DYNAMIC CHARACTER ANIMATION

Stefan M. Grünvogel
Laboratory for Mixed Realities
Institute at the Academy of Media Arts Cologne
Am Coloneum 1
D-50829 Köln, Germany
E-mail: gruenvogel@lmr.khm.de

KEYWORDS
Skeletal animation, motion model, motion tree, motion clip operator

ABSTRACT

For creating real-time animations of 3D characters we introduce an animation engine for the dynamic creation of motions with motion models. Each motion models represents a small task like *walk* or *wave* and has its own set of parameters controlling the specific characteristics of a motion. The style and the characteristics of motion models can be changed in real-time. For this purpose pre-produced animations are combined and changed with clip operators. The animations of several motion models can be combined to play different motion simultaneously. To create the combined motions, the same clip operators are used.

1. INTRODUCTION

Real-time animation of 3D characters is often done by blending or masking short clips of motions produced by motion capturing or keyframe-animation (Theodore, 2002). The clips are short animations e.g. a high foot-kick, a low foot-kick, a slow walkloop, a fast walk loop and so on. There are several procedures to switch between different motions (e.g. from walk to run). The simplest idea is that the animations are made in a way, such that they can be concatenated without noticeable discontinuities. But the amount of work to prepare the animations in this way is tedious and eventually one has to create special transitional animations between two given animations. More sophisticated systems contain a blend mechanism to blend between the animations or insert a transition phase between animations automatically.

Another idea is to combine animations which do not influence the same joints of the character. This is called masking, e.g. if we have a wave motion and a walk motion, then the arms are animated by the wave motion and the feet and the pelvis by the walk motion.

The main drawback of considering motion as a small piece of unchangeable animations is that in reality every human movement can be performed in a great variety. For example, a walk movement can be described by its style (e.g. happy, aggressive, John Wayne), by its speed or by the frequency of the feet touching the ground. A jump movement can be characterised by the hight and the width of the jump.

Furthermore, motions often can be divided into parts which played consecutively, build the whole animation. These parts also are dependent on the style or the special way the motion is executed.

Motivated by the above points we adopted the notion of the motion models which was introduced by Grassia (Grassia, 2000). Motion models denote motions like walk or wave which produce their animation depending on given parameters. We expanded this concept for interactive real-time animation, where the parameters of a motion model can be changed while the animation is played. The advantage is, that we get an abstract interface for motions to create motions in different varieties.

After stated previous work in Section 2 the system environment is explained in Section 3 and the basic components of the animation engine are introduced. In Section 4 the character model is introduced which is an abstract representation of the animated character. Motion models are the crucial elements for the animation of characters, they are introduced in Section 5. In this section several clip operators and the term motion tree are introduced. They are used within the motion model to create the animations. In Section 6 the anatomy of the motion controller which is responsible for the creation of the overall animation of the character is explained. After experimental results in Section 7 we end with the conclusion and further research ideas in Section 8.

2. PREVIOUS WORK

We start first with a short review of how motions are parametrised in previous systems.

(Badler *et al.* , 1993) specify motions in the Jack System by control parameters which describe bio-mechanical variables. They also introduce *motion goals*, which are low level tasks their animation system can solve. A similar approach is studied in (Hodgins *et al.* , 1995).

Within the Improv-System (Perlin & Goldberg, 1996) human motions are described and parametrised by so called *Actions*. These *Actions* can be combined by blending them or building transitions between them. Their parameters denote possible perturbations of the original motion data by coherent noise signals. Perlin and Goldberg also state, that it is not always possible to combine every given motion with any other at the same time. For example it makes no sense to combine a *stand* pose with a *walk* motion. Taking this into consideration, they divide *Actions* into different groups, like *Gestures*, *Stances* etc. These groups provide the necessary information about the allowed combinations with other motions.

In (Sannier *et al.* , 1999) and (Kalra *et al.* , 1998) a real-time animation system VHD is presented which allows users to control the walking of a character with simple commands like *walk faster*.

Grassia (Grassia, 2000) introduces the term *motion model*, which we adopt. Motion models represent elementary tasks which can not be divided further. The level of abstraction of the motion models resembles the approach in (Perlin & Goldberg, 1996). The idea is that every human motion belongs to a certain category e.g. *walk*, *run*, *wave with hands*, *throw*, and each of this motions are independent. Each motion model has its own specific parameters controlling the motion generation process.

3. SYSTEM ENVIRONMENT

We first present the current system environment for the animation of characters. The architecture is similar to the IMPROV system of (Perlin & Goldberg, 1996), where a behaviour engine controls an animation engine. The animation engine is responsible for creating the animation of a character. In our environment, each character is represented by an animation engine (cf. Figure 1). The animation engine produces the animations in real-time and does not keep the geometrical representations (like the mesh and textures) of the character. The animation engine receives commands like e.g. start or stop a walk movement or reposition the character . The animation engine produces the animations and sends the animation data frame by frame to the trick_17 render engine. There the skeleton of the character is adjusted to the corresponding pose, the mesh and skin deformations are calculated and finally rendered.

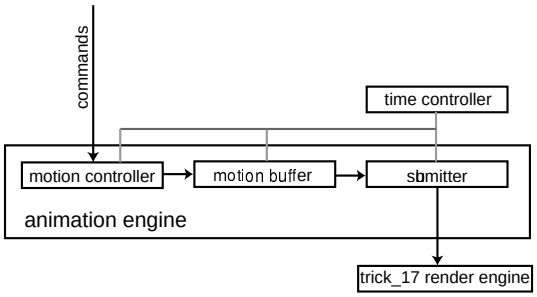

Figure 1: The System Environment.

The animation engine consists of three main components: motion controller, motion buffer and submitter where the last two components are running in a separate thread. Time is discretized by frames in the time controller and the animations of the character are produced with a fixed frame rate.

The motion controller receives commands for the animation engine and produces the overall animation of the character. The motion buffer reads and buffers the animation data from the motion controller and finally the submitter interface sends the data to the trick_17 rendering engine. In general every change of a joint angle is accompanied with a local mesh deformation. To prevent unnecessary recalculations only those joints of the character in the trick_17 render engine receive data for the current frame, for which there is actually new data.

4. THE CHARACTER MODEL

Each instance of an animation engine (which consists of a motion controller, a motion buffer and a submitter) represents one character in the environment. Within the animation engine all necessary informations about the character are collected in the *character model*. The character model holds the definition of the skeleton. The skeleton has a tree structure where each node represents a joint. The joints hold the translation and rotation values for the character in its base pose. The joints can also be accessed by an array data structure for faster reference. The order of the joints in the array data structure corresponds to the order of the animation data in each frame.

If we have several motions where each motion affects a different part of the body, then one can combine these motions easily by animating the different parts with the different motions. This is often called as masking. For supporting this, we additionally keep the information in the character model, which joints of the character belongs to a certain part of body. Currently for each character we have the same decomposition into parts of the body.

5. MOTION MODELS

In this section the purpose and scope of motion models are explained, followed by the clip operators and base motions, the building blocks of motion models. Then by an example an overview is given about the dynamic assembly of different motions within a motion model.

5.1 The Scope Of Motion Models

As mentioned above, the term *motion model* was introduced by Grassia (Grassia, 2000). We use his term in the same way. But his application *Shaker* uses motion models for the creation of non-interactive animations of virtual characters with a graphical user interface used by an expert. Thus his implementation of motion models are not made for an interactive environment in contrast to our implementation. Our animation engine is used within an augmented reality application (http://www.mqube.de) where characters are animated by non-experts in an interactive and dynamically changing environment. Therefore our implementation of motion models has to react in real-time to changes in the environment or commands given by the user.

Though there exists no mathematical definition which motions should be modelled as motion models and which not, there are some basic rules.

The purpose of a motion model is to produce motions which are independent. This means it should be able to recognise that the resulting movement of the body has started, executed and finally finished. Thus one has to think about the complexity and the purpose of the movement a motion model describes. The movements should not be too elementary like the rising of the left foot at the beginning of a walk movement. But they also should not be too complex. An example for a motion which is too complex would be the task to take a chair from one room and carry it to another room. For this purpose one has to lo-

calise the chair, then grasp it, do path planning for finding the way to the next room and so on.

Motion Models describe on the one side basic fundamental movements like walking, running, jumping. On the other side motion models also describe motions which need various informations to make adjustments of the environment (e.g. throwing a ball, grasping a bottle). Complex tasks (like the chair example above) which are too complex for modelling them as a motion model can be divided into subtasks. Then each of these subtasks can be animated by a motion model.

Before creating a new motion model one has to think about what actually characterises this new motion. Thus e.g. a *walk* motion model can be characterized by its walk style (e.g. sad, happy,...), its speed and its direction. The description of physical aspects of a motion, like speed or direction can easily be done by numerical parameters. For describing the style of a motion there are many approaches. Laban Movement Analysts can determine the appropriate Shape and Effort parameters of a movement (cf. (Laban, 1971), (Laban & Lawrence, 1974)) which was actually used by Liwei Zhao (Zhao, 2001) and Diane Chi (Chi, 1999) to create expressive motions. Psychologists describe in experiments the effect of nonverbal behaviour of virtual characters on spectators (cf. (Badler & Allbeck, 2001), (Bente *et al.* , 2000)) which also can be interpreted as a motion style. Furthermore animators often classify the style of their animations by basic human emotions (e.g. sad, angry) or by stereotypes (e.g. John-Wayne Walk, Charly-Chaplin Walk). Currently we use the last approach (basic human motions and stereotypes), because for our augmented reality application we can not assume that the user is accustomed to the terms of Laban movement classification or subtle psychological classifications. We are aware of the fact that with this approach it is possible that (in the worst case) different users may be of different opinions about the emotions a movement expresses. But in principle, the implementation of our motion model is flexible enough for describing styles in a more elaborate way.

For the creation of a new motion model one has to know furthermore, which of these characteristics can be changed while the new motion is executed and what kind of effect these changes have on the motion. The effect may also depend on the current state of the motion. Finally if the motion model has some objectives and these can not be fulfilled, then it must be known what it should do instead. Note that in the current implementation this feature is not yet implemented.

Perlin and Goldberg introduce in their work (Perlin & Goldberg, 1996) *atomic animated actions* (such as walk or wave) which are used by a Behavior Engine to create higher-level capabilities of a character (such as going to a store). They do not define exactly, where the border between atomic actions and the higher-level capabilites lie if one only looks at the resulting motions. But by the examples given in his paper (like Stand, Walk, Wave_Left etc.), the purpose of their atomic animated actions can be compared to the motions produces by motion models. Unlike our approach, the actions in Perlin's paper can be scripted by an author, but this requires low-level editing of the animations. On the other side with our approach the motion specific characteristics can be changed in real-time by high-level parameters.

5.2 Building Blocks Of Motion Models

All motion models (like e.g. the walk motion model) share a common interface, the AbstMotionModel class. The two most important methods are the *doCommand* and the *getSubtree*.

With the doCommand method, the motion model receives commands from the motion controller, which tells the motion model that it has to start or stop its animation or specific parameters of the motion model have to be changed. Each motion command consists of a command identifier which can be *start*, *stop* or *reset*. If the motion model receives *reset*, then it switches to its default state and stops its animation immediately. If it receives *stop*, then the motion model finishes its animation but in a believable way. This means that e.g. if the walk motion model receives the *stop* command while the left foot of the character is in the air, then the animation is played until the foot hits the ground and the character is in a stand pose. If the motion model receives the *start* command, then it just starts its animation with a default behaviour. The command can also possess a set of *motion parameters*. Each motion parameter consists two parts. The first part describes the purpose of the parameter. For each motion model a specific set of parameter purposes are predefined. For example the walk motion model can have parameters with purposes *speed* and *style*, where *speed* indicates that the parameter is used for defining the speed of the character and *style* is used to define the specific walk style of the parameter. The second part of the parameter actually holds the data of the parameter. If the data type of the data and the purpose of the parameter do not match, then this parameter is ignored by the motion model.

The *start* command is also used to change the state of the motion model. If the motion model has received the *start* command and is executing its animation and receives a new *start* command with a new set of parameters, then the motion changes its current motion according to this new set of parameters while it is played.

5.3 Clip Operators

Each motion model creates a certain motion by modifying and blending motion data according to the given parameters. The basis of each motion model are sequences of pre-produced animations. These pre-produced animation clips (explained below) are blended or modified by clip operators. Thus to create e.g. a walk motion, the motion model contains some pre-produced animations which are parts of walking motions and blends them together. To create a walk motion in a specific style (e.g. a happy style) then there must be a set of walk animations which are animated such that blended together the spectator has the impression of a happy walk.

The abstract AbstClip class (c.f. Table 1 and Figure 2) is the common interface for animation data. By start() the start frame and by length() the length of the animation is returned. Note that unlike in (Grassia, 2000) we also can have clips with infinite length. This is due to the fact, that in an interactive environment the length of a motion can depend on the user input and is therefor not predetermined. As an example consider

Method	Description
getActiveJoints()	Returns the joints for which there is actual data available
getFrame(Frame t)	Returns for frame t for each active joint the rotation or translation values if available
start()	Returns the start frame of the clip
length()	Returns the length of the clip

Table 1: The AbstClip Class

an user steering a character through an environment. The user gives the character the command to start to walk and then only changes the direction the character is walking. As long as the users does not stop the character, it will keep on walking. Thus the duration of the walk motion is not known in advance and we assign to it an infinite length. As soon as the users orders the character to stop, the walk motion can be turned into a finite length motion. Clips with finite length are defined on the interval [start(), start() + length() -1] and for clips with infinite duration we have length() ≤ 0.

The getActiveJoints() methods indicates for each joint if the clip has rotation or translation values available for this joint. For example a wave motion where a character waves with his left arm, needs only animation data from the arms and torso, because the feet of the character do not move. Thus one is able to save memory by using the animation data from the arms and torso and save computing time if only these parts of the body have to be manipulated.

The getFrame method returns for each valid frame two arrays of data. The first array represents translation values for the joints (given by 3D vectors) and the second the rotation values (given by unit quaternions). Every joint is assigned to a certain index in the array given by the character model. Played one after another, the array for each valid frame builds the animation of the skeleton.

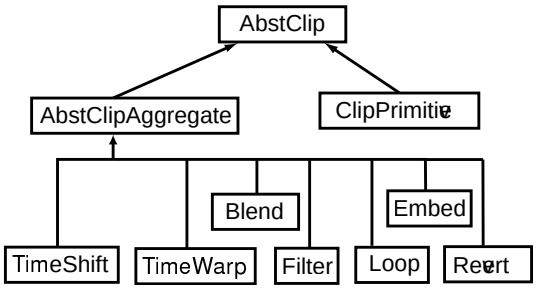

Figure 2: The Clip Classes.

The class ClipPrimitive (cf. Figure 2) holds the pre-produced animation data. To build complex animations these animations are modified by the classes derived from the AbstClipAggregate (cf. Figure 2). These derived classes are operators on clips, having one or two clips as arguments. Because every operator is itself derived from AbstClip, it can also be used as an argument for other clip operators.

5.3.1 *TimeShift*

Given a clip A and a frame number s we can define a time-shifted clip B by

$$B = TimeShift(s, A). \tag{1}$$

For this new clip we have

$$B(t) = A(t - s) \tag{2}$$

for every frame t.

5.3.2 *Filter*

This clip operator applies a finite impulse response (FIR) filter (cf.(Mallet, 1999)) on the animation data. A FIR filter is defined by its impulse response coefficients

$$h[\alpha], h[\alpha + 1], \ldots, h[\omega], \quad h[i] \in \mathbf{R} \tag{3}$$

with $\alpha \leq 0 \leq \omega$.

The translation values of a clip are filtered with techniques explained in (Wickershauser, 1994) where we apply the filter on the x,y and z components of the translation vectors separately.

As mentioned above, rotation data is represented by unit quaternions within the clips. Now unit quaternions can not be filtered in the same way as translation values by filtering the components separately. This would result in non-unit quaternions. Instead we apply the method which is described in (Lee, 2000), where the unit quaternions are locally parametrised by the exp mapping and then the FIR filter is applied on the parameter space.

Note, that also different algorithms are implemented for animation clips with finite and with infinite length.

5.3.3 *Loop*

Given a clip A which start at frame t_0 and a number n a clip B defined as

$$B = Loop(n, A). \tag{4}$$

If A has infinite length, then $B(t) = A(t)$ for every frame t. If A has finite length l and $n \leq 0$, then the clip is repeated an infinite time, i.e.

$$B(t) = A((t - t_0) \bmod l) \tag{5}$$

for all $t \geq t_0$. If $n \geq 1$, then equation (5) holds for $t \in [t_0, t_0 + nl - 1]$.

5.3.4 *Revert*

Given a clip A starting at frame t_0 with length l_0 and a time t_1 we can revert the clip in time by defining

$$B = Revert(t_1, A). \tag{6}$$

If A is finite (i.e. $l_0 \geq 1$) and $t_1 \notin [t_0, t_0 + l_0 - 1]$ or A is infinite and $t_1 < t_0$ then $B(t) = A(t)$ for all frames t. Otherwise B is defined as

$$B(t) = A(2t_1 - t) \tag{7}$$

for all $t \in [t_1, 2t_1 - t_0]$. Thus the resulting clip B is always finite for a valid t_1.

5.3.5 *TimeWarp*

Applies a time warp on the underlying clip (cf. (Witkin & Popović, 1995), (Grassia, 2000)), which squeezes or stretches the animation over time. A time warp is defined by at set of warp keys $K_1, \ldots, K_n$ with $n \geq 1$. Each warp key K_i has the form

$$K_i = (a_i, b_i) \tag{8}$$

where a_i and b_i are frames. Now, given a clip A with length l the time-warped clip B is defined by

$$B = TimeWarp(K_1, \ldots, K_n, A). \tag{9}$$

If for an i we have $l \geq 1$ and $a_i \notin [t_0, t_0 + l - 1]$ or if $l \leq 0$ and $a_i < t_0$ then the warp key K_i is not used for time warping. For the other keys it holds

$$B(b_i) = A(a_i) \tag{10}$$

and between the keys time is interpolated linearly.

5.3.6 *Blend*

Let A and B be two clips, and t_A, t_B, t_s and t_e be frames with

$$t_A \leq t_s \leq t_B \leq t_e. \tag{11}$$

Then we can define the blended clip C from clip A to clip B by

$$C = Blend(t_A, t_B, t_s, t_e, A, B) \tag{12}$$

The frames t_A and t_B are the new start frames of the two input clips A and B (cf. Figure 3) . The frame t_s denotes the frame where we start to interpolate from the first clip A to the second clip B. At frame t_e we have completely blended to clip B. Note that this quite general blend operator has to manage certain difficulties.

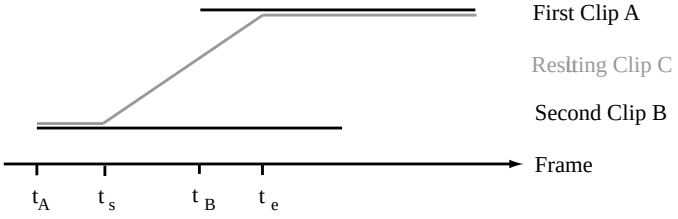

Figure 3: Blend parameters

First it is possible, that for a certain joint J there is animation data available in the first clip A but not in the second clip B (i.e. the joint is active in A and inactive in B). In this case, the joint is active in the resulting clip C where we extend the animation data of joint J at the end of clip A by repeating the last value.

If joint J is inactive in A and active in B, then we extend the animation data of joint B at the start of B backwards in time. Note that this can result in inconsistencies.

Second the resulting clip C has infinite length, if and only if B is infinite.

5.3.7 *Embed*

Let A and B be two clips, and $t_A, t_B, t_s^1, t_e^1, t_s^2, t_e^2$ be frames with

$$t_A \leq t_s^1 \leq t_B \leq t_e^1 \leq t_s^2 \leq t_e^2. \tag{13}$$

Then we can embed the clip B in clip A (cf. Figure 4) by

$$C = Embed(t_A, t_B, t_s^1, t_e^1, t_s^2, t_e^2 A, B). \tag{14}$$

This first blends clip A to clip B with the parameters t_A, t_B, t_s^1, t_e^1 as in equation (12). Then clip B is blended back to clip A, where the blend starts at frame t_s^2 and ends at frame t_e^2. Like for the Blend operator, if the clip B has finite length and t_s^2 or t_e^2 is beyond the end of the clip, the clip is prolonged artificially at the end for computing the blend.

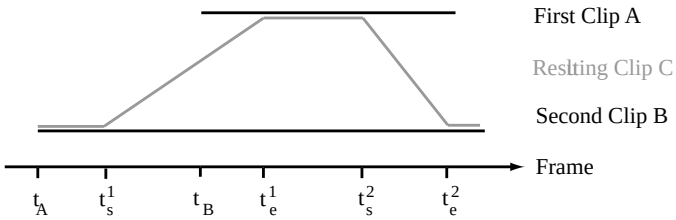

Figure 4: Parameters of the embed operator

By so far we have not implemented any operators, which effect a given ClipPrimitive such that the resulting clips has a new style. One could think for example of noise functions applied on certain joints or the techniques used in (Perlin & Goldberg, 1996). This could be a very promising approach to create variations of the motions without exchanging ClipPrimitives.

5.4 Base Motions

For the combination of different motions there is often some additional information about the motion needed to prevent the creation of artificial jumps or inconsistencies in the movement. Suppose for example that we have two different walk cycles A and B where the walk cycle A ends with the left foot on the ground while the walk cycle B starts with the right foot on the ground. If we would simply blend the beginning of walk cycle B with the end of walk cycle A then an unaesthetic jump would occur in the middle of the blended motion. A better idea would be to blend the end of clip A with those frames in the second clip B, where the left foot is on the ground. Thus we have to identify those frames in clip B where the left foot is in the ground.

Such an additional information is very crucial for combining several clip primitives. Therefor, each motion model holds a set of base motions. Each base motion actually keeps a preproduced animation in the form of a ClipPrimitive together with a list of annotations. Each annotation further describes the clip primitive and supplies the motion model with the additional information it needs to produce its animations correctly.

An annotation consists of two parts: the purpose of the annotation and the value, which can be of any data type. The purpose of the annotation could be e.g. technical information

like the frames where the left toes hit the ground in a walk cycle. Annotations also can describe information like the style of the animation (e.g. happy, nervous etc.). Each base motion contains its own set of annotations. Each motion model needs a certain set of base motions, where these base motions have to contain a motion model specific set of annotations.

Currently we create the annotations manually for the base motions. Thus a kind of knowledge base for the pre-produced animations is build, where every specific and important feature of an animation is stored. But this is also a source of inefficiency, because the manual creations of annotations is time consuming and error-prone. Instead of manual creation one could also think about the automatic creation of annotations by some automatic feature detection algorithm. But here further research has to be done,in particular to find out what kind of features are important for a specific animation and which features are needed for a certain motion model.

5.5 Dynamic Creation Of Motion With Motion Trees

The idea that motion models create animations by combing ClipPrimitives from their base motions is taken from (Grassia, 2000), where it has been applied for the off-line production of animations. In (Grassia, 2000) it is also mentioned that it could be possible to apply this approach for real-time generation of animations. We have actually implemented a simple method for using clip operators and ClipPrimitives to create dynamically changeable motion models. This allows the interactive change of animation properties.

As an example we consider the *walk* motion model. Walking can be divided into three phases. In the start phase the character start to walk from a standing posture. In the walk loop phase a walk cycle is played repeatedly. Finally the stop phase finishes the walk motion by stopping in a standing posture. Therefor the motion model contains at least three base motions containing the ClipPrimitives WalkStart, WalkCycle and WalkStop.

Now assume that the motion model *walk* receives the command to start at a specific frame t_0 with a specific speed s_1. From the annotations of the WalkLoop base motion, we know that the WalkLoop has speed $s_0 > 0$. If the WalkLoop ClipPrimitive starts at frame t_{wl} and has length l_{wl}, then we define the two warps keys

$$\begin{array}{rcl} K_1 & = & (t_{wl}, 0) \quad \text{and} \\ K_2 & = & (t_{wl} + l_{wl} - 1, (l_{wl} - 1)\frac{s_0}{s_1}). \end{array} \tag{15}$$

Thus the clip

$$A = TimeWarp(K_1, K_2, WalkLoop) \tag{16}$$

results in a walk loop clip with speed s_1 and length

$$l_A = (l_{wl} - 1)\frac{s_0}{s_1} + 1. \tag{17}$$

To create a walk loop with infinite length we apply the loop operator on A and get

$$B = WalkLoop(0, A). \tag{18}$$

Finally we assume here that the ClipPrimitive WalkStart ends with the same pose as WalkLoop starts (which we can get from the annotations). Denote by l_{ws} the length of WalkStart. Then we can create the resulting clip by

$$C = Blend(t_1, t_1 + l_{ws}, a, b, WalkStart, B) \tag{19}$$

for some frames a and b which can be fixed or can be determined from the annotations. This term is visualised in the operator tree (which we call *motion tree*) shown in Figure 5.

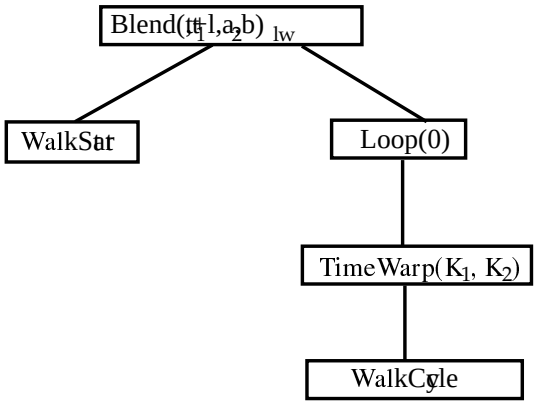

Figure 5: Motion tree of motion walk

The power of using clip operators is that it is possible for motion models to change their animations while their animation is played. For example, assume that the motion model has created the motion tree C defined in equation (19) and that the animation engine has started to play this animation. Assume further, that we are now at frame T with $T > t_1 + l_{ws}$, i.e. currently the animation of the time-warped WalkLoop A is played. Now at frame T the motion model walk receives the command that the animation has to be played with a new speed $s_2 > 0$. Then the motion model has to create a new motion tree. Because clip A is looped and time-shifted by equation (19), the motion model is able to calculate the start frame t_A of the last loop of clip A such that

$$t_A \leq T < t_A + l_A. \tag{20}$$

The clip with the new speed is given by

$$D = TimeWarp(K_2, K_3, WalkLoop) \tag{21}$$

where

$$\begin{array}{rcl} K_3 & = & (t_{wl}, 0) \quad \text{and} \\ K_4 & = & (t_{wl} + l_{wl} - 1, (l_{wl} - 1)\frac{s_1}{s_2}). \end{array} \tag{22}$$

If we define $\delta_A := T - t_A$ then we get the associated frame for clip D by $\delta_D := \frac{s_2}{s_1}\delta_A$. Thus the new tree

$$E = Blend(t_A, t_A + \delta_A - \delta_D, a', b', A, Loop(0, D)) \tag{23}$$

blends from clip A to clip D, where clip D is looped already (cf. Figure 6).

In a similar way, the style of a motion can be changed. If the motion model possesses a base motion with a ClipPrimitive WalkCycle_happy expressing a happy walk animation, then by a similar procedure as above the motion model can change the

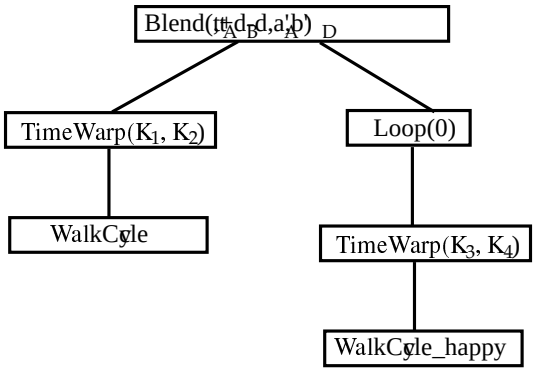

Figure 6: Blend to a walk motion with different speed

style of its motion while its animation is played. In this case, the annotations of the base motions have to be used to determine the correct blend parameters. One strategy is to blend the two clips at those frames together, where in each clip the same foot hits the ground.

Although it seems to be wasteful to throw away old motion trees and then construct new ones on the fly , in practice the motion trees have only a depth up to 6 and the computational time for the creation a new clip operator is restricted.

6. THE MOTION CONTROLLER

The general purpose of the motion controller is to provide an interface of the animation engine for other components to steer the character with motion models, to supply a general facility for repositioning and reorientation of the character, and to generate the appropriate motion.

6.1 The Interface

The interface of the motion controller serves different purposes.

The first one it to manipulate the state of the motion controller. The motion controller can play and pause the animation. It also can be set to default state, where no motion is active. Furthermore, the position and rotation of the character in world space can be changed.

The second purpose is to receive commands for motion models and to manage the animations created by the motion models. The commands for the motion controller consist of two parts. The first part contains the motion command introduced in Section 5.2 while the second part indicates which motion model is addressed by this motion command.

6.2 Managing Motion Models

The motion controller keeps all available motion models for the character. Each motion model can have an influence on the current played animation. To keep track of the state of the motion models and the animations they produce, the motion controller holds list of tokens where this information is stored.

Each token in the token list contains the following information:

- The identifier of the motion model for which the token keeps its information.
- The last motion command the motion model has received.
- The start frame of the current motion of the motion model.
- A flag which indicates if the animation has finite or has infinite length.
- In case of a finite-length animation, the last frame of the animation.
- In case of a finite-length animation, the last frame where the animation has influence on other animations. This frame can be bigger than the last frame of the animation, if the end of the animation is blended together with another animation by a transition. During the blend phase the animation has still influence on the overall animation of the character.

In the following, we describe what happens if the motion controller receives a new command.

6.2.1 *Updating The Token List*

If the motion controller receives a new command it first updates the token list. This means, that for every token in the token list it checks, if this token represents an animation with finite length. If this is the case and the last frame of influence from the animation is smaller than the current frame this animation does not have any influence on the motion of the character any more. Thus the corresponding motion model is reset to a default state and the token is deleted from the token list.

6.2.2 *Inserting Into The Token List*

After having updated the token list there are only tokens in the list for which the corresponding animation still has influence on the character. Next it is checked, if there is another token in the list, which belongs to the same motion model as the new motion command. If this is the case, then the new motion command could change the state of the motion model. The new motion command together with the token from the list is send to the motion model. If the motion models state is changed by the new command, its motion tree is updated and the corresponding information is put into the token. If the motion models state does not change by the new command (e.g. if it receives the start command for a second time with the same parameters), the token is not changed.

If there is no token in the list which belongs to the new motion command a new token is created and send together with the motion command to the motion model. If the motion model actually creates a new motion tree, then the appropriate information is put into the token and the token is appended at the end of the list. But it can also happen that the motion model does not create a new motion tree, e.g. if it receives a stop command, while it has not received a start command. In this case, the token is not appended to the list but instead deleted.

If after the update procedure in Section 6.2.1 and the insert procedure in this section no motion tree of a motion model was

changed or deleted, then the command for the motion controller has no influence on the animation and here the processing of the command finishes. Otherwise the motion tree of the motion controller has to be rebuild, which we describe now.

6.2.3 *Updating The Motion Tree*

For producing the overall motion for the character, the motion controller also keeps a motion tree. This tree is build up with the motion trees from the active motion models using the clip operators introduced in Section 5.3.

The motion controller has to update the motion tree as a result of the operations in Section 6.2.1 and Section 6.2.2. If its current motion tree is nonempty, than the current motion tree is deleted. Thereafter the motion controller retrieves the motion tree of the motion model corresponding to the first token in the list. If there is only one token in the token list, then this motion tree becomes the motion tree of the motion controller. If there are more tokens, then the motion controller retrieves the motion tree of the motion model corresponding to the next token. This motion tree is blended with the motion tree of the motion controller and the result becomes the motion tree of the motion controller. This is done for every token in the token list. As a result the new motion tree of the motion controller is build and the animation of the character is determined by this tree.

Building the motion tree from scratch every time a motion model is changed seems to be very inefficient. But if the motion tree of the motion controller is deleted, the trees of the motion models are not destroyed. Only the clip operators which connect the different trees have to be updated. Thus if the motion tree of the motion controller is rebuild only these last operators and the motion tree of the modified motion model have to be created anew. Furthermore, because every human has only a finite number of parts of the body, this defines a natural limit of the number of motions a character can do simultaneously. Thus the number of these operators is not high.

The hard task for the motion controller is to find the right operators for mixing the motion trees of different motion models together. Before starting a new motion model the motion controller first checks if the joints the motion model needs are in use by other motion models. Each motion model contains a list of the joints and parts of the body which are crucial for the motion. The animation of these joint can not be blended with other animations without destroying the task of the motion model. If these joints are currently blocked by another active motion model then this motion is a appended at the end of the animation. If the blocking motion has finite length, this means that the appended animation is played after the blocking animation has finished, disregarding its actual start frame. If the blocking animation has infinite length, then the appended animation is not played, until the blocking animation is replaced by a finite one.

If there is no conflict the animation of the corresponding joints are blended to the new animation. As an example consider Figure 7 where we started the wave motion model while the walk motion model is executed.

Currently we only have few motion models thus the parameters for the mixing operators between motion models are prescribed for each combination of motion models. Because for a growing number of motion models the complexity increases geometrically, automatic methods for mixing motion models have to be explored. First approaches can be found in (Grassia, 2000).

Figure 7: Blend of the motion model walk and wave.

7. EXPERIMENTAL RESULTS

We have implemented an experimental version of the animation engine with Visual C++ under Windows 2000 and Gnu gcc under Linux on a PC with 1100MHz AMD processor. For graphical output we use our trick_17 render engine, which runs with minor changes both under Windows 2000 and Linux by using OpenGL and GTK+. For test purposes we used a character created in Maya® with about 9000 Polygons, 3.6Mb texture and 69 joints. This character was exported into the proprietary file format of the trick_17 render engine. We use the computer's keyboard to steer the character interactively. The base motions were generated by key frame animation in Maya and by motion capturing. Every base motion is sampled at 30 frames per second.

The performance of the animation engine is promising: we have not found delays or a break in the continuity of the animation which could result from the creation of the motion trees in the motion models or in the motion controller.

8. CONCLUSION AND FURTHER RESEARCH

To summarise, motion models are independent motions and provide a high-level interface for the creation of animations. We have shown that it is possible to use clip operators for the interactive real-time production of animations. This enabled us to create an implementation of dynamic motion models for interactive real-time applications. Thus we are able to change the parameters of a motion model on the fly.

Multiple motion models can be played at the same time if they do not use the same parts of the body. The motion controller was introduced as a central component, responsible for managing the animations of the different motion models.

For creating the transition between a higher number of motion models further research has to be done because of the increasing number of possible combinations. As mentioned above, some work was done in (Grassia, 2000) in this field

but for real-time interactive applications this approach has to be adapted.

The clip operators are a flexible framework which needs further investigation. First the current clip operators have to be improved and new ones have to be added. For example in the current implementation the Loop operator just repeats a given clip. But if one wants to loop a clip where the start and stop frames do not perfectly fit together, small jumps in the animation can be observed. Thus a loop operator is necessary which blends automatically the two ends of the clip together as it gets looped.

Furthermore one has to investigate, if the approach of (Perlin & Goldberg, 1996) can also be used to create different motions with different styles by adding some noise functions onto the animations.

For having success it is necessary to combine the clip operator approach with real-time inverse kinematics such that the character can interact with objects in his virtual world. Thus constraints have to be fulfilled (e.g. that the character's feet do not intersect the floor) which can change during the execution of a motion model.

Base motions are used by the motion models to create their animations, where the annotations are crucial for the determination of the parameters needed in the clip operators. Currently these annotations are created by the animators, but this can be a tedious work if for every new animation the annotations have to be created. Certainly, for a big class of animations (e.g. walk animations) an automated procedure for feature detection would facilitate the creation of the annotations.

ACKNOWLEDGEMENTS

I would like to thank Thorsten Lange for his support on the trick_17 render engine.

This work was supported by the German Ministry for Education and Research (BMBF Grant 01 IR A04 C: mqube - Eine mobile Multi-User Mixed Reality Umgebung, www.mqube.de).

REFERENCES

Badler, N., & Allbeck, J. 2001. Towards Behavioral Consistency in Animated Agents. *Pages 191–205 of:* Magnenat-Thalmann, N., & Thalmann, D. (eds), *Deformable Avatars*. Kluwer Academic Publishers.

Badler, Norman I., Phillips, Cary B., & Webber, Bonnie Lynn. 1993. *Simulating Humans: Computer Graphics and Control*. Oxford University Press.

Bente, G., Krämer, N.C., Trogemann, G., Piesk, J., & Fischer, O. 2000 (November). An Integrated Approach Towards the Generation and Evalutation of Nonverbal Behavior in Face-to-Face Like Interface Agents. *In: Proceedings of the Interactive Intelligence Assistance & Mobil Computing Workshop, Rostock-Warnemünde, Germany*.

Chi, Diane M. 1999. *A Motion Control Scheme for Animating Expressive Arm Movements*. Ph.D. thesis, University of Pennsylvania.

Grassia, F. Sebastian. 2000. *Believable Automatically Synthesized Motion by Knowledge-Enhanced Motion Transformation*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh.

Hodgins, Jessica K., Wooten, Wayne L., Brogan, David C., & O'Brien, James F. 1995. Animating Human Athletics. *Computer Graphics*, **29**, 71–78.

Kalra, Prem, Magnenat-Thalmann, Nadia, Moccozet, Laurent, Sannier, Gael, Aubel, Amaury, & Thalmann, Daniel. 1998. Real-Time Animation of Realistic Virtual Humans. *IEEE Computer Graphics and Applications*, **18(**5), 42–57.

Laban, Rudolf von. 1971. *The Mastery of Movement on Stage*. London: Macdonald & Evans.

Laban, Rudolf von, & Lawrence, F.C. 1974. *Effort*. Second edn. Plymouth: Macdonald & Evans.

Lee, Jehee. 2000. *A Hierarchical Approach to Motion Analysis and Synthesis for Articulated Figures*. Ph.D. thesis, Korea Advanced Institute of Science and Technology, Department of Computer Science.

Mallet, Stéphane. 1999. *A Wavelet Tour of Signal Processing*. Academic Press.

Perlin, Ken, & Goldberg, Athomas. 1996. Improv: A System for Scripting Interactive Actors in Virtual Worlds. *Computer Graphics*, **30**, 205–218.

Sannier, Gael, Balcisoy, Selim, Magnenat-Thalmann, Nadia, & Thalmann, Daniel. 1999. "VHD: A System for Directing Real-Time Virtual Actors. *The Visual Computer*, **15**(7/8), 320–329.

Theodore, Steve. 2002. Understanding Animation Blending. *Game Developer*, **9**(5), 30–35.

Wickershauser, Mladen Victor. 1994. *Adapted wavelet analysis from theory to software*. A K Peters.

Witkin, Andrew, & Popović, Zoran. 1995. Motion Warping. *Computer Graphics*, **29**, 105–108.

Zhao, Liwei. 2001. *Synthesis and Aquisition of Laban Movement Analysis Qualitative Parameters for Communicative Gestures*. Ph.D. thesis, Computer and Information Department, University of Pennsylvania.

AUTHOR BIOGRAPHY

Stefan M. Grünvogel was born in Ellwangen, Germany and studied mathematics between 1990 and 1997. After finishing his diploma thesis in 1997 he worked as a postgraduate at the University of Augsburg in the field of mathematical control theory and finished his dissertation "Lyapunov spectrum and control sets" in 2000. After this he worked for debis before moving in 2001 to the Laboratory for Mixed Realities in Cologne. There he develops a real-time animation system and a choreography editor for the augmented reality project mqube (BMBF grant 01 IR A04 C, www.mqube.de).