

Standardized Prototyping and Development of Virtual Agents

Technical Report
NWU-EECS-10-10

by

Alex S. Hill

for

Justine Cassell
Articulab Laboratory
School of Communication and
McCormick School of Engineering
Northwestern University

November 25, 2008

Abstract

Research efforts that require the use of virtual agents face significant challenges in prototyping and developing virtual agents. Not only do virtual agents require significant technical and artistic skills, but they also must frequently be modernized as the underlying operating systems, rendering libraries and system hardware become more advanced. In order to mitigate these challenges, we are cooperating with an international effort to standardize the development of virtual agents. Some recent products of this effort are a standard for behavior realization called the Behavior Markup Language and systems designed to implement it. We have taken one such system, a combination of the SmartBody realizer and the Panda3D rendering engine, and modified it to meet the needs of a research agenda that includes significant interactions between agents and their surroundings. These modifications include the addition of viseme support and the development of new language tags for grasping objects and executing other more abstract actions in the virtual world. Our resulting system allows us to realize agents that move their lips to either pre-recorded or text-to-speech utterances and that manipulate objects such as Lego blocks in the virtual world. These behaviors can be applied to any of our agent character models and easily prototyped through behavior markup scripts. While the current system is limited to using key-framed gestures, we have some short-term strategies that will allow agents to place objects at arbitrary locations. Our long-term plan is to incorporate procedural controllers into the SmartBody realizer to facilitate referential gestures such as grasping or pointing at objects located arbitrarily in the virtual world.

Introduction

Virtual agents are simulated characters that interact with human beings. Previous research undertaken by members of our lab has involved agents that help users select a home, give directions on campus and engage children in storytelling. The development of these virtual agents presents a significant technical and artistic challenge. Virtual agents require a sophisticated mix of graphics, audio, sensing and cognitive technologies that can rarely run on the same machine, under the same operating system or in the same language. In an effort to make the prototyping and development of virtual agents more manageable, an international effort has begun to standardize interfaces between constituent parts identified in the agent realization process. Our goal has been to support this effort by embracing and augmenting the software it has produced and to make contributions to a standardization effort that reflect the goals of our particular research.

One recent work developed in our lab was NUMACK, Northwestern University Multimodal Autonomous Conversational Kiosk [1]. This highly successful effort generated an agent capable of generating gestures that typical direction givers make while explaining how to find campus buildings. The resulting system was a combination of several different operating systems, programming languages and

hardware configurations running a completely autonomous system. The significant complexity of such a system can make appropriating it for alternate uses extremely difficult. The NUMACK character, for example, was hard-coded into the system using OpenGL Performer, making any alterations to its shape or abilities very cumbersome. These difficulties are also compounded by the fact that each successive research effort is likely to need a modernization of hardware and software to accommodate its new requirements. The difficulty of maintaining and reconfiguring the NUMACK system influenced the design of the next project undertaken at the lab. The Collaborative Storytelling with a Virtual Peer project sought to avoid the technical overhead of prior efforts by relying on short vignettes of agent behavior created with Macromedia Flash and controlled directly by the researcher [2]. While this approach greatly simplified the creation of agent content and avoided complicated dialog managers by using a Wizard-of-Oz (WOZ) approach, it presented serious limitations to the modification and re-use of agent behaviors [3]. Merely changing the color of the shirt worn by the agent would require making changes to every flash animation in the system.

Motivated by the difficulties of developing virtual agents, an international effort has recently begun to commoditize and componentize the development of virtual agent systems. This effort, called the SAIBA initiative, has resulted in a draft specification for a language to describe the behaviors that an agent must realize [4]. Our goal is to embrace this effort by attempting to utilize and enhance the languages and software products it has produced in our research. Our current research effort, targeted at children in the context of playing games and building structures, requires agents that can not only speak and gesture but can also interact with objects in the virtual world. Our software architecture for virtual agent prototyping and development has several other requirements. We require a system that allows content such as characters, objects and gestures to be easily modified and re-used in alternate situations. We also need any scripted behaviors of agents, including speech and gesture, to be re-usable and easily altered.

The SAIBA Initiative

The Situation, Agent, Intention, Behavior and Animation (SAIBA) initiative is an effort to break the requirements for agent development into its constituent parts. The framework has identified three main functional units for virtual agents: 1) planning of a communicative intent, (2) planning of a multimodal realization of this intent and (3) realization of the planned behaviors. These separate units effectively seek to separate high-level planning, modality specific behaviors and the rendering of those behaviors. The framework has proposed two standard languages to facilitate the communication between these units: Functional Markup Language (FML) and Behavior Markup Language (BML). The development of the Behavior Markup Language, between multimodal behavior specification and their realization, has been considered the more tractable task

and, as a result, is the first product of the initiative. A draft specification for BML is available and is actively being developed [5].

Behavior Markup Language divides behaviors into “acts” carried out by individual agents. Each act can consist of any number of specific behaviors such as speech, gesture or facial expression. The XML-based language provides for the creation of sync points that can be used to synchronize the behaviors within an act. For example, one might seek to synchronize a gesture with the speaking of a particular word in an utterance. In the sample below, a beat gesture has been designated to perform its most effortful part, the stroke, when the agent speaks the word “red”.

```
<bml>  
  <speech id="sp1">Meet me at the <sync id="tm1"/>red barn</speech>  
  <gesture id="g1" type="beat" stroke="sp1:tm1"/>  
</bml>
```

The synchronization points that have been identified, in the order of execution, are: start, ready, stroke-start, stroke, stroke-end, relax and end (figure 1). These seven synchronization points can be attributed to any behavior. The current specification for BML also includes the following behavior tags: face, head, gaze, posture and locomotion. The face tag is used to control agent facial expression while the head and gaze tags are used to control head nods and looking at objects and agents in the world respectively.

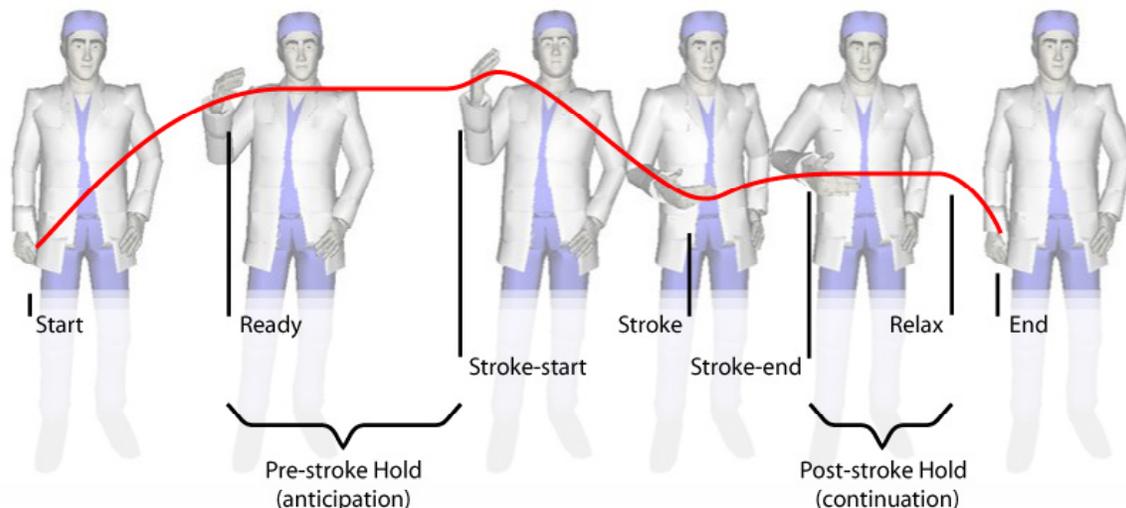


Figure 1: The seven gesture synchronization points defined within BML apply not only to gestures but also to other behaviors such as gaze.

Existing Realization Engines

Given a method of describing multimodal behavior such as BML, one must then build a system that can realize the behaviors it describes. This section describes two systems SmartBody/BMLR and ECAT/ACE that could potentially act as a behavior realization system for our research. The discussion below details how we decided to pursue using the SmartBody/BMLR system for our virtual agent development.

The NUMACK project utilized an underlying motion engine called the Articulated Communicator Engine (ACE) [6]. The ACE system is capable of aligning speech and gesture as proposed by the BML specification. This technology, developed by Stefan Kopp, is unique because it allows agent arm gestures to be defined in terms of their individual components such as hand shape, palm orientation and the like. Given the required position of end effectors such as the hand, ACE uses inverse kinematics to move the position and orientation of the remaining agent skeletal joints smoothly through spline curves. This procedural approach facilitates developing gestures that point to objects in the environment or describe objects of variable size (i.e. the length of a fish). The ACE system uses a language that was influential in the development of BML called MURML (Multimodal Utterance Representation Markup Language). The ACE system does not manage facial visemes nor does it have any facility for animating the type of key-framed animations favored by character modelers.

The output created by the ACE system is confined to the single character model it supports known as MAX. The rendering engine for the MAX agent is tightly integrated with the ACE engine and makes using different agent representations very difficult. An effort was recently undertaken to create ECAT, the Embodied Conversational Agent Toolkit, which effectively disconnects the rendering engine from ACE so that any rendering engine can be used with it [7]. The ECAT project does not attempt to migrate the MURML interpreter to the newer BML language but such a migration has been suggested for sometime in the future. ACE does provide a working interface between ACE and the SGI OpenInventor graphics library and does provide some preliminary work on an interface to the Panda3D graphics engine. An interface between any realizer and Panda3D is attractive to our work because we have been investigating using the open source rendering engine for some time.

The SmartBody behavior realizer is a system developed at UCS by Stacy Marsalla and his team working in collaboration with the Institute for Creative Technology (ICT) [8]. The SmartBody system can align speech and gesture, produce facial visemes and supports key-frame animated gestures. SmartBody currently only supports one type of procedural animation, a controller that allows characters to gaze at objects and other characters within the screen. SmartBody uses the proprietary Unreal game engine to realize its characters. A separate research group, the Center for Analysis and Design of Intelligent Agents

(CADIA), at the University of Reykjavik has developed a realizer for SmartBody that uses the Panda3D rendering engine [9]. The current code, known as BMLR, provides a straightforward pipeline for creating and animating characters using either the Maya or Blender modeling applications. While the SmartBody realizer using the Unreal engine implements facial expressions and speech aligned visemes, the BMLR realizer has not yet implemented this feature. Another feature, the movement of the agent eyes when tracking a target was also not considered by the BMLR developers.

A desire to provide legacy support for prior projects was also a concern in choosing a platform to develop. Our projects using Flash animations and the existing WOZ control panels we had developed were of primary concern. Although the ECAT system made some preliminary development efforts to support our existing WOZ control panels, no progress was made on the actual translation of their content into the MURML that would be required for ECAT to support them. Neither SmartBody nor ACE have any facility that would support synchronizing pre-recorded or synthesized speech with Flash animations. Even if such a facility could be developed, the effort required would likely obviate its utility. Table 1 lists the criteria considered in evaluating the two potential development platforms. We chose to pursue development of the SmartBody and BMLR combination primarily because SmartBody already supports a preliminary version of BML and it was unclear when if ever we could expect the ACE and ECAT combination to support that interface. Although it does not support procedural animations, we felt that SmartBody could be modified to support them because a gaze controller had already been implemented for the system.

Table 1: Relative merits of the two evaluated development systems

	SmartBody/BMLR	ACE/ECAT
Input	BML	MURML (maybe BML)
Output	Panda3D	OpenInventor (maybe Panda3D)
Key-frame Animation	yes	no
Procedural Animation	only gaze	yes
Visemes	partially supported	no
Flash Animation	no	no
Sam/WOZ input	no	no

Another factor in deciding to develop on the SmartBody/BMLR platform was the relative ease with which we felt it could be modified to meet our needs. In addition to adding viseme support, our research needs agents that can interact with objects in the virtual world. SmartBody already maintains the position of objects within the virtual world through the use of what are called pawns. We decided that this existing infrastructure would make grabbing and moving them more easily accomplished. We developed a plan to make three main contributions to the existing SmartBody/BMLR code base, 1) add viseme support to the Panda3D implementation, 2) develop BML tags for agent interactions with

objects and 3) develop a simple inverse kinematics solution that would allow procedural animations such as pointing at or reaching for objects. We accomplished the first two of our goals and have made some progress towards the final goal of procedural controllers. The following section describes in detail the changes we made to the SmartBody and BMLR code to accomplish our goals.

Adding Viseme Support

Visemes are the visual analog to phonemes. Adding viseme support to virtual agents requires synchronizing lip movement with speech generated by the agent. A simple technique for generating lip movement of virtual agents is to modulate the opening of the mouth with the level of the agent audio. For more accurate representations of human speech, the phonemes that make up an utterance are converted into a smaller set of mouth forms known as visemes [10]. The SmartBody system already has a facility for using the list of visemes associated with an utterance to synchronize the transmission of commands for their execution to the rendering engine. The developers of the BMLR code were not focused on providing this functionality and therefore did not build it into their system. Adding viseme support to the BMLR code requires building a character model that can be controlled to display a vocabulary of visemes. The code then needs to be modified to receive viseme commands from SmartBody and manipulate the appropriate parameters in the character model.

The BMLR code developed at CADIA consists of a number of Python classes that allow the creation of SmartBody characters in a Panda3D scene along with the sending of BML messages to SmartBody to define their behaviors. The BMLR code reads in a character and unpacks the skeletal joints associated with the character. After building our own character, our animator only needed to complete the part of the skeleton associated with the head in accordance with the SmartBody specifications to get the eyes to track targets.

The recommended procedure for creating facial animations in Panda3D is to use what are known as morph targets. Creating morph targets involves making numerous copies of a character's head and manipulating the skeletal joints of the face to achieve individual expressions. A separate set of sliders called blend shapes, one for each expression, are then created. These blend shapes can then be adjusted between negative one and one for each facial expression, or each mouth position in the case of generating lip visemes, to create a morphed expression of the character face that represents a weighted blend of each of the blend shapes.

To accommodate finding blend shapes in each character and adjusting them, the file *ClassCacher.py* was augmented to read a **.morphs* file for each character loaded. This file establishes a mapping between a viseme number determined by SmartBody and the name of a blend shape within the character model file.

Because facial expressions such as eye blinks and lip visemes may require different processing, blend shape names pre-pended with “viseme:” are stored in a separate array. The file *CharacterPawn.py* was then altered to search for the stored blend shapes in the already loaded character model and expose the morph joints. A routine, *setViseme*, was also added to set those morph joints. We use a python *LerpFunc* call to smoothly blend morph targets from their current value to their new value when the viseme duration is above a threshold. Because our primary focus was on lip visemes, we are currently managing both facial expressions and lip visemes in the same manner. The parser in *Scene.py* was then altered to call *setViseme* for each character when “SetActorViseme” messages are received from SmartBody. It is worth noting that the mechanism for producing character blinking currently only works for a single character in each scene. For testing, we directed all visemes to a single character, but in the future this problem will need to be addressed in SmartBody.

SmartBody has two mechanisms for generating visemes, using pre-recorded audio files with associated phoneme files or through a remote text-to-speech server. Our changes to the BMLR code allowed us to generate lip action for characters executing the pre-recorded audio and phoneme files provided with SmartBody. The phoneme files used by SmartBody, called **.lff* files, have been created using a proprietary piece of software called the Unreal Impersonator. Because it was our desire to use a different piece of software, *wave2lips*, developed by Jacques H. de Villiers at the Center for Spoken Language Understanding, we made several modifications to the *sbm_speech_audiofile.cpp* file. In order to accommodate any number of different phoneme file formats, we created a data structure to store phoneme names as characters strings along with their timing information. We then wrote routines to extract that information from either **.lff* files or our own **.pho* files. The code was altered to look for a “phoneme” attribute in the speech XML tag during parsing to determine the proper phoneme file extension to open along with an audio file.

The phoneme files are used to construct a list of corresponding visemes from a map file. The default map file was called *doctor.map*, but we moved to using a map file located within the same directory as each character named *phoneme2viseme.map*. Not all researchers use the same set of phonemes or visemes and the set used by SmartBody was hard-coded into the system. In response, we modified the code reading the map file to accommodate a flexible number of visemes associated with each phoneme and a flexible number of possible phonemes. This allows us to create viseme to phoneme mappings based on different phoneme vocabularies. In a similar vein, SmartBody also uses a fixed mapping between visemes and the viseme number sent on to the rendering engine. This hard-coded list of visemes includes a number of facial expressions developed for the Unreal engine renderer. However, we wanted the flexibility to create any morph targets in our characters and modified the code within *comappi.cpp* to read a *viseme2network.map* file to allow additional

mappings between visemes (i.e. eyebrow_raiser) and viseme numbers sent to Panda3D.

Each pre-recorded utterance is referenced by name using a “ref” attribute within the BML speech tag. In order to synchronize pre-recorded utterances with gestures, an *utterances.map* file was used to store the text of each utterance. This system appears to have been designed to synchronize gestures to words in the utterance by means of an annotation scheme. For example, synchronizing a gesture stroke to sp1:T3 indicates that the stroke should occur when the third word in the utterance is spoken. However, we decided to place the text of the utterance directly into the BML description. This allows us to place synchronization points at arbitrary points within the text. The functionality is controlled by the *getMarkTime* function within the *sbm_speech_audiofile.cpp* file. We altered this function to search for standard BML sync points and then estimate the time within the audio file at which the word is uttered by using the same word count method already included in the code.

The SmartBody system makes use of a client-server messaging protocol called ActiveMQ to handle requests to the remote TTS server and the list of visemes it returns. However, the BMLR designers were hesitant to include such overhead in their system and removed ActiveMQ from their compilation of the SmartBody code. We added a condition to the code in *sbm_speech_audiofile.cpp* that allows a speech utterance to be generated using a local TTS system if the “ref” attribute is not included among the speech tag attributes. This modification makes a system call to Cepstral's TTS synthesizer and then *wav2lips* generates a phoneme file from the resulting audio file. Unfortunately this method causes unacceptable delays in the SmartBody processing and as a result we plan to develop our own hashing method for text utterances. This hashing mechanism will allow us to easily prototype utterances by merely adding them to BML files. Once the utterance has been synthesized a first time, the system will find the hashed audio and phoneme files then play them without delay in subsequent executions.

Interactions With Objects

Our primary requirement is that agents can grasp objects, move them to another location and release them. To accommodate this, a new tag was added to the BML language called “grasp”. The grasp tag takes two attributes, “target” and “joint”. The target attribute indicates the name of a SmartBody pawn that should become attached to the agent. The joint attribute indicates at which joint the attachment should occur. The coordination of pawn attachment and dropping is accomplished through the standard gesture sync points. The point synchronized to the stroke of the grasp tag (i.e. <grasp stroke=sp1:red>) indicates the time at which the target pawn should become attached to the specified agent joint. Subsequently, the end sync point of the grasp indicates when the agent should drop the pawn. If an end point has not been specified, no drop will occur. This

allows an agent to grab an object during one behavior act and then drop it at a later time.

In order to accommodate this new behavior, several changes had to be made to both the BMLR code and the SmartBody code. The BMLR code was handling a coordinate system mismatch between the one used by SmartBody and the one used by Panda3D by rotating the Panda3D model by 180 degrees. Although this allows characters in the scene to correctly orient themselves towards pawns in the scene, it did not handle attaching pawns to agent joints correctly. To fix the problem, the model rotation in *ClassCacher.py* file was replaced by a 180 degree rotation of the skeleton base. The *ProcessCommand* function in *Scene.py* was then modified to recognize an “AttachActorPawn” message delivered through the commapi interface. Depending on the boolean value received, the parsing calls either a *GrabPawn* or *DropPawn* function that was added to the file *CharacterPawn.py*. The *GrabPawn* function makes the BMLR Pawn class object a child of the Panda3D NodePath associated with the designated skeletal joint of the agent. Currently, both the grab and drop functions reset the Pawn position and orientation to prevent any discontinuity. This functionality could easily be altered to move the object into a predetermined orientation with respect to the joint (i.e. for matching a tool handle to the hand orientation).

The grasp BML tag was added to SmartBody by adding it as a function handled by the *SbmCharacter* class. The process of adding a new behavior involves first modifying *bml_processor.cpp* to add the TAG_GRASP token, catch XML DOM elements of that name and pass their contents to a function called *parse_bml_grasp*. This grasp function in turn creates a member of a *GraspRequest* class that was added into the *bml.cpp* file. The constructor for this class initializes the default joint to “r_wrist”. The schedule method was overwritten to force the stroke->time variable into the end time. The request is then scheduled by adding a command to the SmartBody command stack of the form: char <name> grab <pawn> <joint> or the form: char <name> grab <pawn> <joint>. The *sbm_character.cpp* file was modified to catch these new SmartBody commands and call the *SbmCharacter:attach_pawn* method. This method, in turn, calls the *CommapiCharacter:AttachPawn* method added to *commapi.cpp* to construct the appropriate “AttachActorPawn” method and send it on to the Panda3D rendering engine.

In our testing with the version of SmartBody that the BMLR developers used, we had some difficulty synchronizing our new grasp behavior with the sync points of gestures. More specifically, we found that behaviors (i.e. a grasp or a head nod) could be synchronized to start at the stroke or end of a gesture but that their end time could not be synchronized. We did find that both a stroke and end time could be set for a grasp behavior by synching these points to other sync points with manually specified times (i.e. <sync name=“red” time=“3”>). The SmartBody developers indicated that this earlier version of SmartBody might not fully enforce synch points and, as such, we did not pursue modifying the code to fix this issue.

Interactions with the Virtual Environment

For the purposes of our research, simply moving objects in the virtual world may not be adequate. For example, we might want to initiate a command to correct the orientation of an object when it is dropped. We decided that we also needed to be able to place arbitrary XML commands within the BML language that are passed through SmartBody and into Panda3D. However, it remains important that these commands themselves can also be synchronized with behaviors (i.e. at the end of a grasp behavior). To accommodate this need we added an “emit” tag to the BML parsed by SmartBody. This behavior tag can be synched to any gesture sync point and passes all of its children nodes as XML on to the Panda3D renderer. The version 1.0 specification of the BML language already includes the emit tag but its implementation is focused primarily on emitting events that influence the execution of other BML behaviors. The emit tag had not been implemented by SmartBody and our changes do not affect any existing functionality. Although it may be acceptable to have any tags bellow the emit tag that are not recognized as BML passed on down to the rendering engine, some refinement of this formalism by be necessary in the future.

The changes made to the BMLR and the SmartBody code to add the emit behavior are very similar to those undertaken for the grasp behavior. The new tag was parsed within *bml_processor.cpp* and an instance of EmitRequest created and scheduled. The EmitRequest scheduling like the GraspRequest scheduling adds a command to the SmartBody command stack of the form: emit <xml>, where <xml> is a text string of the collapsed DOM children under the grasp BML tag. A *send_xml* method was added to the *sbm_character.cpp* file along with the necessary parsing to call it. Finally, the *commapi.cpp* file was modified to send an “ActorXML” message onto the Panda3D rendering engine. We also modified the BMLR code to capture and parse the XML for use in the Python environment. Python is very flexible about constructing commands from character strings and we will likely be using this method to directly pass the name of Python functions to call along with their appropriate arguments.

We took the route of adding the emit behavior to the SbmCharacter class because it was expedient. However, an argument can be made that it is not necessarily appropriate that all XML passed onto the renderer be attribute to an agent. For example, one might want to make adjustments to more global variables in the virtual world that. If disconnected from a specific character, it might be more appropriate to use the method used by the BMLR developers to add their text_speech element to SmartBody. This involved adding their own include *text_speech.h* include file into *mcontrol_util.h* and callbacks in the *sbm_main.cpp* file.

Procedural Controllers

Our current approach to interacting with objects in the virtual environment definitely limits the ease with which new scenarios can be easily prototyped and developed. A more general solution than key-framed gestures is the ability to have the agent reach for an object in an arbitrary location and orientation. In addition, we are also exploring scenarios where a user in the physical world makes adjustments to physical objects that are tracked and acknowledged by the virtual agent. Whether referring to virtual or physical objects, the ability of an agent to make referential pointing gestures towards objects is of real value to our research. We have begun developing a procedural controller for the SmartBody system based on the existing system that controls actor gaze through the manipulation of multiple joints of the spine and eyes. We have implemented a simplified geometry-base inverse kinematics system in the Panda3D environment but have yet to move the implementation into the SmartBody environment.

Some problems in human kinematics are more difficult than others. In the general case, a solution for an under-constrained set of equations must be found by iterative methods such as Newton-Raphson. The SmartBody gaze controller manipulates a range of spine and eye joints to direct agent gaze towards pawns in the environment. The implementation of this behavior is accomplished in a relatively straightforward manner by spreading the heading and pitch of the head amongst the spine joints using an a priori set of weights. Although this means that that SmartBody contains no built-in equation solver, this does not present a problem for doing some well constrained inverse kinematics with the arms. As long as the system behaves as a two joint chain, simple geometric equations and some additional constraints on the joints can lead to acceptable results (figure 2). In the case of the arm, as long as joints up the hierarchy from the shoulder joint are not used for reaching, a two joint chain can be used if the wrist position is chosen as the end effector. Adequate results can be achieved in grasping and pointing at objects with this constraint. For both the grasping and pointing gestures, the object and some heuristics can be used to constrain the position and orientation of the hand and thus fix the wrist position.

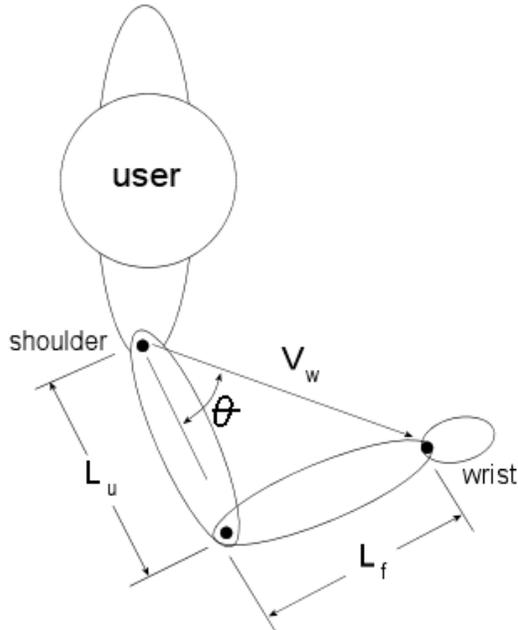


Figure 2: Given the wrist position, the angle of the shoulder away from the wrist and the angle of the elbow joint are a function of the lengths of the arm segments.

The calculation of arm joint rotations proceeds by first breaking the shoulder joint into three transformations, a rotation towards the wrist position M_w , a rotation related to wrist distance M_d and a rotation due to elbow position M_e . Given the vector from the shoulder to the wrist V_w , upper arm length L_u and forearm length L_f , the shoulder matrix, M_s , can be derived as:

$$M_s = M_w * M_d * M_e$$

where $M_w = M_x * M_z$ and

$$M_z = \begin{bmatrix} \hat{Z}_y & -\hat{Z}_x & 0 \\ -\hat{Z}_x & \hat{Z}_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

where $\hat{Z} = [V_{wx} \ V_{wy} \ 0]$ and

$$M_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \hat{X}_y & \hat{X}_z \\ 0 & -\hat{X}_z & \hat{X}_y \end{bmatrix}$$

where $\hat{X} = \begin{bmatrix} 0 \\ \hat{Z} \\ |V_{wz}| \end{bmatrix}$ and

M_d is the rotation matrix formed by θ

$$\text{where } \theta = \cos^{-1} \left(\frac{L_u^2 + |V_w|^2 - L_f^2}{2L_u - |V_w|} \right)$$

A heuristic for determining the rotation due to elbow position M_e can be formed by noting that the elbow is influenced by the physiology of the shoulder joint. When the wrist moves below the shoulder, the arm tends to rotate the elbow down and towards the body. As the wrist rises to shoulder height, regardless of its distance from the body, the arm rotates the elbow out from the body.

When grasping an object, the position of the wrist is dictated by the location of the palm relative to the object. One solution to determining the palm location involves creating a priority list of grasping locations for each object. For each grasp location, the distance from palm to object boundary (and subsequently required hand shape) can either be determined a priori and included in this list or calculated dynamically at runtime. The orientation of the object to the agent body can then be used to rank potential grasp locations. Choosing a candidate grasp location and calculating the resulting wrist location then becomes straightforward. While this approach in no way guarantees a completely natural grasp gesture, it should allow the object designer a sufficient degree of control over where and how objects are grasped.

Pointing gestures can be assumed to determine the shape and orientation of the hand. We have developed a simple heuristic for determining hand location during a pointing task. First, we constrain the position of the hand to a plane formed by the head, the referent object and the shoulder doing the pointing (figure 3). Then, we constrain the hand to lie on a line that is calculated using the maximum distance between the agent head and elbow. Although the elbow position is determined later by a separate heuristic, this roughly minimizes the angle between the wrist and the forearm. To determine the distance along the line to the object, we constrain the hand to lie just on the edge of a conic viewing frustum for the agent. Finally, a minimum distance from the object is enforced which overrides this viewing frustum constraint.

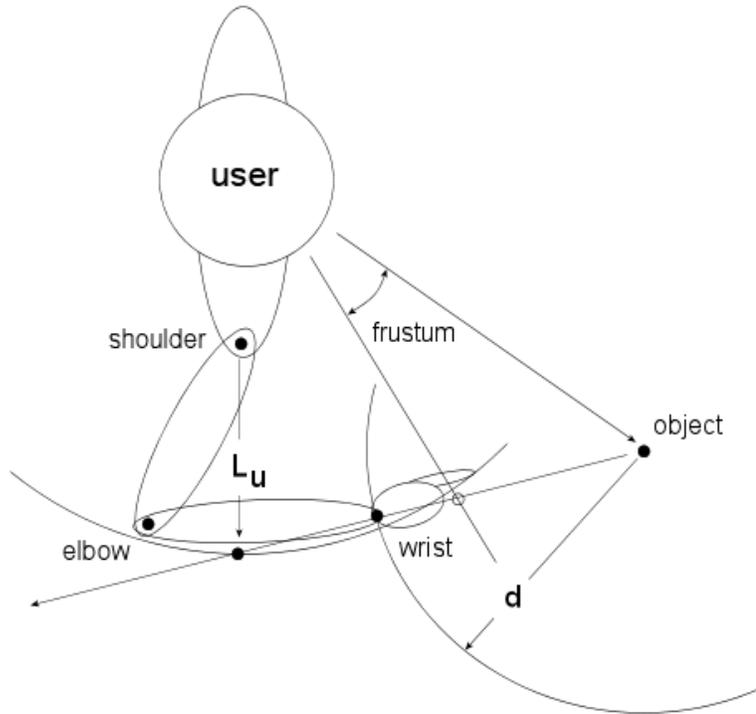


Figure 3: During pointing tasks, the wrist lies on a line between the object and the elbow at its maximum distance from the head. A minimum distance from the object and the edge of the viewing frustum constrain the final position of the wrist on that line.

Given our proposed heuristics for calculating the joint angles of the hand and arm during grasping and pointing gestures, our task becomes integrating these equations into the SmartBody system. We first implemented our equations for the shoulder and elbow position relative to wrist position directly within the Panda3D environment. We attached additional models to the shoulder joint of a virtual agent and used its wrist joint position during key-framed gestures as the end effector. This process allowed us to easily visualize if our calculations were correct or our elbow rotation heuristic was adequate. The arm kinematics can be easily accomplished using Panda3D data types and helper methods such as those that orient a quaternion towards a point in space. However, we formulated our calculations without these tools in order to match to the functionality currently provided by SmartBody.

The SmartBody system uses a series of hierarchical controllers to layer and blend the underlying key-framed gestures onto the agent joints. In order to use our procedural controller in this context, we can represent it in a manner similar to that already used by key-framed animations. This requires defining a function that takes a time value as input and gives joint values as output. By defining a duration to the stroke of our grasping or pointing gesture, we can interpolate the joint values between the current arm position and the final arm position. This requires calculating the final joint position of the arm and then simply interpolating those joints using through quaternions. The final joint values must

be recalculated at each call because we have no guarantee that the shoulder joint or the target object has not changed since the last function call.

Conclusion

In response to the increasing difficulty of developing complex virtual avatar systems, we have embraced the componentized model of agent development proposed by the SAIBA initiative. We evaluated two potential development environments that have come out of the initiative and chose to use the SmartBody BMLR solution from the CADIA at the University of Reykjavik. We identified three necessary improvements to the code that our research requires: viseme support for pre-generated and TTS generated speech, BML language support for interacting with objects within the agent environment, and procedural controllers for executing those behaviors in arbitrary configurations. We detailed the code changes we made to the SmartBody and BMLR Python code to achieve the first of two of these improvements, including the addition of “grasp” and “emit” tags to the BML variant understood by the SmartBody system. We then outlined our strategy for adding procedural behaviors to the SmartBody system by using a simplified kinematic model of the arm and some simple heuristics to determine elbow and wrist position during pointing and grasping. We feel that we have succeeded in contributing to an effort that will allow us to easily prototype and develop virtual agents in the future with a minimum of technical effort.

References

1. Kopp, S., Tepper, P. & Cassell, J. Towards integrated microplanning of language and iconic gesture for multimodal output, *Proceedings of the 6th international conference on Multimodal interfaces*, October 13-15, 2004, State College, PA, USA
2. Cassell, J. Towards a Model of Technology and Literacy Development: Story Listening Systems, *Journal of Applied Developmental Psychology*, 25 (1): 75-105, 2004
3. Mulsby, D., Greenberg, S. & Mander, R. Prototyping an intelligent agent through Wizard of Oz, *Proceedings of the SIGCHI conference on Human factors in computing systems*, p.277-284, April 24-29, 1993, Amsterdam, The Netherlands
4. Kopp, S., Krenn, B., Marsella, S., Marshall, A., Pelachaud, C., Pirker, H., Thórisson, K. & Vilhjálmsson, H. Towards a Common Framework for Multimodal Generation: The Behavior Markup Language. *Intelligent Virtual Agents 2006*, p.205-217
5. <http://wiki.mindmakers.org/projects:bml:main>
6. Kopp, S. & Wachsmuth, I. Model-based Animation of Coverbal Gesture, *Proceedings of Computer Animation 2002*, pp. 252-257, IEEE Press, Los Alamitos, CA, 2002
7. Van der Werf, R. The Embodied Conversational Agent Toolkit: A new modularization approach, *Master of Science Thesis in Human Media Interaction*, University of Twente, The Netherlands, April, 2008

8. Thiebaut, M., Marshall, A., Marsella, S. & Kallmann, M. SmartBody: Behavior Realization for Embodied Conversational Agents, *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2008
9. <http://cadia.ru.is/projects/bmlr>
10. Fisher, C. Confusions among visually perceived consonants. *Journal of Speech and Hearing Research*, 11(4):796–804, 1968