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: {kragtwij,anijholt,zwiers}@cs.utwente.nl

ABSTRACT

We describe a system for the automatic generation of a 3D animation of a drummer playing along with a given piece of music. The input, consisting of a sound wave, is analysed to determine which drums are struck at what moments. The Standard MIDI File format is used to store the recognised notes. From this higher-level description of the music, the animation is generated. 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.

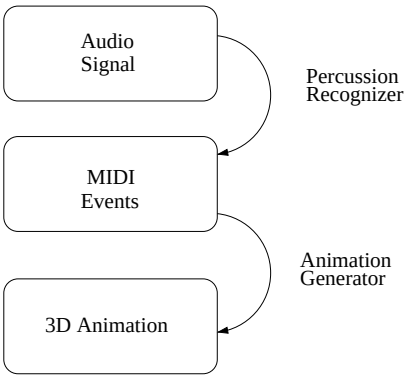

Figure 1: An overview of the system

As figure 2 shows, the total task can be separated into two independent subtasks:

- An analysis of the sound signal and transcription of the percussion part. The system has to determine which drums are hit, at what moments in time. Concentrating on percussion sounds has certain advantages and disadvantages; this is further discussed in section 2.
- The creation of the the movements of a 3D avatar playing on a drum kit. A more detailed explanation on this part is given in section 3 and 4.

2. THE PERCUSSION RECOGNISER

This part of the system is responsible for 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. These recognised notes are stored as MIDI events.

Many attempts in the field of musical instrument recognition concentrate on pitched

sounds [1]. As explained in [9], this is a rather different task than recognising percussive sounds, which have a sharp attack, short duration, and no clearly defined pitch. As shown in [9], individual, monophonic samples of drums and cymbals can be classified very well. In this approach, a few frames of the spectrum, measured from the onset of the sounds, were matched against a database of spectral templates.

In our highly polyphonic, 'real-life' situation, however, the input signal may contain many percussive sounds played simultaneously, and non-percussive instruments (such as guitar and vocals) may be mixed through the signal as well. Therefore, special techniques are needed to separate the percussive sounds from the other sounds. Other researchers have already tried to solve the same problem [13, 14]: Sillanpää et. al. subtract harmonic components from the input signal to filter out non-percussive sounds. Furthermore, they stress the importance of top-down processing: using temporal predictions to recognise soft sounds that are partially masked by louder sounds [14]. Puckette's Pure Data program has an object called *bonk* that uses the difference between subsequent short-time spectra, to determine whether a new 'attack' has occurred.

We are still developing this part of the system, therefore we cannot yet present a final solution of this problem. We plan to solve the problem of polyphony by adding examples that consist of multiple sounds played together to the collection of spectral templates. For example, a bass drum, snare drum and hihat played together. For an off-line situation, where the complete input signal is already known, we plan to apply clustering methods on all fragments of the signal that contain a strong attack. This is based on our hypothesis that specific drum sounds will sound very similar throughout a piece of music. This is especially plausible for commercial recordings, and/or in the case that the music contains sampled drum sounds.

3. BASIC ALGORITHMS

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 4, more advanced techniques (that give better results) will be explained.

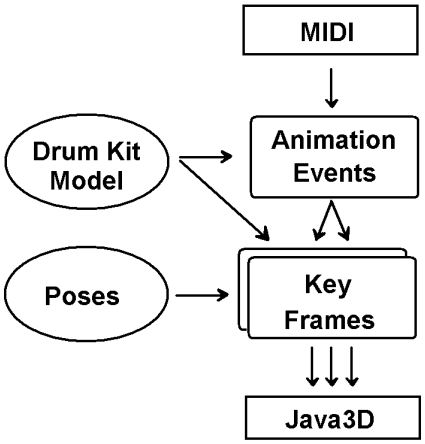

Figure 2: An overview of the system

3.1. Overview of the system

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' [4] that can directly be used by the graphical subsystem to animate objects in the scene.

Our implementation uses the Java3D engine for visualisation purposes [7]; the geometry of the 3D objects we have used is has been created using Virtual Reality Modeling Language (VRML, [15]).

3.2. '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 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.

3.3. 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, etcetera, 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 [8]. The drum kit could be viewed in between these two extreme examples: up to four sounds can be started simultaneously.

3.4. Drum Kit Model

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

3.4.1. Event Types

The General MIDI standard [11] 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 ("Low floor tom", "High floor tom", "Low tom", "Low-mid tom", "Hi-mid tom", "High tom"), 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 'drum event types' in the drum kit model . The MIDI events from the input file can then be mapped onto these drum event types.

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

The drum event types do not neccesarily 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'. A good example of this are the 'RIDE' and 'RIDEBELL' events: both are played on the ride cymbal, but we speak of a *ride bell* (or *cup bell*) when the stick hits the small 'cup' at the center of the cymbal (this gives a bell-like sound, hence the name).

3.4.2. 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 3.6.

3.5. MIDI Parsing

First, the list of MIDI events is transformed into a list of `DrumEvent` objects according to the mapping defined in the drum kit model (see section 3.4). The class `DrumEvent` is an extension of the `AbstractEvent` class (see appendix **??**). Besides having a type code, a `DrumEvent` has an associated velocity vel_{event} in the range [0..1].

Secondly, the list of `DrumEvent` events is parsed to remove double events [1] and distribute the events over the different animated objects. Objects in the scene respond to a subset of drum 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 `DrumEvent`s 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 3.6.

3.6. Event Distribution

Drum 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.

3.6.1. Hand assignment

The first 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.

[1] some MIDI files that we 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.

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 $defaultHand_{eventType}$ 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 3.1. It consists of two phases:

1. default hand assignment

2. hand alternation

Algorithm 3.1 A simple algorithm for event distribution

```
iterate over all events e:
    hand(e) := preferredHand(type(e))
iterate over all triplets of subse-
quent 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))
```

3.7. Pose Creation

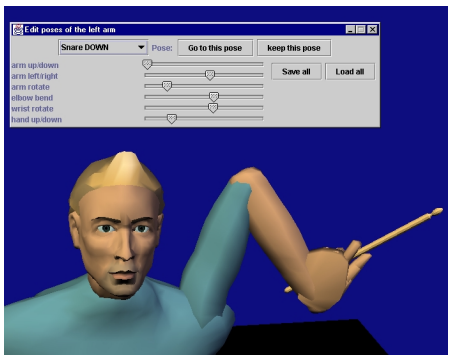

Figure 3: The graphical poser interface, applied to the left arm

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.

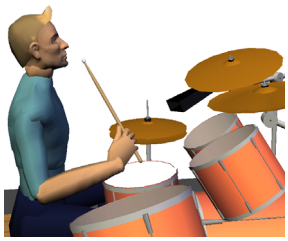

Figure 4: 'MID TOM UP'

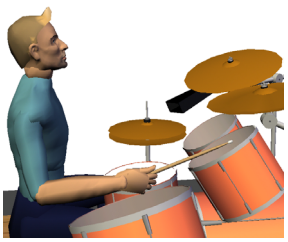

Figure 5: 'MID TOM DOWN'

For each limb, two posews should be specified for each drum 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.

3.7.1. Motivation

We have chosen for manually setting the poses through a GUI interface, instead of using motion capture [16] 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 the setting 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.

3.7.2. Implementation

In the object source files, we have to define the parameters of the object, such as the degrees of freedom and the corresponding rotation / translation axis. For example:

- arm:
  - the shoulder can rotate around its local X,Y and Z axis;
  - the elbow can rotate around its local X and Y axis, to make the lower arm twist and the elbow bend, respectively;
  - the wrist can rotate around its local X and Z axis
- hi-hat:
  - the pedal can rotate around its local Z axis
  - the upper part (the stick to which the upper cymbal is attached) can be translated along the Y axis.

3.8. Key Frame Generation

In this section, the transformation from 'abstract events' (`DrumEvents`) to 'concrete events' (key frames) is discussed. Because a different approach is used for the limbs and the cymbals, they are discussed separately:

3.8.1. Avatar Animation

The poses that were created with the GUI interface (see section 3.7) are used to create key frames for the animation of the limbs. For each arm and leg, its abstract time line (that contains only drum events that should be played by that arm/leg) is parsed in the correct temporal order. For each abstract animation event e, a `Stroke` is added to the animation time line. A `Stroke` consists of three 'concrete animation events' (i.e. key frames): $(e_{before}, e_{contact}, e_{after})$.

The parameter $delta$ is a constant that 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.

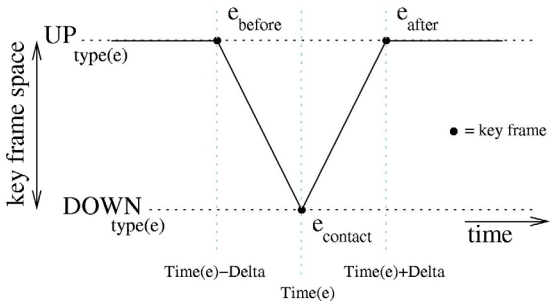

Figure 6: A basic `Stroke`, consisting of key frames 'before', 'contact' and 'after'

If the time gap between subsequent animation events $e1$ and $e2$ is less then $delta$, 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 $eNew$: 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.

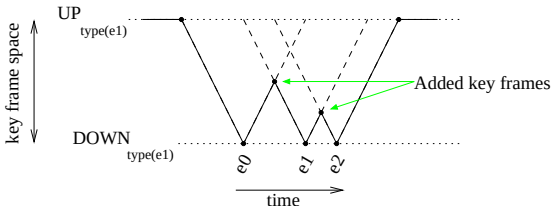

Figure 7: New key frames in the case of overlapping events of the same event type

- 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 $e2_{before}$.

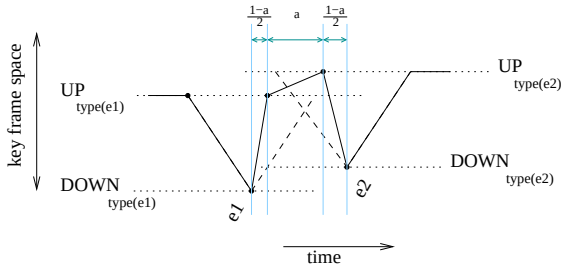

Figure 8: New key frames in the case of overlapping events of different event types

3.8.2. Drum Kit Animation

The event list, containing all `DrumEvents` from every limb is used to animate the 3D drum kit.

3.8.2.1. Pedals

The bass pedal and the hi-hat are animated through the same kind of **Stroke** objects as we use for the arms and legs. 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.

3.8.2.2. Cymbals

For the animation of the cymbals we use **Vibration** objects, that contain a number of key frames 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 exponentionaly decaying sinusoid:

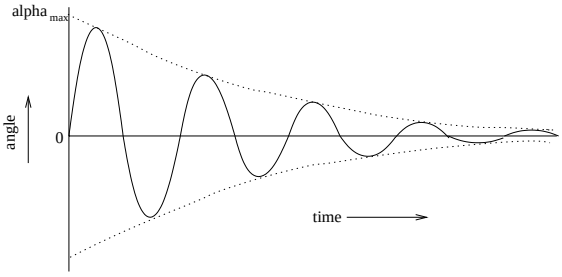

Figure 9: $angle(t)$

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

In the above equation,

- α_{max} represents the maximum angle
- η is the damping factor of the vibration ($0 < \eta < 1$): low values for η result in a fast decay.
- β determines the speed of the vibration: a higher value for β 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 `Vibration`, the remaining events (key frames) are deleted[2].

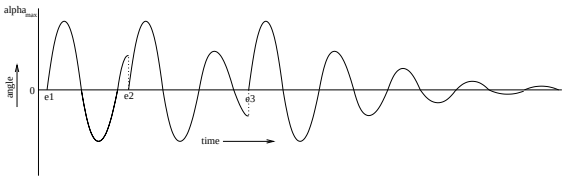

Figure 10: overlapping vibrations for events [e1,e2,e3]

4. IMPROVEMENTS

In this section, some advanced techniques will be discussed that extend the system as described in section 3. 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,

[2]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.

that although some general rules can be followed, there is no 'perfect' solution: different drummers will have their own playing style. Differences may lie in

- 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 seperate 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' [3] or 'traditional' [4]?

- the way of striking the drums: are the palms of the hands kept vertical or more horizontal?

4.1. Event Distribution

In our basic algorithm (see section 3.6), there was a maximum of two simultaneous events that were played by the hands. In this section, we will show how this constraint can be releaved by using the hi-hat pedal in specific situations. First, however, another constraint on the contents of the event list will be discussed.

4.2. Simultaneous hi-hat events

Our input list of MIDI events is not bound by any 'real-world' constraints, and may therefore contain any number of simultaneous events, even when this would be impossible to play on a real drum kit. Consider the set of possible hi-hat events {HIHATCLOSED, HIHATOPEN, HIHATPEDAL} [5] : only one of them can be played at a time.

We must therefore ensure that the event list that is used to create the animation contains no more than one hi-hat event at each moment in time. This is taken care of in the MIDI parsing stage: whenever two or three hi-hat events have the same time stamp, one will be kept and the others are discarded. Which event is kept and which ones are removed is a rather arbitrary choice.

4.2.1. The hi-hat pedal as a substitute

Human drummers often use the hi-hat pedal to play the hi-hat sounds when they have to play two other sounds on the same time as well. This is implemented in our system in the following way: If three or more `DrumEvents` have the same time stamp (not counting BASS events), and one of them is a HIHATOPEN or HIHATCLOSED event [6] , this event is replaced by a HIHATPEDAL event with the same time stamp and velocity.

4.2.2. Hand assignment

The hand assignment algorithm described in section 3.6 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.

[3] in matched grip, both hands hold their stick between thumb and index finger

[4] 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

[5] This is a strongly simplified view of reality, as human drummers are able to play much more different hi-hat sounds than these three. For example, playing with the hi-hat cymbals almost closed sounds entirely different than both HIHATOPEN and HIHATCLOSED. However, the three event types that we consider in our model are the only three that are included in the General MIDI specification, and are used in most situations.

[6] Note that there can only be one such event, because of the filtering as explained in section 4.2.

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:

$$\begin{aligned} -1 &\equiv \text{strictly left-right} \\ 0 &\equiv \text{don't care} \\ 1 &\equiv \text{strictly right-left} \end{aligned}$$

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

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).

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

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

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.

4.2.3. Shortest path methods

A third possible solution to the hand assignment problem might be found in shortest-path methods, as used in [8, 10]. 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[7]. We have not (yet) implemented a shortest-path algorithm in our system.

4.3. Key Frame Generation

4.3.1. 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' then 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 drum 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". The elasticity values are now used in the following way: for each stroke, the TR_{after} key frame is interpolated between the 'UP' and the 'DOWN' pose:

$$\begin{aligned} TR_{before} =& TR_{UP} \\ TR_{contact} =& TR_{DOWN} \\ TR_{after} =& TR_{DOWN} + el_{eventType} \\ & \times (TR_{UP} - TR_{DOWN}) \end{aligned}$$

From this, one can easily deduce that

$$\begin{aligned} el_{eventType} = 0 &\rightarrow TR_{after} = TR_{contact} \\ el_{eventType} = 1 &\rightarrow TR_{after} = TR_{UP} \end{aligned}$$

[7] This is of exponential complexity, as n events can be distributed over the 2 hands in 2^n ways

4.3.2. Note Velocities

In the basic algorithm (see section 3.5), we did not take the velocities vel_{event} of the `DrumEvents` into account. It would of course be more convincing to use different animations for different velocities. use different animations for different velocities will result in a more 'natural' behavior: the 'UP' position should be closer to the drum surface for softer notes, and further away in the case of loud notes. The key frames $[TR_{before}, TR_{contact}, TR_{after}]$ that make up a `Stroke` can therefore be defined as follows (see also figure 11):

$$
\begin{aligned}
TR_{before} &= TR_{DOWN} + vel_{event} \times diff \\
TR_{contact} &= TR_{DOWN} \\
TR_{after} &= TR_{DOWN} + vel_{event} \times el_{eventType} \\
&\quad \times diff \\
diff &= TR_{UP} - TR_{DOWN}
\end{aligned}
$$

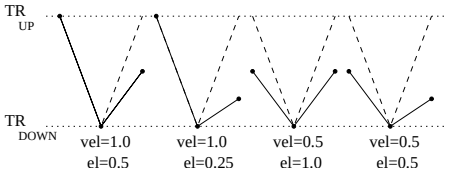

Figure 11: The effect of different velocity and elasticity values.

4.3.3. Extra avatar animation

In this section, a number of extensions are discussed that animate parts of the avatar that were not animated at al in the basic system. This helps a great deal to make the avatar look 'alive'.

4.3.3.1. 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.

4.3.3.2. 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 approximisation of beat notes.

Finding the 'real' beat in a MIDI file is far from trivial, and many other researchers have addressed this problem [3, 2, 5, 6]. Our system could very well be integrated with an intelligent beat detector to create even better looking behaviour.

4.3.4. Key Frame Interpolation

After the basic key frames are set, the motion is fine-tuned by inserting extra key frames accordiapplying a different interpolation script between certain key frame types ('before' /'contact' /'after'). These scripts can also be different for each joints.

The example scripts shown in figure 12 create rather convincing results, because the stick moves slightly 'behind' the hand, giving in a whip-like motion. These interpolation scripts are derived by observing the motion of a human drummer.

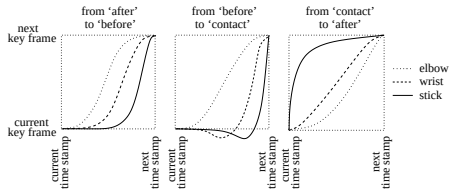

Figure 12: example interpolation scripts for the elbow and the wrist and stick joints

4.4. 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, the our virtual theatre [12] 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 seperate thread is used, which looks up the current audio position and adjusts the start time of the animation accordingly.

5. CONCLUSION

We have chosen for a GUI-based pose editor and script-based key frame interpolation. A screenshot is shown in figure 3. This proves to be a very flexible solution, since there are only a small number of poses, and they have to be set only once for a new drum kit configuration. The system could be extended with motion capturing, dynamics and inverse kinematics to create even more realistic behaviour, but at the cost of losing simplicity and flexibility. The interpolation scripts create natural motion, while the hand assignment algorithm ensures the arms will not cross. Motion capture would require the setup of the virtual drum kit to exactly match the setup of the 'real' kit, so changes cannot easily be made.

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

6. REFERENCES

[1] A. T. Cemgil and F. Gurgen. Classification of musical instrument sounds using neural networks. Technical report, Department of Computer Engineering, Bogazii University, Istanbul Turkey, 1997.

[2] P. Desain and H. Honing. Quantization of musical time: A connectionist approach. *Computer Music Journal*, 13(3):56–66, 1989.

[3] 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.

[4] 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.

[5] 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.

[6] M. Goto and Y. Muraoka. A real-time beat tracking system for audio signals. In *Proceedings of the International Computer Music Conference*, pages 171–174, Sept. 1995.

[7] The Java3D API. http://java.sun.com/-products/java-media/3D/.

[8] J. Kim. Computer animation of pianist's hand. In *Eurographics '99 Short Papers and Demos*, pages 117–120, Milan, 1999.

[9] M. Kragtwijk. Recognition of percussive sounds using evolving fuzzy neural networks. Technical report, University of Otago, Dunedin, New Zealand, July 2000. Report of a practical assignment.

[10] T. Lokki, J. Hiipakka, R. Hänninen, T. Ilmonen, L. Savioja, and T. Takala. Real-time audiovisual rendering and contemporary audiovisual art. *Organised Sound*, 3(3):219–233, 1998.

[11] The general midi specification. http://www.midi.org/about-midi-/gm/gm1sound.htm.

[12] 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.

[13] M. Puckette. Pure data: Recent progress. In *Proceedings of the Third Intercollege Computer Music Festival*, pages 1–4, Tokyo, 1997.

[14] J. Sillanpää et al. Recognition of acoustic noise mixtures by combined bottom-up and top-down processing. In *Proceedings of the European Signal Processing Conference EUSIPCO*, 2000.

[15] Web3d consortium. http://www.vrml.org/.

[16] 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.