

# Implementation of a 3D Virtual Drummer

Martijn Kragtwijk, Anton Nijholt, Job Zwiers  
Department of Computer Science  
University of Twente  
PO Box 217, 7500 AE Enschede, the Netherlands  
Phone: 00-31-53-4893686  
Fax: 00-31-53-4893503  
email: {kragtwijk,anijholt,zwiers}@cs.utwente.nl

## Abstract.

We describe a part of a system which generates a 3D animated drummer based on the contents of a sound wave. The focus of this paper will be on the automatic generation of 3D animations, based on an abstract representation of the music (a MIDI file). The system is implemented in Java and uses the Java3D API for visualisation.

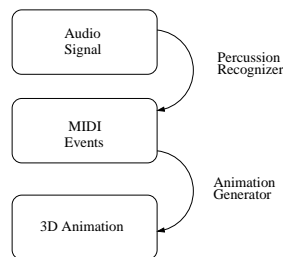
## 1 Introduction

In this paper we describe preliminary results of our research on virtual musicians. The objective of this project is to generate animated virtual musicians, that play along with a given piece of music. The input of this system consists of a sound wave, originating from e.g. a CD or a real-time recording.

There are many possible uses for an application like this, ranging from the automatic generation of music videos to interactive music performance systems where musicians play together in a virtual environment. In the last case, the real musicians could be located on different sites, and their virtual counterparts could be viewed in a virtual theatre by a worldwide audience. Additionally, our department is currently working on instructional agents that can teach music, for which the work we describe in this paper will be a good foundation.

For our first virtual musicians application, we have restricted ourselves to an animated drummer. However, the system is flexible enough to allow an easy extension to other instruments.

As figure 1 shows, the total task can be separated into two independent subtasks:



**Fig. 1.** An overview of the entire system

- Percussion recognition: the translation from a ‘low level’ description of the music

(the sound wave) to a abstract, ‘high level’ description of all percussion sounds that are present in the signal. The recognised notes are stored as MIDI events. This part of the system is subject of our still ongoing research, and will therefore not be further explained here. For an introduction on how we plan to solve this problem, please refer to [6].

- Animation generation: the creation of the the movements of a 3D avatar playing on a drum kit. For the remainder of this article, we will focus on this part of the system. In section 2, the transformation from MIDI to animation is described using a number of relatively simple algorithms, to maintain a clear view on the system as a whole. In section 3, some more advanced techniques are described, that result in more ‘natural’ motion.

In this section, we describe how our system generates animations automatically. The various algorithms discussed here are kept rather simple on purpose, to maintain a clear view on the system as a whole. In section 3, more advanced techniques (that give better results) will be explained.

## 2 Basic Algorithms

### 2.1 Overview of the system

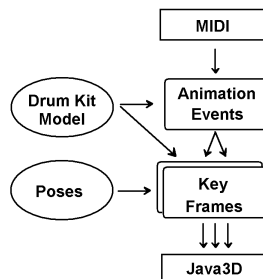


Fig. 2. Animation generation

A general overview of the animation generation is shown in figure 2. An abstract description of the animation (in this case, a list of time-stamped MIDI events) is transformed into a ‘concrete’ animation. This lower-level description of the animation is defined in terms of ‘key frames’ [2] that can directly be used by the graphical subsystem to animate objects in the scene.

Our implementation uses the Java3D engine for visualisation purposes [4]; the geometry of the 3D objects we have used has been created using Virtual Reality Modeling Language (VRML, [10]).

### 2.2 ‘Pre-calculated’ versus ‘real-time’ animation

In our current off-line implementation, the piece of music to be played is completely known in advance as a list of MIDI events. Therefore, the entire animation can be computed before it is started. In a real-time situation, where the system has to respond to incoming MIDI events, this would not be possible. In that case, a short animation should be constructed and started immediately for each note that occurs in the input.

A great advantage of pre-calculating the entire animation is that the transitions between strokes will be much smoother: for each note we already know which drum will be struck next, and the arm can already start moving towards that drum.

### 2.3 Polyphony Issues

Monophonic instruments (such as the trumpet or the flute) are relatively easy to animate, because each possible sound corresponds to exactly one ‘pose’ of all fingers, valves, etc., and only one pose can be active at each moment in time. Highly polyphonic instruments (such as the piano) are much more difficult, because there are many different ways (‘fingerings’) to play the same piece of music, and a search method is needed to find a good solution [5]. The drum kit could be viewed in between these two extreme examples: up to four sounds can be started simultaneously.

### 2.4 Drum Kit Model

In this section we will describe the parameters that are used to model different drum kits.

**Event Types.** The General MIDI standard [8] defines 47 different percussive sounds. The standard includes different versions of the same sound, for example “Crash cymbal 1” and “Crash cymbal 2”. Our application should treat both events in the same way.

Additionally, there are six different tom-tom sounds, while a ‘real’ drum kit usually only has 2 or 3 tom-toms. It may be clear that we have to define a smaller set of ‘animation event types’ in the drum kit model. The MIDI events from the input file can then be mapped onto animation events, that have a type code.

Our current implementation distinguishes between the following animation event types: BASS, SNARE, RIM, HIGHTOM, MIDTOM, FLOORTOM, CRASH, RIDE, RIDEBELL, SPLASH, CHINA, HIHATOPEN, HIHATCLOSED, HIHATPEDAL, COWBELL.

The animation event types do not necessarily have to have a one-on-one correspondence with the objects in the 3D scene, because 2 or more event types can belong to the same drum/cymbal, with a different ‘hit point’.

**Other Parameters.** Other parameters that are defined in the drum kit model:

- For each event type, a preferred hand: -1 (“left”) or 1 (“right”).
- For each event type, a parameter *minTimeGap* that determines how fast that particular event type can be played with one hand. This parameter will be explained in more detail in section 2.6.

### 2.5 MIDI Parsing

First, the list of MIDI events is transformed into a list of ‘animation events’ according to the mapping defined in the drum kit model (see section 2.4). Besides having a type code, an animation event has an associated velocity  $vel_{event}$  in the range [0..1].

Secondly, the list of animation events is parsed to remove double events<sup>1</sup> and distribute the events over the different animated objects. Objects in the scene respond to a

---

<sup>1</sup>some MIDI files that were used contained ‘double events’, that is: multiple events on the same channel, with the same time stamp, the same note number and the same velocity. These extra events do not contain new information, nor do they increase the velocity, therefore we can discard them.

subset of animation event types.

Three new event lists are created (one for the hands, one for the left leg and one for the right leg) and the animation events from the original list are distributed between them. The event list that is used for the hands will later be subdivided for the left and right hand; this is discussed in section 2.6.

## 2.6 Event Distribution

Animation events that can be played by both hands (i.e. all events except BASS and HIHATPEDAL) need to be distributed between the left and right hand in a natural looking way.

The first hand assignment algorithm that we have tested was designed to be as simple as possible. It is based on the following principles:

1. No more than two events, that are played with the hands, can have the same time stamp.
2. For each event type, there is a preferred (default) hand that should be used if possible.
3. When playing fast rolls, both hands should be used.

In our system, these principles were implemented in the following way:

- When more than two events (that should be played with the hands) are found to have the same time stamp, all but two are deleted.
- A parameter *defaultHand<sub>eventType</sub>* is specified for all event types. In our implementation, the SNARE and RIM events have the default hand set to 'left', while 'right' is the default hand for all other events.
- A parameter *minTimeGap* is defined, that determines how fast an event can be played with *one* hand. This parameter can have a different value for different event types, because the tendency to alternate hands varies from one drum type to another. For example, the hi-hat is usually played with the right hand; only in very demanding situations (fast rolls) both hands will be used. On the other hand, hand alternation on the high tom is much more common.

These principles are implemented in algorithm 2.1. It consists of two phases:

1. default hand assignment
2. hand alternation

---

**Algorithm 2.1** A simple algorithm for event distribution

---

```
iterate over all events e:
    hand(e) := preferredHand(type(e))
iterate over all triplets of subsequent events (e1,e2,e3):
    if    hand(e1)=hand(e2)=hand(e3)
        AND
            Time(e2) - Time(e1) <= minTimeGap(type(e1))
        OR
            Time(e3) - Time(e2) <= minTimeGap(type(e3))
    then
        hand(e2) := otherHand(hand(e2))
```

---

## 2.7 Pose Creation

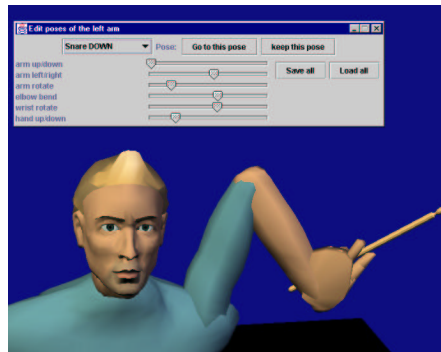


Fig. 3. The graphical poser interface, applied to the left arm

A graphical user-interface (GUI) is provided to create 'poses' manually. Figure 3 shows a screenshot of the GUI applied to the left arm. A pose consists of a set of angles or translation values: one for each degree of freedom. With the horizontal sliders, the user can change these values.

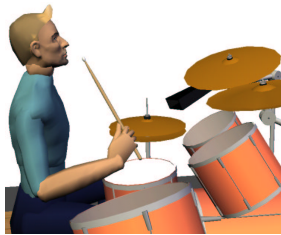


Fig. 4. 'MID TOM UP'



Fig. 5. 'MID TOM DOWN'

For each limb, two poses should be specified for each animation event type that it supports: the 'DOWN' pose (the exact situation on contact) and the 'UP' pose (the situation just before and just after the hitting moment). Examples of 'UP' and 'DOWN' poses are shown in figures 4 and 5.

Once a good position is achieved, it can be stored in the pre-defined list of poses. The entire list can be saved to disk, to preserve the information for a next session.

**Motivation.** We have chosen for manually setting the poses through a GUI interface, instead of using motion capture [11] or inverse kinematics for the following reasons:

**Costs:** Motion capture equipment is expensive, and requires a complete setup with a real drum kit that matches the 3D kit. If one would want to change something in the 3D drum kit (for example, moving a tom-tom) the whole capturing would have to be done all over again.

**Simplicity:** there are only a small number of poses, and they have to be set only once for a new drum kit configuration.

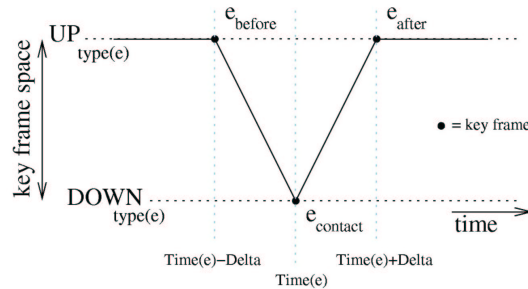
**Flexibility:** besides for the setting of poses for the arms and legs, the interface can also be used for the hi-hat stand and pedal, the cymbal stands, the parts of the bass pedal, and giving the snare, bass drum and tom-toms their position and orientation in the 3D scene.

## 2.8 Key Frame Generation

In this section, the transformation from lists of ‘abstract’ animation events to time lines (containing ‘concrete’ key frames) is discussed. Since a different approach is used for the avatar and the drum kit, they are discussed separately.

**Avatar Animation.** The poses that were created with the GUI interface (see section 2.7) are used to create key frames for the animation of the limbs. For each joint, the list of animation events that should be played by the corresponding limb is parsed in the correct temporal order.

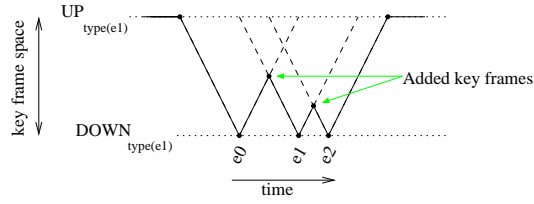
For each abstract animation event  $e$ , three key frames (a ‘stroke’) are added to the animation time line:  $[e_{before}, e_{contact}, e_{after}]$ . In the simplest version of the algorithm, these are equal to the UP, DOWN and UP poses for the corresponding event type. A constant  $delta$  determines the time between the key frames within a stroke (100ms is a useful value). See figure 6 for a graphical representation of a stroke that will be used throughout this chapter.



**Fig. 6.** A basic stroke, consisting of key frames ‘before’, ‘contact’ and ‘after’

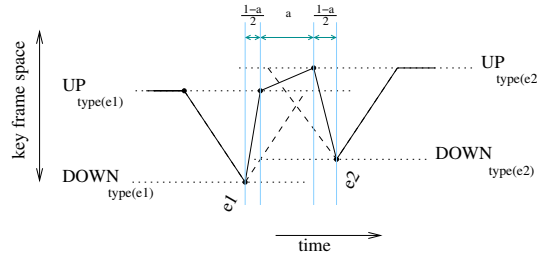
If the time gap between subsequent animation events  $e1$  and  $e2$  is less than  $delta$ , their key frames will overlap, and special care has to be taken. We distinguish between two cases:

- If  $e1$  and  $e2$  are of the same event type (e.g. both are ‘SNARE’ events), the last key frame of  $e1$  and the first key frame of  $e2$  are replaced by an interpolated key frame  $e_{New}$ : the less time between  $e1$  and  $e2$ , the closer the new key frame will be to the ‘DOWN’ key frame, as can be seen from figure 7.



**Fig. 7.** New key frames in the case of overlapping events of the same event type

- If  $e1$  and  $e2$  are of different event types (e.g. a ‘SNARE’ and a ‘HIGHTOM’ event), more time is needed to bring the arm from the ‘after’ key frame of  $e1$  to the ‘before’ key frame of  $e2$ . To accomplish this, the time difference between  $e1_{contact}$  and  $e1_{after}$ , and between  $e2_{before}$  and  $e2_{after}$  is shortened. A parameter  $a$  ( $0 < a < 1$ ) determines the fraction of the time between the events that is used for moving the arm from  $e1_{after}$  to  $e2_{before}$ .



**Fig. 8.** New key frames in the case of overlapping events of different event types

**Drum Kit Animation.** The original event list, containing all animation events (i.e. for all limbs) is used to animate the 3D drum kit.

*Pedals:* The bass pedal and the hi-hat are animated through the same kind of strokes as are used for the avatar. Because the pedals and the feet have their ‘UP’ and ‘DOWN’ key frames at exactly at the same moments in time, the illusion is created that the feet really ‘move’ the pedals.

*Cymbals:* For the vibration animation of a cymbals, a number of key frames are added to its time line starting at the ‘contact’ time stamp of a cymbal event. These key frames are computed by rotating the cymbal object around its local X and / or Z axis. The angles are sampled from an exponentially decaying sinusoid:

$$angle(t) = \eta^t \alpha_{max} \sin(\beta t)$$

In the above equation,

- $\alpha_{max}$  represents the maximum angle
- $\eta$  is the damping factor of the vibration ( $0 < \eta < 1$ ): low values for  $\eta$  result in a fast decay.

- $\beta$  determines the speed of the vibration: a higher value for  $\beta$  corresponds to a shorter swing period.

Overlapping vibrations are much easier to deal with than overlapping strokes. When the first time stamp of a new vibration falls within the time range of an previous one, the remaining key frames are deleted<sup>2</sup>.

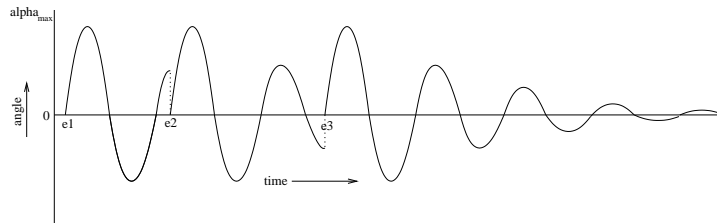


Fig. 9. overlapping vibrations for events [e1,e2,e3]

### 3 Advanced algorithms

In this section, some advanced techniques will be discussed that extend the system as described in section 2. These techniques are designed to make the motion of the virtual drummer appear to be more ‘natural’ and ‘realistic’. One should keep in mind, however, that although some general rules can be followed, human drummers can play the same piece of music in many different ways. Every drummer will have his/her own playing style, with differences in terms of

- the parts of the drum kit: how many and what type of cymbals, toms etc. are used?  
Is there one bass drum with a single pedal, one bass drum with a double pedal, or two bass drums with two separate pedals?
- the setup of the drum kit: ‘normal’ (with the hi-hat on the left side and the lowest tom on the right side, this setup is used by right-handed players) or ‘mirrored’ (for left-handed drummers)? Where are the cymbals placed?
- The hand patterns used on a certain ‘roll’: LLRR, LRLR, LRRL, etc.
- ‘grip’, the way of holding the drum sticks: either ‘matched’<sup>3</sup> or ‘traditional’<sup>4</sup>?
- the way of striking the drums: are the palms of the hands kept vertical or more horizontal?

Almost all of these parameters can easily be saved in a drummer ‘profile’, which also incorporates the parameters from the drum kit model, and the ‘UP’/‘DOWN’ poses for all event types.

<sup>2</sup>Note that this will sometimes cause a sudden discontinuity in the angle, when a new vibration overrides an existing one at a moment that the angle was not 0. In practice, however, this effect is hardly noticed; probably because the viewer’s eye already expects a sharp change in the motion of the cymbal, once it gets hit by the stick.

<sup>3</sup>in matched grip, both hands hold their stick between thumb and index finger

<sup>4</sup>the traditional grip is often used by jazz drummers. The right hand grip (for right-handed players) is the same as with matched grip, while the left hand holds the stick between thumb and index finger and also between ring and middle finger

### 3.1 Event Distribution

The hand assignment algorithm described in section 2.6 is easy to model and gives satisfactory results in most situations. However, a number of problems arise:

- when two simultaneous events have the same default hand (for example, MIDTOM and LOWTOM), the original algorithm would remove one of the events from the list, even when the other hand could have played that event.
- in some cases, the arms are crossed when this is not necessary: consider for example a fast sequence HIHATOPEN-RIDE-HIHATOPEN. Both RIDE and HIHATOPEN have ‘right’ as default hand, and the hand alternation algorithm will assign the RIDE event to the left hand. Most drummers, however, would in this case prefer to play the HIHATOPEN with the left hand and the RIDE with the right hand.

Our second algorithm, that solves these shortcomings, uses default hand assignments for all possible *pairs* of events. For example, we can define that whenever RIDE and HIHATOPEN are played together, the RIDE is played with the right hand and the HIHAT with the left. We should keep some flexibility, as these constraints do not have to be equally strong for all pairs: for example, SNARE+CRASH can be played as left-right just as easy as right-left.

The drum kit model is extended with a function  $pair(eventType, eventType)$ , that returns a floating-point value in the range [-1..1]. The semantics of this value are as follows:

- 1  $\equiv$  strictly left-right
- 0  $\equiv$  don’t care
- 1  $\equiv$  strictly right-left

The improved hand assignment algorithm uses just the  $pair(a, b)$  function for simultaneous events. For events  $[e1, e2]$  with a time gap  $\Delta t$  greater than zero, the default hand values are taken into account as well.

For a pair  $[e1, e2]$  the hand assignment values ( $hand(e1), hand(e2)$ ) are calculated in the following way:

$$\begin{aligned}\Delta t &= Time(e2) - Time(e1) \\ hand(e1) &= \rho^{\Delta t} pair(e1, e2) + (1 - \rho^{\Delta t}) \\ &\quad \times defaultHand(e1) \\ hand(e2) &= \rho^{\Delta t} (-pair(e1, e2)) + (1 - \rho^{\Delta t}) \\ &\quad \times defaultHand(e2)\end{aligned}$$

The decreasing exponential function  $\rho^{\Delta t}$  ( $0 < \rho < 1$ ) ensures that the default hand values are taken more into account when there is more time between  $e1$  and  $e2$ , at the same time lowering the influence of the pair-wise hand preference.

For each event with index  $I$  in the event list, a hand assignment value is calculated twice: in the pair  $[event(I-1), event(I)]$  and in the pair  $[event(I), event(I+1)]$ . Afterwards, these two values are averaged to yield the final hand assignment value for event( $I$ ).

**Shortest path methods.** A third possible solution to the hand assignment problem might be found in shortest-path methods, as used in [5, 7]. These methods consist of the following steps:

1. generate all possible solutions
2. assign a distance value to each solution (e.g. based on distances between drums, penalties for using a certain hand for a certain event type, etcetera)
3. take the solution with the lowest distance value.

Problems with this approach lie in the design of a good distance function, and in the large number of possible solutions<sup>5</sup>. We have not (yet) implemented a shortest-path algorithm in our system.

### 3.2 Key Frame Generation

**Drum Elasticity.** In a real drum kit, one can observe that some drums or cymbals are more ‘elastic’ than others, i.e. the drum stick ‘bounces’ more on one object than on another. Besides the object itself, the elasticity is also dependent on the way of playing: the stick will bounce back more on the hi-hat when it is played ‘closed’ than when it is played ‘open’.

To simulate this phenomenon, we extend the drum kit model with an elasticity parameter  $el_{eventType}$  in the range [0..1] for each animation event type. The value of  $el_{eventType}$  determines how far the drum stick should bounce back to its initial position after contact. In this definition, 0 means “no elasticity” while 1 corresponds to “maximum elasticity”.

**Note Velocities.** In the basic algorithm (see section 2.5), we did not take the velocities  $vel_{event}$  of the animation events into account. Using different animations for different velocities results in a more ‘natural’ animation: the ‘UP’ position should be closer to the drum surface for softer notes, and further away in the case of loud notes.

The note velocities and the elasticity values are used to calculate the key frames (for each joint) for an event  $e$ , by interpolating between the ‘UP’ and the ‘DOWN’ poses for the corresponding event type:

$$\begin{aligned}
 e_{before} &= DOWN_{eventType} + vel_{event} \times diff \\
 e_{contact} &= DOWN_{eventType} \\
 e_{after} &= DOWN_{eventType} + vel_{event} \times el_{eventType} \\
 &\quad \times diff \\
 diff &= UP_{eventType} - DOWN_{eventType}
 \end{aligned}$$

For the drum kit animation, the note velocities can easily be used to scale the amplitudes of the cymbal vibrations.

**Extra avatar animation.** In this section, a number of extensions are discussed that animate parts of the avatar that were not animated at all in the basic system. This helps a great deal to make the avatar look ‘alive’.

<sup>5</sup>This is of exponential complexity, as  $n$  events can be distributed over the 2 hands in  $2^n$  ways

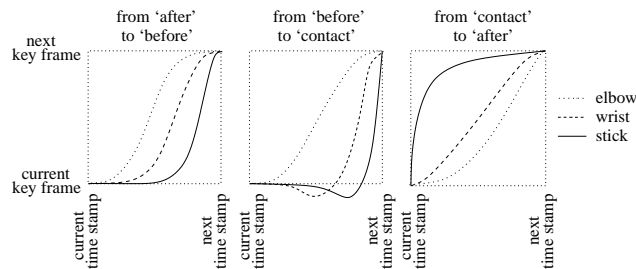
*The head:* The head of the avatar is animated, to create the effect that the avatar ‘follows’ his hands with his eyes. First, we create poses for the head: one for each event type that is supported by the hands. These poses rotate the head so that the eyes are pointed at the associated drum / cymbal. If we then use all events that are played by e.g. the right hand to create a key frame time line, the head appears to ‘follow’ this hand.

*The neck:* The neck joint is used to make the avatar nod with his head on the beat: ‘UP’ and ‘DOWN’ poses are defined for the neck joint, and for each ‘beat’ note a stroke is created. We have used the SNARE event on the left hand as an approximation of beat notes.

Finding the ‘real’ beat in a MIDI file is far from trivial, and many other researchers have addressed this problem [1, 3]. Our system could very well be integrated with an intelligent beat detector to create even better looking behaviour.

**Key Frame Interpolation.** After the basic key frames are set, the motion is fine-tuned by inserting extra key frames according to interpolation ‘scripts’. Different interpolation scripts can be used between specific pairs of key frame types (e.g. between ‘before’ and ‘contact’). Of course, scripts can also vary between different joints.

Figure 10 shows example scripts shown in create rather convincing results, because the stick moves slightly ‘behind’ the hand, and the hand moves ‘behind’ the elbow, resulting in a whip-like motion. These interpolation scripts are derived by observing the motion of a human drummer.



**Fig. 10.** example interpolation scripts for the elbow and the wrist and stick joints

### 3.3 Implementation Notes

The Java3D API is used for the implementation, because it is platform-independent and supports a wide range of geometry file formats. Moreover, our virtual theatre [9] is currently being ported from VRML to Java3D.

The SMF format (Standard MIDI File) is used as intermediate file format between the percussion recogniser and the animation generator. A great advantage of using the SMF is that it allows us to use MIDI files (which are widely available on the WWW) to test the animation generator independent from the percussion recognizer.

For the synchronisation of the animation and the sound, a separate thread is used, which looks up the current audio position and adjusts the start time of the animation accordingly.

## 4 Conclusions

We have presented some algorithms that allow for the automatic generation of 3D animated musicians, based on an abstract representation of the musical part that has to be played. Using this, we have implemented a virtual drummer, but the system could easily be applied to other kinds of instruments.

The algorithms that create the realistic animation of the limbs and sticks *before* the contact moments, as well as the improved hand assignment algorithm (which makes use of the time intervals between subsequent events) can only be used for pre-calculated animations.

Instead of motion capture or inverse kinematics, a simple GUI-based pose editor is used. This proved to be very useful, because there are only a small number of poses, and they have to be set only once for a new drum kit configuration.

The animation results can be viewed at our web site: <http://wwwhome.cs.utwente.nl/~kragtwij/science/>

## References

1. P. Desain and H. Honing. Can music cognition benefit from computer music research? from foot tapper systems to beat induction models. In *Proceedings of the ICMPC*, pages 397–398, Liege: ESCOM, 1994.
2. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Company, second. edition, 1990.
3. M. Goto and Y. Muraoka. Music understanding at the beat level: Real-time beat tracking for audio signals. In *Working Notes of the IJCAI-95 Workshop on Computational Auditory Scene Analysis*, pages 68–75, Montreal, Aug. 1995.
4. The Java3D API. <http://java.sun.com/products/java-media/3D/>.
5. J. Kim. Computer animation of pianist’s hand. In *Eurographics ’99 Short Papers and Demos*, pages 117–120, Milan, 1999.
6. M. Kragtwijk, A. Nijholt, and J. Zwiers. An animated virtual drummer. In *Proceedings of the ICAV3D International Conference on Augmented, Virtual Environments and Three-Dimensional Imaging*, pages 319–322, Mykonos, Greece, 2001.
7. T. Lokki, J. Hiipakka, R. Hänninen, T. Ilmonen, L. Savioja, and T. Takala. Real-time audio-visual rendering and contemporary audiovisual art. *Organised Sound*, 3(3):219–233, 1998.
8. The general midi specification. <http://www.midi.org/about-midi/gm/gm1sound.htm>.
9. A. Nijholt and J. Hulstijn. Multimodal interactions with agents in virtual worlds. In N. Kasabov, editor, *Future Directions for Intelligent Systems and Information Science*, Studies in Fuzziness and Soft Computing, chapter 8, pages 148–173. Physica-Verlag, 2000.
10. Web3d consortium. <http://www.vrml.org/>.
11. V. B. Zordan and J. K. Hodgins. Tracking and modifying upper-body human motion data with dynamic simulation. In *Computer Animation and Simulation ’99*. Springer-Verlag Wien, 1999.