Extending H-Anim and X3D for Advanced Animation Control

Yvonne Jung[*] Johannes Behr[†]

Fraunhofer Institut für Graphische Datenverarbeitung
Darmstadt, Germany

Figure 1: *The verbal and non-verbal communication of this discussion is controlled with our proposed animation controlling component.*

Abstract

In this paper we describe a layered approach to simplify character animation in X3D. Therefore, we present an interface and control language for specifying and synchronizing animations and similar actions at a higher level. Because this requires to have the accordant features on the lower X3D-based levels, we furthermore propose a set of nodes for realizing these demands. This includes for instance an audio node for text-to-speech that automatically calculates the actual phonemes and weighting factors for the corresponding visemes in order to achieve lip synchronization. To bridge the gap between these layers we also propose nodes for controlling animations, which are capable to convert the scripted schedules, and to mix an arbitrary number of interpolation based animations, whilst still being extensible to new concepts of on-line motion generation.

CR Categories: H.5.1 [Information Interfaces and Presentation (e.g., HCI)]: Multimedia Information Systems—Artificial, augmented, and virtual realities; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Animation

Keywords: Humanoid Animation, Virtual Characters, Animation Control and Synchronization, X3D

1 Introduction

In this paper we focus on defining and controlling humanoid animation in the context of X3D. The humanoid animation component (H-Anim) now is part of the X3D standard, and it not only provides a well defined structure of humanoid figures but also support for character animation based on predefined data. For simple scenarios H-Anim works well, but if multiple animations shall be combined and concatenated dynamically during run-time, the application soon gets unmanageable, because of increasing complexity and the missing support for mixing and synchronizing animations.

Not only in the field of X3D but also in general, animating and rendering virtual characters still has a lot of challenges. First, the methods should be easy to use and integrate into different applications. Second, to have a flexible control of the character requires a flexible animation system including body movements (gestures, walking) and speech (TTS, mimics). Then, visual realism means to have realistic models, natural gestures and a realistic simulation of materials. Finally, for interactive systems, everything has to be done in real-time, because in typical applications, like game scenarios and embodied conversational agents for entertainment and education, the interface must react immediately to user input.

Thus, for doing convincing character animation in complex and responsive environments, means for incorporating blending of different animations like waving and turning around at one single time step are needed, which cannot be accomplished with current X3D concepts. The same goes for cross-fading different succeeding animations in order to alleviate jerky leaps between e.g. an idle motion and a subsequent gesture. This gets even more complicated, if motions from a physics simulation or an IK system, which besides this both need some knowledge of the given scene, also shall be incorporated. But currently for instance neither the exchange schema Collada nor X3D support such advanced interaction and animation methods. Moreover, if a virtual character also shall speak, the situation is even worse. Because X3D does not provide support for text-to-speech, all spoken texts have to be prepared in advance, and – after having determined the lengths of all phonemes – put in synchronization with the corresponding visemes somehow.

Another aspect concerns the flexible and powerful control of the behavior of virtual characters, including the ability to implement and author story-lines. In movies, virtual characters are used that are almost life-like, but which have completely scripted actions. In games, characters can be autonomous, but the interaction with them is unnatural and limited to pre-programmed commands and behaviors. If an H-Anim figure within X3D shall be modeled with more advanced behavior like a set of skills and typical actions, this needs to be done externally by using Java, what mostly not only means slow value updates and multiple data storage, but often also to reinvent the wheel. By providing an additional layer on top of H-Anim for scripting and synchronizing animations and other motions based on our proposed animation control mechanism, a lot of complexity can be avoided, and the application developer can concentrate on the story and scene development itself.

[*]e-mail: yvonne.jung@igd.fraunhofer.de
[†]e-mail:johannes.behr@igd.fraunhofer.de

Therefore a higher level of abstraction is needed, which will be described by an additional control language. It provides an abstract layer to the graphics environment and its usage is also suitable for non-graphics experts. It therefore allows controlling character behavior and emotional states in any desired temporal order by researchers in other areas like artificial intelligence or story-telling. The XML-based control language PML is based on concepts taken from RRL [Piwek et al. 2002], and originally was designed as a representation and interface language between other engines like for instance a dialog and an affect engine [Klesen and Gebhard 2007] on the one hand, and X3D on the other hand. Thus, it must be able to specify the properties and the behaviors of characters and objects in a 3D virtual environment independently from their realization in a concrete setting. Although PML focuses on the specification of behaviors of virtual characters it also contains elements to specify and coordinate the presentation of other scene elements over time.

This paper is organized as follows: First, in section 2, related work is discussed. In chapter 3 we then show, how character animation currently is done with X3D, and how this often rather tedious process can be further improved. Thus, in chapter 4 and 5, some extensions to the current standard to meet these demands are presented. Finally, application examples are discussed in section 6, and conclusions and directions for future work are given in section 7.

2 Related Work

The great advancements in real-time virtual character simulation on the one hand and the need for higher level interfaces that allow a more abstract definition and control of object behavior on the other hand both call for better mechanisms of animation control and scheduling in real-time frameworks and middle ware solutions that aim at exposing suitable components for an easy and fast application development. Although for one thing there already exists a wide variety of commercially available real-time 3D frameworks and game engines, often including powerful level editors, they are neither cheap nor standardized. For another there are many free or open source toolkits and frameworks targeting at advanced character simulation, like VHD++ [Ponder et al. 2003], as well as at storytelling, like Facade [Mateas and Stern 2003], but – if standardized formats are used at all – then mostly only for data exchange but not for defining the run-time behavior. Besides this, application development still affords programming a C++ or similar API, and is therefore not accessible for non-programmers.

By introducing the humanoid animation component (H-Anim) in X3D [Web3DConsortium 2005; Web3DConsortium 2007], some of these problems were intended to overcome by specifying the structure and manipulation of articulated, animated, and humanlike characters. However, script nodes and prototyping mechanisms are the only possibilities to achieve some kind of behavior within X3D. This gets even worse for H-Anim figures, which usually are animated with different sets of interpolator nodes storing all keyframes and the corresponding, often dynamically created routes for updating the joint transformations every frame. This not only leads to lots of data, which has to be managed, but also requires the application developer to think about well designed prototypes for hiding and encapsulating the data and routing complexity in order to keep the application manageable. This concept basically has not changed over the last decade [Babski and Thalmann 2000; Weekley et al. 2007], is still labor intense and affords programming skills. Thus, in [Walczak et al. 2006] a method for authoring scenes in the context of networked educational systems, by means of a content production database, a VR modeling language, and so-called VR-Beans for modeling geometry and behavior, was proposed.

The general lack of a unified description for behavior and interactions in current 3D technology with special regards to X3D is discussed in [Dachselt and Rukzio 2003]. Here the authors propose their "Behavior3D" concept, which amongst others provides (based on ideas taken from SMIL 2.0) so-called "TimeContainer" nodes that can have a sequential and a parallel realization in order to overcome certain shortcomings in defining complex animations by supporting better means for synchronization. Likewise a similar time graph structure for X3D was proposed in [Göbel et al. 2004], but it turned out to be hardly extensible concerning more complex setups, albeit advanced temporal control already is integrated into standards like MPEG-4 [Preda and Preteux 2004].

In [Buttussi et al. 2006] the authors are especially trying to simplify the process of modeling skeleton based humanoid animation for creating sign language animations by proposing a visual modeling tool, where complex animations are created from sequences of simple animations by building a linear transition between them. This leads to acceptable results, if the motions are not too different and no penetrations occur. To alleviate this, research lately has focused on motion generation based on motion graphs [Kovar et al. 2002; Lee et al. 2002], which are directed graphs where all edges correspond to motion fragments taken from motion capture data in order to obtain realistic human motion including the subtle details of human movement, which are not present in procedurally generated motions via e.g. inverse kinematics (IK) [Tolani et al. 2000].

Motion synthesis is done via graph walks that require a certain connectivity, which is not given per se, because usually no two motion fragments are sufficiently similar. Therefore transition motions that seamlessly connect two motion fragments have to be created and a set of candidate transition points have to be detected. Additionally an animation controller is needed that assembles motion fragments based on the current state and input. Although these methods lead to convincing results, even for on-line motion generation in game scenarios, they still require preprocessing, manual work, are computational expensive and high memory consumption is still an issue [McCann and Pollard 2007]. Complementary to the motion graph approach are parameterizable motions, where the focus lies on generating parameterizations of example motions such as walking, jogging, and running [Park et al. 2002; Park et al. 2004], but which also still have to deal with the same problems mentioned previously.

For developing interactive virtual humans not only the geometric model and some basic ways of animating it have to be taken into account, but also aspects belonging to different levels of abstraction. These range from the shape, as well as key-framed or per joint based kinematics as defined by the H-Anim specification on the one hand, over physical aspects [Shapiro et al. 2003] like rag-doll simulation, hair simulation or tissue deformation, up to simple behavior like idle behavior, and finally to an AI driven cognitive layer for handling communication, personality, etc. on the other hand. Because developers have to write many lines of code related to all layers of this hierarchy, which easily gets rather unmanageable for more complex applications, in [Ieronutti and Chittaro 2005] the authors propose a generic, layered software architecture that allows to focus on the behavioral aspects, whilst providing animation models that also include collision detection and path planning.

Behavior definition usually is done with the help of authoring tools or by using scripting languages, as in the system proposed by [del Puy Carretero et al. 2005]. Here the authors use VHML [Marriott 2001], a language that was designed to specify the behavior of virtual characters in multimedia applications, and which consists of several sub-languages for describing the character, its gestures, emotions, etc. But such languages and behavior models are mostly highly domain specific and incorporate semantic models. A comparison of common markup languages for scripting and representing virtual characters can be found in [Arafa et al. 2003]. A more recent framework for behavior generation is proposed in [Kopp et al. 2006]. In this work a first specification of the communicative behavior markup language (BML) for mediating between a behavior planning and a behavior realization module is introduced. BML defines behavior elements like gestures and facial expressions and also allows to specify temporal constraints for ensuring temporal alignment. By defining an additional dictionary of behavior descriptions the representation language distinguishes between a more abstract behavior definition and its concrete realization.

In [Yang et al. 2005] a VRML based system consisting of three layers for animating characters is described. Whereas the lowest layer controls the joints, the middle layer combines a predefined schedule and different joint transformations to skills like ”walk” or ”open door”. It was shown that a Java3D based implementation not only was faster than a VRML/ Java version but also easier to implement. The highest level was an English-like scripting language for expressing the composition of skills and for hiding the complexity of lower layers. A similar approach is proposed in [Huang et al. 2003], although in this work the authors use their scripting language already for composing primitive motions based on operators like ’repeat’, ’choice’, ’seq’ and ’par’.

What is common to all approaches is the fact, that some advanced mechanism for animation control is needed, often with different levels of abstraction for reducing complexity and ensuring portability, in contrast to the simple event triggered mechanisms as provided by X3D. Besides this, if multiple animations shall be combined and displayed, a more advanced mechanism beyond ”Script” and ”TimeSensor” nodes for scripting and synchronizing all character actions is also needed, what usually is accomplished by means of a scripting language suitable for the corresponding domain, like scripting dance or verbal and non-verbal communication.

3 Analysis and Layer Design

The original H-Anim standard only defines the skeleton setup (with different levels of articulation), consisting of the rigid segments and joints that are needed to build up a humanoid. Standard VRML or later X3D animation techniques, mainly timers and simple linear interpolators, have been used to change e.g. the joint rotations over time. Later a simple but efficient skins and bones refinement of this ISO standard [Web3DConsortium 2005] also introduced seamless skinning. However, the definition and handling of dynamic changes have never been part of the H-Anim standard. The animation data itself is stored in X3D interpolators; one interpolator per joint, and the data flow is defined via X3D routes.

As already mentioned, for simple scenarios, like a single animation to be played, H-Anim/ X3D works well, but it is hard to use in cases where multiple animation sets are available, which are combined and concatenated dynamically during run-time. The overall structure of such an application gets unmanageable and confusing because of the vast amount of nodes, routes and missing information about membership to specific information. Tracing and debugging is almost impossible, especially when routes are created and deleted during run-time to blend animations together in scripts. However, for realistic scenarios this gets even more complex, because the various types of dynamics concerning humanoids is a very important part and occurs in different ways. The most obvious are the character’s movements, like gestures and locomotion. But also hair is not static and must be simulated to achieve high visual realism. Generally spoken, two types of approaches can be distinguished, the play back of predefined animation data on the one hand, and the on-line computation of animation data on the other hand.

In order to combine and concatenate animations efficiently, as e.g. needed for simulating the communication between different virtual

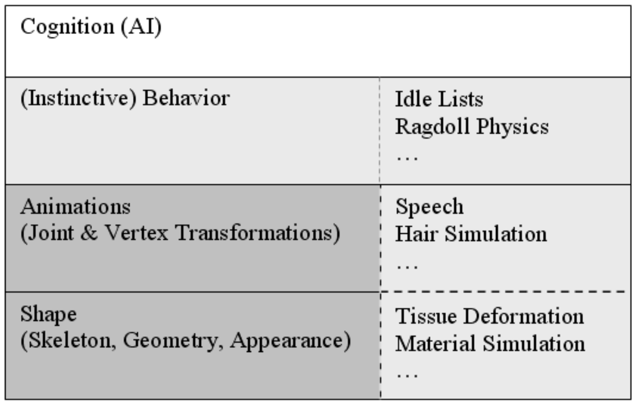

Figure 2: *Layers of complexity (X3D/ H-Anim covers gray parts, whereas parts depicted in light gray should be covered, too).*

characters and a user as shown in the images in Figure 3, additional information about the animations is also needed, like data look-ahead and a list of active animations and animations that will be activated within the next time-frame, which X3D does not provide. To alleviate these drawbacks we have designed animation storage nodes that provide a consistent view on an animation set. Furthermore, to provide a consistent and transparent interface for the application developer we have developed a centralized control component for animations and related actions that is explained in detail later. The proposed animation controller component additionally provides an interface for scripting and scheduling different types of actions and animations for encapsulating simple behavior, and thus for reducing the complexity of application development.

As can be seen in Figure 2, there is a hierarchy of different levels of abstraction concerning modeling virtual characters. The lower levels only deal with geometry, appearance, and the way, joints and vertices can be transformed, whilst the upper levels deal with instinctive behavior as well as cognition. Somewhere in between is physics, ranging from rigid body physics over hair and cloth simulation to the simulation of complex materials like skin. X3D provides some support for both lower layers, but there is no support for the upper ones. Whereas cognition belongs to the field of AI, autonomous or scripted behavior should at least partially be handled by the X3D browser. An example here is the more or less unconscious idle behavior – like blinking with the eyes, or moving slightly around. Idle behavior is displayed in between when no specific actions happen, because it looks quite unrealistic, if a character stands absolutely still. Thus, the X3D runtime needs knowledge of these animations as well, because if suddenly an intentional action shall be executed, the idle animations can not simply be stopped but have to be blended over into the new animations to avoid artifacts.

By using the animation controlling extension, which is explained in section 5.2, not only scripting but also mixing of animations can be easily done in X3D, provided that they are given as rigid body motions. But there are still some issues, that have to be kept in mind when simply mixing animations, mainly due to unsuitable animation data and missing transition animations. If e.g. the spatial distances between the first and the last animation frame are too big, this either leads to jerks or to sliding effects, depending on the blending parameters, which in the latter case introduce damping effects, if too many time steps are averaged. Thus, pre-recorded animation must be planned accurately. It should be defined, which is the starting and which is the ending pose, as well as which joints are involved. Because blending between very different poses often leads to unsatisfactory artifacts, it thus should be avoided.

Figure 3: *Animated characters and a photo from the CeBIT trade fair that shows an application based on our animation framework.*

4 The Control Layer

Similar to the additional programming languages needed for "Script" and "Shader" nodes, a domain specific language is introduced for animations. Besides, when animating and visualizing virtual characters one also has to think about interoperability aspects. Thus, especially in web environments, it should be possible to specify the properties and behaviors of characters and objects in a virtual environment independently from their realization in a concrete setting, whilst still being able to provide detailed information like the required animation parameters and exact timing information. Therefore the Player Markup Language (PML) was developed corporately; see [Klesen and Gebhard 2007; Jung and Knöpfle 2007] for an in-depth discussion and application examples. PML is an XML-based high level markup language for scripting the proposed animation control component, and thus comparable to a "Script" or "Shader" node, as it is a domain specific language to extend the current X3D concept. Additionally, it is designed to be independent of the implementation of a graphics engine and virtual environment, and hence can either be used as descriptive interface markup language between a graphics engine and some higher level behavior and dialog generation engines, or for directly scripting animations.

Because PML is a language for controlling virtual environments with special regards to character animation and user interaction, it defines a format for sending appropriate commands. Additionally, it defines a message format, which can be sent to the animation control component or received from it for enabling interactions with the scene via `<message>` and `<query>` scripts. At the beginning of a new scene all objects and characters are defined by a `<definitions>` script. There exist three types of definitions: repository definitions, character definitions, and object definitions. Repository definitions specify where the resources for the various scene elements are located. Character definitions specify the acoustic parameters of the synthetic voice, the available animations including their default durations, the phoneme to viseme mapping to be used, etc. (a short example that defines a list of idle animations is shown next). Likewise object definitions are used to specify cameras, user interface elements, and various media types that will be used in the scenario. Each such element has a unique 'id' by which it can be referenced via the 'refId' attribute in other elements.

```
<definitions id="iListDef">
  <character id="Valerie" src="Valerie.wrl">
    <multiPoses id="a" src="a.wrl" dur="2350"/>
    <multiPoses id="b" src="b.wrl" dur="2533"/>
    <idlePoses id="iP" random="true">
      <multiPoses refId="a" dur="2350"/>
      <multiPoses refId="b" dur="2533"/>
    </idlePoses>
  </character>
</definitions>
```

In the course of the story all runtime dependent actions like character animations are described by so-called `<actions>` scripts, whereas the temporal order is given by a special scheduling block including sequential and parallel elements. Actions are used to specify the appearance and behavior of all characters and objects in the environment. Some actions like 'show', 'hide', 'transform', or 'startIdleList' can be applied to both, characters and objects, while others are only available for specific scene elements. Below a short example script is shown, in which the previously defined idle list is started. Examples of actions that are only available for virtual characters are 'speak' for verbal output, and 'complexion' for the change in skin color (like blushing and pallor). A PML message is used to control the execution of actions and to exchange information between modules. There are three different types of messages: commands, states, and facts. Commands can be used to trigger the execution of actions; states are used to inform other modules about the execution state, e.g. started, failed, finished, what is important for later synchronization; and facts, which are represented by attribute-value pairs, can be used to inform about user actions. Finally, a query can be used for retrieving scene information.

```
<actions id="iListStart" start="true">
  <character refId="Valerie">
    <startIdleList id="iL" refId="iP" />
  </character>
  <schedule>
    <action refId="iL" begin="0" dur="0"/>
  </schedule>
</actions>
```

The animation tags of a PML actions script can either refer to preloaded animations, which are referenced by their name, or to simulated animations, e.g. via inverse kinematics. Different kinds of animations like morph targets and displacers for facial animation (`<singlePose>`), or key-frame animations (`<multiPoses>`) and simulated animations (`<implicitPose>`) for gestures and postures are distinguished, because every animation type must be handled differently and has a varying set of attributes. An example of a rather unusual animation which can be handled quite easily this way, too, is the change of the face complexion. Usually only the changes in geometry by means of displacers or morph targets are addressed in computer graphics. This is a well known problem, and the classification usually is based on the FACS [Ekman 1982], which identifies certain Action Units for morphing the face geometry. But with the help of modern graphics hardware the more subtle changes concerning face coloring can also be covered via animated skin textures or shader programs (see Figure 6, left).

By introducing a more abstract mechanism to define and synchronize different kinds of animations without having to take care about correct routing, timing etc., it is also much easier to create digital stories with embodied conversational agents in X3D. Such a story can be described with PML by putting together story-lines, i.e. short scene acts, in an easy and intuitive way through PML scripts that define when and what a character or object in the scene

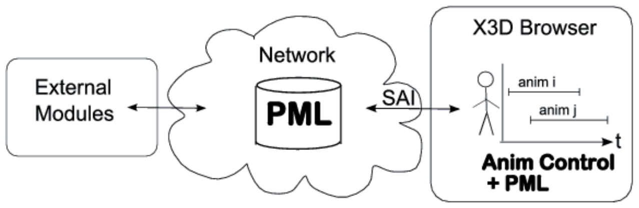

Figure 4: *A rare but essential additional use-case: Utilizing SAI to send PML chunks during runtime to the animation controller.*

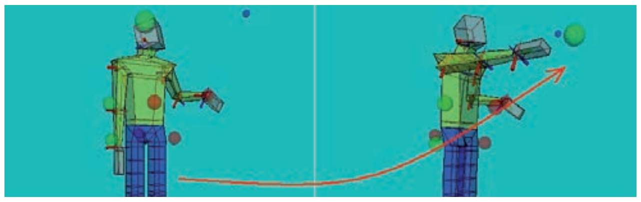

Figure 5: *The boxman is aiming at the green sphere via the HAnimIKSite node – the resulting motion is denoted by the red arrow.*

is doing. By combining this with other script or sensor nodes that define when and how the user can interact with the virtual environment there can also be added some non-linearity and possibilities for user interaction in order to create an interesting story graph. Figure 4 shows a possible system setup. As can be seen, PML can either be used for scripting and synchronizing within the X3D browser, or for handling the communication with modules that do not want to bother with problems concerning low level kinematics.

5 The Execution Layer

5.1 Essential Scene Graph Nodes

For the implementation we have used the InstantReality framework [Avalon 2008] that utilizes OpenSG [OpenSG 2008] for rendering.

5.1.1 Dynamic Gestures

Capturing and processing motion data is a tedious and time consuming task. To increase flexibility a better solution is to automatically generate animation data. Furthermore, there are animations whose appearance is not known upfront because they depend on external parameters. Examples are pointing gestures, where the direction is calculated during runtime (e.g. pointing towards a moving object), and character locomotion, where the target is defined during runtime (”go to door”). As can be seen, procedural animations like these need information about the scene: ”go to A”, ”look at B”, etc. implies having knowledge of A or B. By using external modules for the computation, in dynamic worlds this denotes high latency and unnecessary communication overhead, especially when taking animated terrain or other moving targets into account, and in static scenes it means at least keeping data twice, like for instance for path planning. Therefore built-in support for inverse kinematics would help to alleviate these drawbacks.

But currently the only built-in way of animating humanoids within X3D is via keyframing, although the H-Anim component already contains a node, the *HAnimSite*, that can be used to generate another type of animation, and which generally can serve three purposes: It defines an ’end effector’ location, which can be used by an inverse kinematics system (but without specifying the animation itself and how it could be triggered), an attachment point for accessories, and a location for a virtual camera. But only defining the position of an end effector in the given reference frame usually is not enough for fully parameterizing an IK system, and also doesn’t provide enough information about the targets of the motion to be calculated. For example a simple ’lookAt’ gesture requires knowledge of the target at which to look at, but that target (e.g. the eyes of another character) in general is not part of the humanoid. Therefore we propose the *HAnimIKSite* node for explicitly handling inverse kinematics within X3D. Its interface is shown below (fields already defined in the *HAnimSite* node are omitted for clarity).

The SFFloat field ’fraction’ can be set (e.g. by a *TimeSensor*) to values between 0 (start) and 1 (end) to animate the avatar. The ’target’ field defines the position of the target to aim at or to touch in world coordinates. Usually the translation of a *Transform* node is taken here (symbolized by the green sphere in Fig. 5). If the target should not be aimed with the center of the parent joint of the *HAnimIKSite* (maybe to define a point lying between the eyes or in the hand), then a translation/ rotation of the end effector can be defined via the ’aimingTranslation’/ ’aimingRotation’ field. The length of the kinematic chain is defined by the ’numJoints’ field. The value of the ’motionPathTension’ field has to be between -1 and 1, and defines the tension of the motion path in the sense of the Kochanek-Bartels spline tension parameter. The ’motionPathShape’ field determines the shape of the motion path, currently valid values can be ”auto”, ”quadratic”, and ”cubic”. The ’minTurnAngle’ defines the minimal angle a target has to be away in order that the torso turns, and ’maxTurnAngle’ defines the maximal angle the torso can turn.

```
HAnimIKSite : HAnimSite {
 SFFloat   [in, out] fraction 0
 SFVec3f   [in, out] target 0 0 0
 SFRotation [in, out] targetRotation  0 0 0 1
 SFVec3f     [in, out] aimingTranslation 0 0 0
 SFRotation [in, out] aimingRotation  0 0 0 1
 SFInt32   [in, out] numJoints 3
 SFFloat   [in, out] motionPathTension -0.4
 SFString [in, out] motionPathShape "auto"
 SFFloat   [in, out] minTurnAngle 0.2
 SFFloat   [in, out] maxTurnAngle 1.5
 SFFloat   [in, out] turnFactor 0.6
}
```

5.1.2 Locomotion Generation

In this paragraph we will focus exemplarily on walking. Basically there exist two types of approaches for automatic generation of locomotion. Whereas the first one tries to simulate the physiology of the human body using kinematics, other approaches adapt captured animation data according to external parameters, e.g. interpolating between walking and running to attain jogging. Here the complexity concerning biomechanical constraints is lower, because originalities of human walking are already defined in the animation sets, but they need motion data upfront. A promising approach we have implemented was the one described by [Park et al. 2004], which synthesis animation data from previously captured animation data according to different parameters, e.g. mood of character and style of walking. In a first step one has to pre-process the motion data and create animation sequences, which consist of one walking cycle with fixed speed, angle and mood. To walk on a given path or towards a specified target the sequences are automatically concatenated in our control component during runtime, and interpolated according to the input field values given by the application.

5.1.3 Hair and Skin Simulation

Besides motions that usually are consciously controlled like walking there are also other ones, which can’t be controlled explicitly,

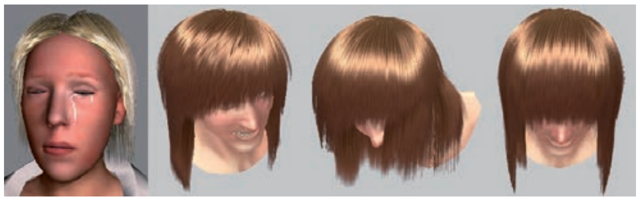

Figure 6: *Left: A woman crying and blushing. Right: Some frames taken from our real-time hair simulation with special hair shader.*

like blushing, tears running down a face, or the motion of hair when moving the head (as shown in Figure 6), but which also have to be simulated in order to create convincing virtual humans. Thus, means for realistic skin rendering also have to be provided. Additionally, we have also implemented nodes for doing hair simulation and rendering, which are described in [Jung and Knöpfle 2007]. The simulation is based on a kinematic multi-body chain, whose nodes are defined by the vertices of the original hair mesh that consists of many quad strips. Two node types are distinguished here: Anchor nodes are connected to the scalp, whereas all other vertices in the chain are free moving, due to external forces like gravity, length conservation, or the force resulting from turning the head (i.e. transforming the joint hierarchy above the hair mesh).

5.1.4 Speech Synthesis

Depending on the application different behaviors are required for animating virtual humans. When using e.g. conversational agents, interpersonal communication and therefore facial expressions, speech, and at least some sort of lip synchronization are crucial. In X3D based applications speech synthesis usually is done externally and the facial animation is updated and controlled via Java and the EAI. But this not only leads to latency but also affords a lot of complexity concerning scripting and scene design. Hence we propose an *AudioTTS* node, which is a text-to-speech (TTS) node that can be referenced by *Sound* nodes instead of an *AudioClip* node. The *AudioTTS* transforms written text to audio data by using a synthetic computer voice. The SFNode field 'voice' contains a *Voice* node, which describes the synthetic voice. The *Voice* node defines the parameters of computer generated voices. The 'name' field contains the name of the voice that corresponds to the voices, which are installed on the computer. If no name is given, the default voice is used. The 'gender' field determines the gender of the voice, possible values are "auto", "male" and "female", whereas the 'age' field determines the age of the voice, valid entries are "auto", "child", "teen", "adult" and "senior". The 'language' field finally controls the language of the voice (e.g. "en" for english or "de" for german; if not given the default language is used).

The SFString field 'text' of the *AudioTTS* node contains the text that gets spoken by that voice. Besides creating the audio data, this node can also provide weights to morph between different geometries (the morph targets). This can be used to animate the lips of avatars by mapping the resulting phonemes to their corresponding visemes. Therefore the 'visemeKey' field is used: If for instance the viseme "a" is represented by the first geometry, an "a" is put at index 0 of this field, etc. The 'weights_changed' outslot provides the weights for the different geometries used to create the final geometry. Usually, there is one geometry for each viseme provided by the text-to-speech system. For each geometry, one weight is calculated, whereas the sum of all weights is 1. After that, all geometries are multiplied with their respective weight and summed up. The result is an animation of the lips that is synchronous to the speech. The value of 'visemeDurationScale' determines how long the visemes

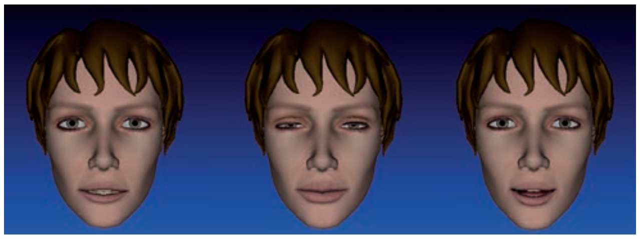

Figure 7: *Talking head, using AudioTTS and CoordinateMorpher.*

are displayed during the animation. By default the value is 0.5, which means that half of the time the current viseme is displayed, a quarter of the time is used to interpolate from the previous viseme to the current viseme, and a quarter of the time is used to interpolate from the current viseme to the next viseme. The 'autoSilentIndex' is the index of the silent or neutral geometry. When this field is non-negative, the node ensures that the corresponding geometry is shown at the end of the animation sequence.

```
AudioTTS : X3DSoundSourceNode {
  SFString [in, out] text         ""
  MFFloat  [out] weights_changed
  MFString []    visemeKey    []
  MFFloat  []    weightValue []
  SFNode   []    voice        NULL
  SFFloat  [in, out] visemeDurationScale 0.5
  SFInt32  [in, out] autoSilentIndex     -1
}

Voice : X3DSoundNode {
  SFString [] name      ""
  SFString [] gender    "auto"
  SFString [] age       "auto"
  SFString [] language ""
}
```

5.1.5 Facial Animation

As already stated, the *AudioTTS* node not only synthesizes speech, but also determines the resulting list of phonemes including the duration of every phoneme, and it also calculates the weights for the different visemes (as shown in Fig. 7). These weights can be used in two ways, either for animating the skin with the help of *HAnimDisplacer* nodes or by directly morphing the mesh, as is explained next. *HAnimDisplacer* nodes are usually used to control the shape of the face. Each *HAnimDisplacer* specifies a morph target that can be used to modify the displacement properties of the corresponding vertices. The scalar magnitude of the displacement is given by the 'weight' field and can be dynamically driven by an interpolator or a script. The mesh therefore can be morphed smoothly using the base mesh and a linear combination of all sets of displacement vectors.

Quite similar to the displacer node is the *CoordinateMorpher* node, another model-free approach for doing animations that was first proposed in [Alexa et al. 2000]. Assume you want to animate a face, and you have given n target states of your modeled face, a neutral one, and $n - 1$ other ones, e.g. a smiling one, one with open and one with closed eyes, one with raised eyebrows, and the other ones for representing the phonemes. The morpher node now regards each of these states as a base vector of an n dimensional space spanning all possible combinations of mesh deformations. In order to get valid linear combinations the coefficients (weights) of all morph targets must sum up to 1. For interpolating between different states additionally a *VectorInterpolator* node was introduced, because for each key time a vector of n key values is needed here. Unlike the *HAnimDisplacer* the morpher node is also suitable for animating a talking head or other objects not compliant to H-Anim. Additionally a *NormalMorpher* was defined here that linearly interpolates among the set of normals – something that is still not defined in the X3D specification for displacer nodes.

When using the X3D *AudioClip* node it is certainly also possible to generate animated speech by preparing all spoken texts in advance or via some external modules. But if lip sync is needed, first the lengths of all phonemes (usually provided by the TTS system, too) have to be determined, before they can be synchronized with the corresponding face displacements. Because a good viseme mapping is complicated, and within the X3D framework decoupled from the 'spoken' phonemes, a mechanism that is less error prone and easier to use has to be incorporated, because precise synchronization and scheduling is needed for simulating virtual characters. Thus, the proposed animation controller component, which is described in the next section, is also capable of synchronizing the computer voice with the corresponding face animations. Moreover, they are automatically combined with other morph targets (e.g. for displaying the emotional state), which are active at the same time.

5.2 Scheduling and Controlling Animations

5.2.1 Connecting the Layers

After having explained the high level language PML, and the low level extensions described above, the question remains, how this advanced animation control approach can be used in concrete settings. Thus, a generic scheduling and controlling element is needed, too. Therefore, we added some additional nodes, whose X3D interfaces are shown next. Figure 8 shows an overview of the proposed system architecture. Here, the *TimelineComposer* node is responsible for all scheduling and also deals as the PML interface and processor. Starting and stopping of animations and other actions is accomplished by setting the 'command' string of the *TimelineComposer* node with a valid PML file or string for defining the desired temporal order. This is similar in spirit to the 'url' field of a *Shader* node, which only is useful when having defined a valid GLSL or Cg shader program, or the 'url' field of a *Script* node, which only is useful when having defined some Java or JavaScript code. Alternatively one could also think of defining the temporal order and triggering the corresponding actions by implementing a time graph structure consisting of parallel and sequential *TimeContainer* nodes, as shortly discussed in section 2, for mapping the animation time to the fraction of the final key-frame intervals. But for more complex schedules this soon gets confusing and hardly maintainable. Furthermore, it does not provide any means for defining animation data centrally and mixing it with other animations.

Whereas the 'command' field contains an incoming PML script, the 'message' eventOut sends an outgoing PML message string. This way, the *TimelineComposer* node handles all communication with the system and forwards PML commands to its parser. During parsing, the scheduling block is sequenced and single action and definition chunks are created and transfered to the appropriate components. When having received a start message, the internal scheduler dispatches the action chunks to the *AnimationController* node of the corresponding character. The MFNode field 'animationController' holds references to the *AnimationController* nodes of all objects, which shall be animated. Whenever an actions script shall be executed, the *TimelineComposer* triggers all *AnimationController* nodes, which in turn access the respective data of their referenced animation container nodes (the *InstantAnimationContainer* for referring to transitions, which are state changes like toggling visibility, and the *TimedAnimationContainer* for storing all time based animations like key-frame animations and inverse kinematics) for processing this request.

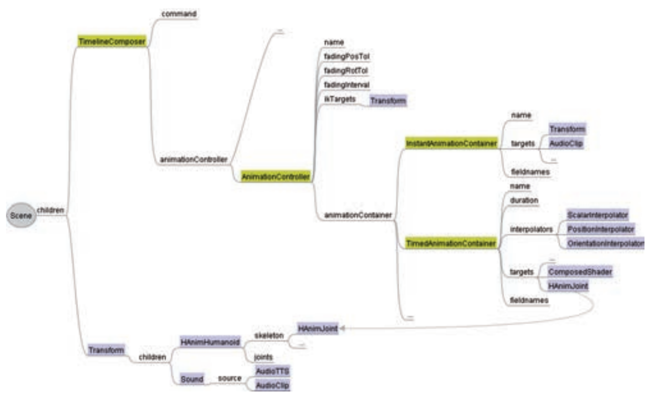

Figure 8: *Overview of proposed system architecture. Whenever the TimelineComposer receives a PML command, all requests are processed, and forwarded to the responsible AnimationControllers.*

```
TimelineComposer : X3DNode {
  SFBool   [in,out] enabled TRUE
  SFString [in,out] command ""
  SFString [out]    message
  MFNode   [in,out] animationController []
}
```

5.2.2 The Animation Controller

The *AnimationController* node controls a set of animations connected with a virtual character or any other object. Because a complex application can lead to an arbitrary number of postures and gestures or respectively animations, the main job of the *AnimationController* is to blend and cross-fade all kinds of animations. This is due to the requirement, that for correct blending, cross-fading, and generally updating the actions of an object at a single time step, the controlling unit needs knowledge of all animations, a task that sometimes can not be handled with the simple scripting and routing mechanisms of X3D. The 'name' field contains the name of the object to be controlled, which is relevant for the later mapping to PML scripts. The 'animationContainer' field contains references to all animations as defined by the animation container nodes. With the help of the 'ikTargets' field possible IK targets can be given, what is needed for parameterizing inverse kinematics animations like "look at A" or "point at B". Although blending avoids jumps in transitions, it can cause undesirable side effects like foot sliding. Therefore the fields 'fadingInterval', 'fadingRotTol', and 'fadingPosTol' can be used to specify the time interval and distances where blending should occur. The default values were empirically determined and led in most cases to the best results.

```
X3DAnimationBase : X3DNode {
  SFString [] name ""
}

AnimationController : X3DAnimationBase {
  SFString []        name ""
  MFNode   [in, out] animationContainer []
  MFNode   [in, out] ikTargets []
  SFFloat  [in, out] fadingInterval 0.2
  SFFloat  [in, out] fadingRotTol 0.7
  SFFloat  [in, out] fadingPosTol 3.0
}
```

The *AnimationController* and the abstract *X3DAnimationContainer* both inherit from *X3DAnimationBase*, an abstract base node that only defines a 'name' field. The *X3DAnimationContainer* contains the animated targets of an animation. The MFNode field 'targets' holds references of targets to be animated or changed (usually the joints), and the MFString field 'fieldnames' contains the names of the corresponding fields in order to find this field inside the target. This is needed, because if for instance an SFVec3f value shall be sent to a target node, e.g. a *Transform* node, it is often ambiguous, which field was meant (in this example it could be either of 'center', 'scale', or 'translation'). The *TimedAnimationContainer* node contains a set of interpolators of an animation (in the 'interpolators' MFNode field) and the original default duration of the animation (in the 'duration' field). Whereas the *TimedAnimationContainer* denotes actions with a certain duration, the *InstantAnimationContainer* denotes transitions, i.e. simple state changes like show, hide, start or stop. It therefore does not contain interpolators but instead it can hold the id of a media-object as defined in a definitions script.

```
X3DAnimationContainer : X3DAnimationBase {
  SFString [] name ""
  MFNode   [] targets []
  MFString [] fieldnames []
}

TimedAnimationContainer : X3DAnimationContainer {
  SFString [] name ""
  MFNode   [] targets []
  MFString [] fieldnames []
  MFNode   [] interpolators []
  SFFloat  [] duration 0
}

InstantAnimationContainer : X3DAnimationContainer {
  SFString [] name ""
  MFNode   [] targets []
  MFString [] fieldnames []
  SFString [in, out] mediaId ""
}
```

6 Examples and Discussion

6.1 Using the Scripting Interface...

Due to the above mentioned drawbacks concerning timing, synchronization, and the lack of an easy to use TTS system, in X3D based applications the output of spoken text is often still done with the help of on-screen displays instead of using audio output. In the following we discuss an example that demonstrates how this task can be simplified with our approach. We begin by describing the high level scripting interface, before explaining the corresponding X3D node extensions. As can be seen next, the phoneme to viseme mapping and all required animations should be defined first. Here the values of the 'id' attributes must match with the 'name' fields of the animation nodes. After that, the animations can be started by routing a filename or string with the PML actions script to the 'command' field of the *TimelineComposer* node. In this example the character asks something, and concurrently makes one gesture (`<par>` block), before doing another one (in `<seq>` block).

```
<definitions id="def1">
  <character id="Valerie" src="Valerie.wrl">
    <voice id="vVal" refId="Klara16"/>
    <viseme>
      <phoneme id="h" refId="H" intensity="1"/>
      ...
    </viseme>
    <multiPoses id="attract" dur="4167"/>
    <singlePose id="H" dur="1000"/>
    ...
  </character>
</definitions>
```

As can be seen in the definitions script above, the type of voice for parameterizing the *Voice* node is also defined, which is only useful in combination with the *AudioTTS* node. When having a closer look at the `<speak>` tag in the actions script shown next, one can notice, that the duration of this action is zero, because the needed phonemes and their lengths have to be internally calculated by the TTS system, and are not known until then. If lip synchronization shall be achieved by using a standard *AudioClip* node instead, via the tag `<audio src='hello.wav'>` an audio file must be provided. Additionally a list of phonemes with their respective durations, which have to be computed in advance, has to be declared as sub-tag of the audio tag like this: `<phoneme refId='h' dur='100'/>`. Although the first alternative is much easier to use, it has the disadvantage, that the exact duration is not known beforehand, because it is internally calculated during run-time, and thus can't be synchronized exactly with other actions. This could be alleviated by providing a higher level of abstraction, where the temporal order is given by using generic alignment attributes instead of a `<schedule>` block, but this not only requires a lot of care in order to avoid invalid states, but it is also not always unambiguously resolvable.

```
<actions id='act1' start='true'>
  <character refId='Valerie'>
    <speak id='a'>
      <text>Hello!</text>
    </speak>
    <animate id='b'>
      <multiPoses refId='present'/>
    </animate>
    <animate id='c'>
      <multiPoses refId='attract'/>
    </animate>
  </character>
  <schedule>
    <seq>
      <par>
        <action refId='c' begin='0' dur='4167'/>
        <action refId='a' begin='1000' dur='0'/>
      </par>
      <action refId='b' begin='0' dur='2733'/>
    </seq>
  </schedule>
</actions>
```

6.2 ...and the Controlling Component

After having explained the high level interface, we will show how this corresponds to our proposed nodes for animation control, and how they can be used in a concrete setting. The following code fragment shows exemplarily how to define interpolator based animations in *TimedAnimationContainer* nodes, and how an *AnimationController* node for a character or object can look like. In this framework, the interpolators are only used as data containers for key-value pairs, as depicted in Figure 9. Thus, there is no need for routes or other difficult to maintain helper structures, because all interpolators, which are active at a given time frame t_i, are solely used for the internal calculation of joint rotations etc., in order to have access to all required animation data for mixing animations correctly and efficiently. This way, both gestures from the example PML script above are automatically cross-faded, and additionally blended with the idle poses from the first example in section 4.

In addition, this concept is extensible not only concerning the scripting interface, but also the controlling component, because it allows to transparently include more sophisticated schemes for blending animations like the usage of transition motions for motion graphs, as well as those for motion generation, like the parameterizable motions used for locomotion generation explained in section

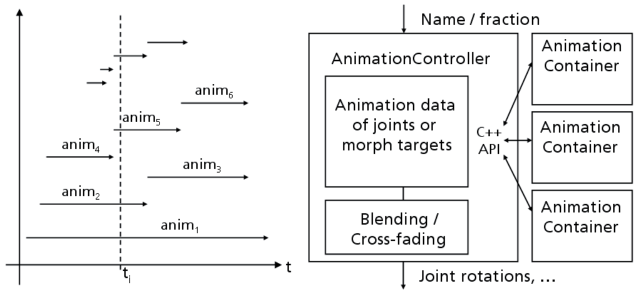

Figure 9: *Timeline at time t_i, and mixing in AnimationController.*

5.1.2. Obviously each animation container has to handle lots of data, especially if the key times and values were taken from motion capture data. Therefore animation containers can be reloaded during run-time by sending an appropriate definitions script. This is quite convenient, because due to the large amount of data, file sizes soon get too big for handling them in any editor.

```
DEF AC_Valerie AnimationController {
  name "Valerie"
  animationContainer [
    DEF attract TimedAnimationContainer {
      name "attract"
      interpolators [
        OrientationInterpolator {...},
        ...
      ]
      duration 4.1667
      targets [ USE Bip01_Spine, ... ]
      fieldnames [ "rotation", ... ]
    },
    ...
  ]
}
```

6.3 Discussion and Possible Extensions

Currently in X3D the only possibility to gain access to data in other files is via the import/ export mechanism, but this is only allowed for ROUTEs between files that are directly inlined, and therefore not viable for our problem. Consequently another option is to allow node re-USE over file boundaries. In this case the node names may not be unique any more, why we have implemented a slightly different addressing scheme that extends the *Inline* node with an additional field 'nameSpaceName', which allows referencing even over file boundaries with the syntax 'nameSpaceName::nodeName'. The question is, in how far this seems to break with the concepts of X3D, which already have been designed more than ten years ago with web based applications in mind. At this time, scene graphs were the only reasonable choice for 3D applications and a typical PC was almost swamped by rendering some gouraud shaded geometric primitives. Nowadays, a typical scene has much more content and a lot of interacting elements that cannot be kept in a single file. By allowing to reference nodes in other files, scene description and handling gets much clearer, but for usage in web environments one has to make sure, that only such nodes can be used, which explicitly may be shared by other people.

Referring to the different layers of abstraction as depicted in Figure 2 we have implemented a Steering component [Reynolds 2005]. It includes a set of nodes for simulating autonomous characters within a scene. They have the ability to navigate around in their world in a life-like and improvisational manner. By combining predefined behaviors like wander, seek or flee behavior, a variety of autonomous

Figure 10: *Left: A virtual tour guide. Right: Language learning.*

systems can be simulated. More on a proof-of-concept level we have also implemented the *Brain* node for representing the cognitive layer in order to create avatars that can communicate with a user of the X3D world. The node is parameterized with an Eliza style AIML file for defining the topic. By setting the SFString 'ask' field, one can receive the clear-text answers to the questions sent via the 'answer' outslot. But it should be stated, that to our opinion with X3D only the "body", namely the visual and auditive representation, should be modeled, but not the "mind".

7 Conclusion and Future Work

In this paper we have discussed, how humanoid animation can be efficiently controlled in the context of H-Anim/ X3D. The current H-Anim standard only defines the skeleton setup, consisting of the rigid segments and joints, and a skins and bones system for seamless skinning. Character animation itself is mainly accomplished with timers and simple linear interpolators based on predefined animation sets. However, definition and handling of animations have never been part of the H-Anim standard, and the built-in X3D animation mechanisms are not suitable for dealing with multiple animations that shall be combined and concatenated dynamically during run-time. Furthermore, X3D does not provide any support for more advanced features like audio nodes for text-to-speech, including the automatic calculation of actual phonemes and weighting factors for achieving lip synchronization.

Thus, to overcome some of the limitations concerning character animation and authoring, in this paper we have presented a few enhancements to the present X3D standard, comprising extensions for speech synthesis and lip synchronization, as well as for the animation of characters and other objects, including the ability to mix an arbitrary number of animations of different types, by providing nodes for controlling animations, which also convert a schedule and mix animations. Furthermore, we have explained the challenges of dynamics related to virtual characters, covering play-back and blending of predefined animations, as well as on-line motion generation. Concerning the latter aspect, it is important to note that the proposed animation control component is also capable of handling this type of animation and is thus extensible in consideration of new concepts of motion generation.

Moreover, we have presented a scripting and interface language that hides complexity and makes the scripting of behavior easier, and thus allows the implementation of story-lines at a higher level, allowing application developers to create and author realistic and interactive 3D environments easily. The proposed extensions were used and evaluated in different scenarios, like in the so-called ZAMB application developed cooperatively with researchers in the field of AI within the Virtual Human project [VirtualHuman 2007] (as shown in Figure 3), or in the applications shown in Figure 10. The left image shows a virtual tour guide explaining a historical site, whereas the right image shows a language learning scenario.

For future developments we plan to extend the inverse kinematics node, to incorporate a path planning component, and we would like to improve the proposed animation scripting language, which currently focuses on the specification of verbal and non-verbal behaviors of virtual characters in multi-party dialogs, by including extensions for walking etc. Furthermore we would also like to integrate motion graphs and a better method for generating suitable transition motions towards a more realistic combination of animations.

References

ALEXA, M., BEHR, J., AND MÜLLER, W. 2000. The morph node. *Web3D - VRML 2000 Proc.*, 29–34.

ARAFA, Y., KAMYAB, K., AND MAMDANI, E. 2003. Character animation scripting languages: a comparison. In *AAMAS '03: Proc. of the 2nd int. joint conference on Autonomous agents and multiagent systems*, ACM, NY, USA, 920–921.

AVALON, 2008. Avalon. http://www.instantreality.org/.

BABSKI, C., AND THALMANN, D., 2000. 3d on the web and virtual humans.

BUTTUSSI, F., CHITTARO, L., AND NADALUTTI, D. 2006. H-animator: a visual tool for modeling, reuse and sharing of x3d humanoid animations. In *Web3D '06: Proc. of the 11th int. conf. on 3D web technology*, ACM, NY, USA, 109–117.

DACHSELT, R., AND RUKZIO, E. 2003. Behavior3d: an xml-based framework for 3d graphics behavior. In *Web3D '03: Proc. of the 8th int. conf. on 3D Web technology*, ACM, NY, USA, 101–112.

DEL PUY CARRETERO, M., OYARZUN, D., ORTIZ, A., AIZPURUA, I., AND POSADA, J. 2005. Virtual characters facial and body animation through the edition and interpretation of markup languages. *Computers & Graphics 29*, 2, 189–194.

EKMAN, P. 1982. *Emotion in the human face*. Cambridge University Press.

GÖBEL, S., SCHNEIDER, O., IURGEL, I., FEIX, A., KNÖPFLE, C., AND A.RETTIG. 2004. Storytelling and computer graphics for a virtual human platform. In *TIDSE 04*.

HUANG, Z., ELIËNS, A., AND VISSER, C. 2003. Implementation of a scripting language for vrml/x3d-based embodied agents. In *Web3D '03: Proc. of the 8th int. conf. on 3D Web technology*, ACM, NY, USA, 91–100.

IERONUTTI, L., AND CHITTARO, L. 2005. A virtual human architecture that integrates kinematic, physical and behavioral aspects to control h-anim characters. In *Web3D '05: Proc. of the 10th int. conf. on 3D Web technology*, ACM, NY, USA, 75–83.

JUNG, Y., AND KNÖPFLE, C. 2007. Real time rendering and animation of virtual characters. *IJVR 6*, 4, 55–66.

KLESEN, M., AND GEBHARD, P. 2007. Affective multimodal control of virtual characters. *IJVR 6*, 4, 43–54.

KOPP, S., KRENN, B., MARSELLA, S., MARSHALL, A. N., PELACHAUD, C., PIRKER, H., THÓRISSON, K. R., AND VILHJÁLMSSON, H. H. 2006. Towards a common framework for multimodal generation: The behavior markup language. In *IVA*, 205–217.

KOVAR, L., GLEICHER, M., AND PIGHIN, F. 2002. Motion graphs. *ACM Trans. Graph. 21*, 3, 473–482.

LEE, J., CHAI, J., REITSMA, P. S. A., HODGINS, J. K., AND POLLARD, N. S. 2002. Interactive control of avatars animated with human motion data. *ACM Trans. Graph. 21*, 3, 491–500.

MARRIOTT, A., 2001. Vhml. http://www.vhml.org/.

MATEAS, M., AND STERN, A., 2003. Facade: An experiment in building a fully-realized interactive drama.

MCCANN, J., AND POLLARD, N. 2007. Responsive characters from motion fragments. *ACM Trans. Graph. 26*, 3, 6.

OPENSG, 2008. Opensg. http://opensg.vrsource.org/.

PARK, S., SHIN, H., KIM, T., AND SHIN, S. 2002. Online locomotion generation based on motion blending. In *Proceedings of ACM SIGGRAPH Symposium on Computer Animation*.

PARK, S., SHIN, H., KIM, T., AND SHIN, S. 2004. Online motion blending for real-time locomotion generation. In *Comp. Anim. and Virt. Worlds*, John Wiley a. sons.

PIWEK, P., KRENN, B., SCHRÖDER, M., GRICE, M., BAUMANN, S., AND PIRKER, H. 2002. Rrl: A rich representation language for the description of agent behaviour in neca. In *Proc. of WS "Embodied convers. agents, let's spec. and eval. them"*.

PONDER, M., PAPAGIANNAKIS, G., MOLET, T., MAGNENAT-THALMANN, N., AND THALMANN, D. 2003. Vhd++ development framework: Towards extendible, component based vr/ar simulation engine featuring advanced virtual character technologies. *cgi*, 96–106.

PREDA, M., AND PRETEUX, F. 2004. Virtual character within mpeg-4 animation framework extension. *IEEE Transactions on Circuits and Systems for Video Technology 14*, 7, 975–988.

REYNOLDS, C., 2005. Opensteer. http://opensteer.sf.net/.

SHAPIRO, A., PIGHIN, F., AND FALOUTSOS, P. 2003. Hybrid control for interactive character animation. In *PG '03: Proc. of the 11th Pacific Conference on CG and Applications*, IEEE Computer Society, Washington, DC, USA, 455–461.

TOLANI, D., GOSWAMI, A., AND BADLER, N. I. 2000. Real-time inverse kinematics techniques for anthropomorphic limbs. *Graph. Models Image Process. 62*, 5, 353–388.

VIRTUALHUMAN, 2007. VH. http://www.virtual-human.de/.

WALCZAK, K., WOJCIECHOWSKI, R., AND CELLARY, W. 2006. Dynamic interactive vr network services for education. In *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology*, ACM, New York, USA, 277–286.

WEB3DCONSORTIUM, 2005. Humanoid animation (h-anim). http://www.web3d.org/x3d/specifications/ISO-IEC-19774-HumanoidAnimation/.

WEB3DCONSORTIUM, 2007. X3d. http://www.web3d.org/x3d/specifications/ISO-IEC-FDIS-19775-1.2/index.html.

WEEKLEY, J. D., BLAIS, C. L., AND BRUTZMAN, D. 2007. Composing behaviors and swapping bodies with motion capture data in x3d. In *Web3D '07: Proc. of the twelfth int. conference on 3D web technology*, ACM, NY, USA, 195–200.

YANG, X., PETRIU, D., WHALEN, T., AND PETRIU, E. 2005. Hierarchical animation control of avatars in 3-d virtual environments. *IEEE Trans. on Inst. and Meas. 54*, 3, 1333–1341.