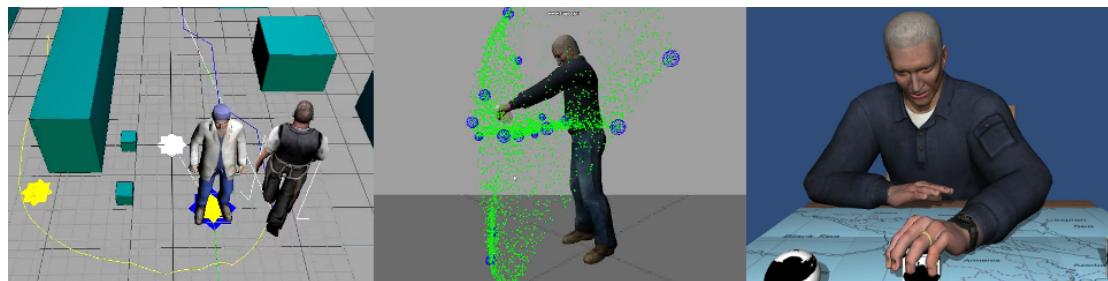


SmartBody



SmartBody is a character animation platform developed at the USC 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 6/9/15

Ari Shapiro, Ph.D.

Research Scientist

Institute for Creative Technologies

University of Southern California

shapiro@ict.usc.edu

Andrew Feng, Ph.D.

Research Associate

Institute for Creative Technologies

University of Southern California

feng@ict.usc.edu

Overview

SmartBody (<http://smartbody.ict.usc.edu>) is a character animation platform developed at the USC Institute for Creative Technologies (<http://www.ict.usc.edu>).

SmartBody provides the following capabilities for a digital 3D 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 and point at objects
- Lip Synchronization and speech - characters can speak with lip-syncing using text-to-speech or prerecorded audio
- Emotion and facial expression
- Gazing - robust gazing behavior that incorporates various parts of the body
- Nonverbal behavior - gesturing, head nodding and shaking, eye saccades
- Animation blending
- Character physics - ragdolls, motion tracking
- Animation constraints - alter character behavior to maintain contact or orientation constraints
- Retargeting - transferring motion from one character to another either offline or in real time.
- Automatic rigging and skinning - automatically adding a skeleton and skin binding information to a humanoid model.

Whereas game engines are focused on rendering, scene management, particle effects and so forth, SmartBody is focused on various behaviors and capabilities of an interactive character. Thus, SmartBody can be thought of as a character animation platform that is compatible with existing game engines.

SmartBody is written in C++ and can be controlled using the Python scripting language or any programming language that can access C++. SmartBody includes its own viewer and renderer and can be run as a standalone system, or incorporated into a game or simulation engines. We currently have interfaces for the following engines:

- Unity
- Ogre
- Irrlicht
- Panda3D

However, it is straightforward to adopt SmartBody to other game engines as well using the C++ or Python API.

SmartBody can be built for the following platforms:

- Windows
- Linux
- OSx

In addition, we have experimental ports to the following mobile and web platforms:

- Android
- iOS

The platform is distributed under the LGPLv3 license. If you would like to incorporate SmartBody into a commercial product and need use of SmartBody outside of LGPLv3 restrictions, please contact us to make such arrangements.

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

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/>

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

Ari Shapiro, Ph.D.

shapiro@ict.usc.edu

Contributors

Development of the SmartBody platform has been a multi-year effort with contributions from many people.

SmartBody Team Lead	Ari Shapiro, Ph.D.
SmartBody Team	Andrew W. Feng, Ph.D.
Inspiration	Stacy Marsella, Ph.D.
Past SmartBody Team Members	Dan Casas, Ph.D. Yuyu Xu Marcus Thiebaux Jingqiao Fu Andrew Marshall Marcelo Kallmann, Ph.D.
SmartBody Support Team and Major Contributors	Ed Fast Arno Hartholt Shridhar Ravikumar Apar Suri Adam Reilly Matt Liewer Teresa Dey Joe Yip
Interns	Sahil Narang Zengrui Wang Tiffany Huang Adil Mohammed Alain Juarez Eder Miguel Yazhou Huang, Ph.D.

Downloading SmartBody

SDK and Binary SmartBody versions

Links to the latest SmartBody SDK can be found on the SmartBody web site:

<http://smartbody.ict.usc.edu/download>

The SDK contains the SmartBody library, the standalone tool that can be used for SmartBody (sbgui), documentation, source code, and code samples. SDK's are available for Windows, Linux and OSX platforms.

The Windows SDK contains binaries, all source code, Visual Studio 2010 projects and prebuilt supporting libraries.

The Linux SDK contains all source code, CMake projects, and instructions for installing supporting libraries via packages on the Ubuntu platform.

The OSX SDK contains binaries, all source code, CMake projects, and prebuilt supporting libraries.

Source Code

The SmartBody source code can be downloaded using Subversion (SVN) on SourceForge. You will need to have a Subversion client installed on your computer. Subversion is a system for source code control. Note that the SmartBody binary will need to be built directly from the source code. If you prefer not to build the code yourself, it is recommended to download one of the SDK versions (above) instead.

Use the SVN command:

```
svn co svn://svn.code.sf.net/p/smartbody/code/trunk smartbody
```

The entire repository is fairly large, several gigabytes in size, including binary libraries for Windows platforms, numerous text-to-speech voices and supporting tools. Note that only the trunk/ needs to be downloaded. Other branches represent older or derivative projects.

Note that the URL of the repository was changed on 3/13/2013. If you downloaded SmartBody from the repository before that date, you can switch your repository to the new URL by going into the SmartBody main directory and running the following SVN command:

```
svn switch --relocate svn://smartbody.svn.sourceforge.net/svnroot/smartbody/trunk https://svn.code.sf.net/p/smartbody/code/trunk
```

You can also do this on Windows using TortoiseSVN by right-clicking on the folder, choose Tortoise SVN -> Relocate... and then enter:

<svn://svn.code.sf.net/p/smartbody/code/trunk>

in the input field.

Windows

We recommend using Tortoise (<http://tortoisessvn.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.

Note that all dependencies to build SmartBody on Windows are contained within the distribution download.

Linux

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

```
sudo apt-get install subversion
```

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

```
svn co svn://svn.code.sf.net/p/smartybody/code/trunk smartybody
```

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

```
svn update
```

Note that all dependencies to build SmartBody on Linux are not contained within the distribution download, although many can be installed via the Linux package system. Separate packages and libraries will need to be installed according to the instructions in [Building SmartBody for Linux](#).

Mac/OSx

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

```
svn co svn://svn.code.sf.net/p/smartybody/code/trunk smartybody
```

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

```
svn update
```

Note that all dependencies to build SmartBody on OSX are not contained within the distribution download. Separate libraries packages will need to be installed according to the instructions in [Building SmartBody for OSx](#).

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](#), [Unity](#) or [Panda3D](#).
3. On an [Android](#) or [iPhone](#) mobile device
4. Embedded as a Python library

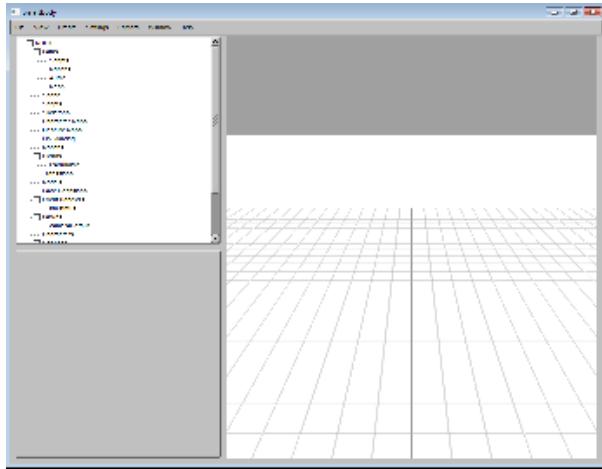
To build SmartBody for your platform, please see the Appendix on [Building SmartBody](#)

Running SmartBody as a Standalone Application

The SmartBody distribution comes with its own application/renderer and toolset called sbgui. This file is located in the /bin directory (if you have downloaded the source code from SVN, it is located in core/smartsbody/sbgui/bin after the sbgui project is built). To run SmartBody within an existing game engine or other framework, please refer to the section on [Integrating SmartBody With a Game Engine](#).

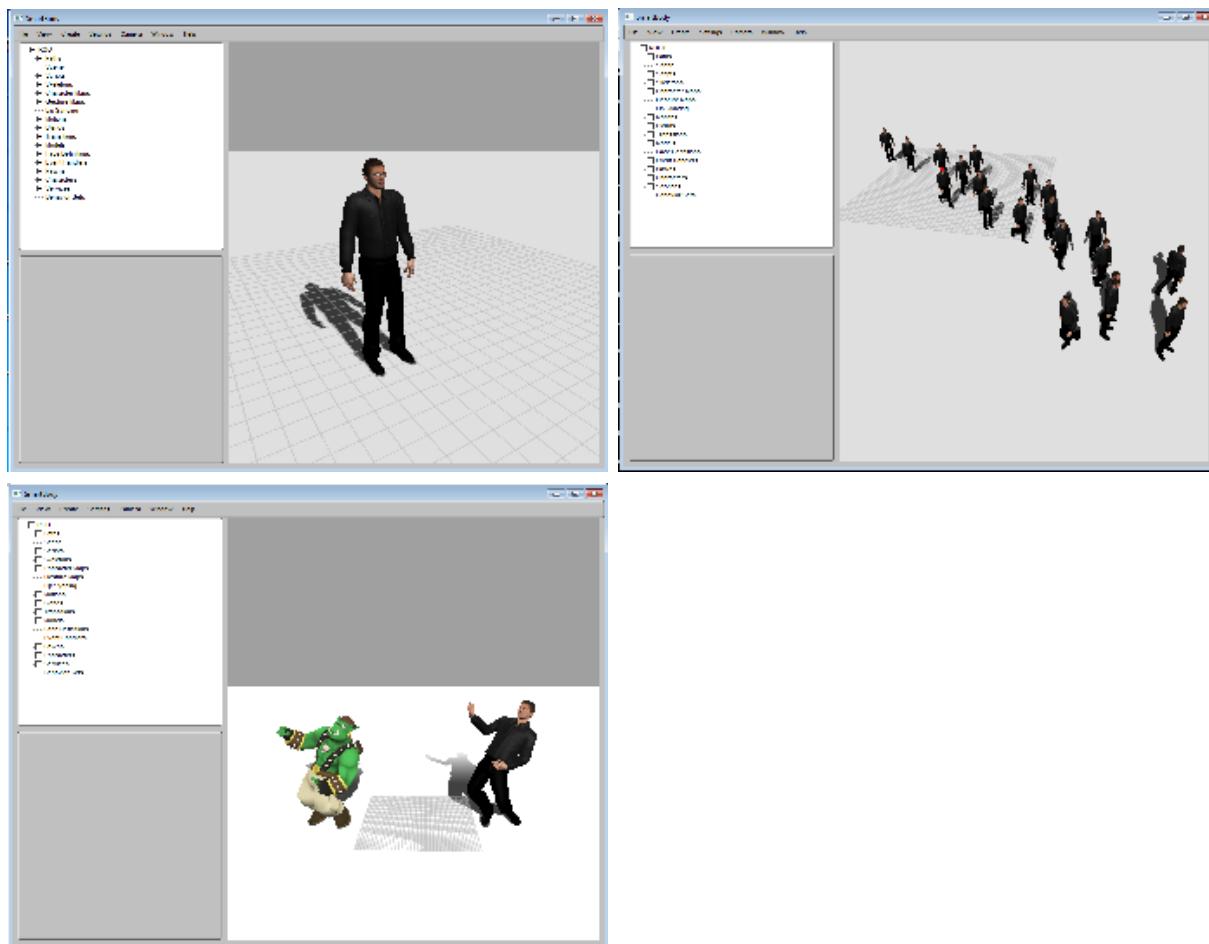
How to Use sbgui

By default, the sbgui application will open showing an empty scene. The right side of the window contains a view of the 3D scene, the left side contains a list of objects in the scene, and the lower left corner contains attributes associated with selected objects.



Quickstart

Choosing "File->Load from script" and then selecting a Python file (.py) will run the contents of that script. There are many example applications located in the examples/ directory that can be tested and experimented with.



(Above) Some example scripts showing characters, crowds and online retargeting.

Sample Scripts

Demo	Description
AddCharacterDemo.py	Example of creating a single male character using one of the default SmartBody characters, including most of SmartBody's capabilities such as speech, facial animation, locomotion, reaching and so forth.
AddCharacterDemoRachel.py	Example of creating a single female character using one of the default SmartBody characters, including most of SmartBody's capabilities such as speech, facial animation, locomotion, reaching and so forth.
BlendDemo.py	Example of using motion blends (combinations of animations that are blended together based on a parameter.)
ConstraintDemo.py	Example of using (positional or rotational) animation constraints in combination with other behaviors.
CrowdDemo.py	Example of crowds with steering and locomotion.
EventDemo.py	Example of how to connect events to animations, then respond to those events via scripting interface.
FacialMovementDemo.py	Example of using the facial animation system.
GazeDemo.py	Example of using the gaze capability.
GesturesDemo.py	Example of how to use the gesturing capability.
HandSynthesisDemo.py	Example of using the hand configuration engine which generates finger motion on top of gestures.
HeadDemo.py	Example of how to control a character's head.
LocomotionDemo.py	Example of using the locomotion system.
OgreCrowdDemo.py	Example of using a crowd of character's with the Ogre Sinbad character.
OgreDemo.py	Example of using the Ogre Sinbad character with a number of capabilities: locomotion, gazing, etc.
OnLineRetargettingDemo.py	Example of how to retarget motion from one character to another in real time.
PerlinNoiseDemo.py	Example of adding noise to motion.
PhysicsDemo.py	Examples of using character physics in various ways (ragdolls, motion tracking, etc.)
ReachDemo.py	Example of reaching/grabbing/touching capability
SaccadeDemo.py	Example of saccadic eye motion

SpeechDemo.py	Example of text-to-speech capability and lip syncing (requires running the speech relay and installation of ActiveMQ service)
SteeringDemo.py	Example of using steering with locomotion, including dynamic and static obstacle avoidance

Using Messaging: The Virtual Human Messaging System

SmartBody can communicate with other applications over a network using a messaging system called the Virtual Human Messaging system (VHMSG). The VHMG system uses an asynchronous transport framework called ActiveMQ, which in turn operates over TCP/IP. Thus, in order to use VHMSG you need to either install an ActiveMQ server or connect to an existing one. ActiveMQ is free and can be downloaded from <http://activemq.apache.org/>. Note that SmartBody will function properly without an ActiveMQ server installed. However, some functionality (such as real-time text-to-speech) requires asynchronous communication over the VHMSG bus.

Messages are typically sent by specifying the component that

Running sbgui on the Command Line

The following command line arguments can be used with sbgui:

Argument	Description
-scriptpath	Path containing scripts to be initially loaded
-script	Initial script to run. If not specified, runs default.py. Note that if a script is specified, SmartBody will run in headless mode (no user interface). To run the user interface, add the following Python commands to the top of the script: <pre>getViewer().show() # shows the user interface window</pre> <pre>getCamera().reset() # resets the camera to the default location</pre>
-mediapath=	Media path for the application. The media path determines the prefix by which all subsequent paths are added. For example, if the media path is: /my/application, then specifying a relative directory for some file parameter, such as 'myfile', results in the mediapath being added to it. In other words, the final path will be: /my/application/myfile
-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=	identifier of SmartBody instance. When running multiple SmartBody instances simultaneously, this distinguishes one from another.

The sbgui application will read a file called .smartbodysettings which will contain a number of configuration paths for various components that SmartBody uses:

```
mediapath=../data
pythonlibpath=../Python27/Lib
ogrepather=../OgreSDK
speechrelaycommand=start TtsRelayGui.exe
renderer=ogre
```

All paths are relative to the location of the sbgui executable. mediapath indicates the media path. pythonlibpath indicates the location of the Python libraries. ogrepather indicates the location of the Ogre support files. speechrelaycommand indicates the launching instruction for the speech relay. renderer indicates whether to use the built-in renderer (custom) or the Ogre renderer (ogre).

If no arguments are given to sbgui, an empty scene will be shown. To load a scene, choose the File -> Load from Script... menu option. Any Python (.py) file that contains SmartBody commands can be used. By using the File->Load from Script... menu option, anything that currently exists in the SmartBody scene will be removed before the commands in the file are run. Below are some example commands that can be used to run the contents of another file:

```
scene.addAssetPath("script", "data/examples")
scene.run("AddCharacterDemo.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 AddCharacterDemo.py script that will be found in one of the paths already specified.

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 Assets to the Scene

Various assets, such as skeletons, motions and models can be added to the scene. There are several ways to add assets:

1. Use drag-and-drop from the file system to the Resource Viewer (the window that contains the tree showing the objects in the scene on upper-left hand side of the application)
2. Use Python commands to load specific data files or folders.
3. Use drag-and-drop to load a model, skeleton, or character into the 3D scene.

To add an asset via Python commands, use the following syntax:

```
scene.addAssetsFromPath("path/to/my/data")
```

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 in the Attribute Editor (window in the lower left hand 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 + Shift + Right Click	zoom
ALT + Middle Click	pan
ALT + Left Click	rotate

Also, the following menu choices are available:

Menu	Action
Camera->Save Camera View	Saves the current camera view
Camera->Load Camera	Changes the current camera to that of a previously saved view
Camera->Delete Camera	Removes a previously saved camera view
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->Frame Selected Object	Frames the camera on the currently selected object
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->Character Cone Sight	Shows the scene from the perspective of the selected character.

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.
Camera->Rotate Around Selected	Rotates the camera around the selected object
Camera->Modes->Default	Camera stays fixed until explicitly moved
Camera->Modes->Free look	
Camera->Modes->Follow renderer	When used as a debugger of another running SmartBody instance, track the position of the camera in the other instance
Camera->Take snapshot->....	Starts the recording of a screen to a specified folder. Select again to stop recording. Useful for recording animated sequences.

Selecting and Positioning Characters

A character or pawn can be selected by either 1) left-clicking on the character in the 3D scene, or 2) Choosing the character's name under the Characters list in the Resource Viewer. 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). The manipulator either shows a bounding box, a translation widget, or a rotation widget:

Manipulator Mode	Key
Selection	q
Translation	w
Rotation	e

A character or pawn can be manually positioned by selecting one of the manipulator handles via a Left Click and then dragging the manipulator handles character to the new location. To rotate the character or pawn, select the object, press 'e', then adjust 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, if they have the locomotion behavior.

Changing Lights

The default scene automatically places two directional lights, one which is rotated 135 degrees along the z-axis, and one rotated negative 135 degrees around the z-axis. The purpose of these automatic lights is to provide some initial lighting of a scene.

If you want to create and position your own lights in the scene, sbgui considers any pawns that start with the name 'light' (such as light0, light1, etc.) to be a light. Lights can be created either via scripting, or through the Create->Light menu option. Once a light is added to the scene, the default lights will no longer be present.

Attribute	Description
enabled	boolean. Whether the light is enabled or disabled. Default is True
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
lightCastShadow	boolean. Whether this light casts a shadow. Default is true
lightShadowMapSize	integer. Size of the light's shadow map. Default is 1024

Other Controls

File->New	Resets SmartBody to run a new simulation. All objects will be removed.
File->Load from script	Loads a SmartBody scene from a Python (.py) file. The existing scene will be reset.
File->Save to script	Saves the SmartBody configuration to a .py file. Note that the state of the simulation (including behaviors) will not be saved, but the setup information (characters, pawns, positions, orientations, asset paths, blends, face definitions, joint maps, lip sync information, camera positions) will be saved.
File->Run Script...	Runs the contents of a Python script. The scene will not be reset (in contrast to the 'Load from script' choice) and commands from the script file will be added to the scene.
Export Scene	Saves out particular scene elements (such as characters, pawns, joint maps, etc.) to a .py file
Import folder	Loads the contents of a previously saved folder, running the initScene.py file.
File->Remote Quick Connect	Connects to the first found another running instance of SmartBody.
File->Remote 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->Remote Disconnect	Disconnect from a remote running instance of SmartBody.
File->Quit	Exit the sbgui application
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->Skin Weight	Visualizes the smooth binding information from the mesh to the skeleton
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->Gaze Limits	Shows the gazing joint limitations for a selected character
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->Character->Show Gesture	Shows the gesture phases for a selected gesture
View->Character->Show Joint Labels	Shows the name of the joints on a selected character
View->Show Selected Object	Shows or hides the bounding box information around the selected object
View->Pawns	Toggles pawn display on and off
View->Show Cameras	Shows or hides cameras in the scene
View->Show Lights	Shows or hides lights in the scene
View->Shadows->Shadow Map	Toggles shadows using shadow maps on and off
View->Shadows->Stencil Shadow	Toggles shadows using stencil shadows on and off
View->Shadows->Disable Shadow	Disables shadows.
View->Grid	Toggles the grid on and off
View->Background Color	Change the background color of the scene
View->Floor->Show Floor	Show the floor, which can be used for drawing shadows
View->Floor->Floor Color	Change the color of the backpack
View->Terrain->Shaded	Show the terrain in shaded mode
View->Terrain->Wireframe	Show the terrain in wireframe mode
View->Terrain->No Terrain	Hide any terrain
View->Navigation Mesh->No Mesh	Hide any navigation meshes in the scene
View->Navigation Mesh->Show RawMesh	Show the navigation mesh in its raw form
View->Navigation Mesh->Show NaviMesh	Show the navigation mesh
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->Pawn from model...	Creates a pawn and assigns the selected model/mesh to that pawn.
Create->Light	Creates a light. Lights are pawns with the name 'lightN' where 'N' is the light number.
Create->Camera...	Creates a camera in the scene
Create->Terrain	Loads a terrain as a heightfield from a .ppm file
Create->Delete Object	Removes an object from the scene.
Create->Delete Selected Object	Removes the selected object from the scene.
Settings->Set Resolution->Default	Returns the scene into the default resolution
Settings->Default Media Path	Sets the default media path which persists between scenes
Settings->Internal Audio	Toggle SmartBody's ability to respond to sound playing commands.
Window->....	Launch the various windows, detailed below
Help->About	Shows information about the current build of SmartBody, including release number.
Help->Documentation	Launch a web player to see the documentation online
Help->Create Python API	Saves the Python API to an HTML file.

Command Window

The Command Window allows you to write commands, as well as examine messages output from SmartBody. The 'Python' tab is used to send commands via the Python scripts. The 'Sbm' tab is used to send commands via the old-style (deprecated) command system.

The screenshot shows the SmartBody Commands window. The title bar says "Commands". The menu bar has "Edit" selected. The main area contains the following Python code:

```
+ 105: l_hip (Quat)
+ 106: l_knee (Quat)
+ 107: l_ankle (Quat)
+ 108: l_forefoot (Quat)
+ 109: l_toe (Quat)
+ 110: r_hip (Quat)
+ 111: r_knee (Quat)
+ 112: r_ankle (Quat)
+ 113: r_forefoot (Quat)
+ 114: r_toe (Quat)
~ position = scene.getCharacter("doctor").getPosition()
print str(position.getData(0)) + " " + str(position.getData(1)) + " " + str(position.
35.0 102.0 0.0
```

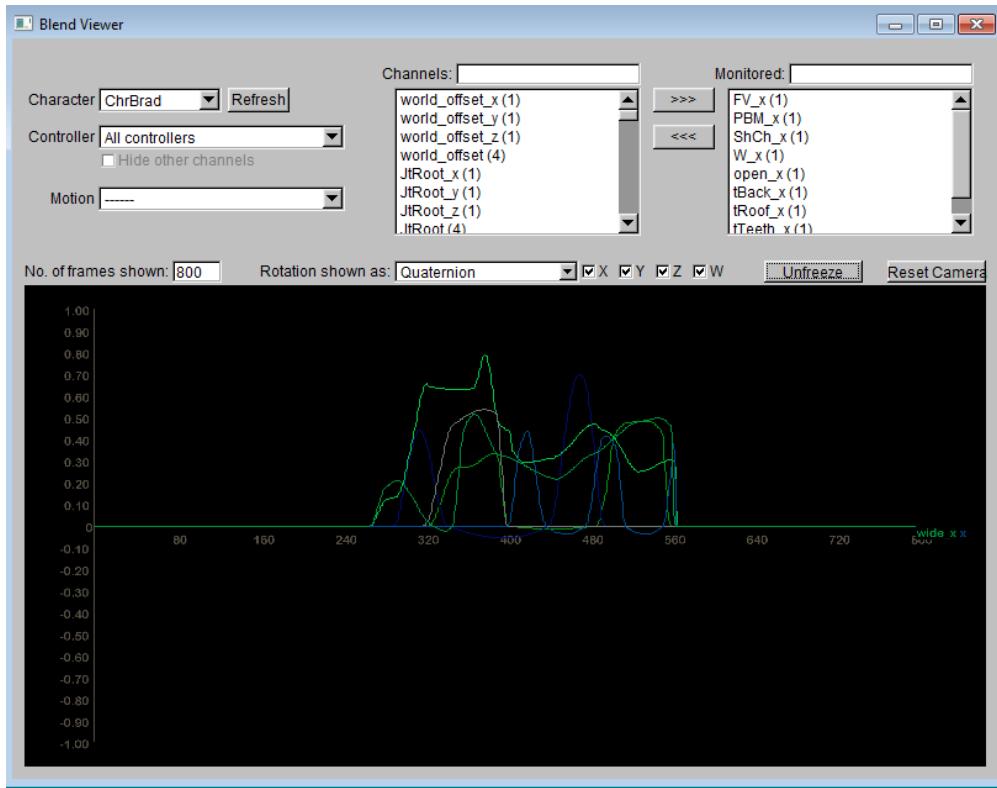
Below the code are scroll bars. At the bottom right is a "Run" button. Below the run button is a toolbar with tabs for "Sbm" and "Python", with "Python" currently selected.

A second code editor window is partially visible below the first. It also contains Python code:

```
position = scene.getCharacter("doctor").getPosition()
print str(position.getData(0)) + " " + str(position.getData(1)) + " " + str()
```

Data Viewer

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



The Character dropdown allows you to select a particular character, then the Channels for that character can be moved to the Monitored list and its value will scroll across the viewer. ALT+Right click will zoom in and out of the viewer, and the Reset Camera button will restore the original view. Selecting the channel from the Monitored list will highlight the selected channel.

The Controller dropdown allows you to see the value of the Channel as it passes through the stack of controllers. By default, the 'All controllers' option shows the data as it is applied to the character. Thus, the influence of the controller on various channels can be seen when switching between adjacent controllers.

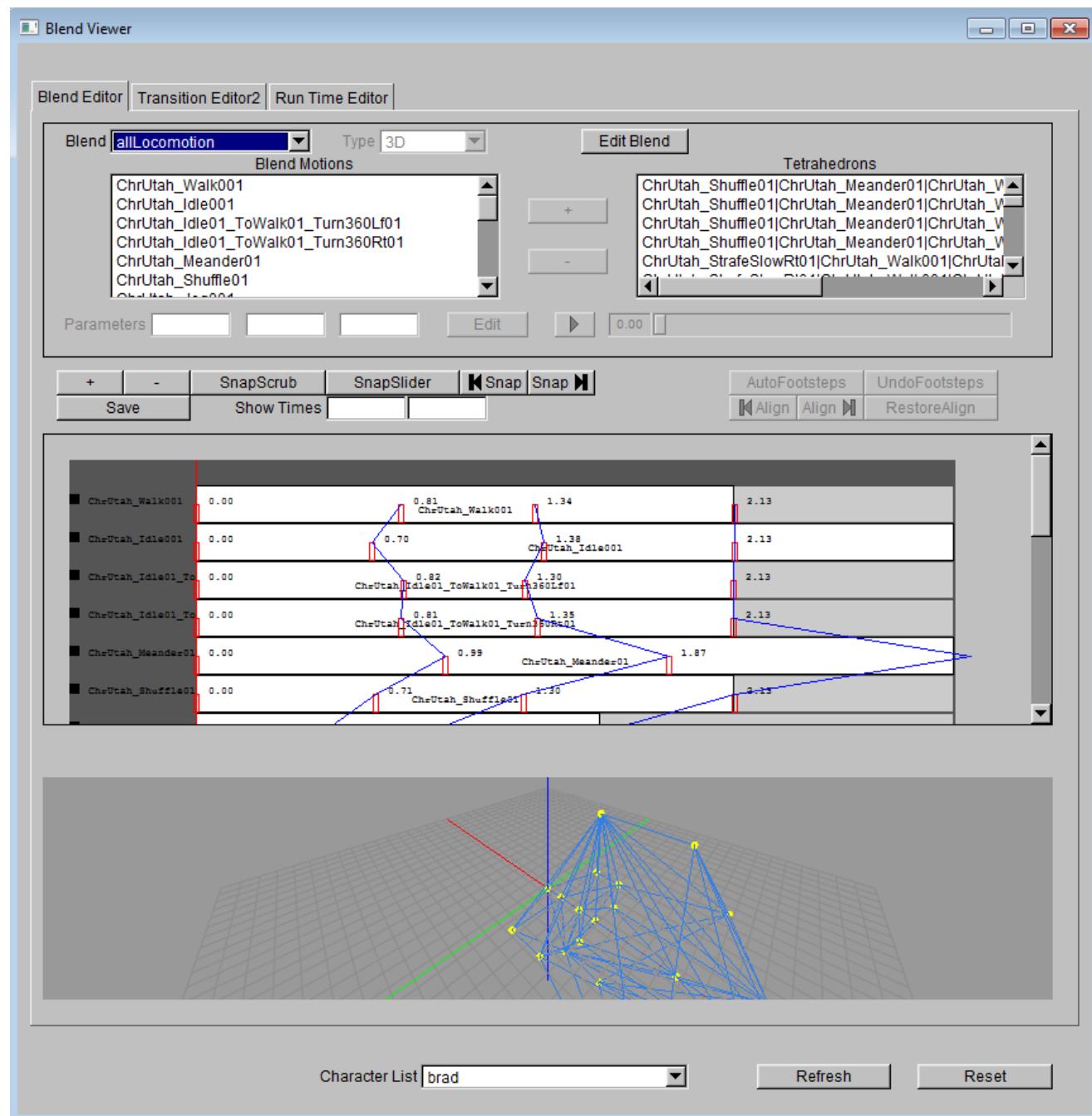
The Motion dropdown allows you to see the value of the Channel as it passes through the stack of controllers, independent of the character.

Blend Viewer

The Blend Viewer allows you to create, edit and test blends and transitions between blends. Blends are groups of animations that can be parameterized by a value. For example, a locomotion blend could contain several different types of movement (walking, jogging, running) with each animation associated with a speed. Animations organized into a blend are associated with each other through correspondance points (points among the motions that are similar, such as timing of foot strikes on the ground).

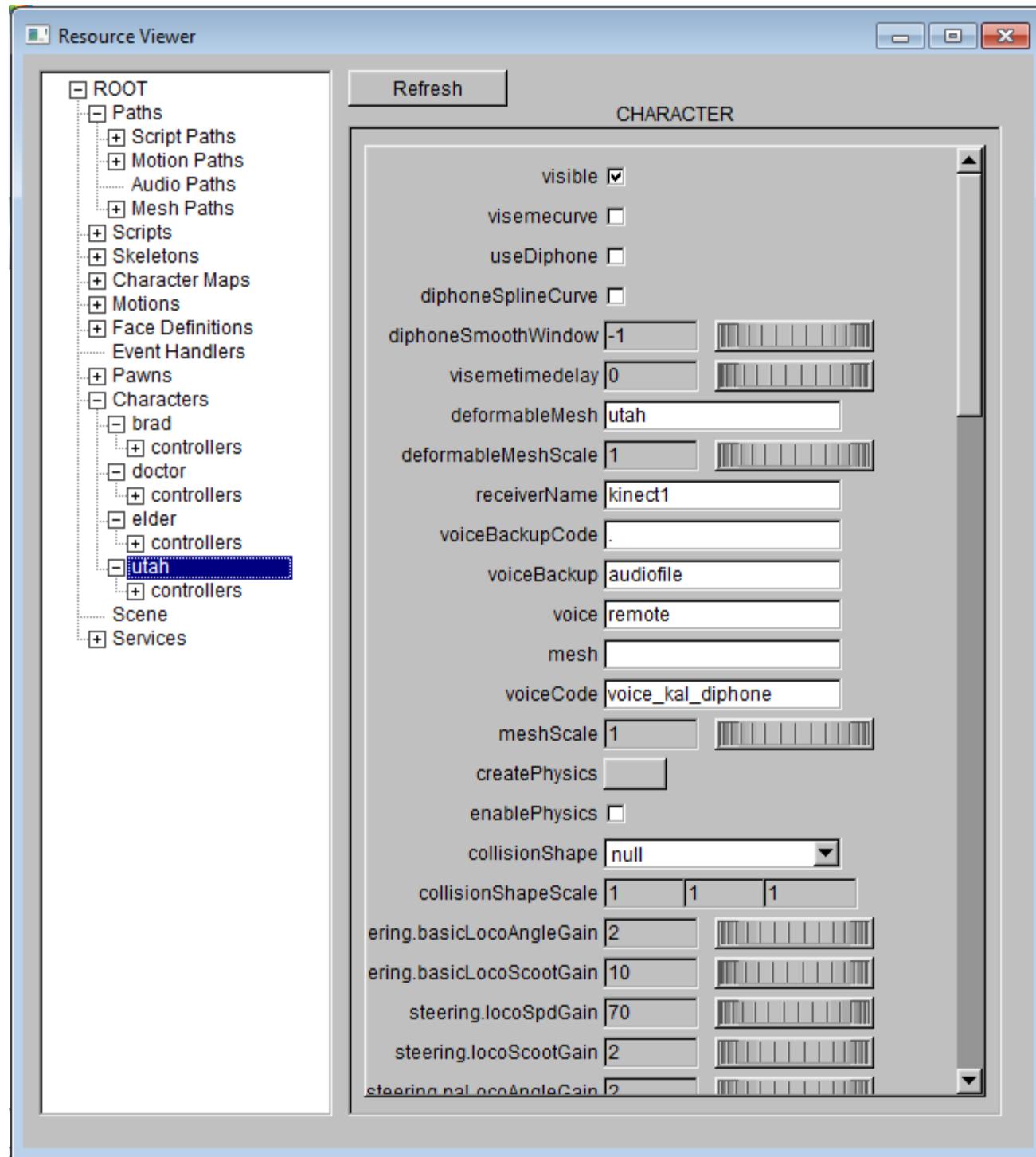
Details on how to use the Blend Viewer can be found in the section on [Configuring Blends and Transitions](#).

D



Resource Viewer

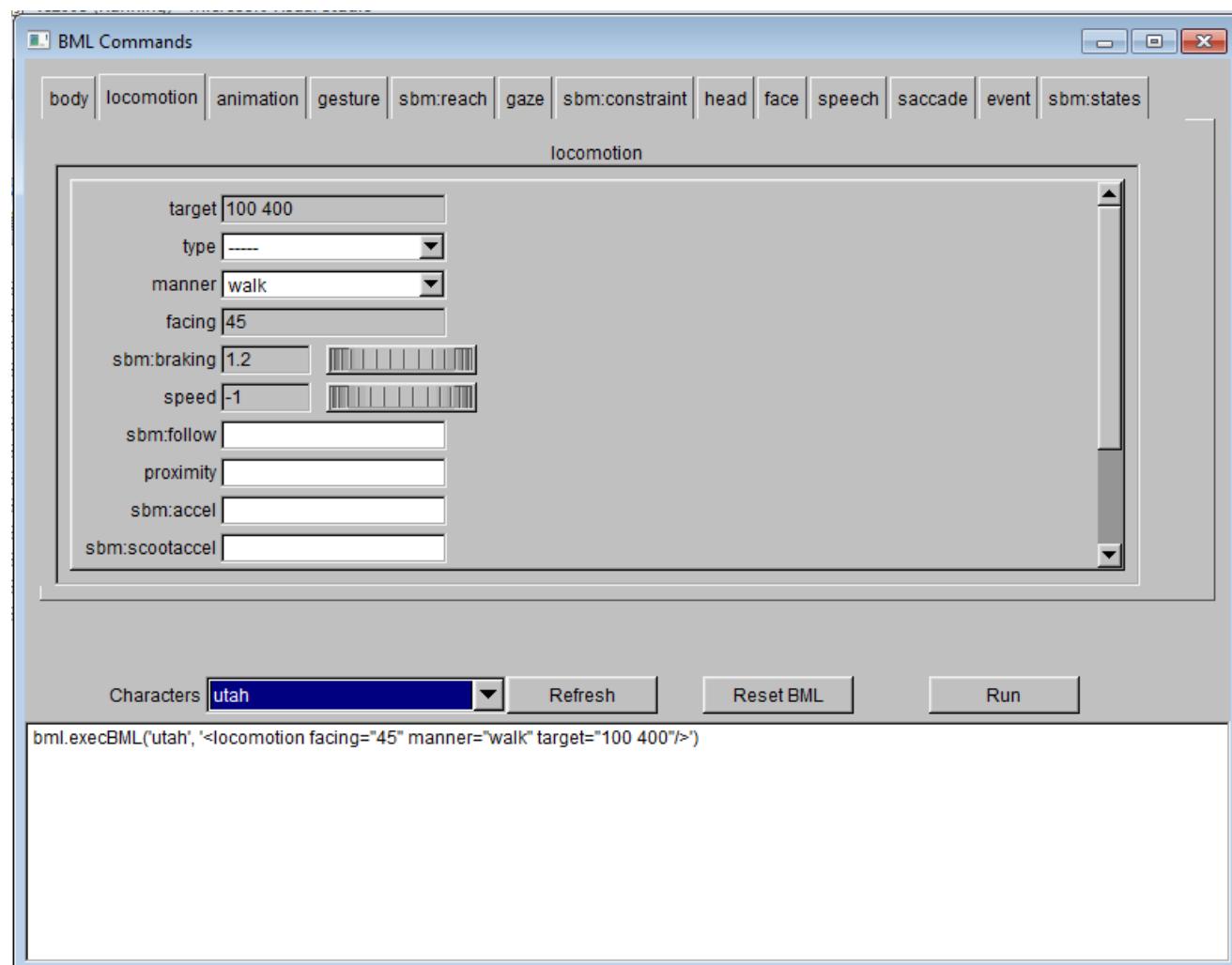
The Resource Viewer allows you to examine the assets loaded into the system (skeletons, animations) as well as the characters, pawns, and services being run. All run time structures, such as joint mappings, face definitions, controllers and scripts can be examined or triggered from this window. In addition, there is an interactive interface for examining and setting attributes. Attributes are dynamic data associated with SmartBody objects. Changing the values of the attributes in the Resource Viewer is equivalent to changing the attributes via a Python command and has the same effect.



BML Creator

The BML Creator Window allows you to create Behavior Markup Language commands easily. The interface shows the different BML commands available as tabs, 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.

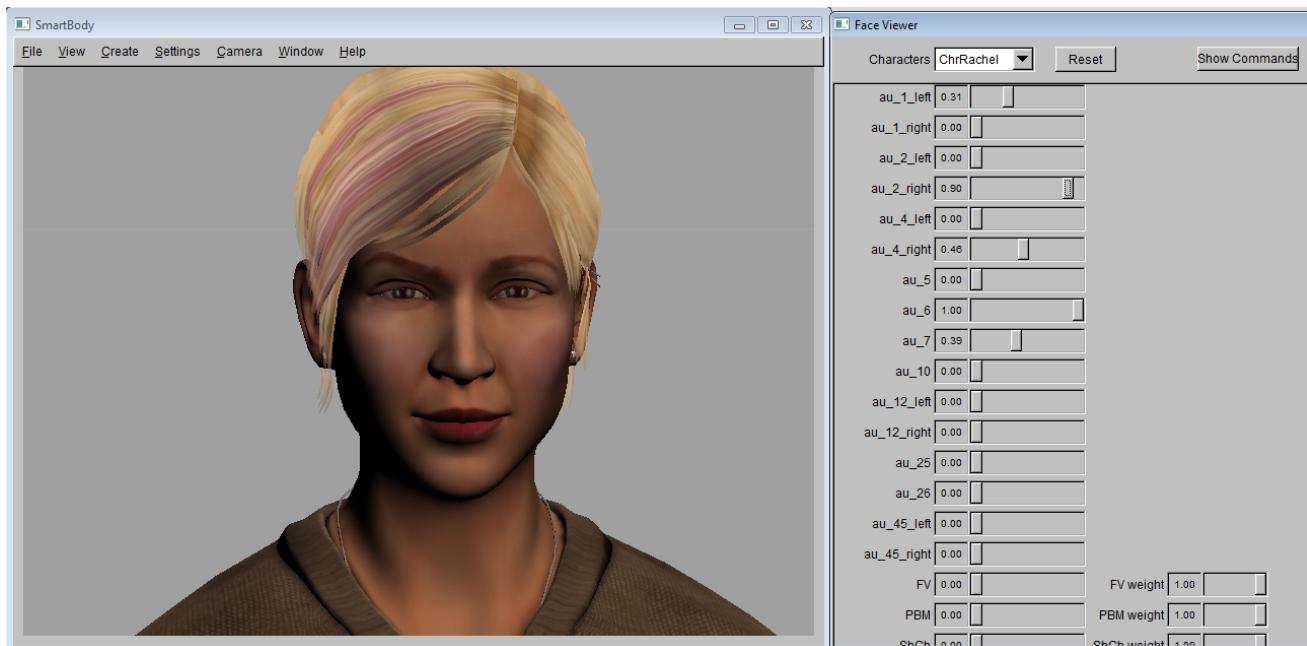
As you select the command option, an Python command will be generated that performs the same command, which can be copied and used in a script for later use. The 'Run' button will execute that BML command on the character selected in the 'Characters' dropdown, or '*' for all the characters.



Face Viewer

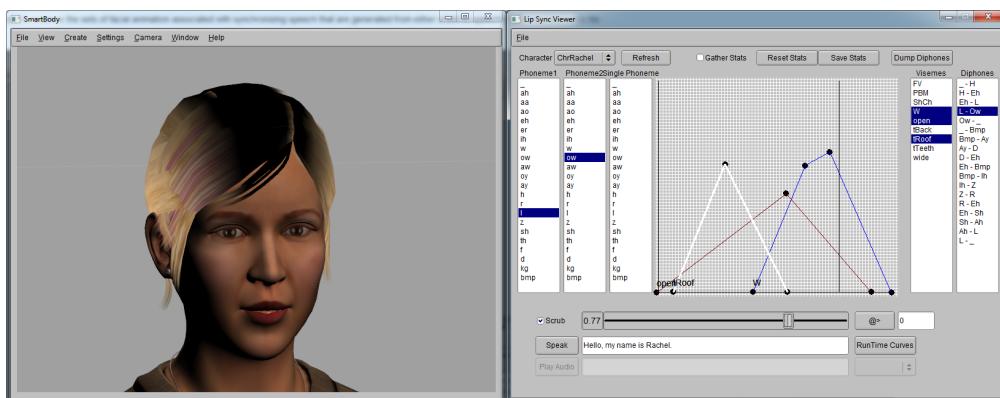
The Face Viewer allows you to interactively control the facial animation of a SmartBody character. The controls that appear are based on a character's Face Definition.

The controls consist of Facial Action Units (FACS), which control various aspects of the face, as well as visemes, which are associated with lip syncing. The visemes can be modulated by changing their associated weights (0 means the viseme will have no effect, 1 means the viseme will be in full effect).

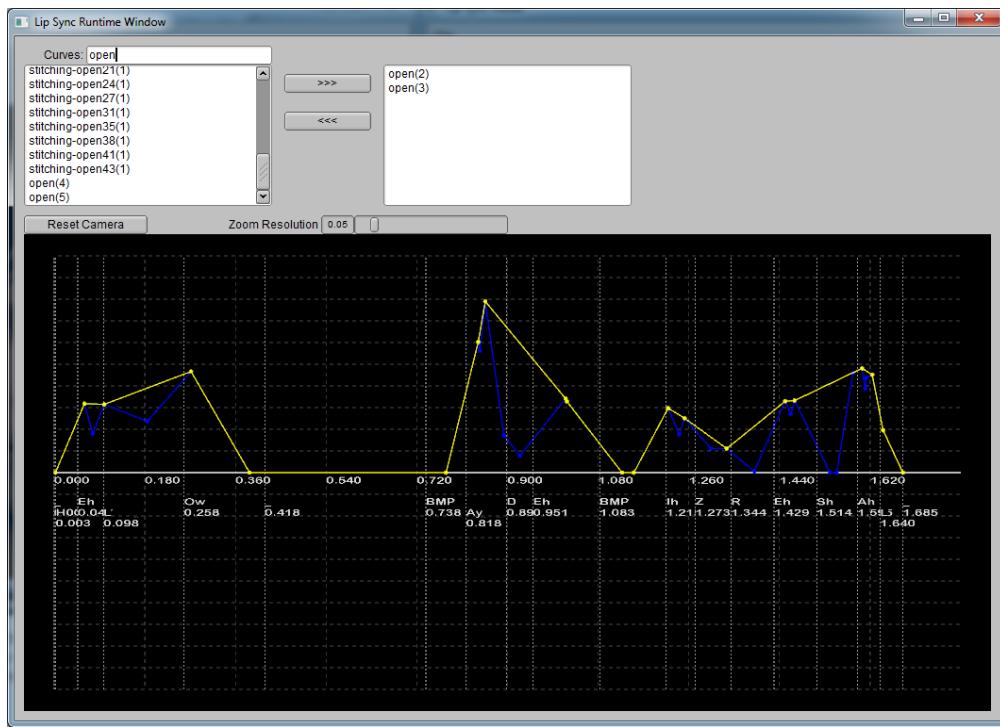


Lip Sync Viewer

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



The curves show the facial animation associated with an adjacent set of phonemes, and the facial poses that are used to generate that motion. In the example above, the curves are activating the W, open and tRoof shapes in order to generate motion for the phonemes L-Ow (like you would say the word 'Low'). The phoneme pairs used for the utterance (in this case, "Hello, my name is Rachel") are displayed on the far right. In this case, the character is using a text-to-speech (TTS) engine. If the character was using a prerecorded voice, then the list of audio that could be played would be displayed in the dropdown box on the bottom right and the 'Play Audio' button would be enabled. Any adjacent phoneme pairs can be scrubbed to show the motion, as well as edited. The 'Dump Diphones' button will save the modified data to a .py file. Runtime curves will be found out in Lip Sync Runtime Window by clicking "RunTime Curves" Button.

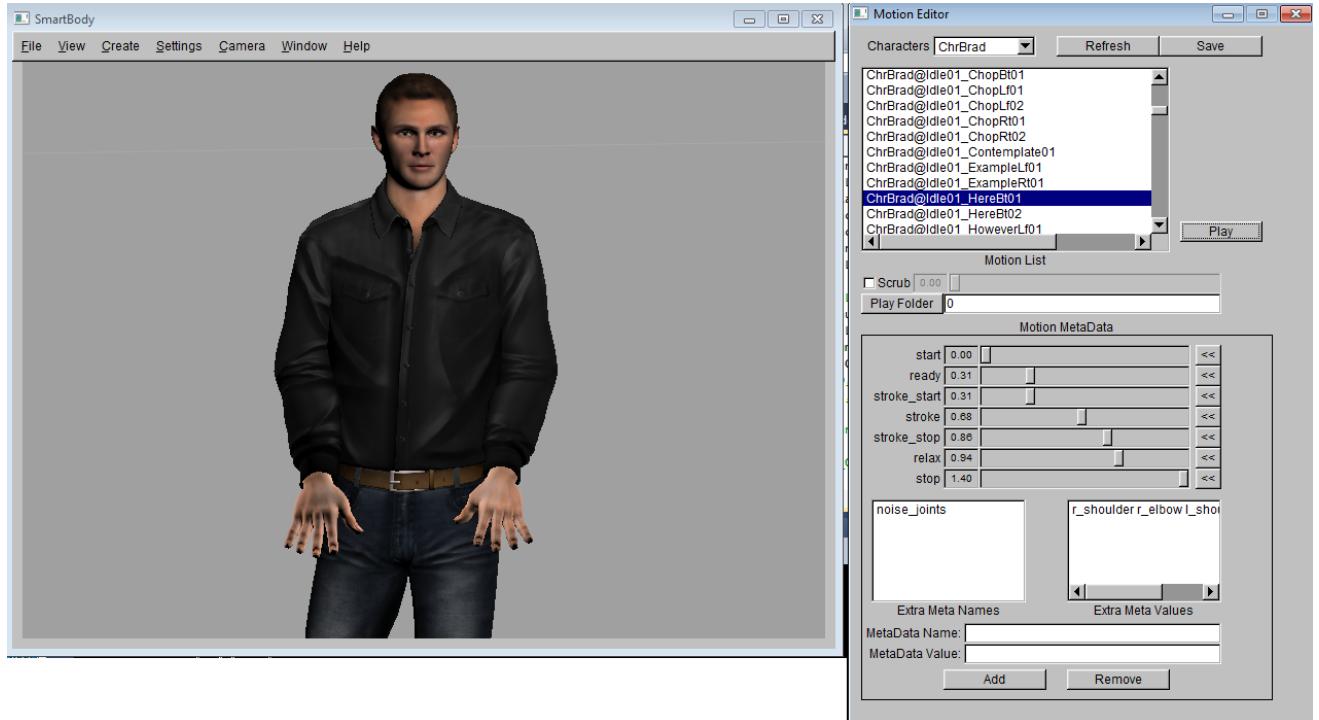


The window above shows the phoneme schedules as well as each operation done to the animation curves. For example, open(2) means it's the output after second operation done to animation curve for facial pose "open". You can zoom in/zoom out to better spectate the curves, also filter function is provided to find the specific curves.

More details about this process can be found on the section [Configuring Facial Animations and Lip Syncing](#).

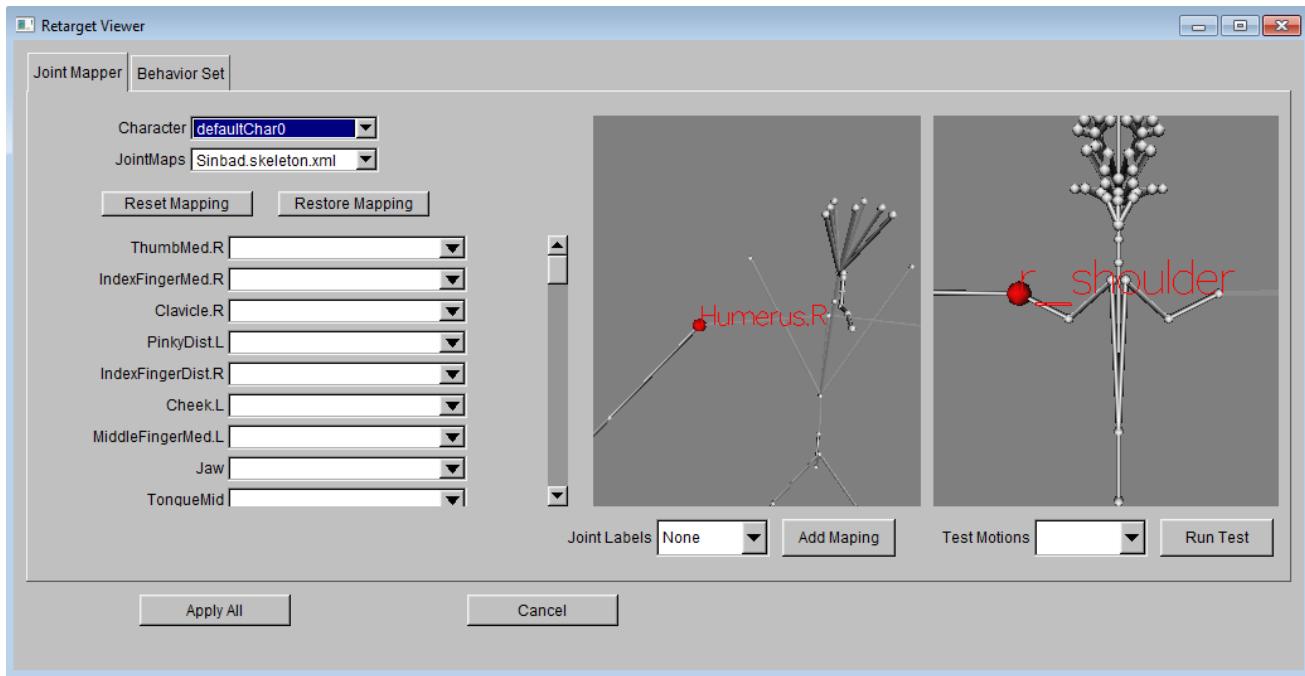
Motion Editor

The Motion Editor Window allows you to play the motions loaded into SmartBody on a character, scrub the motion, and set metadata regarding timing and other data marks on the motion.

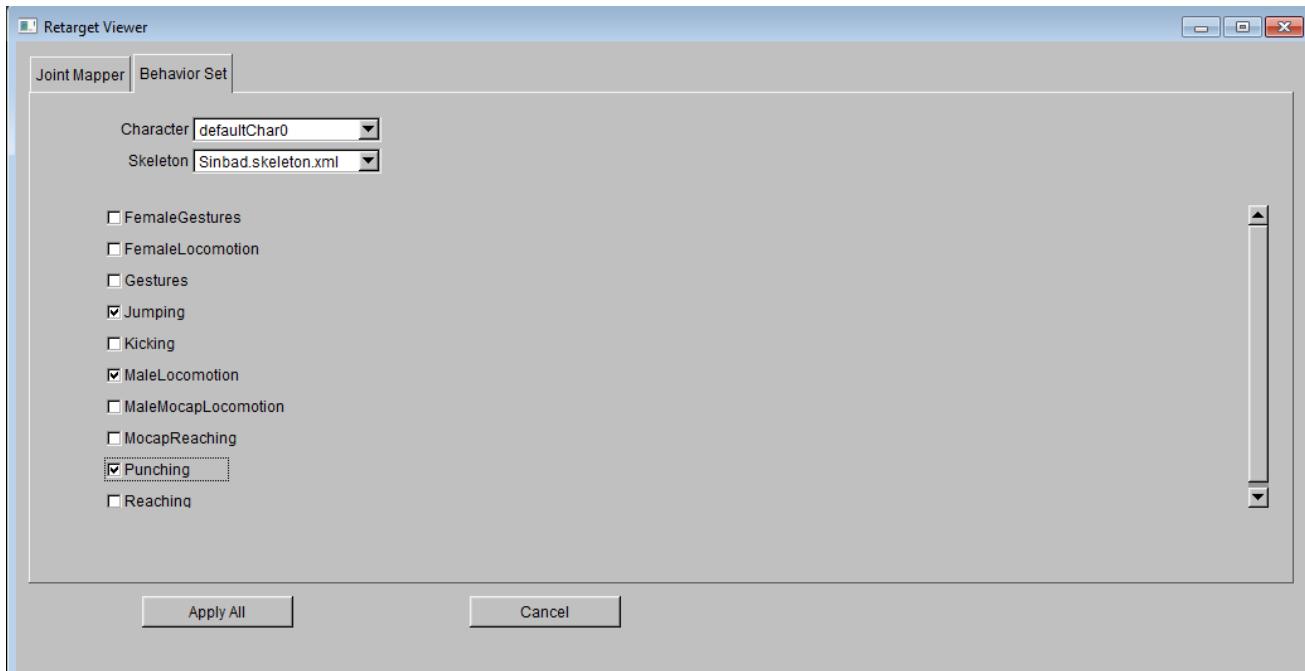


Retarget Viewer

The Retarget Viewer allows you to map a new character's joint and bone hierarchy to the standard SmartBody skeleton so that SmartBody can run the standard set of controllers on the character (such as gazing, head movements, eye saccades and so forth) as well as to prepare the character for retargetting. The standard SmartBody character and its naming appear on the right, and the new character appears on the left. Joints may be individually selected to determine correspondences.



The Behavior Set tab lists a set of related parameterized motions and setup information that can be automatically retargeted and connected to a character once the joint mapping has been completed.



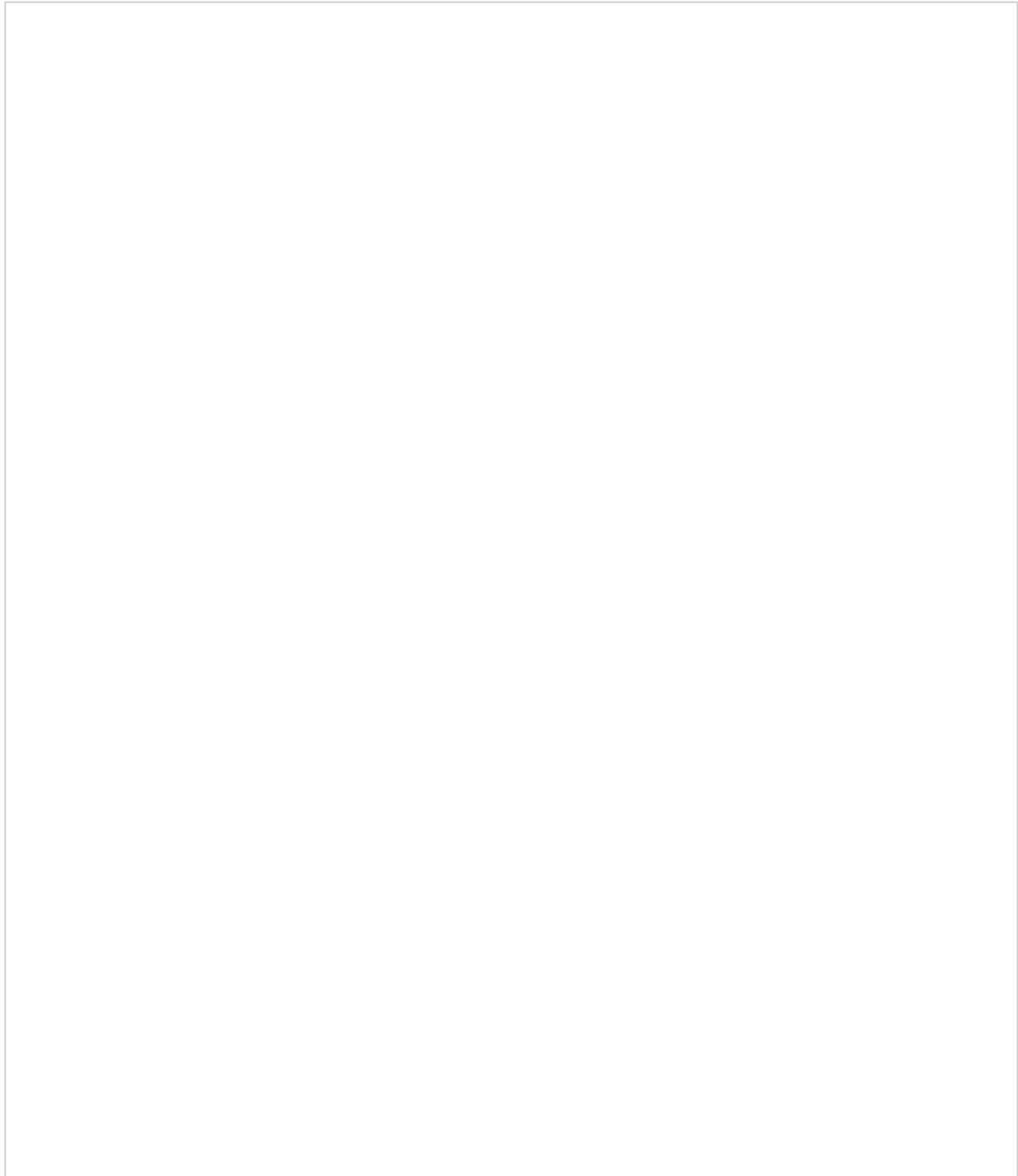
More details of using the Retarget Viewer, Joint Mapping and Behavior Sets can be found in the section on [Motion Retargeting](#).

Controlling the User Interface in sbgui via Python

Mouse and Keyboard Interactions

The GUI for sbgui can be controlled by issuing Python commands that respond to mouse clicks and movements, as well as for keyboard touches.

To respond to a mouse or keyboard touch, you can override the behavior by creating a Python class as follows:



```

import GUIInterface

class MyInterface (GUIInterface.SBInterfaceListener):
    def onStart(self):
        # triggered when the interface is first loaded from the interface manager
    def onMouseClick(self, x, y, button):
        # triggered when a mouse button is clicked
        # x and y are screen pixel coordinates of the mouse location
        # button is the number of the mouse button (1 = left, 2 = middle, 3 = right)
        # return False if you want sbgui to handle this mouse click as it ordinarily does
        # return False if you want sbgui to stop any further handling of the mouse click
        return False

    def onMouseMove(self, x, y):
        # triggered when a mouse button is clicked
        # x and y are screen pixel coordinates
        # return False if you want sbgui to handle this mouse movement as it ordinarily does
        # return False if you want sbgui to stop any further handling of the mouse movement
        return False

    def onMouseRelease(self, x, y, button):
        # triggered when a mouse button is released
        # x and y are screen pixel coordinates of the mouse location
        # button is the number of the mouse button (1 = left, 2 = middle, 3 = right)
        # return False if you want sbgui to handle this mouse release as it ordinarily does
        # return False if you want sbgui to stop any further handling of the mouse release
        return False

    def onMouseDrag(self, x, y):
        # triggered when a mouse is moved while a button is pressed
        # x and y are screen pixel coordinates
        # return False if you want sbgui to handle this mouse drag it ordinarily does
        # return False if you want sbgui to stop any further handling of the mouse drag
        return False

    def onKeyPress(self, c):
        # triggered when a key is pressed
        # c is the character pressed
        # return False if you want sbgui to handle this keyboard press it ordinarily does
        # return False if you want sbgui to stop any further handling of the keyboard press
        return False

    def onKeyboardRelease(self, c):
        # triggered when a key is release
        # c is the character released
        # return False if you want sbgui to handle this keyboard release it ordinarily does
        # return False if you want sbgui to stop any further handling of the keyboard release
        return False

    def onEnd(self):
        # triggered when the interface is unloaded from the interface manager
    # create an instance of the interface
myinterface = MyInterface()
    # get the interface manager
m = GUIInterface.getInterfaceManager()
    # add the interface to the interface manager
m.addInterfaceListener(t)
    # if you want to remove that interface:
#m.removeInterfaceListener(t)

```

As an example, here is a simple class that prints out the location of mouse clicks and which button is pressed:

```
import GUIInterface
class TestInterface(GUIInterface.SBInterfaceListener):
    def onMouseClick(self, x,y,button):
        print str(x) + " " + str(y) + " " + str(button)
        return False
t = TestInterface()
m = GUIInterface.getInterfaceManager()
m.addInterfaceListener(t)
```

Creating a 2D Interface in sbgui

sbgui can draw 2D elements such as text, boxes and a variety of interactive widgets. The CEGUI library has been incorporated into sbgui (<http://cegui.org.uk/>) and any functionality from that 2D GUI system can be combined with SmartBody via Python.

2D Interface Example

Creating a text box:

```
winManager = WindowManagergetSingleton()
myText = winManager.createWindow('OgreTray/StaticText', 'myTextBox')
myText.setText("This is some text")
SystemgetSingleton().getDefaultGUIContext().getRootWindow().addChild(myText)
```

then the text can be changed:

```
myText.setText("The text has changed!")
```

There are some samples in the data/examples/CEGUI folder, and a file called GUIUtil.py which simplifies the code needed for GUI creation. However, you are only limited by the functionality in the CEGUI library.

Running SmartBody as an Embedded Python Library

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

- 1) copy all the dynamic libraries (.dll files on Windows, .so files on linux, .dylib files on OSX) from the bin/ to the python/Lib/site-packages directory
- 2) change the name of SmartBody.dll (in Windows, .dylib in OSX, .so in Linux) to SmartBody.pyd
- 3) create a file in site-packages with the name 'smartbody.pth' and put the word 'SmartBody' inside the file

You can then load a separate Python process which can access the SmartBody engine. For example, you should be able to make any calls to the SmartBody Python API:

```
import SmartBody
scene = SmartBody.getScene()
c = scene.createPawn( "mypawn" )
c.setPosition(SrVec(0, 100, 40))
```

Note that no user interface or 3D display will be automatically generated by SmartBody. In this mode, the engine functionality of SmartBody (creation and configuration of characters, execution of behaviors, and so forth), but any 3D display of information needs to be done manually.

Integrating SmartBody With a Game Engine

SmartBody can be integrated with any game engine or simulation 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 BoneBus interface, which is a network-based API, which would require communication over TCP/IP and UDP.

Suggested Integration Techniques

The main C++ API allows you to have access to most of SmartBody's internal structures, and are generally located in the /sb/ directory. There is also a simplified interface that can be used with C or C++ (located in smartbody-dll/include). In addition, there is a Python API which is nearly identical to the C++ API (generated from Boost-Python) which can be used to control the entire simulation, to set up the scene, or to run snippets of code during runtime.

The following pages describe the C++, C, Python and BoneBus interfaces.

One strategy for integrating SmartBody into a game or simulation is to run a process that indicates when the character's are created and to respond to those events. Then, run the SmartBody engine to create a proxy character in your own engine or simulation based on the function of the SmartBody engine and to have that character's state be controlled by a corresponding SmartBody character:

- a. In your engine/simulation, create a character. Simultaneously, create a SmartBody character in the SmartBody context with the same name.
- b. Implement the SBSceenListener interface which handles character creation/deletion and modification of SmartBody characters, and perform the equivalent actions in your engine/simulation.
- 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 engine/simulation character's state to match it

The following is a snippet from the SBSceenListener C++ interface:

```

class SBSceneListener
{
public:
    SBSceneListener() {}
    virtual ~SBSceneListener() {}
    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 ) {}
    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 ) {}
    virtual void OnLogMessage( const std::string & message ) {}
    virtual void OnEvent( const std::string & eventName, const std::string &
eventParameters ) {}
    virtual void OnObjectCreate(SmartBody::SBOBJECT* object) {}
    virtual void OnObjectDelete(SmartBody::SBOBJECT* object) {}

    virtual void OnSimulationStart() {}
    virtual void OnSimulationEnd() {}
    virtual void OnSimulationUpdate() {}
};


```

Callback	Event
OnCharacterCreate	Character is created
OnCharacterDelete	Character is deleted
OnCharacterUpdate	Character's skeleton, joints or other channels have changed
OnPawnCreate	Pawn is created
OnPawnDelete	Pawn is deleted
OnViseme	Face shape is activated with a particular value
OnChannel	Channel is producing a particular value
OnLogMessage	Message from SmartBody
OnEvent	An event is triggered from SmartBody
OnObjectCreate	Object is created in SmartBody
OnObjectDelete	Object is deleted in SmartBody

OnSimulationStart	Simulation starts
OnSimulationEnd	Simulation ends
OnSimulationUpdate	Simulation steps forward in time

Thus, an example would be to create a subclass of the listener:

```

class MyListener : public SmartBody::SBSceenListener
{
    MyListener ();
    ~MyListener ();

    // triggered when a SmartBody character is created
    void OnCharacterCreate( const std::string & name, const std::string & objectClass
);

    // triggered when a SmartBody character is deleted
    void OnCharacterDelete( const std::string & name );

    // triggered when a SmartBody character changes the skeleton (add/remove joints,
    channels)
    void OnCharacterUpdate( const std::string & name );

    // triggered when an object (character or pawn) is created
    void OnPawnCreate( const std::string & name );

    // triggered when an object (character or pawn) is deleted
    void OnPawnDelete( const std::string & name );

    // triggered when a SmartBody event occurs
    void OnEvent( const std::string & eventName, const std::string & eventParameters );
};

...
...

// create an instance of your own listener
MyListener* myListener = new MyListener();
// get the SmartBody scene
SmartBody::SBSceen* scene = SmartBody::SBSceen::getScene();
// add the listener to the scene
scene->addSceneListener(myListener);
// the callbacks will now be made automatically by SmartBody as those events occur

```

To run the SmartBody engine, then update the simulation with the current time, then read back the state of each character

```

// for each simulation step where currentTime = time in seconds
scene->update(currentTime);

// get all the characters
const std::vector<std::string>& characterNames = scene->getCharacterNames();
for (size_t c = 0; c < characterNames.size(); c++)
{
    SmartBody::SBCharacter* character = scene->getCharacter(characterNames[c]);
    // get the skeleton
    SmartBody::SBSkeleton* sbSkel = character->getSkeleton();
    int numJoints = sbSkel->getNumJoints();
    // for each joint, get the position/orientation
    for (int j = 0; j < numJoints; j++)
    {
        SmartBody::SBJoint* joint = sbSkel->getJoint(j);
        SrQuat orientation = joint->quat()->value();
        SrVec position = joint->getPosition();
        // copy the joint information onto the equivalent game engine character
        ...
        ...
    }
}
}

```

A character or object can be created in SmartBody by either using the C++ interface, as in the following example:

```

// creates a character with a skeleton that has a single joint
SmartBody::SBCharacter* character = scene->createCharacter("thename", "thetype");
// creates a skeleton from an existing asset
SmartBody::SBSkeleton* skeleton = scene->createSkeleton("mycharacter.dae");
// attaches the skeleton to the character
character->setSkeleton(skeleton);
// creates a standard set of controllers such as animation, gazing, head movements,
etc.
character->createStandardControllers();

```

or by sending Python commands to the engine through C++ by loading the contents of a file, or send the commands directly:

```

// runs all Python commands from a file
scene->runScript("myscript.py");
// runs a Python command directly
scene->run("scene.createCharacter(\"mycharacter\", \"mytype\")");

```

A character's skeleton can also be constructed procedurally and dynamically (as opposed loading the asset first, then defining the skeleton from that asset) as follows:

```

SmartBody::SBSkeleton* dynamicSkeleton = scene->addSkeletonDefinition("myskeleton");
SmartBody::SBJoint* baseJoint = skeleton->createJoint("base", NULL);
SmartBody::SBJoint* joint1 = skeleton->createJoint("child", baseJoint);
SmartBody::SBJoint* joint2 = skeleton->createJoint("grandchild", joint1);

...
// then later, create the character and a new instance of this skeleton
SmartBody::SBSkeleton* skeleton = scene->createSkeleton("myskeleton");
character->setSkeleton(skeleton);

```

For pure-Python interfaces, in general, there are Python functions that correspond directly to the C++ functions of the same name.

For example, a Python subclass of the SBSceneListener can be made like this:

```

class MyListener(CharacterListener):

    def OnCharacterCreate(self, name, type):
        print "Character created..."

    def OnPawnCreate(self, name):
        print "Pawn created..."

mylistener = MyListener()

scene.addSceneListener(mylistener)

```

Using the SmartBody library

The SmartBody library 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.

Example Usage

The following is an example of using the SmartBody library, loading a character, then running the simulation:

```
#include "vhcl.h"
#include <sb/SBScene.h>
#include <sb/SBCharacter.h>
#include <sb/SBSkeleton.h>
#include <sb/SBPython.h>
#include <sb/SBSimulationManager.h>
#include <sb/SBXmlProcessor.h>
#include <sb/SBSceneListener.h>
class SimpleListener : public SmartBody::SBSceneListener
{
public:
    SimpleListener() {}
    ~SimpleListener() {}

    virtual void OnLogMessage( const std::string & message )
    {
#ifdef WIN32
        LOG("%s", message.c_str());
#else
        std::cout << message << std::endl;
#endif
    }
};

int main( int argc, char ** argv )
{
    // set the relative path from the location of the simplesmartbody binary to the
    // data directory
    // if you are downloading the source code from SVN, it will be ../../../../data
    // std::string mediaPath = "../../../../data";
    // if you're using the SDK, this path will be ../data
    std::string mediaPath = "../data";
    // add a message logger to stdout
    vhcl::Log::StdoutListener* stdoutLog = new vhcl::Log::StdoutListener();
    vhcl::Log::g_log.AddListener(stdoutLog);
    LOG("Loading Python...");
    // initialize the Python libraries
    initPython("../Python27/Libs");
    // get the scene object
    SmartBody::SBScene* scene = SmartBody::SBScene::getScene();
    SimpleListener listener;
    scene->addSceneListener(&listener);
    // set the mediapath which dictates the top-level asset directory
    scene->setMediaPath(mediaPath);
    // indicate where different assets will be located
    // "motion" = animations and skeletons
    // "script" = Python scripts to be executed
    // "mesh" = models and textures
    scene->addAssetPath("motion", "ChrBrad");
```

```

// load the assets from the indicated locations
LOG("Loading Assets...");
scene->loadAssets();
int numMotions = scene->getNumMotions();
LOG("Loaded %d motions...", numMotions);
// create a character
LOG("Creating the character...");
SmartBody::SBCharacter* character = scene->createCharacter("mycharacter", "");
// load the skeleton from one of the available skeleton types
SmartBody::SBSkeleton* skeleton = scene->createSkeleton("ChrBrad.sk");
// attach the skeleton to the character
character->setSkeleton(skeleton);
// create the standard set of controllers (idling, gesture, nodding, etc.)
character->createStandardControllers();
// get the simulation object
SmartBody::SBSimulationManager* sim = scene->getSimulationManager();
// if you want to use a real-time clock do the following:
bool useRealTimeClock = true;
if (useRealTimeClock)
{
    sim->setupTimer();
}
else
{
    // otherwise, the time will run according
    sim->setTime(0.0);
}
// make the character do something
scene->getBmlProcessor()->execBML("mycharacter", "<body"
posture=\"ChrBrad@Idle01\"/>");

LOG("Starting the simulation...");
double lastPrint = 0;
sim->start();
while (sim->getTime() < 100.0) // run for 100 simulation seconds
{
    scene->update();
    if (!useRealTimeClock)
        sim->setTime(sim->getTime() + 0.16); // update at 1/60 of a second when
running in simulated time
    else
        sim->updateTimer();

    if (sim->getTime() > lastPrint)
    {
        LOG("Simulation is at time: %5.2f\n", sim->getTime());
        lastPrint = sim->getTime() + 10;
    }
    const std::vector<std::string>& characterNames = scene->getCharacterNames();
    for (size_t c = 0; c < characterNames.size(); c++)
    {
        SmartBody::SBCharacter* character = scene->getCharacter(characterNames[c]);
        std::string jointName = "JtRoot";
        SmartBody::SBJoint* joint =
character->getSkeleton()->getJointByName(jointName);
        if (joint)
        {
            SrVec position = joint->getPosition();
            LOG("Character %s joint %s is at position (%f, %f, %f)",


```

```
character->getName().c_str(), jointName.c_str(), position.x, position.y, position.z);
    }
}
sim->stop();
```

```
    return 0;
}
```

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

```
namespace SmartBody
{
    class SBSceenListener
    {
        public:
            SBSceenListener() {}
            virtual ~SBSceenListener() {}
            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 ) {}
            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 ) {}
            virtual void OnLogMessage( const std::string & message) {}
            virtual void OnObjectCreate(SmartBody::SBOject* object) {}
            virtual void OnObjectDelete(SmartBody::SBOject* object) {}

            virtual void OnSimulationStart() {}
            virtual void OnSimulationEnd() {}
            virtual void OnSimulationUpdate() {}

    };
}
```

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 located in the `lib/bonebus/include` directory.

Since this interface uses TCP/IP, it is not as efficient as embedding the SmartBody library within the application.

Tutorials

1. Control SmartBody with Python
2. Creating Objects
3. Loading Assets
4. Creating Characters
5. Controlling Characters Using BMLs
6. Using Animation Blends and Behavior Sets
7. Facial Expressions and Emotion
8. Speech
9. Event Handling
10. Locomotion
11. Steering
12. Retargeting
13. Physics

1. Control SmartBody with Python

In this tutorial, you will learn the basics of controlling SmartBody via Python commands. The section [Using Python with SmartBody](#) describes different ways to sending out Python commands to SmartBody. Though the easiest way is to open the **Command Window** by selecting **Window->Command Window** from the menu.

The first step to access SmartBody via Python is to use 'scene' object. It is the main entity to create characters, query system informations, or obtain other manager class to create different character capabilities.

Here is an example to use the scene object :

```
print 'media path = ' + scene.getMediaPath()
```

The above command obtain and print out the default media path from SmartBody.

You can also print out additional informations such as number of characters and pawns in the scene using other commands :

```
print 'num of pawns in the scene = ' + str(scene.getNumPawns())
print 'num of characters in the scene = ' + str(scene.getNumCharacters())
```

Note : For now, ignore the meaning for media path or pawns as we will discuss more about them in the coming tutorials.

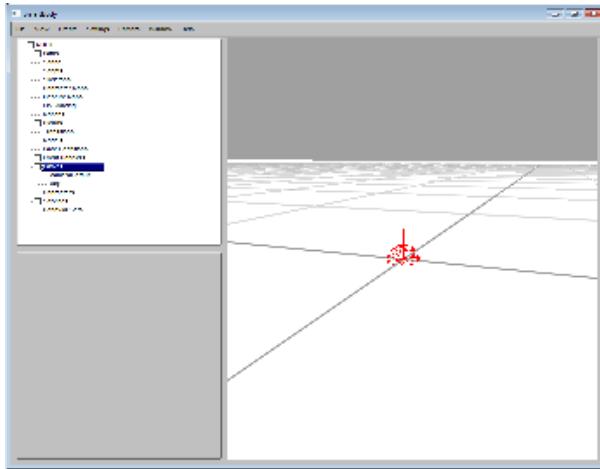
2. Creating Objects in the Scene

This tutorial is about how to create a pawn object in the scene. A 'pawn' is a spatial object that can be moved, rotated in the scene. It can be used to represent things that can be manipulated spatially in the scene such as lights or cameras.

To create a pawn, simply use the following command :

```
obj = scene.createPawn('obj1')
```

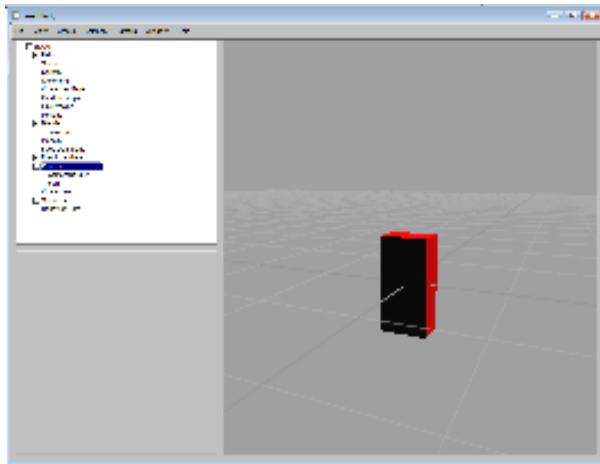
It will create a pawn object with name 'obj1'. The newly created pawn will appear as a red wireframe ball on the screen.



We can then assign the geometry for this pawn by setting its shape properties :

```
obj.setStringAttribute('collisionShape', 'box')
obj.setVec3Attribute('collisionShapeScale', 5.0, 10.0, 3.0)
```

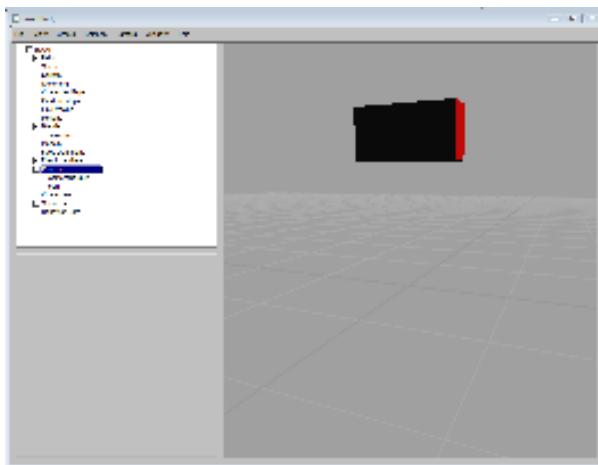
In the above codes, we set the shape to a 'box' and scale the shape size in x,y, and z directions. This will create a rectangular box like the following image (note that the box will penetrate the floor, which can be turned off using View->Show Floor):



We can also apply further operation on the pawn object to adjust its positions and rotations :

```
obj.setPosition(SrVec(0,30,0))
obj.setHPR(SrVec(0,0,90))
```

This will move the object up in Y-axis and rotate it by 90 degrees.



There are more advanced properties such as physics and 3D mesh models that can be applied on a pawn. We will discuss about these in more advanced tutorials.

3. Loading Assets

In this tutorial, we will discuss about loading various assets from file system into SmartBody. As a character animation system, SmartBody requires several different types of data to work. For example, a skeleton is used to define the character structure, a motion is used to animate the character, a skin mesh is used to deform and render the character, and a script file stores macros of multiple Python commands.

To load the data, we need to first get the 'asset manager' object from scene object.

```
assetManager = scene.getAssetManager()
```

Then we can define the local paths for asset manager to find different type of asset files.

```
assetManager.addAssetPath('motion', 'ChrBrad')
assetManager.addAssetPath('mesh', 'mesh')
assetManager.addAssetPath('script', 'scripts')
```

For example, here the first 'addAssetPath' means we will add a search path 'ChrBrad' for 'motion' type of assets into the asset manager. So the system will search 'ChrBrad' directory when trying to load new motions or skeletons. Note that 'ChrBrad' is a relative directory to the system media path (default media path is \$SmartBodyDirectory/data). You can use `scene.setMediaPath` to change the media path directory. Similarly, the other 'addAssetPath' commands will set the search path for 'mesh' and 'script' assets.

Once we setup the search paths for assets, we can have the system load up the skeleton and motion files by calling :

```
assetManager.loadAssets()
```

This command will go through each search path, and load the skeleton and motion asset files inside those paths. Note that 'mesh' and 'script' files will not be loaded by this command, but the search paths will be used when the system needs to look for such assets. For example, you can execute a Python script file directly by calling :

```
scene.run('zebra2-map.py')
```

The system will then search for script file 'zebra2-map.py' in the 'script' asset paths and execute it. If we didn't set the asset paths for 'script', then the system will not be able to find the script file and will produce an error.

Once the system finish loading all motions and skeletons from asset paths, we can query for the list of names of the assets (motions and skeletons) currently loaded into the system :

```

motionNames = assetManager.getMotionNames()
skelNames = assetManager.getSkeletonNames()
for i in range(0,len(motionNames)):
    print 'motion ' + str(i) + ' = ' + motionNames[i]
for i in range(0,len(skelNames)):
    print 'skeleton ' + str(i) + ' = ' + skelNames[i]

```

These asset files can then be used to create and animate a character. We will discuss this in the next tutorial.

4. Creating Character

In this tutorial, we will discuss about setting up a SmartBody character using the assets loaded in the previous tutorial. Load the assets as before, and set the scene scale to 1.0 (meters) instead of the default of .01 (centimeters).

```

# Add asset paths
assetManager = scene.getAssetManager()
assetManager.addAssetPath('motion', 'ChrBrad')
assetManager.addAssetPath('mesh', 'mesh')
assetManager.addAssetPath('script', 'scripts')
# Load assets based on asset paths
assetManager.loadAssets()

# set the scene scale and reset the camera
scene.setScale(1.0)
scene.getActiveCamera().reset()

```

Now assign a correct joint map for the loaded skeletons and motions :

```

scene.run('zebra2-map.py')
zebra2Map = scene.getJointMapManager().getJointMap('zebra2')
bradSkeleton = scene.getSkeleton('ChrBrad.sk')
zebra2Map.applySkeleton(bradSkeleton)
zebra2Map.applyMotionRecurse('ChrBrad')

```

The above code first run 'zebra2-map.py' to define a joint map for 'zebra2' type skeleton and motions. Then it applies the 'zebra2' joint map on the 'ChrBrad' skeleton and its corresponding motions. This step is necessary to adapt the asset data to be used by SmartBody. Since the naming conventions for joint/bone names may be different between SmartBody and the loaded assets, the joint map acts as a converter to translate the joint names so SmartBody can operate on these assets. For now, let's just assume these are necessary steps before creating a character. More details will be explained in future tutorials and in Section [Using Custom Skeletons and Animations](#).

Once the correct joint map is applied, we can create the character with following commands :

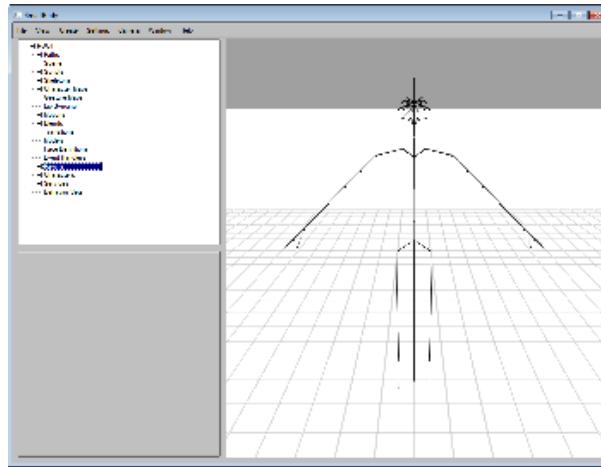
```

# Set up Brad
brad = scene.createCharacter('ChrBrad', '')
bradSkeleton = scene.createSkeleton('ChrBrad.sk')
brad.setSkeleton(bradSkeleton)
# Set standard controller
brad.createStandardControllers()

```

The above commands are straightforward. We first create a character with name 'ChrBrad', and then assign a skeleton 'ChrBrad.sk' to him to define the skeleton hierarchy for the character. Then it calls 'createStandardControllers' to create a set of default controllers for the character to enable the capabilities such as gaze, head nods, etc.

At this point, the character will show up only as skeleton instead of a 3D model like in the following image. This is because we only provides the skeleton information to define the character.

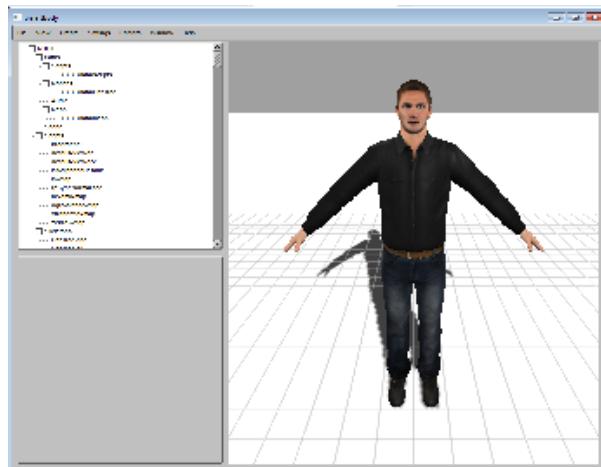


To also load and set the 3D character model, we need to define the character skin mesh for ChrBrad :

```
brad.setDoubleAttribute( 'deformableMeshScale', .01)
brad.setStringAttribute( 'deformableMesh', 'ChrBrad.dae')

# show the character
brad.setStringAttribute( 'displayType', 'GPUmesh' )
```

Here we first set the mesh scale to be 0.01. This is because the character mesh is in centimeter scale but the 'ChrBrad.sk' skeleton is in meter scale. Therefore we need to rescale the mesh to fit the skeleton. Then we set the 'deformableMesh' attribute to 'ChrBrad'. This will ask the system to search for 'ChrBrad' skin mesh in the search directory, load the files, and then apply the mesh to the character. Once the skin mesh is setup correctly, setting the attribute called 'displayType' will tell the renderer to show the mesh instead of bones. You can also select **View->Character->Deformable Geometry** from the menu to render all meshes for all characters.



The full setup script for creating the above character is :

```

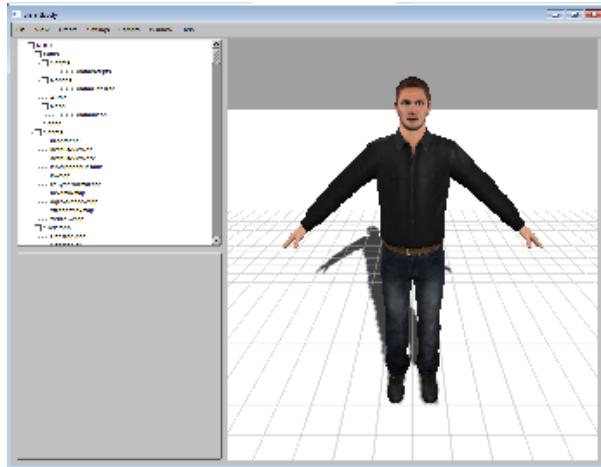
print "|-----| "
print "|      Starting Tutorial 4      | "
print "|-----| "
print 'media path = ' + scene.getMediaPath()
# Add asset paths
assetManager = scene.getAssetManager()
assetManager.addAssetPath('motion', 'ChrBrad')
assetManager.addAssetPath('mesh', 'mesh')
assetManager.addAssetPath('script', 'scripts')
# Load assets based on asset paths
assetManager.loadAssets()
# set scene scale and reset the camera
scene.setScale(1.0)
scene.getActiveCamera().reset()
# run a Python script file
scene.run('zebra2-map.py')
zebra2Map = scene.getJointMapManager().getJointMap('zebra2')
bradSkeleton = scene.getSkeleton('ChrBrad.sk')
zebra2Map.applySkeleton(bradSkeleton)
zebra2Map.applyMotionRecurse('ChrBrad')
# Set up Brad
brad = scene.createCharacter('ChrBrad', '')
bradSkeleton = scene.createSkeleton('ChrBrad.sk')
brad.setSkeleton(bradSkeleton)
# Set standard controller
brad.createStandardControllers()
# Deformable mesh
brad.setDoubleAttribute('deformableMeshScale', .01)
brad.setStringAttribute('deformableMesh', 'ChrBrad.dae')
# show the character
brad.setStringAttribute('displayType', 'GPUMesh')

```

5. Controlling Characters Using BMLs

This tutorial is about controlling the characters to perform various actions. SmartBody uses BML commands to set the character behaviors, such as posture, animation, gaze, and etc.

After you setup the character, he will be in T-pose, which is not very interesting.



So the first thing is to set a more natural posture for the character :

```
bml.execBML( 'ChrBrad' , '<body posture="ChrBrad@Idle01"/>' )
```

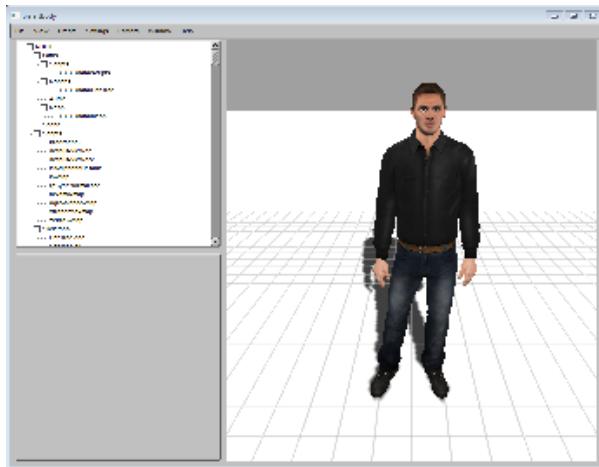
here, 'bml' is a SmartBody object for executing the BML command. The first parameter 'ChrBrad' indicates the name of the character for this BML command, and the second parameter is the BML command.

The format of a typical bml command is in the form of XML :

```
<commandName commandParameter="value" />
```

For example, to set the character posture, the commandName is '**body**' and the commandParameter is '**posture**' with value to be '**ChrBrad@Idle01**'. Depending on the commandName, a BML can have multiple commandParameters, though it is simply one parameter in this case. Note that the BML command needs to be bracketed correctly (start with '<' and end with '/>').

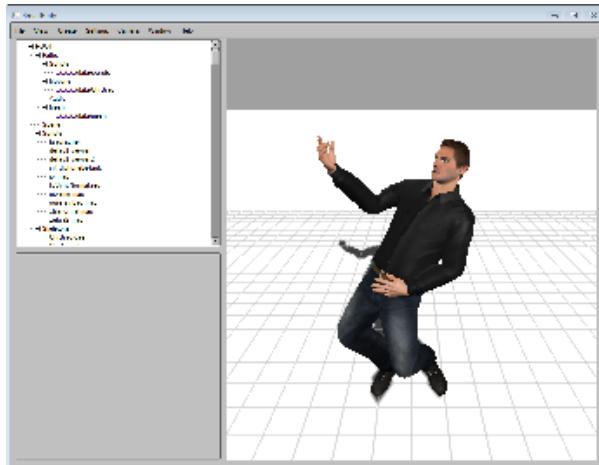
Once we set the posture for Brad character, the idle animation loop will be applied on him.



We can also have Brad play the guitar by sending an animation command :

```
bml.execBML( 'ChrBrad' , '<animation name="ChrBrad@Guitar01"/>' )
```

This command will apply the Guitar animation on Brad. Instead of looping it like posture, the animation will only be played once.



Another behavior is head nod, which will have Brad move his head in different directions :

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

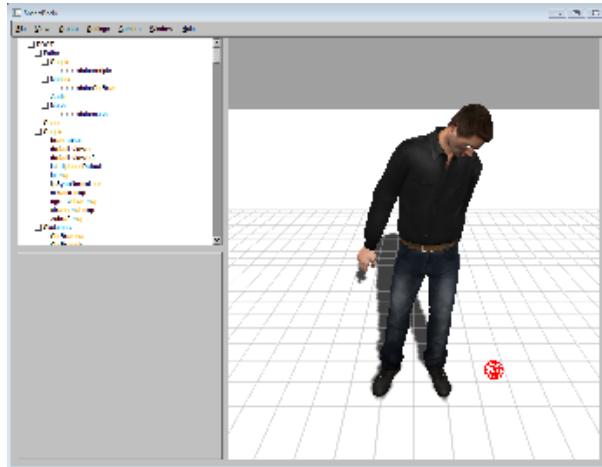
Although here we only specify the head movement type as 'NOD', there are other parameters that can be set such as speed and magnitude of head movements. More details can be found in Section [Head Movements](#).

The last behavior we will discuss about is gaze control. To setup a gaze, simply use the following command :

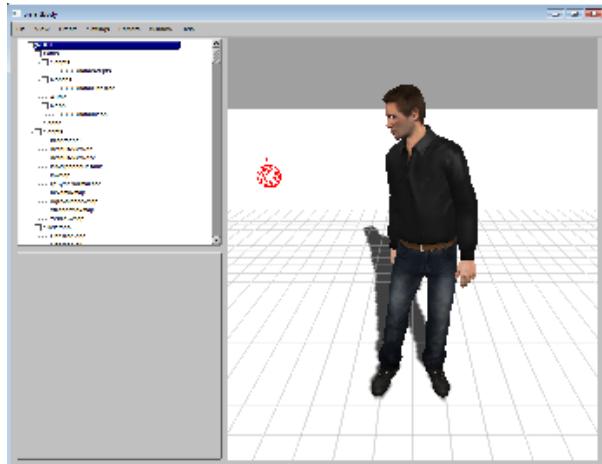
```
obj = scene.createPawn('obj1')
obj.setPosition(SrVec(0.5,0.1,0))
bml.execBML( 'ChrBrad' , '<gaze target="obj1" />' )
```

Here the first two lines are not the real gaze command, but they should look familiar. They simply setup a pawn and set its position. This is because the gaze command require a gaze target, which is usually a pawn object.

Once the gaze is setup, the character will look at the pawn on the ground by rotating his head and spines :



If we try to move the pawn to a different location, the character will also adjust accordingly to track the new gaze target :



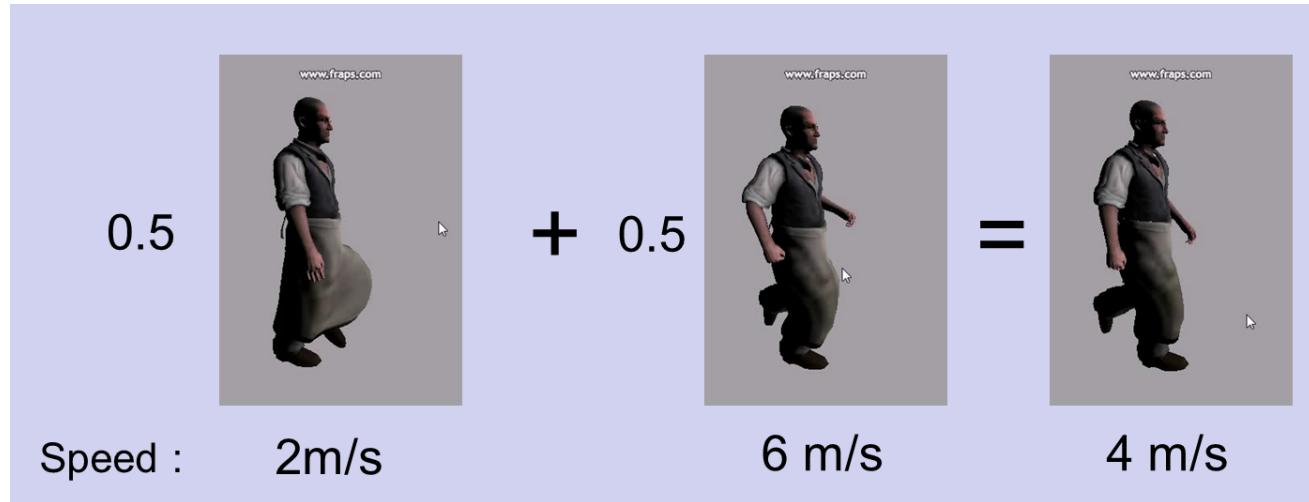
(To move a pawn, you can either apply pawn.setPosition command, or you can drag the pawn directly on the GUI window by selecting it with the mouse, pressing the 'w' key, then moving the manipulator handles.

More details about other BML commands can also be found in Section [BML Behaviors](#).

6. Using Animation Blends and Behavior Sets

Animation blending is a power way of synthesizing various behaviors for a character from a small set of example animations.

For example, if we have a walking animation and a running animation, we can average the two animations together to form another animation that has the character jogging.



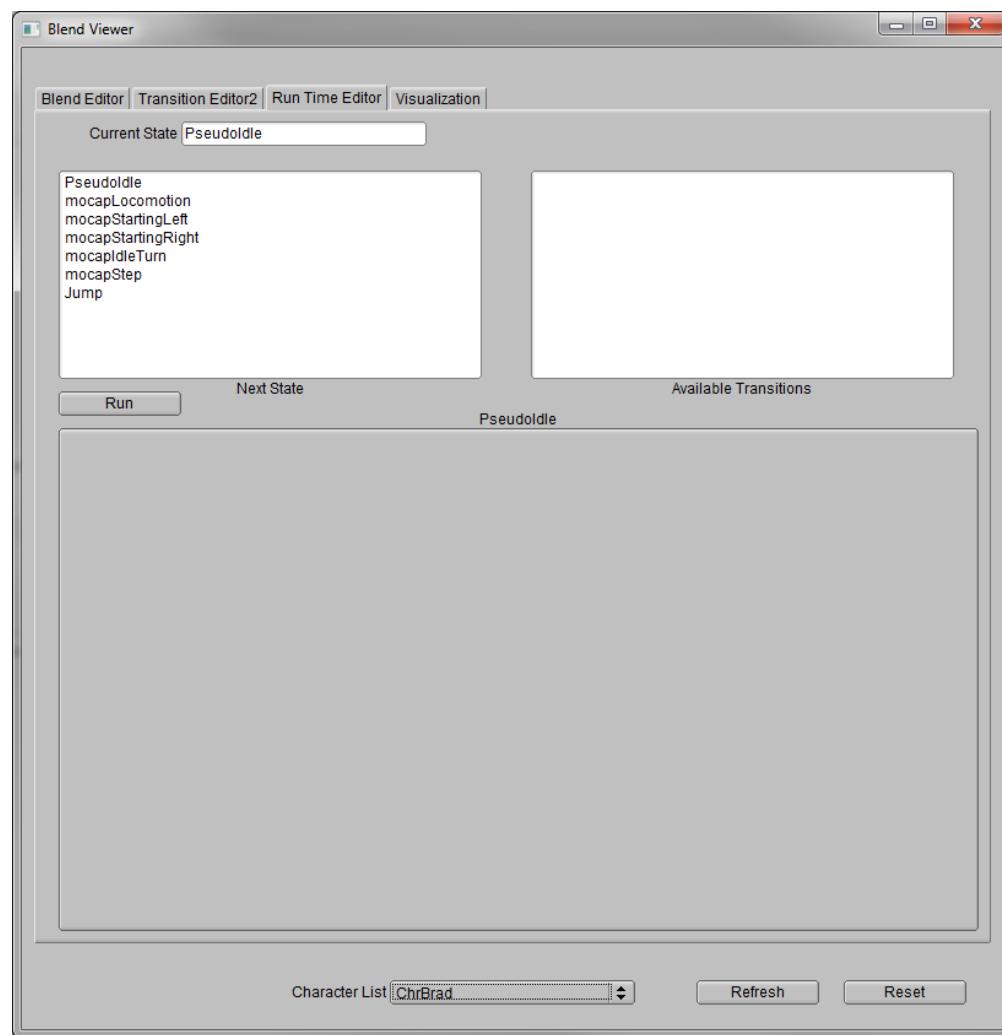
SmartBody system already comes with several different sets of animation data that can be used as animation blend. To make it easier to integrate with different character, these animation data are organized into behavior sets which represent different actions such as locomotion, jumping, and etc. Although it is possible for a user to define his own animation blend using custom data, this tutorial is focused on how to make use of existing behavior sets in SmartBody. Setting up an animation blend and behavior is a complicated task that requires some programming efforts. Thus it is only recommended for advanced users.

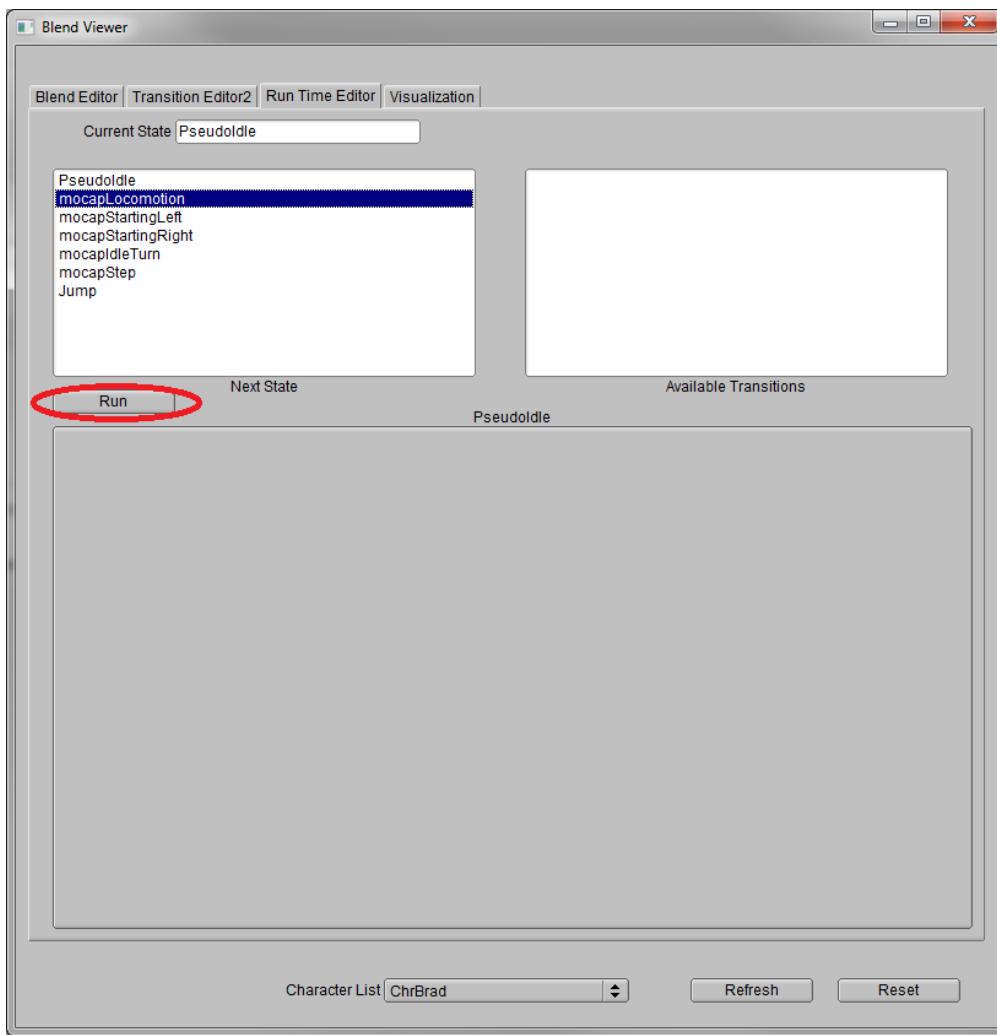
To apply a behavior set on a character, we can use the following scripts :

```
scene.addAssetPath("script", "behaviorsets") # add search path for behavior set
# locomotion
scene.run('BehaviorSetMaleMocapLocomotion.py')
setupBehaviorSet()
retargetBehaviorSet('ChrBrad', 'ChrBrad.sk')
```

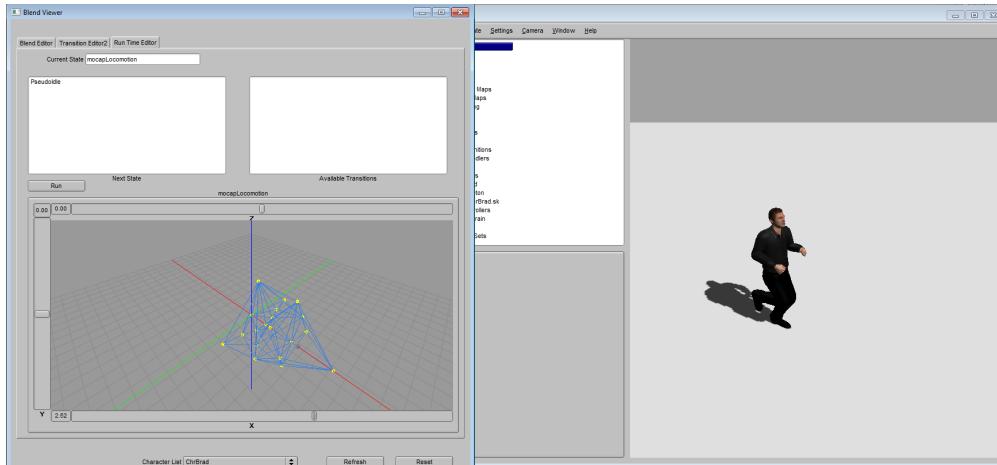
The first is to add a script search path for the behavior setup scripts. Then we simply run the setup script for 'MaleMocapLocomotion' behavior and call its corresponding Python procedure to setup and retarget the behavior set on the character. The whole process of retargeting is automatic, and the user only needs to provide the character name and his corresponding skeleton.

Once the locomotion behavior is set up, we can play with this behavior via Blend Viewer by selecting **Window->Blend Viewer** from menu. Once you open the **Blend Viewer**, it should look like the image on the left. First, make sure we select **Run Time Editor** tab. Then select mocapLocomotion from the list, and press Run button, as shown on the right image.





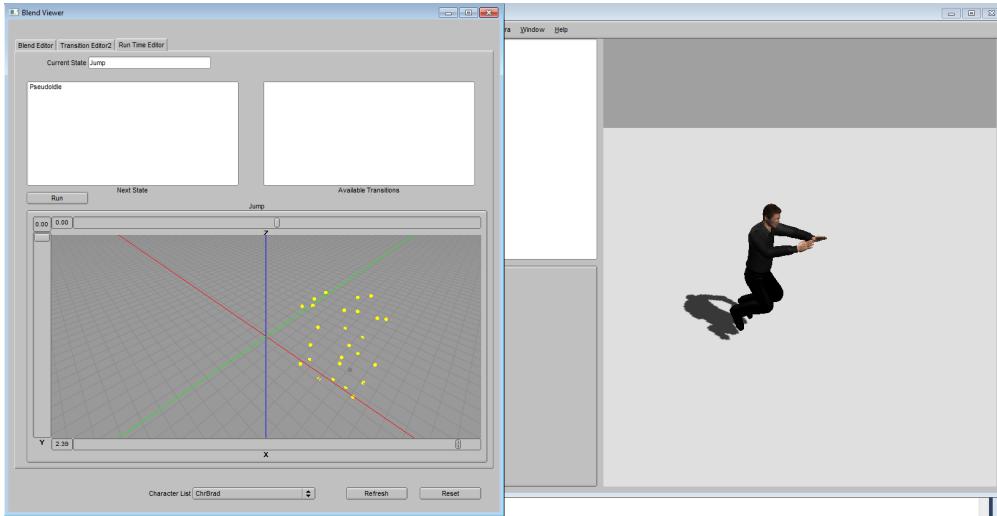
You should see a 3D control in the Blend Viewer. This is the 3D parameter control for the locomotion. You can drag the bar on X-axis to adjust the character's moving speed and Y-axis to adjust the turning speed. During adjustment, you will see the character starts walking around. You can play with these parameter values to have the character walk in different speed and navigate the environments.



Another behavior set we can add is Jumping. It can be added in a similar way like locomotion by using the corresponding setup script for jump behavior.

```
# jump
scene.run('BehaviorSetJumping.py')
setupBehaviorSet()
retargetBehaviorSet('ChrBrad', 'ChrBrad.sk')
```

Once the jump behavior is set up, select 'Jump' in the Blend Viewer and run it. Similar to controlling the locomotion, the jumping can be adjusted by setting the parameters in different axis. Here X-axis is jumping distance, Y-axis is jumping height, and Z-axis is jumping direction. We can apply different parameters to have the character jump around in the environment.



7. Facial Expressions and Emotions

This tutorial is about setting up and adjust the facial expression for a character. SmartBody allows control over a character's facial expressions by setting weights for different face action units. By combining different facial poses with different blending weights, it can enables a character to perform various emotional expression or even perform lip-syncing animation to an utterance.

The first step is to setup the face definition for the character. The goal is to assign a proper face pose to each Face Action Unit according to Face Action Coding System (FACS) and to each viseme for speech animation.

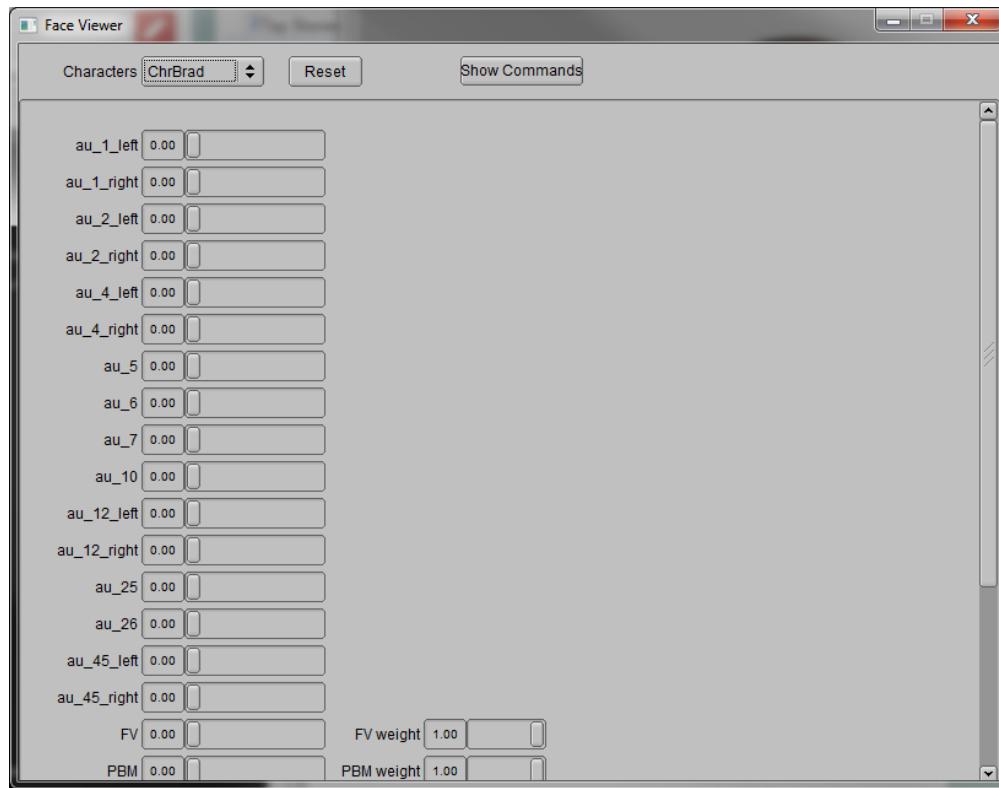
```

bradFace = scene.createFaceDefinition('ChrBrad')
bradFace.setFaceNeutral('ChrBrad@face_neutral')
bradFace.setAU(1, "left", "ChrBrad@001_inner_brow raiser_lf")
bradFace.setAU(1, "right", "ChrBrad@001_inner_brow raiser_rt")
bradFace.setAU(2, "left", "ChrBrad@002_outer_brow raiser_lf")
bradFace.setAU(2, "right", "ChrBrad@002_outer_brow raiser_rt")
bradFace.setAU(4, "left", "ChrBrad@004_brow_lowerer_lf")
bradFace.setAU(4, "right", "ChrBrad@004_brow_lowerer_rt")
bradFace.setAU(5, "both", "ChrBrad@005_upper_lid raiser")
bradFace.setAU(6, "both", "ChrBrad@006_cheek raiser")
bradFace.setAU(7, "both", "ChrBrad@007_lid_tightener")
bradFace.setAU(10, "both", "ChrBrad@010_upper_lip raiser")
bradFace.setAU(12, "left", "ChrBrad@012_lip_corner puller_lf")
bradFace.setAU(12, "right", "ChrBrad@012_lip_corner puller_rt")
bradFace.setAU(25, "both", "ChrBrad@025_lips_part")
bradFace.setAU(26, "both", "ChrBrad@026_jaw_drop")
bradFace.setAU(45, "left", "ChrBrad@045_blink_lf")
bradFace.setAU(45, "right", "ChrBrad@045_blink_rt")
bradFace.setViseme("open", "ChrBrad@open")
bradFace.setViseme("W", "ChrBrad@W")
bradFace.setViseme("ShCh", "ChrBrad@ShCh")
bradFace.setViseme("PBM", "ChrBrad@PBM")
bradFace.setViseme("FV", "ChrBrad@FV")
bradFace.setViseme("wide", "ChrBrad@wide")
bradFace.setViseme("tBack", "ChrBrad@tBack")
bradFace.setViseme("tRoof", "ChrBrad@tRoof")
bradFace.setViseme("tTeeth", "ChrBrad@tTeeth")
brad.setFaceDefinition(bradFace)

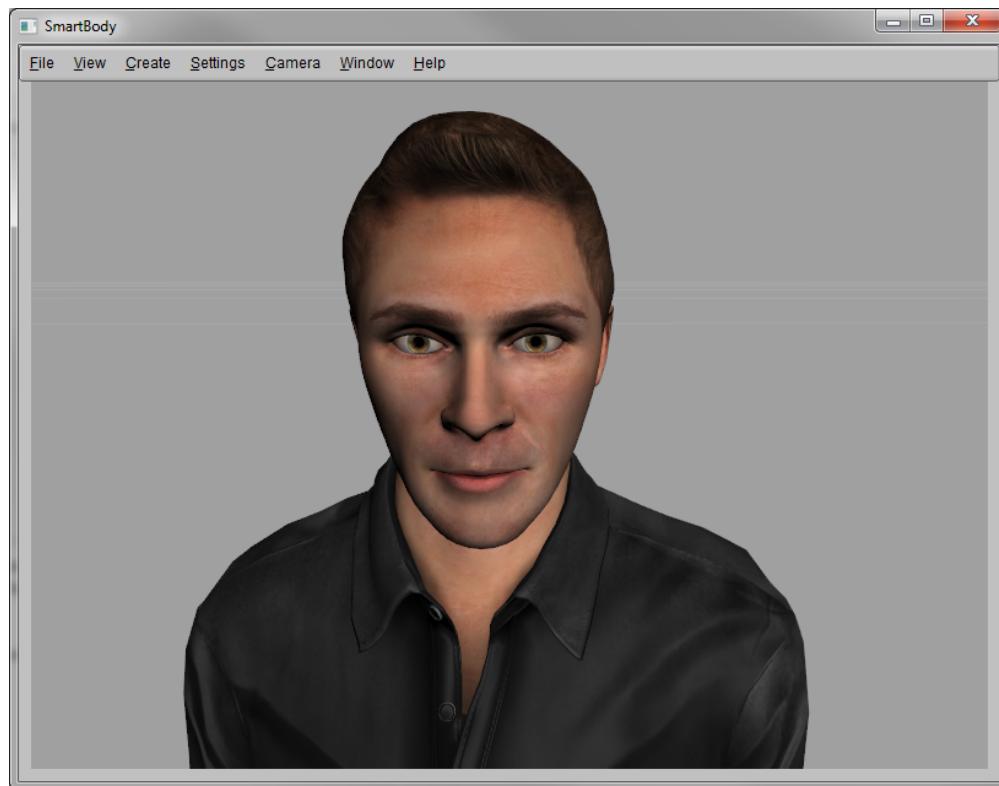
```

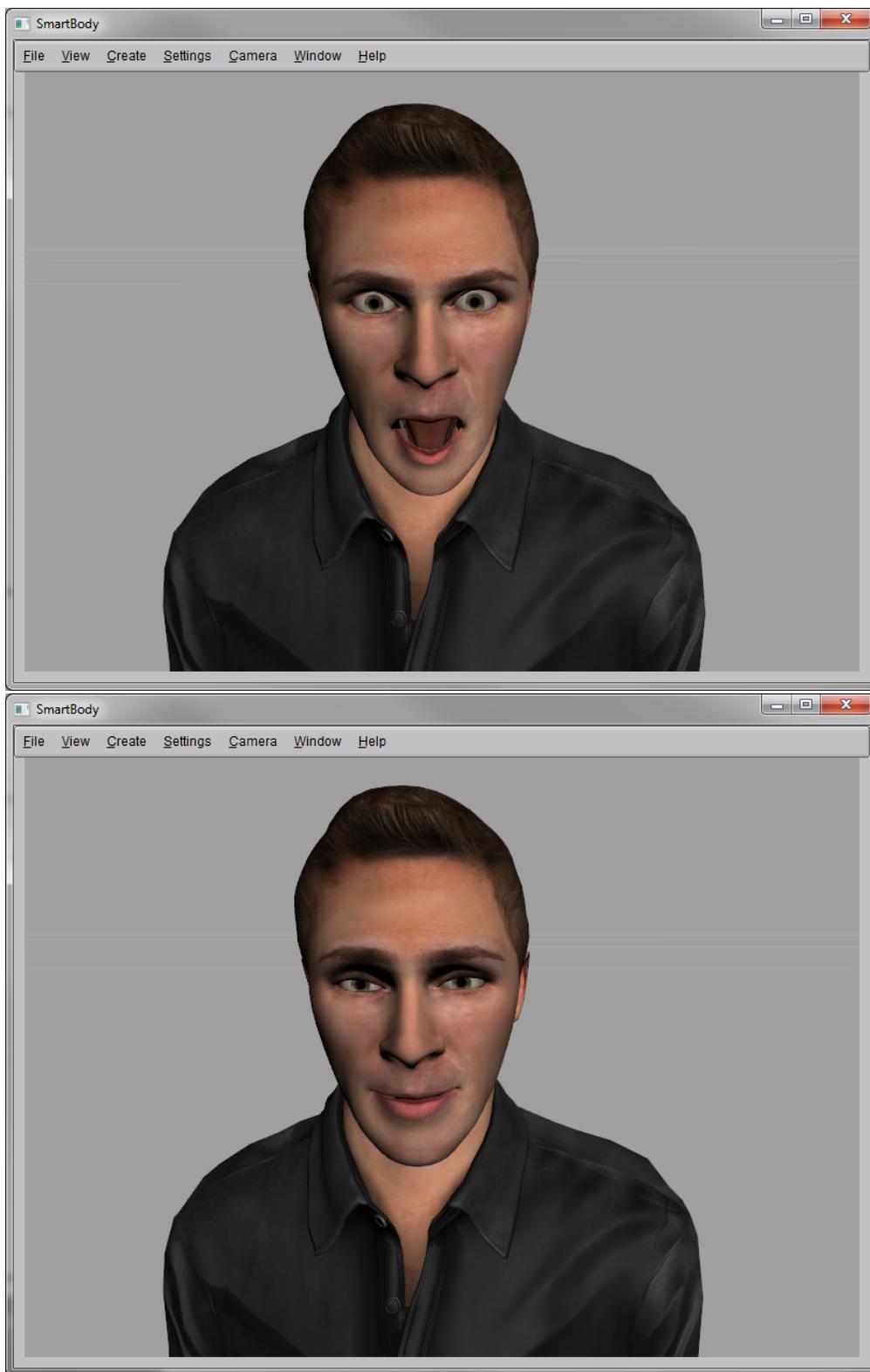
This looks like a big chunk of codes, but it is actually very straightforward. Line 1 simply create an empty face definition for the character. Then we need to first assign a neutral face pose as the baseline for all expressions. Here 'ChrBrad@face_neutral' is the file name for the face pose. After setting the neutral face pose, we also assign face poses for each action unit (AU) and viseme in a similar manner (Line 3 ~ Line 27). Once we finish assigning the face poses, the face definition is set to the character. This way the character can start using these face poses accordingly when synthesizing facial animations.

Once we setup the face poses for the character, we can test the face expressions using **Face Viewer** by selecting **Window->Face Viewer**.



Here each slider indicates the weight for a specific face pose AU or viseme. Since these poses are additive, we can have different weights for multiple face poses. By adjusting the weights, we are able to change the character's facial expressions to reflect different emotions.





More details about configuring face definition and changing face pose weights via BML command can be found in Section [Configuring Facial Animations and Lip Syncing](#) and Section [BML Face Command](#).

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
sbgui	a standalone SmartBody application and scene renderer
SmartBody	a dynamic library that can be incorporated into a game engine or other application
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.
simplesmartbody	a standalone SmartBody application that demonstrates how to use the SmartBody API
ogresmartbody	example of connecting the Ogre rendering engine to SmartBody
irrlichtsmartbody	example of connecting the Irrlicht game engine to 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.

Character Controllers

The main focus of SmartBody is on the control and display of a virtual character. Each character has a set of controllers that operate together to perform various behaviors and functions for the character. The current group of controllers is:

Controller	Purpose
World offset controller	Controls the position and orientation of the character in the 3D space
Posture scheduler	Schedules idle postures (standing, sitting, etc.)
Parameterized animation controller	Controls animations, blends, states and transitions between animation states.
Motion graph controller	Controls animations made using the motion graph.
Basic locomotion controller	Controls the locomotion of the character by using basic animation that moves but does not animate the character.
Single-animation locomotion controller	Controls the locomotion of a character using a single walk cycle (experimental)
Animation controller	Controls other animations (gestures, etc.)
Postprocess controller	Handles IK for footskate cleanup
Hand controller	Adds finger motion to gestures (experimental)
Reach scheduler	Handles arm and body movement for reaching/grabbing/touching/pointing motions
Grab scheduler	Handles fingers and hand movement for reaching/grabbing/touching/pointing motions, as well as any hand configurations
Breathing controller	Handles breathing of character
Gaze scheduler	Handles gaze

Eye saccade controller	Handles saccadic eye movements (fast movements of the eye)
Eyelid controller	Controls the eyelid position in response to the movement of the eyeball, or to change the appearance of the character
Head schedule	Handles head movements (nodding, shaking, etc.)
Face controller	Controls the movement of the face, including expressions and lip synchronization to speech
Physics controller	Handles physics for character, for ragdoll and dynamic controllers
Noise controller	Adds noise to the joints to simulate shaking and holding effects
Constraint controller	Manages positional and rotational constraints for the character.
Motion player	Playback of motion to override existing character state
Data receiver	Overrides character state from a outside source (for puppeteering from a Kinect sensor, for example)
Recorder	Stores the character state for persistence to a file.

The controllers are executed in sequence, with some controllers overriding values from previous controllers, and others modifying values.

SmartBody Components

SmartBody uses a number of 3rd party libraries, including:

BOOST (www.boost.org)

FLTK (www.fltk.org)

FLU - FLTK Utility Widgets

SteerSuite (<http://steersuite.cse.yorku.ca/>)

Open Dynamics Engine (<http://www.ode.org>)

ActiveMQ (<http://activemq.apache.org>)

glew (<http://glew.sourceforge.net>)

recast and detour (<http://code.google.com/p/recastnavigation>)

zlib (<http://www.zlib.net>)

festival (<http://www.cstr.ed.ac.uk/projects/festival>)

Ogre (<http://www.ogre3d.org>)

Pinocchio (<http://www.mit.edu/~ibaran/autorig/pinocchio.html>)

PolyVox (<http://www.volumesoffun.com/polyvox-about/>)

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:

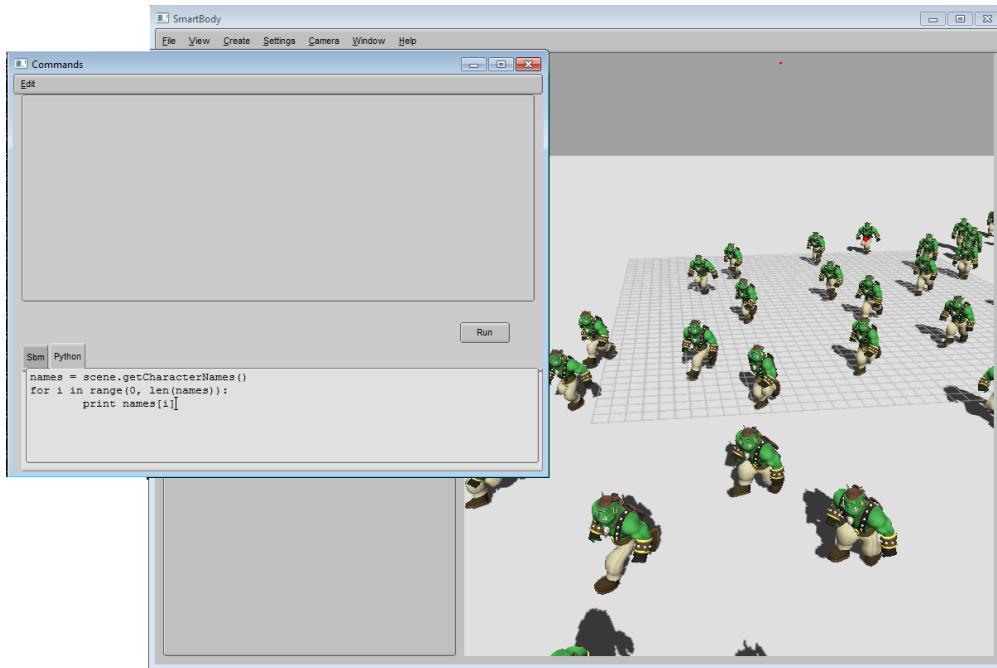
```
sb scene.run(scriptName)
```

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

Sending Python Commands via the SmartBody application sbgui

If you are using the default SmartBody application (sbgui), you can send a Python command by doing the following:

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 function:

```
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.
SBAnimationBlend	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 SBSpace. The scene contains most of the functions needed to access, create and remove all SmartBody-related object. Can also be retrieved via: <code>getScene()</code>
bml	The BML processor, which can be used to send BML commands to various character. Can also be retrieved via: <code>scene.getBMLProcessor()</code>
sim	The simulation manager, which can be used to start, stop and pause the simulation. Can also be retrieved via: <code>scene.getSimulationManager()</code>
asset	The asset manager which holds all the motions, skeletons, models and other assets.

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 package called 'SmartBody' and places an instance of SBSpace 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, or a single asset from a particular file 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 uniformly 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	<p>Turns on or off SmartBody's handing of PlaySound messages.</p> <p>If set to True, SmartBody will capture and play the sounds indicated. If set to False, it will ignore PlaySound messages.</p> <p>When using SmartBody as a standalone application (such as via sbgui), this should be set to True.</p> <p>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.</p>
speechRelaySoundCacheDir	../../../../	<p>Directory where sound files from speech relays will be placed.</p> <p>The external speech relays transform text into an audio file, and place the resultant sound file in a particular directory.</p> <p>By changing this attribute, the directory can be set to any location.</p>
scale	.01	<p>The scale of the scene in meters. By default, the scene scale is set to centimeters.</p> <p>This attribute is queried during locomotion, steering, camera movement and so forth.</p>

colladaTrimFrames	0	<p>The number of frames to be trimmed when loading a COLLADA motion.</p> <p>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.</p> <p>SmartBody will ignore the first such frames, and read the remaining data as the animation.</p>
useFastXMLParsing	False	<p>Use faster parsing when reading XML from a file.</p> <p>SmartBody generally uses Xerces for XML parsing.</p> <p>By setting this value to True, SmartBody will use RapidXML for parsing audiofiles for prerecorded speech.</p>
useFastCOLLADAParsing	True	Uses in-situ XML parser for asset file, which is much faster than the DOM parser
delaySpeechIfNeeded	True	<p>Delays any speech until other behaviors specified in the same BML need to execute beforehand.</p> <p>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.</p>

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

retime	speeds up or slows down a motion by a scaling factor, where 1 = normal speed, .5 = half speed, 2 = twice speed	<code>motion.retime(factor)</code>
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.	<code>motion.retarget(motionname, sourceSkeletonName, destSkeletonName, joints)</code>
trim	Tims the starting and ending frames of a motion	<code>motion.trim(5, 23)</code>

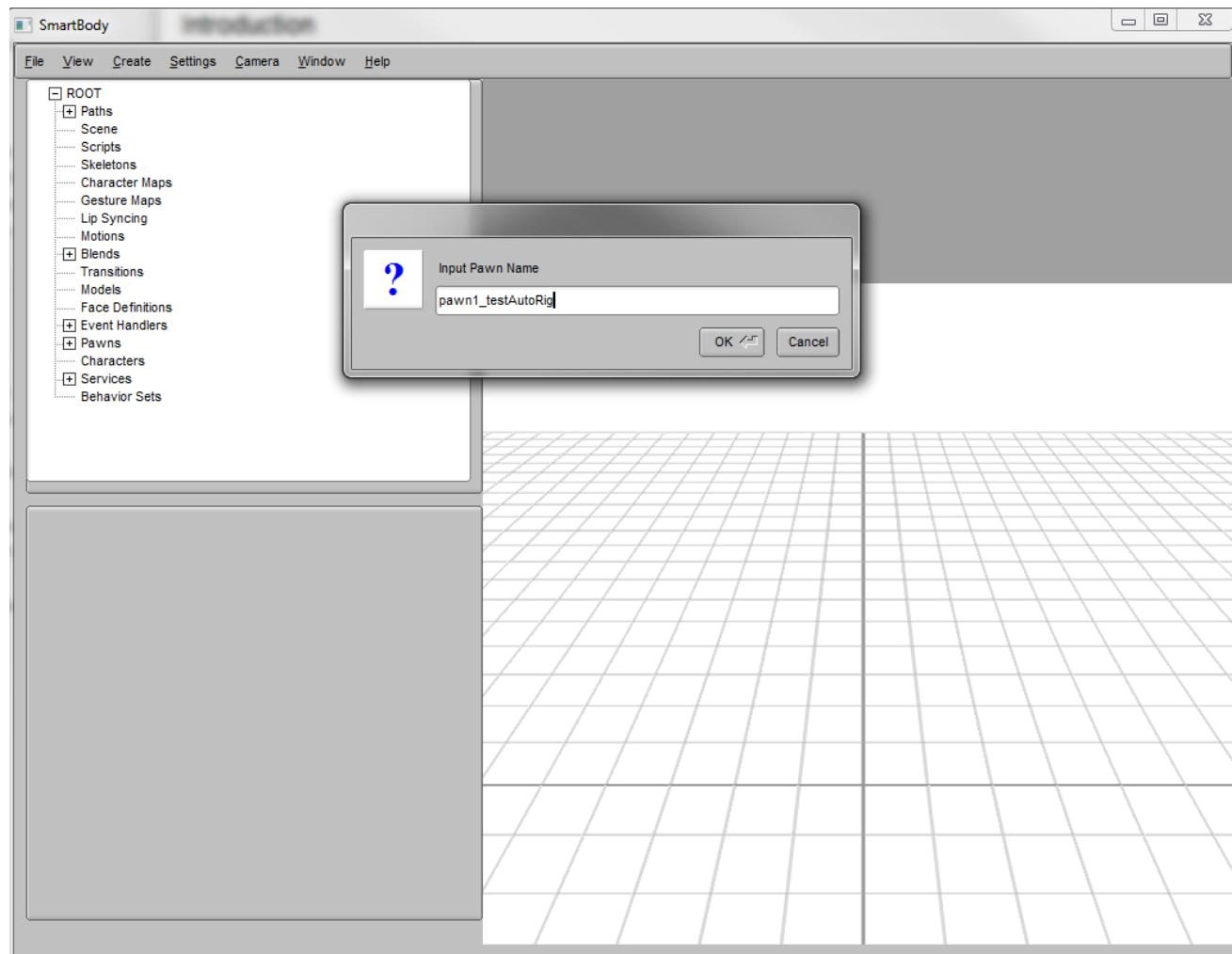
Automatic Rigging

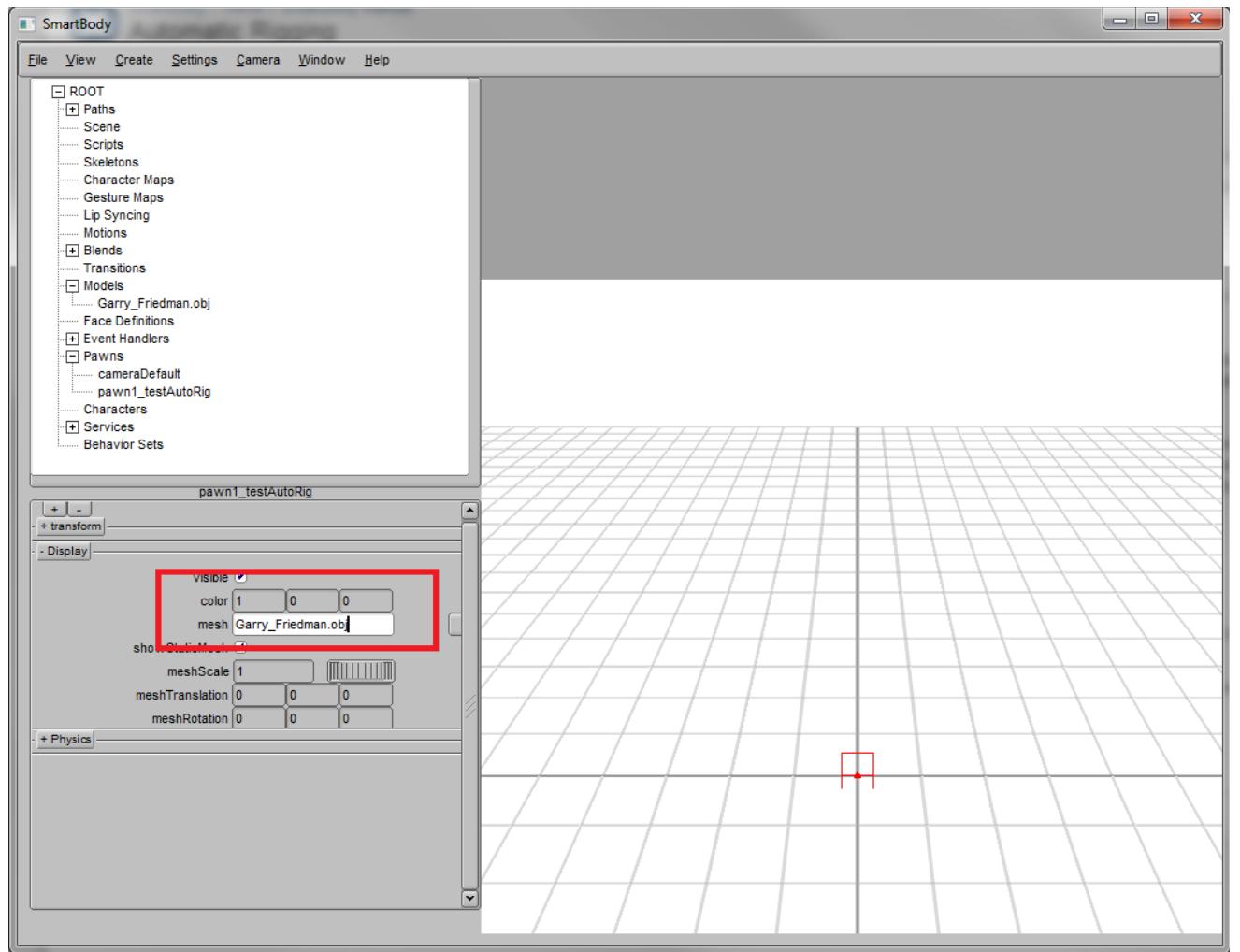
Introduction

Rigging is the process of aligning a skeleton structure into a 3D model and assigning the suitable skinning weights for each vertex. It is a time consuming process even by an experienced animator. Although sometimes it is possible to obtain a rigged model to use in SmartBody, most of the time only 3D models could be obtained either from the web or via 3D scans. Therefore SmartBody provides an experimental auto-rigging functionality into SBGUI tool. The user can input a human-like 3D model and have it immediately animated in SmartBody environment. To achieve auto-rigging, SmartBody integrates Pinocchio (<http://www.mit.edu/~ibaran/autorig/pinocchio.html>) as part of SBGUI tool.

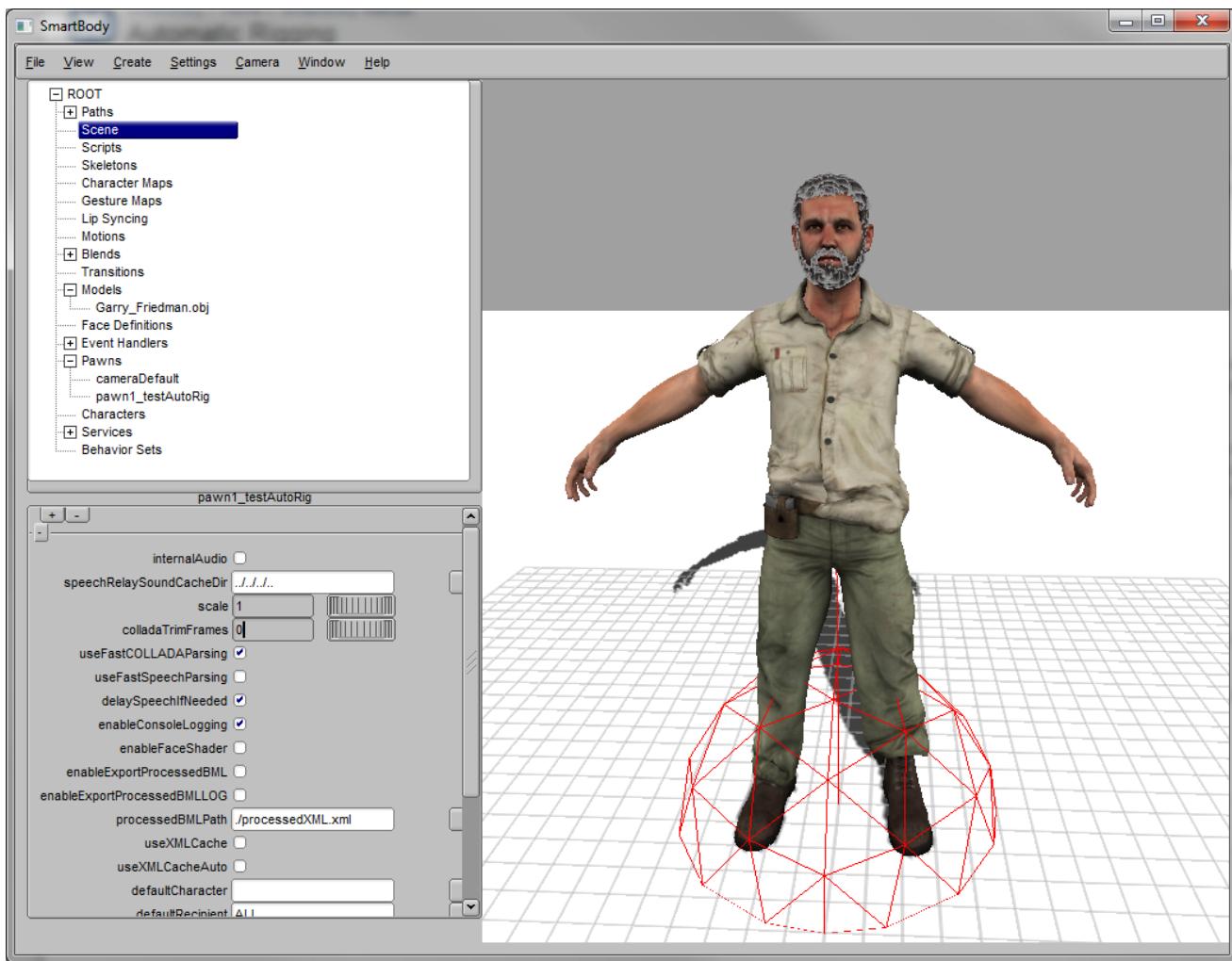
How to Use

First, we need to create a pawn with 3D mesh in the scene. Select **Create->Pawn** in the SBGUI menu to create an empty pawn. Then in Resource Panel, select the newly created pawn and set its attribute "**mesh**" to the correct mesh asset in the system.

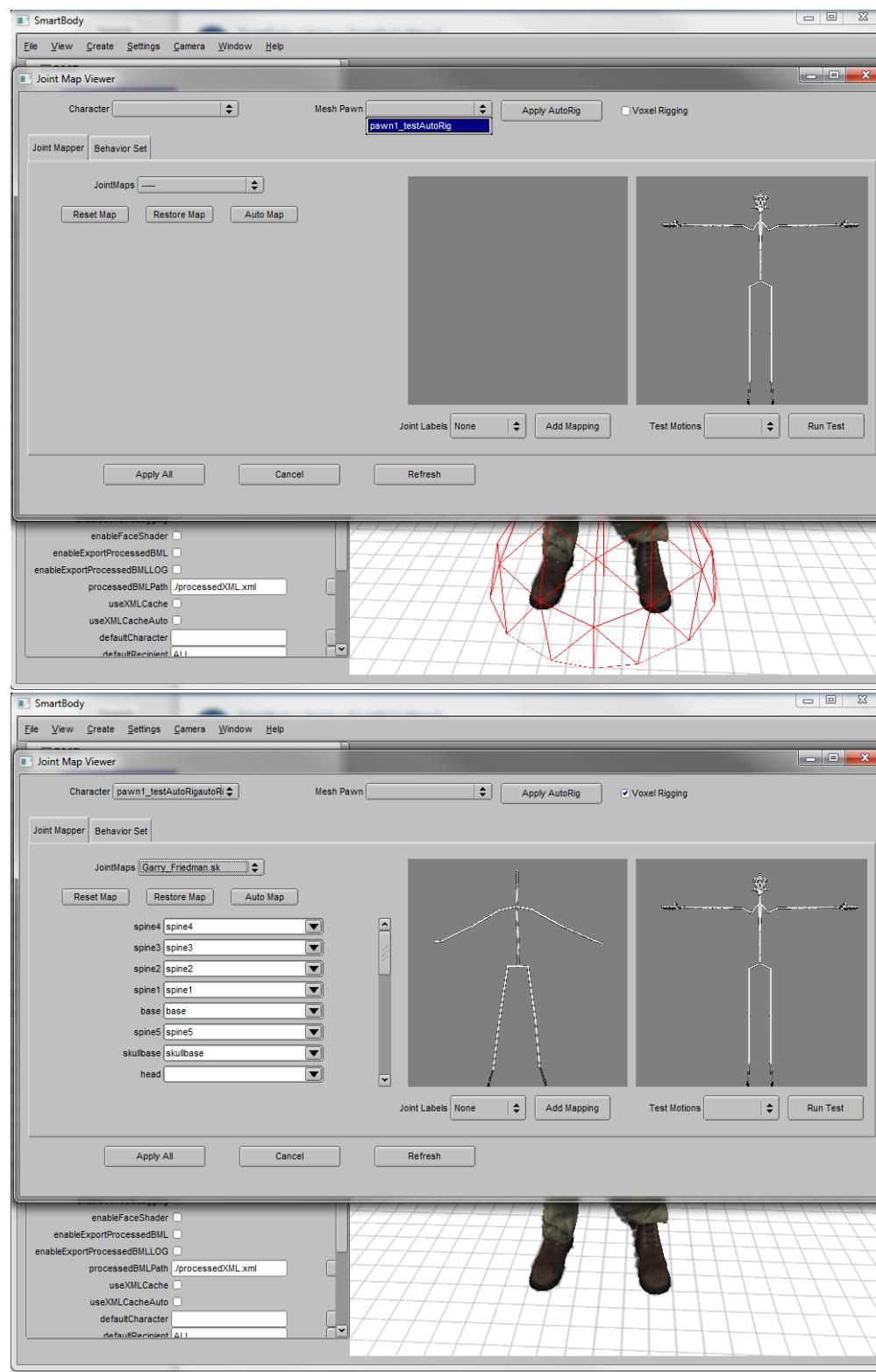


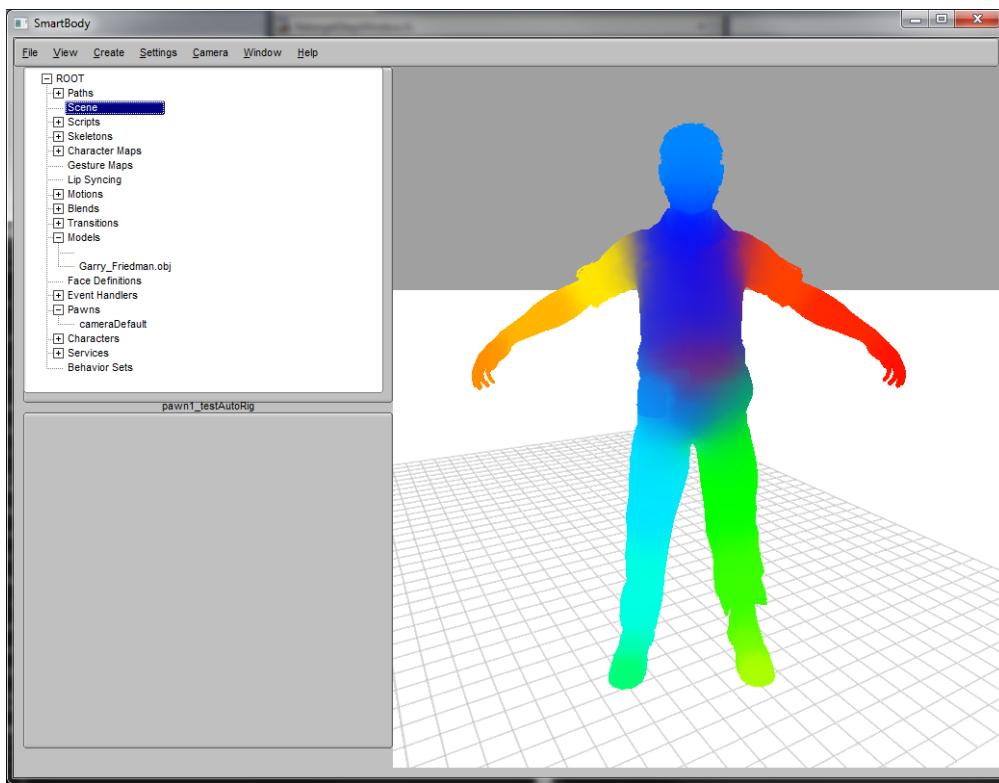


The model then will be shown on the screen. However, since this is only a static mesh without any skeleton or skinning weight information, we need to apply auto-rigging process to synthesize its skeleton and skin bindings.

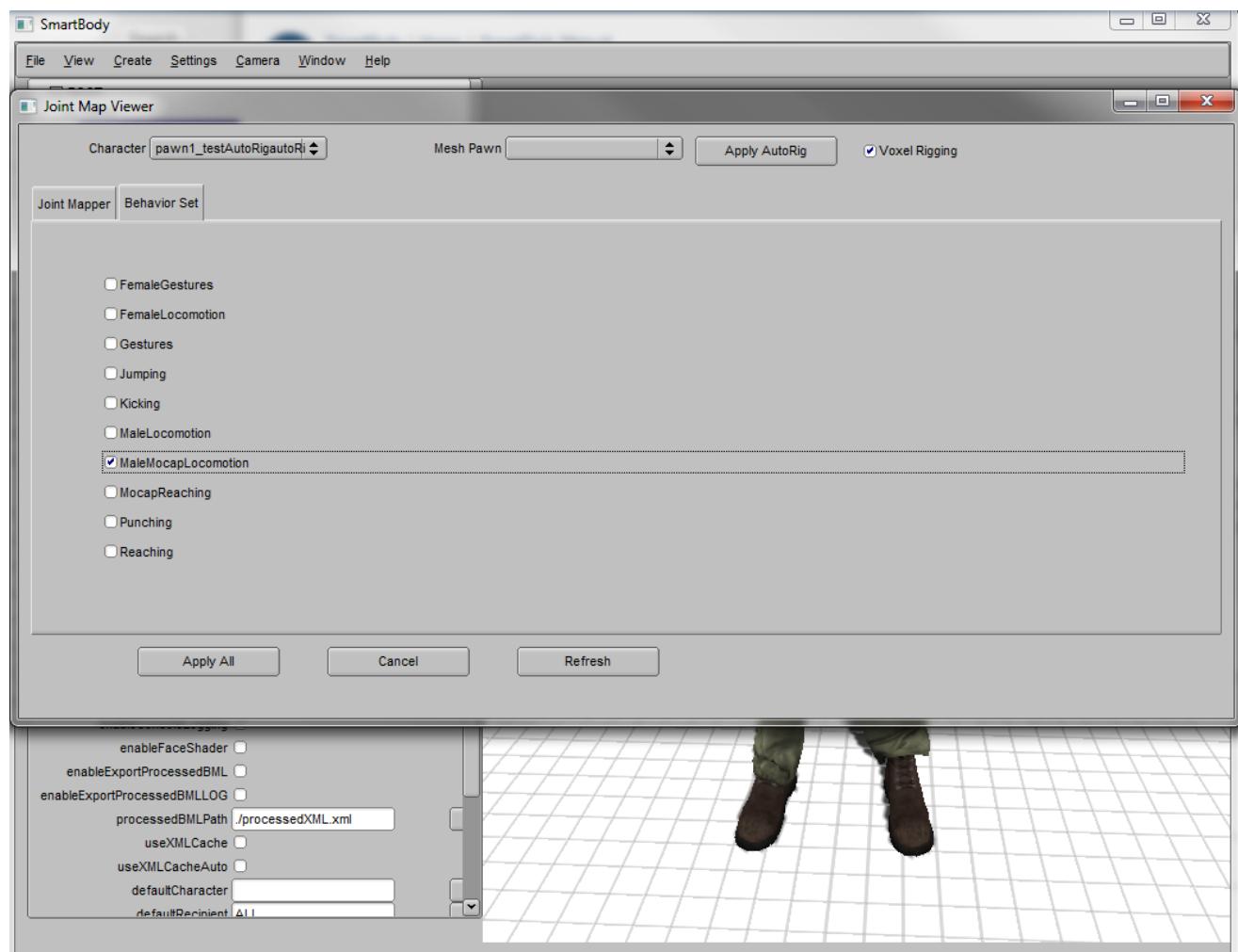


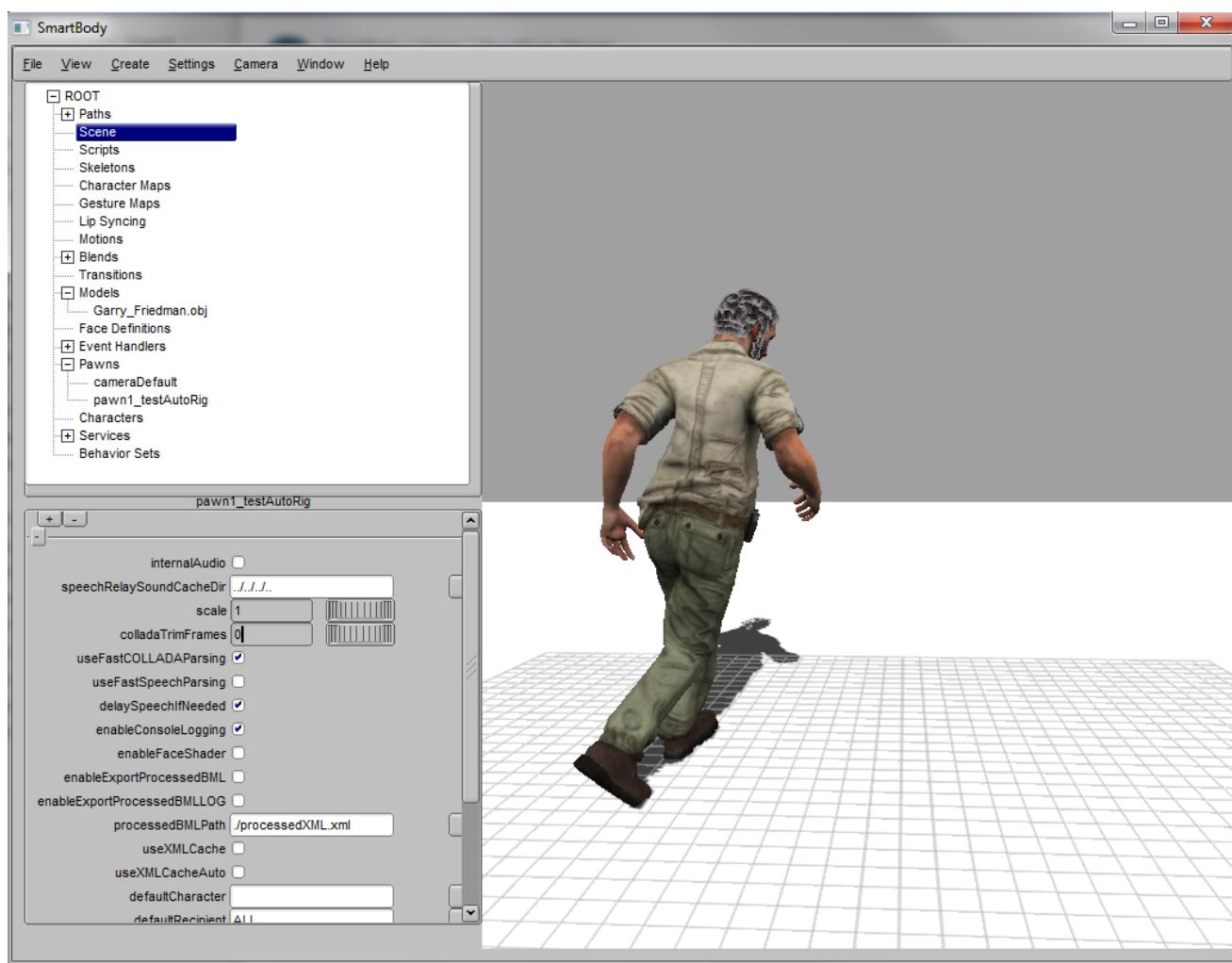
To achieve this, we can open the Retarget Viewer in **Window->Retarget Viewer**. It will show a list of pawns on the upper-right. Select the desired pawn and choose AutoRigging to start the rigging process. The auto-rigging process may take about 10~30 seconds to complete. Once it is done, a new character will be created with newly synthesized skeleton in the viewer. The new character's skin weight can also be visualized by selecting **View->Character->Skin Weight**.





Once we produced the skinned mesh, The joint name mapping and the retarget process can then be applied on this character to produce an animated character.





Note : The original Pinocchio algorithm works only for water-tight and single component mesh. However, a lot of 3D models used in production may contain either holes or additional mesh components for hairs or props. Thus it is difficult to produce a model that can work out-of-box in Pinocchio. To alleviate this problem, SmartBody provides a voxel-rigging option to pre-process the input 3D model into voxel representation. The voxel model is used for auto-rigging in place of the original mesh to ensure a watertight and connected model is used in Pinocchio. The rigging results from Pinocchio are then mapped back to the original model to complete the auto-rigging. This option is more time consuming but can work for most production 3D models. Thus it is recommended when the original method fails. It can be enabled by selecting "VoxelRigging" checkbox.

Limitation

Although the auto-rigging can be applied on most 3D models, there are a few cases it may fail or produce undesired results. First, it only works for human-like models with limbs proportion similar to a human. Thus for characters with very short or long limbs, the method may produce a bad skeleton fitting.

Also, the method can not deal the hollow or empty space inside the model correctly. For example, the robot model here has an arm formed by a few pipes and doesn't have a concrete arm shape. The method will have difficulty inferring the arm joints in this situations. This situation is rare but can happen in some mechanical or cartoon characters.

Motion Retargeting

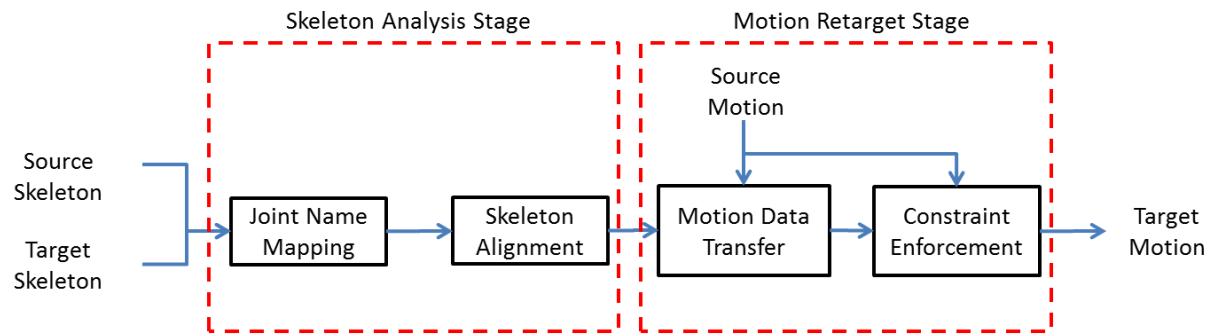
SmartBody includes functionality to retarget or adapt motion to from one character to another. This is an important feature to allow various characters to share animation originally constructed for one character with another.

How Retargeting Works

Motion retargeting is to transform a motion that is originally created for a character to apply it on a new character. The trivial way of doing this is to directly apply the motion on the new character, which means we simply copy over the joint angles in the motion to the new character. However, there are several problems with this trivial approach :

1. The joint names between two characters may be different. For example, a skeleton may have a joint name "`r_wrists`", which may be corresponding to the joint name "`RightHand`" in another skeleton. Correctly matching these joint pairs are required to copy joint angles in a motion to the proper joints.
2. Two characters may have different initial poses. For example, some characters use T-pose as their initial pose while some use A-pose. Thus the same joint rotation will have different result given different initial poses.
3. The corresponding joints between two characters may have different local rotation frame. For example, "*rotate the shoulder joint about X-axis for 90 degrees*" may move the arm up for one character, but move the arm forward for another character.
4. There may be motion artifacts such as foot sliding on the new character since two characters have different proportions and scale.

SmartBody retargeting system address these issues to create a new motion that is suitable for the new character.



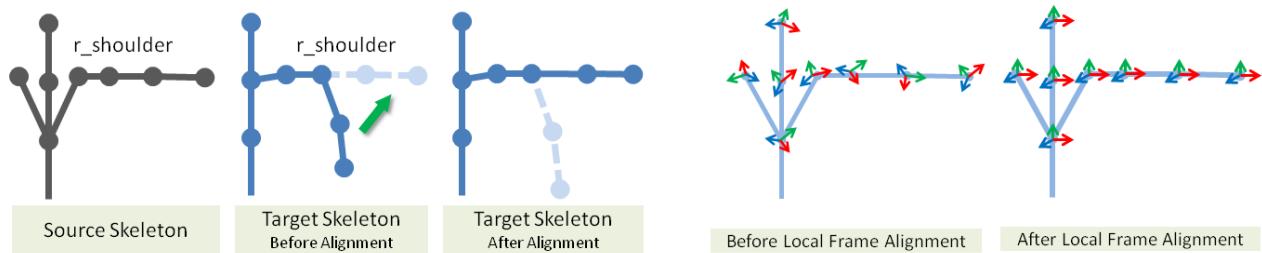
The above diagram summarizes the steps for retargeting. The system accepts a source skeleton, its corresponding source motion, and a new target skeleton as the input. Then it produces a new target motion that is suitable for the target skeleton by converting the source motion. This conversion is based on the difference between source skeleton and target skeleton.

Joint Name Mapping

Joint name mapping address the issue that two characters may have different joint names. It relies on the automatic joint name guessing to map joint names from both characters to the standard SmartBody joint names. More details can be found in [Automatic Skeleton Mapping](#). Note that our name guessing method is a heuristic algorithm using the skeleton topology and conventional joint names to infer the joint name. Therefore at times it may produce incorrect mapping results. The user can later modify the guessed mapping with either Python API or GUI.

Skeleton Alignment

Skeleton alignment address the problem that two characters may have different initial poses, and their corresponding joints could have different local rotation frames.



Our method address the first problem by aligning the rest pose between two skeleton. As shown in the image, we will recursively rotate each joint in the target skeleton to match the global bone orientation in the source skeleton. We will record the alignment rotations at each joint and apply them during retargeting to compensate for the initial pose differences.

The second problem is to compensate for different joint local frames. Our method will transform each joint local frame to the canonical frame with X, Y, Z axes point Right, Up, And Forward. This will align the joint local frames at both characters, and these alignment rotations again will be used to compensate the final joint rotation during retargeting.

The detailed formulations for these steps are outside the scope of this manual. We refer interested readers to our paper for more detail : [Automating the Transfer of a Generic Set of Behaviors Onto a Virtual Character](#).

Motion Data Transfer

Once we have all the rotation data from skeleton alignment, a new target motion can be generated for the new character. Given a source motion corresponding to the source skeleton, the system use the alignment rotations for initial pose and local frames to convert the joint rotations from source motion. A new target motion will be generated and can be readily apply on the new target character. For motions without significant leg movements (ex. gesture motions), this stage is usually enough to produce reasonable motion retargeting. However, for motions with leg movements (ex. locomotion), a constraint enforcement stage may be used to remove foot sliding artifacts.

Constraint Enforcement

Since two character may have different proportion and sizes, the same motion may look different between them. For gesture motions, this is usually acceptable. However, for motions that require more precision such as reaching and locomotion, this difference may create less accurate or lower quality animations with foot sliding. Therefore in this stage, IK is used as a post-processing to adjust the arms or legs to hit the correct trajectory in retargeted motion. The system takes the desired joints' positional trajectories (ex. ankle joints for locomotion), and scale them according to the size difference between two characters. These new trajectories are then used as IK target to adjust the retargeted motions. This post-processing will produce a new motion with its important joints following the correct trajectory like original motion.

Off-Line Retargeting

Offline retargeting converts a motion created for a source character to a new motion suitable for the new target character. The resulting motion can be saved to a motion file so it can be used natively for the new character.

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. *srcSkelName* 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 *retargetMotionWithGuessMap*. *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.

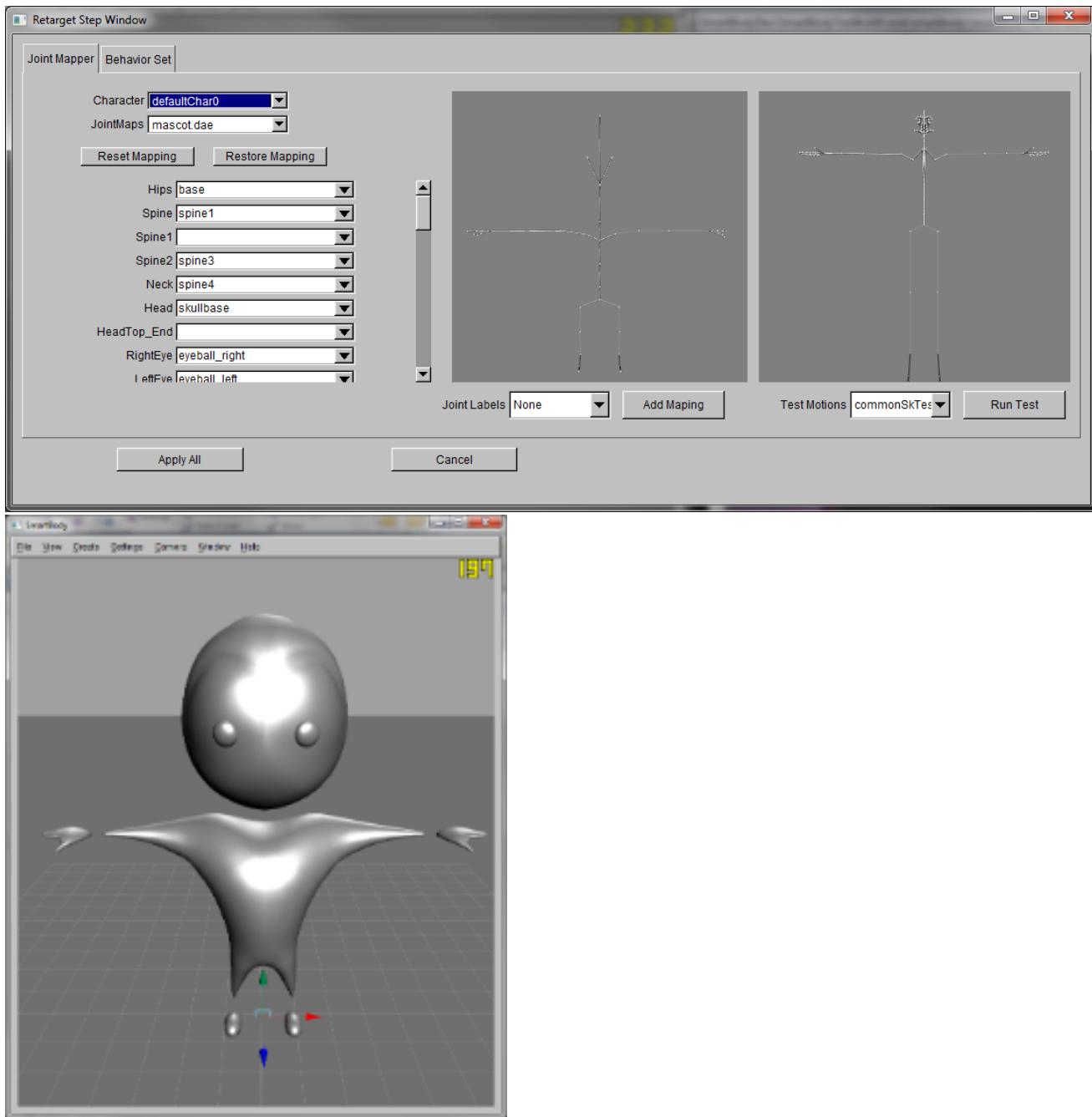
(Todo : better explanation of retargeting process in SmartBody. How to use motion.retarget and motion.constrain API to perform advanced retargeting)

On-Line Retargeting

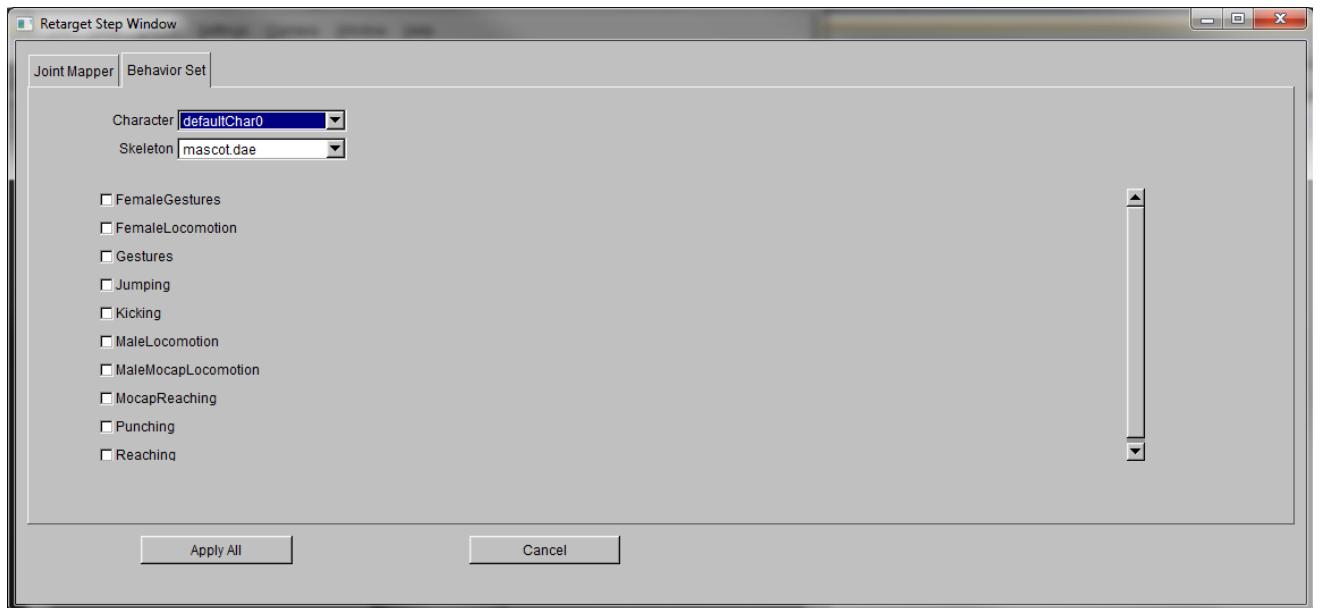
On-line retargeting does not create a new motion instance. Instead, it works as a filter when playing back motions to convert the joint angles to suitable values for the new target character on-the-fly. This feature simplifies the overhead to process each motion for each new character, but at the same time incurs some performance penalty at run-time. This also allows a rapid transfer of the default behavior sets such as locomotion, gesture, jumping, and other motions in SmartBody to a new character.

Retarget Viewer

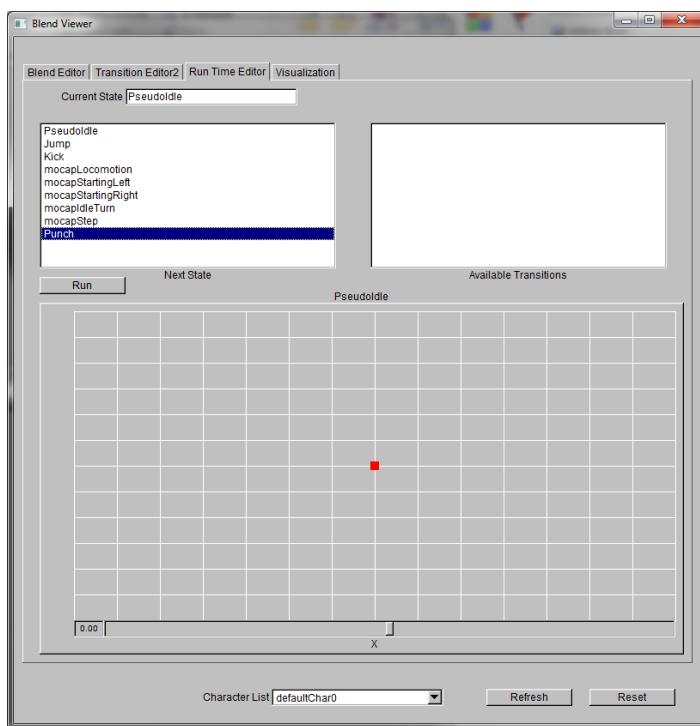
The easiest way to test out the SmartBody on-line retargeting capability on a new character is to use **Retarget Viewer**. It can be used to transfer the behavior sets in SmartBody to a custom character. It can be triggered automatically when drag-and-drop in a skeleton file (.sk for SmartBody skeleton file , or .dae for OpenCollada file). It contains two tab windows, the Joint Mapper and the Behavior Set.

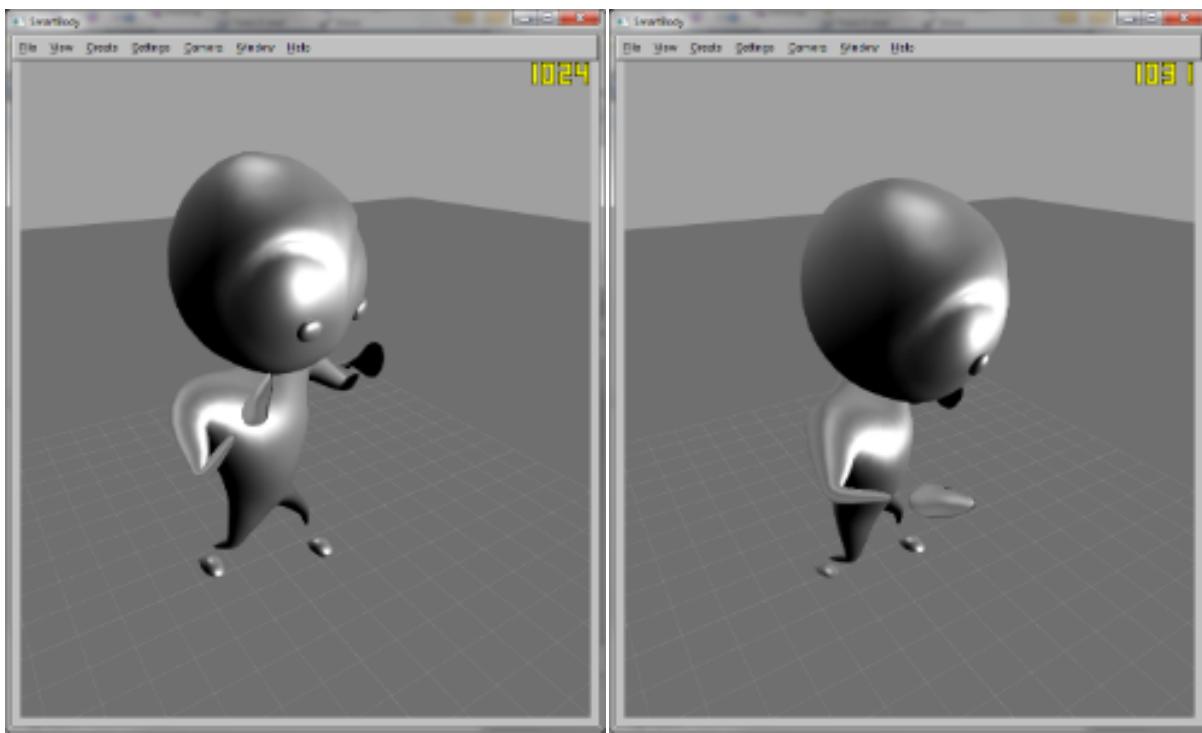


When a character first being retargeted, our system will automatically guess a correct joint name mapping to our standard skeleton. Sometimes the guessing might not get all the joint names correct, or the user may want to fine tune the joint name mapping result. The Joint Mapper let user adjust the joint name mapping result interactively. For example, the above is the screenshots from our system. The left image shows the **Joint Mapper Window**, which is used to adjust the guessed joint name mapping for '**mascot.dae**' (with its mesh shown on the right image). We can see that some joint names are already filled out automatically from joint name guessing. The user can then make further changes in the name mapping using the drop-down list for each joint. Once the joint name correspondences between the custom skeleton and our standard skeleton are established, we can move on to retarget the desired behavior animation set to the new character.



The **Behavior Set Window** let the user select the desired animation set to be used for the new custom character. The selected behavior will be automatically retargeted and applied on the new character. The following images shows the results of retargeting the "Punch" motion blend..





API Function For On-Line Retargeting

The above functionality can also be achieved by using SmartBody Python API. The following shows the steps to build the on-line retargeting so a **source motion** for a **source skeleton** can be playback normally on a new **target skeleton**.

1. We first need to find the correct joint name mapping for both the source skeleton, source motions, and target skeleton. The joint name guessing can be done using [Automatic Skeleton Mapping](#).
2. We also need to tell SmartBody which skeleton is the default skeleton for the source motion, or the motion blend that contains the motion. This way SmartBody can properly convert the source motion to use on a target skeleton based on its default skeleton.

It can be done by API call :

```
motion.setMotionSkeletonName( skelName )
```

The equivalent API call for motion blend is :

```
blend.setBlendSkeleton( skelname )
```

3. Once we have proper joint names and default skeleton setup, we can create a retarget instance between source and target skeleton. The API calls look like this :

```
retarget = retargetManager.getRetarget( srcSkelName,tgtSkelName )
retarget.initRetarget(endJoints,relativeJoints)
```

This code segment will create a retarget instance from **srcSkelName** to **tgtSkelName** and initialize it. Here, **endJoints** and their children will be skipped during the whole retargeting, and **relativeJoints** will be skipped during skeleton alignment stage. In most cases, you want to set wrist/ankle joints to be the **end joints** so the fingers and feet are not adjusted during retargeting. Also, it is usually not recommended to re-align the spine joints.

Therefore they should be put in ***relative joints***.

4. After creating the retarget instance, we can create the new character using target skeleton, and playback the source motion normally on this character. The suitable retarget instance will be used to convert joint angles automatically for the new character.

Example

The following code snippets demonstrate an example of setting up on-line retargeting.

```
# want to retarget a motion that is created for ChrBrad.sk to common.sk
srcSkelName = "ChrBrad.sk"
tgtSkelName = "common.sk"
motionName = "ChrBrad@Guitar01"

# apply zebra2 joint map for ChrBrad
scene.run("zebra2-map.py")
zebra2Map = scene.getJointMapManager().getJointMap("zebra2")
bradSkeleton = scene.getSkeleton("ChrBrad.sk")
zebra2Map.applySkeleton(bradSkeleton)
zebra2Map.applyMotionRecurse("ChrBrad")

# set the correct corresponding skeleton for Guitar motion
motion = scene.getAssetManager().getMotion(motionName)
motion.setMotionSkeletonName(srcSkelName)

# these joints and their children are not retargeted
endJoints = StringVec();
endJoints.append('l_forefoot')
endJoints.append('l_toe')
endJoints.append('l_wrists')
endJoints.append('r_forefoot')
endJoints.append('r_toe')
endJoints.append('r_wrists')
# these joints are skipped during skeleton alignment
relativeJoints = StringVec();
relativeJoints.append('spine1')
relativeJoints.append('spine2')
relativeJoints.append('spine3')
relativeJoints.append('spine4')
relativeJoints.append('spine5')
relativeJoints.append('r_sternoclavicular')
relativeJoints.append('l_sternoclavicular')
relativeJoints.append('r_acromioclavicular')
relativeJoints.append('l_acromioclavicular')

# create the retarge instance
retargetManager = scene.getRetargetManager()
retarget = retargetManager.getRetarget(srcSkelName,tgtSkelName)
if retarget == None:
    retarget = retargetManager.createRetarget(srcSkelName,tgtSkelName)
    retarget.initRetarget(endJoints,relativeJoints)
# afterward, we can use the motion 'ChrBrad@Guitar01' created for 'ChrBrad.sk' and
apply it to another character created by 'common.sk'
```

Here we have a skeleton 'ChrBrad.sk' and its corresponding motion 'ChrBrad@Guitar01', and we want to apply this motion for a different skeleton 'common.sk'. (Line 1 – 4)

In order to do this, we first need to remap the joint names for 'ChrBrad.sk' and its motions. (Line 6 – 11)

Then we also need to let SmartBody knows that the skeleton 'ChrBrad.sk' is associated with the motion 'ChrBrad@Guitar01'. (Line 14 – 15)

There are some parameters that can be adjusted for the retargeting. Here, **endJoints** and their children will be skipped during the whole retargeting, and **relativeJoints** will be skipped during skeleton alignment stage. We found that by setting wrist/foot as end joints and spinal joints as relative joints give good results in most cases. (Line 17 – 35)

Once we have everything set up, we can go ahead to create a retarget instance from 'ChrBrad.sk' to 'common.sk'. Once this retarget instance is created, we can start apply the motion 'ChrBrad@Guitar01' normally on any characters created from skeleton 'common.sk'. The on-line retargeting will be triggered automatically to convert the joint angles so the motion looks proper on the new characters. (Line 37 – 42)

The full example script that demonstrates the above operations can be found in **/data/examples/OnLineRetargettingDemo.py**

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 SBEventHandler class as follows:

```
class MyEventHandler(SBEventHandler):
    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

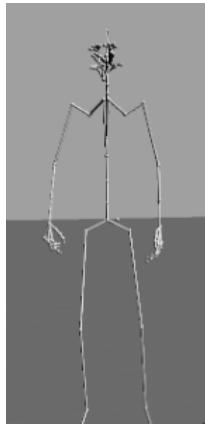
SmartBody characters can perform various skills and tasks if they are configured to do so. Some capabilities require a particular skeleton topology or joint names, while others require a set of configuration data, or motion files. For example, characters can nod, shake or move their heads sideways, but 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 <i>spine4</i> , <i>spine5</i> and <i>skullbase</i>
Gazing	Skeleton with the following joints: <i>spine1</i> , <i>spine2</i> , <i>spine3</i> , <i>spine4</i> , <i>spine5</i> , <i>skullbase</i> , <i>face_top_parent</i> , <i>eyeball_left</i> , <i>eyeball_right</i>
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: <i>eyeball_left</i> , <i>eyeball_right</i>
Grabbing, touching, reaching	Skeleton topology where shoulder names are: <i>l_sternoclavicular</i> and <i>r_sternoclavicular</i> . Also, a Reach Configuration
Constraints	None
Physics	Skeleton with root joint called <i>base</i> .

Softeyes	Skeleton with the following joints: <i>eyeball_left</i> , <i>eyeball_right</i>
Blinking	Either a Face Definition that includes a viseme called <i>blink</i> , 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, .asd/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 verify 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 exist 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")
```

Where '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:

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 map 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** contain 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.

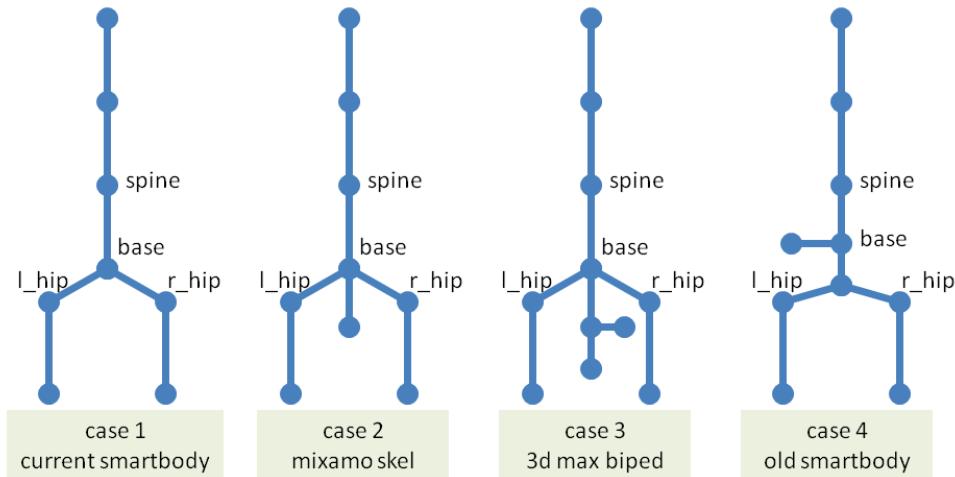
Automatic Skeleton Mapping

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:

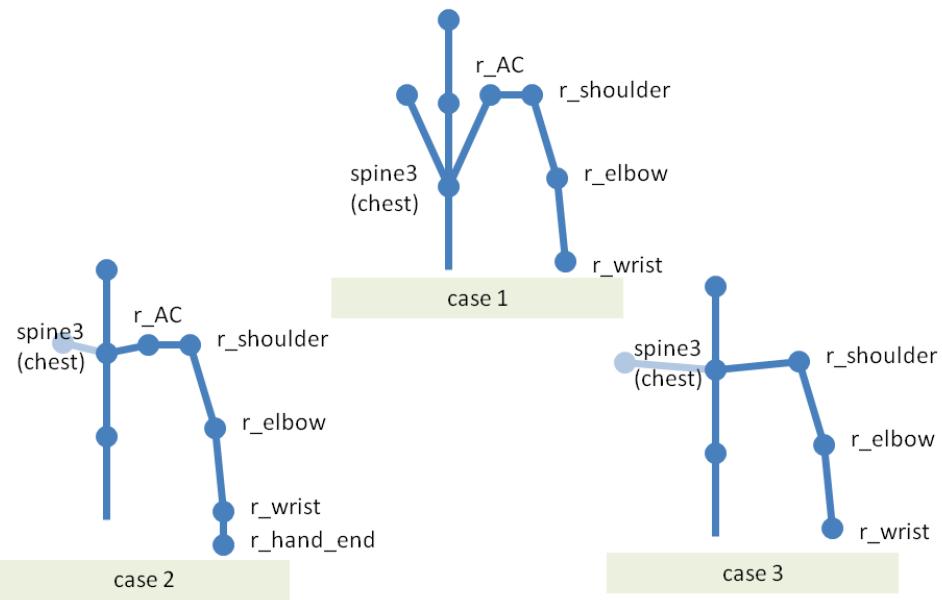
```
mycharacter = scene.getCharacter(characterName)
myskeleton = mycharacter.getSkeleton()
jointMapManager = scene.getJointMapManager()
sbJointGuessMap = jointMapManager.createJointMap(characterName+"-GuessMapping", True) # True to display
mapping, False to hide this output
sbJointGuessMap.guessMapping(myskeleton)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

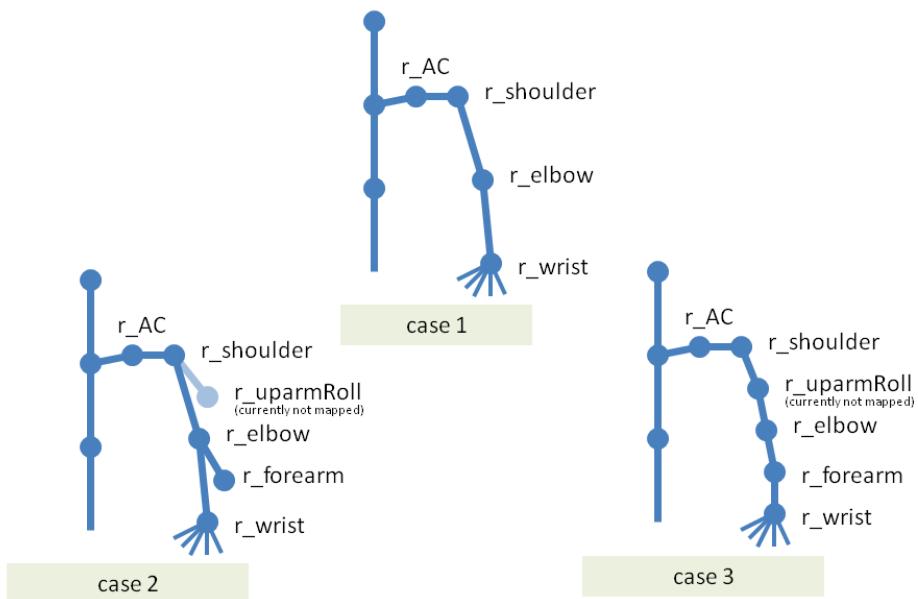
Base Joint Search



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 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, phone bigram based method, and a low-quality pose-based method.

High Quality Lip Syncing Using the Phone Bigram Method

The recommended method is to use SmartBody's phone bigram 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 phone bigram based lip syncing, load the included phone bigram animations once:

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

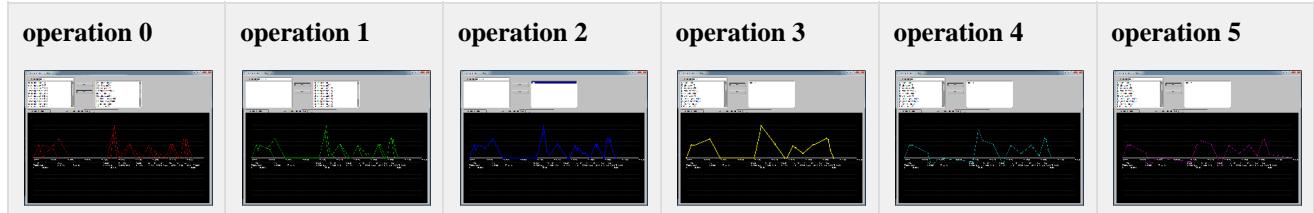
this will create the default lip sync set that can be used by any character. Then set the following attributes on your character:

```
mycharacter = scene.getCharacter(name)
mycharacter.setBoolAttribute("usePhoneBigram", True)
mycharacter.setBoolAttribute("lipSyncSplineCurve", True)
mycharacter.setDoubleAttribute("lipSyncSmoothWindow", .2)
mycharacter.setStringAttribute("lipSyncSetName", "default")
```

Offline setting up the phone bigram curves

Artist can use lip sync viewer to mark up phone bigram curves and save them to .py file.

Run time operations (behind the scene)



When you executes a speech bml command, the following operations are actually happening behind the stage (you can see more details from Lip Sync RunTime Viewer [Lip Sync Viewer](#))

- (0): layout all phone bigram curves onto the timeline.

This step will take all the artist generated phone bigram curves associated with phoneme pairs and time warp them to fit into the time span of corresponding phoneme pairs.

- (1): stitch the curves

This step stitches the overlapping curves for the same face pose.

- (2): extract the out-lope of the curve

This step take the output of step 1 and extract the out-lope of the curve, generating a complete curve for each face pose.

- (3): smooth curve given window size

This step perform a smoothing pass over each curve using a user-specified window to scan through temporal domain and find local maximas.

- (4): constrain curves given rules defined for different facial poses

Since facial poses are activated based on parametric values, any two poses could be arbitrarily combined together to form a new face shape. However, certain poses, when activated simultaneously, would produce unnatural results. For example, the face pose for 'open' should not be activated along with the pose for 'PBM'. Since one pose is open mouth and the other is close mouth, combining them together may cause interference and cancel each other. This can cause important information being eliminated from the resulting facial motion. A threshold is defined how much two poses may interfere with each other.

- (5): apply speed limit to curve slopes

To produce natural lip syncing animation, a character should only move his lips with a reasonable speed. However, the animator may create a bigram curve with high slope and the time warping could compress the curve to be excessively sharp. This step cap the parametric speed of piece wise linear curve by a threshold defined by attribute.

Attributes used for phone bigram based method

- usePhoneBigram: Bool Attribute. Toggle whether to use phone bigram based method
- lipSyncConstraint: Bool Attribute. Global toggle that set constrains on face poses.
- constrainFV: Bool Attribute. Whether to constrain face pose FV.
- constrainW: Bool Attribute. Whether to constrain face pose W.
- constrainWide: Bool Attribute. Whether to constrain face pose Wide.
- constrainPBM: Bool Attribute. Whether to constrain face pose PBM.
- constrainShCh: Bool Attribute. Whether to constrain face pose ShCh.
- openConstraintByPBM: Double Attribute. Open facial shape constraint by PBM facial shape.
- wideConstraintByPBM: Double Attribute. Wide facial shape constraint by PBM facial shape.
- shchConstraintByPBM: Double Attribute. Wide facial shape constraint by PBM facial shape.
- openConstraintByFV: Double Attribute. Open facial shape constraint by FV facial shape.
- wideConstraintByFV: Double Attribute. Wide facial shape constraint by FV facial shape.
- openConstraintByShCh: Double Attribute. Open facial shape constraint by ShCh facial shape.
- openConstraintByW: Double Attribute. Open facial shape constraint by W facial shape.
- openConstraintByWide: Double Attribute. Open facial shape constraint by Wide facial shape.
- lipSyncScale: Double Attribute. Scale factor for lip sync curves.
- lipSyncSetName: String Attribute. Name of the lip sync set to be used when using phone bigram method.
- lipSyncSplineCurve: Bool Attribute. Toggle between spline and linear curve.
- lipSyncSmoothWindow: Double Attribute. Smooth window size. If it's less than 0, don't do smooth.
- lipSyncSpeedLimit: Double Attribute. Speed Limit of facial shape movement.

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

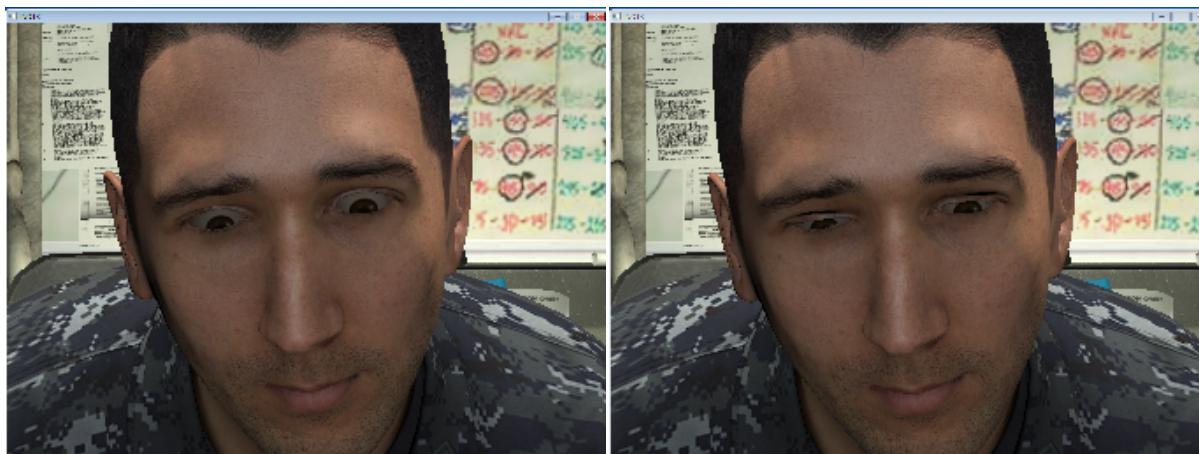
SmartBody characters will automatically blink at an interval between 4 and 8 seconds. The blinking is triggered by activating FACS units 45 LEFT and 45 RIGHT. Thus, in order for characters to blink, they must have those FACS units defined. Please consult the section on Configuring Visemes and Facial 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 'softseye' 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 softseye correction - the upper eyelid is lowered to just above the iris.

SmartBody leverages the blinking mechanism in order to adjust the eyelids. The softseye 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 softseye feature is enabled. For example, eyes that appear half-closed will only perform half-blanks, 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 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
<code>breathing.motion</code>	<p>The name of the motion to be used as an animation overlay to control breathing.</p> <p>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.</p> <p>SmartBody will modulate the effect of the breathing overlay on the character.</p> <p>Default is ""</p>
<code>breathing.useBlendChannels</code>	<p>Determines whether to add two channels, breath_x and breath_y to the character's channel array, rather than using a breathing motion overlay.</p> <p>The breathX channel indicates the respiratory volume.</p> <p>The breathY channel indicates the normalized breathing phase value (from 0 to 1).</p> <p>Default is False</p>
<code>breathing.bpm</code>	<p>Number of breaths per minute.</p> <p>Default is 15</p>

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 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.getGestureMapManager().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, lexeme, type, hand, style, posture)
```

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", "", "BOTH_HANDS", "", "HandsAtSide_Motex")
    gMap.addGestureMapping("HandsAtSide_RArm_GestureYou", "YOU", "", "RIGHT_HAND", "", "HandsAtSide_Motex")
    gMap.addGestureMapping("LHandOnHip_Arms_GestureWhy", "WHY", "", "BOTH_HANDS", "", "LHandOnHip_Motex")
    gMap.addGestureMapping("LHandOnHip_RArm_GestureOffer", "OFFER", "", "RIGHT_HAND", "", "LHandOnHip_Motex")
    gMap.addGestureMapping("LHandOnHip_RArm_SweepRight", "SWEEP", "", "RIGHT_HAND", "", "LHandOnHip_Motex")
    gMap.addGestureMapping("LHandOnHip_RArm_SweepRight", "YOU", "", "RIGHT_HAND", "", "ChrUtah_Idle003")
```

Once gesture map is setup, it can be assigned to each character by setting following attribute.

gestureMap: gesture map used under neutral emotion state, also the default one.

gestureMapEmphatic: gesture map used under angry emotion state, if not set, will use gestureMap instead.

gestureMapMeek: gesture map used under sad emotion state, if not set, will use gestureMap instead.

Gesture Motion

Sync Points

Gesture motion sync points are important for gesture holding and transition. When artist creates the gesture motion, it's preferred artist understand how these sync points are used and mark them correctly. For more detailed information, please go to BML standard 1.0 webpage.

<http://www.mindmakers.org/projects/bml-1-0/wiki#Timing-and-synchronisation>

start: start time.

ready: ready time. Gesture will get to the pose where it starts the stroke. (Also when doing pre stroke gesture hold, the inserted motion frames are referring to ready frame).

stroke_start: stroke start time. Gesture sometimes has a pre-stroke hold time period defined by time gap of ready and stroke_start.

stroke: stroke time. This is where gesture hit the emphasis point overshoot. If gesture is transition from one holding position directly to another holding position, (stroke - stroke_start) will be the transition duration. More detailed can be found in BML behaviors' gesture section.

stroke_stop: stroke end time. This is where the gesture hit the emphasis point and come back from the overshoot, if a gesture emphasis point doesn't have overshoot, then stroke is the same with stroke_stop. (Also when doing post stroke gesture hold, the inserted motion frames are referring to the stroke_stop frame).

relax: relax time. This is when gesture starts to go back to rest pose. The time gap between stroke_stop and relax defined a post-stroke holding period.

stop: end time.

Perlin Noise

Artist can add perlin noise to specific joints at given frequency and scale during gesture holding period by adding extra meta data information to the end of .skm motion files. Following are examples:

```
noise_joints: "r_shoulder r_elbow l_shoulder l_elbow"
noise_scale: 0.05
noise_frequency: 0.015
```

Gesture Co-articulation and Pruning

Gestures can be blended in at anytime indicated by BML, but under a lot of circumstances, simply blending over-generated gesture animations will look unnatural. Co-articulation is important for creating human like gesture behaviors including holding gestures, transition from one gesture's stroke end to another gesture's stroke start etc. Gesture pruning is also key element in filtering out gestures that's not fit within time constrain, e.g. blend time too short for a huge wrist spatial transition.

Several attributes will affect the co-articulation and pruning.

gesturePolicy: random or first. If set to random, gesture animation will randomly be picked from the pool, otherwise always pick the first one.

gestureRequest.autoGestureTransition: enable gesture co-articulation and pruning, otherwise will be simple blending.

gestureRequest.enableTransitionToStroke: ideally co-articulation ensures transition from one gesture's stroke end to another gesture's stroke start, because the meaning of gesture is contained from stroke start to stroke end. However, if time span for transition is real short, this attribute allows to lose the constrain by blending from stroke end to stroke directly. Trade off is the second gesture will lose the meaning from stroke start to stroke.

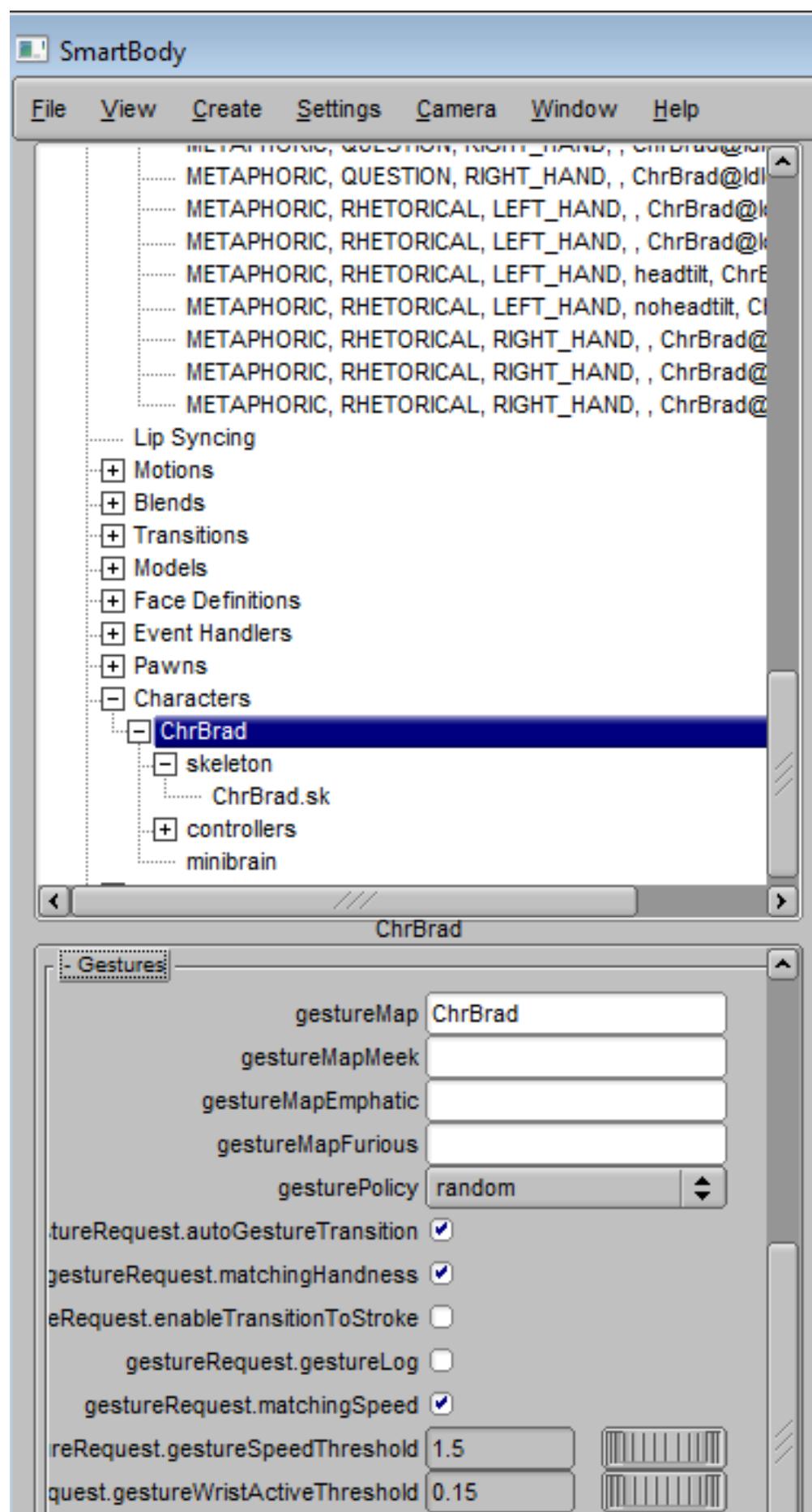
gestureRequest.gestureLog: enable log

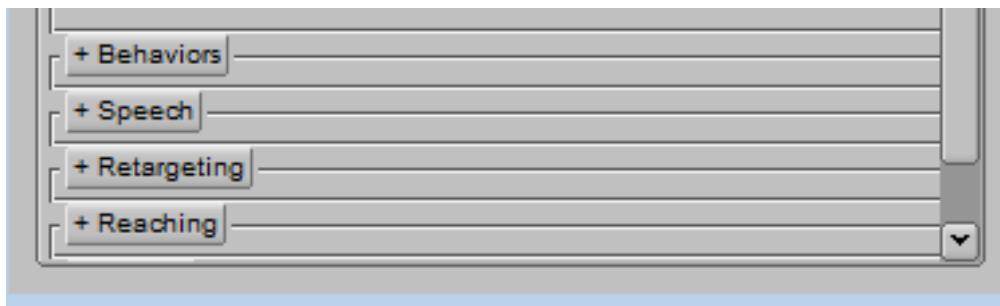
gestureRequest.gestureSpeedThreshold: default value is 1.5. If gesture's wrist joint transition speed exceeds this threshold, it will be pruned.

gestureRequest.gestureWristActiveThreshold: default is 0.15. If during transition, wrist speed is below the threshold it's considered in-active.

gestureRequest.matchingSpeed: this attribute will ensure gesture transition speed will match the speed of next gesture's stroke phase. If transition speed is smaller than next gesture's stroke speed, elongate the first gesture's holding period. If it's larger, than check if transition speed exceeds gestureSpeedThreshold.

gestureRequest.matchingHandness: if this toggle is on, gesture animation picked will take into consideration previous co-articulated gesture animation's handness and try to match it.





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/stersuite/>) 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:

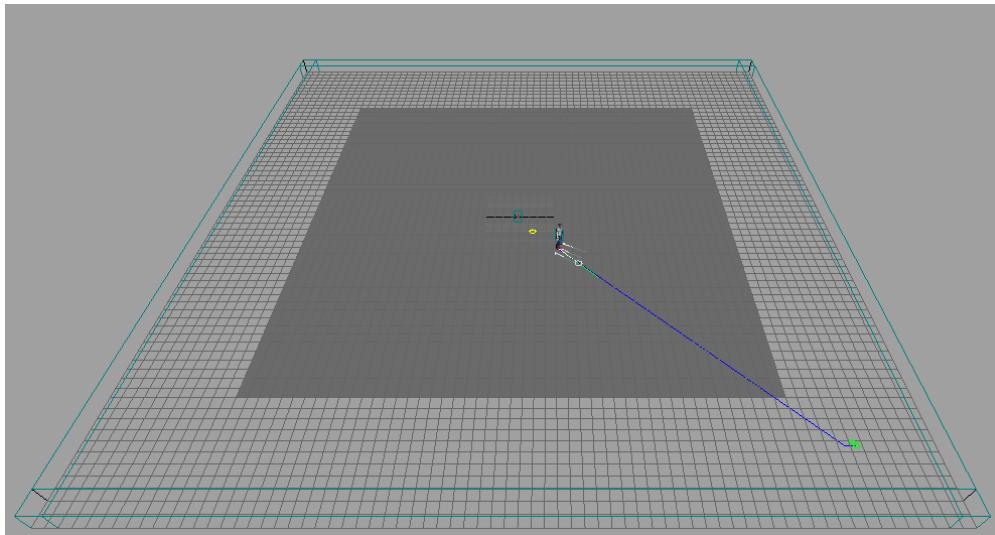
```
steer = scene.getSteerManager()
steer.setDoubleAttribute("gridDatabaseOptions.gridSizeX", 200)
steer.setDoubleAttribute("gridDatabaseOptions.gridSizeZ", 200)
steer.stop()
steer.start()
```

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.setBoolAttribute("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 closely. 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 Single-Cycle Locomotion

SmartBody characters can synthesize locomotion using only a single walking cycle file. The motion synthesis uses the [steering engine](#) as a controller for the position objective and can follow the computed path.

Setup

The locomotion system requires a retargeted motion file and footstep annotations in order to work. A number of options can be defined to modify the behaviour of the character:

Attribute	Description
walkCycle	Name of the file that contains the walking cycle
walkSkeleton	Name of the file that contains the associated skeleton of the cycle
walkScale	The scale of the motion data, usually 1 for meters and 0.01 for centimeters
walkSpeedGain	The desired speed of the motion. Default is 1 for no alteration
LEndEffectorJoint	Name of the left foot on the skeleton. It represents the joint that goes in contact with the floor while walking
REndEffectorJoint	Name of the right foot on the skeleton. It represents the joint that goes in contact with the floor while walking
CenterHipJoint	Name of the joint that connects both lower extremities. Normally is on the middle of the hips
FadeIn	Interpolation value needed for motion continuity between the fly phase and the landing phase
FadeOut	Interpolation value needed for motion continuity between the departing phase and the fly phase
TurningRate	Rate at which the character turns. If it is too high, the character legs will likely collide
footPlantRight	Foot contact information of the right foot
footPlantLeft	Foot contact information of the left foot

startsWithRight	True if the motion starts with the right foot on the floor, false if it starts with the left one. Default is true
-----------------	---

After the parameters are set, the system can start to generate locomotion, a possible configuration looks like this:

```
brad.setDoubleAttribute("walkSpeedGain", .8)
brad.setStringAttribute("CenterHipJoint", "JtPelvis")
brad.setStringAttribute("footPlantLeft", "10,22")
brad.setStringAttribute("footPlantRight", "0,6,29,36")
brad.setStringAttribute("walkCycle", "ChrBrad@Walk01B")
brad.setStringAttribute("walkSkeleton", "ChrBrad.sk")
```

The motion file needs to self concatenate. In order to achieve this, it runs through a smoothing process. It is assumed that the file provided was already segmented and has the following characteristics:

- It starts and ends on a similar pose, this pose is not repeated among the cycle.
- The cycle contains 2 full steps.
- The initial pose starts(ends) with one foot on full contact with the ground and one flying on the air.
- Both feet have a minimum distance at the start(end) of the cycle.





The picture on the left presents the starting-and ending- pose. The one on the right is when when foot contact starts for the left foot and ends for the right foot

Foot contact

The foot contact information assumes the walking cycle start with the right foot on the ground, if not, the motion is mirrored so it starts that way. Given that you will see a motion that starts with the right foot in contact with the floor anyway, the amount of information on *footPlantRight* and *footPlantLeft* will be the same regardless of the foot that starts in contact with the ground.

footPlantRight: It contains the information of the foot that starts in contact with the ground. It has to be a list of four integers:

1. It is 0 and represents the first frame of the motion in contact with the floor.
2. Is the first frame that is not on full contact with the floor, the toes can still be in contact.
3. Is the first frame greater than the previous value that has a complete foot contact with the floor.
4. Is the last frame of the cycle, the foot should be in contact with the floor at this moment.

footPlantLeft: It contains the information of the foot that starts and end on the air in the cycle. It has to be a list of two integers

1. Is the first frame that is on full contact with the floor.
2. Is the first frame greater than the previous value that is not on complete contact with the floor.

Limitations

Due to the nature of the controller and its low requirements, some limitations are encountered. The first one is that the style is fixed by the cycle, that limits considerably the speed to which it can deform without introducing artifacts. A second limitation is that it doesn't blend in or out of an idle position, if it begins to walk, there will be no motion connecting the cycle and the idle pose.

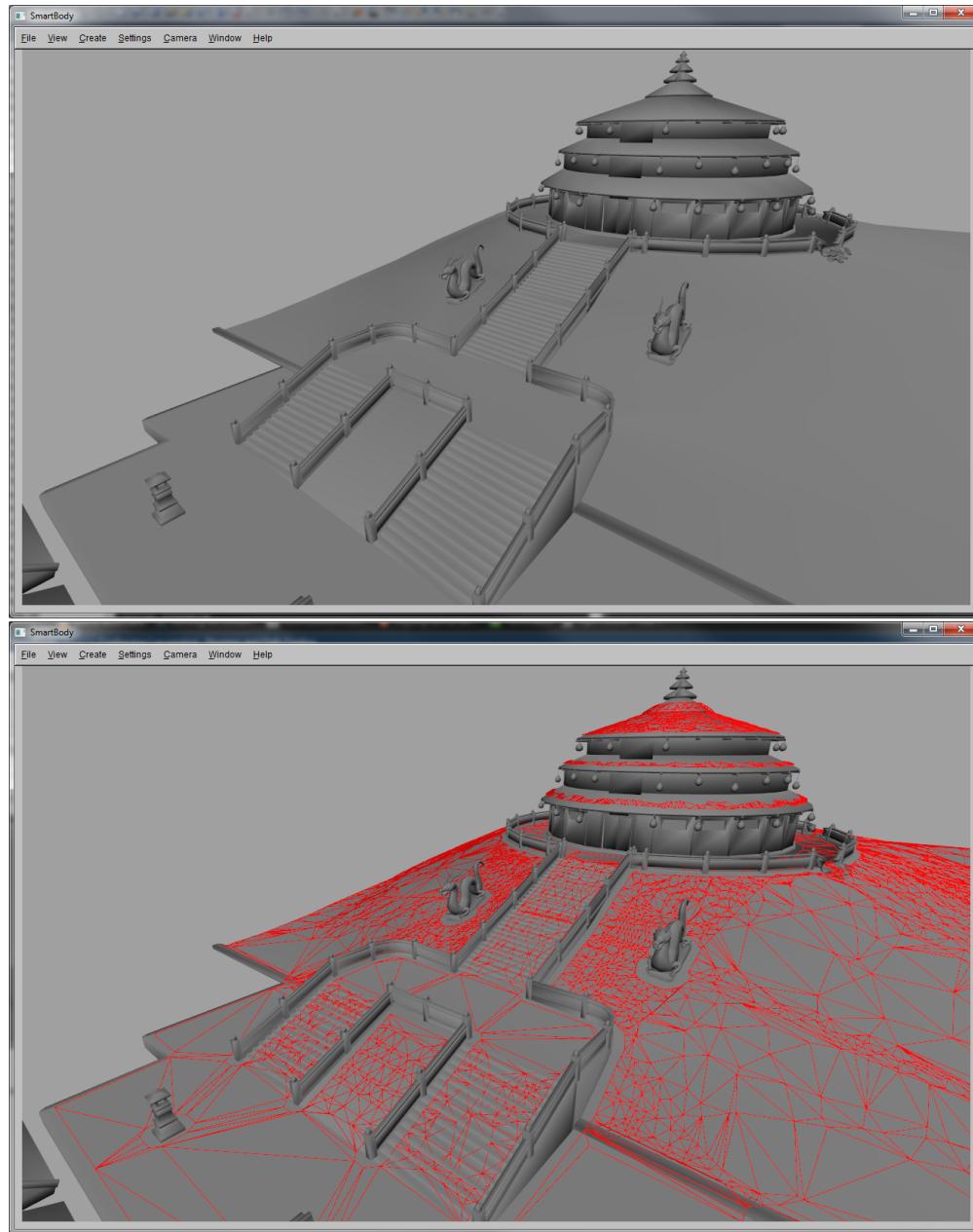
Regardless of the limitations is suitable to mention that this controller relies only on one motion file while the example one needs 20, each one with specific characteristics.

Pathfinding with Navigation Mesh

Introduction

Navigation mesh provides an efficient way to generate a valid path in the environment. The advantage of using navigation mesh is that it can quickly determine the "walkable" area and planning a path through the area.

The following is an example of navigation mesh, the left image is the mesh of an environment and the right image shows the navigation mesh on top of the original model.



We can see that the navigation mesh defines the walkable area along stairs but exclude other areas with obstacles.

Efficient path planning can be done on the navigation mesh (instead of the original mesh) to quickly find a valid path throughout the mesh.

SmartBody makes use of the Recast-Detour library (<https://code.google.com/p/recastnavigation/>) to build the navigation mesh and pathfinding.

Building Navigation Mesh

To create a navigation mesh in SmartBody, simply use the API call :

```
scene.createNavigationMesh(meshFileName)
```

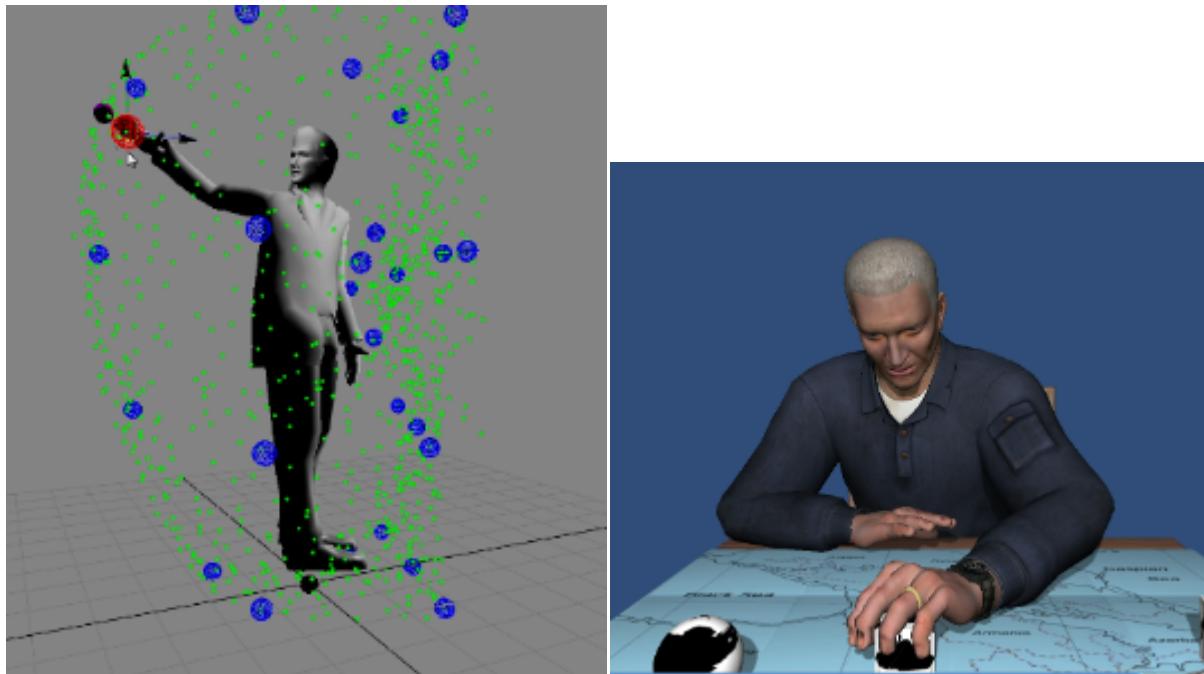
Here ***meshFileName*** is the full path to the mesh file. Currently, it only supports .OBJ file as input mesh. We will add support for other mesh file format in future updates. Once the navigation mesh is built, it can be viewed in the Viewer by selecting **View->Navigation Mesh->Show NaviMesh**.

Pathfinding in Navigation Mesh

The character can be steered in the navigation mesh environment automatically. Inside Viewer, simply right-click on the target location, and a new path along the navigation mesh will be computed automatically to steer the character.

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

Characters can synthesize speech and match that speech with behaviors from either prerecorded audio or from a text-to-speech (TTS) engine. The details of speech BML format and usage can be found at [Speech](#).

Prerecorded audio

To make a SmartBody character speak using recorded voice (via a .wav file), it is necessary to create a schedule of phonemes (individual sounds that comprise a spoken word) and the timings of each phoneme that can then be processed by SmartBody, and in turn, lip synched accurately by the character.

SmartBody uses an XML format that contains a list of words, their timings, and their phonemes. To use such a file:

- 1) Make sure the character uses 'audiofile' for its voice:

```
mycharacter = scene.getCharacter("Brad")
mycharacter.setStringAttribute("voice", "audiofile")
```

- 2) Set the main path to find the sound files (relative paths will prepend the media path):

```
scene.addAssetPath("audio", "sounds")
```

- 3) Set the specific directory for your character. If all the sound files are in the same directory, a '.' can be used:

```
mycharacter.setStringAttribute("voiceCode", ".")
```

- 4) Place the .wav file, .bml file (containing the phoneme and word schedule) into the directory

- 5) Play the sound via BML. For example, if the .wav file is called 'introduction.wav':

```
bml.execBML("Brad", "<speech ref='introduction' />")
```

CSLU Phoneme Scheduler

To generate phoneme schedules through prerecorded audio, phoneme alignment code from CSLU Toolkit (<http://www.cs.lu.ogi.edu/toolkit/>) is leveraged. You would need to download separately due to their license. Also Cygwin is needed to run the scripts. Following are the detailed instructions:

- Download CSLU Toolkit, copy /fa from CSLU_ROOT/Toolkit/2.0/apps/fa and /Tcl80 from CSLU_ROOT/Tcl80 to SB_ROOT/tools/CSLUPhonemeScheduler
- Put all the .wav files and their corresponding transcription .txt files into /files
- Run auto.sh, .bml files would be generated under /files.

Note that:

- .wav files has to be mono track, preferable 16KHZ.
- file layout can be found at SB_ROOT/tools/CSLUPhonemeScheduler/readme.txt

FaceFX Phoneme Scheduler

You can alternatively use commercial software FaceFX to generate phoneme schedules. To find how to configure FaceFX, check [FaceFX](#).

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 controllers and 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. Blends can 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 anything 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 its 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 correspondence 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 three sets 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 blend can be activated via the <blend> BML command, such as:

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	

Creating Transitions Between Blends

Characters can transition between blends by specifying a transition. A transition indicates how a character will utilize one source blend when transitioning to a destination blend. Transitions indicate moments where the source blend will be fading out, and the destination blend will start fading in. Currently, only one fade-in point can be specified on the destination blend, while multiple fade-out points can be specified on the source blend.

A transition can be specified as follows:

```
blendManager = scene.getBlendManager()
transition = blendManager.createTransition('blend1', "blend2")
transition.setEaseInInterval("motionOnBlend2", 0, 2)
transition.addEaseOutInterval("motionOnBlend1", 4, 6)
transition.addEaseOutInterval("motionOnBlend1", 7, 9)
```

This will create a transition between blends 'blend1' and 'blend2', where 'blend1' will start fading out between seconds 4 and 6 and start fading in to 'blend2' between seconds 0 and 2. If the motion in 'blend1' has already passed the 4 second mark, then the second fading out point will begin at second 7 and last until second 8.

Automatically Transitions Between Blends

Triggers can be constructed that automatically transitions between blends based on various aspects of the simulation, such as character location, attributes, objects in the scene, and so forth.

To construct such a trigger, you build a rule that is then attached to the transition:

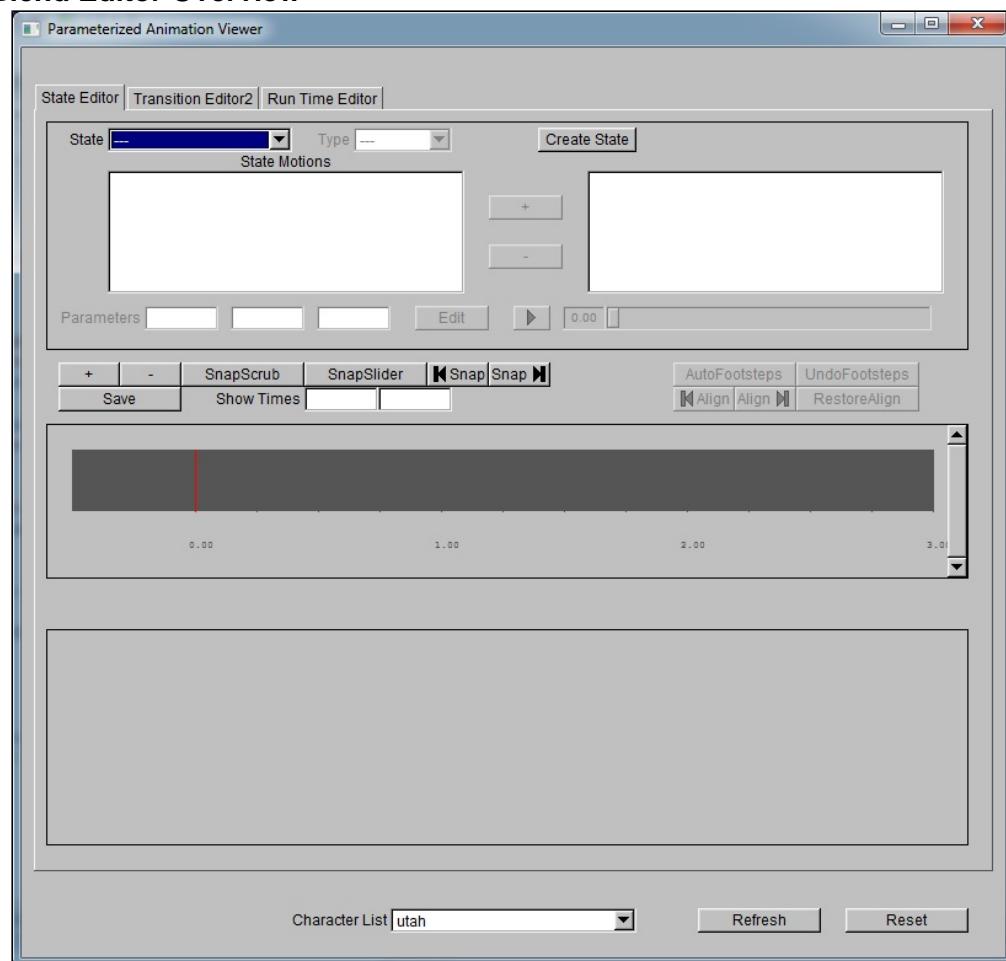
```
class MyTransitionRule(TransitionRule):
    def check(self, character, blend):
        blendManager = scene.getBlendManager()
        vec = blendManager.getCurrentBlendParameters(character.getName())
        if (vec.getData(0) < 2.0):
            return True
        else:
            return False

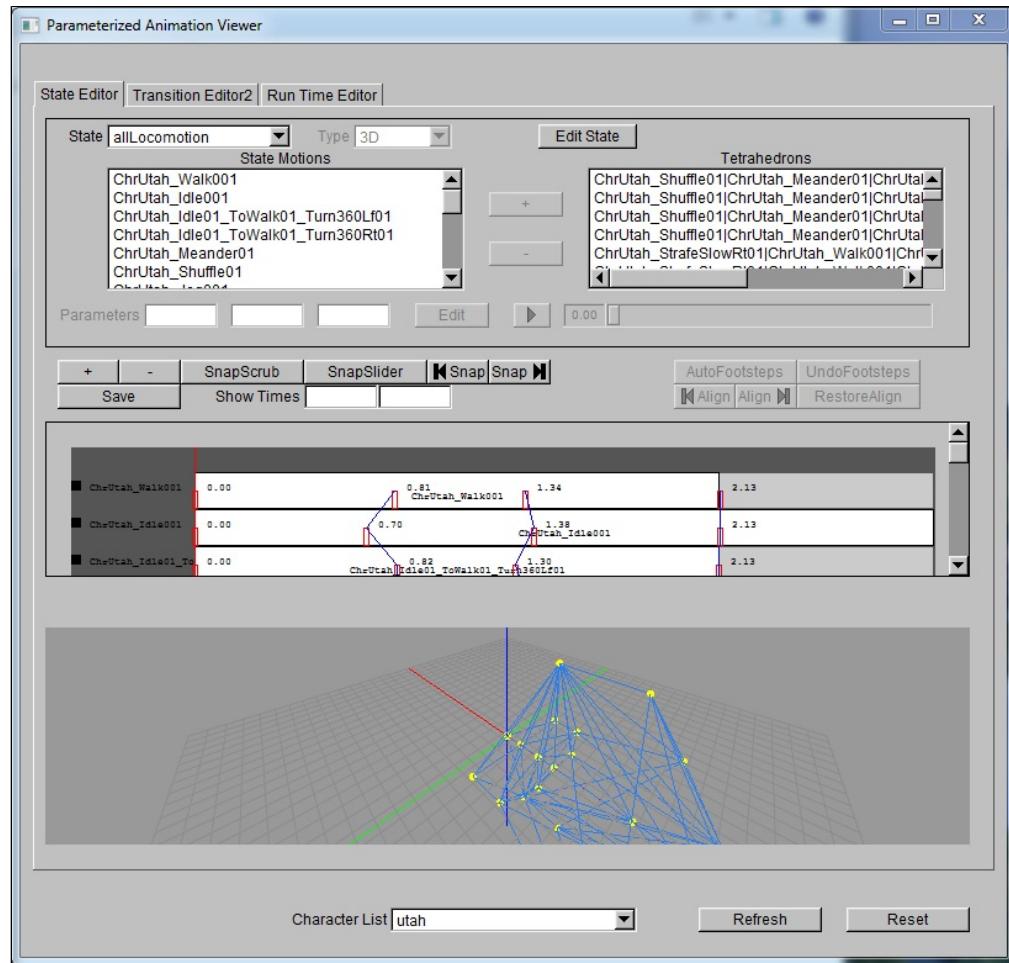
myrule = MyTransitionRule()
blendManager = scene.getBlendManager()
transition = blendManager.getTransition('blend', 'blend2')
transition.setTransitionRule(myrule)
```

In this case, the transition rule will trigger if the value of the first parameter of the blend is less than 2.

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 blend to 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 drawn if user hit Play Button. A alignment line will be drawn if user did align action.

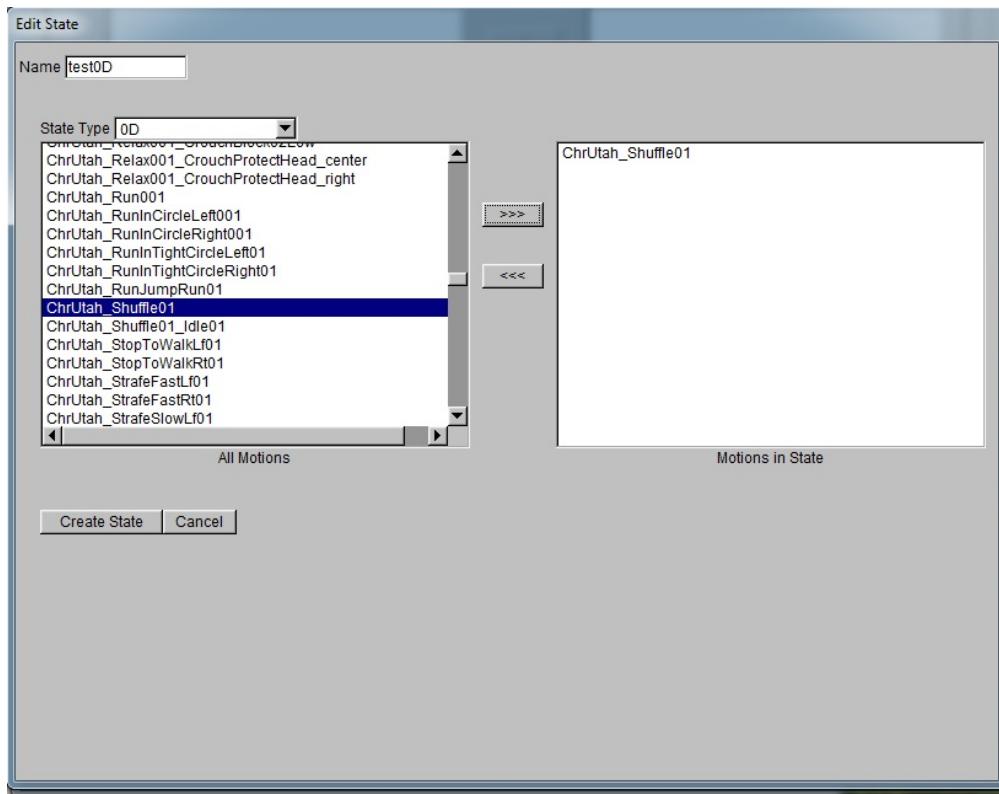
Third section, parameter space visualizer.

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

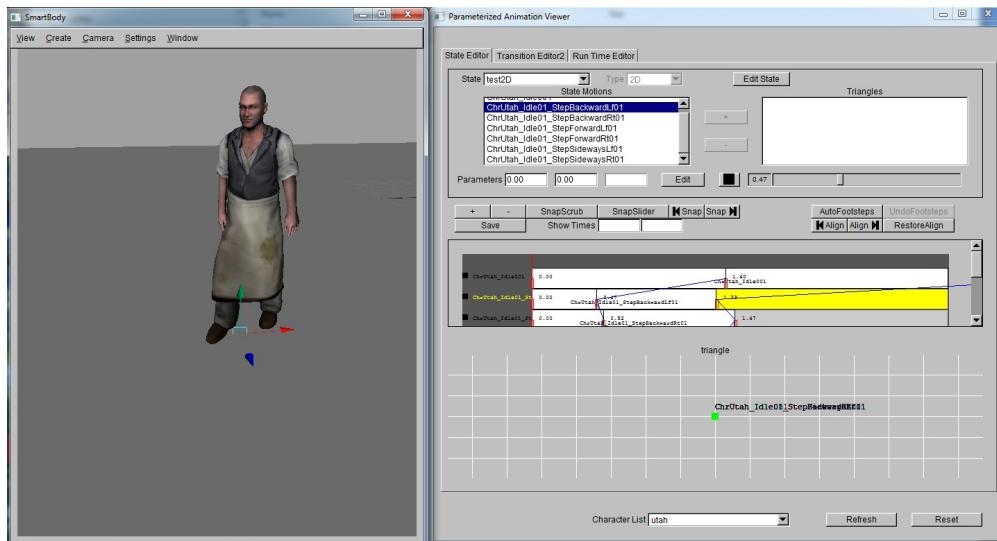
Create Blends

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.



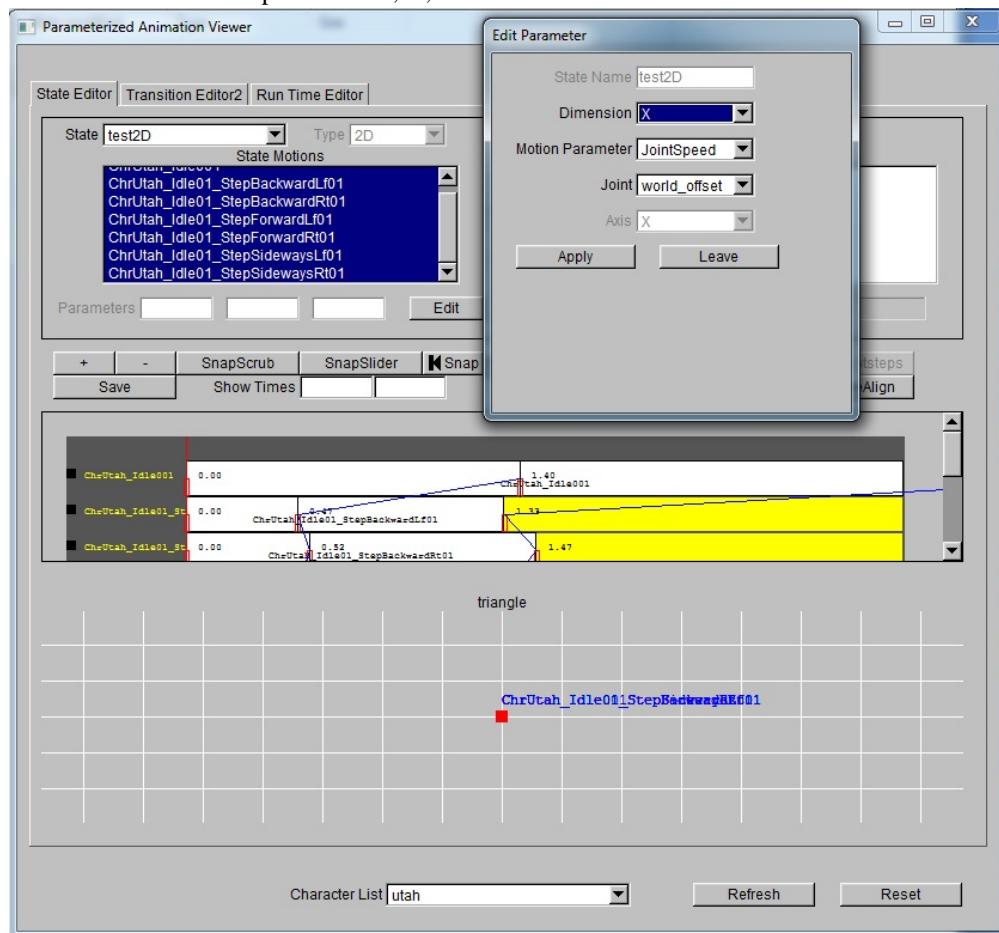
- Setup correspondence points:** By default, there would be two correspondence points added for each motion at beginning and end. User can press "+" button to add markers. By pressing Play Button, user can see how the motions looks on specific character. By dragging the frame slider, a scrub line will 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.

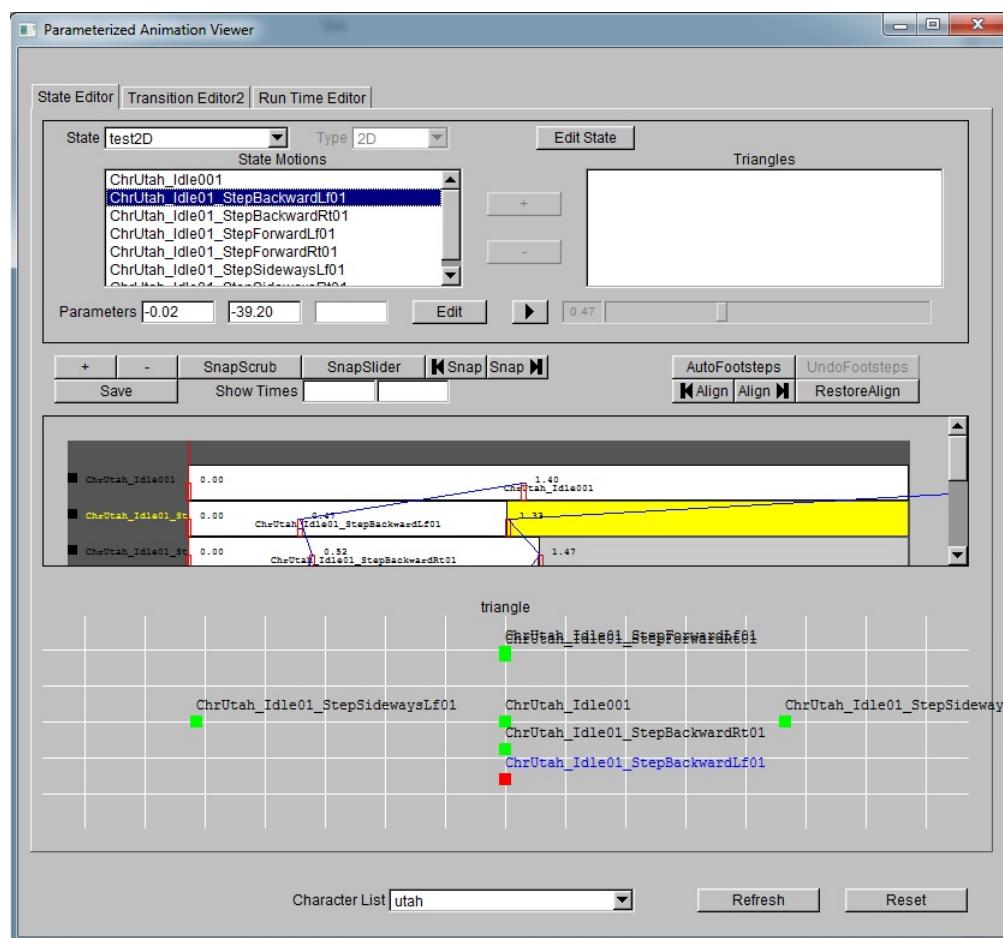


- Edit Parameters:** this step is not needed for 0D blend. User can directly modify the parameters input field or use Edit button to do batch edit. First select motions need to be batch processed. Then click Edit button. This will create a Edit Parameter window. Once hit apply apply button, new parameter 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.

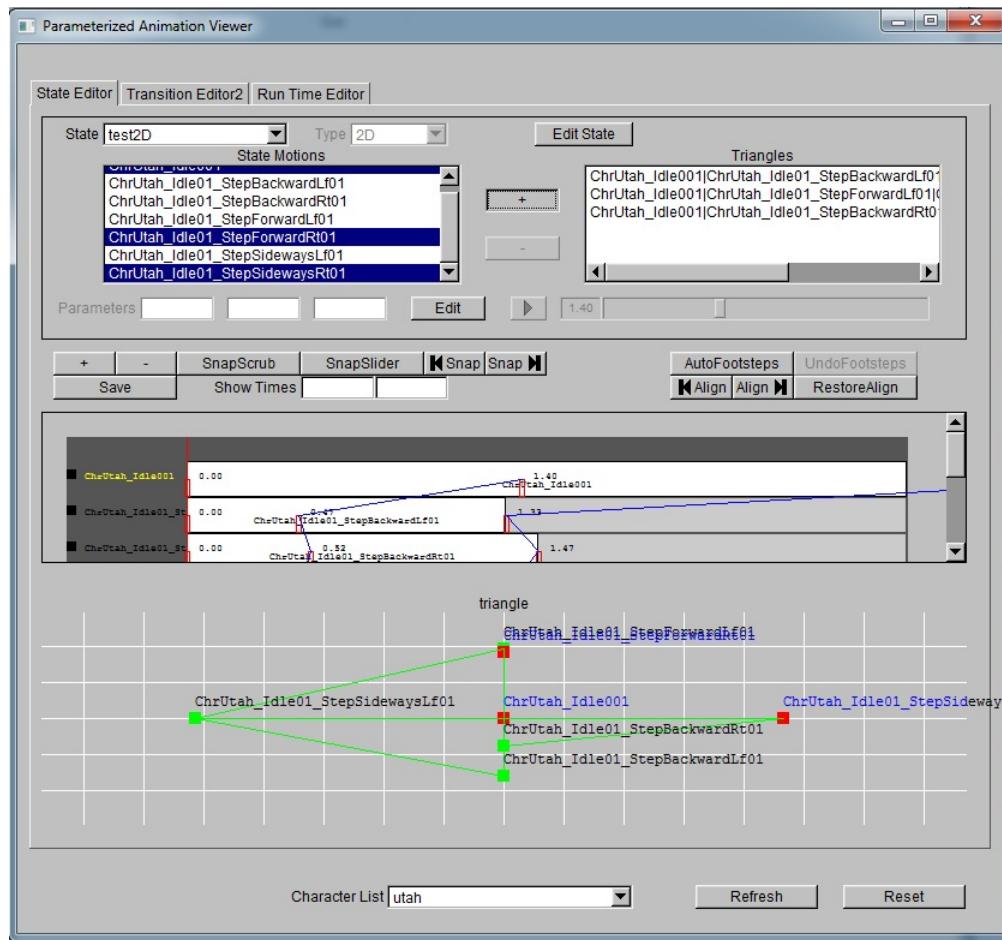
-Axis: only applies to JointSpeedAxis, JointAngularSpeedAxis and JointTransition, it defines which axis to

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





- Crafting Shapes:** This step is only needed for 2D blend and 3D blend. 2D blend requires triangles crafted and 3D blend requires 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.py](#)[test1D.py](#)[test2D.py](#)[test3D.py](#)

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

- **Motion preparation:** it includes mirroring, duplicate cycles, aligning.

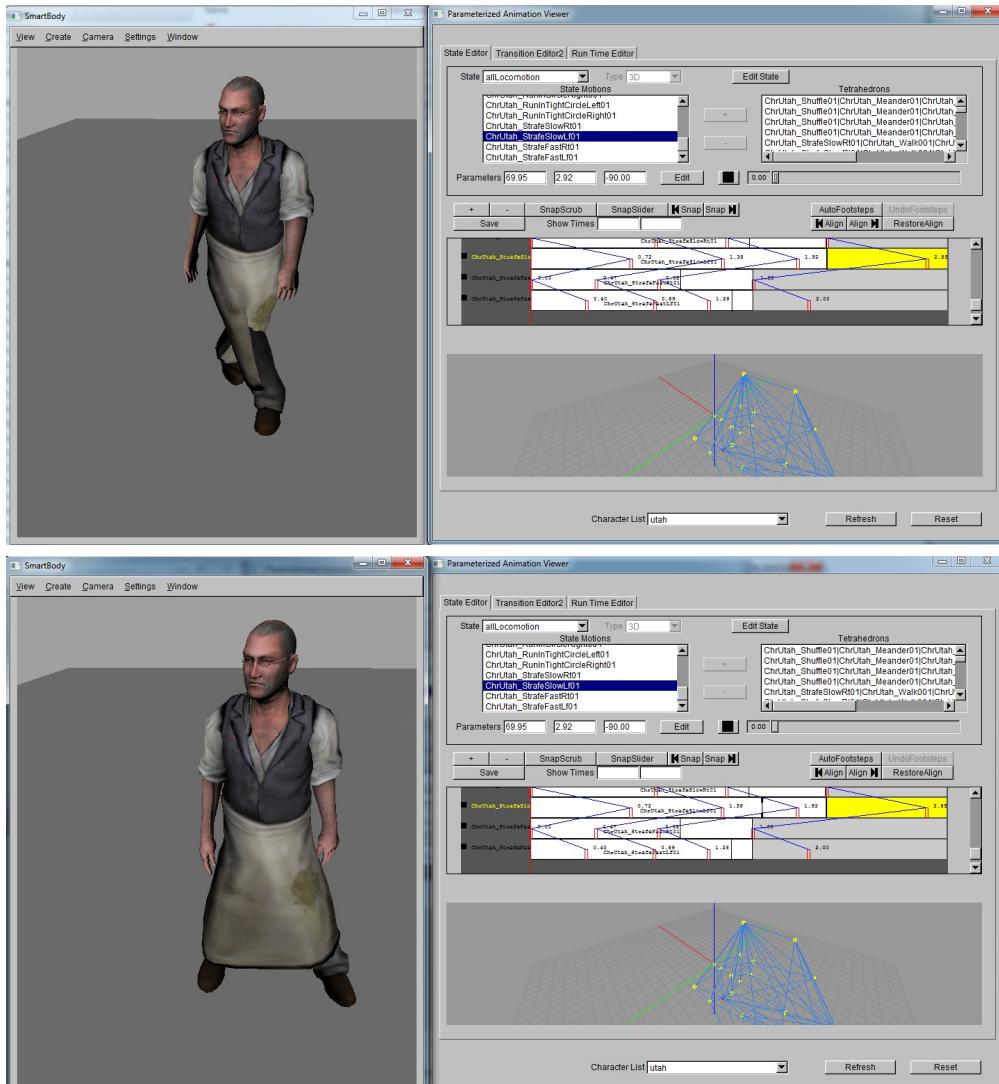
-**Mirror:** art asset sometimes only have half of the motions. Use following python command, user can easily mirror the right motion to the left motion.

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

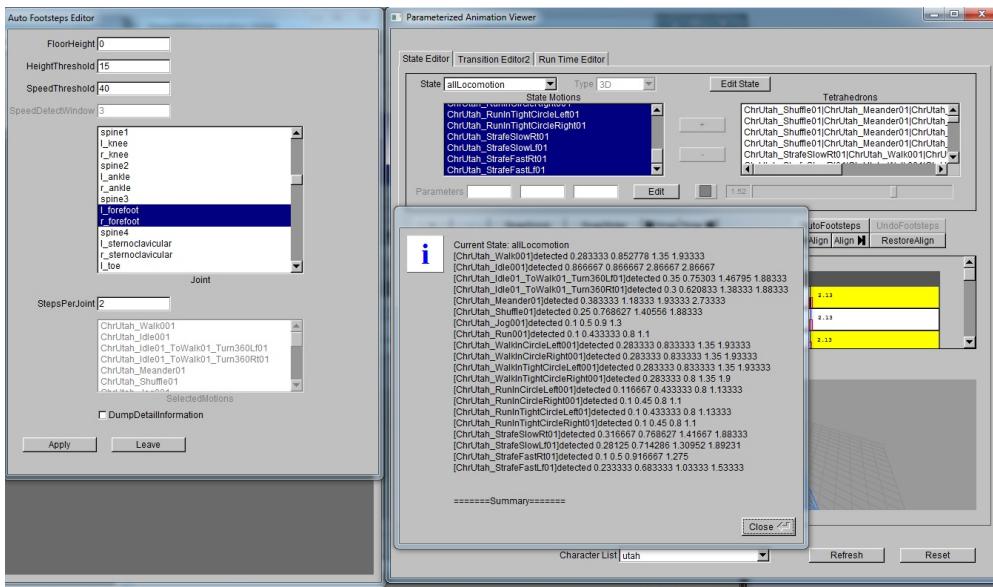
-**DuplicateCycles:** sometimes you may have walk animation with one cycle and jog animation with two cycles. To make all the animations consistent on cycles without going back to artist. User can use the following command:

```
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/s². 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 :

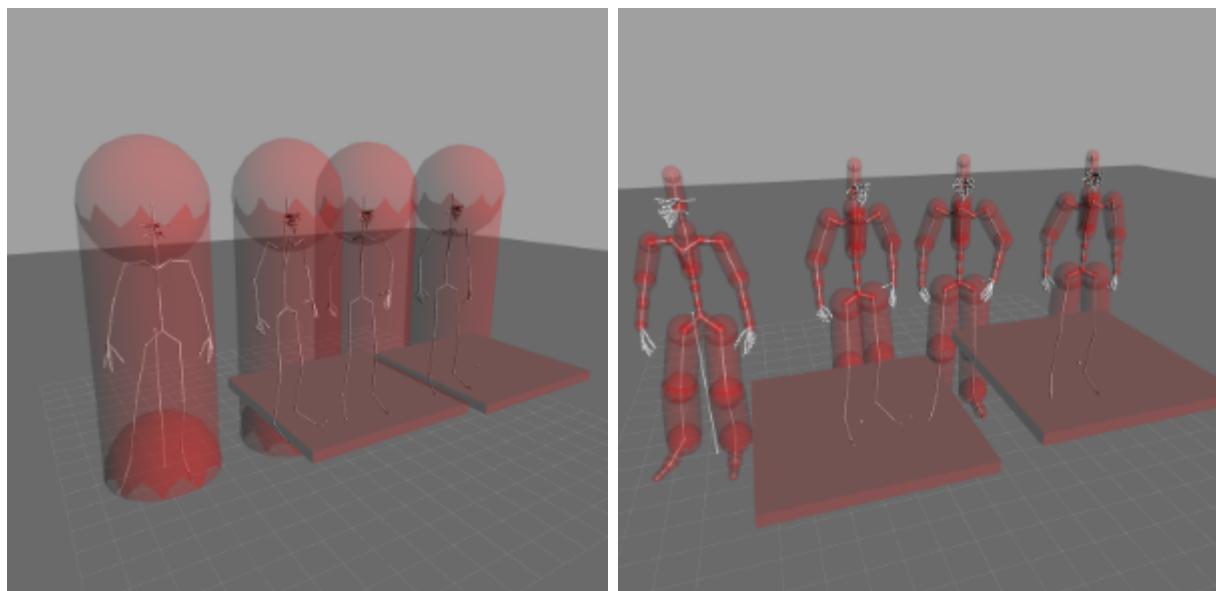
1. Select the body link to be constrained :
 - a. bodyLink = phyManager.getJointObj("charName", "jointName"); # get corresponding body link from a character
 - b. bodyLink.setBoolAttribute("constraint", true); # enable constraint
2. If we do not specified a constrain target (the target object which the 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.

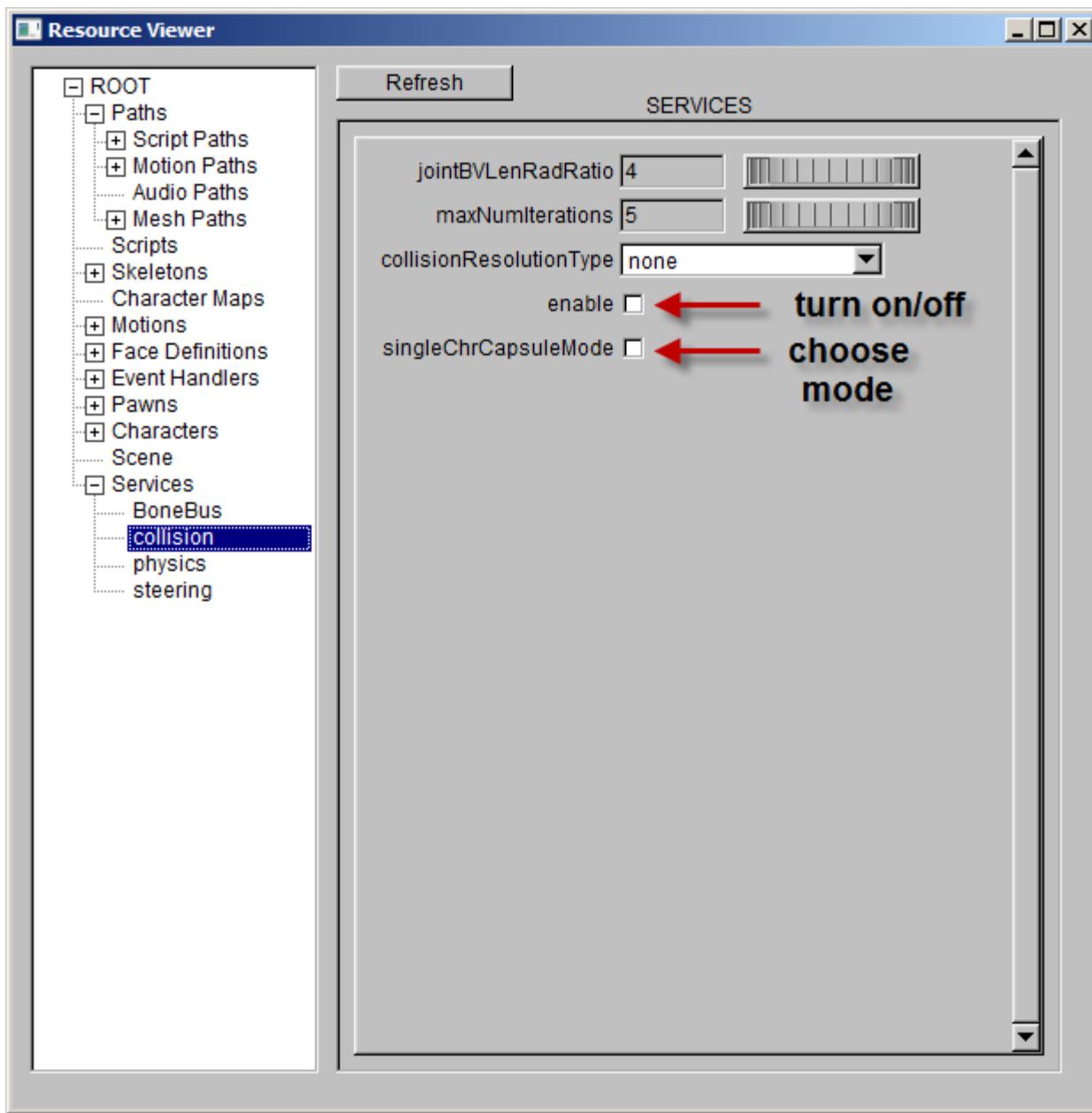
Configuring Collision Detection

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

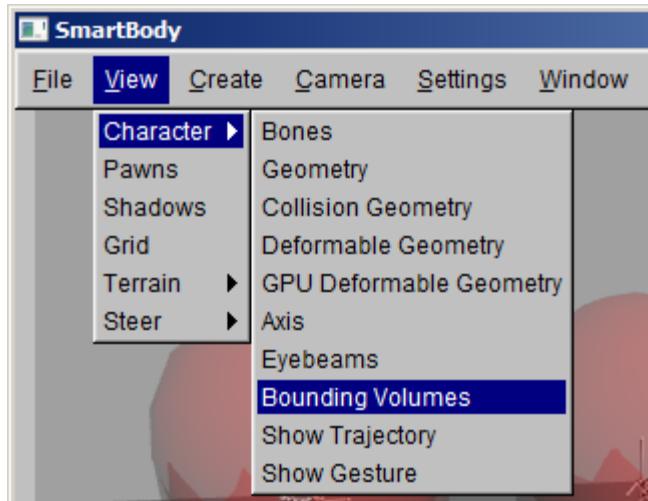
single capsule mode (left), and multiple capsule mode (right).



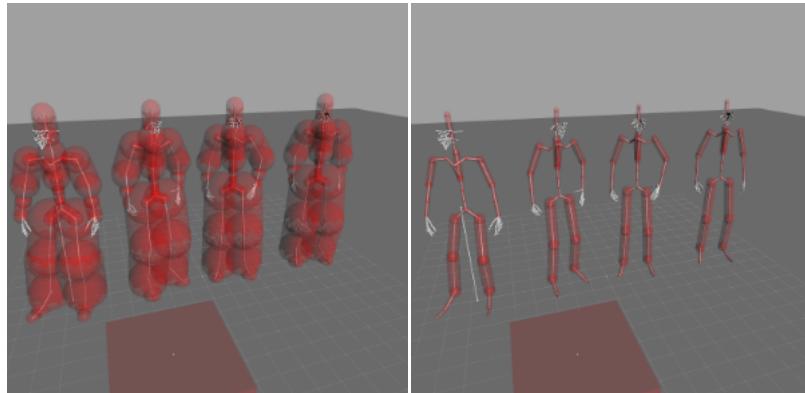
To turn on collision detection, choose the mode inside Resource Viewer -> Service -> Collision, then check "enable"



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

Configuring Eye Saccades

SmartBody 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. (Details can be found at [Eye Saccade](#))

Saccade requires a skeleton that includes 2 joints, one for each eye named *eyeball_left* and *eyeball_right*.

Currently SmartBody provides three basic behaviors: listening, talking and thinking. Custom behavior will be supported in the future. However you can adjust these basic behaviors by changing the attributes. Following are the attributes and their default values. (Default values for talking and listening mode come from paper "Eyes Alive", Lee, Badler, Badler with some adjustments for SmartBody skeleton setup).

Attributes	Description	Default Value for different <mode>		
		listening	talking	thinking
saccade.<mode>.percentage0	chance of saccade falls into 0 degree bin. considering you are looking into a 2D plane and dividing the plane into 8 direction bins. the percentage for 8 bins should add up to 100.	15.54	15.54	5.46
saccade.<mode>.percentage45	chance of saccade falls into 45 degree bin	6.46	6.46	10.54
saccade.<mode>.percentage90	chance of saccade falls into 90 degree bin	17.69	17.69	24.69
saccade.<mode>.percentage135	chance of saccade falls into 135 degree bin	7.44	7.44	6.44
saccade.<mode>.percentage180	chance of saccade falls into 180 degree bin	16.80	16.80	6.89
saccade.<mode>.percentage225	chance of saccade falls into 225 degree bin	7.89	7.89	12.80
saccade.<mode>.percentage270	chance of saccade falls into 270 degree bin	20.38	20.38	26.38

saccade.<mode>.percentage315	chance of saccade falls into 315 degree bin	7.79	7.79	6.79
saccade.<mode>.magnitudeLimit	magnitude limit for the saccade	10.0	12.0	12.0
saccade.<mode>.percentageMutual	chance of saccade to be gaze mutual , meaning saccade fixation is the target. another type is gaze away.	75.0	41.0	20.0
saccade.<mode>.mutualMean	saccade interval is decided by Gaussian curve. mean for the Gaussian curve, saccade type as gaze mutual.	237.5	93.9	180.0
saccade.<mode>.mutualVariant	variant for the Gaussian curve, saccade type as gaze mutual.	47.1	94.9	47.0
saccade.<mode>.awayMean	mean for the Gaussian curve, saccade type as gaze away.	13.0	27.8	180.0
saccade.<mode>.awayVariant	variant for the Gaussian curve, saccade type as gaze away.	7.1	24.0	47.0

<mode> can be one of the following: listening, talking and thinking. For example, you can do following python commands to reduce the magnitude limit of talking mode from 12.0 to 8.0.

```
character = scene.getCharacter("mycharactername")
character->setDoubleAttribute("saccade.talking.magnitudeLimit", 8.0f)
```

Note that these attributes would talk effect immediately.

Configuring Hand Synthesis

The Hand Synthesis algorithm generates hand motion using body motion. Due to self-occlusion, hands are extremely difficult to capture using traditional motion capture techniques. Most modern systems use independent, complicated techniques like gloves or 16-markers to capture hand data and attach it to the capture motion using post-processing. This is very time consuming and can't be used with our fast avatar capture system. This algorithm "infers" hand motion using a database of captured motions. The following document explains how to configure and run the system.

Hand configuration

In order to synthesize hands, a database of body and hand motions needs to be configured. This can be done conveniently using the python API. Please refer to "HandSynthesisDemo.py" for an example. The main steps involved are

1. Use the global configuration manager to create a new configuration with a unique name
2. Add motions to the configuration. These motions MUST have both body and hand data. The data will be separated automatically
3. Multiple configurations can be setup and switched during synthesis

Hand Synthesis

Follow the following steps to synthesize hand motion for a particular character:

1. Ensure that 'HandSynthesis.py' script was run
OR
2. Create a character, setup hand configurations and any retargetting necessary (Refer to the example script for help)
3. Select the character, go to the "Hand" attribute group and open it to access the various options
4. Use 'hand.config' to select a hand configuration to use (setup in the previous section)
5. Use 'hand.debug' to enable printing debug info
6. Use 'hand.motionName' to select the motion for which hand data needs to be generated
7. Use 'hand.switchConfig' to switch between different configurations (separate from synthesis)
8. Use 'hand.init' to initialize the synthesis
9. Once hand data has been generated, 'hand.enableBody' and 'hand.enableHand' can be used to enable or disable hand or body data (so they can be viewed separately)



Related articles



[Configuring Hand Synthesis](#)

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 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"/><event start="a:start+1"  
message="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 `<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 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 motion1 is 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 its 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 specified, 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 blends can 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 blend must 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

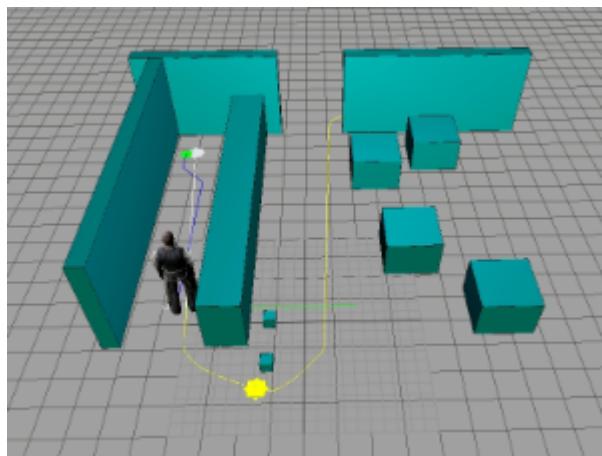
Parameter	Description	Example
name	Name of the blend to use. (Required)	<blend name="allJump" />
mode	Whether a new blend is being scheduled, or the current one is being updated. Options: <ul style="list-style-type: none">■ update■ schedule Default: schedule	<blend mode="schedule" name="allJump" />
sbm:wrap-mode	Wrap mode of the blend . Options: <ul style="list-style-type: none">■ Loop■ Once Default: Loop	<blend name="allJump" sbm:wrap-mode="Loop" />
sbm:schedule-mode	Schedule mode of the blend . Options: <ul style="list-style-type: none">■ Now■ Queued Default: Queued	<blend name="allJump" sbm:schedule-mode="Now" />
sbm:blend-mode	Blend mode of the blend . Options: <ul style="list-style-type: none">■ 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" />

<code>z</code>	Third parameter value of a 3D blend .	<code><blend mode="schedule" name="allLocomotion" x="100" y="2" z="-20"/></code>
<code>start</code>	Time offset to schedule the blend .	<code><blend name="allJump" start="3.0"/></code>

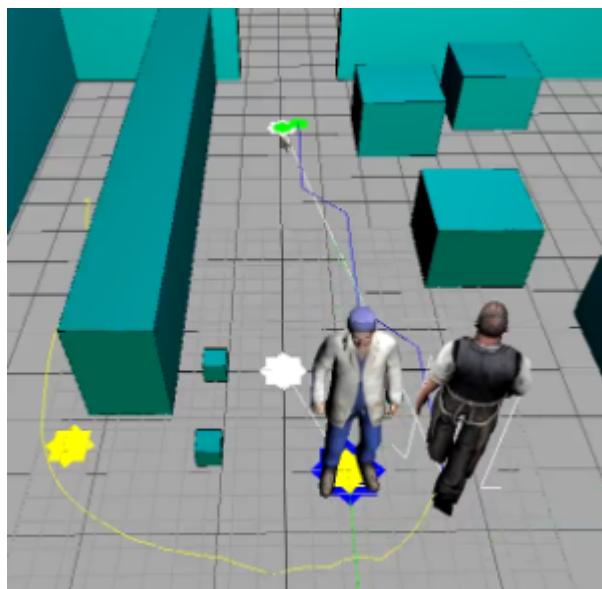
Locomotion

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 angles. Full locomotion will 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 shows 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 shows 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" />
speed	Speed of character	<locomotion target="100 300" speed="2.5" />
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 perform gestures, often with their hands and arms. In their simplest form, a gesture is an animation that is compatible with the underlying idle posture. A gesture could also use a set of animations in order to blend data, or use an algorithm to procedurally generate motion (such as a pointing gesture at a target). The type of gestures used is determined by the underlying data set, and there are generally no restrictions as to the type and style.

Requirements

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

Usage

Gesture lookup

With gesture map configured properly, you can retrieve the gesture according to your input. Lexeme, type, hand, style, posture are the four input that will determine the gesture you query. If the information given is not entire, you will get one animation from all the gesture animations that meet the criteria, note that your character's current posture has to match the posture from the gesture map unless you have set the gesture BML command to use sbm:additive attribute. Depending on gesture policy, you can either randomly get one or get first one from the animation list. To change gesture policy, set the attribute "gesturePolicy" to "random"|"first".

Gesture timing control

Start, ready, stroke, relax, end time markers are the same usage as <animation> BML. The extra feature for Gesture is holding capability. This will be achieved by stroke_end. e.g. <gesture lexeme="DEICTIC" stroke="3" relax="4"/>. If the gesture motion stroke relax time gap is 0.4sec, then by playing the above bml, you will hold gesture for 0.6sec. But if the motion time gap is 2sec, then this bml won't have any effect.

You can start gesture immediately on stroke by cutting off the motion before stroke. e.g. <gesture lexeme="DEICTIC" start="1" ready="1" stroke_start="1" stroke="2"/>. This will blend into stroke directly, transition time will be determined by stroke_start, the blend duration is (stroke - stroke_start). This feature can be used to transition from one gesture stroke to another gesture stroke. e.g.

```
<gesture id="a" lexeme="DEICTIC" type="LEFT" stroke="2" relax="4">
<gesture id="b" lexeme="DEICTIC" type="MID" start="a:relax" ready="a:relax"
stroke_start="a:relax" stroke="6" relax="8"/>
<gesture id="c" lexeme="DEICTIC" type="RIGHT" start="b:relax" ready="b:relax"
stroke_start="b:relax" stroke="10"/>
```

This gesture will first play gesture a, hold for x amount of time at stroke, transition 2sec to gesture b's stroke, hold for sometime and transition 2sec to gesture c's stroke and relax. By setting the attribute "bmlRequest.autoGestureTransition" to True. You just only need to specify stroke and stroke_end, the motion cutting part will be handled automatically. Previous BML looks like:

```
<gesture id="a" lexeme="DEICTIC" type="LEFT" stroke="2" relax="4">
<gesture id="b" lexeme="DEICTIC" type="MID" stroke="a:relax" relax="6"/>
<gesture id="c" lexeme="DEICTIC" type="RIGHT" stroke="b:relax" relax="8"/>
```

To make gesture holding looking more natural and less stiff, Perlin noise can be added by specifying sbm:joint-range, sbm:frequency and sbm:scale. e.g.:

```
<gesture id="a" lexeme="DEICTIC" type="LEFT" stroke="2" relax="4"
sbm:joint-range="l_shoulder" sbm:frequency="0.03" sbm:scale="0.02"/>
```

Parameters

Parameter	Description	Example
-----------	-------------	---------

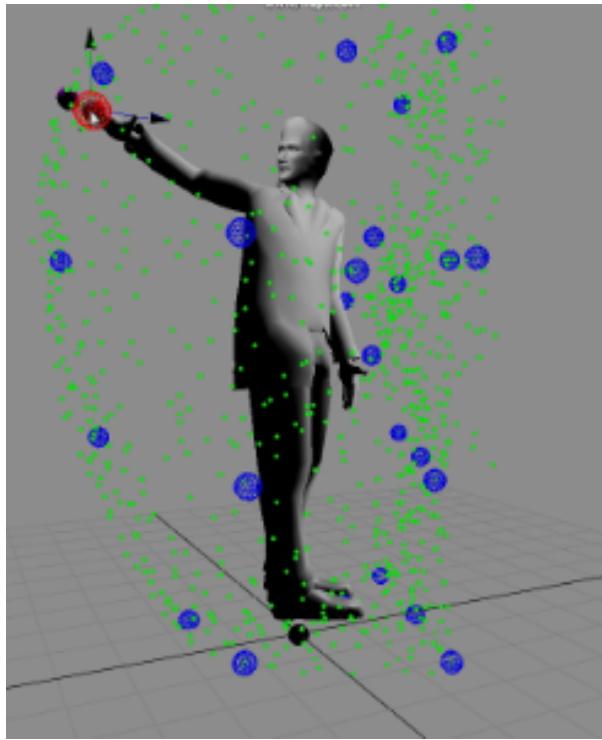
lexeme	type of gesture (required). This can be anything: DEICTIC, BEAT, YOU, ME, NEGATION, etc.	<gesture lexeme="YOU" />
mode	Handedness of the gesture. LEFT_HAND, RIGHT_HAND or BOTH_HANDS	<gesture lexeme="YOU" mode="LEFT_HAND" />
type	type of lexeme	<gesture lexeme="YOU" type="LEFT" />
name	name of the motion to use as a gesture. This is used as a substitute for using the gesture map.	<gesture name="BeatLeftHand" />
sbm:style	style of the lexeme	<gesture lexeme="YOU" type="LEFT" sbm:style="energetic" />
target	target for POINT and REACH gestures	<gesture lexeme="YOU" target="brad" />
start	start point of the gesture	<gesture lexeme="YOU" start="2" />
ready	ready point of the gesture	<gesture lexeme="YOU" start="2" ready="3" />
stroke_start	stroke start point of the gesture	<gesture lexeme="YOU" stroke_start="3" stroke="4" />
stroke	stroke point of the gesture	<gesture lexeme="YOU" stroke="5" />
stroke_end	stroke end point of the gesture	<gesture lexeme="YOU" stroke="3" stroke_end="4" />
relax	relax point of the gesture	<gesture lexeme="YOU" relax="3" end="4" />
end	end point of the gesture	<gesture lexeme="YOU" relax="3" end="4" />
sbm:joint-range	joints that will be added with perlin noise during gesture holding	<gesture lexeme="YOU" stroke="3" relax="4" sbm:joint-range="l_elbow l_shoulder l_wrists" />

sbm:frequency	Perlin noise frequency	<gesture lexeme="YOU" stroke="3" relax="4" sbm:joint-range="l_elbow l_shoulders l_wrists" sbm:frequency="0.05"/>
sbm:scale	Perlin noise scale	<gesture lexeme="YOU" stroke="3" relax="4" sbm:joint-range="l_elbow l_shoulders l_wrists" sbm:scale="0.03"/>
emotion	Emotion state. Can be: angry, neutral or sad.	<gesture lexeme="YOU" start="2" emotion="angry"/>
sbm:additive	Whether if gesture is additive to current posture	<gesture lexeme="YOU" start="2" sbm:additive="true"/>

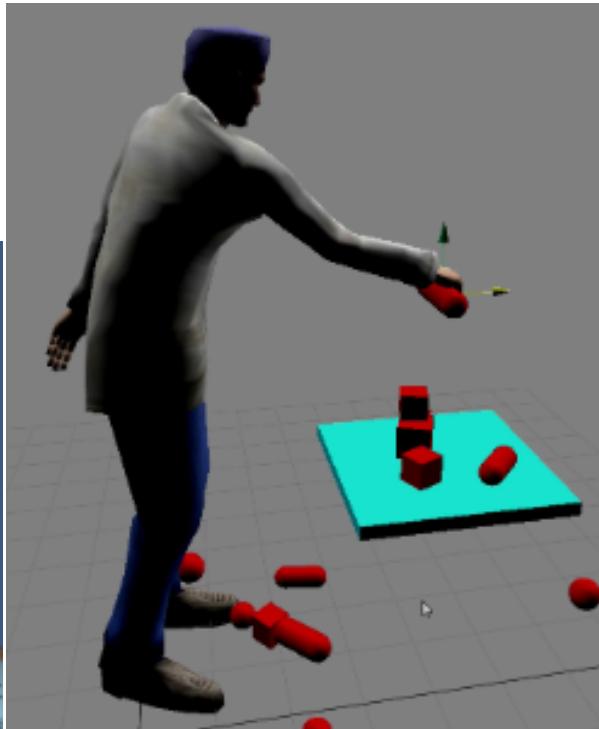
Reach, Grab, Touch, Point

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(SBEventHandler):
    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:

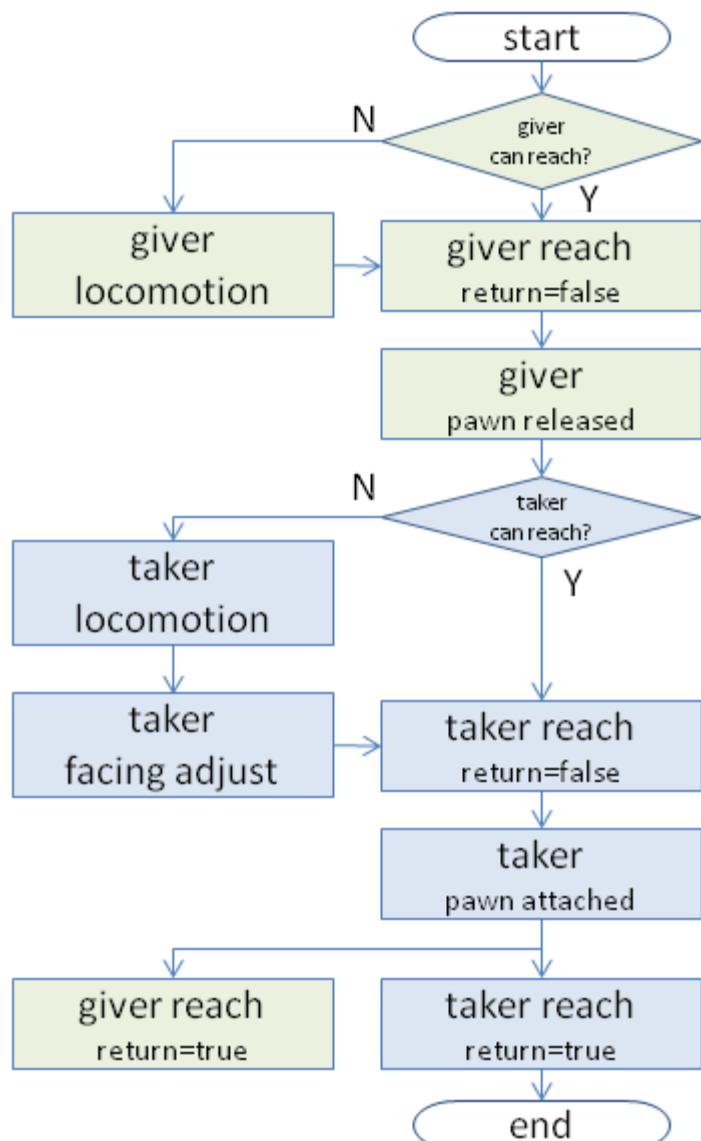
```
reset()
```

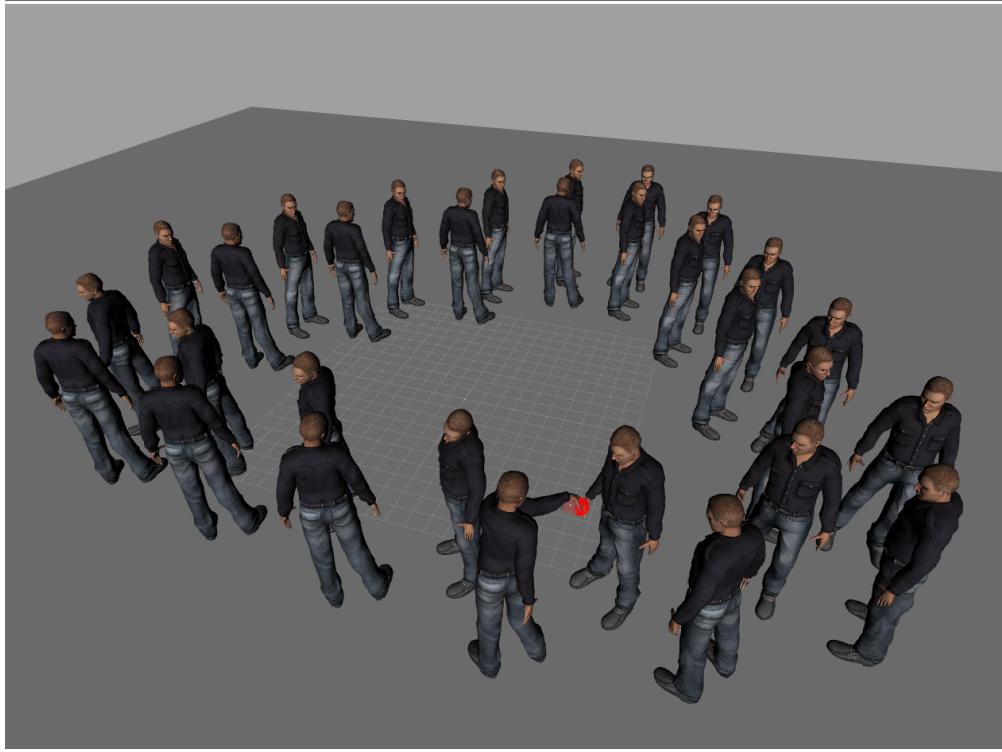
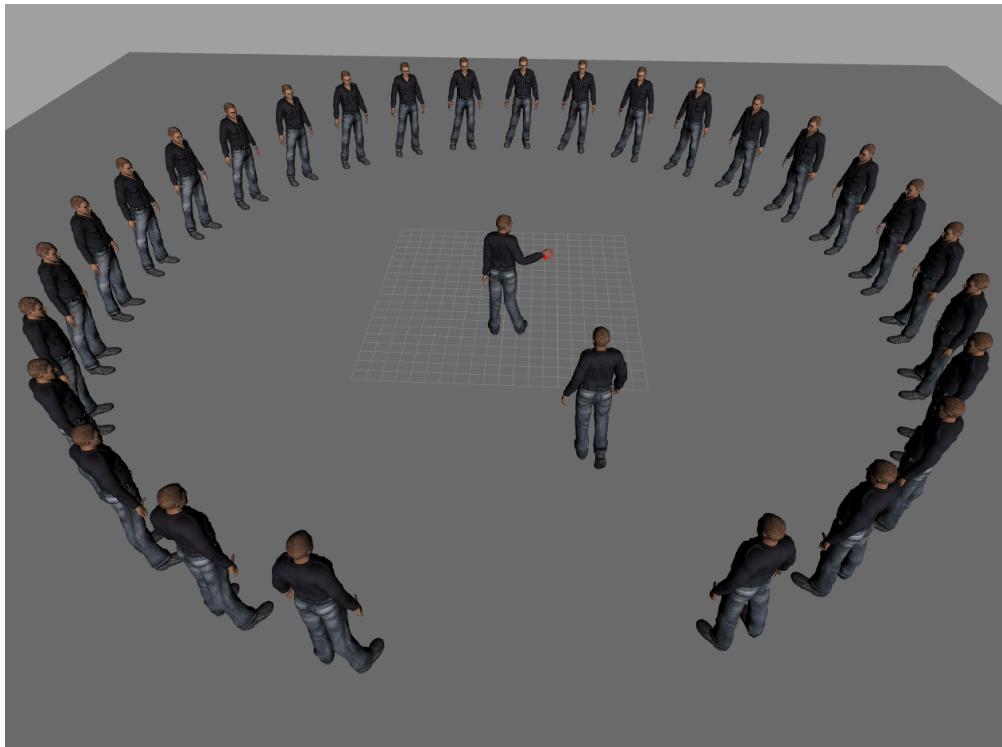
6. if agent(s) got stuck in the middle of the state, use this to reset:

```
masterReset()
```

7. you can directly specify giver, taker, obj as follows:

```
giverHandToTaker('elder', 'utah', 'box')
giverHandToTaker('brad', 'doctor', 'box')
```



2. "Hot Potato" demo with multiple 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 at 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

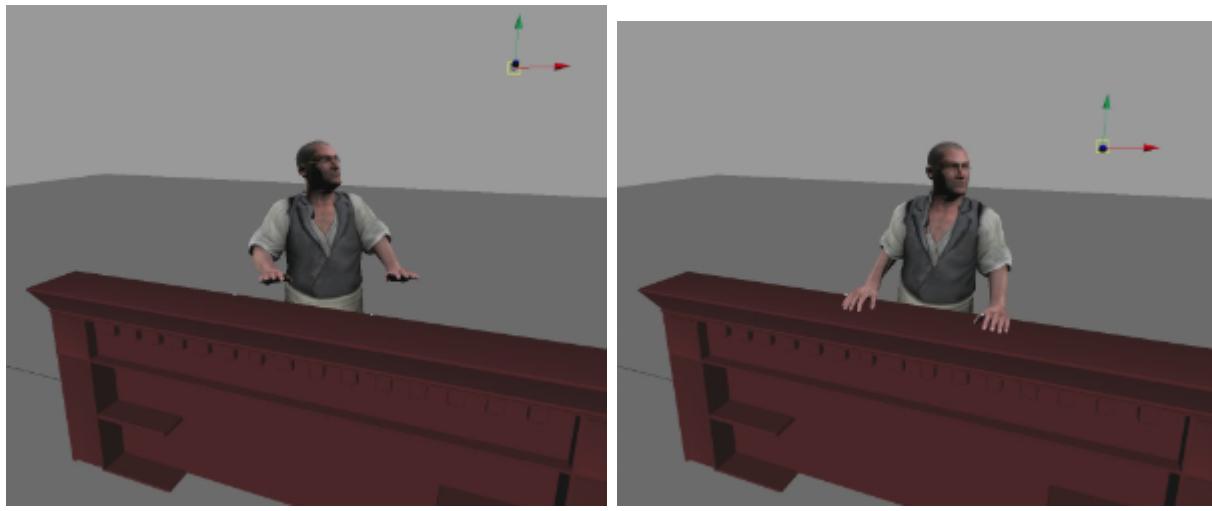
Parameter	Description	Example
target	what or who to gaze at. the target can either be the name of character or pawn, or the name of a joint on a character or pawn. To specify a joint, use the syntax <i>name:joint</i>	<gaze target="utah"/> <gaze target="utah:l_wrist"/>
sbm:target-pos	the position to gaze at. defined by x, y, z	<gaze sbm:target-pos="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.	<pre><gaze target="utah" sbm:joint-speed="500" /></pre> <pre><gaze target="utah" sbm:joint-speed="800 1500" /></pre>
sbm:joint-smooth	Decaying average smoothing value	<pre><gaze sbm:handle="mygaze" sbm:joint-smooth="1" /></pre>
sbm:fade-in	Fade-out interval to fade out a gaze, requires sbm:handle to be set	<pre><gaze sbm:handle="mygaze" sbm:fade-out="1" /></pre>
sbm:fade-out	Fade-in interval to reestablish a faded-out gaze, requires sbm:handle to be set	<pre><gaze sbm:handle="mygaze" sbm:fade-in="1" /></pre>
start	When the gaze will start	<pre><gaze target="utah" start="2" /></pre>

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 if 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	<pre><sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist"/></pre>
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.	<pre><sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist" sbm:root="base"/></pre>
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	<pre><sbm:constraint sbm:handle="myconstraint" sbm:fade-out="1' /></pre>
sbm:constraint-type	the constraint type to be enforced. it can be positional or rotational constraint	<pre><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" /></pre>

pos-x	the x positional offset added on top of positional constraint	<pre><sbm:constraint effector="r_wrists" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrists" sbm:root="base" sbm:constraint-type="pos" pos-y="10" pos-x="15" pos-z="5" /></pre>
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	<pre><sbm:constraint effector="r_wrists" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrists" sbm:root="base" sbm:constraint-type="pos" rot-y="10" rot-x="15" rot-z="5" /></pre>
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	<pre><sbm:constraint effector="r_wrists" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrists" sbm:root="base" start="3" ready="4" /></pre>
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, a 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 'fac's' is currently supported.	<face type="fac's" au="1" amount="1" />
side	Which side of the face will be activated; LEFT, RIGHT or BOTH	<face type="fac's" au="1" side="LEFT" amount="1" />
au	Action Unit number	<face type="fac's" au="26" amount="1" />
start	when the face movements starts	<face type="fac's" au="26" amount="1" start="2" />
ready	when the face movement is fully blended in, default is .25 seconds	<face type="fac's" au="26" amount="1" start="0" ready="1" />
stroke	the mid-point of the face movement	<face type="fac's" au="26" amount="1" stroke="4" />
relax	when the face movement starts blending out, default is .25 seconds	<face type="fac's" au="26" amount="1" relax="2" />
end	when the face movement ends	<face type="fac's" 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, speech 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 time 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 XML file. 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. Alternatively, 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.	<pre><speech type="text/plain"> Four score and seven years ago </speech></pre> <pre><speech type="application/ssml+xml"> Four score and seven years ago </speech></pre>
ref	Speech file reference. Used to determine which sound file and XML file to use that is associated with the speech.	<pre><speech ref="greeting"/></pre> <pre></speech></pre>
<mark name="" />	<p>Marker used to identify timings of speech before actual timings are known.</p> <p>Typically, the names are "Tn" where n is a whole number that is incremented</p> <p>before and after each word. T0, T1, T2, etc.</p>	<pre><speech type="text/plain"></pre> <pre><mark name="T0" />Four</pre> <pre><mark name="T1" /></pre> <pre><mark name="T2" />score</pre> <pre><mark name="T3" /></pre> <pre><mark name="T4" />and</pre> <pre><mark name="T5" /></pre> <pre><mark name="T6" />seven</pre> <pre><mark name="T7" /></pre> <pre><mark name="T8" />years</pre> <pre><mark name="T9" /></pre> <pre><mark name="T10" />ago</pre> <pre><mark name="T11" /></pre> <pre></speech></pre>

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).

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	<p>Starts running an eye saccade model (listen, talk or think).</p> <p>Note that using the mode attribute, saccades will be run randomly according to the data in the model.</p>	<saccade mode="talk" />
finish	<p>Stops running eye saccades according to one of the above modes.</p> <p>Can be 'true' or 'false'.</p>	<saccade finish="true" />
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" />

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 start="3" message="print 'hello' />
```

This will send out an event 3 seconds in the future which prints the word 'hello'.

```
<event start="2">
<![CDATA[
n = scene.getNumCharacters()
names = scene.getCharacterNames()
for i in range(0, n):
    print names[i]
]]>
</event>
```

This will send out an event at time 2 to print out the names of the characters in the same. Note that a multi-line command can be used by using the CDATA XML directive which tells the XML parser to interpret the contents as free text.

```
<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 start="sp1:T4" message="print 'hello' />
```

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

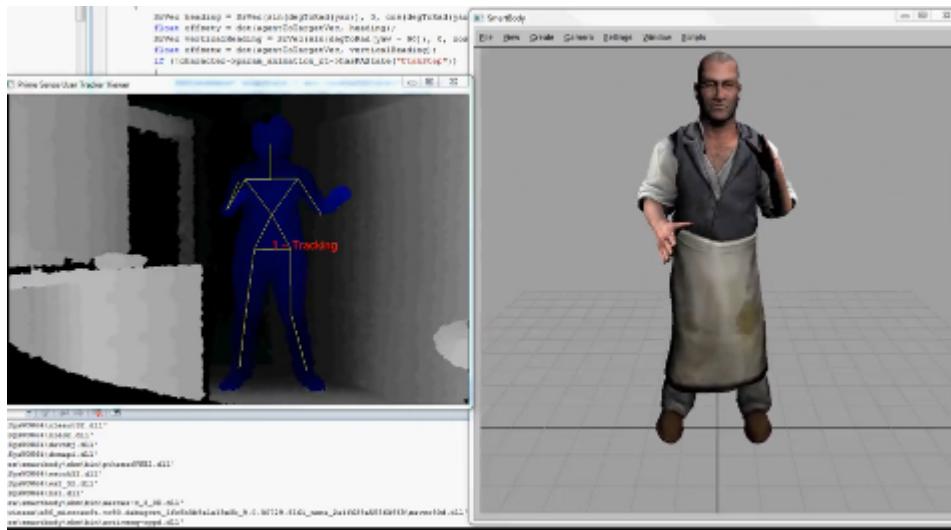
Parameters

Parameter	Description	Example
start	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 start="3" message="print 'hello'/'>
message	event to be sent. There is no limit to the complexity of the message, but it must adhere to XML syntax. In lieu of using the message attribute, the CDATA directive can be used to create a multi-line command as shown in the example above.	<event start="3" message="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 (kinecttracker for OpenNI, or SBKinectViewer for Microsoft Kinect SDK)
2. The 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 retargeting is done manually 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. For Kinect SDK application, we also send out the Kinect skeleton information in addition to joint rotation. So that SmartBody can make use of our on-line retargeting function to convert the Kinect data for many different characters on the fly.

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.

Controlling SmartBody Characters With Kinect

SmartBody characters can be configured to respond to external applications, such as kinect or other motion tracking devices. To enable this functionality, first tell SmartBody that an external application will be controlling the character via the following command:

```
scene.command("receiver enable")
```

Next, each SmartBody character has an attribute called 'receiverName' which determines the id of an external controller. The kinecttracker application uses the name 'utah' by default, so this can be set on the character by doing:

```
scene.setStringAttribute("receiverName", "utah")
```

Note that more than one character can be controlled by the same kinect by assigning the same receiverName to each character.

To stop controlling the character via the kinecttracker, call:

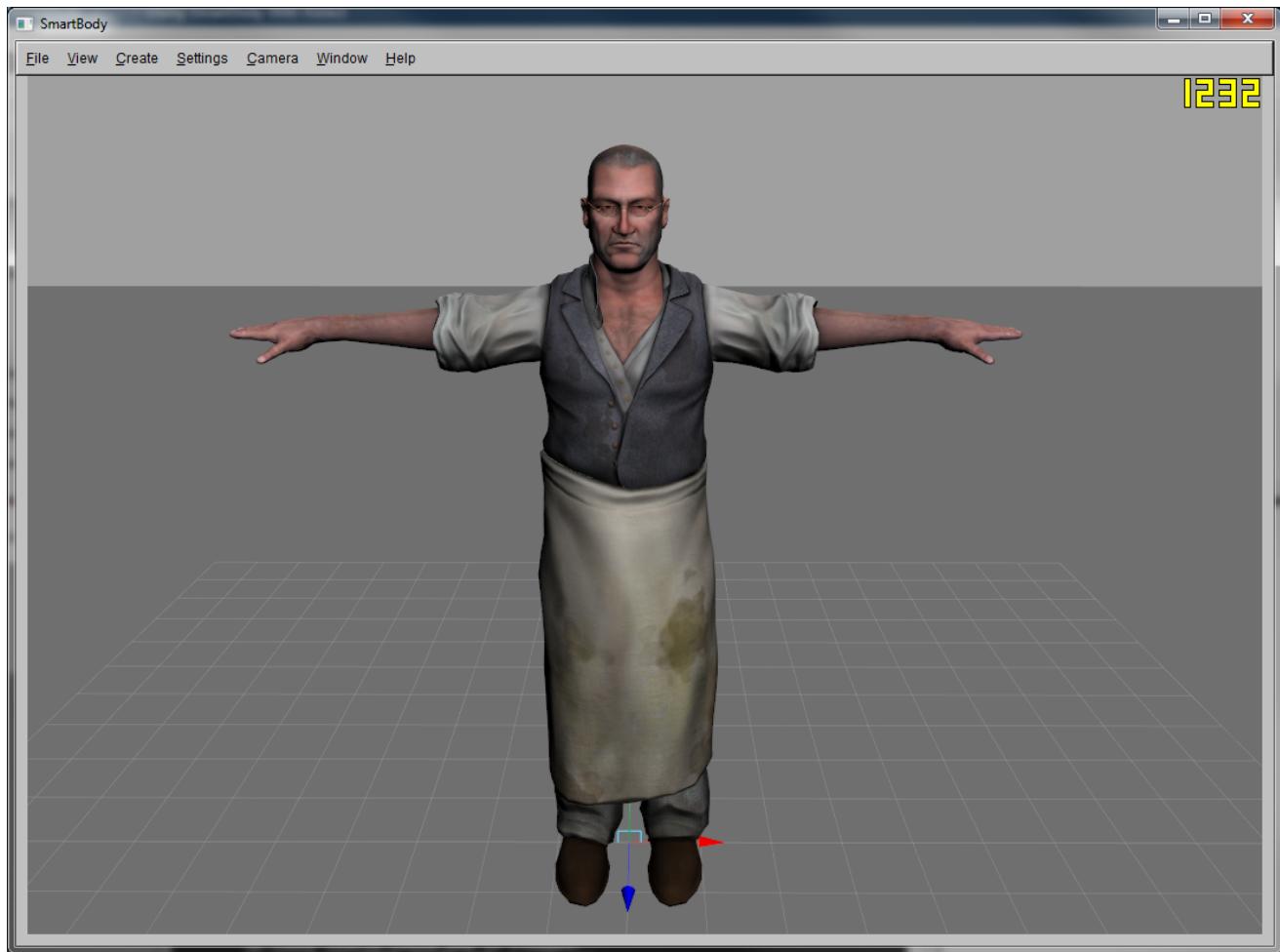
```
scene.command("receiver disable")
```

Building Microsoft Kinect SDK Application on Windows

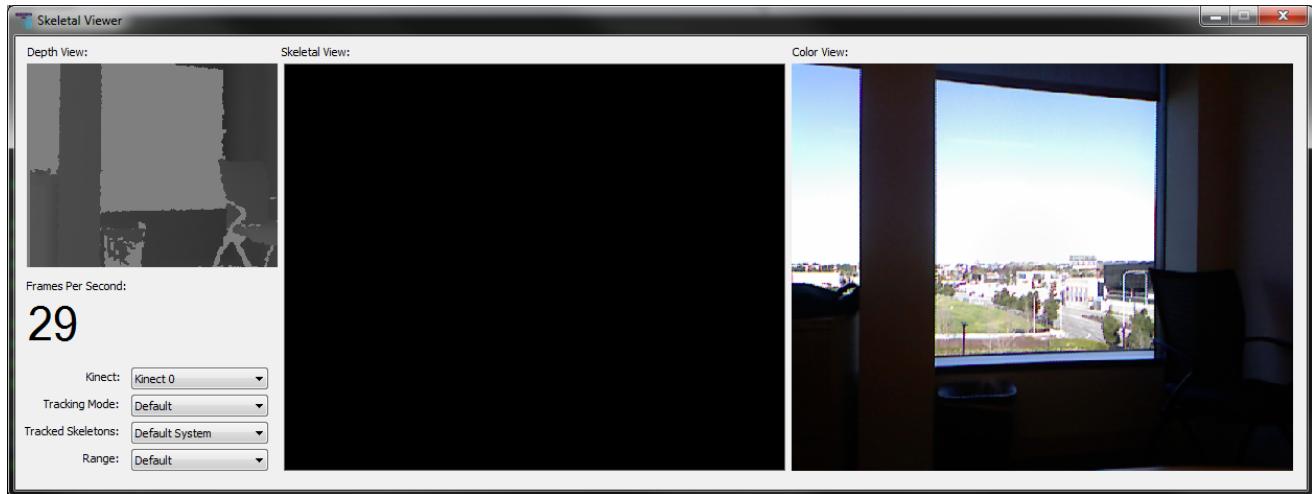
1	<p>Download and install Microsoft Kinect SDK, and Kinect for Windows Developer Toolkit at: http://www.microsoft.com/en-us/kinectforwindows/develop/developer-downloads.aspx</p> <p>The SDK will install the drivers on your machine.</p>
2	<p>Go to trunk\lib\SBKinectViewer, select the solution file (currently only support 2010).</p> <p>Note that you have to first build smartbody-fltk project before building SBKinectViewer, you have to use the same visual studio version for building both projects.</p>
4	<p>Run smartbody fltk viewer, load up KinectDemo.py from trunk\data\examples, and run SBKinectViewer built in step 2 at the same time.</p>

Use Microsoft Kinect SDK Application

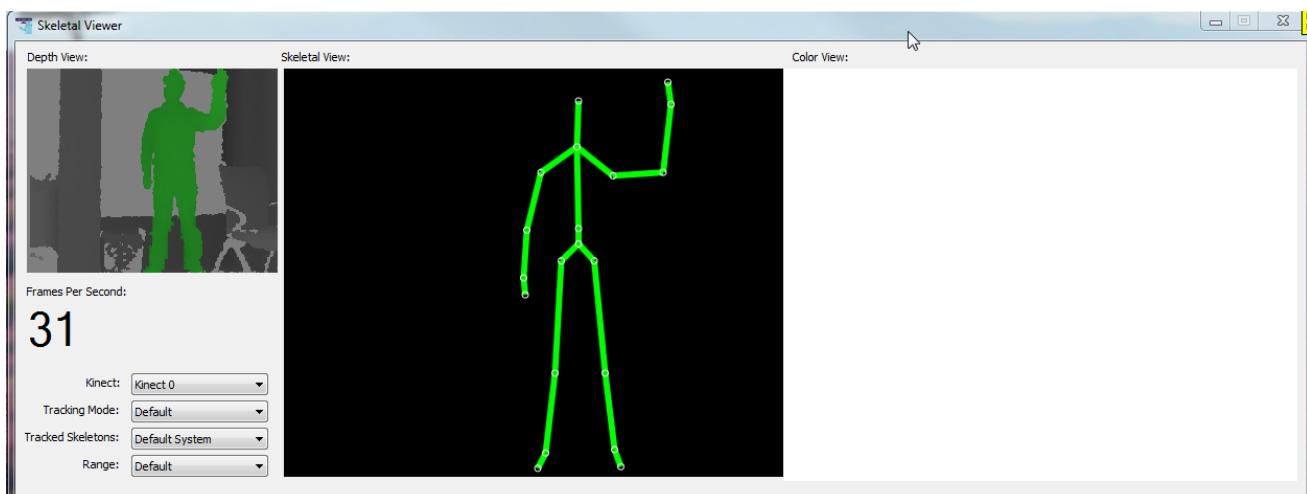
First, start **smartbody-fltk** viewer, and load up KinectDemo.py in the folder \data\examples. Once it's finished loading, you should see a Utah character like the image below.



Second, start **SBKinectViewer** located in folder `\lib\SBKinectViewer`. It should looks like the image below :



Then the user needs to step into the view of Kinect Camera until a full skeleton is seen in the Skeletal View. Note that the Color/Depth/Skeletal Views below will show the **mirror** version of the capture.



Once the skeleton is tracked by Kinect, the joint angle data will be sent to SmartBody and converted for Utah character. You will see Utah is mimicking the pose of the user.



Building OpenNI Kinect Application on Windows

Note: Kinect SDK driver might have a conflict with OpenNI driver, so be sure to uninstall Kinect SDK driver before

using SmartBody Kinect Application.

1	<p>Download and install OpenNI 32-bit Development Edition</p> <p>http://www.openni.org/wp-content/uploads/2012/12/OpenNI-Win32-1.5.4.0-Dev1.zip</p> <p>Select 'OpenNI Binaries' , 'Stable', then choose the Windows x86 32-bit Development Edition</p>
2	<p>Download the Kinect drivers</p> <p>http://www.openni.org/wp-content/uploads/2012/12/Sensor-Win32-5.1.2.1-Redist.zip</p>
3	<p>Download and install NITE</p> <p>http://www.openni.org/wp-content/uploads/2012/12/NITE-Win32-1.5.2.21-Dev.zip</p> <p>Select 'OpenNI Compliant Middleware Binaries' , 'Stable', then choose the PrimeSense NITE Windows x86 Development Edition</p>
4	<p>Go to trunk\lib\kinecttracker, select the solution file according to your visual studio version and build(currently supporting 2010 and 2008).</p> <p>Note that you have to first build smartbody-fltk project before building kinecttracker, you have to use the same visual studio version for building both projects.</p> <p>Please make sure win32 platform is chosen when building the solution.</p> <p>Also make sure OpenNI is installed at C:\Program Files (x86)\OpenNI, if not, you need to manually change the project including path to the your customized OpenNI location.</p>
5	<p>Run smartbody fltk viewer, load up KinectDemo.py from trunk\data\examples, and run kinecttracker.exe built in step 4 at the same time.</p>

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	<p>Retrieve and install the OpenNI drivers</p> <pre>cd OpenNI wget http://www.greenfoot.org/doc/kinect/OpenNI-Linux32.tar.bz2 tar -jxf OpenNI-Linux32.tar.bz2 sudo ./install.sh cd ..</pre>
2	<p>Retrieve and install NITE</p> <pre>mkdir NITE cd NITE wget http://www.greenfoot.org/doc/kinect/NITE-Linux32.tar.bz2 tar -jxf NITE-Linux32.tar.bz2 echo 'OKOIk2JeIBYC1PWVnMoRKn5cdY4=' sudo ./install.sh cd ..</pre>
3	<p>Install the Kinect driver for Linux</p> <pre>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 ..</pre>
4	<p>Make sure that your ActiveMQ service is running, since messages between the standalone kinect application and SmartBody are managed via ActiveMQ.</p> <p>If it is not running, run:</p> <pre>sudo activemq start</pre>

5	<p>Build the kinecttracker application located in smartbody/lib:</p> <p>uncomment the line in smartbody/lib/CMakeLists.txt that says: add_directory (kinecttracker)</p> <p>by removing the hash mark (#) in front of that line. This will add the kinecttracker application to the standard SmartBody build process.</p> <p>Then build SmartBody:</p> <pre>cd smartbody/build make install</pre> <p>You should see an application called smartbody/lib/kinecttracker/kinecttracker</p>
6	<p>Run sbgui:</p> <pre>./smartbody/core/smartbody/sbm/bin/sbgui</pre>
7	<p>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':</p> <p>From Python:</p> <pre>c = scene.getCharacter("utah") c.setAttribute("receiverName", "kinect2")</pre>
7	<p>Run the kinecttracker application:</p> <pre>./smartbody/lib/kinecttracker/kinecttracker</pre> <p>Then approach the kinect camera and track your pose.</p> <p>You should be able to control one or more characters with the kinect data.</p>

Building OpenNI Kinect Application on OSX

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

Asset Formats

SmartBody can read and utilize several motion formats:

- .bvh
- .ASF/AMC
- .dae (COLLADA)
- .fbx

In addition, SmartBody has several native formats for motion (.skm) and skeleton definition (.sk).

Note that .fbx format must be converted to the native skeleton (.sk) and motion (.skm) format before use using the tools/FbxToSBConverter application.

.skm Motion Format

The native SmartBody motion format (.skm) looks like this:

```
SkMotion

name "<NameOfMotionHere>"

channels <numberOfChannels>
<jointName> <channelType>
...
...

frames <numberOfFrames>
kt <time> <channel1> <channel2> <channel3>
...
<metadata> : <value>
```

where <numberOfChannels> is the number of translation or rotation channel

where <jointName> is the name of the joint

where <channelType> is one of XPos, YPos, ZPos, or Quat

where <numberOfFrames> is the number of frames

where <time> is the time of a particular frame

where <channel1>, <channel2>, <channeln> are the values of the channel in the order specified above. In the data, translation channels (XPos, YPos, ZPos) have a single value, whereas quaternion channels use 3 values: the x/y/z values of the axis when decomposing a quaternion into an axis/angle, multiplied by the length of the angle (thus the axis/angle representation is represented by only three values, and can be reconstituted into four values by determining the magnitude of the three values which determines the angle, and dividing by that length, leaving the x/y/z axis)

where <metadata> is the name of some metadata (such as start: ready: or other descriptor) and <value> is the value of that data (could be a motion time)

Appendix 1: Building SmartBody

There are several SmartBody applications that can be built:

Application	Comment
sbgui	a standalone SmartBody application and scene renderer (formerly called sbm-fltk)
SmartBody	a dynamic library that can be incorporated into a game engine or other application
simplesmartbody	a standalone SmartBody application that demonstrates how to connect to the SmartBody API
ogresmartbody	Example of using the Ogre rendering engine with SmartBody
irrlichtsmartbody	Example of using the Irrlicht rendering engine with SmartBody
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 with two separate means of connecting: 1) Directly embedding SmartBody (similar to ogresmartbody, above). 2) Communicating to a SmartBody process over the BoneBus (network protocol)
sbdesktop	(Windows only) A fully-functional SmartBody environment rendered on the desktop.
SBKinectViewer	an application that sends information from a Kinect device to another SmartBody application

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

Mobile Application	Comment
sbjniapp	(Android only) Simple OpenGL ES front end for SmartBody for Android devices
sbogreapp	(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	(<i>iOS only</i>) Ogre3D engine connected with SmartBody for iOS devices
smartbody-unity	(<i>iOS only</i>) Unity3D project for SmartBody

Finally, there is also a web application that can be built:

Web Application	Comment
simplesmartbody	(Flash) A simple SWF application with StageD3 as front end and SmartBody as core engine.

Building SmartBody for Windows

You will need Visual Studio 2010 to build SmartBody (there is an older, unsupported build for Visual Studio 2008) '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://activemq.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
xutils-dev
libxi-dev
freeglut3-dev
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
libfltk1.3-dev
libboost-all-dev
libode-dev
ncurses-dev
libprotobuf-dev

You can run the following command to install the above packages:

```
sudo apt-get install cmake g++ libxerces-c3-dev libgl1-mesa-dev libglu1-mesa-dev xutils-dev libxi-dev
freeglut3-dev 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
libfltk1.3-dev libboost-all-dev libode-dev libprotobuf-dev
```

Linux packages needed for text-to-speech engine

festival-dev
ncurses-dev

You can run the following command to install packages needed for TTS:

```
sudo apt-get install festival-dev
```

Linux packages needed for Ogre3D viewer

libzzip-dev
libxaw7-dev

libxxf86vm-dev

libxrandr-dev

libfreeimage-dev

nvidia-cg-toolkit

liboios-dev

libogre-dev

ogre-1.8-samples-data

You can run the following command to install packages necessary for Ogre:

```
sudo apt-get install libzzip-dev libxaw7-dev libxxf86vm-dev libxrandr-dev libfreeimage-dev nvidia-cg-toolkit
liboios-dev libogre-dev ogre-1.8-samples-data
```

Linux packages needed for test suite

imagemagick

You can run the following command to install the packages necessary to run the test suite:

```
sudo apt-get install 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	

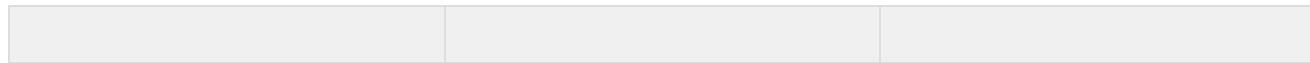
2	<p>Download and install the Boost numeric bindings:</p> <p>Download from:http://mathematica.ticia.n.de/dl/software/boost-numeric-bindings/boost-numeric-bindings-20081116.tar.gz</p> <p>Place in lib/ and unpack using:</p> <pre>tar -xvzf boost-numeric-bindings-20081116.tar.gz cd boost-numeric-bindings sudo mkdir /usr/local/include/boost sudo mkdir /usr/local/include/boost/numeric sudo cp -R boost/numeric/bindings /usr/include/boost/numeric</pre>	
3	<p>Download and install ActiveMQ.</p> <p>Download from: https://cwiki.apache.org/confluence/display/AMQCPP/Download</p> <p>unpack to temp folder <code>./configure --disable-ssl</code> <code>make</code> <code>sudo make install</code> <code>sudo ldconfig</code></p>	
4	<p>(Optional) Build the elsender:</p> <pre>cd lib/elsender xbuild elsender.csproj (ignore post-build error)</pre>	

5	<p>Build speech tools and Festival that are located in the local SmartBody directory:</p> <pre>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</pre>	
6	<p>Make and install SmartBody:</p> <p>First, generate the Makefiles with cmake:</p> <pre>mkdir buildfolder cmake ..</pre> <p>Go to the build directory, make and install the applications</p> <pre>cd buildfolder make install</pre>	

Building SmartBody for OSx

To build the OSX version:

(NOTE: There is an error when using cmake 2.8.4. Please download and install cmake 2.8.10 from: <http://www.cmake.org/files/v2.8/cmake-2.8.10.2.tar.gz>)



1	<p>Download and install boost:</p> <p>Download Boost from: https://sourceforge.net/projects/boost/files/boost/1.53.0/</p> <p>Place in lib/ and unpack, build and install into /usr/local using:</p> <pre><code>./bootstrap.sh ./bjam sudo ./bjam install</code></pre>	
2	<p>Download and install the Boost numeric bindings:</p> <p>Download from:http://mathematica.de/dl/software/boost-numeric-bindings/</p> <p>boost-numeric-bindings-20081116.tar.gz</p> <p>Place in lib/ and unpack using:</p> <pre><code>tar -xvzf boost-numeric-bindings-20081116.tar.gz cd boost-numeric-bindings sudo cp -R boost/numeric/bindings /usr/local/include/boost/numeric</code></pre>	
3	<p>Download and install FLTK 1.3.</p> <p>Download from: http://www.fltk.org/software.php?VERSION=fltk-1.3.x-r10259</p> <p>Place in lib/, unpack and configure:</p> <pre><code>./configure --enable-shared make sudo make install</code></pre>	

4	<p>Download and install the ActiveMQ libraries.</p> <p>Download from: https://cwiki.apache.org/confluence/display/AMQCPP/ActiveMQ-CPP+3.8.3+Release</p> <p>Then rebuild activemq-cpp using:</p> <pre>./configure --disable-ssl make sudo make install</pre>	
5	<p>Get Open Dynamics Engine (ODE).</p> <p>Download ode-0.12 from: https://sourceforge.net/projects/openode/files/</p> <p>Place in lib/, unpack and configure:</p> <p>32 bits:</p> <pre>./configure --with-drawstuff=none --enable-double-precision</pre> <p>64 bits:</p> <pre>./configure --with-drawstuff=none --with-pic --enable-double-precision</pre> <pre>make sudo make install</pre>	
6	<p>Build and install xerces</p> <p>http://www.takeyellow.com/apachemirror//xerces/c/3/sources/xerces-c-3.1.1.tar.gz</p> <pre>tar zxvf xerces-c-3.1.1.tar.gz cd xerces-c-3.1.1 ./configure make sudo make install</pre>	

7	<p>Build and install google protocol buffer</p> <p>https://code.google.com/p/protobuf/downloads/list</p> <pre>tar zxvf protobuf-2.5.0 cd protobuf-2.5.0 ./configure make sudo make install</pre>	
8	<p>Build and install GLEW</p> <p>https://sourceforge.net/projects/glew/files/glew/1.6.0/glew-1.6.0.tgz/download</p> <pre>tar zxvf glew-1.6.0.tgz cd glew-1.6.0 make sudo make install</pre>	
9	<p>Build and install OpenAL</p> <p>http://connect.creativelabs.com/openal/Downloads/openal-soft-1.13.tbz2</p> <pre>tar zxvf openal-soft-1.13.tbz2 cd openal-soft-1.13 cmake . make sudo make install</pre>	
10	<p>Build and install FreeALUT</p> <p>http://distro.ibiblio.org/rootlinux/rootlinux-ports/more/freealut/freealut-1.1.0.tar.gz</p> <pre>unzip freealut-1.1.0-src.zip cd freealut-1.1.0-src cmake . make sudo make install</pre>	
11	<p>Build and install libsndfile</p> <p>http://www.mega-nerd.com/libsndfile/files/libsndfile-1.0.25.tar.gz</p>	

12	<p>(Optional) Build the elsender:</p> <pre>cd lib/elsender xbuild elsender.csproj (ignore post-build error)</pre>	
13	<p>(Optional) Build the Ogre engine:</p> <p>Download the 1.8.1 version fromfrom: www.ogre3d.org/download/source</p> <pre>./configure sudo make install</pre>	
14	<p>Build speech tools and Festival that are located in the local SmartBody directory:</p> <pre>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</pre>	
15	<p>Make and install SmartBody:</p> <p>First, generate the Makefiles with cmake:</p> <pre>mkdir buildfolder cmake ..</pre> <p>Go to the build directory, make and install the applications</p> <pre>cd buildfolder make install</pre>	

Troubleshooting

To determine which dynamic libraries are associated with the sbgui executable you can run the otool command:

```
otool -L sbgui
```

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: sbjniapp (A simple app using the Java Native Interface), sbogreapp (an Ogre-based app), and vh-wrapper (a set of libraries that can be used in conjunction with the Unity game engine).

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	<p>Download and install the android-ndk from: http://developer.android.com/sdk/ndk/index.html</p> <p>(Note that when building on the windows platform, you will also need to install cygwin 1.7 or higher from http://www.cygwin.com/)</p> <p>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")</p>
3	<p>Install Eclipse (http://www.eclipse.org/) and its Android ADT plug-in (http://developer.android.com/sdk/eclipse-adt.html)</p> <p>The supporting libraries have been built using Android version 2.3.3 or higher.</p>

4	<p>(Optional) Build sbjniapp</p> <p>sbjniapp is a hello world project for SmartBody. It has very basic rendering and minimal functionality, but it helps demonstrate the SmartBody port on android.</p> <ul style="list-style-type: none"> i) Go to (SmartBodyDir)/android/sbjniapp/ 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 /sbjniappdata and its files to sdcard directory of your android device. (Usually under /sdcard/) iv) Use Eclipse to open the project (SmartBodyDir)/android/sbjniapp/. v) Select Project->Build Project. Connect the device and then run the program as "Android Application".
5	<p>(Optional) Build sbogreapp</p> <p>sbogreapp combines SmartBody and Ogre for high quality rendering. Currently, it is very slow when rendering in deformable model mode.</p> <ul style="list-style-type: none"> a. Go to (SmartBodyDir)/android/sbogreapp/ 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/sbo greapp/sbogredir and its files to sdcard directory of your android device. (Usually under /sdcard/) d. Use Eclipse to open the project (SmartBodyDir)/android/sbogreapp/. e. Select Project->Build Project. Then connect the Android 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 architecture (x86, etc) or you prefer to rebuild all libraries from their sources, you will need to perform the following steps:

Building Supporting Libraries

1	<p>Building Boost for Android:</p> <p>Download BOOST library, extract it into SmartBody/lib</p> <p>modify libs\filesystem\v2\src\v2_operations.cpp, change:</p> <pre># if !defined(__APPLE__) && !defined(__OpenBSD__) # include <sys/statvfs.h> # define BOOST_STATVFS statvfs # define BOOST_STATVFS_F_FRSIZE vfs.f_frsize # else #endif __OpenBSD__ # include <sys/param.h> #endif</pre> <p>to:</p> <pre># if !defined(__APPLE__) && !defined(__OpenBSD__) && !defined(__ANDROID__) # include <sys/statvfs.h> # define BOOST_STATVFS statvfs # define BOOST_STATVFS_F_FRSIZE vfs.f_frsize # else #endif __OpenBSD__ #ifndef __ANDROID__ # include <sys/vfs.h> #endif</pre> <p>modifiy the file SmartBody/android/boost/userconfig.jam, look for :</p> <p>ANDROID_NDK =/android-ndk ; and change the directory "..../android-ndk" so it points to the android NDK directory</p> <p>You may also need to change all arm-linux-androideabi-xxx to the corresponding toolchain name based on your target architecture and platform.</p> <p>(use Cygwin in Windows platform)</p> <pre>./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</pre>
2	<p>Building iconv</p> <p>TODO</p>

3	Building xerces TODO
4	Building clapack TODO

Building SmartBody for iOS

- Requirements
- Compiling with console
- Compiling with XCode
 - Libraries
 - Applications
- Notes

Requirements

To run SmartBody on iOS devices you will need:

- Mac computer, where you should have fairly recent mac operating system.
- iOS device on which the software should also be recent.
- Apple developing ID which allows you to develop on your devices.
- XCode whose downloads require apple ID, it's also the port where you download the application onto your device.

To be more clearly, all the things required for developing iOS application doesn't have to be all up-to-date, but they need to be matching versions and if you accidentally upgrade your device there's no backward compatibility. It's always a good habit to keep updating all your software.

Due to software version problem, the xcode version and location described below are yield to change.

Compiling with console

There's many steps in the following tables, but you can choose either build yourself or pre-built version can be found at [trunk/ios/libs/iphoneos](#)

Compiling using console

1

Cross compiling apr (You can find prebuilt library at trunk/ios/libs/iphoneos)

- Download apr-1.3.* from <http://archive.apache.org/dist/apr/>
- Copy setup-iphoneos.sh from trunk/ios/activemq/apr to the folder where you extracted the above downloaded contents.
- Change the SBROOT inside setup-iphoneos.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
- 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 (You can find prebuilt library at trunk/ios/libs/iphoneos)

- Download apr-util-1.3.* from <http://archive.apache.org/dist/apr/>
- Copy setup-iphoneos.sh from trunk/ios/activemq/apr-util to the folder where you extracted the above downloaded contents.
- Change the SBROOT inside setup-iphoneos.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 setup-iphoneos.sh

3

Cross compiling activemq-cpp-library (You can find prebuilt library at trunk/ios/libs/iphoneos)

- Download activemq-cpp-library-3.4.0 from <http://apache.osuosl.org/activemq/activemq-cpp/source/>
- Copy setup-iphoneos.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 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 setup-iphoneos.sh

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 (You can find prebuilt library at trunk/ios/libs/iphoneos)

- Download xerces-c-3.1.1.tar.gz from <http://xerces.apache.org/xerces-c/download.cgi>
- Copy setup-iphoneos.sh from trunk/ios/xerces-c to the folder where you extracted the above downloaded contents.
- Change the SBROOT inside setup-iphoneos.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 setup-iphoneos.sh

5

Cross compiling ODE (You can find prebuilt library at trunk/ios/libs/iphoneos)

- Download ode-0.11.1.zip from <http://sourceforge.net/projects/opende/files/>
- Copy setup-iphoneos.sh from trunk/ios/ode to the folder where you extracted the above downloaded contents
- Change the SBROOT inside setup-iphoneos.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 setup-iphoneos.sh

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 ieecek.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
- Run setup-iphoneos.sh

7	<p>Cross compiling python (You can find prebuilt library at trunk/ios/libs/iphoneos)</p> <ul style="list-style-type: none"> • 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 - <ul style="list-style-type: none"> • 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. • Run setup-iphoneos.sh.
8	<p>Cross compiling libsndfile (You can find prebuilt library at trunk/ios/libs/iphoneos)</p> <ul style="list-style-type: none"> • Download from http://sourceforge.net/projects/csound/files/csound5/iOS/libsndfile-iOS.zip/stats/timeline • Build and copy libalut.a to trunk/ios/libs/iphoneos
9	<p>Cross compiling freealut (You can find prebuilt library at trunk/ios/libs/iphoneos)</p> <ul style="list-style-type: none"> • Download freealut from https://github.com/vancegroup/freealut • Copy CMakeList.txt, setup-iphoneos.sh and toolchain -iphoneos.cmake to downloaded freealut top level. • Change the SBROOT inside setup-iphoneos.sh • Run setup-iphoneos.sh

10	<p>Cross compiling pocket sphinx (Not required)</p> <p>Download from http://www.rajeevan.co.uk/pocketsphinx_in_iphone/. The steps are on the website.</p> <p>P.S.</p> <ul style="list-style-type: none">• 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 with XCode

Libraries

	Compiling using Xcode4
1	<p>Build bonebus</p> <ul style="list-style-type: none"> • Open smartbody-iphone.xcworkspace, select the scheme to be bonebus, build
2	<p>Build boost libraries (You can find prebuilt library at trunk/ios/libs/iphoneos, version 1.51)</p> <ul style="list-style-type: none"> • Download boost_1_44_0.tar.gz(boost_1_51_0.tar.gz) from http://www.boost.org/users/history/version_1_44_0.html(http://www.boost.org/users/history/version_1_51_0.html) and unzip to trunk/ios/boost. Make sure the folder name is boost_1_44_0(boost_1_51_0), replace the existing folders if necessary. • Open boost/boost.xcodeproj, select boost_system, boost_filesystem, boost_regex, boost_python, build them separately. Make sure the header search path and lib search path are correct according to boost version. Also make sure that source files for each target are included properly under xcode project according to the right boost version. • Download boost_numeric_bindings from http://mathematician.de/dl/software/boost-numeric-bindings/boost-numeric-bindings-20080714.tar.gz and unzip it to trunk/ios/boost, make sure the name is boost_numeric_bindings
3	<p>Build steersuite</p> <ul style="list-style-type: none"> • Open smartbody-iphone.xcworkspace, select the steerlib, pprAI, build them separately.
4	<p>Build vhmsg</p> <ul style="list-style-type: none"> • Open smartbody-iphone.xcworkspace, select scheme vhmsg and build.
5	<p>Build vhcl</p> <ul style="list-style-type: none"> • Open smartbody-iphone.xcworkspace, select scheme vhcl and build.
6	<p>Build wsp</p> <ul style="list-style-type: none"> • Open smartbody-iphone.xcworkspace, select scheme wsp and build.

7	<p>Build smartbody-lib (Currently including boost1.5, python2.65)</p> <ul style="list-style-type: none"> • Open smartbody-iphone.xcworkspace, select scheme smartbody-lib and build. <p>P.S. This xcode project needs maintenance by the developer to make sure it incorporates all the file from SmartBody core.</p>
8	<p>Build smartbody-dll</p> <ul style="list-style-type: none"> • Open smartbody-iphone.xcworkspace, select scheme smartbody-dll and build.
9	<p>Build vhwrapper-dll (Not required, for Unity only)</p> <ul style="list-style-type: none"> • Open smartbody-iphone.xcworkspace, select scheme vhwrapper-dll and build.

Applications

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	<p>Build smartbody-openglES</p> <p>Go to trunk/ios/applications/minimal, open smartbody-iphone.xcodeproj, build and run.</p> <p>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</p>
Building smartbody-ogre	
1	<p>Build ogre iphone</p> <ul style="list-style-type: none"> • 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/vn_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 Assets 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	<p>Compile xcode project and run</p> <ul style="list-style-type: none"> • 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_device libraries. After above steps are done, build and run!
---	---

Notes

- smartbody-iphone.xcworkspace needs to change if there's file changes under SmartBody.
- 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 modified maybe needed to make it working.
- **Since python26 has only been built for iOS device, SmartBody won't be available on iPhone simulator for now.**

Building SmartBody for Flash

- Overview
- Requirements
- Compiling and Running
 - Repository files layout
 - Compiling Third Party Libraries
 - Compiling PreLoader and Console
 - Compiling SmartBody and Minimal SmartBody Flash Application
 - Compressing Data for Minimal SmartBody Flash Application
 - Running Minimal SmartBody Flash Application
 - Notes
- Test, future improvements and others
 - Performance
 - Test running on different web browsers
 - Future improvements
 - Embedding in Unity4
 - Compare FlasCC, NaCL and Emscripten

Overview

Adobe Flash's recent effort on FlasCC compiler makes native code running on web browser possible. The Stage3D technology also provides a fully hard-ware accelerated architecture that brings visuals across browsers. With these, SmartBody now can run as a standalone application on browsers that has latest flash plugin properly installed.

Using FlasCC, SmartBody will be compiled into a static library that can be linked by other application. SmartBody has a

simple rendering application distributed in repository, if more complicated requests are needed, you can choose to use game engine that has FlasCC integrated. The best integrated engine is Unreal Engine3, following link shows its capability <http://www.unrealengine.com/flash/>.

Here are the link to our simple flash demo demonstrating the basic capabilities:

<http://smartbody.ict.usc.edu/HTML/videos/flascc/SBFlashDemo-ChrBrad.mp4>

<http://smartbody.ict.usc.edu/HTML/videos/flascc/SBFlashDemo-doctor.mp4>

Requirements

You need to download the latest Flash C++ Compiler(FlasCC SDK) from <https://www.adobe.com/products/gaming/tools.html>, registration will be needed to get into downloading page. If you want to compile samples that comes with FlasCC, you also need to download Flex SDK at <http://www.adobe.com/devnet/flex/flex-sdk-download-all.html>. You can find FlasCC documentation from <http://www.adobe.com/devnet-docs/flascc/>. Following are the instructions of setting up FlasCC.

Installation Step	Windows	Mac
1	Exact the contents of the FlasCC zip file onto anywhere on your hard drive.	Extract the contents of the FlasCC DMG file onto your hard drive
2	Ensure 64-bit Java is installed and Java is on your PATH and pointing to the right directory (so it doesn't confuse with other version you have on your machine)	Ensure 64-bit Java is installed
3	Exact Flex SDK to anywhere on your hard drive	Exact Flex SDK to anywhere on your hard drive
4	Lauch run.bat under FlasCC folder, this is the cygwin environment where all the compiling happens	Launch terminal and get ready to compile

Make sure you also get a debugger for flash player from <http://www.adobe.com/support/flashplayer/downloads.html> both on Windows and Mac.

Compiling and Running

The first thing to do is to change tool chain by adding **sdk/usr/bin** folder to the head of your path like following (change the path accordingly to your flascc directory):

```
Mac> export PATH=~/flascc/sdk/usr/bin:$PATH
Win> export PATH=/cygdrive/c/flascc/sdk/usr/bin:$PATH
```

Second, create a local directory under flascc/sdk/usr so all the third party libraries will be installed there.

Repository files layout

All the files related to flash compiling are under trunk/flash.

- actionscripts: contains all the actionscripts related to console and preloader.
- cmakefiles: contains all the cmake files needed to compile SmartBody and minimal flash demo.
- data: gives an example of how assets would be assembled.

- demo: contains the complete built minimal smartbody flash demo.
- install: includes all the pre-built third party libraries and actionscripts.
- minimalflashapp: inside is the .cpp file for minimal flash demo, it's based on the simplesmartbody.cpp, also export.txt that contains compiling symbols.
- mms.cfg: this file is used to override flash setting on your local machine.

How to use these files are explained in the following sessions.

Compiling Third Party Libraries

You can choose to use pre-built third party libraries from /trunk/flash/install folder, just manually copy them under your /flascc/sdk/usr/local. If you decide to build your own, following are the steps:

	Third Party Libraries Compiling
1	<p>compiling xerces-c</p> <p>Download xerces-c-3.1.1.tar.gz from http://xerces.apache.org/xerces-c/download.cgi</p> <pre>./configure --disable-shared --disable-network --prefix=/flascc/sdk/usr/local</pre> <p>make</p> <p>make install</p>
2	<p>compiling python 2.6</p> <p>Download Python from: http://www.python.org/downloads/releases/2.6.8/</p> <pre>./configure --disable-shared --without-signal-module --prefix=/flascc/sdk/usr/local</pre> <p>make</p> <p>make install</p> <p>During compiling you will meet some errors, following is the changes needed to make.</p> <p><code>./Modules/posixmodule.c: change L_PUSH to TIOCPKT ; include<sys/ioctl.h>; define out BSD block that contains fpgetmask</code></p> <p>Inside Makefile: get rid of build interpreter.</p> <p>Python will be installed but the interpreter won't be working correctly and there's trouble setting Py_Home</p>

3	<p>compiling boost</p> <p>Download Boost from: http://sourceforge.net/projects/boost/files/boost/1.44.0/boost_1_44_0.tar.gz/download</p> <p>add \$TOOLSETROOT to bootstrap.sh at the top</p> <p>change ./build.sh "\$TOOLSET" to ./build.sh "\$TOOLSET" "\$TOOLSETROOT"</p> <p>./bootstrap.sh --prefix=/flascc/sdk/usr/local</p> <p>./bjam</p> <p>./bjam install</p>
4	<p>compiling ODE</p> <p>Download ode-0.11.1 from:http://sourceforge.net/projects/opende/files/</p> <p>./configure --prefix=/flascc/sdk/usr/local</p> <p>make</p> <p>make install</p>
5	<p>compiling GLS3D (OpenGL For Stage3D)</p> <p>Download GLS3D code from https://github.com/adobe/GLS3D.</p> <p>make FLASCC=/flascc/sdk</p> <p>manually copy the include and lib to /flascc/sdk/usr/local</p> <p>More details about use of GLS3D can be found at http://www.adobe.com/devnet/games/articles/compiling-opengl-games.html</p>
6	<p>compiling alcfreeglut (freeglut2.8.0 ported to Flash with FlasCC and Stage3D)</p> <p>Download glut code from https://github.com/alexmac/alcfreeglut</p> <p>make FLASCC=/flascc/sdk GLS3D=/path/to/GLS3D</p> <p>manually copy the include and lib to /flascc/sdk/usr/local</p>

Compiling PreLoader and Console

Console class is the first class that is created and executed when SWF loads, it controls when and how the FlasCC compiled code is initialized and executed.

FlasCC compiled code starts executing when the whole SWF file has been loaded in the browser, it's a best practice to show some kind of loading screen before then. Default PreLoader.swf shows a black screen with a white loading bar that progresses across the screen as the SWF loads. By using HTTP VFS, the zipped data folder will be downloaded during the pre-load.

From your console, navigate to trunk/flash/actionscripts folder, do following command to compile console.abc, PreLoader.swf

make FLASCC=/flascc/usr/sdk

Compiling SmartBody and Minimal SmartBody Flash Application

First, copy all the cmake files from trunk/flash/cmakefiles to the corresponding locations, it should be straightforward by looking at the directory layout.

Second, generate the Makefiles with cmake:

```
mkdir builddirectory
```

```
cmake ..
```

Third, copy simplesmartbody.cpp from trunk/flash/minimalflashapp to trunk/core/smartybody/simplesmartbody, overriding the original one. Copy exports.txt from trunk/flash/minimalflashapp to trunk/<builddirectory>/core/smartybody/simplesmartbody. Copy built Console.abc and PreLoader.swf from trunk/flash/actionscripts to trunk/<builddirectory>/core/smartybody/simplesmartbody.

Go to the build directory, make and install the applications

```
cd builddirectory
```

```
make
```

After make, smartbody-lib.a, smartbody-dll.a and simplesmartbody.swf will be generated.

Compressing Data for Minimal SmartBody Flash Application

You need to zip all the files inside data folder and copy it to trunk/<builddirectory>/core/smartybody/simplesmartbody. Data folder has to follow some standard hierarchy so that application can be launched properly. You can check out an example data folder from trunk/flash/data. Following are some notes for data folder layout.

- data.zip should not contain a data folder. It should contain everything under it.
- it has to have a sbm-common/scripts folder where inside that folder default-init.py will be the startup script.
- at top folder, you will see two .txt files. AnimationList.txt and XMLFileList.txt. AnimationList.txt contains a list of animations that later on one of them will be random picked and played inside application. XMLFileList.txt contains a list of sound files (without extension), one of them will also be random picked.

If you have sound files, it doesn't need to be zipped. You will need to convert them into .mp3 format and put them under the same directory of your final swf.

Running Minimal SmartBody Flash Application

After finishing above steps, you will see a minial smartbody flash application generated under trunk/<builddirectory>/core/smartybody/simplesmartbody.

You can run your flash debugger and drag drop simplesmartbody.swf to it. Another way is to launch SmartBodyDemo.html in various web browser.

Make sure you applies mms.cfg to your local flash setting which extends the ScriptStuckTimeout. You can find mms.cfg from trunk/flash

Windows 32-Bit	%systemroot%\System32\Macromed\Flash
Windows 64-Bit	%systemroot%\SysWOW64\Macromed\Flash
Mac	/Library/Application Support/Macromedia

Notes

- While extracting library under Windows, make sure to use 7-zip instead of Winzip. Winzip extracted files will have format issues with files constructed on *nix system.
- gcc compiler sometimes doesn't like cygwin's way of defining directory. So if "/cygdrive/e" doesn't work, try "E:/"
- While compiling large 3rd party libraries, it sometimes take a lot of memory on java virtual machine. If memory is not assigned enough, it would throw out an exception. To avoid that, passing in a CFLAGS -jvmopt=-Xmx6000M.
- The reason of mms.cfg existing is because the computation cost of SmartBody. Sometimes function overhead is bit big so it takes more than 15 seconds to execute so it would trigger default flash timeout exception. To fix that, SmartBody codes need to be more optimized.
- With different experiments i prefer building 3rd party libraries using Mac and building SmartBody with Windows under Cygwin. Advantages of using Cygwin: fast, you can assign more java virtual machine memory on windows(more often windows has better configuration than mac book). Downsides of Cygwin: sometimes you will find weird compiling errors with 3rd party source codes.
- The assets used for FlasCC SmartBody has to be optimized so that application can run at 60 fps (mainly mesh and texture resolution).
- The minimal application is a combination of c++ native code, flash++ API and action scripts.
- Make sure if you run on web browser, its flash plugin is update to date and properly installed.
- This is minimal version of SmartBody, so some features are disabled, including locomotion, steering, reaching.

Test, future improvements and others

Performance

With a single SmartBody character rendered with textures using CPU deformable geometry where the texture and mesh are low resolution, we are getting 60 fps.

Test running on different web browsers

	Flash Debugger	Google Chrome	Firefox	IE	Safari
Windows	OK	__gnu_cxx::__concurrence_unlock_error	OK	OK	not supported
Mac	OK	__gnu_cxx::__concurrence_unlock_error	OK	not supported	OK

Needs to look into concurrence unlock error.

Future improvements

- Python interpreter is not built correctly, it doesn't affect the usage of python in SmartBody, but it might affect use of Cerebella later on. Need to setup python home correctly so it can load the python modules.
- Some of the functions needs to be optimized or break down into fine pieces so it would not trigger flash's 15 second default timeout exception.
- CPU shading is taking up too much computing time, there's a way to use Stage3D's GPU shader language AGAL to do the mesh binding.

Embedding in Unity4

Unity doesn't support FlasCC directly so you won't be able to link your rendering application to the pre-built static SmartBody library, but theoretically you can load a swf from another swf. Construct a swf file that serves as a connector from Unity rendering application to SmartBody library should be able to achieve it.

Compare FlasCC, NaCL and Emscripten

There's other solutions besides FlasCC. NaCL <https://developers.google.com/native-client/> and Emscripten <https://github.com/kripken/emscripten>.

Emscripten compiles native code into javascript that can be used directly inside html inside any web browser, it has a good amount support for OpenGL in a OpenGL ES standards. But it has its limitations mostly when it comes to multi-threaded codes, full limitation can be found here <https://github.com/kripken/emscripten/wiki/CodeGuidelinesAndLimitations>.

NaCL compiles native code into its own NaCL modules and talks to Browser using Pepper Plug-in API. It supports almost 3D rendering, multi-threading, audio, FileIO etc. It is actively maintained by Google. It right now has integration with Unity4 and Ogre3D. Downsides is obvious, you will be only running it on Google Chrome with NaCL plugin installed.

FlasCC is developed by Adobe to compile native codes into flash. Flash has been existing for a long time thus it's well adopted by almost all the web browser, there's a lot of tools helping developing and debugging flash related application, also it already has a sound community with experienced developer. Downsides flash plugin is always needed and it has to be installed properly.

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.

Appendix 3: Running SmartBody on other platforms

Running SmartBody on an Android device

The SmartBody framework is fully compatible with the Android platform. The SmartBody distribution contains several Android apps that can be run: a simple example of using SmartBody with the Java Native Interface (JNI) called sbjniapp, a simple example of using SmartBody with the Ogre rendering system called sbogreapp, and a SmartBody application integrated into the Unity game engine called sbunity.

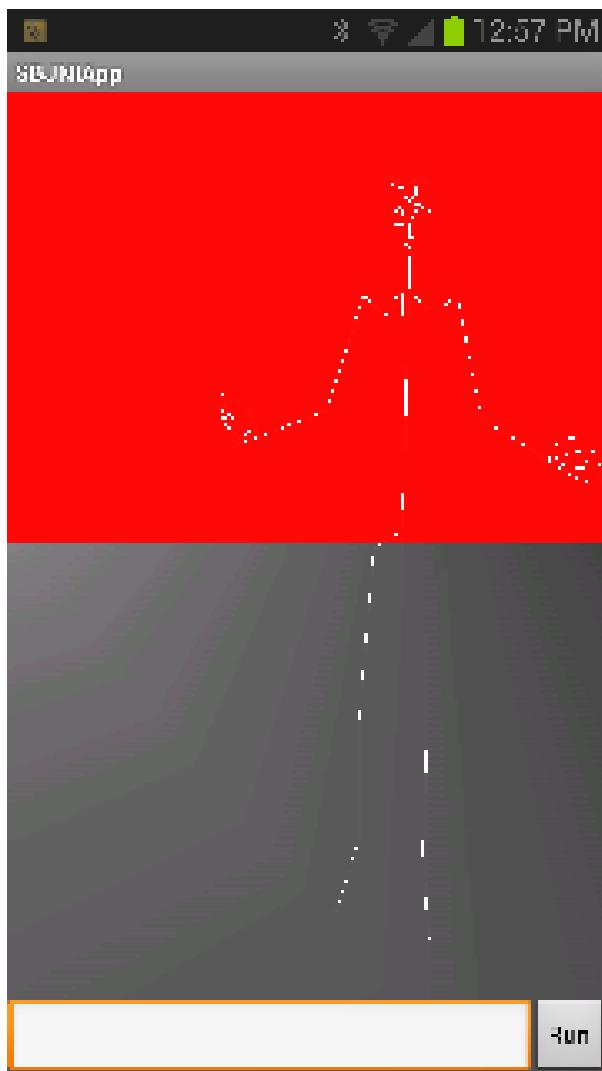
To build any of these applications from the distribution, please see the instructions in the Appendix on [Building SmartBody for Android](#).

Running sbjniapp

The sbjniapp app is a simple example of using SmartBody with the Java Native Interface (JNI) and rendering using OpenGL.

1. Copy the file smartbody/android/sbjniapp/sbjniapp.apk to your Android device.
2. Install the sjniapp.apk app
3. Copy the entire directory smartbody/android/sbjniapp/sbjniappdir to the /sdcard directory on your Android device

Note that the data files under /sdcard/sbjniappdir can be modified. The app will look for the file /sdcard/sbjniappdir/default.py and run the commands contained within it. Any Python script can be substituted for the default.py file which is automatically run by the app. There are no camera controls, but any Python command can be sent to SmartBody by entering it in the input box.



Running sbogreapp

The sbogreapp app is a simple example of using SmartBody with the Java Native Interface (JNI) and rendering using Ogre.

1. Copy the file smartbody/android/sbogreapp/sbogreapp.apk to your Android device.
2. Install the sbogreapp.apk app
3. Copy the entire directory smartbody/android/sbogreapp/sbogreappdir to the /sdcard directory on your Android device

Note that the data files under /sdcard/sbogreappdir can be modified. The app will look for the file /sdcard/sbjniappdir/initOgre.py and run the commands contained within it. Any Python script can be substituted for the initOgre.py file which is automatically run by the app. Swiping the screen rotates the camera. Any Python command can be sent to SmartBody by entering it in the input box. The 'Use Deformable Model' checkbox switches between viewing the character as a rendered mesh and as spheres.



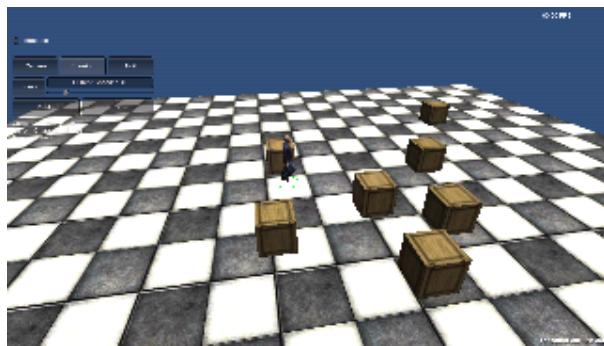
Running SBUnity

The SBUnity app is an example of combination SmartBody with the Unity game engine.

1. Copy the file smartbody/android/SBUnityApp/sbmUnity.apk to your Android device.
2. Install the sbmUnity.apk app
3. Copy the entire directory smartbody/android/SBUnityApp/SbmUnity to the /sdcard directory on your Android device

The app has several controls:

- Press the 'Steering' button, then tap anywhere on the screen and the character will walk to the tapped location. The speed of movement may be changed, and the 'Jump' button will cause the character to jump. 'Pick Up' will cause the character to reach for the nearest ball, and 'Put Down' will cause him to put it down near himself.
- Press the 'Camera' button, then swipe the screen to rotate the camera, pinch to zoom. In Camera mode, the camera may also be changed to 'Character' view (position the camera in front of the character), '3rd Person' (position the camera behind and above the character), 'Main' (view from a controlled location), or 'Reset' (reset the camera to its original position)
- Press the 'Edit' button, then 'Create Box' then tap a location and a box will be created, or 'Create Ball' and a ball will be created at the next tapped location.



Running SmartBody on a web page in Flash

Running SmartBody with Panda3D

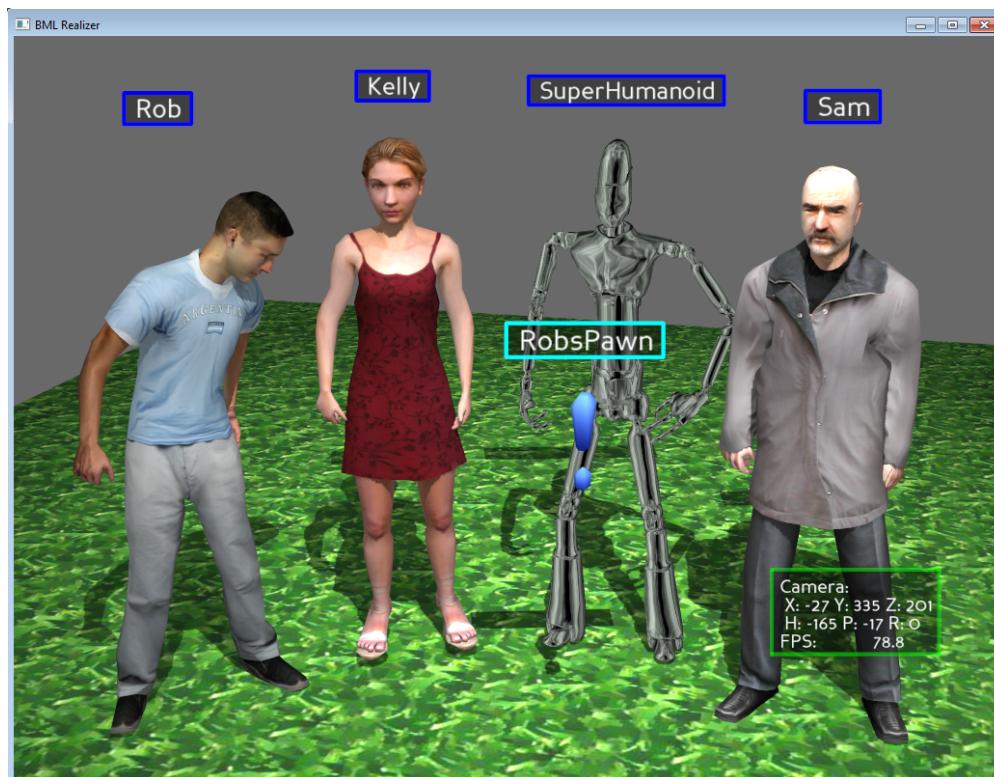
SmartBody can be used with the Panda3D game engine (<http://www.panda3d.org>).

The PandaBMLR interface is a Python-based interface that uses the BoneBus protocol.

The PandaBMLREmbedded interface is a Python-based interface that uses SmartBody as an embedded Python library controlled by Panda. We recommend using the embedded version for better performance.

To use the embedded version:

1. After building SmartBody, copy all the dynamic libraries found in core/smartbody/sbm/ to the lib/Panda3D/python/Lib/site-packages into a directory called SmartBody
2. Change the SmartBody.dll file to be named SmartBody.pyd
3. Add a file called 'SmartBody.pth' in the lib/Panda3D/python/Lib/site-packages directory that contains the word 'SmartBody' to let Python know that there is an external library that needs to be loaded.
4. Run the file core/PandaBMLR/RUN_DemoBasicEmbedded.bat



Running SmartBody with the Ogre Renderer

The SmartBody distribution includes an application called OgreViewer which contains an example of how to run SmartBody either embedded in-process or via a network protocol (BoneBus).

SmartBody characters are constructed and connected to their Ogre counterparts. SmartBody then controls the Ogre characters directly. Note that a custom Ogre interface can be constructed by using the [SmartBody API](#). The OgreViewer application is used as one possible example of how SmartBody and Ogre can communicate together.

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 will likely result in mismatches between the SmartBody skeleton and the Ogre skeleton.

The OgreViewer will run in embedded mode automatically if no command line parameters are entered. By contrast, 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	<p>Use a configuration file that contains startup instructions. Here is an example of the valid contents:</p> <pre>[GENERAL] UseBoneBus=False ScriptPath=../../../../data/sbm-common/scripts DefaultPyFile=default-init.py MediaPath=. Scene=vh_basic_level.mesh</pre> <p>Where 'UseBoneBus' can be 'True' or 'False' 'ScriptPath' is the location of the startup script. 'DefaultPyFile' is the name of the script that will be run on startup. It must be located in the 'ScriptPath' directory. 'MediaPath' is the prefix attached to all asset queries. 'Scene' is an Ogre .mesh file that contains the initial scene geometry</p>
-mediapath	<p>Media path for the application. The media path determines the prefix by which all subsequent paths are added.</p> <p>For example, if the media path is: /my/application, then specifying a relative directory for some file parameter, such as 'myfile', results in the mediapath being added to it. In other words, the final path will be: /my/application/myfile</p>
-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 configuration file 2 = world scene in feet 3 = world scene in centimeters 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 or right
W,S	Move the camera forward or backward
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, and the characters actions will be seen in the Ogre window. Please see the documentation on [running the SmartBody standalone program](#). The OgreViewer itself has no interactive controls, and thus any interactive commands either need to be embedded within the C++ code or sent through via a VHMessage through ActiveMQ. This can be done, for example, via the elsender application that is part of the SmartBody distribution. 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 sbm-common/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 sbgui with the appropriate parameters:

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

Running SmartBody with the Unity Game Engine

The Unity interface to SmartBody is available in the Virtual Human Toolkit available here: <https://vhtoolkit.ict.usc.edu/>

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 subsequent 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

Appendix 4: Using the simplified interface to SmartBody

smartbody-dll provides a simplified interface to SmartBody in a dynamically linked library. This can be used when a C-style API is needed to interface with another component. This interface provides access to the state of the SmartBody scene, but does not provide low level access directly via C-functions. Instead, it provides a set of C-functions that call Python functions that can be used to access the internals of SmartBody.

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(currentTimeInSeconds);
```

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.
OnChannel1	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

