

GENERATING DYNAMIC MOTIONS FOR ARTICULATED FIGURES

Stefan M. Grünvogel
Laboratory for Mixed Realities,
Institute at the Academy of Media Arts Cologne
Am Coloneum 1,
D-50829 Cologne, 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 motion models, which model a certain kind of motion like *walk* or *wave*. Each motion model has its own set of parameters controlling the specific characteristics of a motion. These parameters can be changed while a motion model is executed, thus this allows a change of the characteristics of a motion in real-time. The motion models produce animations by applying operators on short clips of animation and blending the results together. The parameters of the motion model determine the operators and the animation clips which are used to create the appropriate animation.

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 which can stand for their own like e.g. a high foot-kick, a low foot-kick, a slow walkloop, a fast walk loop and so on. If in the real-time application a clip is played (for example the walk loop) and then the user switches to another movement (e.g. a run loop), then either a short transition from one movement to the other is calculated or there is a third connecting clip between these two motions. Furthermore different clips not concerning the same joints of the character can be mixed by masking, i.e. if we have e.g. 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 animation is that in reality every human movement can be done 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 height and the width of the jump.

Furthermore, motions often can be divided into parts which are 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 real-time animation, where the parameters of a motion model can be changed during its animation is played. The advantage is, that this results in an abstract interface for each motion, which can create a motion in high varieties.

2. PREVIOUS WORK

(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*, which can stand for itself. Each motion model has its own parameters which controls the process of motion generation.

3. SYSTEM ENVIRONMENT

Before going into the details of the motion models we first present the current system environment for the animation of characters. Each character is represented by an animation engine (cf. Figure 1) which creates the animations in real-time. The animation engine receives commands controlling the char-

acter like e.g. start or stop a walk movement or positioning the character at an arbitrary position. The animation engine sends the produced animation data to the trick_17 render engine.

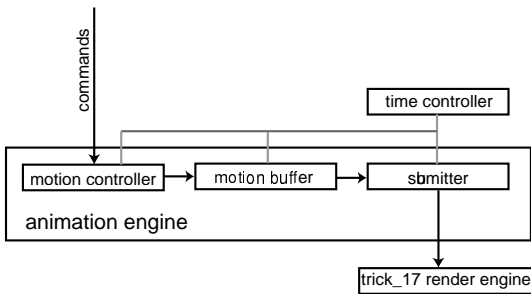


Figure 1: The System Environment.

The animation engine consists of three components each running in a separate thread. Time is discretized in frames by 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 send the data to the trick_17 rendering engine. For each frame a complete posture of the skeleton of the character is send to the render engine. The trick_17 renderer then calculates the mesh deformation of the character according to the posture and renders the picture.

4. THE SCOPE OF MOTION MODELS

Though there exists no 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 *can stand for their own*. 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 too complex motion would be the task to take a chair from one room and bring it to another room. For this purpose one has to localized the chair, then grasp it, doing 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.

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

5. BUILDING BLOCKS OF MOTION MODELS

Each instance of an animation engine represents one character. The character is defined by a tree structure (called character model) describing its skeleton. Because the mesh deformation of the character is done by the trick_17 renderer by the posture of the skeleton, we do not store the mesh data in the animation engine.

Motion models are a very simple common interface, the AbstMotionModel class. Every motion model is derived from this class. Motion Models get initialised by a character model and a source of the pre-produced animation clips. They have a doCommand-method which is used by the motion controller to control the generation of animation sequences within a motion model. At present, the motion controller receives commands like start, stop and stop_hard. The start command contains parameters which further describe the resulting motion. These parameters are motion model specific. E.g. the walk motion model has parameters controlling the style (happy, sad, etc.) and the speed of the walk. The parameters of a motion model have to be chosen in such a way, that the important characteristics of a motion can be influenced. The stop command just advises to motion model to correctly stop its motion at the actual state. If in a walk motion the stop command arrives at the motion model during the left foot is still in the air, the motion model correctly finishes this last step. The stop_hard method just finishes the motion immediately, i.e. as soon as the motion model receives this command it stops producing the animation which can result in an (for the observer) incorrect movement of the left foot. Although the visual result in the last case is in general not convincing, this effect is sometimes needed.

The building blocks of motion models are base motions and clips. The idea is that each motion model creates a certain motion by modifying and blending motion data according to the given parameters. As a basis each motion model has small sequences (base motions) of pre-produced animations which are used for mixing and blending.

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. The getActiveJoints method returns the joints of the skeleton for which the clip actually produces animation data. 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). Played one after another, the array for each valid

frame builds the animation of the skeleton.

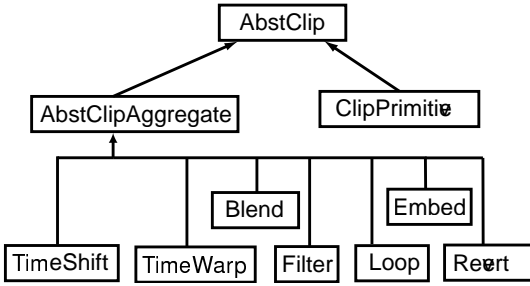
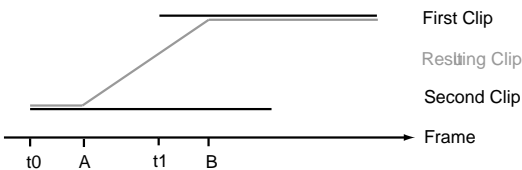


Figure 2: The Clip Classes.

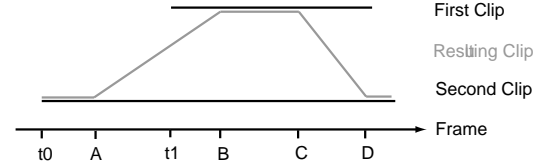
The class ClipPrimitive actually holds pre-produced animation data. These animations are manipulated by the classes derived from the AbstClipAggregate (cf. Figure 2). These derived classes are operators on clips. Because every operator is an AbstClip, it can also be used for the input to other operators.

Here we give a brief description of the implemented operators.

- *TimeShift(Frame nShift, AbstClip *pkClip)*
Shifts a clip on the timeline by nShift frames.
- *Filter(FilterCoef *pkFIR, AbstClip *pkClip)*
Filter the animation data with a FIR filter (cf. (Mallet, 1999), (Lee, 2000)). The impulse response coefficients of the filter are given by pkFIR. This can be used e.g. to smooth noisy animation data.
- *Loop(int nLoops, AbstClip *pkClip)*
Repeats pkClip either nLoops-time if $nLoops \geq 1$ or repeats the clip an infinite times otherwise. This is the only operator which can produce infinite length clips from finite ones. If pkClip has infinite length nothing is done.
- *Revert(Frame nNewStart, AbstClip *pkClip)*
Reverts pkClip in time at the frame nNewStart.
- *TimeWarp(Array<TimeWarpKeys> *pkWarpKeys, AbstClip *pkClip)*
Applies a time warp on the underlying clip (cf. (Witkin & Popović, 1995), (Grassia, 2000)), which squeezes or stretches the animation over time.
- *Blend(Frame t0, Frame t1, Frame A, Frame B, AbstClip *pkFirst, AbstClip *pkSecond)*
Blends the pkFirst clip to the pkSecond clip. t_0 and t_1 set the start frame of the first resp. second clip. A denotes the frame where we start to interpolating from the first to the second clip, B the frame where we blended completely to the second clip.



- *Embed(Frame t0, Frame t1, Frame A, Frame B, Frame C, Frame D, AbstClip *pkFirst, AbstClip *pkSecond)*
Blend from the pkFirstClip to the pkSecond clip and back to the pkFirstClip. The parameters t_0, t_1, A and B are the same as in Blend. At frame C pkSecond is blended back to pkFirst, and at D only the animation of pkFirst is played.



Note that often the algorithms within the operators for infinite length clips are different from the finite length clips. E.g. for finite length clips the algorithms in the Filter clip can be highly optimized (cf. Wickershauser (Wickershauser, 1994)). Special care is also needed if two clips are blended or embedded with different sets of active joints.

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). But this could be a very promising approach to create variations of the motions without exchanging ClipPrimitives.

6. CREATING MOTION WITH MOTION TREES

Motion models create animations by combing ClipPrimitives with the clip operators described in the previous section.

As an example we consider the walk motion model. Walking can be divided into three phases: the start phase, where we start to walk from a standing posture, the walk loop and the stop phase ending again in a standing posture. As the motion model walk gets the command to start at a specific frame t_0 with a specific speed w_1 , it creates the following term,

$$\text{Blend}(\text{TimeShift}(t_0, \text{WalkStart}), \text{Loop}(\text{TimeWarp}(w_1, \text{WalkCycle}))). \quad (1)$$

This term can be visualized in the operator tree (which we call motion tree) shown in Figure 3 (A).

The two ClipPrimitives WalkStart and WalkCycle contain the animations for first and the second phase of the motion. The WalkStart gets time-shifted to the start frame t_0 of the animation. The WalkCycle is time-warped with warp-keys W_1 resulting from w_1 for controlling the speed of the animation. The result is looped for an infinite time producing an infinite-length clip and is blended with the end of the time-shifted WalkStart clip. The result is a clip which lets the character start walking. If the motion model started and the current animation is in the WalkCycle loop (i.e. the animation is produced by the right branch in tree of Figure 3 (A)) one can simply change the speed of the character. As the motion model receives the command to change the speed to w_2 , it replaces expression (1) by

$$\begin{aligned}
& \text{Blend}(\text{TimeShift}(t_1, \\
& \quad \text{Loop}(\text{TimeWarp}(W_1, \text{WalkCycle})), \\
& \quad \text{Loop}(\text{TimeWarp}(W_2, \text{WalkCycle}))). \quad (2)
\end{aligned}$$

Here t_1 is t_0 plus the time passed since the last full pass of $\text{TimeWarp}(W_1, \text{WalkCycle})$, and W_2 are the appropriate keys which are derived from the speed w_2 .

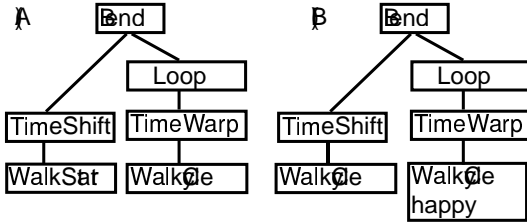


Figure 3: Walk Motion.

The style of the motion can be easily altered by changing the underlying ClipPrimitives. As an example, assume that the ClipPrimitives WalkStart and WalkCycle represent neutral motions, and we also have a WalkCycle_happy, representing a motion which expresses more joy and dynamic. If we currently are in the WalkCycle phase of the walk motion then the style of the motion can be changed by creating the motion tree as in Figure 3 (B), which is a visualization of the following expression:

$$\begin{aligned}
& \text{Blend}(\text{TimeShift}(t_1, \text{WalkCycle}), \\
& \quad \text{Loop}(\text{TimeWarp}(W_1, \text{WalkCycle_happy}))). \quad (3)
\end{aligned}$$

There the WalkCycle clip is played to its end and then blended with the looped WalkCycle_happy clip.

For correctly blending and manipulating these ClipPrimitives additional information is needed. E.g. for blending the WalkCycle into the WalkCycle_happy it is important to know the frame, when the feet of the character touch the ground and when they lift off. This information is used to determine the correct parameters for the Blend clip.

Thus each ClipPrimitive object within a motion model belongs to a base motion (cf. (Grassia, 2000)). Base motions consist of ClipPrimitives and Annotations, which hold the additional information for the animation data. These are on the one hand informal annotations, like the style of the motion (e.g. aggressive, tired, happy) and on the other hand special spatio-temporal relations.

Thus a motion model can hold several records of base motions each representing the movement in a different style. The commands from the motion controller determine the special parameters which are used for the construction of the motion tree with the help of the Annotations.

7. THE MOTION CONTROLLER

The motion controller receives commands from the animation engine, controls the motion models and produces the overall animation of the character.

The command set of the animation engine is fairly simple. There are two classes of commands. The first class controls the behaviour of the animation engine (e.g. resetting or positioning the character). The second class of commands (motion commands) is used for starting, stopping or changing the animations of motion models. The motion commands also hold parameters which are specific for the motion model. To keep an overview over the active motion models the motion controller administers the motion commands in a command list. Only those commands are kept in the list for which the corresponding motion model still produces animation data.

The motion controller also holds a motion tree which is build from the motion trees of the active motion models. This is done by passing through the command list and getting for each command the motion tree of the appropriate motion model. Then the motion trees from the motion models are blended and mixed together with the help of the clip operators from Section 5.

A rebuild of the motion tree is only necessary, if a new command is appended to the control list or the state of an active motion model is changed (e.g. by changing parameters or by stopping the motion model). In both cases, the concerned motion model builds its motion tree anew. The motion trees of the other motion models stay unchanged. The motion controller then deletes its old motion tree and builds a new one by parsing through the command list and composing the motion trees from the motion models.

At a first glance this seems to be an expensive operation. But practical experience shows that only very few motion models are active at the same time. Every human has only a finite number of parts of the body, thus this defines a natural limit of the number of motions a character can do simultaneously. Thus besides the cost of building the motion tree of the changed motion model, we only have to blend a few motion trees.

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 models then the motion command is rejected. Otherwise the animation of the corresponding joints are blended to the new animation. As an example consider Figure 4 where we started the wave motion model while the walk motion model is executed. At the moment 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 4: Blend of the motion model walk and wave.

8. 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` renderer, which runs with minor changes both under Windows 2000 and Linux by using OpenGL and GTK+. For test purposes we used a character with about 9000 Polygons and 3.6Mb texture, which was created in Maya and exported into the proprietary file format of the `trick_17` renderer. We use the computer's keyboard to interactively steer the character. The base motions were generated by keyframe animation in Maya and 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 come from creation of the motion trees in the motion models or in the motion controller.

9. CONCLUSION

To summarise, motion models create movements which can stand for their own. They provide a high level interface for each motion and allow to change a movement during it is executed by the character. Multiple Motion models can be played at the same time.

For creating the transition of a higher number of motion models, further research has to be done. Also the use of clip operators which can change the characteristics of animation clips has to be investigated.

ACKNOWLEDGEMENTS

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

This work was supported by the BMBF grant 01 IR A04 C (mqube - Eine mobile Multi-User Mixed Reality Umgebung).

REFERENCES

- Badler, Norman I.; Phillips, Cary B. and Webber, Bonnie Lynn. 1993. *Simulating Humans: Computer Graphics and Control*. Oxford University Press.
- 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. and O'Brien, James F. 1995. Animating Human Athletics. *Computer Graphics*, **29**, 71–78.
- Kalra, Prem; Magnenat-Thalmann, Nadia; Moccozet, Laurent; Sannier, Gael; Aubel, Amaury and Thalmann, Daniel. 1998. Real-Time Animation of Realistic Virtual Humans. *IEEE Computer Graphics and Applications*, **18**(5), 42–57.
- Lee, Jehoo. 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 and 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 and 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 and Popović, Zoran. 1995. Motion Warping. *Computer Graphics*, **29**, 105–108.

AUTHOR BIOGRAPHY

Stefan M. Grünvogel was born in Ellwangen, Germany and went after his military service to the University of Augsburg, Germany where he 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).