DECENTRALIZING MUSIC, ITS PERFORMANCE, AND PROCESSING

Axel Berndt

Department of Simulation and Graphics
Otto-von-Guericke University, Magdeburg, Germany
aberndt@isg.cs.uni-magdeburg.de

ABSTRACT

A musical piece is not one single entity. It is a collaboration of multiple simultaneously acting entities—its parts. Each one is subordinate to local and global compositional and performative structures. Based on this view of music we developed an agent-based music system approach, implemented in the *MuSIG* engine, which turned out to be a versatile and intuitive basis for a variety of tasks. Its capability to render expressive performances made it a valuable tool to investigate historically informed practices and for music production. Its ability to transition different performance styles seamlessly and in realtime, even with varying musical material, is most interesting for entertainment purposes such as game scoring.

This paper reports of the *MuSIG* engine's architecture, development, and our experiences. It further describes how a core problem of musical nonlinearity, namely the generation of reasonable transitions, can be tackled through the decentralized approach.

1. BACKGROUND

Imagine the following situation. An orchestra, sitting in a recording studio, plays music for a film. The conductor is the only one who sees the picture sequence and directs accordingly. As the scene changes a different character of expression is required, a faster tempo, louder dynamics, aggressive accentuation. He changes his conduction, the musicians react and adapt their performance. Now imagine, this is not for a film but a computer game. The performance takes place during the course of the game. The orchestra is represented by synthesizers and samplers, the conductor by a music engine. Although the music is precomposed it is not fixed in its arrangement, performance, and character of expression. It is able to organically follow the progress of the interactive scene in the way film music does.

The *MuSIG* engine, of which this paper reports, has been developed as such a module for game engines. It implements techniques for the expressive performance of MIDI-based music data and their flexible adaptation [4]. In principle, all musical material is precomposed but automatically rearranged, edited, adapted to the demands of the interactive context. The biggest challenge therefore is to find musically adequate adaptation methods. A well-known situation is, that algorithmic intervention is perceived as inadequate interference which injures musical coherency. This is not originated from the algorithms in the first place.

It already begins with the way musical data is formally represented. The waveform, for instance, is totally ineligible to convey information on musical structure, polyphony or even single notes—think of the difficulties to find clear onsets in a wave signal, not to mention tone endings. Other representations, like the MIDI format [11] or MusicXML [12], allow to distinguish notes, at least, but still reveal nothing about the aesthetic structures they constitute or the way they should be performed. How to make musically meaningful decisions or processing on this basis? For the *MuSIG* engine two basic design decisions were authoritative:

1. a musical piece is not one single entity but an assembly of multiple coordinated entities (parts/channels),

2. the MIDI data are complemented by additional structural and performative information.

This led to an agent-based approach to music representation, performance, and processing. The resulting architecture turned out to be useful not just for audiovisual media scoring. It constitutes a valuable tool in the context of a musicologically inspired project to study historically informed performance practices through an analysis-by-synthesis approach. The built-in performance features were extended, accordingly, to allow for a more flexible and adequate shaping of human-like performances. These features encapsulate plenty of low-level work and allow fast and flexible creation of expressive performances. The engine became valuable as a nonautonomous performance system and music production tool even beyond musicological applications.

This paper details key aspects of the *MuSIG* engine's conceptual approach and implementation that prove beneficial to its versatility and supportive to the development of musically meaningful processing techniques. Section 2 explains the aims which led to our conceptual design decisions. The engine's architecture is introduced in Section 3. It constitutes the interface to an agent-based music architecture which is detailed in Section 4. The engine's approach to musical nonlinearity for interactive media scoring is explained in Section 5. Subsequent to a discussion in Section 6, the paper is concluded in Section 7.

2. AIMS

The purpose of music engines in games is to handle musical data and play them back. Some also include arrangement techniques to adapt musical expression to the interactive context [1, 2, 9]. To seize further potential beyond the arrangement of fixed musical snippets, some systems allow to render different expressive performances of a compositional material [10] and the generation [15, 8] or recombination [17] of musical data.

In case of the *MuSIG* engine the musical material, whether composed or automatically generated, is given as MIDI data, likewise several performance descriptions that can be rendered into expressive MIDI sequences. The engine implements techniques to transition different such performance styles, orchestrations, and even different versions of a compositional material (e.g., a plain version, an ornate, and several others with differing melodic or rhythmic properties).

To evade conflicts with the performative and compositional structure, these transitions are grounded on the principle to vary as few and as little as necessary for a smooth musical connection. To further strengthen musical coherence and conclusiveness, all transitional changes are integrated into the structural context. This means that any changes are aligned with meta-structural features like phrase borders or emphatic stress-points.

All this runs interactively while the music is playing back. This is facilitated by an agent-based approach. An agent represents a musical part or MIDI channel. Each agent acts autonomously and coordinates itself with the other agents through globally accessible metadata.

In fact, the outer part of the engine implements no global performance rendering or processing techniques. It rather acts as an interface to the agents, allows to trigger their built-in processing functionalities, and to control their realtime performance. The engine's application programming interface (API) conceals all this from the application developer and provides intuitive high-level control comparable to that of a CD-player. No specialized musical knowledge is needed to handle the engine controls.

3. ENGINE ARCHITECTURE

The *MuSIG* engine basically consists of three parts: the API, a list of music objects, and the Player to play them back. Figure 1 visualizes the engine's architecture. This Section details API and Player. The structure of the music objects is described separately in Section 4.

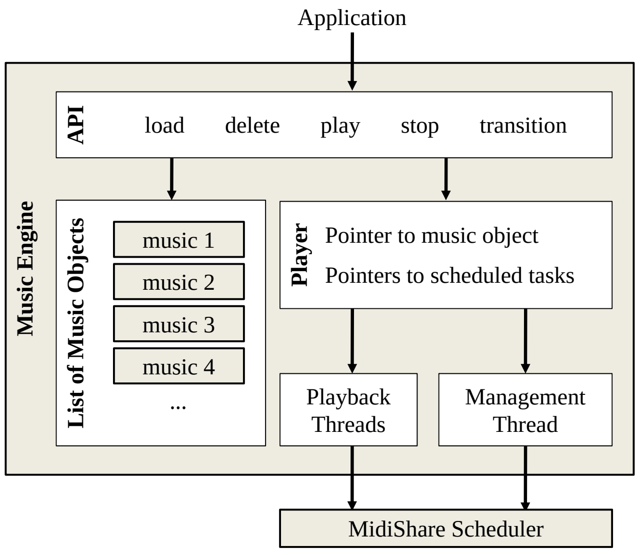

Figure 1. *MuSIG* engine's basic architecture.

All implementations were done in C/C++. *MidiShare* was used as MIDI API [7]. It furthermore takes over the realtime event scheduling. The *TinyXML* library was applied for XML processing [13].

To keep the engine controls as simple as possible, the API offers just a hand full of very high-level function calls. Musical data can be loaded into the music objects list and deleted from it. Those which are present in the list are available for playback. Starting and stopping playback are simple function calls, likewise the transitioning to different performance styles or musical data.

The play and transition functions pass a reference (more precisely a C pointer) through to the Player, which points at the respective item in the music objects list. Further parameters of these functions indicate the performance style, in which to render the raw MIDI data before playing them back, and the playback starting position or maximal transition duration, respectively. This is, concisely said, all what the application programmer has to handle. Everything else runs under the hood.

Even the next stage, the Player, implements no processing functionalities. These are provided by the agents (channel objects, see Section 4) which are contained in the music objects. The Player's task is to trigger them. It furthermore starts playback threads and the management thread. Both are functions which, once called, autonomously send their next call to the *MidiShare* realtime scheduler. The Player protocols all scheduled function calls (*typeProcess* events in *MidiShare* jargon). This makes the Player the predestined interface to interrupt, continue, and terminate this behavior.

Before starting playback, the Player calls the music object—thereby its channel objects/agents—to render the desired performance style into expressive MIDI data. Thereafter, playback is done by the agents themselves. Each one runs in its own independent realtime thread, represented by a playback task. This is not only needed to enable each agent performing with its own independent tempo. It is also the key to bypass the tempo/timing concept of the MIDI standard and introduce our own expressive musical timing concept which allows for continuous tempo transitions, asynchrony, random imprecision, and self-compensating micro deviations. A detailed description of our timing concept and its implementation is given in [3].

No events[1] from the actual MIDI sequence were sent to the scheduler. Instead, the playback function is called when such an event is due. It then triggers the agent's event processing for this and each succeeding event with the same due date. This decentralization of the event processing enables each agent to run in its own playback mode. Two such modes are provided up to now: the standard MIDI mode and the Vienna mode. The latter generates some additional controller messages for a proper playback on the *Vienna Instruments* software sampler [14]. More explanation on playback modes is given in Section 4.

The agent's event processing returns a pointer at the next due event. Its MIDI tick date is converted into a millisecond value [3]. The corresponding playback function call/*typeProcess* event is sent to the scheduler and logged in the Player's scheduled tasks list.

This list is maintained by the management task which checks regularly for dispatched entries to erase them and keep the list short. This is not just for memory efficiency, but to keep track of the playback state. If the list is empty, that means no further call is sent to the scheduler, the playback is finished. In this case the management function resets the Player and terminates. Otherwise, its next call is scheduled at a certain interval, e.g., twice a second.

Keeping track of pending and dispatched playback calls is further necessary to properly intervene in the playback. For interruption, all pending events have to be cancelled. This is, of course, not possible with events which have been dispatched already and with those which may be processed currently. Their pointers are not valid anymore. Such inconsistencies can be excluded through the management task.

Everything the management function does up to now can be done much easier, of course. The ultimate reason to introduce this mechanism is to easily ensure consistency and exclusiveness of data access. The management thread runs at the same interrupt level as the playback threads. No agent can execute event processing on its MIDI sequence when the management function is processed. Thereby, external intrusion can be processed without corrupting proper playback. This is, for instance, used in the final part of music transitions (described in Section 5). Just after all agents have created a new MIDI sequence in a separate buffer (during playback) it is then shifted into the playback buffer at once—just like *double buffering* in computer graphics [6].

[1]The term 'event' is used synonymical for MIDI message.

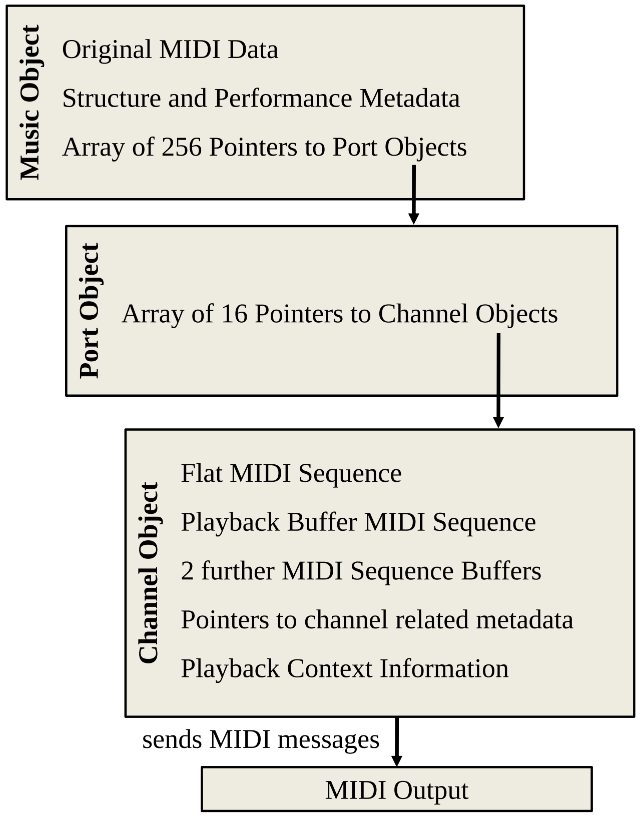
Music Object
Original MIDI Data
Structure and Performance Metadata
Array of 256 Pointers to Port Objects
Port Object
Array of 16 Pointers to Channel Objects
Channel Object
Flat MIDI Sequence
Playback Buffer MIDI Sequence
2 further MIDI Sequence Buffers
Pointers to channel related metadata
Playback Context Information
sends MIDI messages
MIDI Output

Figure 2. The *MuSIG* engine's music architecture.

4. MUSIC ARCHITECTURE

One all-embracing sequence of MIDI messages may suffice to play a musical piece back in a media player but it is insufficient for any further processing. Working on one single part would necessitate a filter operation to find relevant messages first. The implementation of channel-exclusive playback modes, which handle MIDI data differently, would be unnecessarily inconvenient. Furthermore, MIDI messages are low-level information; the problems to make musically meaningful decisions on this basis have been mentioned.

A musical piece as it is represented within the *MuSIG* engine is more than its pure MIDI data. These are organized according to an architectural concept to differentiate global and local information and to easily perform processing tasks in their most local context without filtering overhead. This is even closer to the way music is structured—not one all-comprising 'flat' entity but multiple part-wise contributing entities. MIDI data are further complemented by additional information on compositional structure and expressive performance. The general concept for music architecture is illustrated in Figure 2.

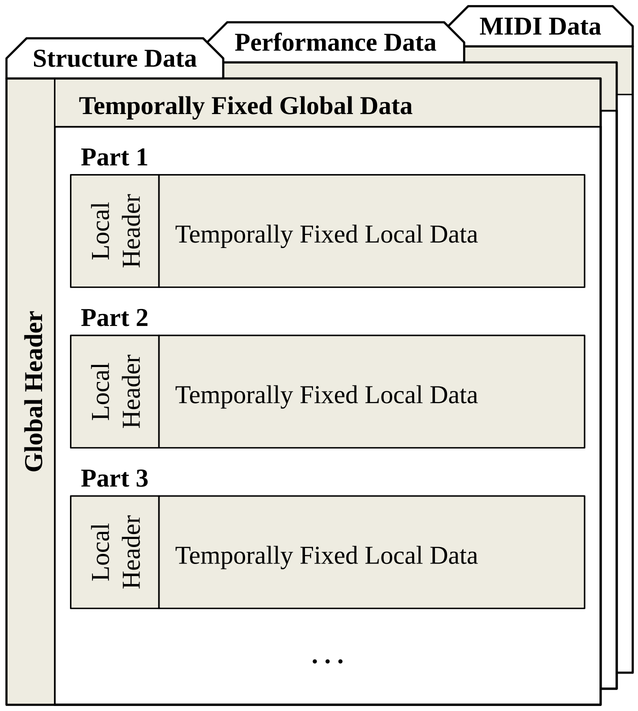
MIDI Data
Performance Data
Structure Data
Temporally Fixed Global Data
Global Header
Part 1
Local Header
Temporally Fixed Local Data
Part 2
Local Header
Temporally Fixed Local Data
Part 3
Local Header
Temporally Fixed Local Data
...

Figure 3. The organization concept of musical data.

The global class for a musical piece is the music object class. It loads a MIDI file and corresponding structural and performative metadata which are supplied by two separate XML files. Both types of metadata are structured in accordance to their area of applicability (see the schematization in Figure 3). Some are global others relate to a single part; some are general header information (e.g., an articulation style which formally defines articulation instructions), others are temporally fixed (e.g., articulation instructions applied to notes). The latter are organized in sequentially ordered lists.

Structural metadata give information on the formal segmentation of the musical piece. Three segmentation layers are differentiated (with increasing temporal granularity): the section, phrase, and figure layer. The latter compares to motifs regarding its average temporal extent. Further melodic, rhythmic, and harmonic information can be included to concretize structural properties. If these information are globally defined, they represent the formal structure of all parts. However, if any part defines its own local properties, they dominate possibly contrary global ones for this particular part. This principle applies equally to performance information.

Performance metadata are formal descriptions to convert the raw MIDI data into expressive versions. Tempo, rubato (micro-deviations), asynchrony, 'human' imprecision, dynamics, metrical emphasis, and articulation are included. Where to do a ritardando, accelerando, de-/crescendo, how strong, which notes to articulate staccato, tenuto, legato and so forth, all this is explicitly given in a formal high-level representation. It can be symbolic output of external performance generation systems ([16] gives an overview of such systems) or manually edited. Multiple performance variants can be defined. Performance transition techniques (see Section 5) allow to combine them interactively during playback.

The music object class implements methods to load the XML data, check them for consistency, repair inconsistencies, and optimize them for fast access during realtime playback. Further preprocessing splits the original MIDI sequence into several sequences which contain only the MIDI events of single MIDI ports. Instances of the port object class are created and their pointers stored in the music object's designated array. In the same way the port objects split their sequence into channel-exclusive sequences and instantiate channel objects—the agents.

The channel object first unites noteOn and corresponding noteOff events into *MidiShare typeNote* events. This is not only closer to music notation, it lowers the effort to articulate them, later on. Furthermore, each channel object has to create an endTrack event, because the original MIDI sequence usually contains only one which goes to channel 0 after splitting. The resulting MIDI sequence is stored as the basis for any further music processing, e.g. for performance rendering. It remains unchanged from now on. This means, each expressive version will be a direct derivative of this clean initial version. The music object further passes pointers through to its channel objects which enable direct access to locally relevant structure and performance metadata. This defines the agent's perception of its environment. Further references, e.g. to other agents, are easily possible but not used yet (see Section 6).

Now, all necessary information (a clean flat MIDI sequence and performance descriptions) are given to render expressive MIDI sequences. Therefore, each channel object creates a copy of its flat sequence. This is modified and complemented by further MIDI messages to integrate all performance aspects except for timing (i.e., tempo, rubato, asynchrony, random imprecision). The resulting sequence is put into the playback buffer. Thereafter, the channel object's playback context information are initialized. These comprise

– a pointer at the next due event,

– a flag to indicate whether the channel has to loop or stop when the endTrack event is reached,

– a synchronization date which is used to compensate numerically caused timing drifts,

– a pointer at the current performance style which is still needed for tick-to-millisecond conversion (timing is rendered in realtime),

– a playback mode switch, and

– MIDI controller numbers to indicate those which are designated to control the *Vienna Instruments* sampler.

The event processing differs depending on the playback mode. Two modes are currently implemented. The standard MIDI mode sends most upcoming events as they are, whereas, the Vienna mode sends additional controller messages for a proper playback on *Vienna Instruments* and filters some other inconvenient events. Additional controllers adjust, for instance, the articulation of each note.

However, both modes differ in some respects from a conventional media player's MIDI playback. First of all, only events, which are implemented by synthesizers and samplers, are sent. Markers, lyrics etc. are skipped. This reduces MIDI traffic. Furthermore, *typeNote* events need to be split back into noteOn and noteOff to play them back. The noteOn event is created and sent directly when a *typeNote* occurs, the corresponding noteOff is created and inserted into the playback buffer sequence. When it occurs, it is sent and removed from the sequence again. Another major difference to usual MIDI playback lies in the way dynamics are implemented. Note-wise dynamics are, of course, still rendered into *typeNote*/noteOn velocity attributes but for subnote-dynamics *MidiShare typePrivate* events are used. They are translated either into usual channelVolume messages or respective controller events for the Vienna mode by the playback event processing. Output events are sent to *MidiShare* for immediate output and the pointer at the succeeding due event is returned to the playback task.

Each channel object is self-responsible for its playback, playback modes can differ between channels, even their timings can be autonomous. This kind of decentralization continues in all further music processing. Each channel renders its expressive MIDI sequences on its own and when playback is manually stopped, the call is propagated through the music object and port objects down to each playing channel which itself stops all sounding notes and resets its playback information. Over the metadata pointers, which primarily refer to channel-related structure and performance data, each channel has the possibility to navigate to other channels' and global information. This is, for instance, used when all agents are performing synchronously with the same global tempo but autonomously with distinctive local mircovariations.

5. MUSICAL AND PERFORMATIVE NONLINEARITY

The *MuSIG* engine's approach to musical nonlinearity is based on precomposed material. The music is given, and through its performance metadata also several ways to perform it. One such performance style does not only describe how to shape tempo, dynamics, articulation and so on. It further includes instructions about which channels should be playing and which not. In this way, each performance style can feature a distinctive instrumentation and even exclusive musical material or combinations of it. An example: one style performs a melody on channel 2 and its accompaniment on channels 3 and 4; another style may play the same accompaniment but a different melody on channel 1. It is furthermore possible to mute notes within a channel's MIDI sequence—a kind of a 'mute' articulation. This is used, for instance, to introduce little variants and ornamentations. One performance style may include trills while another plays the same notes straight.

For interactive media scoring, it is not only feasible to play music back in these expressive performance styles but to transition them dynamically during playback. However, generating such transitions cannot be done naively. Expressive performance and orchestration have the task to emphasize chosen aspects of the compositional structure to make them clearer to the listener. Performance transitions have to go along with this. They have to be likewise embedded in the structural context in order not to conflict with it. Good transitions seem to be motivated not only by external influence but by the music itself.

In this regard, the *MuSIG* engine's decentralized concept prove to be particularly beneficial. Each agent/channel creates its own locally optimal transition. Thereby, the result keeps polyphonic properties of the musical structure and both involved performance styles (the current and the next). Nonetheless, a first task has to be done globally—the creation of a working-copy of the current performance style. Then, each agent is called to compute and apply its changes to transition to the target style. An application-given latency constraint ensures that all agents keep reasonably together. It sets the maximal duration until they have to reach the target style.

First, each agent analyzes the structural context within the latency frame to find the strongest structural border. This will be the transition's destination. Possible destinations are (with decreasing valence): the beginning of the next section, phrase or figure (endings are only needed when the agent shall finish its performance), the last barline or beat within the frame, or the end of the latency frame itself which marks the worst case if no information on the music is given, even no time signature. The weaker the destination, the more important is a sufficiently long transitional period (up to the whole latency) to implement changes as subtle as possible.

However, changes are not made to MIDI data in the first place but to the working-copy of the current performance style. Only in the end, when the transitioning performance style is ready, a new MIDI sequence is rendered. The transitioning style keeps the current style's data up to the destination date, followed by the target style's remaining performance data. Changes can be applied to the time frame from the current playback position up to the maximum latency date. Having the transition represented as a self-contained performance style opens up the flexibility to create further transitions to other styles even if the current transition is still playing (transition out of a transition). Later during playback, when the agent reaches the endTrack event, before starting over with the next loop iteration, it will switch away from the transition style to the actual target style and the corresponding expressive MIDI sequence.

The agents edit only their local performance data. Temporally fixed global data are made local, i.e., copied into the local domain, therefore. This enables the agents to handle polyphonically divergent situations (differing destination points and valences) to the local optimum. Only the global tempo map needs to be treated globally to ensure overall synchrony. All further actions depend not only on the valence of the destination point but also on the goal which an agent pursues in the transition. Three categories can be distinguished:

1. cueing a muted agent,
2. ending a performing agent,
3. transitioning a performing agent.

Furthermore, two classes of performance features have to be distinguished. **Point features** relate to single notes (e.g., articulations and metrical accentuation). **Temporally extensive features** embrace non-zero time intervals (e.g., tempo and dynamics instructions).

Transitioning point features, like articulations of single notes, over a certain period makes no sense. Instead, these features are directly switched to the target style's setting. This change takes place at the cue point of newly starting agents (1st category) and at the transition's destination point for both other categories. This direct change may be conspicuous but through its alignment with structural properties it is not perceived as inappropriate. It rather seems to acknowledge interpretational intention which has been prepared by the transition of temporally extensive features.

However, transitioning temporally extensive features is more complex. Here, it shall be described using the example of dynamics. Each performance style defines for each agent a dynamics map—a global one can be made local, as mentioned above. A dynamics map is a temporally ordered list of dynamics instructions. One such instruction I_n has the form (simplified)

$$I_n = (d_n, v_{1_n}, v_{2_n}, s_n)$$

with d_n being the MIDI tick date of the instruction. Its range is terminated by the succeeding instruction I_{n+1}. Instruction I_n describes a gradual dynamics change in the interval $[d_n, d_{n+1})$ from dynamics value v_{1_n} to v_{2_n}. Both are given either as MIDI velocity values or symbolic representations such as *f, mp, pp* etc. The shape of this change can be linear, potential or sigmoid. Even the sigmoidal characteristic does not have to be balanced but can tend along the time axis to either describe very determined crescendi and decrescendi or rather neutral ones. This property is set by the term s_n. Basically, all temporally extensive features can be formally represented in this way. This facilitates a homogeneous algorithmic approach to transition them. Therefore, the last current-style instruction in

$$[\text{currentPlaybackPosition}, \text{destinationDate})$$

is taken and designated as I_t. If none exists in this interval, one is created at playback position. It takes over the dynamics at this point in v_{1_t} and its predecessor's target value in v_{2_t}. The predecessor's corresponding attribute is set to v_{1_t}, accordingly. I_t constitutes the starting point for all transitional changes. As target instruction, designated I_{t+1},the first target-style instruction in the interval

$$[\text{destinationDate}, \text{latencyMaximumDate}]$$

is taken. Again, if none can be found, one is inserted at destination date. Both instructions, I_t and I_{t+1} embrace the destination date in the way $d_t < \text{destinationDate} \leq d_{t+1}$.

In contrast to this, for starting and ending agents (categories 1 and 2) the instruction dates are constrained to

$$d_t = \text{currentPlaybackPosition}$$

$$d_{t+1} = \text{destinationDate}.$$

For the starting case, v_{1_t} is set to 0. For the ending case, $v_{1_{t+1}}$ and $v_{2_{t+1}}$ are set to 0, and any succeeding instructions after I_{t+1} are removed from the dynamics map.

If the destination date is set on the last barline or weaker (i.e., it could not be aligned with section, phrase or figure borders), the dynamics transition will smoothly fade to the target value, i.e. $v_{2_t} = v_{1_{t+1}}$, in either category. With decreasing valence, the fading characteristics s_t are set more neutral.

In case of the strongest possible destination point (section border), v_{2_t} is set to 0 for newly starting agents, and remains unchanged for categories 2 and 3. This means, a starting agent begins directly with the new section, an ending one finishes the current section, a transitioning agent finishes the current section in the old style and begins the next section directly in the target style. Such a direct change/switch is, of course, only possible with section borders because they close a musically coherent episode and begin another.

Phrases and figures are formally less concluding. Newly starting agents can still cue in directly but the other categories need to be treated more carefully. Ending agents (category 2) imitate the gesture of ending dynamically. Therefore, attribute v_{2_t} is reduced to $5/6$ for phrase endings and $2/3$ for figure endings. This reinforces its concluding character. The term s_t is adjusted, accordingly, to mold a very determined characteristic.

For category 3 transitioning, the agent changes attribute v_{2_t} in accordance with the condition to vary as little as necessary. Therefore, the dynamics situation in the target style at d_{t+1} is analyzed. If the dynamics are continuous, v_{2_t} is set to $v_{1_{t+1}}$, resulting in a smooth transition. But in the discontinuous case, i.e. a stepwise dynamics change, v_{2_t} is set to that corner of the step which is nearest to its current value.

Finally, all agents render new MIDI sequences into separate buffers. These are then shifted into the playback buffer at once via the management function.

Through complying not only with compositional structure but also with preexistent performance features within the latency boundaries, the transition will still be reasonable even in the absence of structural metadata. However, the transition to higher-value structural borders is, of course, generally more determined and felicitous.

This approach allows even to finish playback fairly reasonable. This can be done by transitioning to a special performance style which mutes all agents and defines a slow tempo. The transition results in a decrescendo and ritardando to the determined destination point. This works most conclusive in case of very strong structural borders (e.g., section ending) and identical destination points for all agents. Even weak destination points would suffice; the resulting transition will be a *decrescendo al niente*, i.e., a complete fadeout.

6. DISCUSSION AND FUTURE PERSPECTIVES

Several conceptual and implementation aspects of the *MuSIG* engine deserve closer analysis and discussion. First of all it has to be stated that the engine, since MIDI-based, does not provide instrumental sounds. It sends MIDI messages. The system's synthesizer or sampler creates the sounds. These can differ extremely from system to system. All this is well-known. But it brings a particular problem for performance rendering. Articulations may not always be satisfying. Dynamics may work good on some systems, on others it may be unbalanced—such differences were systematically analyzed by [5]. According to the quality and liveliness of the instrumental sounds it can even be advisable to adjust the musical tempo. For the less lively sounds musicians/producers tend to choose a faster tempo to conceal these problems from the listener and to avoid boredom.

Such adaptation cannot be done automatically up to now, with one exception. The *MuSIG* engine offers two playback modes, the standard MIDI mode and the Vienna mode. The latter ensures a proper performance on the *Vienna Instruments* sampler. Many samplers and synthesizers necessitate such specialized playback modes to seize their full potential. The use of MIDI's breath controller for richer articulations and timbral variations is a typical example.

To provide homogeneous sound quality on all systems, more is needed than the engine and the music data. Sample data or even synthesizer software have to be enclosed. The engine itself requires not much hardware capacity. MIDI processing is done very fast and the musical data (MIDI and metadata) take just a few hundred kilobyte. But sound generation usually requires lots of resources. When applying it in the context of a computer game, it competes with other tasks like physics simulation and game-ai. This limits the number of instruments and their sound quality considerably.

Furthermore, although the computation of performance transitions is relatively uncritical it is nonetheless subject to soft realtime demands. Playback goes on while the transition is created. Some of the early transitional changes may be missed when the transitioning sequence is shifted into the playback buffer. This can cause a little discontinuity when dynamics or tempo are to be changed gradually. Although theoretically possible, we could not create discontinuities in experiments which were big enough to be actually audible. In fact, human performances feature a bigger variance than these discontinuities ever showed. Nonetheless, to provide more stability a certain value can be added to the current playback position from when on the transition starts. That value can be set to the average or worst computation time of past transitions. Before rendering the transition sequence, the dates of all passed changes can also be adapted relatively easy. The agent concept can also help to tackle these problems. Up to now, all channels create their transition data and MIDI sequence but switching over is done globally. This can already be done by each channel itself. Only the transition of timing properties has to be done globally.

Moreover, our agent-based approach to music performance and processing features a far bigger potential. The agents know little of their environment—up to now only the metadata. A future extension will be to connect them with other agents. This enables self-organizing behavior which is particularly interesting for realtime interaction with the performance. Imagine a group of street musicians. One of them, jostled by a passing person, gets out of synchrony. This means technically that his synchronization date changes. His colleagues notice the problem after a while and begin to adapt their synchronization date gradually, likewise does the mistaking player.

This is just one example from the performance field. Knowing the environmental context is even more important when the agents shall not only perform one piece of music but transition to a completely different one (different music object). Each channel object needs a reference to its correspondent in the target music object to have access to its melodic material. The idea is to bend the current melodic shape to get a proper connection to the target music without loosing structural (e.g., motif establishing) features. The underlying scale for the varied melody must follow from a globally determined harmonic modulation. To coordinate all channels, it is necessary to set a global target point for the transition, e.g., the last of all local target points to ensure sufficient conditions for all agents. Finally, each agent sets a jump mark to change over from the current to the target music, that means, when its corresponding target channel object takes over.

7. CONCLUSION

This paper presented an approach to a realtime music performance and arrangement engine based on a decentralized view of music. This resulted in a specific architecture of musical data and an agent-based approach to polyphonic performance and processing. It is furthermore the key to a very flexible approach to interactive musical nonlinearity.

All this has been implemented in the *MuSIG* engine. It was first conceived as a game engine module with an intuitive API. But it became obvious that the engine can also serve well as a tool and experimental environment for expressive performance research. As such it allows to create performative characteristics in a high-level description language. The realtime arrangement methods even allow to combine multiple performance styles interactively.

The *MuSIG* engine and its decentralized concept will also be a prospective basis for future developments like, e.g., compositional transition techniques.

8. REFERENCES

[1] S. Aav, "Adaptive Music System for DirectSound," Master's thesis, University of Linköping, Department of Science and Technology, Norrköping, Sweden, Dec. 2005.

[2] A. Berndt, K. Hartmann, N. Röber, and M. Masuch, "Composition and Arrangement Techniques for Music in Interactive Immersive Environments," in *Audio Mostly 2006: A Conf. on Sound in Games*. Piteå, Sweden: Interactive Institute/Sonic Studio, Oct. 2006, pp. 53–59.

[3] A. Berndt and T. Hähnel, "Expressive Musical Timing," in *Audio Mostly 2009: 4th Conf. on Interaction with Sound*. Glasgow, Scotland: Glasgow Caledonian University, Interactive Institute/Sonic Studio Piteå, Sept. 2009, pp. 9–16.

[4] A. Berndt and H. Theisel, "Adaptive Musical Expression from Automatic Realtime Orchestration and Performance," in *Interactive Digital Storytelling (ICIDS) 2008*, U. Spierling and N. Szilas, Eds. Erfurt, Germany: Springer, Nov. 2008, pp. 132–143, LNCS 5334.

[5] R. B. Dannenberg, "The Interpretation of MIDI Velocity," in *Proc. of the Int. Computer Music Conf. (ICMC)*. Tulane University, New Orleans, USA: International Computer Music Association, Nov. 2006, pp. 193–196.

[6] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, revised 2nd ed., ser. The Systems Programming Series. Addison-Wesley, July 1997.

[7] GRAME, *MidiShare Developer Documentation*, Computer Music Research Lab., France, Jan. 2006, version 1.91.

[8] M. Hoeberechts and J. Shantz, "Realtime Emotional Adaptation in Automated Composition," in *Audio Mostly 2009: 4th Conf. on Interaction with Sound*. Glasgow, Scotland: Glasgow Caledonian University, Interactive Institute/Sonic Studio Piteå, Sept. 2009, pp. 1–8.

[9] M. Z. Land and P. N. McConnell, "Method and apparatus for dynamically composing music and sound effects using a computer entertainment system," United States Patent Nr. 5,315,057, USA, May 1994, filed Nov. 1991.

[10] S. R. Livingstone, "Changing Musical Emotion through Score and Performance with a Compositional Rule System," Ph.D. dissertation, The University of Queensland, Brisbane, Australia, 2008.

[11] MIDI Manufacturers Association, *The Complete MIDI 1.0 Detailed Specification*, Nov. 2001, version 96.1 2nd edition.

[12] Recordare LLC, "MusicXML Definition," http://www.recordare.com/xml.html [last visited: Dec. 2009], Nov. 2009, version 2.0.

[13] L. Thomason, "TinyXML," http://www.grinninglizard.com/tinyxml/index.html [last visited: Dec. 2009], May 2007, version 2.5.3.

[14] Vienna Symphonic Library GmbH, "Vienna Instruments," http://vsl.co.at/ [last visited: March 2010], 2010.

[15] I. Wallis, T. Ingalls, and E. Campana, "Computer-Generating Emotional Music: The Design of an Affective Music Algorithm," in *Proc. of the 11th Int. Conf. on Digital Audio Effects (DAFx-08)*, Espoo, Finland, Sept. 2008, pp. 7–12.

[16] G. Widmer and W. Goebel, "Computational Models of Expressive Music Performance: The State of the Art," *Journal of New Music Research*, vol. 33, no. 3, pp. 203–216, Sept. 2004.

[17] R. W. Wooller and A. R. Brown, "Investigating morphing algorithms for generative music," in *Third Iteration: Third International Conference on Generative Systems in the Electronic Arts*, Melbourne, Australia, Dec. 2005.