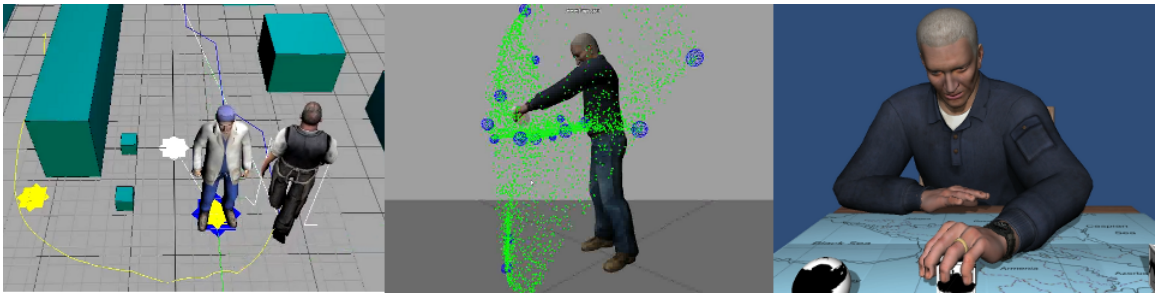
SmartBody



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

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

SmartBody Manual

Updated 2/7/12

Ari Shapiro, Ph.D.

shapiro@ict.usc.edu

Overview

SmartBody is a character animation platform originally developed at the University of Southern California.

SmartBody provides the following capabilities in real time:

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

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

- Unity
- Ogre
- Panda3D
- GameBryo
- Unreal

It is straightforward to adopt SmartBody to other game engines as well using the API.

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

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

The SmartBody website is located at:

http://smartbody.ict.usc.edu

The SmartBody email group is located here:

smartbody-developer@lists.sourceforge.net

You can subscribe to it here:

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

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

Ari Shapiro, Ph.D.

shapiro@ict.usc.edu

Contributors

SmartBody Team Lead	**Ari Shapiro, Ph.D.**
SmartBody Team	Andrew W. Feng, Ph.D. Yuyu Xu
Inspiration & Guru	Stacy Marsella, Ph.D.

Past SmartBody Team Members	Marcus Thiebaux Jingqiao Fu Andrew Marshall Marcelo Kallmann, Ph.D.
SmartBody Contributors	Ed Fast Arno Hartholt Shridhar Ravikumar Apar Suri Adam Reilly Matt Liewer

Downloading SmartBody

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

Use the SVN command:

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

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

Windows

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

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

Linux

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

```
apt-get install subversion
```

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

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

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

```
svn update
```

Mac/OsX

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

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

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

```
svn update
```

Building SmartBody

There are several SmartBody applications that can be built:

Application	Comment
sbm-fltk	a standalone SmartBody application and scene renderer
smartbody-dll	a dynamic library that can be incorporated into a game engine or other application
sbm-batch	a standalone SmartBody application that connects to smartbody-dll without a renderer
SbmDebuggerGui	(*Windows only*) a debugger/visualizer for running SmartBody processes
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

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

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

Building SmartBody for Windows

You will need Visual Studio 2008 or higher to build SmartBody. In the top level SmartBody directory, Open the solution file 'vs2008.sln' 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++
liblog4cxx10-dev
libxerces-c3-dev
libgl1-mesa-dev
libglu1-mesa-dev
libglut-mesa-dev
xutils-dev
libxi-dev
freeglut3-dev
libglut3
libglew-dev
libxft-dev
libapr1-dev
libaprutil1-dev
libcppunit-dev
liblapack-dev
libblas-dev
build-essential
mono-devel
mono-xbuild
python-dev
libopenal-dev
libsndfile-dev
libalut-dev

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

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

Linux packages needed for test suite
imagemagick

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

To build the Linux version:

1	Install the packages indicated above	

2	Download and install boost:
	
	Download Boost from: http://sourceforge.net/projects/boost/files/boost/1.44.0/boost_1_44_0.tar.gz/download
	
	Place in lib/ and unpack, build and install into /usr/local using:
	
	`./boostrap.sh`
	`./bjam --with-pythonsudo`
	`./bjam install`	

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

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

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

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

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

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

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

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

Building SmartBody for OSX

To build the OSX version:

1	Download and install boost:
	
	Download Boost from: http://sourceforge.net/projects/boost/files/boost/1.44.0/boost_1_44_0.tar.gz/download
	
	Place in lib/ and unpack, build and install into /usr/local using:
	
	`./boostrap.sh`
	`./bjam --with-pythonsudo`
	`sudo ./bjam install`	

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

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

4 Download and install the ActiveMQ libraries.

Download from: http://www.apache.org/dyn/closer.cgi/activemq/activemq-cpp/source/activemq-cpp-library-3.4.0-src.tar.gz

Activemq-cpp will need to be changed slightly to work with OsX, as seen in this bug fix:

https://issues.apache.org/jira/browse/AMQCPP-369

Change the following in the activemq-cpp source code:

```
--- activemq/activemq-cpp/trunk/activemq-cpp/src/main/decaf/util/logging/Handler.h    2011/05/02 14:52:26    1098605
+++ activemq/activemq-cpp/trunk/activemq-cpp/src/main/decaf/util/logging/Handler.h    2011/05/02 14:52:30    1098606
@@ -49,9 +49,6 @@
    class DECAF_API Handler : public io::Closeable {
    private:


-       // Default Logging Level for Handler
-       static const Level DEFAULT_LEVEL;
-
        // Formats this Handlers output
        Formatter* formatter;


--- activemq/activemq-cpp/trunk/activemq-cpp/src/main/decaf/util/logging/Handler.cpp    2011/05/02 14:52:26    1098605
+++ activemq/activemq-cpp/trunk/activemq-cpp/src/main/decaf/util/logging/Handler.cpp    2011/05/02 14:52:30    1098606
@@ -28,11 +28,8 @@
using namespace decaf::util::logging;


////////////////////////////////////////////////////////////////////////////////
-const Level Handler::DEFAULT_LEVEL = Level::ALL;
-
-////////////////////////////////////////////////////////////////////////////////
Handler::Handler() : formatter(NULL), filter(NULL), errorManager(new ErrorManager()),
-                    level(DEFAULT_LEVEL), prefix("Handler") {
+                     level(Level::ALL), prefix("Handler") {
}


Then rebuild activemq-cpp using:

./configure --disable-ssl
make
sudo make install
sudo ldconfig
```

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

6	Build and install xerces http://www.takeyellow.com/apachemirror//xerces/c/3/sources/xerces-c-3.1.1.tar.gz	
7	Build and install GLEW https://sourceforge.net/projects/glew/files/glew/1.6.0/glew-1.6.0.tgz/download	
8	Build and install log4cxx http://www.apache.org/dyn/closer.cgi/logging/log4cxx/0.10.0/apache-log4cxx-0.10.0.tar.gz	
9	Build and install OpenAL http://connect.creativelabs.com/openal/Downloads/openal-soft-1.13.tbz2	
10	Build and install FreeALUT http://connect.creativelabs.com/openal/Downloads/ALUT/freealut-1.1.0-src.zip	
11	Build and install libsndfile http://www.mega-nerd.com/libsndfile/files/libsndfile-1.0.25.tar.gz	

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

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

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

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

Building SmartBody for Android

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

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

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

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

6	(Optional) Build vh-wrapper vh-wrapper is a SmartBody interface to Unity3D. Note that SmartBody connects to Unity via Unity's native code plugin interface, which presently requires a Unity Pro license. In addition, the Unity project needs to be compiled for Android, so a Unity Android license is needed as well. a. Go to SmartBody/android/vh_wrapper/ b. ndk-build c. rename libvhwrapper.so to libvhwrapper-dll.so (for some reason, Android does not accept a build target name with "-") d. copy libvhwrapper-dll.so to the plug-in directory of Unity project. Build the Unity project for Android.

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

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

4	Building clapack TODO

Building SmartBody for iOS

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

	Compiling using console

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

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

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

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

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

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

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

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

	Compiling using Xcode4
1	Build bonebus • Open smartbody-iphone.xcworkspace, select the scheme to be bonebus, build
2	Build boost • Download boost_1_44_0.tar.gz from http://www.boost.org/users/history/version_1_44_0.html and unzip to trunk/ios/boost. Make sure the folder name is boost_1_44_0. • Open smartbody-iphone.xcworkspace, select boost_system, boost_filesystem, boost_regex, build them seperately. • Download boost_numeric_bindings from http://mathema.tician.de/news.tiker.net/download/software/boost-numeric-bindings/boost-numeric-bindings-20081116.tar.gz and unzip it to trunk/ios/boost, make sure the name is boost_numeric_bindings

3	Build steersuite • Open smartbody-iphone.xcworkspace, select the steerlib, pprAI, build them seperately.
4	Build vhmsg • Open smartbody-iphone.xcworkspace, select scheme vhmsg and build.
5	Build vhcl • Since the vhcl_log.cpp hasn't been changed from VH group, you have to copy trunk/ios/vhcl/vhcl_log.cpp to trunk/lib/vhcl/src/vhcl_log.cpp for now. • Open smartbody-iphone.xcworkspace, select scheme vhcl and build.
6	Build wsp • Open smartbody-iphone.xcworkspace, select scheme wsp and build.
7	Build smartbody-lib • Open smartbody-iphone.xcworkspace, select scheme smartbody-lib and build. P.S. This xcode project needs maintenance by the developer to make sure it incorporates all the file from SmartBody core.
8	Build smartbody-dll (Not required) • Open smartbody-iphone.xcworkspace, select scheme smartbody-dll and build.
9	Build vhwrapper-dll (Not required, for Unity only) • Open smartbody-iphone.xcworkspace, select scheme vhwrapper-dll and build.

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

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

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

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

	Building smartbody-ogre

1	Build ogre iphone • Download OgreSDK: http://www.ogre3d.org/download/sdk • Build the Ogre iPhone libraries as indicated here: http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Building%20From%20Source%20-%20iPhone&redirectpage=Building%20From% • 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 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 wrong with the texture and shader. So maybe should just run on armv7 iPhone/iPad

	Building smartbody-unity
1	Check out the project • Currently all the Smartboy Unity project for IOS is sitting inside vh repository at https://svn.ict.usc.edu/svn_vh/trunk/lib/vhunity/samples/smartbody_mobile_minimal and https://svn.ict.usc.edu/svn_vh/trunk/lib/vhunity/samples/smartbody_mobile • smartbody mobile minimal includes only smartbody while smartbody mobile can include all the other components like face detection, audio acquisition etc.
2	Build unity project into xcode • Open any scene from Assests e.g. smartbdoySceneOutDoor.unity • Copy the static libraries inside (SmartBody)trunk/ios/libs/iphoneos to (Unity)Assets/Plugins/iOS. Do not include liblapack.a libf2c.a libblas.a, these would cause duplicated symbol problems. Make sure you are using libactivemq-cpp.a for unity which you can get from trunk/ios/activemq-cpp/libs/activemq-cpp-unity. • Make sure your unity is under iOS platform. • Go to build setting, set the ios platform application name, resolution etc and hit build button.
3	Compile xcode project and run • Add Accelerate.framework(which contains the lapack library) into xcode project. • If you are using smartbody mobile minimal scenes, build and run! • If you are using smartbody mobile scenes. You will need to include CoreVideo.framework, CoreGraphic.framework, OpenAL.framework. Also make sure that in the xcode project setting, include the header search path for opencv_device; in the Libraries folder, drag in opencv_divice libraries. After above steps are done, build and run!

SmartBody iOS build maintenance

This piece of documentation is written mainly for the developer.

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

changed outside, they have to be copied over again and modifies maybe needed to make it working.

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. The following is a brief list of capabilities and requirements for them to work:

Capability	Requirements
Head movements: nodding, shaking, tossing	Skeleton with 3 joints in spine and neck named *spine4*, *spine5* and *skullbase*
Gazing	Skeleton with the following joints: *spine1*, *spine2*, *spine3*, *spine4*, *spine5*, *skullbase*, *face_top_parent, eyeball_left*, *eyeball_right*
Lip syncing	Face Definition that includes viseme definitions. For joint-driven faces, one motion per viseme and one neutral face motion. For shape-driven faces, only configuration of Face Definition is necessary.
Face movements	Face Definition that includes Action Unit (AU) definitions. For joint-driven faces, one motion per Action Unit and one neutral face motion. For shape-driven faces, only configuration of Face Definition is necessary.
Locomotion	Motion examples of movement that match the character's skeleton.
Idling	Motion files that match the topology of the character's skeleton.
Animations	Motion files that match the topology of the character's skeleton.
Gestures	A gesture mapping and motions files that match the topology of the character's skeleton.
Text-to-speech (TTS)	Either built-in Festival TTS relay, or an external relay using Festival, Microsoft Speech, or any other TTS engine.
Eye saccades	Skeleton with the following joints: *eyeball_left*, *eyeball_right*
Grabbing, touching, reaching	Skeleton topology where shoulder names are: *l_sternoclavicula*r and *r_sternoclavicula*r. Also, a Reach Configuration
Constraints	None
Physics	Skeleton with root joint called *base*.
Softeyes	Skeleton with the following joints: *eyeball_left*, *eyeball_right*
Blinking	Either a Face Definition that includes a viseme called *blink*, or a Face Definition that includes Action Unit 45 definition. For joint-driven faces, one blink motion and one neutral face motion. For shape-driven faces, only configuration of Face Definition is necessary.

Configuring Visemes and Facial Animations

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

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

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

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

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

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

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

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

Details

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

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

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

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

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

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

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

To define a set of visemes, 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. For example, many text-to-speech relays (Microsoft, Festival, Cerevoice, etc.) will use the following set of visemes:

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

whereas FaceFX will use the following for prerecorded audio:

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

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

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")
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 Breathing

Configuring Gestures

Configuring Locomotion

Configuring Reaching, Grabbing and Touching

Configuring Soft Eyes

Configuring Speech

Controlling Characters with BML

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

About BML

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

How to Specify a BML Command

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

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

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

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

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

Quick BML Reference for SmartBody

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

Behavior	Example	BML Command
Gaze	look at the object called 'table'	`<gaze target="table"/>`

Locomotion	move to location (10, 75)	`<locomotion target="10 75"/>`
Head Movement	nod your head	`<head type="NOD"/>`
Idle	assume an idle posture called 'idling_motion1'	`<body posture="idling_motion1'/>`
Animation	play an animation called 'dosomething'	`<animation name="dosomething"/>`
Gesture	point at character1	`<gesture type="POINT" target="character1"/>`
Reach	Grab the object 'cup'	`<sbm:reach target="cup"/>`
Constraint	Constraint your hand to 'ball'	`<sbm:constraint target="ball"/>`
Face	Raise your eyebrows	`<face type="FACS" au="1" side="both" amount="1"/>`
Speech	Say 'hello, how are you?'	`<speech type="text/plain">hello how are you?</speech>`
Eye saccade	Move your eyes around automatically	`<saccade mode="LISTEN"/>`
Event	Send out an event 3 seconds in the future	`<sbm:event stroke="3" message="sbm echo hello"/>`

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

Timing BML Commands

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

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

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

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

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

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

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

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

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

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

Head Movement	start ready stroke relax end	ready = ramp-in time stroke = middle of head movement relax = ramp-out time
Idle	start ready	
Animation	start ready stroke relax end	ready = ramp-in time stroke = emphasis point of the animation relax = ramp-out time
Gesture	start ready stroke relax end	ready = ramp-in time stroke = emphasis point of the animation relax = ramp-out time
Reach	start	
Constraint	start ready	ready = time needed to achieve constraint
Face	start ready stroke relax end	ready = ramp-in time stroke = emphasis point of the face motion relax = ramp-out time
Speech	start ready stroke relax end	start, ready, stroke = time of first word spoken relax, end = time of last word spoken
Eye saccade	..?	

Compounding & Synchronizing BML Commands

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

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

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

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

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

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

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

Synchronizing Multiple Character's Behaviors with BML

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

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

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

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

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

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

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

BML Behaviors

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

Each behavior listed on the following pages will contain:

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

Idling or Posture

Description

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

Requirements

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

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

Usage

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

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

Parameters

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

Animation

Description

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

Requirements

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

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

Usage

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

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

Parameters

Parameter	Description	Example
start	starts the motion at a time in the future	`<animation name="motion1" start="3"/>`
ready	the time when the motion is fully blended. The total ramp-in time is (ready-start)	`<animation name="motion1" start="3" ready="5"/>`
stroke	for a gesture, the time of the emphasis point of a gesture	`<animation name="motion1" stroke="4"/>`
relax	time when the animation starts to fade out	`<animation name="motion1" relax="5"/>`

end	indicates when the animation ends	`<animation name="motion1" end="8"/>`
time	multiplier for the speed of the animation (2x plays the animation twice as fast). Note that specifying a time multiplier will automatically adjust the synchronization points.	`<animation name="motion1" time="2"/>`

Timewarping Motions

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

1. If only one synchronization point is specified, align the behavior such that the behavior will occur at normal speed in line with the synchronization point. Example, let's assume that motion1 lasts for 3 seconds, with it's stroke point at second 2:
   `<animation name="motion1" stroke="5"/>`

   will play *motion1* such that the middle (or stroke) of the motion occurs at second 5, with the rest of the motion aligned to that time. In other words, the beginning of the motion will play at second 4, and finish at second 6.

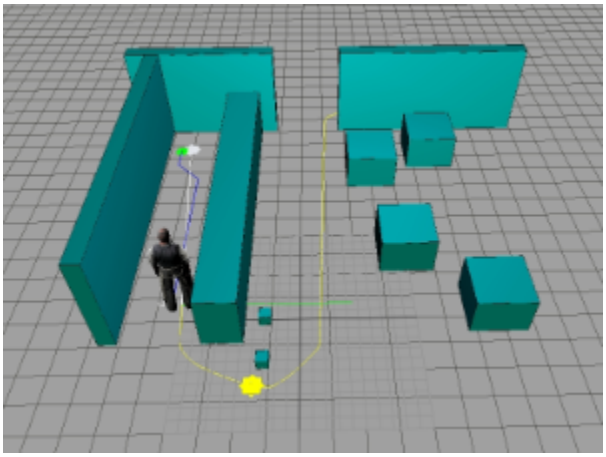
2. If two synchronization points are specified, then timewarp the rest of the motion according to the relative scale of two synchronization points. For example:
   `<animation name="motion1" start="1" stroke="5"/>`

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

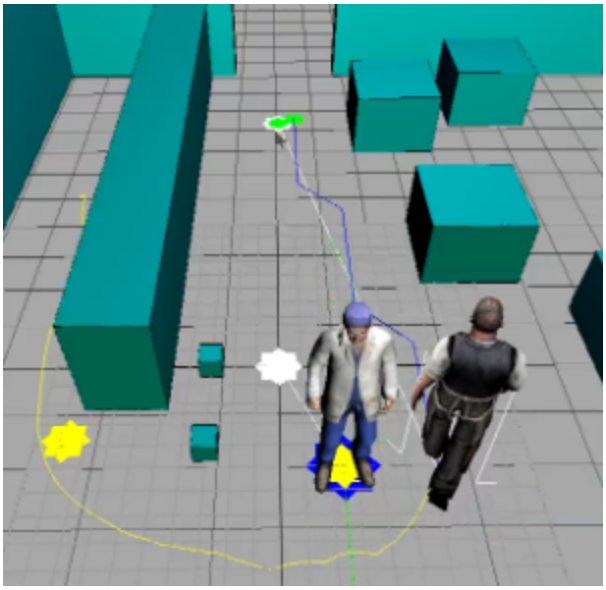
Locomotion

Description

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



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



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

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

Requirements

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

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

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

Parameters

Parameter	Description	Example
target	move to an (x,z) location in the world	`<locomotion target="100 300"/>`
type	Type of locomotion to be used: basic, example, procedural.	`<locomotion target="100 300" type="basic"/>`
manner	Manner of movement: walk, jog, run, sbm:step, sbm:jump	`<locomotion target="100 300" type="basic" manner="sbm:step"/>`
facing	Final facing direction in global coordinates in degrees of the character after locomotion finishes	`<locomotion target="100 300" facing="90"/>`
sbm:follow	Instructs a character to follow another character as it moves.	`<locomotion sbm:follow="utah"/>`

proximity	How close the character should come to the goal before finishing the locomotion	`<locomotion target="100 300" proximity="75"/>`
sbm:accel	Acceleration of movement, defaults to 2	`<locomotion target="100 300" sbm:accel="4"/>`
sbm:scootaccel	Acceleration of sideways (scooting) movement, defaults to 200	`<locomotion target="100 300" sbm:scootaccel="300"/>`
sbm:angleaccel	Angular speed acceleration, defaults to 450	`<locomotion target="100 300" sbm:angleaccel="600"/>`
sbm:numsteps	Number of steps to take, defaults to 1	`<locomotion target="100 300" sbm:numsteps="2"/>`

Gesture

Description

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

Requirements

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

Usage

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

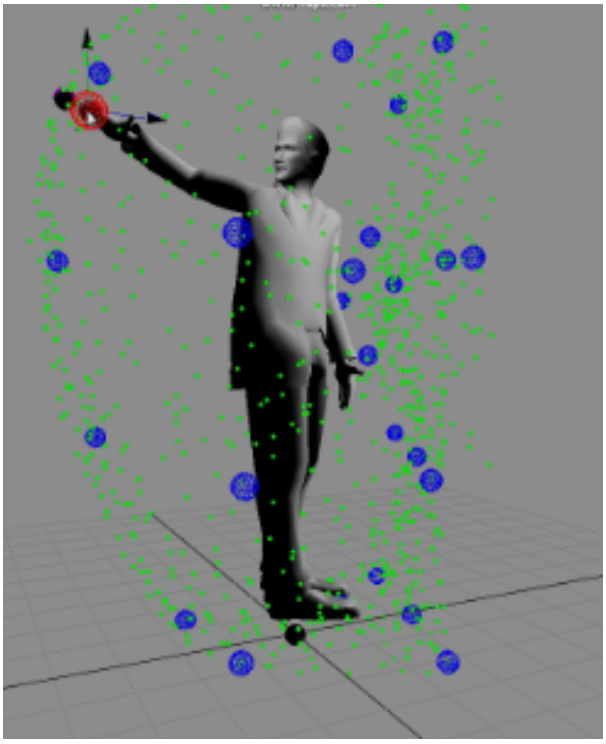
Parameters

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

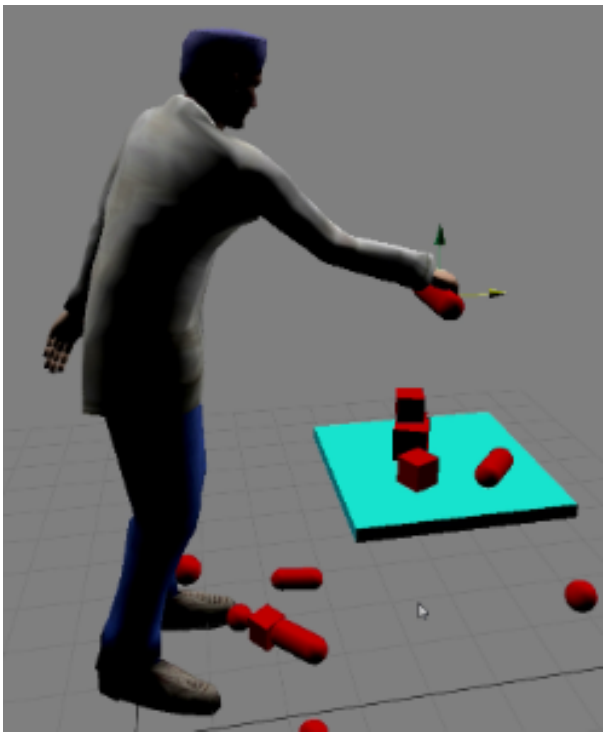
Reach, Grab, Touch

Description

Characters can reach, touch or grab 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.

Please see the section on Configuring Reaching and Grasping For Character 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"/>
```

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	`<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"/>`

Gaze

Description

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

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

Requirements

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

Usage

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

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

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

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

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

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

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

Parameters

Parameter	Description	Example

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

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

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

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

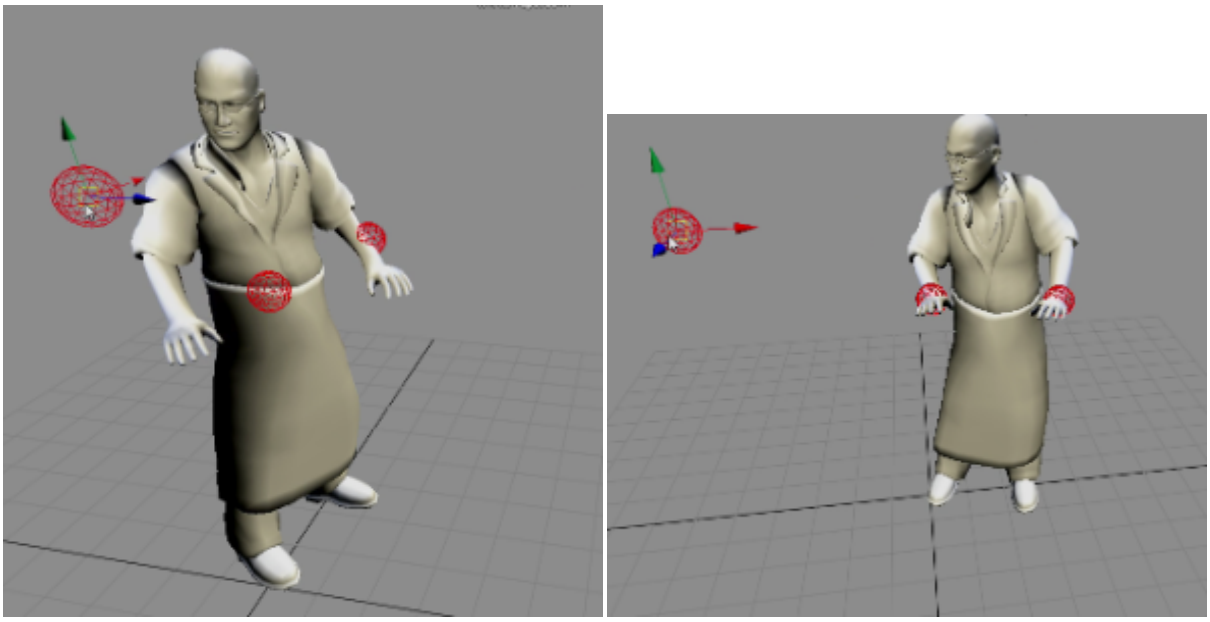
sbm:joint-speed	Overall task speed for NECK or NECK EYES.
	
	Default values for HEAD and EYES are 1000. Specifying one parameter changes the NECK speed.
	
	Specifying two parameters changes the HEAD and EYES speed.
		`<gaze target="utah" sbm:joint-speed="500"/>`
		
		`<gaze target="utah" sbm:joint-speed="800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

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

Constraint

Description

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



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

Requirements

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

Usage

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

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

Parameters

Parameter	Description	Example
target	target pawn/joint whose the positional or rotational values will be used as constraints	`<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist"/>`
sbm:handle	Handle of this constraint instance, can be reused during later constraint commands.	`as above`
sbm:root	the root joint for the current character. by default it is the base joint.	`<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist" sbm:root="base"/>`

effector	the end effector where the constraint will be enforced through optimization	as above
sbm:effector-root	the influence root of this end effector. anything higher than this root will not be affected by optimization	as above
sbm:fade-in	the time for the constraint to blend in. this option is ignored if set to negative	as above
sbm:fade-out	the time for the constraint to blend out. this option is ignored if set to negative	<sbm:constraint sbm:handle="myconstraint" sbm:fade-out="1'/>
sbm:constraint-type	the constraint type to be enforced. it can be positional or rotational constraint	<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist" sbm:root="base" sbm:constraint-type="pos"/>
pos-x	the x positional offset added on top of positional constraint	<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist" sbm:root="base" sbm:constraint-type="pos" pos-y="10" pos-x="15" pos-z="5"/>
pos-y	the y positional offset added on top of positional constraint	as above
pos-z	the z positional offset added on top of positional constraint	as above
rot-x	the x rotational offset added on top of positional constraint	<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist" sbm:root="base" sbm:constraint-type="pos" rot-y="10" rot-x="15" rot-z="5"/>
rot-y	the y rotational offset added on top of positional constraint	as above
rot-z	the z rotational offset added on top of positional constraint	as above
start	when to start the constraint	<sbm:constraint effector="r_wrist" sbm:effector-root="r_sternoclavicular" sbm:fade-in="1" sbm:handle="myconstraint" target="elder:l_wrist" sbm:root="base" start="3" ready="4"/>
ready	when the constraint is fully engaged	as above

Head Movements

Description

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

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

Requirements

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

Usage

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

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

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

Parameters

Parameter	Description	Example
type	Type of head movement: NOD (up-down), SHAKE (left-right), TOSS (side-side) WIGGLE and WAGGLE (multiple nods of varying intensity)	`<head type="NOD"/>`
repeats	number of head movements	`<head type="NOD" repeats="2"/>`
amount	magnitude of head movement, from 0 to 1, default is .5	`<head type="NOD" amount=".8"/>`
sbm:smooth	smoothing parameter when starting and finishing head movements	
sbm:period	period of nod cycle, default is .5	`<head type="NOD" sbm:period=".5"/>`
sbm:warp	warp parameter for wiggle and waggle, default is .5	`<head type="WIGGLE" sbm:warp=".8"/>`

sbm:accel	acceleration parameter for wiggle and waggle, default is .5	`<head type="WIGGLE" sbm:accel=".9"/>`
sbm:pitch	pitch parameter for wiggle and waggle, default is 1	`<head type="WAGGLE" sbm:pitch=".8"/>`
sbm:decay	decay parameter for wiggle and waggle, default is .5	`<head type="WAGGLE" sbm:decay=".7"/>`
start	when the head movements starts	`<head type="NOD" start="2"/>`
ready	preparation phase of head movement	`<head type="NOD" start="2" ready=".5"/>`
stroke	the mid-point of the head movement	`<head type="NOD" stroke="3"/>`
relax	finishing phase of head movement	`<head type="NOD" relax="4"/>`
end	when the head movement ends	`<head type="NOD" end="5"/>`

Face

Description

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

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

Requirements

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

Usage

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

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

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

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

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

Parameters

Parameter	Description	Example
type	Type of facial expression. Only 'facs' is currently supported.	`<face type="facs" au="1" amount="1"/>`
side	Which side of the face will be activated; LEFT, RIGHT or BOTH	`<face type="facs" au="1" side="LEFT" amount="1"/>`
au	Action Unit number	`<face type="facs" au="26" amount="1"/>`
start	when the face movements starts	`<face type="facs" au="26" amount="1" start="2"/>`

ready	when the face movement is fully blended in, default is .25 seconds	`<face type="facs" au="26" amount="1" start="0" ready="1"/>`
stroke	the mid-point of the face movement	`<face type="facs" au="26" amount="1" stroke="4"/>`
relax	when the face movement starts blending out, default is .25 seconds	`<face type="facs" au="26" amount="1" relax="2"/>`
end	when the face movement ends	`<face type="facs" au="26" amount="1" start="0" end="4"/>`

Speech

Description

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

Requirements

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

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

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

Usage

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

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

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

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

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

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

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

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

Synchronizing Speech Using Prerecorded Audio

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

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

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

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

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

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

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

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

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

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

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

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

</speech>
</bml>
```

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

Parameters

Parameter	Description	Example

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

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

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

Eye Saccade

Description

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

Requirements

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

Usage

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

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

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

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

Parameters

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

direction	polar direction of eye movement, from -180 to 180 degrees	`<saccade direction="45" magnitude="15" sbm:duration=".5"/>`
magnitude	amount of eye saccade movement in degrees	`<saccade direction="-45" magnitude="10" sbm:duration=".5"/>`
sbm:duration	duration of eye saccade	`<saccade direction="0" magnitude="15" sbm:duration="1"/>`

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

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

Event

Description

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

Requirements

None

Usage

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

This will send out an event at time 0 to the VH message bus that will be received by SmartBody (which handles all messages with an 'sbm' prefix) and then print out the word 'hello'.

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

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

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

Parameters

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

sbm:states

Description

Requirements

Usage

Parameters

Character Physics

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

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

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

Character Physics Features

The current supported features include :

- **Ragdoll Animations :** The character will naturally die down based on gravity and other collisions. The future extension includes adding mechanisms for the character to naturally get back from a lie down pose.

- **Proportional Derivative (PD) Motion Tracking :** The character will track an input kinematic motion as much as possible. This enables the character to execute a motion under physical simulation, and at the same time respond directly to external push or collisions. The current implementation does not control the character balance. Instead, the root joint is driven directly by the kinematic motion to prevent character from falling down.

- **Kinematic Constraint :** A physics-based character is simulated by a set of connected body links. Instead of using physical simulation, each body link can also be constrained to follow a kinematic motion. This allows the user to "pin" a body link in a fixed position, or have a specific link to follow the original kinematic motion.

- **Collision Events :** A collision event will be sent out from the simulation system when a physics character is collided with other objects. User can provide a python script to handle this collision. For example, the character can gaze at the object that is hitting him, or execute a reaction motion. This helps enhance realism and interactivity of a character with the virtual environment.

How to Setup a Physics Character

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

The overall procedural of setting up a physics character :

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

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

    b. phyManager.createPhysicsCharacter("charName");

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

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

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

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

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

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

Setting Physics Parameters

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

1. **gravity :** A typical gravity should be 9.8 m/s^2. In practice, it should be set according to the unit currently used. For example, if your character is created in centimeter (cm), then you should set gravity to 980 instead of 9.8.
   To set the gravity with Python script :
    a. phyEngine = phyManager.getPhysicsEngine(); # get the physics engine currently used
    b. phyEngine.setDoubleAttribute("gravity", valueOfGravity); # set the gravity value.
2. **dt :** The time step taken for each physics update. Physics engine usually requires small time steps for the simulation to be stable. For simple features like rigid body dynamics or ragdoll simulation, the time step can be as large as the screen refresh rate (60 Hz). For more

advanced features like pose tracking, the time step needs to be much smaller (~ 1000Hz) to maintain a stable simulation.
To set the time step :

a. phyEngine = phyManager.getPhysicsEngine(); # get the physics engine currently used
b. phyEngine.setDoubleAttribute("dT", value); # set the dt value. It must be positive.

3. **Ks :** This parameter determines how strong is the pose tracking. The higher this parameter is, the more closely the physical simulation will match the kinematic motion. The higher value also gives the character a more "rigid" feeling when there are external forces or collisions. However, setting this parameter too high would cause the physical simulation to become unstable. This will in turn require dt to be decreased to avoid such instability. In general, it should be set to counter the gravity so the character can just stay upright and follow the kinematic motion.
To set Ks :
    a. phyEngine.setDoubleAttribute("Ks", value); # set the Ks value. It must be positive.

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

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

6. **KScale :** This parameter scale the Ks, Kd for a specific joint. This allows different Ks, Kd setting for each joint. Since each body link may have different mass and different number of descendent body links, it would not be feasible to use a single Ks, Kd for pose tracking. For example, a shoulder may need a larger Ks to generate more torque so the arm can be lifted to match the desired pose, while the wrist will need smaller values. The general guideline for setting this parameter should be according to the effective mass for the joint – the sum of mass from all descendent body links.
To set KScale :
    a. phyJoint = phyEngine.getPhysicsJoint("charName","jointName"); # get corresponding physics joint
    b. phyJoint.setDoubleAttribute("KScale", scaleValue); # set the KScale. It must be positive.

Setting Up Constraint

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

To setup constraints :

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

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

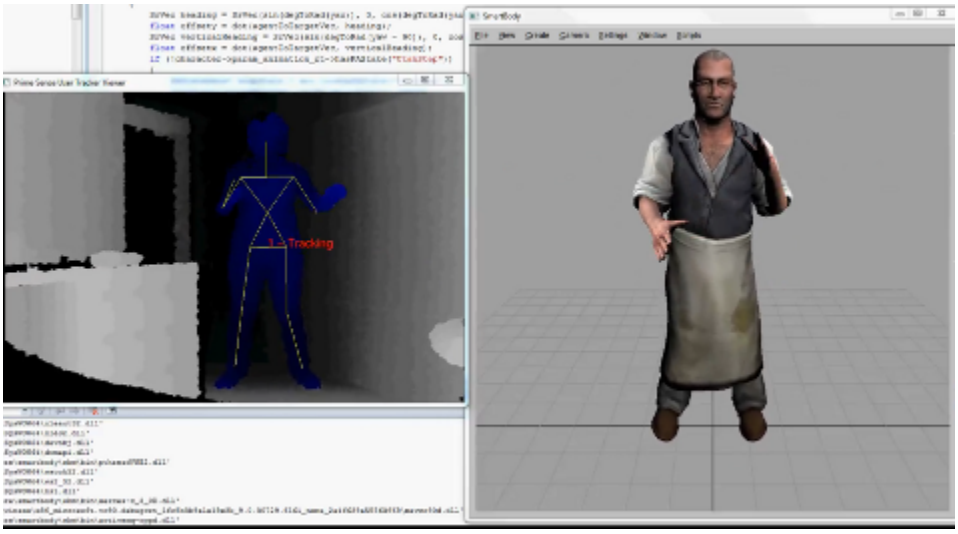
Note that although the physical simulation will try to accommodate user specified constraints as much as possible, it can not handle constraints that conflict the character setup. For example, if user sets up constraints that control each hand, the distance between two constraints can not exceed the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

Using SmartBody With Kinect

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

1. A standalone application reads the data from the Kinect
2. The standalone application sends a message over the ActiveMQ network containing joint positions and orientations
3. SmartBody receives the ActiveMQ message, then overrides the joint position and orientations of the joints of one or more characters

4. Any joints not overridden (such as the face and fingers) will retain the existing character positions and orientations.



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

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

Building Kinect Application on Windows

1	Download and install OpenNI 32-bit Development Edition http://www.openni.org/Downloads/OpenNIModules.aspx Select 'OpenNI Binaries' , 'Stable', then choose the Windows x86 32-bit Development Edition
2	Download and install NITE http://www.openni.org/downloadfiles/ Select 'OpenNI Compliant Middleware Binaries' , 'Stable', then choose the PrimeSense NITE Windows x86 Development Edition
3	Download the Kinect drivers https://github.com/avin2/SensorKinect/blob/unstable/Bin/SensorKinect091-Bin-Win32-v5.1.0.25.msi
4	

Building Kinect Application on Linux

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

	Instructions

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

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

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

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

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

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

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

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

Building Kinect Application on OSX

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