Character Animation for Real-Time Natural Gesture Generation

by

Kenny Chang

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 21,1999
[June 1999]

Author _____
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by _____
Justine Cassell
Professor, MIT Media Laboratory
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Character Animation for Real-Time Natural Gesture Generation

by

Kenny Chang

Submitted to the

Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

ABSTRACT

This work describes a character animation system designed for the real time generation of natural gestures. The system is part of a project to build a conversational humanoid, Rea, which supports both multi-modal input and output. An animation system built for this purpose must integrate all different approaches to animation control and have excellent external synchronization features. This thesis describes a way to conceptualize and organize different motor control systems to drive animation for a virtual character as well as a technique for synchronizing gesture and speech output. It also presents the design and implementation of Pantomime, an animation system based on these concepts.

Thesis Supervisor: Justine Cassell

Title: Professor, MIT Media Laboratory

1 Introduction

Character animation systems have been built for various different applications of virtual characters, primarily for directed action or scripted performance. Each system follows one animation approach that has techniques which best suit its target application. Little work has been done in building an animation engine specifically designed for realizing natural gestures, particularly, those generated in real time. Animation engines for these interactive characters have either use one approach to produce all necessary animations [30][13] or patched together various control routines[23][9]. There has been little focus on building a coherent, well-organized animation engine to support the set of controls needed by such an application.[1]

In fact, there has really been little need for a general animation engine for interactive characters. The range of tasks required of these characters have been relatively limited and it was sufficient to produce all the necessary animations by pushing the limits of a single animation approach or patching in control modules on demand. However, an animation engine for controlling a natural looking character to produce a wide variety of gestures inherently requires a multitude of control systems. For example, some necessary types of controls include inverse kinematics for arms, hand shapes, head turns, pointing, beat drivers, and gaze control, which includes saccade and track. Implementing all of these with one animation approach is unintuitive, and may even be impossible. Patching a small number of these together in a motor-module is feasible, but in the long run, supplying a character with a full complement of animation primitives to perform natural gestures leads to a cluttered and unmanageable system.

The focus of this research is to design and build Pantomime, a character animation system for natural gesture generation. After much reflection, it was unwillingly accepted that it was unreasonable to follow one or two approaches for animating a virtual character to produce a large set of natural gestures. Further,

---

[1] Blumberg et al. used a multi-level direction architecture for a motor system in ALIVE [17] similar to what will be described in this thesis. However, that motor system handles mostly direct level control and each motor skill is a pre-scripted sequence of "direct" actions like 'walk' and 'sit'. The motor system described in this thesis extends control to task level command and generalizes motor skills to include more complex ways to realize actions such as automated object tracking.

as mentioned previously, for a character of this type, there are just too many different control systems required to simply add one function after another to an animation module. Consequently, a structure must be conceived in which all types of control techniques can be applied to virtual characters in an orderly and consistent manner.

The Pantomime animation system was founded on the author's observations about how motor control systems seem to work in the human brain and how it may apply to driving an animated character to do natural gestures. In addition to the architectural framework motivated by these observations, other design drives of the system are the naturalness of output animation, flexibility, and speed as well as run-time motion realization, and accurate timing capabilities. Some of the factors to be compromised in favor of these objectives are simplicity, memory space, and physical constraint regulation. This system controls a skeletal model of a 3D virtual character with a set of degrees of freedom chosen to suit principally upper body gestures.

The following section will be a discussion of the motivations behind this research. After that will be a review of research and systems that are relevant to the development of Pantomime. The next section is an overview of Rea: the Conversational Humanoid, the research project that this thesis works to advance and an in depth look at the history relevant parts of Rea and the problems encountered there. Section 5 presents the theory upon which the system is built and section 6 is a description of the system's design. Implementation is thoroughly discussed in the ensuing section with a brief tour of how to use the system. Next is a report of an instantiation of the system built for Rea. The system is then evaluated in comparison to other existing systems and how their features may be used in the scheme described here. Last, but not least, is a discussion of crucial elements still missing in design and the direction of future work needed to extend it.

2 Motivation

Complex automated systems are an integral part of our lives today. The level of complexity and broadening user-base of these systems has made traditional interfaces cumbersome and increasingly inadequate. Complex systems undoubtedly necessitate complex interfaces. However, the training time

obligated by the learning curve can not satisfy the demand for use. Thus arises the need for better and more innovative interfaces.

A conversational humanoid resolves this steep learning barrier by introducing the human-to-human conversation as an interface, which everyone implicitly knows. Rather than reducing the difficulty of the interface, which may leave it crippled in some way, a conversational interface takes advantage of the user's existing knowledge to reduce the additional education required to effectively use a system.

At Professor Justine Cassell's Gesture and Narrative Group at the MIT Media Lab, we are working on a third-generation conversational humanoid REA (Real-Estate Agent)[7]. Rea accepts speech-to-text and 3D position of the user's head and hands as input. Based on this continuous data stream, Rea constructs a discourse model of the conversation. In conjunction with SPUD [26], a real-time text generation system, modified to generate high level description of gestures, our system attempts to engage in a natural dialogue with the user. As output, REA generates speech from the SPUD generated text and has a large screen display of an articulated virtual character, which performs the gesture output.

In building a virtual character for realizing gestures in real-time, the virtual character must be coupled with some sort of control system to interface with the internal generation system. This control system should be able to support a wide range of motor-skills for gesturing[2] and present a reasonable way to command them. These motor skills must also be realized on the actual character without any awkwardness, as if performed by a real actor or animated by an animator. The content of this thesis is on how to make that possible.

3 Related Work

Character animation has existed since the early seventies and has developed and matured along various different paths. The disciplines that applied modeling of human motions include crash simulation, the entertainment industry, and human-factors analysis [3]. Each of these fields have refined animation and

[2] This is primarily upper-body motion, arm, hands, and head, but also includes posture.

modeling techniques specialized to their application to some might consider an art form. Further, most of these techniques have been applied offline or in pseudo-real-time.

Using virtual characters for gesture performance, or in a wider sense, as interactive agents in a virtual environment has been a fairly recent development. In these years, computational and 3D rendering power and their prices have improved dramatically to make real-time 3D animation possible and thus spur research in this area. We can now take the offline techniques developed in the past as well as new methods made possible by technological advances and use them to animate virtual characters in real-time. Here we review a couple of the notable advances in character animation and human motion modeling technology.

3.1 Entertainment

Some of the more popular examples today take the form of computer graphics animated films such as Disney and Pixar's "A Bug's Life" and more photo-realistic examples like Mighty Joe Young. These have taken traditional 2D cel-animation techniques and reapplied them to the production of 3D animation [16]. In the movie industry, 3D animators are now so adept at sequencing character models that animated characters appear as real as human actors. These types of productions require large teams of animators to carefully script out actions and then render them into 2D images as part of a film.

More closely related to the topic of this thesis is the work done in the computer/video game industry. Game titles like "Tomb Raider", "Crash Bandicoot Warped", and to a lesser extent, "Quake", all employ the use of 3D animated characters. These systems generally play back scripted animations or motion-captured on the characters at runtime and usually implement some transition techniques for transitioning between scripted motions.

3.2 HCI and Human Factors Design

In human factors design, virtual characters have been used to simulate human presence in workspaces. These systems allow designers to visualize and test their designs on anthropomorphically correct subjects without actually having to build it. These systems require very detailed and accurate simulations of the

human body and its kinetic constraints. Some of the important aspects considered by these systems are reaching, grasping, and bending.

The Jack system developed by Professor Norman I. Badler at the University of Pennsylvania is a tool for human factor analysis. Jack was created in the late 1980s and featured extensive algorithms for reaching and grasping. Since it was an ergonomics system, the key motivation behind Jack was correctness in modeling a human body. Their character model, skeletal in nature, was fully anthropometric and considered joint limits and other human constraints. Jack's animation is supported by the Parallel-Transition Networks (PaT-Net) which is an animation control scripted by parameterized finite state machines. Each node of this network is a specification of a goal state or posture. The animation is a product of transitions between these nodes.

Jack was in fact the earliest precursor to Rea's animation system, perhaps more sophisticated than the current incarnations. Gesturing Jack was featured in Animated Conversation [9], which involved automatic generation of a conversation between two agents. The system produced a dialog containing natural gestures synchronized to speech with intonation. The entire interaction was produced by a dialog planner. Although the result was well timed and animated, the synthesis took several hours so it was not conducted in real time.

At the Information Sciences Institute at USC, Rickel and Johnson have built a virtual pedagogical agent called Steve (Soar Training Expert for Virtual Environments) [23]. Steve instructs a student immersed in the virtual world on how to inspect a high-pressure air compressor aboard a ship. The virtual character is a floating humanoid with no lower body. Steve's architecture is split three ways into perception, cognition, and motor control. The motor control receives high level motor commands from the cognition module, but motor actions can be triggered by any of the three main modules. Communication with the graphics system is performed through messages to update joint values. Currently, Steve supports three main classes of motor skills, locomotion, gaze, and hand control, each following a different control approach, but all put together as three parts of a single motor control module.

At the MIT Media Laboratory, the SCOOT! toolkit developed by Professor Bruce Blumberg's Synthetic Characters group is an animation system that applies spatial and temporal blending to scripted animations. This system allows a behavioral motor control such that a character can be directed to perform "happily" or "sadly". Their SWAMPED![13] system, which allows the control of autonomous animated characters with a plush toy as a tangible iconic interface, was built using this toolkit. In this system, characters are directed by the user and a competing behavior model that determine what and how motor skills are to be performed. Motor skills are scripted animations created using modeling packages such as 3D StudioMAX. The SCOOT! motor system uses a skeletal representation for their character, and employs a rather interesting network rendering scheme [see end of Section 6.1]. Although scripted animations tend to look better because they are pre-planned, they are less flexible than procedurally driven motions in the context of a conversational character because they can only be slightly modified during playback, and not quite in the spirit of runtime-generated gestures.

3.3 Virtual Reality

Virtual worlds have been a central playground for developing virtual characters. Indeed, what is the point of a virtual world unless there are some characters to inhabit it? There have been two approaches to virtual characters in the virtual reality (VR) domain; the avatar, and the virtual actor, although they are two sides of the same coin. Avatars are character representations of a user and actors are characters that follow an authored behavioral script. All these characters need animations systems to drive their motor movements, although their animations are typically previously modeled.

Hannes Vilmhjalmsson's BodyChat [30], also developed at Gesture and Narrative Language (GNL) group, is a graphical chat system where the participants are represented by animated avatars in a virtual world. This system, another precursor to Rea, attempts to automagically insert interactional cues into the avatar's animation throughout the conversation. The current release utilizes Rea's earlier animation framework [see Section 4.2.1] and has a set of procedurally scripted gestures such as "wave" and "glance".

At NYU's Media Research Laboratory, Professor Ken Perlin's Improv [22] animation scripting system provides a real-time procedural animation platform which allows the scripting animated characters in a

distributed virtual environment. Improv is an extensive virtual actors system, which includes a behavior model, an action scripting system, a network distribution scheme, and an animation system. The entire system allows a user to control the behavior of a virtual actor in a world with varying degrees of autonomy. The area of interest of Improv with regard to this thesis is, of course, the animation system. Improv's animation system features good temporal blending, an application of Perlin's own procedural noise synthesis to insert temporal noise into the animation. Further, there is support for action compositing, which allows multiple actions to be active in parallel, and rules for arbitration between actions that contend for degree-of-freedoms. Improv also implements sophisticated user-level interaction and network distribution models, which are beyond the scope of this thesis.

Professor Daniel Thalmann's work at the Computer Graphics Lab at the Swiss Federal Institute of Technology works on various levels of virtual actors and the world they inhabit. Some of their interesting and related work include deformation models[2] for body and face surfaces, physically based motion as well as behavioral approaches, and models for walking, grasping, and motion synchronization.

3.4 Others

There has been little literature oriented specifically towards gesture animation aside from facial gestures. Facial animation systems typically use the FACS (Facial Action Coding System) [20] notational system to specify facial animations. This parameterization is based on anatomical studies and specifies the movement produced by one or more face muscles as Action Units. Facial animation systems usually employ deformable meshes backed by either a muscle model [33] or superposition of facial keyframes. The Improv system employs the latter method. In Animated Conversation, the hand was specified by a superposition of hand shapes defined by the alphabet of the American Sign Language, which turned out to work surprisingly well for a set of hand eigen-shapes.

Gandalf [27], a previous system of the Gesture and Narrative Language Group, was a character that could interactively talk to a user about the solar system. It was the direct ancestor of Rea and had an animated face and one hand to animate deictic gestures. Gandalf used the animation system called ToonFace [28]

developed by Kris Thórisson. Rea's second animation system was based on certain concepts from this system [see Section 4.2.2].

The work in this thesis differentiates itself from the above mentioned research in that it centers on building a sufficient animation engine for natural gesture output. The primary problem that becomes evident when creating such a system is that it requires various animation and control techniques. For example, pointing gestures requires inverse kinematics for the arm, but forming the extended index finger requires a keyframe-like animation. Gaze control requires yet another approach to control and animation, and natural idle behaviors need to incorporate stochastics. With the number of different approaches that are applicable, having a more sensible integration of different animation control techniques becomes important, leading us to more carefully consider the architecture of a motor system and how different techniques can work together to produce natural animations for a gesturing character.

4 Background

4.1 REA Overview [7]

Rea ("Real Estate Agent") is a computer generated humanoid that has an articulated graphical body, can sense the user passively through cameras and an audio input, and is capable of speech with intonation, facial display, and gestural output. Rea's domain of expertise is real estate and her role is a real estate agent showing the user the various features of houses she is trying to sell.

Rea is designed to conduct a mixed initiative dialog, with the goal of describing a house that fits a user's requirements in addition to responding to the user's verbal and non-verbal input, which may lead the conversation in new directions. For example, when the user makes cues typically associated with turn taking behaviors such as gesturing, Rea allows herself to be interrupted, and then takes the turn again when she is able. Rea is also able to initiate conversational repair when she fails to understand what the user says. Rea's dialog is generated by an incremental natural language generation system, SPUD, backed by an Eliza-like [34] engine that responds ambiguously to the user's last utterance. In all instances of Rea's response and dialog, voice and/or gesture modalities are used.

While Rea is capable of understanding speech and can talk reasonably about real estate, primary effort to date has been in the interactional component of the conversation. Study in this aspect has been approached from the discourse theory perspective. Some of the discourse functions currently being managed are [7]:

User presence acknowledgment – Rea acknowledges the users presence by change in posture, turning to face and tracking him/her.

Feedback functions – Rea gives feedback in several modalities: she may nod her head or utter a paraverbal, e.g. "mmhmm" or a short statement such as "I see" in response to short pauses in the user's speech; she raises her eyebrows to indicate partial understanding of a phrase or sentence.

Turntaking functions – Rea follows the interactional state of a conversation, one of which is the holder of the speaking turn [18], speaking when she has the turn. Rea currently always allows verbal interruption, yielding the turn as soon as the user begins to speak. User gestures are also interpreted as an expression of the desire to speak, and terminates her utterance at the nearest sentence boundary. Finally, she turns her head to face the user at the end of her speaking turn to indicate she is done.

Greeting and Farewell functions – Rea executes ritualistic greeting and farewell sequence in multiple modalities, i.e. waving and saying goodbye.

Emphasis function – people may emphasize particular linguistic items by prosodic means (pitch accents) or by accompanying the word with a beat gesture (short formless wave of the hand). Recognizing emphasis is important for determining which part of the utterance is key to the discourse. For example, the user may say, "I'd like granite floor tiles," to which Rea can reply "granite is a good choice here;" or the user might say "I'd like granite floor tiles," where Rea can reply "tile would go well here."

The Rea architecture [8] [Figure 1] features a mirrored design where the input passes through a series of modules paralleled by the modules that the output is sent through. Internally, information is communicated through KQML frames with interaction and propositional content to maintain every aspect of the

conversation throughout the system. Each module represents a significant part of Rea's "thinking" process, and is written in C++ or CLIPS [11]. Table 1 lists the modules and a description of what each one does.

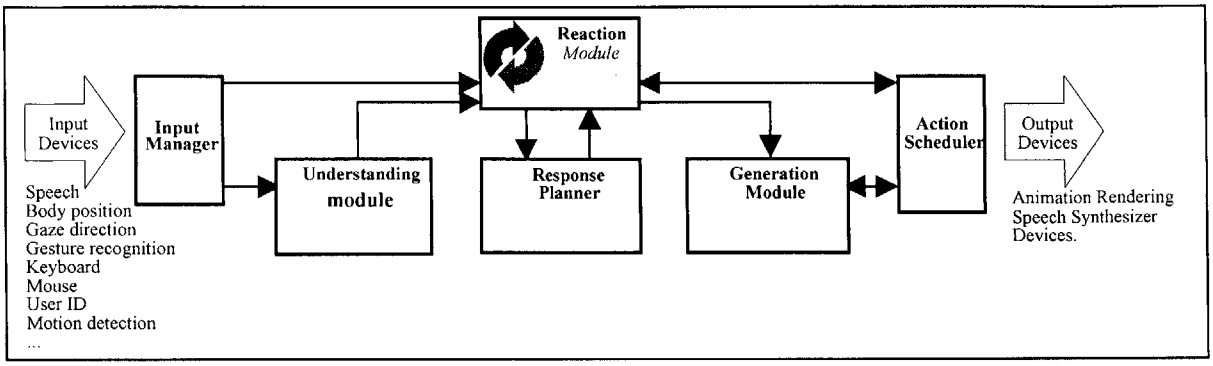

Figure 1 – Rea Software Architecture Diagram [8]

Module	Description
Input Manager	Performs multi-modal synchronization and integration. Currently supports three types of input: 3D user head and hand position [1], audio detection, grammar-based speech recognition using ViaVoice98. Features are sent to the Input Manager time stamped with start and end times in milliseconds. The various computers are synchronized to within a few milliseconds of each other. This synchronization is key for associating verbal and nonverbal behaviors.
Understanding Module	Fuses all input modalities into a coherent understanding of what the user is doing based on the current conversational state.
Reaction Module	Responsible for the "action selection" component of the architecture, which determines at each moment in time what the character should be doing.
Response Planner Module	Formulates sequences of actions, some or all of which will need to be executed during future execution cycles, to carry out desired communicative or task goals.
Generation Module	Realizes a complex action request from the Reasoning Module by producing one or more coordinated primitive actions (such as speech or gesture generation, or facial expression), sending them to the Action Scheduler, and monitoring their execution.
Action Scheduling	"Motor controller" for the character, responsible for coordinating action at the lowest level. It takes multiple action requests from multiple requestors (i.e. the Reasoning and Generation Modules) and attempts to carry them out. [using speech synthesis and animation engine]

Table 1 – Rea Software Architecture Description

The work described in this thesis sits just past the Action Scheduler module as part of the list of output devices, labeled as animation rendering. Pantomime is motivated by the requirements of the gesture output produced by Rea. The problems of interface, integration, and synchronization all arise out of the need to

realize the myriad of gestures that Rea has to execute. Pantomime is third in the line of animation engines used by Rea, the next section describes its predecessors and the problems they faced.

4.2 REA's Animation Engines

4.2.1 Version 1

The first animation engine for REA was created by Joey Chang [10]. It took advantage of OpenInventor's SoEngine nodes tightly coupled with a scheduler to drive joints in a VRML [31] body model. The SoEngine node is an object that outputs values at specific time indexes. Although OpenInventor is not a threaded application, the rendering loop is constructed such that SoEngine behaves like a separate thread that modifies the scene graph at particular moments in time.

REA's first body model was constructed unlike traditional character models in that each joint was constructed with one to three transformation nodes, one for each degree-of-freedom. The body model was encoded as a VRML file and during initialization, taken apart and put back together with these additional transform nodes. When an animation sequence was called, SoEngines were created to drive the necessary DOFs for the duration of the sequence. For animations with multiple segments of animation, the SoEngines will create a new SoEngine to drive the next segment before terminating itself at the end of its segment. Using this rather unique technique, animations were scripted in OpenInventor code and performed on demand. Because this was written in code, it was possible to vary the animations depending on input parameters.

The greatest problem with this first system was the unpredictable interaction between animation sequences. Since each animation segment initiated the next segment on its own, sequences could not be terminated before they were complete. If one initiated another animation sequence on the same joint before the previous one was complete, then there are suddenly two sources controlling one joint because the previous is still driving parts of the joint. The results of these unintentional interactions were very entertaining.

4.2.2 Version 2

In an attempt to solve the process interaction problem in the first version, the second animation engine for REA was written by the author with the help of Hannes Vilhjalmsson using a technique used in Gandalf [28], a precursor to Rea. The approach taken by this attempt was to dynamically pre-render the joint values at the time the animation was requested, and play them back at the appropriate time. This system is the one that is currently in use since the summer of 1998.

In this incarnation, Rea's character model is also defined in a VRML file but it complies to a minimal set of the VRML Humanoid (H-Anim) [32] joint standard where each joint is essentially a transformation node, containing rotation, translation, scale, and center fields. The H-Anim standard list of joints corresponding roughly to the human skeleton. Modifying the fields in the joint leads to rotating or moving the particular joint in the body. Using this body model loaded into TGS OpenInventor [19], Rea is procedurally controlled through functions called by the scheduler and currently supports a set of hand-coded gestures except for gaze, which is parameterized by user location, and inverse kinematics for the arms.

To facilitate the control and implementation of this system, the body was divided into several parts, organizing the DOFs into groups called Shapes. For example, the conglomerate degrees of freedom for the hand was called the HandShape, and for the arms, ArmShape and so forth. Each complete specification of values for a Shape is a keyframe in the animation sequence. An animation sequence for a particular sequence was specified by a Path, which was a directed graph where each node is a Shape and has one edge pointing to the next Shape. Each edge specifies a time and an interpolator to use to reach the next Shape.

Animation sequences are initiated with a mandatory approach time and interpolator, which specifies the time to reach the first Shape from the current configuration. This allows for seamless transitions between gestures so there are no sudden "jumps" in the character's performance. On sequence initialization, the system uses the Path specification and renders a list of future DOF values. The pre-rendered sequence is then played back as time progresses. If another sequence is requested, the current sequence is terminated and a new sequence is rendered starting from the current Shape configuration of the body.

While this technique solved the fickle process interaction problem of the previous system, this version introduced problems of its own. There are three issues that are most pressing and inherent in this system's design and approach.

The primary dilemma was that the Shape divisions permanently locked groups of degrees of freedom into dependency sets; meaning the grouping suggested that a particular number of degrees of freedom are dependent upon each other. For most intents and purposes, reasonable division choices perform quite well and the system is acceptable. For example, moving the arm almost always are only exclusively dependent on all the degrees of freedom chosen for the ArmShape and movement of the hands are dependent on that in the HandShape. One rarely finds that arm DOFs are dependent on hand DOFs or individual DOFs in the hand are independent of each other.

The restriction became more apparent when we began working on head-turns and head-nods. In the current Shape division, the 3 DOFs of the neck were coded as one Shape. The complication occurred when a head-turn was in progress and a head-nod was called, the head-turn was terminated in favor of the head-nod and would never be completed. In comparison to the first version of REA's animation system, in this framework, a lot of additional work must be done to correctly handle such a situation.

The second problem manifested itself as we began to increase our repertoire of gestures. Although certain gestures can be produced by using or extending existing code, it was often necessary to write new functions to support specific gestures or actions. The system began to become disorganized as it was littered with calls to do this particular gesture and that particular gesture. We began to walk down the path to an ad hoc animation system we've been trying so hard to get away from.

The source of both of these problems is that we did not recognize the variety of animation techniques necessary to satisfy the set of motor skills for driving gestures. Each Shape division essentially uses has a keyframe animation system to satisfy all types of motor skills. However, with each ability that require a different animation approach, a function must be added to cast that the results of that approach into a keyframe animation. As more abilities get added to a body part, the code for that body part tends to become large, unwieldy and increasingly painful to use.

Another major problem present in the current system is its timing capabilities. Since the system pre-rendered its frames, it is tied to a particular frame rate. It is possible to skip frames during playback but unless there is good control of the rendering loop in the graphics sub-system, timing is going to be an issue with this approach.

The particular timing problem in Rea's system is in synchronizing the speech generation with the gesture generation. Since the speech generation commands are sent across the network, and system does not give precise phoneme timings, it was very hard to start a gesture and hope for it to coincide with the appropriate phoneme. Additionally, the system was tied to OpenInventor's rendering loop, which does not guarantee callbacks at specific times, which means as the rendering slows down, the speed at which the gesture is performed slows down. Due to these two hurdles, Rea's speech-gesture timing is frequently off.

In light of the problems faced by the first two versions of the animation systems, and the fact that these problems are so intrinsic to the design, it was apparent that a new paradigm needed to be engineered to resolve these issues. Analysis of these mistakes, and a simple realization, has boiled down to some ground truths from which grew the design of Pantomime.

5 Theory

Earlier systems for driving virtual characters have focused on one primary control technique targeted for their application. For example, *Jack* uses a series of networks to specify conditions that must be met in order for an effector to reach a target configuration. Improv's animation engine is essentially uses keyframe animation but applies coherent noise functions to interpolate and blend between pre-scripted key poses. In the gesture generation context, both of these approaches are valid and extremely valuable, yet have certain limitations, i.e. *Jack* and Improv can't easily integrate each other's techniques. Since we are unwilling and unable to abandon any of these options, we must find a way to cast them in a framework that allows us to use all of them.

Pantomime was designed with the intent to systematically integrate various different animation techniques in a coherent framework to form a competent system for gesture generation. Existing systems either are

engrained in one animation approach or are just a patchwork of different approaches. Therefore, a new design is needed to provide the structure for every one of these approaches to work together logically.

There are three concepts that are the basis to this design:

1. There are various different animation control techniques for driving a virtual character, and every one is best suited to its task, and we need all of them.

2. These techniques have a common feature, which means there is a way they can work together. The common denominator is that they all control single-valued degree-of-freedoms (DOFs).

3. Synchronization is key, at least for gesture generation.

Each of these ideas is an integral part of the justification for why a control system for virtual character should be organized as described in this thesis. The following sub-sections will examine these concepts in detail, and also discuss how an application can uniformly interface with different control techniques.

5.1 All drivers are different but necessary

After taking an ad-hoc approach to building animation control for four months, I have come to the realization that there is no true general way to build control for humanoid characters. A hint to this insight comes from an observation about how certain motor skills work in the human body. It appears that the body develops specific control mechanisms as necessary to perform particular motor skills. Each of these mechanisms synchronizes a set of muscles to move joints, i.e. degree-of-freedoms in the body, to carry out a particular action. Since muscle movements correspond to joint rotations, we can say that these control mechanisms synchronize and control sets of DOFs.

Observations also show that various different control mechanisms can affect a single set of DOFs. For example, reaching employs a completely different kind of body control than hand-waving, but both utilize the same DOFs associated with the arm. This implies that, in a sense, maybe the mind has different ways to control the same parts of the body, and so it is logical for an animation system to have different ways to control the same set of DOFs of a virtual character.

A more scientifically backed example of this is the motor-control system for the human eye [6]. The human eye performs two primary types of motions, gaze-holding and gaze-shifting. Gaze-holding, or track, is the ability of the human eye to lock the eyes on a particular target in spite of the head or target's motion. This is called optokinetic response and requires feedback from either the balance organs from the inner-ear or the retinal image. On the other hand, gaze-shifting, or saccade, is a sort of a ballistic control, where the eyes are thrown to focus and hold on a target. This motion is executed without feedback mechanisms. These two controls are inherently different, yet we can not do without either system.

The point here is that there is no one best way of performing animation control for the human body. The control schemes for different motor skills are inherently different and range from inverse kinematics to planning models, each well suited for its task and unfit for others. To illustrate this with a problem we've encountered, producing a deictic gesture is a task that one can not perform without using inverse kinematics. However, using that technique instead of a handshape database to produce iconic gestures is difficult and overkill, and vice versa.

Consequently, it is not only desirable, but also necessary to embrace all these schemes to tractably create a control system for a virtual character to produce a wide range of natural gestures. The question is, then, how do we make a system such that all these control schemes can work together in a coherent manner?

5.2 Working together

Since the goal is to model the motor skills of a human being, it makes most sense to fall back to the human brain to solve the problem of integrating the various control schemes into one framework. At this point, we also make a terminology definition: since control schemes "drive" motor-functions, we will refer to the software equivalents of human motor control systems as drivers.

Following section 5.1, it appears that the body has specific mechanisms to perform a class of motor skills. For tasks that we do not have mechanisms for, we have to consciously control the body through existing mechanisms until one has been developed. One example is playing the piano. Before one has learned to play, one must think about controlling each finger to press on the appropriate key. Once the motor skills

have been obtained, one only need to think about playing the appropriate note and the reflexes will respond accordingly to cause the fingers to hit the right keys.

This supports a hierarchical view and extendable view of drivers, that there are low-level drivers and high-level drivers in the sense of a layered programming model. Higher level drivers can rely on or modify existing lower level drivers to do their work. It also says that a motor-control system is dynamic, that you should develop new control schemes and add them to your repertoire when necessary.

The key issue, however, is of course how a collection of drivers attempting to control a virtual character from different approaches should or could work together. The first step is to define how they should affect the character model. But before doing that, we should notice that almost all character animation technologies, at the lowest level, control individual degrees-of-freedom (DOF's) of joints in a human skeletal model [32]. For clarity's sake, a DOF is an Euler-like angle, which compose to a rotation of the joint in question. With this piece of information and a little cheating, we can take that first step simply by stating:

> All character models should be a hierarchical skeletal model consisting of a set of joints. These joints should be modified through the values of their individual DOF's.

We should hear little argument from researchers in the field since this is how most people do it anyway.

We now turn our attention to how drivers should interact with each other. The human body has a number of involuntary actions that are either repetitive or have some statistic rate of occurrence. The most obvious being breathing and eye-blinking. Humans also exhibit "natural" idle drift behavior, i.e. not being able to sit still. However, when we do need to perform an action, some of those actions are overridden and control is specifically given to those systems that drive the motor-function. One example is when you engage in a staring contest, you can force yourself to not blink.

Parallel to such a description, the author's view of character animation control is of a hierarchical nature. Arbitration for control of DOF's is done by ranked priorities; higher priority drivers can usurp control of DOFs from lower priority drivers.

Drivers running basic continuous processes such as eye-blinking and stance should have low priority but should always be trying to drive their respective DOF's. When there is no higher priority driver trying to perform an action, these drivers keep the body in motion, enhancing naturalness. When a higher priority driver needs to perform an action, the lower priority drivers should relinquish control of those DOF's. When the actions are done, the higher priority drivers should release control of the DOF's, which allows the lower priority drivers to pick up the control again.

Typically high priority drivers should be those that handle commands external to the animation system, i.e. the scheduler, and execute one time actions such as 'point' or 'wave'. Lower priority drivers are those that run in the "background", such as idle behaviors that get overridden when an action needs to be performed. The external controller should manage priorities for all drivers to produce a final animation that is the sum product of their interactions. [see Section 6.3]

5.3 Synchronization

Timing is a very important factor in the production of natural gestures. Because humans are very sensitive to timing, any delay or preemption of actions is immediately noticed as an irregularity. As a result, an animation system designed for gesture output must be very aware of synchronization issues.

When speaking of synchronization, there are two issues that are referred to. One is synchronization of the actions between DOF's, i.e. timing between gestures on the left and right hand, and the other is synchronization with events outside the animation system, specifically, speech. The discussion of the first is a way of conceptualizing drivers and the second is of how to time animations with external events.

5.3.1 Synchronizing gestures

Synchronization between body parts is better known as coordination in physiology. In the performance of gestures, coordination of the left and the right hand conveys significant meaning to the gesture. A humorous example of coordination is trying to pat your head while rubbing your stomach. At first, it is difficult because the most common reflex is to mirror the actions of both arms. However, one quickly learns to synchronize the rubbing action with the patting action. Although there is mostly likely deeper and

more correct meaning in this event, what I derive from this is that this ridiculous task has become easy to perform because one has acquired the driver for it.

So what does this say about how drivers work? Drivers should have a sense of the DOF's they control to realize particular motor skills as a set of dependent variables. For example, if a driver is trying to twiddle a virtual character's thumbs, it should be saying the DOF's of the left and right thumbs are dependent. This means when that driver is active, all those DOF's should be under the control of that driver.

5.3.2 Synchronizing to external events

Having well synchronized speech and gestures is a key concern for multi-modal conversational agents. Following something akin to the Grician maxims of Quality and Relation [13], an out of place gesture may cause implicature, leading the hearer to make conjectures about why the gesture is out of place. This means there a need for tighter control of speech and gesture timing because for certain speech and gesture pairs, for example, beats, the tolerance for temporal displacement is extremely small. This margin of error for synchronization is may be on the order of tens of milliseconds.

In the case of the Rea system, the synchronization issue may be an intrinsic problem in the speech generation system, however, since the speech system is off-the-shelf, and the animation system is custom built, synchronization must be fixed via the animation engine. This will probably be true for many other systems as well because standard text-to-speech systems are readily available. The text-to-speech engine used in the Rea system does not generate word timings so there is no way to predict how long it will take for a particular piece of text to be spoken or when a particular phoneme is reached. This means it is next to impossible to accurately schedule gestures for performance in synch with particular words. However, what is available is an event generated after the production of each phoneme, so there is some sense of the progress of the speech production.

With this capability, it is possible to guarantee a certain speech and gesture synchronization accuracy by the following technique. Take an animation system that is driven by clock ticks or cycling, i.e. repeatedly executing an update routine at roughly regular intervals. Suppose the current time of the system is passed to the update routine on every cycle and require all parts of the system use that time to perform its

functions. Then, that time value you pass into the update routine effectively becomes the system's notion of time.

Given a system with capabilities described above, one can follow a strategy for scheduling commands issued to the animation system such that the body of a character is at guaranteed to be at a specific configuration when a given syllable is spoken. The strategy is as follows:

1. Obtain syllable production estimates.
2. Schedule gesture g execution time t_E based on these estimates.
3. Update an animator time t_A based on events received from the speech system.
4. When $t_E = t_A$, execute command, remember module m which drives g.
5. Time progress module m with t_A.
6. Time progress other modules with current system time.
7. repeat from 3.

Step 1 is an empirical characterization of the speech production time of the text-to-speech (TTS) system. These estimates may be obtained by sending various speech strings to the TTS and recording the times and events received. These times should then be written to a lookup table so that given a string of text, the system, during step 2, can calculate time until a certain syllable is produced. Note that the execution time for a gesture is the start of the preparation, so for a given gesture, the scheduling system should preempt the synchronization point by the preparation time.

There are various ways to perform step 3. If the production time estimates are fairly accurate, then a naïve approach to calculating t_A may suffice:

$$t_A = \min\left(dt + \sum_i t_s, \sum_{i+1} t_s \right)$$

where t_s is the estimated syllable production time, i is syllable number (1st syllable is 0) and dt is the time since last syllable. Decomposition of text to syllables may be too much work, a simplification would be to synchronize on word boundaries instead of syllable boundaries. In this case, t_s would be word production times. For most intents and purposes, this probably won't degrade performance too much because the most prominent syllable tends to be the first. Also, the naïve formula will only work if the time differences are very small and the phoneme estimates are pretty good. However, if time estimates are grossly off, then the system will appear to "stick" or "snap".

A more sophisticated algorithm will choose to gradually speed up or slow down time to seamlessly abide by its tolerance. For example, one can estimate the number of phonemes in a word. Then, as the received phonemes approach the number of estimated phonemes, slow down the advancement of t_A. Once the word ending has been received, then speed up time advancement per received phoneme to catch up. This approach to synchronization will force the animation system to behave in time to the external event as long as all the drivers act on the times given to them.

5.4 Driver Manager

When generating commands for gesture production, we do not want to think about which motor drivers does what. The interface between a character animation system and its scheduler should be in terms of motor-functions. The commands we want to be sending should be in the form of motor-function and motor skills. For example, we want to say wave, point to there, look there, and look at that. The Driver Manager is the module that maps a motor skill to the control system that drives it. This is roughly what Blumberg [4] refers to as the Controller which sits between the actuators of the motor skills and the behavior system.

Further, since there will most likely be a large number of drivers servicing a large number of motor-functions, it also makes a coherent point of interface a necessity. Finally, in most cases, motor-functions are richly parameterized. To make programming and interfacing easier, the parameters should have defaults that are used when the issuer of the command does not specify values.

5.5 So, what is a complete set of drivers?

Whether an animation system built on these principles can really produce good natural gestures and perform as intended ultimately boils down to the set of drivers implemented for it. Then the question becomes what are the drivers we need for animating natural gesture. There is no real answer to that question as certain systems may want to create drivers to handle complex motor commands, for example, play patty cake. However, there is a set of core driver types that service a wide variety of motor-functions. For other, more complicated motor skills, new drivers should be written using the best animation technique currently available.

In general, the following list of motor control systems is a good starting set for a virtual humanoid:

Gaze Control

The gaze of a virtual character is very important for conveying the interactional content of a conversation. There is no need to state that gaze conveys information about the attentional state of a conversant. But more than that, gaze sends important interactional signals such as speaker continuation which disallows the taking away of a turn from the speaker[12]. As illustrated in section 5.1, there are two motor skills associated with gaze, tracking and saccade, and these two skills are essential to a gesturing humanoid.

In being able to effect gaze control, a system needs information about the current configuration of the character, and the positions of objects in the world relative to the character.

Inverse kinematics

The most important application of inverse kinematics in a virtual character is for driving arm motions such as deictic gestures and motions which causes the hand to trace a specific path. Inverse kinematic systems for arms calculate joint rotations based on a specification of a vector position of an end-effector, a reference point on the arm, usually the wrist or the center of the palm.

There are also various other control systems in the body that apply inverse kinematics, such as walking and reaching. However, since the focus of this thesis is on gesture production, we will not discuss more physics based methods.

keyframing

Keyframing systems perform animation by interpolating values between successive key configurations of DOF's, or frames. Many animation techniques are essentially sophisticated extensions of keyframe animation [21]. However, in its most basic form, keyframe animation is still a very powerful technique. Keyframes are good for gesturing particular hand shapes, where what is important is that the hand forms a particular shape. One example is the "thumbs-up" gesture. Gestures like that fit very well into the keyframe scheme. Keyframes are also great for scripting gestures, because it is easy to create a complex gesture sequence just by specifying a number of important configurations. Although it is slightly counter to having "generated" gestures, keyframe capability is just good to have around.

Motion-capture playback

Many systems use motion capture from real humans moving to produces very realistic character animations. What better way to animate humans than to use real humans? The flip side of this is that it is a recorded animation so it can only be changed to a small degree and reflects the personality of the human actor. Over time, the animation will also feel robotic since it's the same animation and must start at the same place all the time unless some modulation is performed on it. However, playback is still an essential driver because a character may need to replicate gesture it has seen or demonstrate something that has been performed by someone else. For gesture generation, this technique is best suited for producing emblematic gestures. These gestures are culturally specific, learned signs that can be used as if they are words, and they usually have to appear in the same form all the time [18].

6 Design

An effective conversational humanoid should appear as natural as possible. The role of the animation system here is to provide a set of primitive behaviors such that the realization of commands from the internal system present themselves in the most acceptable manner to the user. To make such a system possible, the design must adhere to the concepts described in Section 5. Additionally the design of the system should resolve to an implementation that is reasonable for the technology available at hand.

In addition to the concepts described in Section 5, there are performance concerns that are the goals of this design. These concerns are speed, ease of use, and flexibility.

The speed issue is most closely tied to the naturalness of a character's animation sequence. Because the human eye is very sensitive to timing issues, any lag in rendering or frames dropped will be easily noticed. Therefore, speed is a key consideration in this design and should be so in its implementation.

In the design of all API's, the ease of use is a contributing factor to its strength and performance. If a toolkit does not provide enough programming support, its potential can not be fully realized. Although its discussion seems to depart from the theme of gesture production, it is in fact rather important. As hinted in Section 5.4, the API to a humanoid motor control system has significant bearing upon how gestures are specified and realized.

The design of Pantomime is very simple [Figure 2], and practically a mirror of the ideas described in Section 5. A module called the Arbitrator receives data for driving joints from drivers in the form DOF values. The Arbitrator decides which driver should be allowed to drive which DOF's based on priorities and sends the joint information to a module which updates a skeletal model of the character. The Arbitrator acts for arbitration alone, and doesn't need to know anything else except to map DOFs to values. The drivers and the BodyModel encapsulate the entire motor control and physical capabilities of a character.

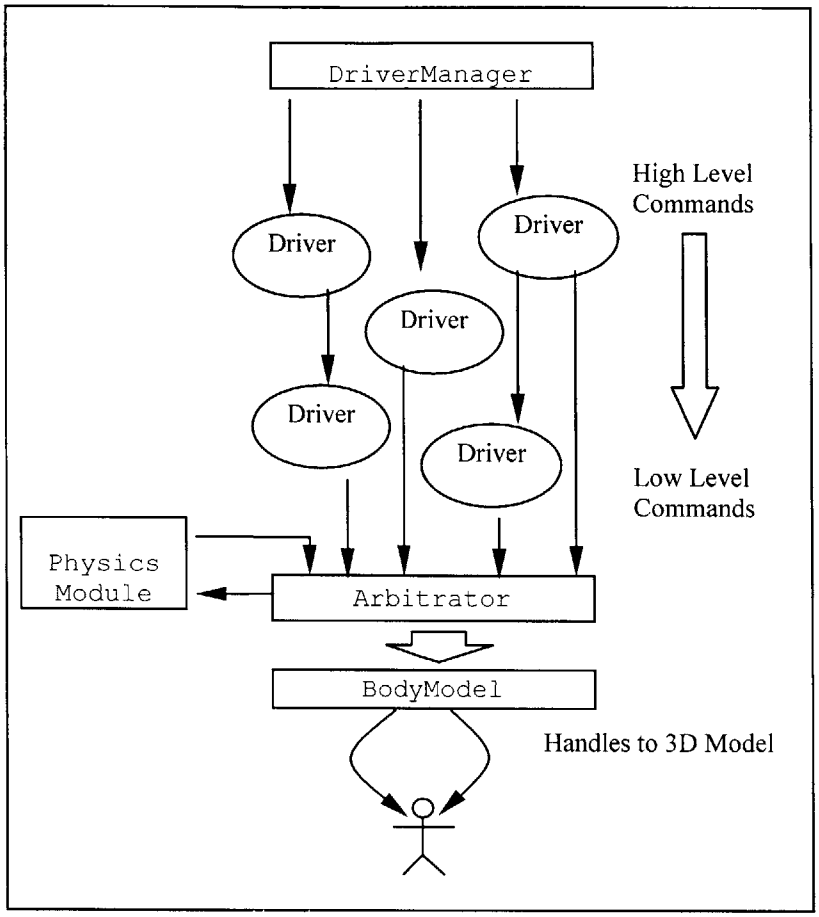

Figure 2 – Design Diagram

The Driver Manager serves to encapsulate a collection of drivers and presents their capabilities as a whole to the external controller. These capabilities are exported as motor-functions. Given a request for a particular motor-function, the Driver Manager sends the request to the appropriate driver supplemented with the appropriate default values. The Driver Manager also handles the passing of the system time to all the drivers.

6.1 BodyModel Abstraction

The BodyModel module has not been discussed previously in this thesis as it has no bearing on the control of motor-functions. This module signifies an interface between the Arbitrator and whatever skeletal model a system decides to use. This module releases the motor-control system from the underlying graphics system. There are various advantages to doing so and there are features typical of character animation systems which allows that.

Driving an animated character through a skeletal model requires limited access to the scene graph. Since we've chosen to abstract the control of the model into manipulating a bounded number of human joints, we only need a few handles to the scene graph. Further, the motor system primarily pushes data towards the graphics system and it is very infrequently the other way around. In the case of querying body configuration, we can cache the joint rotations that were sent to the graphics layer and return the cached values instead of touching the scene graph.

Portability is a major advantage of separating the motor control system from the graphics system. It gives a system flexibility to swap in models represented and stored in different formats. Separation also allows for different rendering configurations. For example, rendering can be done over a network, in a separate process, or linked in to the application as a library [24]. The actual animation may also be produced over the web, i.e. as part of a virtual community. Finally, modifying the scene graph in any way may hinder performance because of internal updates or the cost of communicating with the graphics system. The hope is to update the scene graph as infrequently as possible, at most 30 Hz, and if the motor control system runs differently than that, we don't want to hinder rendering performance. The graphics system can render and request updates from the motor control system at its own pace.

Speed, which is directly related naturalness, is the immediate concern here. By leaving hardware configuration options open to the application developer, he or she can use the best configuration possible for the application. For example, Blumberg's Swamped! system employs a novel network rendering scheme to optimize their performance. Their configuration runs a Java application on a PC which drives the behaviors of their characters. The Java application communicates over a 100Mbps Ethernet to an 8-processor Silicon Graphics Onyx2 running Performer that is solely dedicated to rendering the graphics [25].

6.2 Physics Module

The physics module is a subsystem that monitors the DOF values sent to the Arbitrator. It checks the data values for violation of any physical constraints, i.e. rotating any joint too fast or over rotating any joint. Conceivably, it may adjust posture to allow for certain body configurations or more natural performance of motions.

The current implementation of the Pantomime design is actually missing the physics module. Due to time constraints it was decided that implementing appropriate drivers was more important than having a physics module, which can be considered a refinement to the system. Drivers themselves can also serve as mini-physics modules, performing checks on themselves such that they do not exceed the physical limits of a joint.

6.3 Role of the animation engine and the scheduler

In designing an animation engine for natural gesture generation, one would always like to automate a number of idle behaviors. However, in doing so, the line between the animation engine and the scheduling/behavior engine becomes blurred. Which module should perform which idle processes, arbitration, and action blending? Therefore, it seems necessary to try to define the roles of the scheduler and the animation framework proposed in this thesis.

It is not possible to list which behaviors belongs to which module, however, since this animation system supports continuous processes, it is expected that the scheduler be the manager of these processes. As the manager, the scheduler should perform things such as turning the continuous processes on and off, tuning their parameters to suit the situation, and overseeing interactions between animation engine processes. Additionally, since the animation engine is considered to drive a generic figure, giving the ability to perform actions, it should control things that all humans do. On the other hand, the scheduler decides *how* and *when* actions are performed, and that depends on the person, so it should handle behaviors that are unique to an individual.

Finally, one will find that the system is lacking any action queuing functionality. Since the animation engine is designed to work with a scheduling module that plans when to execute particular motor-actions, it would be redundant work to implement features for accepting commands for execution at a future time. The expectation is that the scheduling system makes calls to the animation engine at the appropriate time actions should begin and the animation engine executes that command immediately. Note: for gestures, the starting time is defined to be the time the gesture begins its approach, not the stroke of the gesture.

To make this distinction more concrete, consider the following scenario. The scheduler starts the breathing driver on initialization and sets an initial breathing rate. The scheduler then receives a command to say: "The guards are coming!", with a beat gesture on "guards" and is told that the character is nervous. During speech production, a syllable before "guards" is spoken, the scheduler calls the animation engine to make a beat gesture. As a result of nervousness, the scheduler turns up the breathing rate so that the character appears to be breathing harder. The actual type of scheduling performed always depends on the kind of animation engine a scheduler has available, in the case of Pantomime, then, it will depend on the level of abstraction presented by the collection of drivers in the system.

7 Implementation

The system described by the above design has been implemented in C++ and compiled on the SGI IRIX platform. However, the system has been written to compile on all platforms. Using the concepts of component software and interfaces, it has been possible to create a core system independent of graphics API and character representation. The implementation is to be used like a developer toolkit to add control of a virtual character to an application.

There are four central classes in the Pantomime system: the Arbitrator, the BodyModel, the BaseDriver, and the DriverManager. Of these, the BodyModel and the BaseDriver are base classes that implement the communication with the Arbitrator. The application developer creates subclasses of these classes to implement system-specific and application specific code, respectively. Although the BodyModel and the BaseDriver are almost virtual base classes, i.e. all functions declared virtual, they do contain code to auto-magically perform certain tasks to guarantee the validity of the transactions.

7.1 Arbitrator

The Arbitrator acts as a factory for a class of objects named DOFChannel [Figure 3], which are channels through which drivers may communicate new DOF values to the Arbitrator. At any one time, there can only be one valid DOFChannel for each degree-of-freedom granted to a driver. A DOFChannel is granted to a driver if there are no drivers with higher priority holding a channel for the same degree-of-freedom. If a channel is usurped from a driver by a higher priority driver, the original driver is notified and releases and

destroys its channel. When a driver is done with modifying a DOF, then it is its responsibility to release the DOFChannel so the DOF is freed up for other drivers to use.

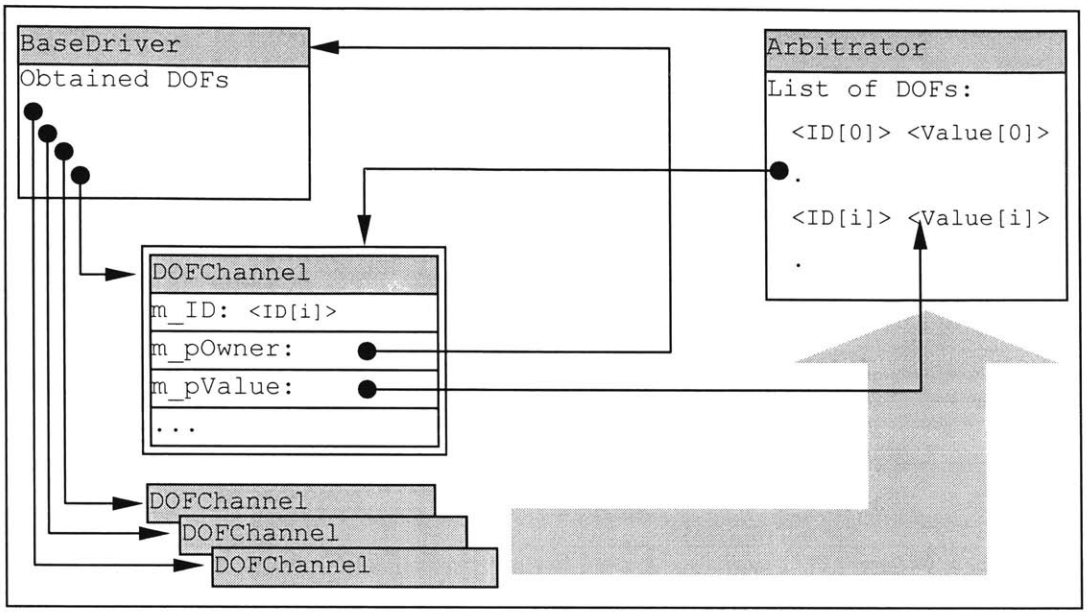

Figure 3 – DOFChannel

Some of the design concerns for the system affected choices made during the implementation. Speed was one of them. As a result, the system is rather inefficient in its use of memory. For example, the BaseDriver has an array the size of the number of possible DOF's to keep track of the DOFChannels it has been given. Lots of space would have been saved if we had used a list. However, since the number of DOF's were bounded, and the number of drivers relatively small, it was a reasonable price to pay for the speed improvement over searching in a list.

7.2 BodyModel

The BodyModel is the hardware-specific part of the Pantomime system. As mentioned, it is the abstraction of the character model away from the graphics system. In the case of Rea, it has been subclassed to create the VRMLBodyModel which interacts with the TGS OpenInventor toolkit and the H-Anim compliant data that represents Rea's body.

This module accepts input from the Arbitrator in the form of multi-valued quantities such as joint rotations and vectors, as opposed to single-valued DOF's. It communicates with the Arbitrator through the

IBodyModelDataSource interface implemented by the Arbitrator. Upon each cycle, the BodyModel calls the Arbitrator set by the member BodyModel::setBodyModelDataSource() to update the joint values using BodyModel::setJoint(). This organization allows the BodyModel to cycle independently of the rest of the system, synched to the updates of the graphics system. This reduces any lag caused by rendering or network communication.

More importantly, the system makes possible the creation of a BodyModel subclass that allows rendering to occur over a network, such that the actual animation could be produced over the web, as part of a virtual community, or on a more powerful graphics system. This can all be done without affecting any other part of the system.

7.3 DriverManager

The DriverManager is the point of interface for the application the system, besides the application-specific drivers themselves, of course. Drivers register with the DriverManager to be automatically updated every cycle as the DriverManager is updated. For those drivers supporting control through motor-functions, their exported motor-functions are also registered with the DriverManager so they can be accessed through this interface.

The DriverManager maintains a list of motor-functions, their parameter names and default values, and the driver export them. To avoid ambiguity in the unlikely case that two driver export the same motor-function, subsequent drivers to register an existing motor-function are ignored. When a motor-function is invoked, through the DriverManager::execute() member function, the DriverManager searches the list of motor-functions and extracts the associated driver and default values. It then creates an object containing the given arguments and default values for unspecified parameters and calls the associated driver to execute the motor-function with those arguments.

In Rea's implementation, motor-functions are invoked with KQML frames, i.e. the KQMLFrame class. The type of the frame is the motor-function, and the key-value pairs are the parameter names and values of the arguments. For systems not wishing to use the KQML frame representation, they may write specific

DriverManager::execute() member methods to parse their own representation, construct the argument object, and call the appropriate drivers' execute() method.

7.4 BaseDriver and MotorFunctionExporter

The BaseDriver is the base class of all objects who claim to be drivers. The BaseDriver class implements the request and release of DOFs from the Arbitrator. Specific drivers subclass the BaseDriver and overload the virtual members onTick(CTime &) and the onDOFLoss(DOF_id &) functions.(See Table 2 for a description of CTime and DOF_id) The onTick(CTime &) is the function that is called on every cycle. It is passed a time value indicating the current time index of the system. This is for accurate synchronization purposes. Drivers perform animation and state updates in this function. onDOFLoss(DOF_id &) is called whenever a driver loses control of a DOF to another driver, any clean up or state change when this occurs should be performed in here.

The BaseDriver also implements a Master/Slave driver capability. This is to support building more complex drivers using existing drivers. Each driver may create instances of other drivers and register them as slaves. A master can have multiple slaves, but each slave can have only one master. Slaves automatically inherit the priorities of their masters but these priorities may be reset. onDOFLoss() events received by slaves are also received by masters, but after the slave has executed its handler. However, to time advance its slaves, the master must specifically call the slave's onTick() method or its own BaseDriver::onTick() method, which will time advance all the slaves.

The MotorFunctionExporter class exists to support the motor-function abstraction provided by the DriverManager. This class is a subclass of the BaseDriver. It specifies two additional virtual functions which must be implemented by a driver if it wants to present its services as motor-functions. They are the listInterface() and execute() member functions. These two functions are used by the DriverManager to respectively query and carry out the motor-functions supported by the driver. To export its motor-functions, a MotorFunctionExporter must be added to the DriverManager using the addMFExporter() method.

A motor-function is identified by a string value, its name. Pantomime provides several tools to create an InterfaceList object which represents the list of motor-functions supported by a driver and their prototypes and defaults. These tools include a constructor to create one of these objects from an array of strings, each of which is a frame specifying a motor-function and its defaults.

Class/Type	Description
CTime	Class for abstracting the representation of time. Any interaction with time within the animation system should use the CTime class to allow for portability. This includes any newly written application-specific drivers. CTime is supposed to be of microsecond accuracy.
CInterpolator	Class with a number of static functions all matching the interface interpFunc, which is float interpFunc(float from, float to, float amt). These are interpolator functions provided for interpolating between two values.
DOF_id	The type for identifying DOF's. The possible values are of the form DOF_<joint-name>_<ROT|XL>_<X|Y|Z>. e.g. DOF_L_SHOULDER_ROT_Z.
CString	The string class for manipulating strings throughout the system. Should be compatible with most string implementations such that it may be swapped out to resolve name conflicts. Specifically, the Microsoft C++ CString class should implement the same functions as this class.
KQMLFrame	A general frame class used by the DriverManager and the InterfaceList to represent motor-function commands. Also used throughout the Rea system to represent conversational information. Accepts a string representation of the following form as a constructor argument: (<type> :<key> <value> :<key> <value>).

Table 2 – List of Supporting Classes/Types

7.5 Using the API

To integrate the Pantomime engine to an entirely unrelated system, there are two primary tasks to perform. The two points of integration are the BodyModel and the drivers. A BodyModel first needs to be created to match the graphics system used in the application. The current implementation of Pantomime has a VRMLBodyModel which works with H-Anim compliant VRML models running TGS OpenInventor with VRML extensions. Next, a complement of drivers to control the virtual character needs to be chosen to satisfy the requirements of the application. Although the current implementation already has a set of drivers written, some application-specific motor skills may require specific drivers to be created to service special commands. Some existing drivers may also need to be modified to correctly receive sensory information from the world.

When the previous has been done, a DriverManager should be created with instances of the drivers, the system-specific BodyModel, and an Arbitrator. Both the DriverManager and the BodyModel should then be clocked, but each may be run in different cycling loops. This allows the BodyModel updates to be synchronized to the rendering cycle and the drivers to be synchronized to the application. The character can then be controlled using either motor-functions through the DriverManager or the member functions of the drivers themselves. It is probably easier and more consistent to use the DriverManager, but this introduces a tiny overhead.

A graphics system-specific BodyModel can be created by subclassing the BodyModel class, connecting the joint handles in the Pantomime system to the actual joints in the graphics system. Currently, two examples of this exist, the DummyBodyModel, which outputs joint rotations to a text file, and the VRMLBodyModel, as described above. BodyModels rendering over networks may choose to cache a set of data with the BodyModel [25] so the application can have quick access to the current body configuration rather than having to communicate over the network to directly query the scene graph.

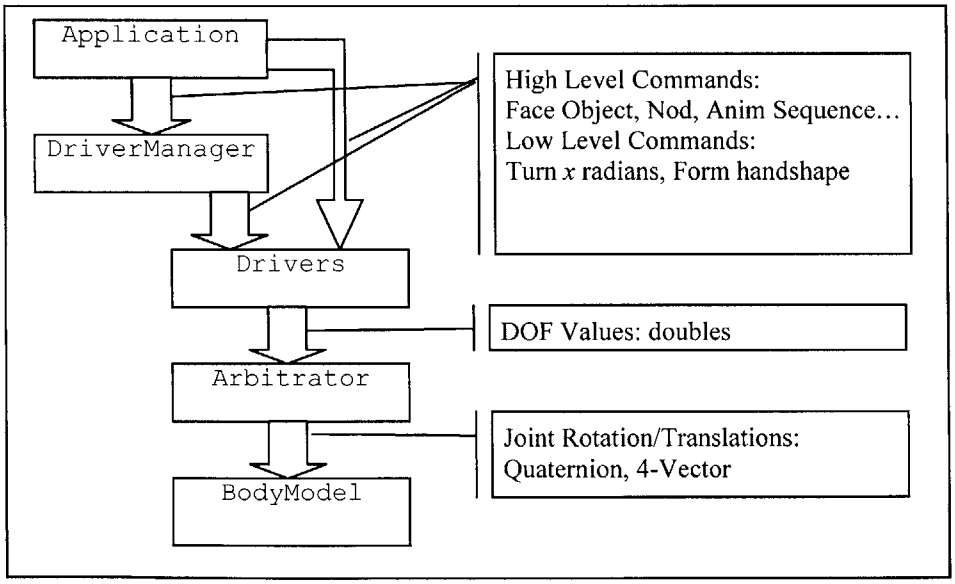

Figure 4 – Higher level commands boil down from the application through the drivers, Arbitrator, and finally to the BodyModel which accepts graphics system level information.

Then application-specific drivers must be written. Application-specific drivers refer to drivers that interact with the application environment. For example, the gaze driver requires information about where objects

are, so it must have application-specific code to find information about the world. There are various examples of this in the Rea implementation, and developers may choose to wrap parts of their application in Rea's representations to use the existing drivers.

To create a new driver, one subclasses the BaseDriver class and writes custom code to implement a particular control system. The onTick(CTime &) and onDOFLoss(DOF_id &) virtual members must be overloaded. To further extend a driver to support motor-function calls, one should instead subclass MotorFunctionExporter, which itself is a subclass of BaseDriver. Two addition virtual members must be overloaded, listInterface() and execute(). Descriptions of these four methods can be found in Section 7.4.

A driver may create instances of other drivers to help in their task. This functionality is provided in the BaseDriver and a helper driver is considered a slave of the creating driver. To add an instance of a driver as a slave, one uses the addSlave() member method. Section 7.4 has more details on the Master/Slave mechanism. An alternate method to build a driver on top of another is to subclass an existing driver. By doing so, a driver can inherit all the functionality of the parent driver and use it to implement higher order services. The essential virtual methods such as onTick() can be overridden to perform different time-dependent actions. However, this technique only works if the new driver only depends on one driver.

Table 2 lists some of the utility classes that may be helpful in writing drivers. Note that throughout the system, time should be uniformly represented by the CTime class.

8 Results and Conclusion

Using the Pantomime system, we were able to write drivers for animating Rea that offers a plethora of methods for animating gestures. These drivers were based on various different techniques, some were grounded on finite state methods, others used keyframes, and some used existing drivers to perform their tasks. Table 3 in the appendix enumerates the drivers implemented for Rea, their dependencies and approaches. Table 4 lists the motor-functions exported by the drivers.

Using the new motor system in Rea made creating gestures more intuitive and easier. Some actions such as 'gaze-away-and-return' became trivial because of the layered control feature of the system. One only

needed to issue a saccade command and specify a hold time, which overrides the tracking system. Afterwards, the tracking driver regains control of the eyes and automatically returns the eyes to focus on the user. There were also no problems with actions holding on to DOFs that they did not use. The only problem was that some times the programmer would set a motor skill on infinite duration and forget to terminate it which led a particular driver to hold on to a set of DOFs and not let go.

Although timing in general improved as a result of speed and use of absolute time values, the synchronization technique described in Section 5.3.2 was also attempted in the implementation of the Pantomime engine in the Rea system. However, the results were unsuccessful due to the fact that the end-of-phoneme and end-of-word event arrivals were untimely. This was because Rea sends text-to-speech commands and receives the progress events across an ethernet network, which introduces many variable delays. In general, measured event times were widely dispersed and sometimes arrived in clusters, so it was infeasible to go on to apply the rest of the technique.

However, applying a similar idea, we assumed a linear relationship between number of characters in a string and the speech event arrival time and determined the multiplier and offset experimentally. We then scheduled the gestures using this linear model. The results were satisfactory and better than before for a number of utterances.

It may still be possible to apply the technique mentioned in this thesis to the Rea by ignoring the phoneme events and just watching for the word completion events. However, because the uncertainty is so great in word completion time, a more sophisticated algorithm would be needed for calculation of animator time, such that the time is sped up or slowed down rather than halted when the estimate is off.

Ultimately, though, the best situation would be if the events were not sent across the network, i.e. speech output runs on the same machine as the gesture output. Phoneme events are very frequent and any delay, either in the network or the receiving process, will cause them to pile up at the event queue and thus affect their arrival time.

The system's strengths can be summarized in three features: flexibility, synchronization, and speed. Pantomime gives the flexibility for the application developer to use any type of animation control scheme

he/she wants for any motor skill. One is not constrained to using any single approach. This system gives sufficient control of time progression to support precise synchronization with external events. Lastly, the system's relative light weight and configurability makes possible for an application to take all the speed optimization advantages it has.

9 Evaluation

Evaluation of Pantomime will take the form of comparison to selected existing animation systems and how notable features for those systems fit into the framework of Pantomime. Current character animation systems can do a decent job as engines for gesture output, but alone, they can't do a great job. Their design goals tune their systems to specifically enhance their application, which limits their use of other animation techniques. For example, stochastic blending does not really fit into *Jack*'s PaT-Net framework unless it is somehow worked in at the lowest levels of implementation. The following comparisons will follow a pattern of the particular system being able to do some things very nicely but unable to reconcile with other approaches. They will also discuss how the key approaches used in the system can be integrated with Pantomime.

Although Pantomime also has a specific design goal, that of natural gesture animation, this goal happens to require the use of various different animation approaches. However, the one glaring factor that is not addressed by this goal is physical correctness, which gives the system a great speed advantage but loses out on guaranteeing that results are physically realizable, i.e. a hand may go through a wall or bend beyond acceptable joint constraints. However, to solve this problem, individual drivers may be built to satisfy the particular constraints related to the action or rely on the currently mythical Physics Module.

9.1 Parallel Transition Network (PaT-Net)

Norman Badler, Center for Human Modeling and Simulation, University of Pennsylvania.

One of Rea's aims has been to surpass what has been done with Animated Conversation and therefore Gesture Jack. Also, Jack is the only other system that has the most similar goals to Rea, and has been implemented to do similar tasks, the bulk of the evaluation will be on how Pantomime differs from Jack's animation system.

As previously stated, Jack's animation system is based on PaT-Nets. PaT-Net is a high level animation abstraction which doesn't really address how animation primitives are performed. For example, PaT-Nets can specify that a crawl is left-foot forward then right foot forward, but it doesn't say anything about how each foot is actually moved, whether inverse kinematics are used or just joint rotations.

For certain classes of animations, where one can encode actions as goal states and transitions between them, PaT-Nets are a very logical representation to use. However, there are certain actions that PaT-Nets can't do without additional support from a new animation primitive. In fact, in the end, an animation system using PaT-Nets will have to rely on animation primitives such as inverse kinematics or keyframe animation, etc.

PaT-Nets can be integrated into the Pantomime system by writing a driver that is a PaT-Net interpreter which calls on existing drivers as primitives. Having a PaT-Net driver would allow users to script or build new animations using networks. However, networks aren't always the best solution for representing things. That is why people create programming languages, so we don't have to work with computers like FSM's. Using a PaT-Net driver within Pantomime will allow the use of networks while leaving open the option of writing custom animation procedures using the flexibility of an object oriented language like C++. Pantomime allows for building animations using representations that best fits the task.

Finally, another advantage Pantomime has over the Jack system is speed. Because Jack follows a physics-based approach, the system is very tightly bound to the underlying graphics system. The 3D graphics must have support for collision detection, gravity, and other computationally intensive tasks. However, in this day and age, raw processor power is cheap and Jack can most likely run at very acceptable frame rates on a powerful Silicon Graphics machine. In light of this, the limitation is then portability and how well its ability to integrate with other computationally intensive systems.

Pantomime is relatively light in computation, and even space requirements, in comparison to Jack, depending on the type of drivers used. The system is well isolated to separate the graphics from the application and world interaction is handled through the drivers, which are partly application specific.

9.2 Improv

Ken Perlin, Media Research Lab, New York University.

Improv's animation system is a great system for producing natural looking motions. It is perfect for animating cyclic motion such as 'dance' and 'walk' without making it look like it's just the same clip played over and over again because input values to the interpolation are noisy. Drivers controlling repetitive actions, for example, hand-waving, can utilize this technique to make actions more naturalistic.

Another of Improv's features that is desirable is its temporal blending capabilities, which merges DOF values set from two actions as a function of time. For example, transitioning between actions by dialing down one action while dialing up the other. This technique can be realized in Pantomime by a blending driver that accepts the outputs of two or more drivers as inputs. This blending driver should then perform the blending and send the DOF values to the Arbitrator for update.

Improv also notes its Layered Behavior feature, which is really the ability to compose independent actions. In this regard, action composition is built in to the system because for drivers controlling different DOFs, composition of actions are immediate and require no particular semantics for arbitration. Resolution of actions controlling the same DOFs is controlled by driver priorities.

In comparison to other animation systems, Improv's major fault in light of gesture generation lies in its probabilistic nature. Non-deterministic interpolation makes it very hard to synchronize gestures with speech. There is allowance for synchronizing between gestures by running them off the same master clock but it is not immediately obvious how one can synchronize with external events. However, in conjunction with more deterministic methods, Perlin's stochastics can greatly enhance the believability of a character.

9.3 SCOOT!

Bruce Blumberg, Synthetic Characters Group, Media Laboratory, MIT.

SCOOT! allows for modification of an animation by behavioral parameters. This improves the expressiveness of the performance of the characters because it is affected by their current emotional state.

On the other hand, this system is more grounded in scripted animations than others because the animations have to be previously modeled and played back.

This type of animation scheme is restrictive for gesture production because animations can not be generated on the fly. Rather, it is possible to generate animations and add them to the action database dynamically but this can not easily be done. Also, should something occur to change the current animation sequence, it can not be easily modified. Doing so would be similar to the approach in Rea's second animation system [see Section 4.2.2], except in a more roundabout way. In essence, it is more an action selection and playback system than a control system. Although small action sequences can be concatenated to make action "phrases" to play out a wide variety of animations, such a system will not easily allow for producing motions such as eye-tracking or deictic gestures.

Additionally, since SCOOT! is so specifically targeted to expressive performance of scripted animations, it does not easily allow the integration of other animation techniques such as inverse kinematics or Perlin's stochastic blending. It would be difficult to provide more sophisticated control of the body for gesture production. On the other hand, this sort of playback can be well integrated with keyframe motion drivers or motion capture playback drivers to allow certain gestures to appear more expressive of a character's intent.

10 Future Work

There are many aspects of the Pantomime system that can stand improvement or hold additional features. The following will outline some of the more important directions that future development should take.

Pantomime easily allows the parallel control of unrelated DOFs by multiple drivers. However, a big gap in the design and implementation of this system is the specification of how two drivers can together affect the value of a single DOF. Such a feature can be made possible if drivers presented an Arbitrator-like interface such that they looked like an Arbitrator to other drivers and can act transparently as a destination for DOF data values. Given such a driver-to-driver interface, it is then possible to create a specialized driver that takes input from two or more drivers and perform operations on their output and send it on to the real

Arbitrator, as if only one driver was controlling the DOF. With this feature, a network of drivers can be built to create more sophisticated interactions between different control systems.

The system also lacks a distinct mechanism for time blending motor-functions. Currently, drivers are contracted to begin their work from the current body configuration. This doesn't look bad, really, however, to allow for more interesting transitions, perhaps procedures should be defined for the gradual hand-off of control of DOFs from one driver to another. This feature would greatly benefit co-articulation of gestures.

Perception is an integral part of a motor control system [5]. The current system does not address how drivers should sense things in the world and get feedback about the current body configuration. In the Rea implementation, each driver that requires information about the world maintains a reference to the character and through it, queries information about the world and requests transformations to various body coordinate systems. This method ties the motor system to the application through its dependencies on the implementation of the virtual environment. A way needs to be conceived for the drivers to gain access to a virtual character's senses in a uniform fashion.

Finally, implementation of new drivers is also a definite source of future work, however, it is expected that drivers be developed as needs arise, and should take advantage of existing techniques refined by years of development. What is more useful, perhaps more important, is a mechanism to script the driver hierarchy and the connections between them such that the system need not be recompiled to reorganize drivers. Better yet, it should be easy to make it possible to dynamically load drivers using shared libraries because all drivers implement a common interface.

The hope of the author is for this system to bring together all the various animation approaches developed by previous research to create a complete animation system for virtual characters. Then all virtual characters, directed or autonomous can control their bodies through a variety of animation approaches in a transparent way, making it possible to produce better more natural behaviors.

11 Acknowledgements

The author would like to thank Professor Justine Cassell for supervising this thesis, and especially for her time and patience in working with the author. Also, many thanks go to the GNL Humanoids: Hannes Vilhjalmsson, Tim Bickmore, Hao Yan, and David Mellis, especially to Hannes and David for their help in implementation. The rest of the Gesture and Narrative Language group also deserves much credit for the input and contribution to the ideas presented in this thesis. Finally, sincere gratitude to my family and closest friends, for their continued support and food in the late hours of the night.

12 References

1. Azarbayejani, A., Wren, C. and Pentland A. (1996) Real-time 3-D tracking of the human body. In *Proceedings of IMAGE'COM 96,* Bordeaux, France, May 1996.
2. Babski, C., Thalmann, D. (1999) A Seamless Shape For HANIM Compliant Bodies, Proceedings VRML99 (to appear).
3. Badler, N. I., Phillips, C. B., and Webber, B. L. (1993) Simulating Humans: Computer Graphics Animation and Control, New York, New York: Oxford University Press.
4. Blumberg, B. M., and Galyean, T. (1995) "Multi-Level Direction of Autonomous Creatures for Real-Time Virtual Environments". Computer Graphics (Proceedings SIGGRAPH '95), 30(3):47-54, 1995.
5. Brooks, R.A., Breazeal, C., Marjanovic, M., Scassellati, M., Williamson, M. (1999) "The Cog Project: Building a Humanoid Robot". To appear in a Springer-Verlag Lecture Notes in Computer Science Volume.
6. Carpenter, R. H. S. (1988). Movements of the Eyes. (2nd ed.). London: Pion. (http://www.cai.cam.ac.uk/caius/subjects/medicine/oculo.html)
7. Cassell, J., Bickmore, T., Billinghurst, M., Campbell, L., Chang, K., Vilhjálmsson, H., Yan, H., "Embodiment in Conversational Interfaces: Rea", to appear in CHI99, Pittsburgh, PA, 1999.
8. Cassell, J., Bickmore, T., Billinghurst, M., Campbell, L., Chang, K., Vilhjálmsson, H., and Yan, H. 1998. "An Architecture for Embodied Conversational Characters." Proceedings of the First Workshop on Embodied Conversational Characters, October 12-15, Tahoe City, California.
9. Cassell, J., Pelachaud, C., Badler, N., Steedman, M., Achorn, B., Becket, T., Douville, B., Prevost, S., Stone, M. (1994) "Animated Conversation: Rule-Based Generation of Facial Expression, Gesture and Spoken Intonation for Multiple Conversational Agents." Proceedings of SIGGRAPH '94. (ACM Special Interest Group on Graphics)
10. Chang, J. (1998) "Action Scheduling in Humanoid Conversational Agents". M.Eng thesis, Electrical Engineering and Computer Science, Massachusetts Institute of Technology. Cambridge, MA.
11. CLIPS Reference Manual Version 6.0. *Technical Report,* Number JSC-25012, Software Technology Branch, Lyndon B. Johnson Space Center, Houston, TX, 1994.
12. Duncan, S. Jr. (1974) On the structure of speaker-auditor interaction during speaking turns, *Language and Society, Vol. 3*, 161-180. New York, New York: Cambridge University Press.
13. Grice, P. (1989) Studies in the Way of Words. Cambridge, MA: Harvard University Press.
14. Johnson, M. P., Wilson, A., Blumberg, B., Kline, C., and Bobick, A. (1999) "Sympathetic Interfaces: Using a Plush Toy to Direct Synthetic Characters". In Proceedings of CHI 99.

15. Kendon, A. The negotiation of context in face-to-face interaction. In A. Duranti and C. Goodwin (eds.), *Rethinking context: language as interactive phenomenon.* Cambridge University Press. NY, 1990.

16. Lasseter, J. (1987) "Principles of Traditional Animation Applied to 3D Computer Animation", SIGGRAPH '87, Computer Graphics, Vol. 21, No. 4, pg. 35-43.

17. Maes, P., Darrell, T., Blumberg, B. M., Pentland, A. (1996) "The ALIVE System: Wireless, Full-Body Interaction with Autonomous Agents". ACM Multimedia Systems, Special Issue on Multimedia and Multisensory Virtual Worlds, ACM Press, Spring 1996.

18. McNeill, D. (1992) *Hand and Mind: What Gestures Reveal about Thought*, Ch. 2. Chicago: University of Chicago Press.

19. Open Inventor v. 2.5, Template Graphics Software, Inc. (http://www.tgs.com/)

20. P. Ekman and W. Friesen. *Facial Action Coding System.* Consulting Psychologists Press, Inc., 1978.

21. Perlin, K. (1995) "Real Time Responsive Animation with Personality", IEEE Transactions on Visualization and Computer Graphics; Vol. 1 No. 1.

22. Perlin, K. and Goldberg A. (1996) Improv: A System for Scripting Interactive Actors in Virtual Worlds. SIGGRAPH '96, Life-like, Believable Communication Agents: Course Notes.

23. Rickel, J., and Lewis Johnson, W. (1999) "Animated Agents for Procedural Training in Virtual Reality: Perception, Cognition, and Motor Control", to appear in Applied Artificial Intelligence, 1999.

24. Russell, K. and Blumberg B. (1999) Behavior-Friendly Graphics. To be presented at Computer Graphics International 1999. (http://characters.www.media.mit.edu/groups/characters/resources.html)

25. Russell, K., Johnson, M. P., and Blumberg, B. M.(1999) The Gonzo Graphics Architecture: How to do Fast 3D Graphics in Java., http://vismod.www.media.mit.edu/people/kbrussel/gonzo/.

26. Stone, M. and Doran, C. (1997) "Sentence Planning as Description Using Tree-Adjoining Grammar". Proceedings of ACL 1997, pages 198--205.

27. Thórisson, K.R. (1996) "Communicative Humanoids: A Computational Model of Psychosocial Dialogue Skills". Ph.D. Thesis, Media Arts and Sciences, MIT Media Laboratory.

28. Thórisson, K.R. (1996) "ToonFace: A System for Creating and Animating Interactive Cartoon Faces." MIT Media Laboratory, Learning and Common Sense Section Technical Report 96-01.

29. Tolani and Badler (1996) "Real-Time Inverse Kinematics of the Human Arm". Presence, Vol. 5, No. 4, Fall 1996, MIT, pp. 393-401.

30. Vilhjálmsson, H.H. (1998). "BodyChat: Autonomous Communicative Behaviors in Avatars." Proceedings of Autonomous Agents '98 (2nd Annual ACM International Conference on Autonomous Agents) Minneapolis, May 9-13, 1998 p.269.

31. Virtual Reality Modeling Language, International Standard ISO/IEC 14772-1:1997.

32. VRML Humanoid specification version 1.0, http://ece.uwaterloo.ca:80/~h-anim/spec.html

33. Waters, K. (1987) "A Muscle Model for Animating Three-Dimensional Facial Expression". SIGGRAPH '87, Computer Graphics , Number 4, July 1987.

34. Weizenbaum, J. (1966) "Eliza. A computer program for the study of natural language communication between man and machine". *Communications of the ACM*, 1966, 9, 26-45.

13 Appendix

Table 3 – Implemented Drivers

Driver	ArmKFMotionDriver	**Dependencies**	ArmShapeDriver
Purpose	keyframe animation		

Description	Plays back a named animation sequence for the arm. Each sequence is a list of keyframes and time values that specify how long to reach that keyframe and maintain it. Also allows loop playback of a sequence. Uses ArmShapeDriver to move the hand to particular configurations.
	`Data file format ::= <sequence>...<sequence>`
	`<sequence> ::= (armkeyframes`
	`:name <seq_name>`
	`:sequence [<keyframe 0>...<keyframe n>])`
	`<seq_name> ::= name of the animation sequence, a quoted string`
	`<keyframe i> ::= (keyframe :shape <armshape_name>`
	`:approach <approachTime>`
	`:duration <holdTime>)`
	`<approachTime> ::= time to reach keyframe from current configuration in milliseconds.`
	`<holdTime> ::= time to maintain keyframe in milliseconds.`
	`<armshape_name> ::= name of an armshape in the shapes database.`		

Driver	ArmShapeDriver	**Dependencies**	None
Purpose	Forms a named pre-scripted armshape.		

Description	Given a preloaded database of shapes, moves the arm to the named handshape in the specified time. Uses technique and code from Rea's animator 2.0.
	`<Shape File Spec> ::= <shape>...<shape>`
	`<shape> ::= <armshape>|<handshape>`
	`<armshape> ::= (ArmShape :name <shape_name>`
	`:value <Arm DOF values>)`
	`<Arm DOF values> ::=`
	`[<SHOULDER_ROTX> <SHOULDER_ROTZ> <SHOULDER_ROTY> <ELBOW_ROTX> <ELBOW_ROTY>]`		

Driver	BeatDriver	**Dependencies**	None
Purpose	Performs beats		
Description	On the start command, takes a snapshot of the arm's current configuration and puts the driver into a ready state. On every beat command, transitions to a beat state and beats the arm from the current arm position to a calculated "beat" position, which is an outward rotation and down motion of the wrist and arm. When the "beat" configuration is reached, or when a complete command is given, returns to the ready arm configuration and the ready state.		
Driver	DirectDriver	**Dependencies**	None
Purpose	Directly drives a DOF		
Description	Interpolates a particular DOF to a goal position in the specified time.		

Driver	EyeBlinkDriver	Dependencies	None
Purpose	Make eye-blinks		
Description	Uses goal-states to drive each eye-blink. Each state contains a goal configuration of the eye-lid. When each goal is reached, transitions to the next goal state and so forth		
Driver	EyeBlinkReflex	Dependencies	EyeBlinkDriver
Purpose	Drive reflexive eye-blinking		
Description	Chooses successive eye-blink arrival times based on an exponential distribution. Subclasses EyeBlinkDriver and executes eye-blinks using the blink() function of EyeBlinkDriver on each arrival time.		
Driver	EyebrowsDriver	Dependencies	none
Purpose	Controls movement of eyebrows		
Description	Raise eyebrows to a certain amount.		
Driver	FacingDriver	Dependencies	none
Purpose	Controls body orientation		
Description	Turns the body to face various directions.		
Driver	HandShapeDriver	Dependencies	None
Purpose	Forms a named pre-scripted hand shape.		
Description	Given a preloaded database of shapes, moves the hand to the named handshape in the specified time. Uses technique and code from Animator 2.0. <Shape File Spec> ::= <shape>...<shape> <shape> ::= <armshape>|<handshape> <handshape> ::= (HandShape :name <shape_name> :value <Hand DOF values>) <Hand DOF values> ::= [<THUMB_METACARPAL_ROTY> <THUMB_METACARPAL_ROTX> <THUMB_PROXIMAL_ROTZ> <THUMB_DISTAL_ROTZ> <INDEX_METACARPAL_ROTX> <INDEX_METACARPAL_ROTZ> <INDEX_PROXIMAL_ROTZ> <INDEX_DISTAL_ROTZ> <MIDDLE_METACARPAL_ROTX> <MIDDLE_METACARPAL_ROTZ> <MIDDLE_PROXIMAL_ROTZ> <MIDDLE_DISTAL_ROTZ> <RING_METACARPAL_ROTX> <RING_METACARPAL_ROTZ> <RING_PROXIMAL_ROTZ> <RING_DISTAL_ROTZ> <PINKY_METACARPAL_ROTX> <PINKY_METACARPAL_ROTZ> <PINKY_PROXIMAL_ROTZ> <PINKY_DISTAL_ROTZ> <WRIST_ROTX> <WRIST_ROTZ>]		
Driver	GazeDriver	Dependencies	None
Purpose	Controls eye-gaze saccade skills.		
Description	Calculates goal orientation given a point in world space and moves eyes to lock on that point. Alternatively, moves the eyes a number of radians away from the current configuration.		

Driver	HandKFMotionDriver	Dependencies	HandShapeDriver
Purpose	keyframe animation		

Description	Plays back a named animation sequence for the hand. Each sequence is a list of keyframes and time values that specify how long to reach that keyframe and maintain it. Also allows loop playback of a sequence. Uses HandShapeDriver to move the hand to particular configurations.
	`<sequence> ::= (handkeyframes`
	`:name <seq_name>`
	`:sequence [<keyframe 0>...<keyframe n>])`
	`<seq_name> ::= name of the animation sequence, a quoted string`
	`<keyframe i> ::= (keyframe :shape <handshape_name>`
	`:approach <approachTime>`
	`:duration <holdTime>)`
	`<approachTime> ::= time to reach keyframe from current configuration in milliseconds.`
	`<holdTime> ::= time to maintain keyframe in milliseconds.`
	`<handshape_name> ::= name of a handshape in the shapes database.`		

Driver	HeadDriver	Dependencies	None
Purpose	Controls the facing direction of the head.		
Description	Turns to head to face an object in the world or an offset number of radians away from the current facing direction.		
Driver	IKArmDriver	Dependencies	None
Purpose	Inverse kinematic parameterization for arms.		
Description	Calculates DOF values given a wrist position and elbow rotation to create a goal configuration. Interpolates from current arm configuration to the goal in the specified duration.		
Driver	LipDriver	Dependencies	None
Purpose	Control lip shape DOF.		
Description	Rea uses a set of lip shapes to form the phonemes she speaks. This driver simply sets the mouth of the virtual character to a given lip shape index.		
Driver	TrackingDriver	Dependencies	None
Purpose	object tracking for the head and eyes.		
Description	Given a named object in the world, first performs a saccade to focus on that object and then updates eyes to stay on that object. If eyes turn past a certain angular tolerance from center, then turns the head towards the object such that the eyes can stay within that angular tolerance and remain focused on the object.		

Table 4 – Supported Motor-functions

Unless specified otherwise, *approach* is the time to reach the destination in seconds and *duration* is the time to hold a pose in seconds.

Function:	<l|r>armIK	**Driver:** IKArmDriver
Parameters:	x=0.0, y=-0.75, z=0.0, phi=0.0, approach=1.0, duration=-1.0, coord="as", rho=0.0	
Description:	Inverse kinematics for the arm. *x, y, z,* specifies the position of the wrist in the coordinate space specified by *coord*. *phi* is the elbow rotation around the vector from the shoulder to the wrist, starting from the lowest point of the circle. *rho* is the twist of the forearm. *coord* currently supports "as" and "gs". "as" is arm space, absolute coordinates, where the origin is at the shoulder. "gs" is gesture space, relative coordinates, where the origin is in between the shoulders, 1.0 in *y* and *z* is arm length, and 1.0 in *x* is arm length + half shoulder width.	
Function:	<l|r>armkfmotion	**Driver:** ArmShapeDriver
Parameters:	name="", repeat=1, msg=""	
Description:	Starts a keyframe animation sequence *name* for the arm. Repeats the sequence *repeat* times. If *repeat* is less than zero, then the sequence is repeated indefinitely.	
Function:	<l|r>armshape	**Driver:** LArmShape
Parameters:	shape="", approach=-1.0, velocity=-1.0, duration=1000, msg=""	
Description:	Sets the arm to the handshape *shape*. Takes *approach* milliseconds to reach the shape or moves to the shape at *velocity* radians/sec, whichever one is not negative. Results are undefined if both are the same sign. Holds the shape for *duration* milliseconds.	
Function:	<l|r>beat	**Driver:** BeatDriver
Parameters:	msg="", approach=0.3, retract=0.3	
Description:	performs beats. (see BeatDriver in Table 3)	
Function:	blinkreflex	**Driver:** EyeBlinkReflex
Parameters:	msg=""	
Description:	Turns the blinking reflex "on" and "off"	
Function:	DIRECT-<DOF ID>	**Driver:** DirectDriver
Parameters:	position=0.0, duration=1.0	
Description:	Direct control of a single DOF. DOF ID is in the form of <L|R>_<Joint>_<ROT|XL>_<X|Y|Z>.	
Function:	eyeblink	**Driver:** EyeBlinkDriver
Parameters:	which="both"	
Description:	Forces an eye blink.	
Function:	eyebrows	**Driver:** EyebrowsDriver
Parameters:	approach=0.3, amount=0.5, hold=0.1	
Description:	Raises the eyebrow *amount* and holds it for *hold* seconds.	
Function:	eyetrack	**Driver:** TrackingDriver
Parameters:	target="", msg="", tol=0.1	
Description:	sets the eyes to track the target, if msg is "stop", then the tracking is stopped.	

Function:	eyetrack-tol	**Driver :**	TrackingDriver
Parameters:	val=0.1		
Description:	Sets the tolerance of the eye turn amount past which the head to begin to follow to compensate.		
Function:	facing	**Driver:**	FacingDriver
Parameters:	dir="", duration=1.0, amount=1.0		
Description:	Sets the facing direction of the character. *dir* can be "full", "half", and "away". *duration* is the number of seconds to reach the direction.		
Function:	<l|r>handkfmotion	**Driver:**	HandKFMotionDriver
Parameters:	name="", repeat=1, msg=""		
Description:	Starts a keyframe animation sequence *name* for the hand. Repeats the sequence *repeat* times. If *repeat* is less than zero, then the sequence is repeated indefinitely.		
Function:	<l|r>handrelaxer	**Driver:**	HandRelaxer
Parameters:	msg="on"		
Description:	If msg is "on", starts the HandRelaxer, if it is "off", stops it.		
Function:	<l|r>handshape	**Driver:**	HandShapeDriver
Parameters:	shape="", approach=-1.0, velocity=-1.0, duration=1000, msg=""		
Description:	Sets the hand to the handshape *shape*. Takes *approach* milliseconds to reach the shape or moves to the shape at *velocity* radians/sec, whichever one is not negative. Results are undefined if both are the same sign. Holds the shape for *duration* milliseconds.		
Function:	headaway	**Driver:**	HeadDriver
Parameters:	dYRot=0.2, dXRot=0.0, approach=0.5		
Description:	Turns the head *dYRot* and *dXRot* radians away from the current direction sideways and vertically.		
Function:	headnod	**Driver:**	HeadDriver
Parameters:	amt=0.3, duration=1.0		
Description:	Performs a head nod down to *amt* of full head tilt and back in *duration* seconds.		
Function:	headtowards	**Driver:**	HeadDriver
Parameters:	object="", x=0.0, y=0.0, z=1.0, approach=0.5		
Description:	If *object* is "", turns the head towards a point in world coordinates, otherwise, turns the head towards *object*.		
Function:	lipshape	**Driver:**	LipDriver
Parameters:	viseme="CLOSED", index=-1		
Description:	If index is greater than zero, then the lips are set to that shape, otherwise, *viseme* is used to select the shape. *viseme* can be: "CLOSED","D","F","TH","B","R","E","A","O","OO","SMILE".		

Function:	saccade **Driver:** GazeDriver
Parameters:	target="", coord="eyespace", x=0.0, y=0.0, z=0.5, hold=0.5, dYRot=0.0, dXRot=0.0
Description:	Makes the eyes saccade to a particular position and hold for *hold* seconds. If *target* is not "", focuses on *target*. If both *dXRot* and *dYRot* are 0.0, then focuses on a point specified by *x*, *y*, *z* in the *coord* coordinate space. Otherwise, turns the eyes *dXRot* and *dYRot* from the current position. *coord* can be "eyespace" or "worldspace". "eyespace" is in absolute coordinates where the origin is between the eyes and z points outwards in the direction of the nose and y is up.

Prototypes for Important Classes

Arbitrator

Sub-class of: `IBodyModelDataSource`

Inherits from IBodyModelDataSource:

`setBodyModel(),updateModel()`

Overrides:

`updateModel()`

Constructor

```
Arbitrator(CString name);
```

Public Members

```
DOFChannel * requestDOF(BaseDriver &requestor, const DOF_id id)
    throw (NotFoundException, DOFDeniedException);
boolean      releaseDOF(BaseDriver &requestor, const DOF_id id);
DOF_Val      getDOFValue(const DOF_id) const;
```

BaseDriver

Constructor

```
BaseDriver(const CString name, const unsigned short priority);
```

virtual members

```
virtual void onTick(void);
virtual void onDOFLoss(DOF_id id);
```

public members

```
void           setArbitrator(Arbitrator &a);
unsigned short getPriority() const;
void           setPriority(const unsigned short p);
```

Master-slave support

```
void        addSlave(BaseDriver *pSlave) throw (NullPointerException);
void        removeSlave(BaseDriver *pSlave);
BaseDriver *getSlave(unsigned int i);
BaseDriver *getMaster(void);
boolean     hasMaster(void) const;
```

BodyModel

base class for all classes representing a body model

Constructor

```
BodyModel(CString &name);
```

Virtual members

```
virtual void setJoint(const JointID jID, const SbRotation r);
```

Public members

```
void setBodyModelDataSource(IBodyModelDataSource * pSrc);
```

Static members

```
static boolean      isValidJoint(const JointID jID);
static const char * getJointName(const JointID jID);
```

DriverManager

Constructor

```
DriverManager(Arbitrator &Arb, BodyModel &BM);
```

Public Members

```
void addMFExporter(MotorFunctionExporter *pDrv);
void addDriver(BaseDriver *d);
void removeDriver(BaseDriver *d);
MotorFunctionExporter *getMFExporter(const CString &motorFunction);
ParamList *getPrototype(const CString &motorFunction);
void tick(const CTime &time);
void execute(KQMLFrame &f);
```

IBodyModelDataSource

Anything that attaches to a BodyModel must be a subclass of this class.

Constructor

```
IBodyModelDataSource(void)
```

Virtual Members

```
virtual void updateModel(void);
```

Public Members

```
void setBodyModel(BodyModel  * pModel);
```

MotorFunctionExporter

Sub-class of: `BaseDriver`

Inherits from BaseDriver:

```
onTick(), onDOFLoss(), setArbitrator(), getPriority(), setPriority(),
addSlave(), removeSlave(), getSlave(), getMaster(void), hasMaster(void)
```

Constructor

```
MotorFunctionExporter(const CString &name, const unsigned short priority)
```

Virtual members

```
virtual const InterfaceList * listInterface(void) const;
virtual int   execute(const CString &funcName, const Arguments &args);
```