpeedAxis()	Determines the average angular velocity of a joint around the X, Y or Z axis	
getJointSpeed()	Determines the average linear velocity of a joint	
getJointSpeedAxis()	Determines the average linear velocity of a joint around the X, Y or Z axis	

Configuring Blends Using the Blend Editor

Blend Editor Overview





Under SmartBody viewer, go to window, select Parameterized Animation Viewer, you will see above window. There are three tab groups, blend editor, transition editor, run time editor. By default, run time editor is selected. Click blend editor, we will start here.

The Blend Editor contains three sections.

First section, general state management.

- **State List**: list of all the current available states. By default, the value is set to empty. Empty state will switch the button right to state list to "Create Blend", other valid state will switch the button to "Edit Blend".
- **Type List**: type of current selected blend, not editable. Options include 0D, 1D, 2D, 3D.
- **Create/Edit Blend Button**: button that leads user to create or edit a state. Details can be found in create state section.
- **Blend Motions Browser**: display all the motions contained in selected blend.
- **Shape Browser**: display all the shapes for selected blend. This only works for 2D and 3D blend. For 2D, shape means triangles, for 3D, it means tetrahedrons.
- **"+" Button**: create a shape by selecting motions from motion browser to the left. This only works for 2D and 3D state. For 2D, only by selecting three motions, this button will be active. For 3D, user has to select four motions to activate this.
- **"-" Button**: delete selected shapes from selected state.
- **Parameters Inputs**: inputs that display x, y, z parameters for selected motion. User can change the value directly. x parameter will be active for 1D, 2D and 3D state. y parameter will be active for 2D and 3D state. z parameter is active for only 3D state.
- **Edit Button**: batch edit motion parameters. Details can be found later.
- **Play Button**: play the selected motion.

- **Frame Slider**: play the selected motion at specific frame.

Second section, correspondence points editor.

- **"+" Button**: adds one correspondence point to every motion inside selected blend. The points will be added according to the position of scrub line.
- **"-" Button**: delete one correspondence point from every motion inside selected blend.
- **SnapScrub Button**: move the current selected correspondence point to the scrub line. It also do a correspondence points check to make sure all the correspondence points are in ascending order.
- **SnapSlider Button**: move the current selected correspondence point to the value of the slider and then do a valid check.
- **SnapToBegin Button**: move the current selected correspondence point to 0.
- **SnapToEnd Button**: move the current selected correspondence point to end of the motion.
- **Save Button**: save the current blendto a destination python file.
- **Show Times Inputs:** Specify the minimal and maximal show time for correspondence points editing widget.
- **AutoFootSteps Button**: button allows detecting foot steps automatically. Details can be found in later section.
- **UndoFootSteps Button:** restore the result before doing auto foot steps detection. It only keep tracks of one action.
- **AlignToBegin Button**: cut the last frame of selected motion and paste it to the begin.
- **AlignToEnd Button**: cut the first frame of selected motion and paste it to the end.
- **RestoreAlign Button:** restore all the alignment action done to selected motion.
- **Correspondence Points Edit Widget**: display each motion as a block and each correspondence point as a mark. By selecting the motion, corresponding track will be highlighted. User can also drag the marker. A scrub line will be draw if user hit Play Button. A alignment line will be draw if user did align action.

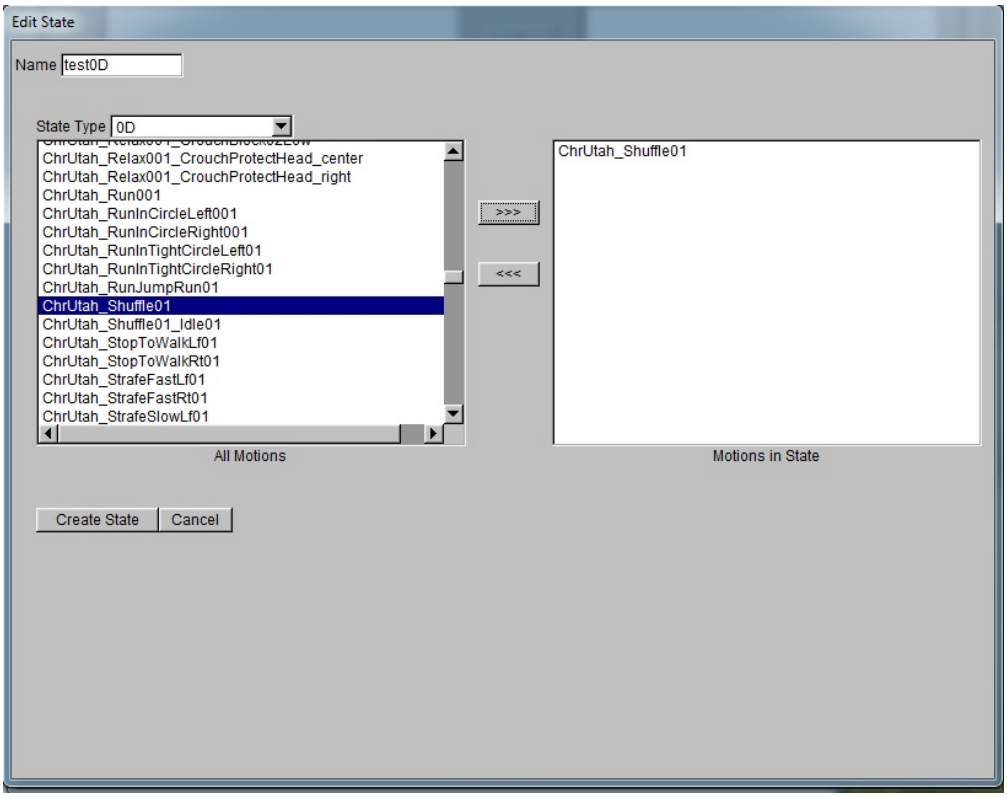
Third section, parameter space visualizer.

- Draw parameters in 2D/3D space. 3D blendwill have a 3D visualizer and 0D, 1D, 2D blendwill have a 2D visualizer. Shapes will also be draw here. Triangles for 2D blend and tetrahedrons for 3D blend.

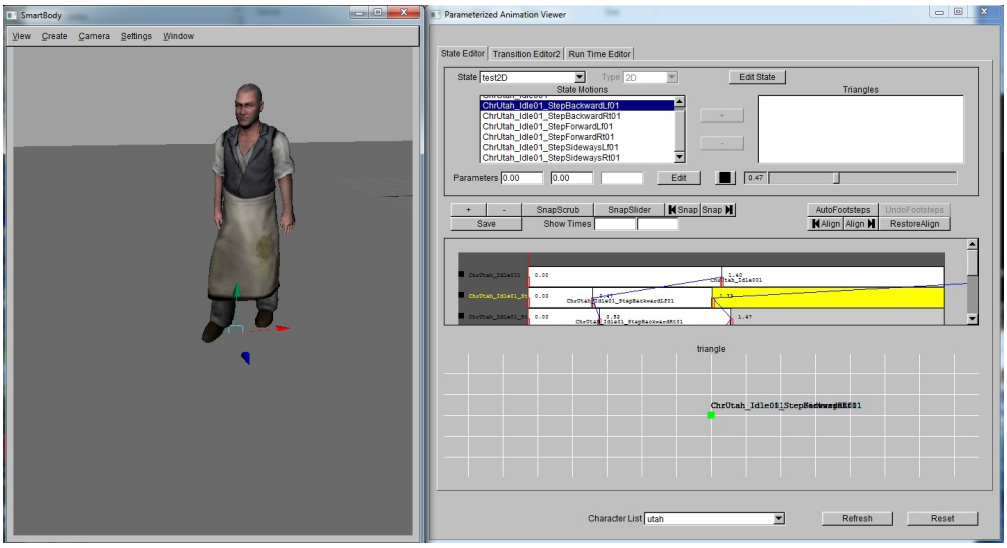
Create Blends

Basic steps to create a blend includes: add motions into the blend, set type, setup correspondence points, edit parameters, crafting shapes, save out.

- **Add motions and set type**: select empty blend, and click create blend button. You will be able to go to Edit Blend window.



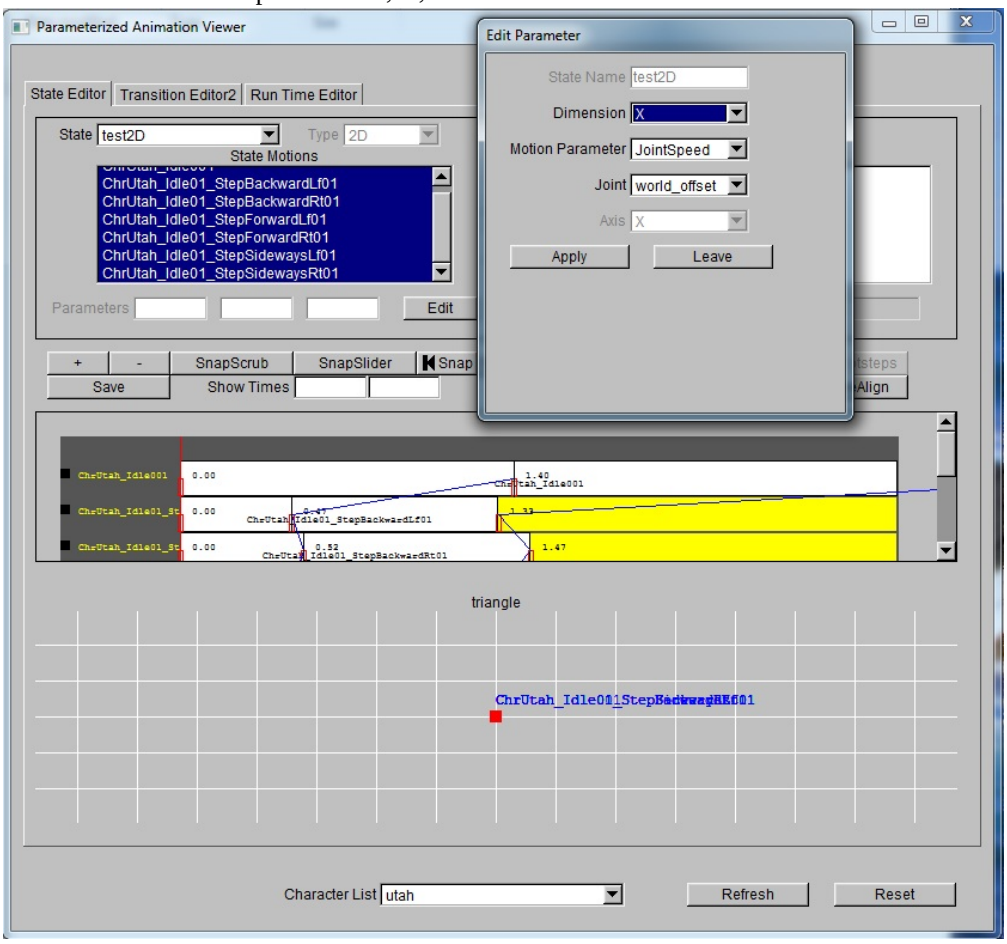
- **Setup correspondence points**: By default, there would be two correspondence points added for each motion at beginning and end. User can press "+" button to add markers. By pressing Play Button, user can see how the motions looks on specific character. By dragging the frame slider, a scrub line will be drawn on motion track. Correspondence points will be added accordingly to the scrub line. Then user can adjust it by dragging it or press SnapScrub/SnapSlider/SnapToBegin/SnapToEnd button.

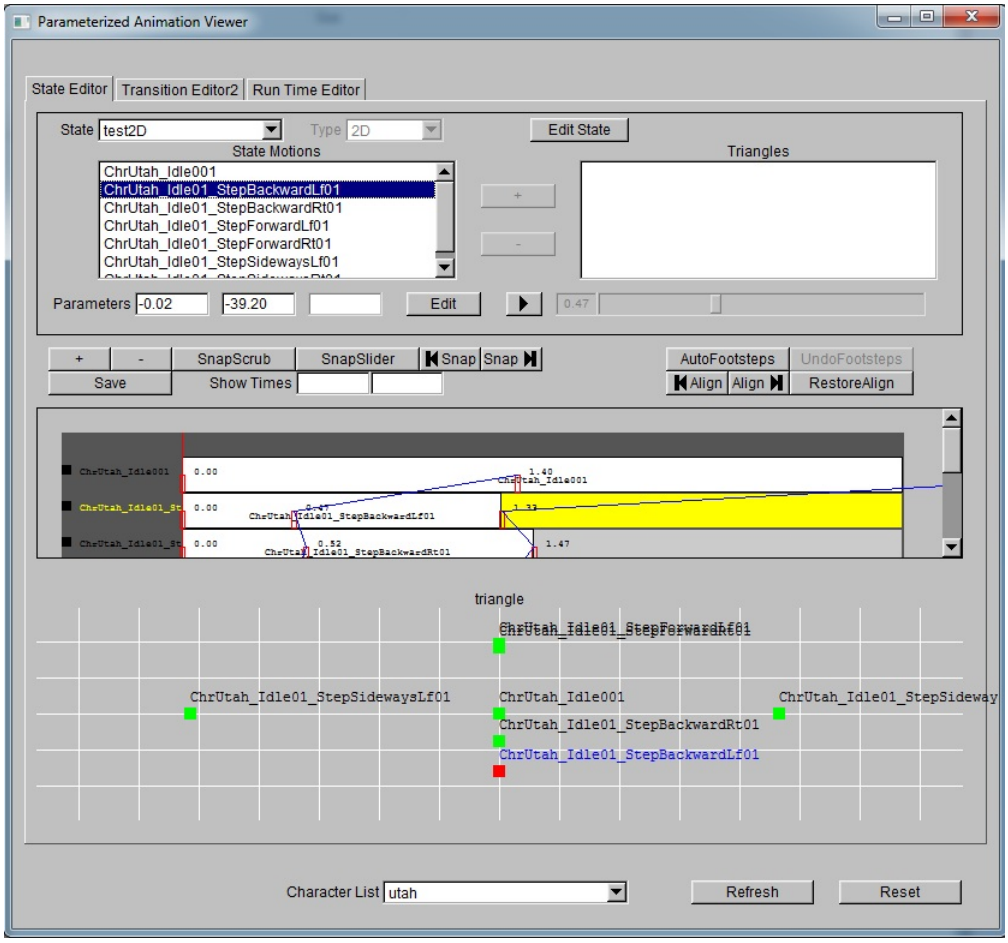


- **Edit Parameters**: this step is not needed for 0D blend. User can directly modify the parameters input field or use Edit button to do batch edit. First select motions need to be batch processed. Then click Edit button. This will create a Edit Parameter window. Once hit apply apply button, new parameter will be shown in parameter visualizer main window.
  -**Dimension**: which parameter to apply. Options are X, Y, Z.
  -**Motion Parameter**: what parameter to extract from selected motions. Options are JointSpeed, JointSpeedAxis, JointAngularSpeed, JointAngularSpeedAxis, JointTransition.
  -**Joint:** which joint to extract data from.

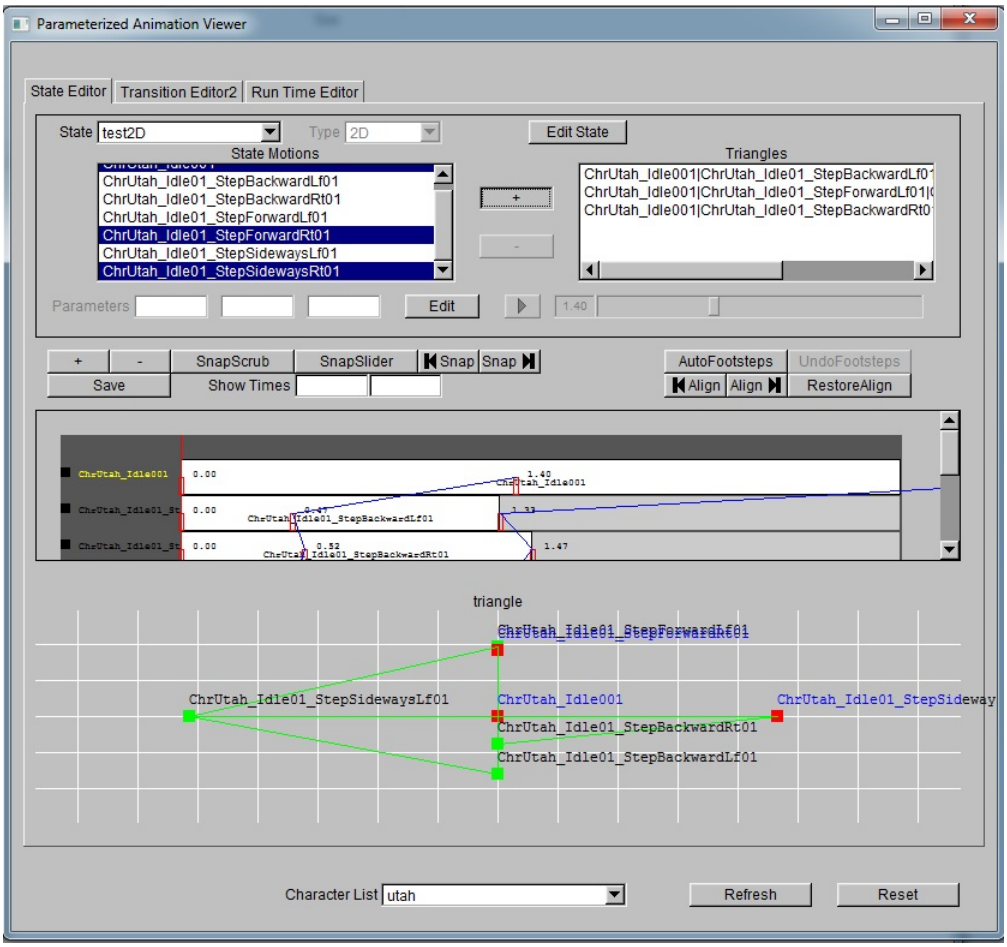
-**Axis:** only applies to JointSpeedAxis, JointAngularSpeedAxis and JointTransition, it defines which axis to

extract the data from. Options are X, Y, Z.





- **Crafting Shapes:** This step is only needed for 2D blend and 3D blend. 2D blendrequires triangles crafted and 3D blendrequires tetrahedrons. To create these shapes, select the 3/4 motions and press "+" Button.



- **Save:** Once all are done, save the file out by hitting save button. Here is examples of output file for each dimension. [test0D.pytest1D.pytest2D.pytest3D.py]

Advanced steps illustrated below will help speed up the setting up process. These steps includes motion preparation, auto footstep detection. They are particularly important when creating complicated blend, e.g. locomotion state.

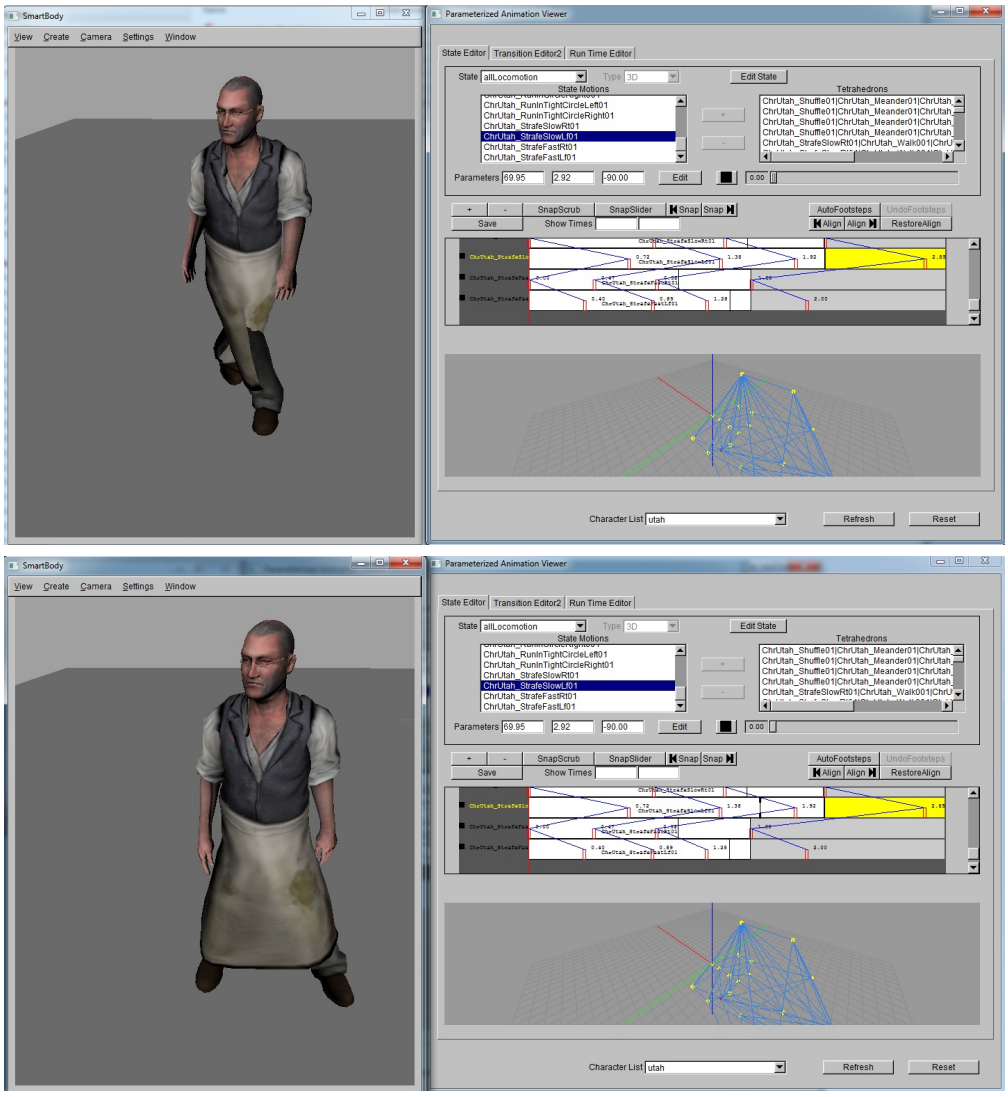
- **Motion preparation**: it includes mirroring, duplicate cycles, aligning.
  -**Mirror**: art asset sometimes only have half of the motions. Use following python command, user can easily mirror the right motion to the left motion.

```
motion = scene.getMotion("ChrUtah_RunInCircleLeft001")
mirrorMotion = motion.mirror("ChrUtah_RunInCircleRight001")
```

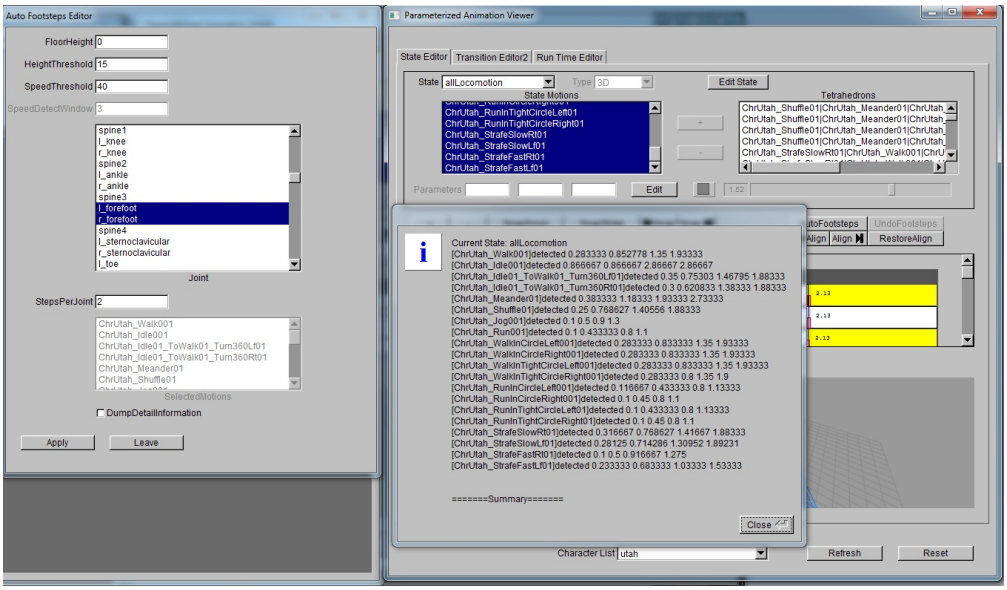
  -**DuplicateCycles:** sometimes you may have walk animation with one cycle and jog animation with two cycles. To make all the animations consistent on cycles without going back to artist. User can use the following command:

```
motion = scene.getMotion("ChrUtah_Run001")
duplicateCycleMotion = motion.duplicateCycle(1, "ChrUtah_Run001_duplicate1")
```

  -**Align:** for some blend, user may want all the animations start with the same posture. For example, inside locomotion state, it would be great if all the animation start with the left foot. To achieve that, user can use the align function. After aligning, an alignment black line will be drawn on the highlighted motion block.



- **Auto Footstep detection:** For locomotion blend, it would be really helpful if there is an automatic method to detect footsteps. To do that, select motions to run the process then click AutoFootsteps button, a new window will show up.
  -**FloorHeight**: defining floor height
  -**HeightThreshold:** under what height would the foot be considered on the ground
  -**SpeedThreshold:** below what speed would the foot be considered still
  -**SpeedDetectWindow:** how large is the window for calculating joint speed. Not active.
  -**Joint:** what joints will be processed
  -**StepsPerJoint**: how many steps user want to get for each joints. This input field is active only when all the motions inside the state are selected. If only subset of the motions are selected, the desired steps for joint would be based on how many correspondence points are there for each motion inside the blend.
  -**SelectedMotions:** what are the motions current selected. Not active.
  -**DumpDetailInformation**: Print the detailed output from footsteps detection algorithm or not.
  After applying the algorithm, a popup window will show up printing out the final result, new corresponding point will show up in the main window.



Configuring Character Physics

Physics-based characters can produce realistic reaction to external forces such as gravity, or collision contact. SmartBody now supports physics simulation to drive the character animations.

A physics character is basically a set of connected body links based on his skeleton topology. During setup, user needs to provide appropriate mass, collision geometries, and joint parameters for the character. At run-time, physical simulation will update the new body link position and orientation at each time step to achieve realistic motions. A basic example is the ragdoll, which simulate the character with only gravity and collisions. In addition to ragdolls, SmartBody also supports more advanced feature like pose tracking. It enables the character to follow an input kinematic motions under physical simulation.

In our current implementation, Open Dynamic Engine (ODE) is used as our simulation engine. In the future, we would like to support multiple physics engines such as PhysX or Bullets.

Character Physics Features

The current supported features include :

- **Ragdoll Animations :** The character will naturally die down based on gravity and other collisions. The future extension includes adding mechanisms for the character to naturally get back from a lie down pose.
- **Proportional Derivative (PD) Motion Tracking :** The character will track an input kinematic motion as much as possible. This enables the character to execute a motion under physical simulation, and at the same time respond directly to external push or collisions. The current implementation does not control the character balance. Instead, the root joint is driven directly by the kinematic motion to prevent character from falling down.
- **Kinematic Constraint :** A physics-based character is simulated by a set of connected body links. Instead of using physical simulation, each body link can also be constrained to follow a kinematic motion. This allows the user to "pin" a body link in a fixed position, or have a specific link to follow the original kinematic motion.
- **Collision Events :** A collision event will be sent out from the simulation system when a physics character is collided with other objects. User can provide a python script to handle this collision. For example, the character can gaze at the object that is hitting him, or execute a reaction motion. This helps enhance realism and interactivity of a character with the virtual environment.

How to Setup a Physics Character

Physics-based character can be regarded as a set of connected body links. Each body link is simulated as a rigid body and all body links are connected based on character joints. In order to simulate the character correctly, user needs to provide correct mass and geometry information for each body links, as well as some joint properties such as joint limits.

The overall procedural of setting up a physics character :

1. **Initialize a kinematic character with appropriate skeleton.**

2. **Create physics character using Physics Manager.**
    a. phyManager = scene.getPhysicsManager();

    b. phyManager.createPhysicsCharacter("charName");

3. **Although step 2. will procedurally setup collision geometries and masses based on default parameters. Although this will generate the default parameters for you, they are usually not ideal for a specific character or scenario.**

4. **Setup collision geometries manually.**
    a. phyBodyLink = phyManager.getJointObj("charName","jointName"); # get corresponding body link for joint "jointName" from character "charName".
    b. phyBodyLink.setStringAttribute("geomType",geomShape); # This command set the collision geometry to a different shape. Here 'geomShape' is a string value. It can be either "box", "capsule" , or "sphere".
    c. phyBodyLink.setVec3Attribute("geomSize",size); # This command set the size for collision geometry.

5. **Setup mass.**
    a. phyBodyLink = phyManager.getJointObj("charName","jointName"); # get corresponding body link for joint "jointName" from character "charName".
    b. phyBodyLink.setDoubleAttribute("mass",massValue); # set the mass for this body link.

6. **Setup joint limit**
    a. phyJoint = phyManager.getPhysicsJoint("charName","jointName"); # get corresponding physics joint with "jointName" from character "charName".
    b. phyJoint.setVec3Attribute("axis0", dirVec); # set rotation axis0 according to dirVec. We can set axis1, axis2 similarly.
    c. phyJoint.setDoubleAttribute("axis0LimitHigh", highLimit); # set the maximum rotation angle for axis0 in positive direction. highLimit must be larger than zero. "axis1" and "axis2" can also be done similarly.
    d. phyJoint.setDoubleAttribute("axis0LimitLow", lowLimit); # set the rotation angle limit for axis0 in negative direction. lowLimit must be negative.

The above process setup a physics character with appropriate joint limits and body link geometry and mass. A proper joint limit setting should allows the character to perform all required kinematic motions within its joint limit while preventing unnatural joint angles. A proper geometry and mass should be set to approximate the actual body link shape and mass distribution. This allows more accurate collision detection and realistic response when interacting with the environment.

These settings can also be done using GUI in Resource Viewer–>Service–>Physics. (To-Do : add GUI picture and step by step guide).

Setting Physics Parameters

The results of physics simulation are affected by many parameters. If these parameters are not set correctly, we may not obtain the desired character motions. Some default values have been set to work in a typical settings and environment. However, user may want to fine tune these parameters when using a different skeleton, different scale unit to obtain best results.

1. **gravity :** A typical gravity should be 9.8 m/s^2. In practice, it should be set according to the unit currently used. For example, if your character is created in centimeter (cm), then you should set gravity to 980 instead of 9.8.
   To set the gravity with Python script :
   a. phyEngine = phyManager.getPhysicsEngine(); # get the physics engine currently used
   b. phyEngine.setDoubleAttribute("gravity", valueOfGravity); # set the gravity value.
2. **dt :** The time step taken for each physics update. Physics engine usually requires small time steps for the simulation to be stable. For simple features like rigid body dynamics or ragdoll simulation, the time step can be as large as the screen refresh rate (60 Hz). For more advanced features like pose tracking, the time step needs to be much smaller (~ 1000Hz) to maintain a stable simulation.
   To set the time step :
   a. phyEngine = phyManager.getPhysicsEngine(); # get the physics engine currently used
   b. phyEngine.setDoubleAttribute("dT", value); # set the dt value. It must be positive.
3. **Ks :** This parameter determines how strong is the pose tracking. The higher this parameter is, the more closely the physical simulation will match the kinematic motion. The higher value also gives the character a more "rigid" feeling when there are external forces or collisions. However, setting this parameter too high would cause the physical simulation to become unstable. This will in turn require dt to be decreased to avoid such instability. In general, it should be set to counter the gravity so the character can just stay upright and follow the kinematic motion.
   To set Ks :
   a. phyEngine.setDoubleAttribute("Ks", value); # set the Ks value. It must be positive.

4. **Kd :** The damping parameter. This parameter is set to counter some unnatural oscillation body movements when Ks is high. If Kd is high, the resulting motion would be slower and less responsive. Similar to Ks, setting this parameter too high may also cause instability.
   To set Kd :
   a. phyEngine.setDoubleAttribute("Kd", value); # set the Kd value. It must be positive.

5. **MaxSimTime :** This parameter sets the maximum allowing time used for physics engine update. By default, the physics simulation will keep updating to match the current system time. This keeps the physics engine to sync up with the kinematic animation. However, if the scene is complicated or if the dt is very small, the physics engine may take a significant chunk of time for updating. This makes it more and more difficult for physics engine to sync up with the system and the performance will be slowed down significantly. By setting this parameter to a proper value (by default it is 0.01 second), it prevents the physics simulation from taking too much time and drag down the system performance. Note that when both dt and MaxSimTime are very small, the physics engine may not be able to iterate enough to follow the motion. This could cause the pose tracking to be inaccurate and slower physics response.
   To set MaxSimTime :
   a. phyEngine.setDoubleAttribute("MaxSimTime", value); # set the MaxSimTime value. It must be positive.

6. **KScale :** This parameter scale the Ks, Kd for a specific joint. This allows different Ks, Kd setting for each joint. Since each body link may have different mass and different number of descendent body links, it would not be feasible to use a single Ks, Kd for pose tracking. For example, a shoulder may need a larger Ks to generate more torque so the arm can be lifted to match the desired pose, while the wrist will need smaller values. The general guideline for setting this parameter should be according to the effective mass for the joint – the sum of mass from all descendent body links.
   To set KScale :

a. phyJoint = phyEngine.getPhysicsJoint("charName","jointName"); # get corresponding physics joint
b. phyJoint.setDoubleAttribute("KScale", scaleValue); # set the KScale. It must be positive.

Setting Up Constraint

Physics character by default is driven purely by simulation. Although pose tracking can be used to bias the simulation toward desired kinematic motion, it is difficult to keep the character stand upright or fixed some body link exactly. The constraint is introduced to provides this functionality. For example, instead of developing a complicated balance controller to have the character maintain upright pose, we can constrain the character's root to prevent him from falling down. Also, we can constrain a character's hands to a moving objects and have that object drive the character's global movements. This can create effects like a character grabbing a bar or ladder from a helicopter, etc. Note that although some body links are constrained, the other parts are still in effects for physical simulation and pose tracking can still work for the rest of joints.

To setup constraints :

1. Select the body link to be constrained :
    a. bodyLink = phyManager.getJointObj("charName","jointName"); # get corresponding body link from a character
    b. bodyLink.setBoolAttribute("constraint", true); # enable constraint

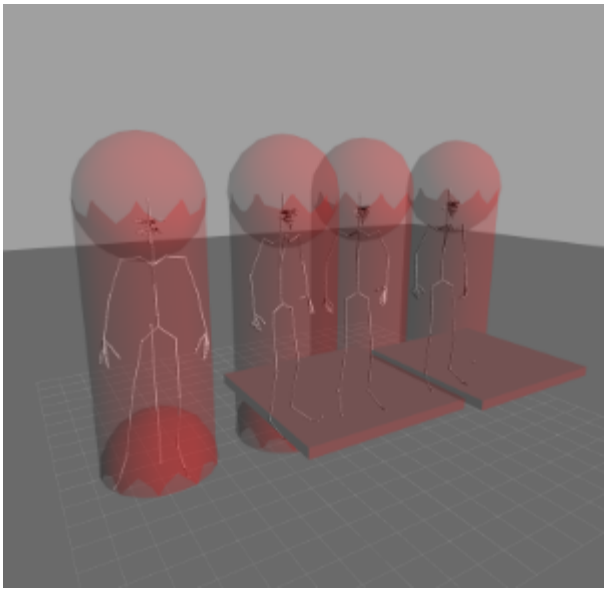
2. If we do not specified a constrain target (the target object which the body link will follow), then the body link will follow its kinematic motion. This is used when user want some body links (for example, the foot) to exactly match the desired kinematic trajectory.
3. If we want to fix the body link to a target object, we need to specify the name for that object.
    a. bodyLink = phyManager.getJointObj("charName","jointName"); # get corresponding body link from a character
    b. bodyLink.setBoolAttribute("constraintTarget",targetPawnName); # set the constraint target to a pawn

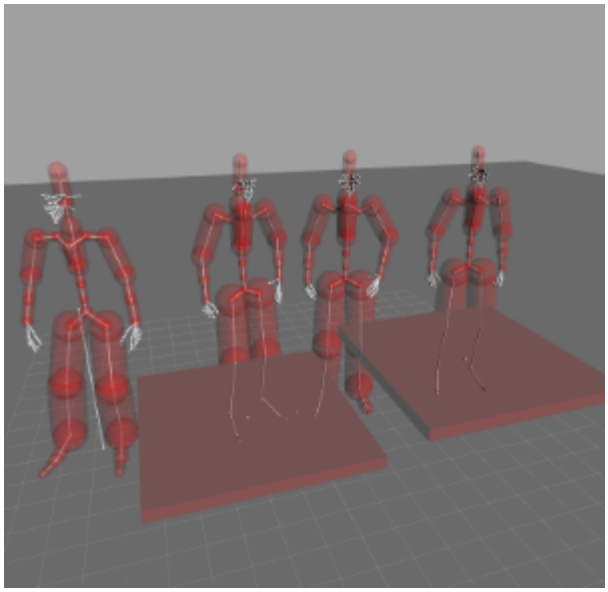
Note that although the physical simulation will try to accommodate user specified constraints as much as possible, it can not handle constraints that conflict the character setup. For example, if user sets up constraints that control each hand, the distance between two constraints can not exceed the total length of both arms. Otherwise the character will not be able to perform such a task and simulation would become unstable to satisfy the conflicting goals.

Configuring Collision Detection

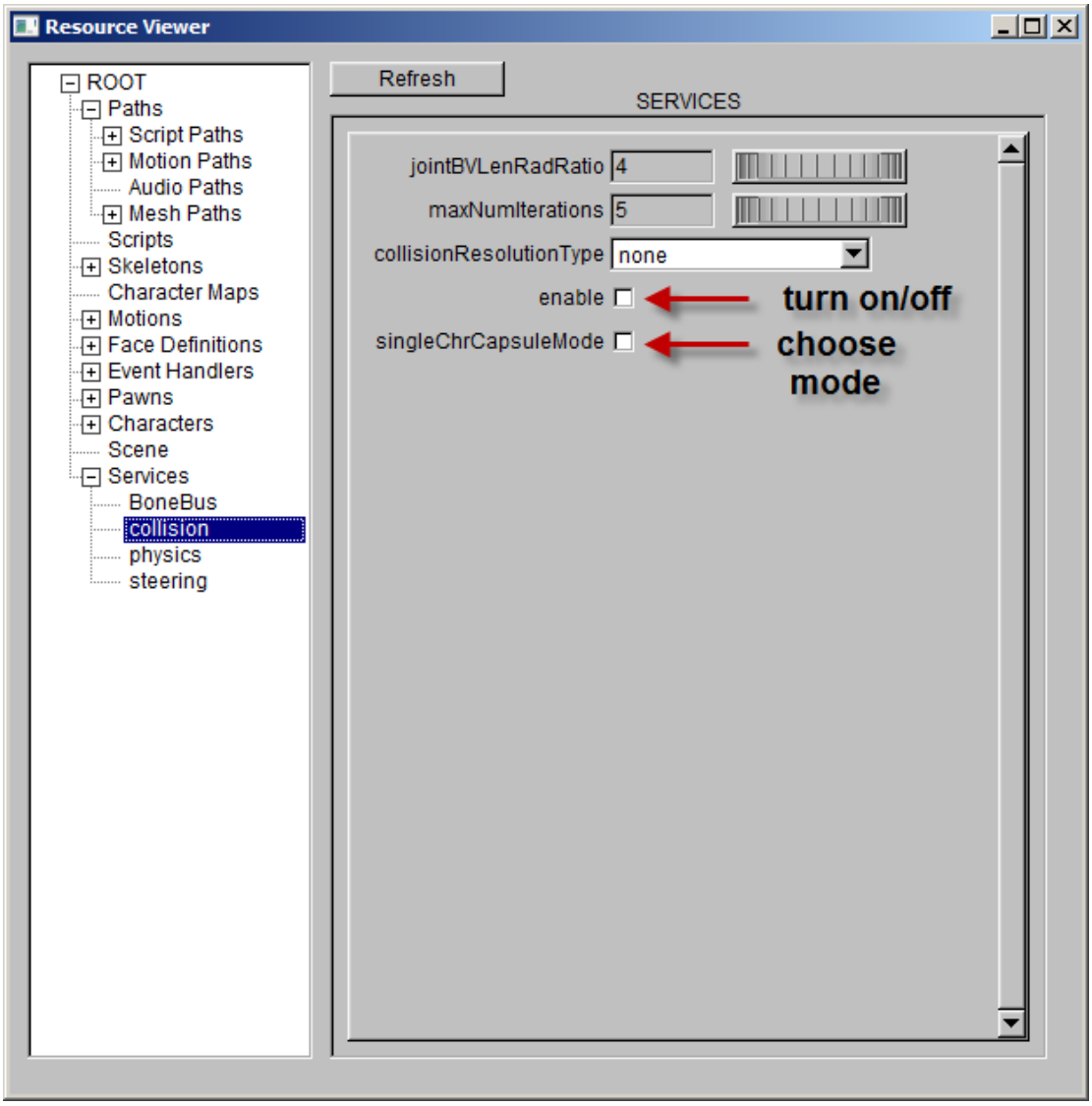
SmartBody has two collision detection modes (see pictures below):

single capsule mode (left), and multiple capsule mode (right).

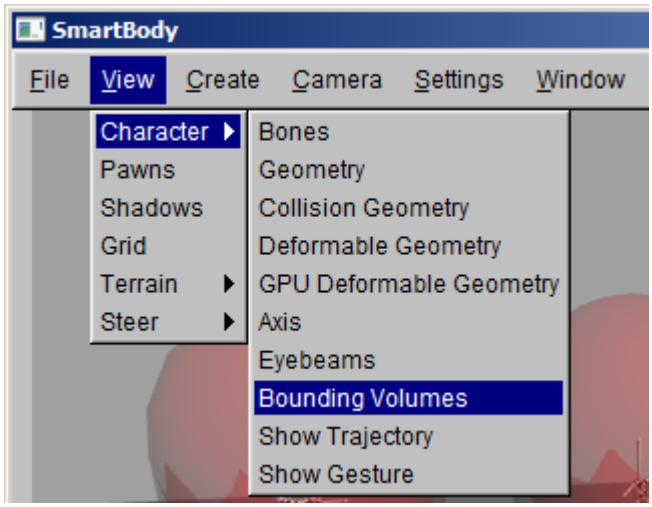




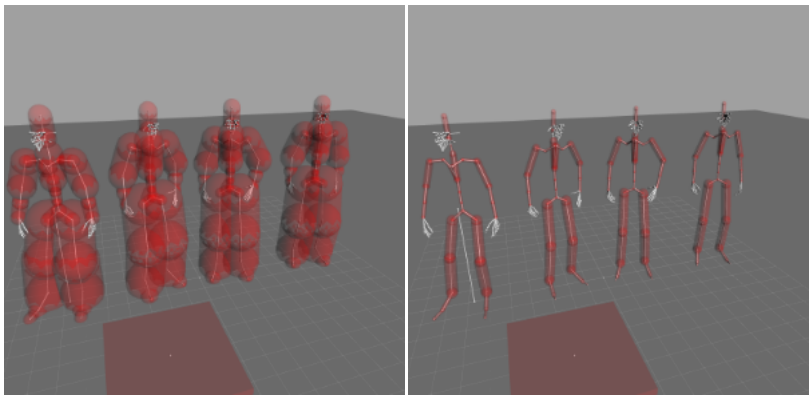
To turn on collision detection, choose the mode inside Resource Viewer -> Service -> Collision, then check "enable"



To see the collision geometries, enable viewing of the bounding volume here:



The parameter "jointBVLenRadRatio" (length-radius ratio) gives the user control over the automatically generated capsule for each bone. Its default value is 4.0 as shown above. The result looks like these shown below when set to 2.0 (left) and 8.0 (right).



When enabling the collision detection service using multiple capsule mode, it automatically creates capsules for each joint on all the characters inside the scene (self-collision within each character is disabled), and it will also add all the pawns that have cubic collision shape into the collision space. The service will send out events when collisions are detected, in the form of the string **"Collision detected between A and B".** A and B are object names followed by "_BV" (meaning Bounding Volume), and then followed by ":JOINTNAME". Here is a list of such events as examples:

Collision detected between table_BV and table2_BV ('table' and 'table2" are colliding)

Collision detected between table2_BV and elder_BV:l_hip ('table2' and the l_hip joint on character 'elder' are colliding)

Collision detected between doctor_BV:l_shoulder and brad_BV:r_shoulder (the l_shoulder of 'doctor' is colliding with r_shoulder on 'brad')

If using single capsule mode, only one capsule is created for each character and no pawn is added in collision space, i.e. this mode only detects collisions between characters. The service will also send out events, in the form of **"Collision detected between character A and character B"**, where A and B are character's names. For example:

Collision detected between character doctor and character brad

Controlling Characters with BML

SmartBody characters can be controlled by using the Behavioral Markup Language (BML).BML contains instructions for characters to walk, talk, gesture, nod, grab objects, look at objects, and so forth.

About BML

The purpose of BML is to provide a common language for controlling virtual humans and embodied conversational agents, such that behavior designers do not have to focus on the behavioral realization (i.e. what does smiling look like?) but rather can focus on the behaviors generation and coordination with other behaviors.SmartBody supports the Vienna Draft version of BML with enhancements. SmartBody does not yet support the 1.0 specification as detailed at http://www.mindmakers.org/projects/bml-1-0/wiki/Wiki.

How to Specify a BML Command

Using Python, use the implicit *bml* object in the Python dictionary and call the *bmlExec()* function:

```
bml.execBML('utah', '<head type="NOD"/>')
```

where utah is the name of the character, and the second parameter to the bmlExec function is the BML, described below. Note that using Python, commands can be specified using the single quote, instead of the double quote character, which is advisable, since most BML commands will contain many double quote character for use with attributes. Alternatively, you could specify BML using Python like this:

```
bml.execBML("utah", "<head type=\"NOD\"/>")
```

Notice that double quotes are used for the function parameters, while double quotes contained within the BML are escaped using the slash character.

Quick BML Reference for SmartBody

Each BML command specifies a behavior that a character will perform.

Behavior	Example	BML Command
Gaze	look at the object called 'table'	`<gaze target="table"/>`
Locomotion	move to location (10, 75)	`<locomotion target="10 75"/>`
Head Movement	nod your head	`<head type="NOD"/>`
Idle	assume an idle posture called 'idling_motion1'	`<body posture="idling_motion1'/>`
Animation	play an animation called 'dosomething'	`<animation name="dosomething"/>`
Gesture	point at character1	`<gesture type="POINT" target="character1"/>`
Reach	Grab the object 'cup'	`<sbm:reach target="cup"/>`
Constraint	Constraint your hand to 'ball'	`<sbm:constraint target="ball"/>`

Face	Raise your eyebrows	`<face type="FACS" au="1" side="both" amount="1"/>`
Speech	Say 'hello, how are you?'	`<speech type="text/plain">hello how are you?</speech>`
Eye saccade	Move your eyes around automatically	`<saccade mode="LISTEN"/>`
Event	Send out an event 3 seconds in the future	`<sbm:event stroke="3" message="sbm echo hello"/>`

In general, BML commands that are not part of the original BML Vienna Specification and are specific to SmartBody use the prefix *sbm:*.

Timing BML Commands

Each BML command specifies a behavior, which, by default, start immediately, and end at various times depending on the specific behavior. For example, a nod lasts one second by default, a gesture lasts as long as the animation used to specify it, and so forth. A behavior can be scheduled to play or finish playing at different times using synchronization points. Each behavior generated by a BML command uses a set of synchronization points. Most use the minimal set of points - *start* and *end*. Some behaviors are deemed persistent, and have no finish time, such as gazing or idling, and thus have no *end* synchronization point. A behavior can be designed to to start or stop at a specific time in the future. For example:

```
<head type="NOD" start="2"/>
```

Indicates that you would like your character to start nodding his head two seconds from the time the command is given. Some BML commands, such as <animation> and <gesture> also contain implicit synchronization points that indicate the phases of the action. For example, the *stroke* synchronization point indicates the emphasis phase of a gesture, so:

```
<gesture type="BEAT" stroke="5"/>
```

indicates to make a beat gesture where the *stroke* (emphasis) point is five seconds after the BML command was given. The gesture will be automatically started such that the *stroke* phase of the gesture will occur at the five second mark. For example, if the gesture ordinarily takes 2 seconds to complete, with the stroke phase at the 1 second mark, then the above command will start the gesture four seconds after the command was given, yielding the stroke phase at the 5 seconds, and completion at the 6 seconds.

BML commands can also use relative timings by using the + or - modifiers, such as:

```
<head type="NOD" start="2" end="start+5"/>
```

which indicates to finish the head nod five seconds after it started, in this case, finishing at seven seconds.

The following table shows the synchronization points used for each behavior.

Behavior	Synchronization Points	Comments
Gaze	start	
Locomotion	start ready	start = start time of idle motion ready = time when motion is fully blended with last idle motion

Head Movement	start ready stroke relax end	ready = ramp-in time stroke = middle of head movement relax = ramp-out time
Idle	start ready	
Animation	start ready stroke relax end	ready = ramp-in time stroke = emphasis point of the animation relax = ramp-out time
Gesture	start ready stroke relax end	ready = ramp-in time stroke = emphasis point of the animation relax = ramp-out time
Reach	start	
Constraint	start ready	ready = time needed to achieve constraint
Face	start ready stroke relax end	ready = ramp-in time stroke = emphasis point of the face motion relax = ramp-out time

Speech	start ready stroke relax end	start, ready, stroke = time of first word spoken relax, end = time of last word spoken
Eye saccade	..?	

Compounding & Synchronizing BML Commands

BML commands can be compounded together in blocks. For example, to have your character both raise his eyebrows while nodding, a BML block could look like this:

```
<face type="FACS" au="1" side="both" amount="1"/><head type="NOD"/>
```

There is no limit to the number of BML commands that can be compounded together. Either BML command could be explicitly started or timed by adding the appropriate synchronization points, such as:

```
<face type="FACS" au="1" side="both" amount="1" start="2"/><head type="NOD" start="4"/>
```

which instructs the character to move his eyebrows at two seconds, and nod his head at four seconds. The synchronization points can also be used by adding an *id* to a BML behavior, then using that id to synchronize other behaviors. For example:

```
<face id="foo" type="FACS" au="1" side="both" amount="1" start="2"/><head type="NOD" start="foo:start+2"/>
```

Thus the eyebrow raise has an *id* of *foo*, and the head nod will occur two seconds after the start of the *foo* behavior.The id is unique to each behavior block, and thus the same name can be reused on a different behavior block.

Synchronizing Multiple Character's Behaviors with BML

BML blocks that contain behaviors can only be specified per character. You are allowed to send multiple commands to different characters, such as the following:

```
bml.execBML('utah', '<head type="NOD"/>')
bml.execBML('harmony', '<head type="NOD"/>')
```

but since each block contains behaviors for only a single character, each character's BML cannot be explicitly synchronized with each other. However, you can use the <sbm:event> BML tag to trigger an event that will synchronize one character to the other, as in:

```
harmonyBML = "<head type=\"NOD\"/>"
harmonyName = "harmony"
bml.execBML('utah', '<head id="a" type="NOD"/><sbm:event stroke="a:start+1" message="sbm python
bml.execBML(harmonyName, harmonyBML)"/>')
```

which will trigger a BML nod behavior once Utah's nod has been in effect for one second. The syntax of the <sbm:event> tag is as follows:

- the *sbm* keyword tells SmartBody to respond to this message.
- the *python* keyword tells SmartBody that the rest of the command will be using Python

Note that all <sbm:event> BML tags are sent over the VH message bus (the ActiveMQ server), thus the need to use the *sbm* keyword. In general, any command can be placed in the message="" attribute, as long as the contents comply with XML syntax.

BML Behaviors

Each BML behavior has a number of parameters that can be used to alter its performance. Also note that many behaviors cannot be used unless they have the proper animations, skeleton topology, and so forth.Behaviors are typically implemented by means of a Controller, which will have different requirements. For example, a head nod controller requires a skeleton that has three neck and spine joints, whereas an animation controller requires a motion asset, but is indifferent about the skeleton topology, as long as the motion data matches the skeleton topology of the character. Certain behaviors have multiple modes; a low-quality mode when the configuration isn't available or set up, and a high-quality mode when the proper data or configuration is made. For example, the locomotion behavior will move any character around in the virtual environment, regardless of topology, but will do so without moving the character's body. Once the proper locomotion files are provided, the character will accurately step and turn in a realistic manner.

Each behavior listed on the following pages will contain:

- a description of the behavior
- a list of parameters that can modify the performance of a behavior
- a description of the setup requirements to use this behavior

Idling or Posture

Description

Characters can perform an idle motion, usually a repeatable animation that engages the entire body of the character that represents the subtle movements of the character while it is not performing any other behaviors.An idle behavior will repeatedly play a looped motion. Other behaviors will be layered on top of the idle motion, or replace it entirely. Subsequent calls to the idle behavior will override the old idle behavior and replace it with the new idle behavior.

Requirements

To run the <body> behavior, SmartBody needs motion files whose joint names match those on the character's skeleton.The character and the motion can be any topology and any number of joints. Data from a motion file that matches the joint names of a skeleton will be used, and any joint names that do not match will be ignored.

Motion files need are required to have metadata that indicates their blend-in, blend-out times.

Usage

```
<body posture="idlemotion1"/>
```

where idlemotion1 is the name of the motion to be played. Note that the idle motion will be played at a location and orientation in the world based on the character's offset.

Parameters

Parameter	Description	Example
start	starts the idle posture at a time in the future	`<body posture="idlemotion1" start="3"/>`
ready	the time when the posture is fully blended. The total ramp-in time is (ready-start)	`<body posture="idlemotion1" start="3" ready="5"/>`

Animation

Description

Characters can playback motions generated from animation files in various formats such as .bvh, .amc, .fbx and .skm. The motion controller operates after the posture/idling controller in the controller hierarchy. Thus, any joints specified in the motion file will override the data established by the posture controller. A motion that uses all of the joints of the character will override completely the underlying posture, while a motion that only uses a few joints will only override the motion on those joints, leaving the rest of the joints to use the motion specified on the underlying posture.

Requirements

To run the <animation> behavior, SmartBody needs motion files whose joint names match those on the character's skeleton.The character and the motion can be any topology and any number of joints. Data from a motion file that matches the joint names of a skeleton will be used, and any joint names that do not match will be ignored.

Motion files need are required to have metadata that indicates their blend-in, blend-out times.

Usage

```
<animation name="motion1"/>
```

where motion1is the name of the motion to be played. Note that the animation will be played at a location and orientation in the world based on the character's offset.

Parameters

Parameter	Description	Example
start	starts the motion at a time in the future	`<animation name="motion1" start="3"/>`
ready	the time when the motion is fully blended. The total ramp-in time is (ready-start)	`<animation name="motion1" start="3" ready="5"/>`
stroke	for a gesture, the time of the emphasis point of a gesture	`<animation name="motion1" stroke="4"/>`
relax	time when the animation starts to fade out	`<animation name="motion1" relax="5"/>`
end	indicates when the animation ends	`<animation name="motion1" end="8"/>`
time	multiplier for the speed of the animation (2x plays the animation twice as fast). Note that specifying a time multiplier will automatically adjust the synchronization points.	`<animation name="motion1" time="2"/>`

Timewarping Motions

Animation behaviors can be timewarped (stretched or compressed) by specifying more than one synchronization point. SmartBody handles timewarping of animations as follows:

1. If only one synchronization point is specified, align the behavior such that the behavior will occur at normal speed in line with the synchronization point. Example, let's assume that motion1 lasts for 3 seconds, with it's stroke point at second 2:
   ```
   <animation name="motion1" stroke="5"/>
   ```
   will play *motion1* such that the middle (or stroke) of the motion occurs at second 5, with the rest of the motion aligned to that time. In other words, the beginning of the motion will play at second 4, and finish at second 6.
2. If two synchronization points are specified, then timewarp the rest of the motion according to the relative scale of two synchronization points. For example:
   ```
   <animation name="motion1" start="1" stroke="5"/>
   ```
   will play *motion1* by timewarping it (in this case, slowing it down) by 2x, since the original motion took two seconds to go from the start point to the stroke point, and the user is specifying that that phase should now take 4 seconds, yielding a slowdown of 2x. Thus, the remainder of the motion which has not been explicitly specified by the user, will also play at 1/2 speed. Thus the entire motion will now take 6 seconds to play, and finish at second 7.
3. If three or more synchronization points are specifed, then the behavior will be unevenly timewarped such that each phase of the behavior will be stretched or compressed according to the closest explicitly specified behavior segment. For example, let's assume that the synchronization points for motion1 are: start = 0, ready = 1, stroke = 2, relax = 3, end = 4. Then by specifying:<animation name="motion1" start="1" ready="1.5" stroke="4"/> Then the start-ready phase will be double in speed (since the user explicitly requested it), the ready-stroke phase will now be slowed down by 3x times (originally took 1 second, but the user requested that it now take 3 seconds), and the stroke-relax and relax-end phases will also be slowed down by 3x, since their closest explicit request was a 3x slowdown.

Blend

Description

SmartBody blends consist of one or more animations that can be parameterized in multiple ways. For example, a blend could consist of a set of walking, jogging and running motions, or several animation of the character pointing in different directions. Such blendscan be controlled programmatically (such as through the locomotion controllers) or directly via BML. This can be used, for example, for controlling a character's locomotion interactively.

Requirements

The single animation, one-, two- or three- dimensional blendmust be created first. Please see the section on Creating Blends for more details.

Usage

```
<blend name="allJump"/>
```

```
<blend name="allLocomotion" x="100" y="2" z="-20"/>
```

Parameters

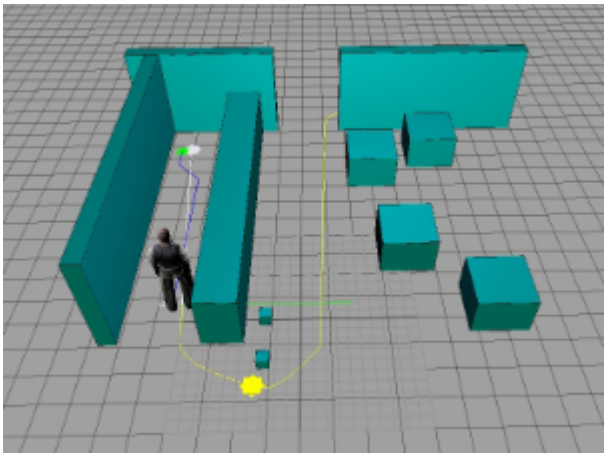
Parameter	Description	Example
name	Name of the blend to use. (Required)	`<blend name="allJump"/>`
mode	Whether a new blend is being scheduled, or the current one is being updated. Options: ■ update ■ schedule Default: schedule	`<blend mode="schedule" name="allJump"/>`
sbm:wrap-mode	Wrap mode of the blend . Options: ■ Loop ■ Once Default: Loop	`<blend name="allJump" sbm:wrap-mode="Loop"/>`
sbm:schedule-mode	Schedule mode of the blend . Options: ■ Now ■ Queued Default: Queued	`<blend name="allJump" sbm:schedule-mode="Now"/>`
sbm:blend-mode	Blend mode of the blend . Options: ■ Overwrite ■ Additive Default: Overwrite	`<blend name="allHeadTilt" sbm:blend-mode="Additive"/>`
sbm:partial-joint	Starting joint inside skeleton hierarchy that defines the affected blending joints	`<blend name="allHeadTilt" sbm:partial-joint="spine4"/>`

x	First parameter value of the blend . The only parameter for a 1D blend , or the first parameter for a 2D or a 3D blend	`<blend mode="schedule" name="allStartingLeft" x="90"/>`
y	Second parameter value of the blend . Second parameter for a 2D or a 3D blend	`<blend mode="schedule" name="allStep" x="22" y="45"/>`
z	Third parameter value of a 3D blend .	`<blend mode="schedule" name="allLocomotion" x="100" y="2" z="-20"/>`
start	Time offset to schedule the blend .	`<blend name="allJump" start="3.0"/>`

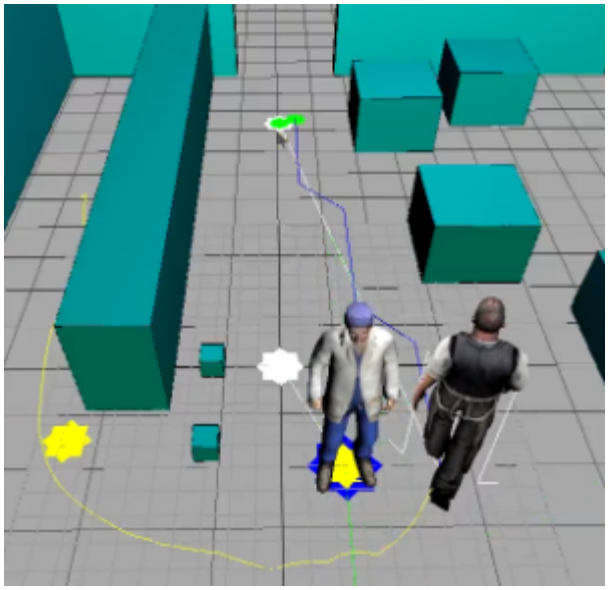
Locomotion

Description

Characters can move to and from areas in the world. Simple locomotion will move the entire character without animating it at various speeds and turning anlges. Full locomotion wil move the character along with natural-looking footsteps and upper body motion. Full locomotion will consist of a set of example motions which are then parameterized according to speed, turning angle or other similar criteria.



The above image show the simple locomotion, in which the character moves but does not animate the rest of his body in order to coordinate with the motion ("meat hook" animation).



The above image show the full locomotion, where the character's motion is defined by a large set of example motions.

Both the simple and the full locomotions modes utilize the underlying steering algorithm in order to avoid moving and non-moving obstacles.

Requirements

For the simple locomotion mode, no additional motion data is needed. The character will be moved in the 3D world by changing the character's offset.

For full locomotion, a set of motion examples are required that represent different aspects of locomotion. A set of parameterized motion examples is called a state. To activate full locomotion, a SmartBody character needs the following states:

State	Description
Locomotion	State that includes movements of all different speeds, such as walking, jogging, running and side-moving (strafing)
Step	Single stepping in all directions from a standing state
IdleTurn	Turning in place to face different directions
StartingRight	Starting to walk from a standing position, beginning with the right foot
StartingLeft	Starting to walk from a standing position, beginning with the left foot

Parameters

Parameter	Description	Example
target	move to an (x,z) location in the world	`<locomotion target="100 300"/>`
type	Type of locomotion to be used: basic, example, procedural.	`<locomotion target="100 300" type="basic"/>`
manner	Manner of movement: walk, jog, run, sbm:step, sbm:jump	`<locomotion target="100 300" type="basic" manner="sbm:step"/>`
facing	Final facing direction in global coordinates in degrees of the character after locomotion finishes	`<locomotion target="100 300" facing="90"/>`
sbm:follow	Instructs a character to follow another character as it moves.	`<locomotion sbm:follow="utah"/>`
proximity	How close the character should come to the goal before finishing the locomotion	`<locomotion target="100 300" proximity="75"/>`
sbm:accel	Acceleration of movement, defaults to 2	`<locomotion target="100 300" sbm:accel="4"/>`
sbm:scootaccel	Acceleration of sideways (scooting) movement, defaults to 200	`<locomotion target="100 300" sbm:scootaccel="300"/>`
sbm:angleaccel	Angular speed acceleration, defaults to 450	`<locomotion target="100 300" sbm:angleaccel="600"/>`
sbm:numsteps	Number of steps to take, defaults to 1	`<locomotion target="100 300" sbm:numsteps="2"/>`

Gesture

Description

Characters can control animation via a set of gestures. Gestures can be of a particular type (such as BEAT, NEGATION, POINT) or can be a reference to an existing animation.

Requirements

To use the <gesture> tag, each character requires a gesture map. Please see the section on Configuring Gestures For Characters.

Usage

```
<gesture lexeme="BEAT"/>
```

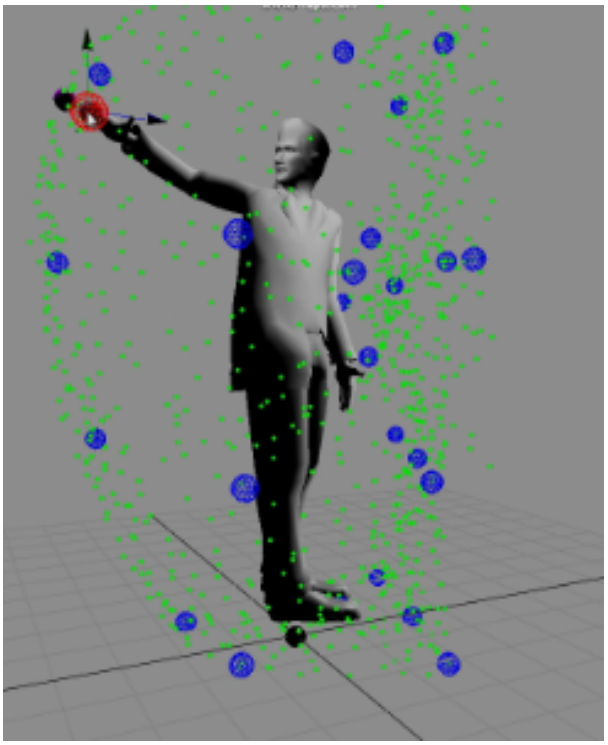
Parameters

Parameter	Description	Example
lexeme	type of gesture	`<gesture lexeme="BEAT"/>`
name	name of the motion to use as a gesture	`<gesture name="BeatLeftHand" />`
mode	Handedness of the gesture. LEFT, RIGHT or BOTH	`<gesture lexeme="BEAT" mode="LEFT"/>`
target	target for POINT and REACH gestures	`<gesture lexeme="POINT" target="BRAD" />`
start	start point of the gesture	`<gesture lexeme="BEAT" start="2"/>`
ready	ready point of the gesture	`<gesture lexeme="BEAT" start="2" ready="3"/>`
stroke	stroke point of the gesture	`<gesture lexeme="BEAT" stroke="5"/>`
relax	relax point of the gesture	`<gesture lexeme="BEAT" relax="3"/>`
end	end point of the gesture	`<gesture lexeme="BEAT" end="4"/>`

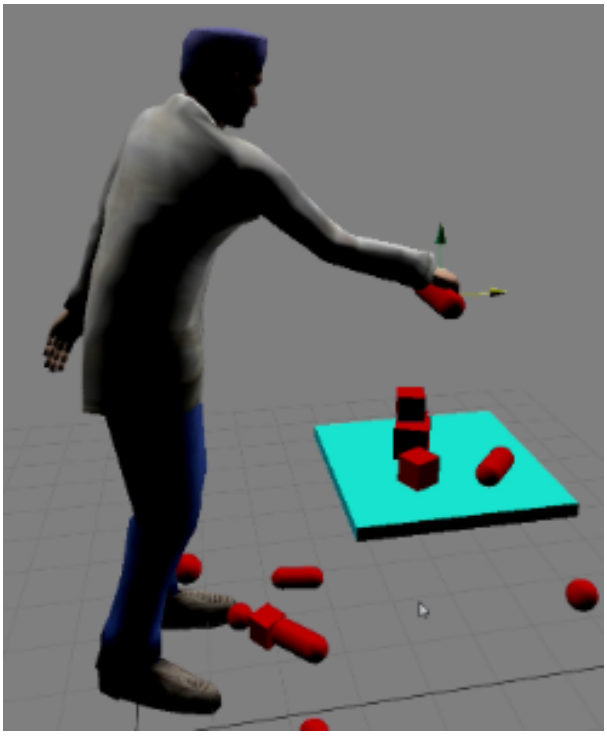
Reach, Grab, Touch, Point

Description

Characters can reach, touch. grab or point at objects in their environment.The reaching algorithm is example-based, so the reaching motion will be of similar quality to the original motion. In addition, SmartBody implements a grabbing control when characters are instructed to pick up or drop objects. The grabbing controller will modify the configuration of the hand so that it matches the shape of the target object.



Shown here is an example of interactive reaching. The blue spheres denote example reaches from a standing position, while the green dots indicate interpolated examples between reaches. The reaching algorithm finds the nearest green example, then uses inverse kinematics (IK) to move the arm to the exact desired location.



In the above two examples, the character reaches for an object then grasps it. Notice that the hand grasp is different for the cube-shaped object (left) than for the capsule-shaped object (right).

Requirements

For reaching, the characters need a set of reaching motion examples.

For grasping, two hand poses are needed: a rest hand pose, and a closed hand pose.

For pointing, a single hand pose that represents the pointing hand pose.

Please see the section on Configuring Reaching and Grasping For Characters for more details.

In the absence of a set of example reaching motions, SmartBody will use inverse kinematics to reach, grasp and touch.

Usage

```
<sbm:reach sbm:action="touch" target="ball"/>
<sbm:reach sbm:action="pick-up" target="ball"/>
<sbm:reach sbm:action="put-down" target="ball"/>
<sbm:reach sbm:action="point-at" target="ball"/>
```

Parameters

Parameter	Description	Example
target	the reach target. Can either be a character/pawn name, or a character:joint	`<sbm:reach target="ball"/>`
sbm:action	pick-up, put-down, touch, point-at	`<sbm:reach sbm:action="touch" target="ball"/>`
sbm:handle	the name of the reaching instance which can be reused later	`<sbm:reach sbm:handle="ball1" sbm:action="pick-up" target="ball"/>`
sbm:foot-ik	whether or not to restrict potential foot sliding by enabling inverse kinematics on the feet. Default is false.	`<sbm:reach sbm:handle="ball1" sbm:action="pick-up" target="ball" sbm:foot-ik="true"/>`
sbm:reach-finish	whether to complete the reaching action by returning to the rest pose	`<sbm:reach target="ball" sbm:reach-finish="true"/>`
sbm:reach-velocity	the end-effector velocity when interpolating between reach poses. Default is 60	`<sbm:reach target="ball" sbm:reach-velocity="100"/>`

sbm:reach-duration	the time to allow the hand to rest on the target object before automatically returning the hand to the rest position. If this value is < 0, the duration is infinite.	`<sbm:reach target="ball" sbm:reach-duration="2"/>`
start	time when the reach motion will start.	`<sbm:reach target="ball" start="5"/>`

Reach Events

The event "reachNotifier" is sent out during a reach behavior. This was created to facilitate the customizing of high-level behaviors with reach/grasp being one of the building blocks.

The user can easily define the following event handler in Python script, which will be triggered during different stages of the reach action.

```
class ReachingHandler(EventHandler):
  def executeAction(self, ev):
    params = ev.getParameters()
    # handle the event here ...
reachingHdl = ReachingHandler()
evtMgr = scene.getEventManager()
evtMgr.addEventHandler("reachNotifier", reachingHdl)
```

The "params" is simply a string in the form of "bml char CHRNAME KEYWORD" or "bml char CHRNAME KEYWORD: VALUE".

CHRNAME is the name of the character;

KEYWORD is one of the following: "pawn-attached", "pawn-released", "reach-returned", "reach-complete", "reach-stateCurrent", "reach-stateNew". See table below.

KEYWORD	Note
pawn-attached, pawn-released	detect if pawn is attached or detached from character's hand
reach-returned, reach-complete	detect if reach has finished. note that if reach-return=false, it still sends out reach-complete when character holds at reach stroke.

reach-stateCurrent, reach-stateNew	current/new reach state (currently it sends out either Return or Idle) for ex: use "reach-stateNew: Idle" to detect when reach has finished.

VALUE is either "Return" or "Idle".

Together with event handlers for locomotion and facing adjustment, characters can be programmed to perform interesting tasks. Please see the example scripts "hot-potato.py" and "hot-potato-multiagent-demo.py" for details.

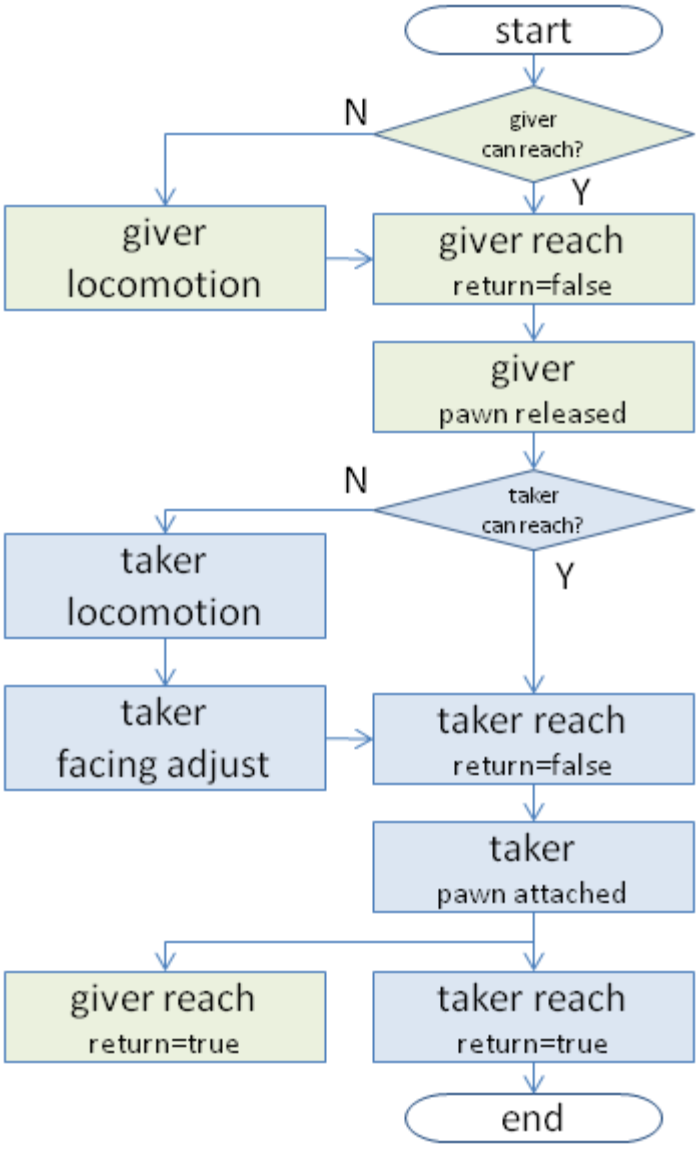
Examples

1. Passing an object from one character (giver) to another (taker).

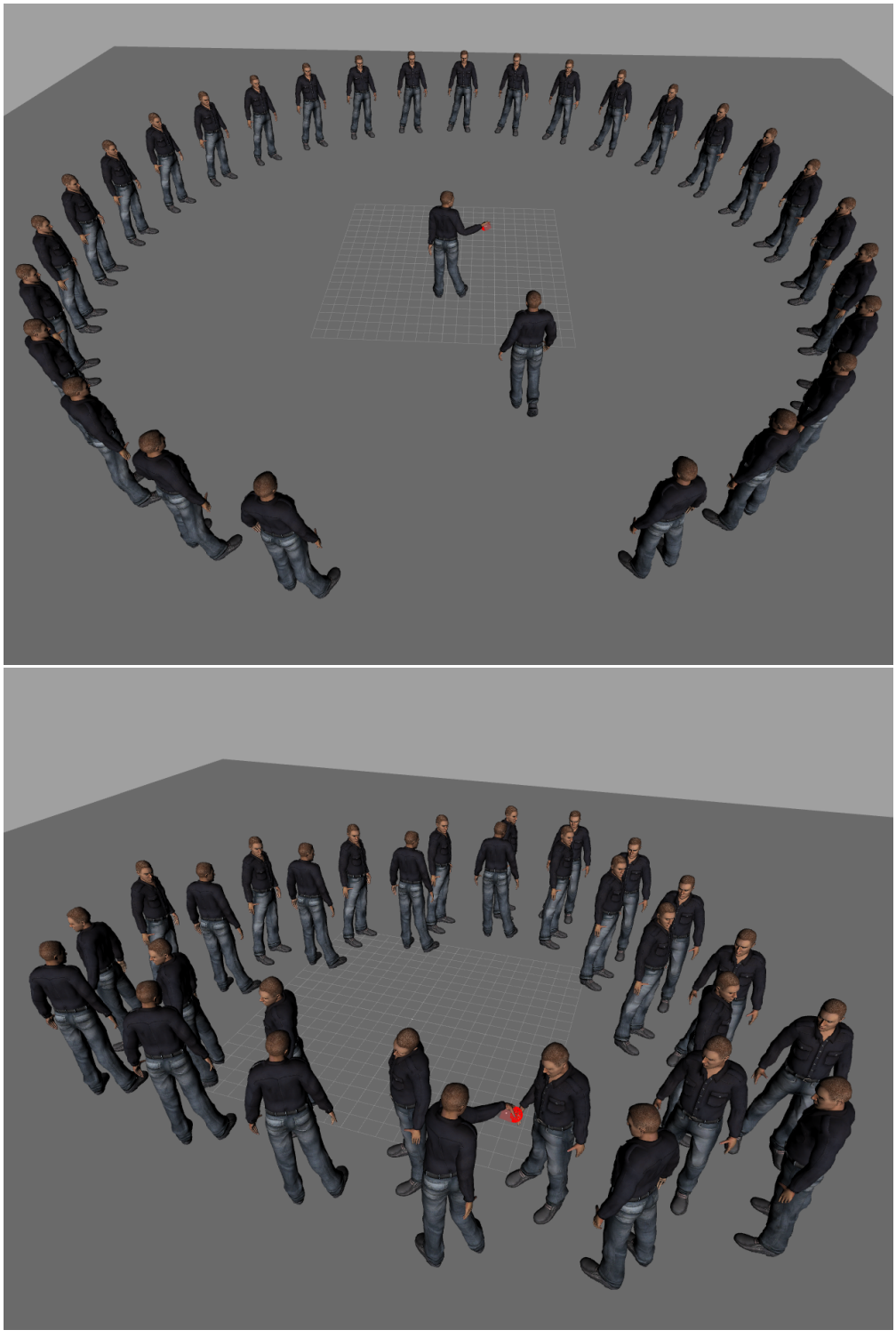
See "hot-potato.py" and the control diagram below for details.

To run this example, follow these steps:

```
# 1. load the script
scene.run("hot-potato.py")

# 2. set up names for agents and object
giver = 'doctor'
taker = 'brad'
obj = 'box'

# 3. make sure object is somewhere can be picked up
dummyPos = offerHandPos(giver, taker)
target = scene.getPawn(obj)
target.setPosition(dummyPos)

# 4. this triggers the object passing, from giver to taker
giverHandToTaker(giver, taker, obj)

# 5. swap the giver and taker, and run it again:
swap()
giverHandToTaker(giver, taker, obj)

# 5. taker release the object:
reset()

# 6. if agent(s) got stuck in the middle of the state, use this to reset:
masterReset()

# 7. you can directly specify giver, taker, obj as follows:
giverHandToTaker('elder', 'utah', 'box')
giverHandToTaker('brad', 'doctor', 'box')
```


start
giver can reach?
N
Y
giver locomotion
giver reach return=false
giver pawn released
taker can reach?
N
Y
taker locomotion
taker facing adjust
taker reach return=false
taker pawn attached
giver reach return=true
taker reach return=true
end

2. "Hot Potato" demo with miltiple agents:



The above two pictures are snapshots taken from the demo. left picture shows the initial setup with 30 agents standing in a circle. As one agent passes the object to the next agent, they form an interesting pattern shown in the right picture.

To run this demo, simply start SmartBody with "hot-potato-multiagent-demo.py". Change "totalChr" inside the script for total head count of the agents in the demo.

Gaze

Description

Characters can gaze or look at other objects in the environment, including other characters and pawns, as well as individual body parts on other characters or themselves. The gazing can be done with four different body areas: eyes, neck, chest and back. Each area can be controlled individually or simultaneously with the other body areas.

The above image is an example of each character gazing using all four body areas at the pawn that is being interactively moved around.

Requirements

Each gaze body area requires a different set of joints. The full set of required joints is as follows:

Usage

```
<gaze target="utah"/>
```

Note that subsequent gazes that use the same body area will run in place of older gazes using the same body areas. Subsequent gazes that use a subset of body areas will override only those body areas in the old gaze that the new gaze

uses. For example:

```
<gaze target="utah"/>, then
<gaze target="brad"/>
```

will cause an initial gaze at utah using all body areas, but then all body areas would shift to gazing a brad. If subsequently, the following command was sent:

```
<gaze target="elder" sbm:joint-range="NECK EYES"/>
```

then the neck and eyes body parts will now gaze at the elder, while the CHEST and BACK body parts will continue to gaze at brad. If somehow the elder gaze was eliminated, the older gaze would then retake control of the NECK EYES and gaze at brad again.

Parameters

Parameter	Description	Example

target	what or who to gaze at. the target can either be the name of character or pawn,
	
	or the name of a joint on a character or pawn. To specify a joint, use the syntax *name:joint*
		`<gaze target="utah"/>`
		
		`<gaze target="utah:l_wrist"/>`

sbm:target-pos	the position to gaze at. defined by x, y, z	`<gaze target="0 0 0"/>`

sbm:joint-range	Which parts of the body to engage during a gaze. Can be any combination of:
	
	EYES, NECK, CHEST, BACK. Can be specified by expressing a range, such as EYES CHEST,
	
	which would include the EYES, the NECK and the CHEST.
		`<gaze target="utah" sbm:joint-range="EYES NECK"/>`
		
		`<gaze target="utah" sbm:joint-range="EYES"/>`
		
		`<gaze target="utah" sbm:joint-range="CHEST BACK"/>`

direction	When gazing at an object, the offset direction from that object.
	
	Must be coupled with the angle attribute.
	
	Direction and it's respective angles are can be:
	
	LEFT (270), RIGHT (90), UP (0), DOWN (180), UPLEFT (315),
	
	UPRIGHT (45), DOWNLEFT (225), DOWNRIGHT (135) or POLAR (DEG).
	
	If the direction is POLAR, the n (DEG) is the polar angle of the gaze direction.
		`<gaze target="utah" direction="LEFT"/>`
		
		`<gaze target="utah" direction="POLAR 137"/>`

angle	Amount of offset from the target direction. If none is specified, default is 30 degrees.	`<gaze target="utah" direction="LEFT" angle="10"/>`
sbm:priority-joint	Which body area should acquire the target. Can be: EYES, NECK, CHEST, BACK. Default is EYES	`<gaze target="utah" sbm:priority-joint="NECK"/>`

sbm:handle	name of the gaze used to recall that gaze at a later time	`<gaze target="utah" sbm:handle="mygaze"/>`

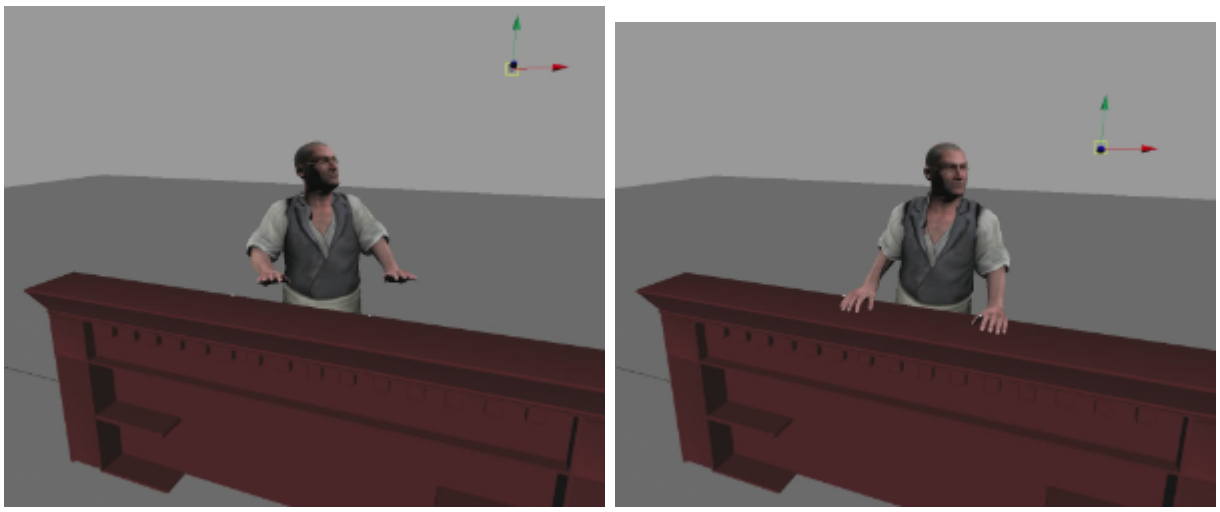
sbm:joint-speed	Overall task speed for NECK or NECK EYES.
	
	Default values for HEAD and EYES are 1000. Specifying one parameter changes the NECK speed.
	
	Specifying two parameters changes the HEAD and EYES speed.
		`<gaze target="utah" sbm:joint-speed="500"/>`
		
		`<gaze target="utah" sbm:joint-speed="800 1500"/>`

sbm:joint-smooth	Decaying average smoothing value	`<gaze sbm:handle="mygaze" sbm:joint-smooth="1"/>`
sbm:fade-in	Fade-out interval to fade out a gaze, requires sbm:handle to be set	`<gaze sbm:handle="mygaze" sbm:fade-out="1"/>`
sbm:fade-out	Fade-in interval to restablish a faded-out gaze, requires sbm:handle to be set	`<gaze sbm:handle="mygaze" sbm:fade-in="1"/>`
start	When the gaze will start	`<gaze target="utah" start="2"/>`

Constraint

Description

SmartBody contains a constraint system that allow a character to maintain an end effector (hand, foot, head) at a particular position. This is useful is you want a character to maintain contact with a particular point in the world (such as on the wall, a table, or another character). Such constraints can be used when trying to maintain contact with an object while gazing and engaging enough of the body to break the point of contact.



The left image shows the character gazing at the pawn (red sphere). Since character's upper body is engaged in the gaze, his hands also shift from their original position (imaging his hands resting on a table). With two constraints enabled (one

for each hand), the right image shows the character gazing at the pawn again, but the hands maintain their original positions.

Requirements

A chain of joints from the root to the end effector of any name.

Usage

```
<sbm:constraint
effector="r_wrist"
sbm:effector-root="r_sternoclavicular"
sbm:fade-in="1"
sbm:handle="myconstraint"
target="elder:l_wrist"/>
```

Note that any number of end effectors can simultaneously use a different constraint. Note that using constraints is somewhat performance-intensive, and may reduce the frame rate of the simulation. Also note that the constraint system in SmartBody is intended to keep the character in particular positions or orientations, but not for bringing the character into those positions. Thus, constraints are most effective when their effect is subtle or small.

Parameters

Parameter	Description	Example
target	target pawn/joint whose the positional or rotational values will be used as constraints	`<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist"/>`
sbm:handle	Handle of this constraint instance, can be reused during later constraint commands.	`as above`
sbm:root	the root joint for the current character. by default it is the base joint.	`<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist" sbm:root="base"/>`
effector	the end effector where the constraint will be enforced through optimization	`as above`
sbm:effector-root	the influence root of this end effector. anything higher than this root will not be affected by optimization	`as above`
sbm:fade-in	the time for the constraint to blend in. this option is ignored if set to negative	`as above`
sbm:fade-out	the time for the constraint to blend out. this option is ignored if set to negative	`<sbm:constraint sbm:handle="myconstraint" sbm:fade-out="1'/>`
sbm:constraint-type	the constraint type to be enforced. it can be positional or rotational constraint	`<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist" sbm:root="base" sbm:constraint-type="pos"/>`

pos-x	the x positional offset added on top of positional constraint	<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist" sbm:root="base" sbm:constraint-type="pos" pos-y="10" pos-x="15" pos-z="5"/>
pos-y	the y positional offset added on top of positional constraint	as above
pos-z	the z positional offset added on top of positional constraint	as above
rot-x	the x rotational offset added on top of positional constraint	<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist" sbm:root="base" sbm:constraint-type="pos" rot-y="10" rot-x="15" rot-z="5"/>
rot-y	the y rotational offset added on top of positional constraint	as above
rot-z	the z rotational offset added on top of positional constraint	as above
start	when to start the constraint	<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist" sbm:root="base" start="3" ready="4"/>
ready	when the constraint is fully engaged	as above

Head Movements

Description

Characters can perform head nods, head shakes, and head tosses (sideways head movements).

In the above image, the character's head tilt position is controlled via BML. Nods (movement around the x-axis) can be combined with shakes (movement around the y-axis) and tosses (movement around the z-axis).

Requirements

The <head> behavior requires a skeleton that includes 3 joints named *spine4*, *spine5* and *skullbase* in that order, with *spine4* the parent of *spine5*, which is the parent of *skullbase*. The head movement is spread among those joints.

Usage

```
<head type="NOD"/>
```

Head movements will take 1 second by default. This can be adjusted by setting the synchronization points. For example, to set a 3 second head shake:

```
<head type="SHAKE" start="0" end="3"/>
```

Parameters

Parameter	Description	Example
type	Type of head movement: NOD (up-down), SHAKE (left-right), TOSS (side-side) WIGGLE and WAGGLE (multiple nods of varying intensity)	`<head type="NOD"/>`
repeats	number of head movements	`<head type="NOD" repeats="2"/>`
velocity	frequency of head movements, default is 1. .5 indicates twice the speed, 2 indicates half speed	`<head type="NOD" velocity=".5"/>`
amount	magnitude of head movement, from 0 to 1, default is .5	`<head type="NOD" amount=".8"/>`
sbm:smooth	smoothing parameter when starting and finishing head movements	
sbm:period	period of nod cycle, default is .5	`<head type="NOD" sbm:period=".5"/>`
sbm:warp	warp parameter for wiggle and waggle, default is .5	`<head type="WIGGLE" sbm:warp=".8"/>`
sbm:accel	acceleration parameter for wiggle and waggle, default is .5	`<head type="WIGGLE" sbm:accel=".9"/>`
sbm:pitch	pitch parameter for wiggle and waggle, default is 1	`<head type="WAGGLE" sbm:pitch=".8"/>`
sbm:decay	decay parameter for wiggle and waggle, default is .5	`<head type="WAGGLE" sbm:decay=".7"/>`
start	when the head movements starts	`<head type="NOD" start="2"/>`
ready	preparation phase of head movement	`<head type="NOD" start="2" ready=".5"/>`
stroke	the mid-point of the head movement	`<head type="NOD" stroke="3"/>`
relax	finishing phase of head movement	`<head type="NOD" relax="4"/>`

end	when the head movement ends	`<head type="NOD" end="5"/>`

Face

Description

Characters can change the expressions in their faces by activating one or more Facial Action Units, abbreviated as Action Units (AU). Each AU activates a different part of the face, say, raising an eyebrow or widening the nose. The AUs can be combined together to form more complex facial expressions

Note that the facial movements expressed by the <face> BML command are separate from lip synching, which manages the mouth and tongue shapes in order to match speech. There are AUs that can operate on the tongue, mouth and lips, but they would be activated separately from the lip sync activation. For example, if you wanted your character to open his mouth and raise his eyebrows to express surprise, the open mouth expression should be activated by triggering AU 26 which lowers the jaw. While a similar expression could be accomplished by using a viseme which would lower the jaw, visemes are typically only activated in response to speech, while AUs are activated via BML commands.

Requirements

Characters need to have a Face Definition set up with any number of AUs.Please see the section on Configuring Characters for more detail. Note that both joint-driven faces and blendshape-driven faces are supported. In the case of joint-driven faces, an static pose will correspond to each AU. In the case of blendshape-driven faces, SmartBody will transmit an activation value for that shape, and it is the task of the renderer to interpret that activation value and activate the appropriate shape.

Usage

```
<face type="facs" au="1" amount="1"/>
```

Complex facial movements will contain numerous <face> commands using different attributes. For example, a happy expression could be AU 6 and AU 12:

<face type="facs" au="6" amount="1"/><face type="facs" au="12" amount="1"/>

or sadness could be AU 1 and AU 4 and AU 15:

<face type="facs" au="1" amount="1"/><face type="facs" au="4" amount="1"/><face type="facs" au="15" amount="1"/>

Parameters

Parameter	Description	Example
type	Type of facial expression. Only 'facs' is currently supported.	`<face type="facs" au="1" amount="1"/>`
side	Which side of the face will be activated; LEFT, RIGHT or BOTH	`<face type="facs" au="1" side="LEFT" amount="1"/>`
au	Action Unit number	`<face type="facs" au="26" amount="1"/>`
start	when the face movements starts	`<face type="facs" au="26" amount="1" start="2"/>`
ready	when the face movement is fully blended in, default is .25 seconds	`<face type="facs" au="26" amount="1" start="0" ready="1"/>`
stroke	the mid-point of the face movement	`<face type="facs" au="26" amount="1" stroke="4"/>`
relax	when the face movement starts blending out, default is .25 seconds	`<face type="facs" au="26" amount="1" relax="2"/>`
end	when the face movement ends	`<face type="facs" au="26" amount="1" start="0" end="4"/>`

Speech

Description

Characters can synthesize speech and match that speech with lip syncing either from prerecorded audio, or from a text-to-speech (TTS) engine. Note that the <speech> tag is a request to generate speech, but it does not contain a full description of

Requirements

A character must have a Face Definition set up with a set of visemes. If the character does not have a Face Definition with visemes that match those of the speech engine, then the sound of the speech will still be played, but the character will do no lip synchronization.

To generate the sound for prerecorded audio files, SmartBody requires that the sound file (.wav, .au) must be placed in a directory alongside of an XML file that includes the visemes and timings for that audio. Both the XML file and the audio file must have the same name, with different suffices (the suffix for the XML file will be .xml). Please see the documentation on Prerecorded Audio for more details.

For text-to-speech (TTS), a TTS relay must be running or the characters must use the internal Festival TTS engine.Please see the section on Using Text-To-Speech for more details.

Usage

```
<speech>hello, my name is Utah</speech>
```

By default, speech BML can be described either by using plain text (type="text/plain") or SSML (type="application/ssml+xml"). If no type attribute is specified, the BML realizer assumes type="text/plain". Note that some speech relay systems can support SSML tags, where you can specify loudness, prosody, speach breaks and so forth, but this depends on the abilities of those relays, and not on SmartBody.

Note that the purpose of the <speech> tag is to either request the TTS engine to generate the audio and the subsequent viseme timings, or to gather an existing audiofile and timings and to play it. Please refer to the section on Using Speech for more details.

Synchronizing Speech Using a Text-to-Speech (TTS) Engine

In order to synchronize other behaviors with speech using TTS, the speech content must be marked with <mark name=""> tags as follows:

```
<speech type="application/ssml+xml" id="myspeech">
<mark name="T0"/>hello
<mark name="T1"/>
<mark name="T2"/>my
<mark name="T3"/>
<mark name="T4"/>name
<mark name="T5"/>
<mark name="T6"/>is
<mark name="T7"/>
<mark name="T8"/>Utah
<mark name="T9"/>
</speech>
<head type="NOD" start="myspeech:T4"/>
```

The <mark> tags are instructions for the text-to-speech engine to replace those markers with the actual timings of the BML.

The above command will place synchronization markers before and after the spoken text, which allows you to coordinate other behaviors, in this case a head nod, at various points during the speech. Note that the <mark> marker immediately before a word is coordinated with the start of that word, while the marker after the word is coordinated with the end of that word. In the above example, the character will start the head nodding at the same type that the word 'name' is beginning to be uttered. In addition, other behaviors can access the start and end synchronization points of a speech, which correspond to the point where the first word is spoken and after when the last word is spoken, respectively.

Synchronizing Speech Using Prerecorded Audio

Speech that uses prerecorded audio can also be synchronized with other behaviors. When using prerecorded audio, we assume that the speech timings for the utterance are already known and have been recorded into an XMLfile. This XML file will also include visemes and their respective timings that are synchronized with the words, like the following:

```
<bml>
<speech type="application/ssml+xml">
<sync id="T0" time=".1"/>hello
<sync id="T1" time=".2"/>
<sync id="T2" time=".35"/>my
<sync id="T3" time=".4"/>
<sync id="T4" time=".6"/>name
<sync id="T5" time=".72"/>
<sync id="T6" time=".9"/>is
<sync id="T7" time=".1.07"/>
<sync id="T8" time="1.4"/>Utah
<sync id="T9" time="1.8"/>

<lips viseme="_" articulation="1.0" start="0" ready="0.0132" relax="0.0468" end="0.06"/>
<lips viseme="Z" articulation="1.0" start="0.06" ready="0.0952" relax="0.1848" end="0.22"/>
<lips viseme="Er" articulation="1.0" start="0.22" ready="0.2442" relax="0.3058" end="0.33"/>
<lips viseme="D" articulation="1.0" start="0.33" ready="0.3586" relax="0.4314" end="0.46"/>
<lips viseme="OO" articulation="1.0" start="0.46" ready="0.4644" relax="0.4756" end="0.48"/>
<lips viseme="oh" articulation="1.0" start="0.48" ready="0.4888" relax="0.5112" end="0.52"/>
</speech>
</bml>
```

The above XMLfile will reside in a directory, and in that same directory the audio file (.wav) should exists with the same name, with the .wav extension. To use such data:

```
<speech ref="myspeech"/>
```

Where the files myspeech.bml, and myspeech.wav are in the location designated for audio files (based on the mediapath and the voice code, please see the section on configuring Prerecorded Speech for Characters for details on where this location should be).

In order to coordinate behaviors with the prerecorded speech, you include <mark> tags, the same way you would do so using TTS. For example:

```
<speech type="application/ssml+xml" id="myspeech">
<mark name="T0"/>hello
<mark name="T1"/>
<mark name="T2"/>my
<mark name="T3"/>
<mark name="T4"/>name
<mark name="T5"/>
<mark name="T6"/>is
<mark name="T7"/>
<mark name="T8"/>Utah
<mark name="T9"/>
</speech>
<head type="NOD" start="myspeech:T4"/>
```

Note that the name attribute in the <mark> tags in your BML command must match the id attribute of the <sync> tags in the XML file. Using this example, the head nod will occur at time .4, since that is the <sync> time as described in the XML file.

Please note that prerecorded audio can contain instructions for individual visemes using the <lips> tags as in the above example. In that case, each viseme has an explicit start and end time. Altervatively, the BML file can contain instructions to play an arbitrary curve for each viseme using the <curves> tag, as follows:

```
<bml>
<speech type="application/ssml+xml">
<sync id="T0" time=".1"/>hello
<sync id="T1" time=".2"/>
<sync id="T2" time=".35"/>my
<sync id="T3" time=".4"/>
<sync id="T4" time=".6"/>name
<sync id="T5" time=".72"/>
<sync id="T6" time=".9"/>is
<sync id="T7" time=".1.07"/>
<sync id="T8" time="1.4"/>Utah
<sync id="T9" time="1.8"/>

<curves>
<curve name="NG" num_keys="9" >0.645000 0.000000 0.000000 0.000000 0.710728 0.607073 0.000000 0.000000 1.011666
0.000000 0.000000 0.000000 1.994999 0.000000 0.000000 0.000000 2.044607 0.020316 0.000000 0.000000 2.061665
0.000000 0.000000 0.000000 2.211665 0.000000 0.000000 0.000000 2.266716 0.262691 0.000000 0.000000 2.311665
0.000000 0.000000 0.000000 </curve>
<curve name="Er" num_keys="3" >0.028333 0.000000 0.000000 0.000000 0.184402 0.998716 0.000000 0.000000 0.261667
0.000000 0.000000 0.000000 </curve>
<curve name="F" num_keys="3" >0.445000 0.000000 0.000000 0.000000 0.578196 0.982450 0.000000 0.000000 0.661667
0.000000 0.000000 0.000000 </curve>
<curve name="Th" num_keys="8" >1.361666 0.000000 0.000000 0.000000 1.450796 0.852155 0.000000 0.000000 1.545923
0.000000 0.000000 0.000000 1.622119 1.000000 0.000000 0.000000 1.694999 0.000000 0.000000 0.000000 2.194999
0.000000 0.000000 0.000000 2.308685 0.990691 0.000000 0.000000 2.411665 0.000000 0.000000 0.000000 </curve>
<curve name="Z" num_keys="3" >-0.221667 0.000000 0.000000 0.000000 0.028095 0.995328 0.000000 0.000000 0.195000
0.000000 0.000000 0.000000 </curve>
</curves>

</speech>
</bml>
```

Note that the curve data is in the format: time, value, tangent1, tangent2 (the tangent data can be ignored). The <curve> data expresses an activation curve for a particular viseme.

Parameters

Parameter	Description	Example

type	Type of content. Either text/plain or application/ssml+xml. Default is text/plain.	`<speech type="text/plain">`
		`Four score and seven years ago`
		`</speech>`
		
		`<speech type="application/ssml+xml">`
		`Four score and seven years ago`
		`</speech>`

ref	Speech file reference. Used to determine which sound file and XML file to use that is associated with the speech.	<speech ref="greeting"/> </speech>

<mark name=""/>	Marker used to identify timings of speech before actual timings are known.
	
	Typically, the names are "Tn" where n is a whole number that is incremented
	
	before and after each word. T0, T1, T2, etc.
		`<speech type="text/plain">`
		`<mark name="T0"/>Four`
		`<mark name="T1"/>`
		`<mark name="T2"/>score`
		`<mark name="T3"/>`
		`<mark name="T4"/>and`
		`<mark name="T5"/>`
		`<mark name="T6"/>seven`
		`<mark name="T7"/>`
		`<mark name="T8"/>years`
		`<mark name="T9"/>`
		`<mark name="T10"/>ago`
		`<mark name="T11"/>`
		`</speech>`

Eye Saccade

Description

The rapid movements of the eyes are called saccades. Characters can use eye saccade models that emulate listening, speaking and thinking behaviors. In addition, explicit eye saccades can be specified to show scanning an object, looking away to reduce cognitive load, looking up to express thinking, and so forth.

Requirements

The <saccade> behavior requires a skeleton that includes 2 joints, one for each eye named *eyeball_left* and *eyeball_right*.

Usage

```
<saccade mode="TALK"/>
```

Runs the TALK saccade model. This randomizes the eye movements according to a data model as described by the "Eyes Alive" SIGGRAPH paper by Lee, Badler and Badler (you can read the paper "Eyes Alive", Lee, Badler, Badler). Note that the user does not have individual

Eye saccades can also be run on an individual basis by using the direction, magnitude and sbm:duration attributes.

```
<saccade direction="45" magnitude="15" sbm:duration=".5"/>
```

Parameters

Parameter	Description	Example
mode	Starts running an eye saccade model (LISTEN, TALK or THINK). Note that using the mode attribute, saccades will be run randomly according to the data in the model.	`<saccade mode="TALK"/>`
finish	Stops running eye saccades according to one of the above modes. Can be 'true' or 'false'.	`<saccade finish="true"/>`
angle-limit	limit of eye movement. Defaults to 10 degrees in LISTEN mode, and 12 degrees in TALK mode	`<saccade mode="TALK" angle-limit="20"/>`
direction	polar direction of eye movement, from -180 to 180 degrees	`<saccade direction="45" magnitude="15" sbm:duration=".5"/>`
magnitude	amount of eye saccade movement in degrees	`<saccade direction="-45" magnitude="10" sbm:duration=".5"/>`
sbm:duration	duration of eye saccade	`<saccade direction="0" magnitude="15" sbm:duration="1"/>`

In addition, the eye saccade model can be adjusted by setting up the percentage of time that the eye will use a particular saccadic angle as follows:

Parameter	Description	Example
sbm:bin0	Percentage chance of saccade using 0 degree angle	`<saccade sbm:bin0="20" sbm:bin45="10" sbm:bin90="0" sbm:bin135="0" sbm:bin180="0" sbm:bin225="0" sbm:bin270="0" sbm:bin315="70" />`
sbm:bin45	Percentage chance of saccade using 45 degree angle	see example above
sbm:bin90	Percentage chance of saccade using 90 degree angle	see example above
sbm:bin135	Percentage chance of saccade using 135 degree angle	see example above
sbm:bin180	Percentage chance of saccade using 180 degree angle	see example above
sbm:bin225	Percentage chance of saccade using 225 degree angle	see example above
sbm:bin270	Percentage chance of saccade using 270 degree angle	see example above
sbm:bin315	Percentage chance of saccade using 315 degree angle	see example above
sbm:mean	gaussian distribution mean, default is 100	`<saccade sbm:mean="40" sbm:variant="20" />`
sbm:variant	gaussian distribution variant, default is 50	same as above

Event

Description

Events can be triggered based on various synchronization points of a BML command. These events can be handled by SmartBody's event handlers, or they can be explicit commands to perform some action or change character state. One way to use such events is to coordinate multiple characters with a single BML command, since BML typically only contains instructions for a single character. For example, a BML command directing a character to talk could include an event that is triggered when that utterance is finished for a different character to nod their head in agreement.

Requirements

None

Usage

```
<event stroke="0" message="sbm python print 'hello'/>
```

This will send out an event at time 0 to the VH message bus that will be received by SmartBody (which handles all

messages with an 'sbm' prefix) and then print out the word 'hello'.

```
<speech id="sp1">
<mark name="T0"/>hello
<mark name="T1"/>
<mark name="T2"/>how
<mark name="T3"/>
<mark name="T4"/>are
<mark name="T5"/>
<mark name="T6"/>you?
<mark name="T7"/>
</speech>

<event stroke="sp1:T4" message="sbm python print 'hello'/>
```

This will send out an event before the word 'are' is spoken.

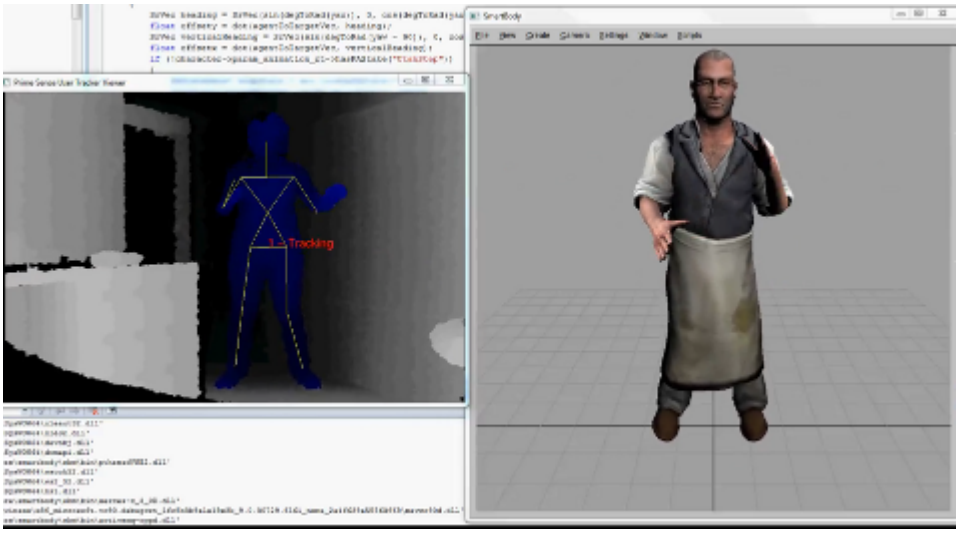
Parameters

Parameter	Description	Example
stroke	When the event will occur. Can be an absolute local time associated with BML block, or a relative time based on synchronization points of other BML behaviors.	`<event stroke="3" message="sbm python print 'hello'/>`
message	event to be sent. Will be sent over the VH message bus. There is no limit to the complexity of the message, but it must adhere to XML syntax.	`<event stroke="3" message="sbm python print 'hello'/>`

Using SmartBody With Kinect

SmartBody characters can receive data and be controlled by a Kinect camera. This works as follows:

1. A standalone application reads the data from the Kinect
2. The standalone application sends a message over the ActiveMQ network containing joint positions and orientations
3. SmartBody receives the ActiveMQ message, then overrides the joint position and orientations of the joints of one or more characters
4. Any joints not overridden (such as the face and fingers) will retain the existing character positions and orientations.



Note that the skeleton from the Kinect does not match the topology or bone lengths of the SmartBody character. The online retargeting is done by mapping the standard names of the Kinect character to similar names on the SmartBody character. Thus the "RightWrist" of the Kinect character will map to the "r_wrist" of the SmartBody character.

Also note that any motion capture system could use a similar mechanism to control a SmartBody character by sending the appropriate SmartBody commands in order to override the joint information.

Building Kinect Application on Windows

1	Download and install OpenNI 32-bit Development Edition http://www.openni.org/Downloads/OpenNIModules.aspx Select 'OpenNI Binaries' , 'Stable', then choose the Windows x86 32-bit Development Edition

2	Download and install NITE http://www.openni.org/downloadfiles/ Select 'OpenNI Compliant Middleware Binaries' , 'Stable', then choose the PrimeSense NITE Windows x86 Development Edition
3	Download the Kinect drivers https://github.com/avin2/SensorKinect/blob/unstable/Bin/SensorKinect091-Bin-Win32-v5.1.0.25.msi
4	

Building Kinect Application on Linux

The original instructions for installing OpenNI and NITE are found here: http://www.greenfoot.org/doc/kinect/ubuntu.html, but is detailed below:

	Instructions

1	Retrieve and install the OpenNI drivers
	
	`cd OpenNI`
	
	`wget` http://www.greenfoot.org/doc/kinect/OpenNI-Linux32.tar.bz2
	
	`tar -jxf OpenNI-Linux32.tar.bz2`
	
	`sudo ./install.sh`
	
	`cd ..`

2	Retrieve and install NITE
	
	`mkdir NITE`
	
	`cd NITE`
	
	`wget` http://www.greenfoot.org/doc/kinect/NITE-Linux32.tar.bz2
	
	`tar -jxf NITE-Linux32.tar.bz2`
	
	`echo '0KOIk2JeIBYClPWVnMoRKn5cdY4=' | sudo ./install.sh`
	
	`cd ..`

3	Install the Kinect driver for Linux
	`mkdir Kinect`
	`cd Kinect`
	`wget` http://www.greenfoot.org/doc/kinect/SensorKinect-Linux32.tar.bz2
	`tar -jxf SensorKinect-Linux32.tar.bz2`
	`sudo ./install.sh`
	`cd ..`

4	Make sure that your ActiveMQ service is running, since messages between the standalone kinect application and SmartBody are managed via ActiveMQ. If it is not running, run: `sudo activemq start`

5	Build the kinecttracker application located in smartbody/lib:
	uncomment the line in smartbody/lib/CMakeLists.txt that says:
	add_directory (kinecttracker)
	by removing the hash mark (#) in front of that line. This will add the kinecttracker application to the standard SmartBody build process.
	Then build SmartBody:
	`cd smartbody/build`
	`make install`
	You should see an application called smartbody/lib/kinecttracker/kinectracker

6	Run sbm-fltk: `./smartbody/core/smartbody/sbm/bin/sbm-fltk`

7	The character that is controlled by kinect depends on the attribute 'receiverName', which is set to 'kinect1' by default, which is the first character the kinect tracks. If you want the SmartBody character to respond to the second or greater skeleton tracked by kinect, set the 'receiverName' to 'kinect2' or 'kinect3':
	
	From Python:
	`c = scene.getCharacter("utah")`
	`c.setAttribute("receiverName", "kinect2")`

7	Run the kinecttracker application: `./smartbody/lib/kinecttracker/kinecttracker` Then approach the kinect camera and track your pose. You should be able to control one or more characters with the kinect data.

Building Kinect Application on OSX

We use the OpenNI library (http://www.openni.org)

Appendix 1: Building SmartBody

There are several SmartBody applications that can be built:

Application	Comment
sbm-fltk	a standalone SmartBody application and scene renderer
smartbody-dll	a dynamic library that can be incorporated into a game engine or other application
sbmonitor	(*Windows only*) A separate application that can be used to inspect, monitor and control a running SmartBody instance.
sbm-batch	a standalone SmartBody application that connects to smartbody-dll without a renderer
TtsRelayGui	(*Windows only*) incorporates any text-to-speech engine that uses the TtsRelay interface
FestivalRelay	Text-to-speech engine that uses Festival
MsSpeechRelay	(*Windows only*) Text-to-speech engine that uses Microsoft's built-in speech engine
OgreViewer	The Ogre rendering engine connected to SmartBody
sbdesktop	(*Windows only*) A fully-functional SmartBody environment rendered on the desktop.

In addition, there are several mobile applications that can be built:

Mobile Application	Comment
sbmjni	(*Android only*) Simple OpenGL ES front end for SmartBody for Android devices
sbm-ogre	(*Android only*) Ogre3D engine connected with SmartBody for Android devices
vh-wrapper	(*Android only*) SmartBody interface to Unity3D.
smartbody-openglES	(*iOS only*) Simple OpenGL ES front end for SmartBody for iOS devices
smartbody-ogre	(*iOS only*) Ogre3D engine connected with SmartBody for iOS devices
smartbody-unity	(*iOS only*) Unity3D project for SmartBody

Building SmartBody for Windows

You will need Visual Studio 2008 or higher to build SmartBody. In the top level SmartBody directory, Open the solution file 'vs2008.sln' for Visual Studio 2008, or 'vs2010.sln' for Visual Studio 2010 and choose Build -> Build Solution. All libraries needed for the build have been included in the SmartBody repository.

You will need to install an ActiveMQ server if you wish to use the message system, although you can run SmartBody without it. The message system is used to send commands remotely to SmartBody. Download and install it from: http://ac

tivemq.apache.org/activemq-543-release.html

Building SmartBody for Linux

The Linux build requires a number of packages to be installed. For Ubuntu installations, the command apt-get can be used to retrieve and install those packages, for example:

```
sudo apt-get install cmake
```

Other Linux flavors can use rpm or similar installer.

You need to have the following packages installed:

Linux Packages Needed for SmartBody build
cmake
g++
libxerces-c3-dev
libgl1-mesa-dev
libglu1-mesa-dev
libglut-mesa-dev
xutils-dev
libxi-dev
freeglut3-dev
libglut3
libglew-dev
libxft-dev
libapr1-dev
libaprutil1-dev
libcppunit-dev
liblapack-dev
libblas-dev
libf2c2-dev
build-essential
mono-devel
mono-xbuild
python-dev
libopenal-dev
libsndfile-dev
libalut-dev

Linux packages needed for text-to-speech engine
festival-dev

Linux packages needed for Ogre3D viewer
libzzip-dev
libxaw7-dev
libxxf86vm-dev
libxrandr-dev
libfreeimage-dev
nvidia-cg-toolkit
libois-dev

Linux packages needed for test suite
imagemagick

The Ogre-based renderer is not built by default. To build it, you will need to download the 1.6.5 source (www.ogre3d.org/download/source), build and install it into /usr/local. Next, uncomment the core/ogre-viewer directory from the core/CMakeLists.txt file. This will add the OgreViewer application to the build - the binary will be located in core/ogre-viewer/bin.

To build the Linux version:

1	Install the packages indicated above	

2	Download and install Python 2.6:
	
	Download Python from: http://www.python.org/download/releases/2.6.8/ (note that the latest version of Python 2.6 may be different)
	
	Place in lib and unpack, build and install into /usr/local/ using:
	
	`./configure --enable-shared`
	
	`make`
	
	`sudo make install`	

3	Download and install boost:
	
	Download Boost from: http://sourceforge.net/projects/boost/files/boost/1.44.0/boost_1_44_0.tar.gz/download
	
	Place in lib/ and unpack, build and install into /usr/local using:
	
	`./bootstrap.sh`
	`--with-python-version=2.6`
	`./bjam --with-python`
	`sudo ./bjam install`	

4	Download and install the Boost numeric bindings:
	
	Download from:http://mathema.tician.de/news.tiker.net/download/software/boost-numeric-bindings/boost-numeric-bindings-20081116.tar.gz
	
	Place in lib/ and unpack using:
	
	`tar -xvzf`
	`boost-numeric-bindings-20081116.tar.gz`
	`cd boost-numeric-bindings`
	`sudo cp -R boost/numeric/bindings`
	`/usr/local/include/boost/numeric`	

5	Download and install FLTK 1.3.
	
	Download from: http://ftp.easysw.com/pub/fltk/1.3.0/fltk-1.3.0-source.tar.gz
	
	Place in lib/, unpack and configure:
	
	`./configure --enable-gl`
	`--disable-xinerama`
	`make`
	`sudo make install`	

6	Download and install ActiveMQ.
	
	Download from: http://www.apache.org/dyn/closer.cgi/activemq/activemq-cpp/source/activemq-cpp-library-3.4.0-src.tar.gz
	
	unpack to temp folder
	
	`./configure --disable-ssl`
	`make`
	`sudo make install`
	`sudo ldconfig`	

7	Get Open Dynamics Engine (ODE).
	
	Download ode-0.11.1 from:http://sourceforge.net/projects/opende/files/
	Place in lib/, unpack and configure:
	
	32 bits:
	
	`./configure --with-drawstuff=none --enable-double-precision`
	
	64 bits:
	
	`./configure --with-drawstuff=none --with-pic --enable-double-precision`
	
	`make`
	
	`sudo make install`	

8	(Optional) Build the elsender:
	
	`cd lib/elsender`
	`xbuild elsender.csproj (ignore post-build error)`	

9	(Optional) Build the Ogre engine:
	
	Download the 1.6.5 version fromfrom: www.ogre3d.org/download/source
	
	`./configure`
	`sudo make install`	

10	Build speech tools and Festival that are located in the local SmartBody directory:
	`cd lib/festival/speech_tools`
	`./configure`
	`make install`
	`cd ../festival`
	`./configure`
	`make install`	

11	Make and install SmartBody:
	First, generate the Makefiles with cmake:
	`mkdir buildfolder`
	`cmake ..`
	Go to the build directory, make and install the applications
	`cd buildfolder`
	`make install`	

Building SmartBody for OSx

To build the OSX version:

1	Download and install boost:
	Download Boost from: http://sourceforge.net/projects/boost/files/boost/1.44.0/boost_1_44_0.tar.gz/download
	Place in lib/ and unpack, build and install into /usr/local using:
	`./bootstrap.sh`
	`–with-python-version=2.6`
	`–with-python-root=/System/Library/Frameworks/Python.framework/Versions/2.6`
	`./bjam --with-python python=2.6`
	`sudo ./bjam install`
	Note that this will link boost against the 2.6 version of Python, even though the 2.7 version might also be installed.	

2	Download and install the Boost numeric bindings:
	
	Download from:http://mathema.tician.de/news.tiker.net/download/software/boost-numeric-bindings/boost-numeric-bindings-20081116.tar.gz
	
	Place in lib/ and unpack using:
	
	`tar -xvzf boost-numeric-bindings-20081116.tar.gz`
	`cd boost-numeric-bindings`
	`sudo cp -R boost/numeric/bindings /usr/local/include/boost/numeric`	

3	Download and install FLTK 1.3.
	
	Download from: http://ftp.easysw.com/pub/fltk/1.3.0/fltk-1.3.0-source.tar.gz
	
	Place in lib/, unpack and configure:
	
	`./configure --enable-gl --enable-shared --disable-xinerama`
	`make`
	`sudo make install`	

4	Download and install the ActiveMQ libraries.
	
	Download from:http://www.apache.org/dyn/closer.cgi/activemq/activemq-cpp/source/activemq-cpp-library-3.4.4-src.zip
	
	Then rebuild activemq-cpp using:
	
	`./configure --disable-ssl`
	`make`
	`sudo make install`	

5	Get Open Dynamics Engine (ODE).
	
	Download ode-0.11.1 from:http://sourceforge.net/projects/opende/files/
	Place in lib/, unpack and configure:
	
	32 bits:
	`./configure --with-drawstuff=none --enable-double-precision`
	
	64 bits:
	`./configure --with-drawstuff=none --with-pic --enable-double-precision`
	`make`
	`sudo make install`	

6	Build and install xerces
	
	http://www.takeyellow.com/apachemirror//xerces/c/3/sources/xerces-c-3.1.1.tar.gz
	`tar zxvf xerces-c-3.1.1.tar.gz`
	`cd xerces-c-3.1.1`
	`./configure`
	`make`
	`sudo make install`	

7	Build and install GLEW
	
	https://sourceforge.net/projects/glew/files/glew/1.6.0/glew-1.6.0.tgz/download
	`tar zxvf glew-1.6.0.tgz`
	`cd glew-1.6.0`
	`make`
	`sudo make install`	

8	Build and install OpenAL
	
	http://connect.creativelabs.com/openal/Downloads/openal-soft-1.13.tbz2
	`tar zxvf openal-soft-1.13.tbz2`
	`cd openal-soft-1.13`
	`cmake .`
	`make`
	`sudo make install`	

9	Build and install FreeALUT
	
	http://connect.creativelabs.com/openal/Downloads/ALUT/freealut-1.1.0-src.zip
	
	`unzip freealut-1.1.0-src.zip`
	`cd freealut-1.1.0-src`
	`cmake .`
	`make`
	`sudo make install`	

10	Build and install libsndfile http://www.mega-nerd.com/libsndfile/files/libsndfile-1.0.25.tar.gz	

11	(Optional) Build the elsender:
	
	`cd lib/elsender`
	`xbuild elsender.csproj (ignore post-build error)`	

12	(Optional) Build the Ogre engine:
	
	Download the 1.6.5 version fromfrom: www.ogre3d.org/download/source
	
	`./configure`
	`sudo make install`	

13	Build speech tools and Festival that are located in the local SmartBody directory:
	
	`cd lib/festival/speech_tools`
	`chmod +x configure`
	`./configure`
	`make clean`
	`make`
	`make install`
	`cd ../festival`
	`chmod +x configure`
	`./configure`
	`make clean`
	`make`
	`make install`	

14	Make and install SmartBody:
	
	First, generate the Makefiles with cmake:
	`mkdir buildfolder`
	`cmake ..`
	
	Go to the build directory, make and install the applications
	`cd buildfolder`
	`make install`	

Troubleshooting

A common problem when building the OSx version is confusion between the various versions of Python that are included as part of the OSx operating system. SmartBody currently uses the Python version 2.6. To determine which dynamic libraries are associated with the sbm-fltk executable you can run the otool command:

```
otool -L sbm-fltk
```

In the associated output, make sure that the Python 2.6 libraries are associated like this:

```
/System/Library/Frameworks/Python.framework/Versions/2.6/Python (compatibility version 2.6.0, current version 2.6.1)
```

Building SmartBody for Android

The SmartBody code is cross-compiled for the Android platform using the native development kit (NDK), which allows you to use gcc-like tools to build Android applications. This means that SmartBody is nearly fully-functional as a mobile application, since it uses the same code base as the desktop version of SmartBody. Many of the supporting libraries (such as ActiveMQ, boost, Xerces, Python, Festival, etc.) have already been built as static libraries and exist in the smartbody/android/lib directory.

Note that there are three different examples of Android applications using SmartBody that can be built:sbmjni, sbm-ogre, and vh-wrapper.

If your target hardware is the ARM architecture, some dependency libraries are already prebuilt at SmartBodyDir)/android/lib. Therefore you can build the SmartBody projects directly as below:

	Build Instructions for Android

1	Download and install android-sdk from : http://developer.android.com/sdk/index.html
2	Download and install the android-ndk from: http://developer.android.com/sdk/ndk/index.html (Note that when building on the windows platform, you will also need to install cygwin 1.7 or higher from http://www.cygwin.com/) Follow the installation instruction for both the SDK and the NDK. Be sure to set the correct path to android-sdk/bin, android-ndk/bin so the toolchain can be access correctly. For NDK, you also need to export the environment variable NDK_ROOT to the NDK installation directory (for example, export NDK_ROOT= "/path/to/ndk/directory")
3	Install Eclipse (http://www.eclipse.org/) and its Android ADT plug-in (http://developer.android.com/sdk/eclipse-adt.html) The supporting libraries have been built using Android version 2.3.3 or higher.
4	(Optional) Build sbm-jni sbmjni is a hello world project for SmartBody. It has very basic rendering and minimal functionality, but it helps demonstarte the SmartBody port on android. i) Go to (SmartBodyDir)/android/sbm-jni/ ii) ndk-build (Similar to gcc, you can set the option -j $number_threads to accelerate the build process with multi-threading). iii) copy the directory **(SmartBodyDir)/android/sbm-jni/sbmjniData** and its files to sdcard directory of your android device. (Usually under /sdcard/) iv) Use Eclipse to open the project (SmartBodyDir)/android/sbm/. v) Select Project->Build Project. Connect the device and then run the program as "Android Application".

5	(Optional) Build sbm-ogre sbm-ogre combines SmartBody and ogre for high quality rendering. Currently, it is very slow when rendering in deformable model mode. a. Go to (SmartBodyDir)/android/sbm-ogre/ b. ndk-build (Similar to gcc, you can set the option -j $number_threads to accelerate the build process with multi-threading. c. copy the directory **(SmartBodyDir)/android/sbm-ogre/SbmOgre** and its files to sdcard directory of your android device. (Usually under /sdcard/) d. Use Eclipse to open the project (SmartBodyDir)/android/sbm-ogre/. e. Select Project->Build Project. ThConnect the device and then run the program as "Android Application".
6	(Optional) Build vh-wrapper vh-wrapper is a SmartBody interface to Unity3D. Note that SmartBody connects to Unity via Unity's native code plugin interface, which presently requires a Unity Pro license. In addition, the Unity project needs to be compiled for Android, so a Unity Android license is needed as well. a. Go to SmartBody/android/vh_wrapper/ b. ndk-build c. rename libvhwrapper.so to libvhwrapper-dll.so (for some reason, Android does not accept a build target name with "-") d. copy libvhwrapper-dll.so to the plug-in directory of Unity project. Build the Unity project for Android.

If you are targeting other hardware achitecture (x86, etc) or you prefer to rebuild all libraries from their sources, you will need to perform the following steps:

	Building Supporting Libraries

1	Building Boost for Android: Download BOOST library, extract it into SmartBody/lib modify libs\filesystem\v2\src\v2_operations.cpp, change: # if !defined(__APPLE__) && !defined(__OpenBSD__) # include <sys/statvfs.h> # define BOOST_STATVFS statvfs # define BOOST_STATVFS_F_FRSIZE vfs.f_frsize # else #ifdef __OpenBSD__ # include <sys/param.h> #endif to: # if !defined(__APPLE__) && !defined(__OpenBSD__) && !defined(__ANDROID__) # include <sys/statvfs.h> # define BOOST_STATVFS statvfs # define BOOST_STATVFS_F_FRSIZE vfs.f_frsize # else #ifdef __OpenBSD__ # include <sys/param.h> #elif defined(__ANDROID__) # include <sys/vfs.h> #endif modifiy the file SmartBody/android/boost/userconfig.jam, look for : ANDROID_NDK = ../android-ndk ; and change the directory "../android-ndk" so it points to the android NDK directory You may also need to change all arm-linux-androideabi-xxx to the corresponding toolchain name based on your target architecture and platform. (use Cygwin in Windows platform) ./bootstrap.sh ./bjam --without-python --without-math --without-mpi --without- iostreams toolset=gcc-android4.4.3 link=static runtime-link=static target-os=linux --stagedir=android stage

2	Building iconv TODO
3	Building xerces TODO
4	Building clapack TODO

Building SmartBody for iOS

The iOS build requires two steps: building libraries via the command console, then building applications and libraries via XCode.

	Compiling using console (You can find prebuilt version under trunk/ios/libs/iphoneos if you don't want to build from scratch)

1	Cross compiling apr
	• Download apr-1.3.* from http://archive.apache.org/dist/apr/
	• Copy setup-iphoneos.sh and setup-iphonesimulator.sh from trunk/ios/activemq/apr to the folder where you extracted the above downloaded contents.
	• Change the SBROOT inside setup-iphoneos.sh and setup-iphonesimulator.sh to your trunk directory.
	• Make sure you have gcc-4.2 installed on the system. If not, please download the version of xcode 4.02 iOS4.3 which contains gcc-4.2 and install it (make sure you change the installation folder to something other than /Developer, make it something like /Developer-4.0, uncheck update system environment options). You can download previous xcode4.02 from https://developer.apple.com/downloads/index.action , search for xcode 4.02 iOS4.3. P.S. Latest xcode 4.2 and iOS5.0 is using LLVM GCC compiler which has trouble building apr, apr-util and activemq, not sure if it can compile the other third party library, for now just stick on gcc4.2.
	• If, in the previous step, you had to download a different version of xcode, make sure in both the scripts, you change the following variable from -
	  • export DEVROOT=/Developer/Platforms/iPhoneOS.platform/Developer to
	  • export DEVROOT=/Developer-4.0/Platforms/iPhoneOS.platform/Developer (if you installed the xcode in /Developer-4.0)
	• Run setup-iphoneos.sh or/and setup-iphonesimulator.sh
	• Go to trunk/ios/activemq/apr/iphone*/include/apr-l
	  • edit line 79 of apr_general.h to be: -
	  • #if defined(CRAY) || (defined(__arm) && !(defined(LINUX) || defined(__APPLE__)))

2	Cross compiling apr-util
	• Download apr-util-1.3.* from http://archive.apache.org/dist/apr/
	• Copy setup-iphoneos.sh and setup-iphonesimulator.sh from trunk/ios/activemq/apr-util to the folder where you extracted the above downloaded contents.
	• Change the SBROOT inside setup-iphoneos.sh and setup-iphonesimulator.sh to your trunk directory
	• If in the step 1, you had to download the gcc-4.2 (from xcode), change the DEVROOT variable in the scripts as in the previous step from -
	  • export DEVROOT=/Developer/Platforms/iPhoneOS.platform/Developer to
	  • export DEVROOT=/Developer-4.0/Platforms/iPhoneOS.platform/Developer (if you installed the xcode in /Developer-4.0)
	• Run both of the scripts

3	Cross compiling activemq-cpp-library
	• Download activemq-cpp-library-3.4.0 from http://apache.osuosl.org/activemq/activemq-cpp/source/
	• Copy setup-iphoneos.sh and setup-iphonesimulator.sh from trunk/ios/activemq/activemq-cpp to the folder where you extracted the above downloaded contents.
	• Change src/main/decaf/lang/system.cpp line 471 inside activemq folder (above mentioned folder) from "#if defined (__APPLE__)" to "#if 0"
	• Change the SBROOT inside setup-iphoneos.sh and setup-iphonesimulator.sh to your trunk directory
	• If in the step 1, you had to download the gcc-4.2 (from xcode), change the DEVROOT variable in the scripts as in the previous step from -
	  • export DEVROOT=/Developer/Platforms/iPhoneOS.platform/Developer to
	  • export DEVROOT=/Developer-4.0/Platforms/iPhoneOS.platform/Developer (if you installed the xcode in /Developer-4.0)
	• Run both of the scripts
	Note: for smartbody iphone running on unity, we need to rename variables inside activemq-cpp-library decaf/internal/util/zip/*.c to avoid conflict symbols. If you don't want to do that, you can directly use the one under trunk/ios/activemq/activemq-cpp/libs/activemq-unity

4	Cross compiling xerces-c
	• Download xerces-c-3.1.1.tar.gz from http://xerces.apache.org/xerces-c/download.cgi
	• Copy setup-iphoneos.sh and setup-iphonesimulator.sh from trunk/ios/xerces-c to the folder where you extracted the above downloaded contents.
	• Change the SBROOT inside setup-iphoneos.sh and setup-iphonesimulator.sh to your trunk directory
	• If in the step 1, you had to download the gcc-4.2 (from xcode), change the DEVROOT variable in the scripts as in the previous step from -
	  • export DEVROOT=/Developer/Platforms/iPhoneOS.platform/Developer to
	  • export DEVROOT=/Developer-4.0/Platforms/iPhoneOS.platform/Developer (if you installed the xcode in /Developer-4.0)
	• Run both of the scripts

5	Cross compiling ODE
	• Download ode-0.11.1.zip from http://sourceforge.net/projects/opende/files/
	• Copy setup-iphoneos.sh and setup-iphonesimulator.sh from trunk/ios/ode to the folder where you extracted the above downloaded contents
	• Change the SBROOT inside setup-iphoneos.sh and setup-iphonesimulator.sh to your trunk directory
	• If in the step 1, you had to download the gcc-4.2 (from xcode), change the DEVROOT variable in the scripts as in the previous step from -
	  • export DEVROOT=/Developer/Platforms/iPhoneOS.platform/Developer to
	  • export DEVROOT=/Developer-4.0/Platforms/iPhoneOS.platform/Developer (if you installed the xcode in /Developer-4.0)
	• Run both of the scripts

6	Cross compiling clapack (Not required. If you chose not to cross compile, make sure you add Acceleration framework from xcode project)
	• Download clapack-3.2.1-CMAKE.tgz from http://www.netlib.org/clapack/
	• Copy toolchain-iphone*.cmake and setup-iphone*.sh from trunk/ios/clapack to the folder where you extracted the above downloaded contents
	• If in the step 1, you had to download the gcc-4.2 (from xcode), change the IPHONE_ROOT variable in the toolchain-iphone*.cmake from -
	  • /Developer/Platforms/iPhoneOS.platform/Developer to
	  • /Developer-4.0/Platforms/iPhoneOS.platform/Developer (if you installed the xcode in /Developer-4.0)
	• To make a Unity compatible build, you need to modify the cmake file. If not used for Unity Iphone, you can skip the following step:
	
	-Go to clapack/CMakeLists.txt, comment out include(CTest) and add_subdirectory(TESTING)
	-Go to clapack/BLAS/CMakeLists.txt, comment out add_subdirectory(TESTING)
	-Go to clapack/F2CLIBS/libf2c/CMakeLists.txt, take out main.c from first SET so it became
	set(MISC
	f77vers.c i77vers.c s_rnge.c abort_.c exit_.c getarg_.c iargc_.c
	getenv_.c signal_.c s_stop.c s_paus.c system_.c cabs.c ctype.c
	derf_.c derfc_.c erf_.c erfc_.c sig_die.c uninit.c)
	-Go to clapack/SRC/CMakeLists.txt, take out ../INSTALL/lsame.c from first SET so it became
	set(ALLAUX maxloc.c ilaenv.c ieeeck.c lsamen.c iparmq.c
	ilaprec.c ilatrans.c ilauplo.c iladiag.c chla_transtype.c
	../INSTALL/ilaver.c) # xerbla.c xerbla_array.c
	• Change the SBROOT inside setup-iphoneos.sh and setup-iphonesimulator.sh and run the scripts.

7	Cross compiling python
	• Go to trunk/ios/python, modify the SBROOT inside setup-iphoneos.sh
	• If in the step 1, you had to download the gcc-4.2 (from xcode), change the DEVROOT variable in the scripts as in the previous step from -
	  • export DEVROOT=/Developer/Platforms/iPhoneOS.platform/Developer to
	  • export DEVROOT=/Developer-4.0/Platforms/iPhoneOS.platform/Developer (if you installed the xcode in /Developer-4.0)
	• Run the script, you may need to add sudo before the script in case the copying step fails (last step is to copy libpython2.6.a out)
	• Execute "chmod +w libpython2.6.a" command in the console to make libpython2.6.a writable.

8	Cross compiling pocket sphinx (Not required) Download from http://www.rajeevan.co.uk/pocketsphinx_in_iphone/. The steps are on the website. P.S. • Since the results are not that good on Unity and I don't have time fully test out, the code is not intergrated into smartbody yet. • When integrating pocketsphinx with Unity, it would have duplicated symbol problem(this might be the reason of bad recognizing result, it's under sphinx/src/util). I already built a library for Unity that can be used directly. • Also for Unity you may need get prime31 iphone plugin AudioRecorder

	Compiling using Xcode4
1	Build bonebus • Open smartbody-iphone.xcworkspace, select the scheme to be bonebus, build

2	Build boost • Download boost_1_44_0.tar.gz from http://www.boost.org/users/history/version_1_44_0.html and unzip to trunk/ios/boost. Make sure the folder name is boost_1_44_0. • Open smartbody-iphone.xcworkspace, select boost_system, boost_filesystem, boost_regex, boost_python, build them seperately. • Download boost_numeric_bindings from http://mathema.tician.de/news.tiker.net/download/software/boost-numeric-bindings/boost-numeric-bindings-20081116.tar.gz and unzip it to trunk/ios/boost, make sure the name is boost_numeric_bindings
3	Build steersuite • Open smartbody-iphone.xcworkspace, select the steerlib, pprAI, build them seperately.
4	Build vhmsg • Open smartbody-iphone.xcworkspace, select scheme vhmsg and build.
5	Build vhcl • Go to trunk/ios/vhcl/vhcl_audio.cpp and vhcl_audio.cpp, comment #if 1, uncomment #if (WIN_BUILD) • Open smartbody-iphone.xcworkspace, select scheme vhcl and build.
6	Build wsp • Open smartbody-iphone.xcworkspace, select scheme wsp and build.
7	Build smartbody-lib • Open smartbody-iphone.xcworkspace, select scheme smartbody-lib and build. P.S. This xcode project needs maintenance by the developer to make sure it incorporates all the file from SmartBody core.

8	Build smartbody-dll • Open smartbody-iphone.xcworkspace, select scheme smartbody-dll and build.
9	Build vhwrapper-dll (Not required, for Unity only) • Open smartbody-iphone.xcworkspace, select scheme vhwrapper-dll and build.

Once those steps have been completed, you can build any of three applications:

- smartbody-openglES - a simple example of using SmartBody with OpenGL
- smartbody-ogre - an example of using SmartBody with the Ogre3D rendering engine
- smartbody-unity - The Unity3D game engine connected to SmartBody.

Make sure that your iOS device is connected and follow any of the three applications below:

	Building smartbody-openglES
1	Build smartbody-openglES Go to trunk/ios/applications/minimal, open smartbody-iphone.xcodeproj, build and run. P.S. Under smartbody-openglES project Frameworks, you should see all the libraries existing. If not, go over previous steps to check if anything is wrong

	Building smartbody-ogre
1	Build ogre iphone • Download OgreSDK: http://www.ogre3d.org/download/sdk • Build the Ogre iPhone libraries as indicated here:http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Building%20From%20Source%20-%20iPhone&redirectpage=Building%20From%20Source%20(for%20iPhone) • Make sure all the libraries exist inside build/lib/Debug

2	Build smartbody ogre application • Go to trunk/ios/applications/ogreIphone, open smartbody-ogre.xcodeproj • go to smartbody-ogre project, set OGRE_SDK_ROOT to your ogreSDK directory, • set OGRE_SRC_ROOT to your ogre source directory. • Select scheme smartbody-ogre, build and run. p.s. • If the program hangs on boost thread function, try rebuild the ogre iphone dependencies boost libs (pthread, date_time), alternative way is to build the whole ogre iOS libraries with Boost symbol turned off which may affect the results. • ogre 1.8 seems to have trouble when building for iphone/ipad, use ogre 1.7.3.It is extremely slow running on armv6 ipod(after testing), and there's something wrong with the texture and shader. So maybe should just run on armv7 iPhone/iPad

	Building smartbody-unity
1	Check out the project • Currently all the Smartboy Unity project for IOS is sitting inside vh repository at https://svn.ict.usc.edu/svn_vh/trunk/lib/vhunity/samples/smartbody_mobile_minimal and https://svn.ict.usc.edu/svn_vh/trunk/lib/vhunity/samples/smartbody_mobile • smartbody mobile minimal includes only smartbody while smartbody mobile can include all the other components like face detection, audio acquisition etc.
2	Build unity project into xcode • Open any scene from Assests e.g. smartbdoySceneOutDoor.unity • Copy the static libraries inside (SmartBody)trunk/ios/libs/iphoneos to (Unity)Assets/Plugins/iOS. Do not include liblapack.a libf2c.a libblas.a, these would cause duplicated symbol problems. Make sure you are using libactivemq-cpp.a for unity which you can get from trunk/ios/activemq-cpp/libs/activemq-cpp-unity. • Make sure your unity is under iOS platform. • Go to build setting, set the ios platform application name, resolution etc and hit build button.

3	Compile xcode project and run • Add Accelerate.framework(which contains the lapack library) into xcode project. • If you are using smartbody mobile minimal scenes, build and run! • If you are using smartbody mobile scenes. You will need to include CoreVideo.framework, CoreGraphic.framework, OpenAL.framework. Also make sure that in the xcode project setting, include the header search path for opencv_device; in the Libraries folder, drag in opencv_divice libraries. After above steps are done, build and run!

SmartBody iOS build maintenance

This piece of documentation is written mainly for the developer.

- smartbody-iphone.xcworkspace needs to change if there's cpp file added or code re-arrangement.
- vhcl_log.cpp is modified and copied over to make ios build work. So if the source changed, this file may need to be changed.
- vhwrapper.h and vhwrapper.cpp are copied from VH svn, they are used to build smartbody unity application. So if these two files got changed outside, they have to be copied over again and modifies maybe needed to make it working.

Appendix 2: Python API for SmartBody

The Python API for SmartBody can be found at the following location:

http://smartbody.ict.usc.edu/HTML/smartbody.html

At any time, an updated set of HTML pages can be generated from SmartBody by running the following commands:

```
from pydoc import *
d = HTMLDoc()
content = d.docmodule(sys.modules["SmartBody"])
import io
f = io.open('./smartbody.html', 'w')
f.write(unicode(content))
f.close()
```

This will write the documentation into a file 'smartbody.html' in the execution directory.