Software Support for Interactive Multimedia Performance[1]

Roger B. Dannenberg

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
USA
email: dannenberg@cs.cmu.edu

ABSTRACT. A set of techniques have been developed and refined to support the demanding software requirements of combined interactive computer music and computer animation. The techniques include a new programming environment that supports an integration of procedural and declarative score-like descriptions of interactive real-time behavior.

1. Introduction

My work for the past few years has focused on the use of real-time computer systems to carry on compositional processes during a performance. The rationale behind this work is that composing during a performance allows a work of music to benefit from the ideas of a composer as well as those of an improviser. Of course, this could be said about many approaches ranging from aleatoric music to jazz. Almost all music contains elements of composition and improvisation. However, composition and improvisation almost always occur in just that order: composition comes first, and improvisation completes the process.

1.1. Interactive Composition

Composition by computer at the time of performance allows the composer to respond to live improvisation and vice versa, creating a sort of dialog between the composer and improviser. One could argue that composition at performance time *is* improvisation by definition, but my view is that composition is different. Composition is characterized by a careful working out of plans and ideas using relatively formal structures and relationships. Improvisation is more spontaneous and often focuses on expressiveness rather than carefully crafted structures.

Few if any humans can compose in real-time, integrating new ideas from improvisers all the while. However, humans *can* develop computer programs that embody fairly sophisticated

---

[1]Published as: Dannenberg, ‘‘Software Support for Interactive Multimedia Performance,’’ in ***Proceedings The Arts and Technology 3***, Connecticut College (April 1991), pp 148-156.

compositional strategies and which *do* run in real-time. One might call these programs *meta-compositions* because the programs themselves are composers.

1.2. Music and Animation

Another idea I have been pursuing is that of mixed media performances of synchronized computer animation and music. Because composition is being carried out by computer, it is possible, in real time, to generate graphics and music together. Composing programs tend to make compositional ideas and methods very explicit, making it possible to achieve a high degree of consistency and synchrony between sound and image.

We could continue from this point with more justification and analysis of computer programs that compose, but that is not the goal of this paper. Instead, I will take it as a given that these real-time interactive computer music and animation programs are interesting, and I will describe a number of techniques that I have developed to simplify their construction. Making these programs easier to create is important so that more composers and performers can apply their talents to this interesting new art form.

1.3. Overview

The software approach that I have taken is based on a collection of software called ‘‘The CMU MIDI Toolkit’’, or CMT. This toolkit has been described previously [6], but a number of new features have been added to make CMT more versatile. A composition, *Ritual of the Science Makers*, has been completed recently and uses many of the new features. This work will serve to motivate much of the following discussion.

The next section will describe the general structure of *Ritual*. A number of problems that arise in such a composition are discussed in Section 3 along with my solutions. Section 4 discusses some of the problems that have yet to be solved in a satisfactory manner, and this is followed by a summary and conclusions.

2. The Structure of an Interactive Composition

Ritual of the Science Makers is written for flute, violin, cello and computer system. The acoustic instruments are interfaced to the computer via microphones and pitch-to-MIDI converters so that the computer receives the pitch, volume, time and duration of each performed note. The computer outputs MIDI to a synthesizer and an effects processor, and the computer also generates graphical animations which are projected onto a large screen.

Throughout the work, incoming notes from the instruments serve to initiate various processes that typically generate long sequences of synthesized pitches lasting anywhere from a second to a minute or so. The process can be influenced by the pitch and timing of the note as well as by which instrument played the note. In addition to generating sounds, the program also generates graphical images. These images usually evolve over time in a way that is analogous (in some fashion) to the way the sounds evolve. In the opening, for example, there are crashing sounds that die away, and each crash is accompanied by a graphical plot of the amplitude of the crash. The images look something like trumpet bells, as if they are heralding the opening of the work.

There are about ten sections in *Ritual*, and each sets up a different set of responses to the

instrumental parts. Many of the transitions between sections are subtle and are played without pause, so the listener will perceive more evolutionary changes than abrupt changes in direction.

Although most of the instrumental parts could be improvised, a written score eliminates a great deal of rehearsal time. Many of the parts were in fact improvised by the composer during the creation of the piece and then notated for the performers. There is still room for interpretation, and the computer system insures that the synthesized sounds and images will be synchronized with the performance.

3. Implementation Issues

What are the problems of creating interactive multimedia composing programs? The following list is undoubtedly incomplete, but at least it describes important problems for which we have useful solutions:

- **Multitasking.** Music generation typically requires many independent threads of control, but traditional approaches add program complexity.
- **Sequencing.** Programming is essential for interactive systems, but event sequences are clumsy to express as programs.
- **Parameter adjustment.** Compiled programming languages contribute to execution speed, but make it awkward to adjust and refine many parameters.
- **Scheduling.** Long computations can interfere with the real-time responsiveness of interactive programs.
- **Testing.** Interactive systems, especially those with multiple performers, are difficult to test because testing requires real-time input and because input may not be reproducible for debugging purposes.

Let us consider these issues in greater detail.

3.1. Multitasking.

Interactive music programs often require several musical lines or simultaneous notes. For example, in *Ritual*, there are many places where a sequence of computed notes can be launched independently by each acoustic instrument. This seems to call for a separate process to compute each sequence. Processes, however, suffer from a number of problems: they require a fair amount of storage to avoid stack overflow, they require careful synchronization, especially when processes are preemptable, and processes are often computationally expensive to create, start, and stop. Debugging multiple processes is also known to be difficult and is often not supported by debuggers.

An alternative is based Doug Collinge's language for music called Moxie [5]. The idea is to implement one function, called `cause`, that calls another function with parameters at some time in the future. For example, `cause` can be used to schedule a note-off message in the future:

```
play_a_note(p)
{
    /* start note on chan 3, pitch p, vel 100: */
    midi_note(3, p, 100);
    /* after delay of 150, turn same note off: */
    cause(150, midi_note, 3, p, 0);
}
```

This is accomplished, not by simply waiting, but by putting the call into a sorted queue of waiting calls. This allows other functions to be called in the meantime. It is common to write functions that call themselves via `cause`, creating a sequence of calls to the functions, spread out in time [6].

```
lots_o_notes()
{
    play_a_note(60);            /* make a sound */
    cause(200, lots_o_notes); /* repeat every 2.00 seconds */
}
```

3.2. Sequencing.

Programming languages make it awkward to express arbitrary sequences of notes and other events. Facilities such as Moxc are excellent when sequences are to be computed according to some algorithm, but few programming languages are convenient for predetermined (composed) sequences. On the other hand, specialized score languages do not normally have adequate flexibility to represent the desired decision-making logic that will control sequences in real time.

For example, one section of *Ritual* consists of a notated sequence of synthesized drums, consisting of over 600 notes. Although these notes could be generated by a program, doing so would be very tedious in a typical programming language. However, in other parts of Ritual, notes are computed as transpositions of the pitches of live instruments. This sort of computed response cannot be expressed using a score language.

The solution I have adopted is to incorporate both a programming language (C) and a score language (Adagio) into a single system. Multiple Adagio sequences can be loaded, started, and stopped under program control. The real-time interactive portions of the program are still written using Moxc and the C programming language, but conventional music sequences are written in Adagio and ‘‘conducted’’ from within the Moxc programming environment. Furthermore, Adagio has been extended to enable C functions to be called from within the score. This allows computed event sequences to be ‘‘launched’’ from within a sequence.

To illustrate how C functions are used in sequences, here is a fragment of an Adagio score from the middle of the first section of *Ritual*. The goal here is to create some time-varying values which affect pitch and loudness choices in other parts of the program. The functions `loud_mask` and `click_mask` were written to make gradual changes in pitch and loudness over time. Calls to these functions are then entered into an Adagio score:

```
!call number(13) N0 T4000
v4 z30 n0 r
!call loud_mask(4,10,20,70,120,U2000) N0
!call click_mask(4,90,110,30,90,U2000) N150
```

The purpose of the first line is simply to print the number 13 on the computer console, indicating

that this section of the score has been reached. As indicated by `T4000`, this happens at score time 4000 (40 seconds). The number is printed by calling the C routine named `number`. The second line sends a MIDI ‘‘change to program 30’’ command (`z30`) on channel 4 (`v4`). The third line invokes the C routine `loud_mask`, which applies a crescendo tendency to notes on channel 4. The last line applies a decreasing pitch tendency to these notes as well.

The important observation here is that neither C nor Adagio is an adequate language for the tasks at hand. C is best for defining processes, and Adagio is best for defining temporal sequences. We can simplify programming by making it possible to invoke Adagio scores from C and C routines from Adagio. Another advantage of the use of Adagio is that sequences can be captured from live performances and then incorporated into the interactive composition. (This would not be the case if all sequences had to be expressed as programs.)

3.3. Parameter Adjustment.

Unlike most real-time systems, interactive music and animation systems require extensive adjustments and ‘‘tuning’’ of many parameters for artistic results, something that is not supported well by the traditional edit/compile/test cycle of program development.

In particular, *Ritual* makes use of a programmable effects processor to add echo and reverberation to synthesized sounds. Getting the right values for a large number of effects parameters is a matter of successive refinement, and each change requires careful listening and evaluation. Similar procedures are necessary to adjust MIDI volume controls to achieve a good balance among the synthesized voices.

My solution to this problem is to use the Adagio score language to enter most parameter changes. This can be done directly when the parameters are MIDI control changes. To allow changes in parameters that control computations, Adagio has been extended to allow the setting of C variables and array elements. In effect, this provides a very limited form of C interpreter. Since Adagio scores are interpreted, they can be loaded very quickly, avoiding the delays associated with recompilation.

In the following example from *Ritual*, the first line sets the C variable `section` to the value 1. The next three lines set velocity offsets for channels 4, 6, and 8 to 20:

```
!seti section 1
!setv vel_offset 4 20
!setv vel_offset 6 20
!setv vel_offset 8 20
```

Velocity offsets are not a standard feature of the CMU MIDI Toolkit, but they were added to *Ritual* so that volume levels could be adjusted easily on a per-channel basis. Putting these offsets into scores has the further advantage that the offset can be programmed to change at any time in the course of the work. Scores are used in *Ritual* to send out many MIDI control changes as well as adjust program parameters like `vel_offset`.

Another approach to this problem is to use an interpreted language or one that provides incremental compilation. Lisp, Smalltalk, and Forth fall into this category and have been used for effective music systems. Our choice of C is based on the desire for greater execution speed on small portable computer music systems. Higher level systems such as Play [4], Mabel [2],

and Max [9] achieve efficiency through pre-compiled modules that are interactively connected, but do not provide the generality of ordinary programming languages.

3.4. Scheduling

As discussed above, Moxc provides a simple approach to computing multiple independent streams of events. Routines (function calls) are scheduled by Moxc for execution at a particular time, and all routines run to completion before any other routine can be run. As long as routines take very little time to run, the overall timing accuracy is high. Unfortunately, graphics operations can be relatively slow, and some other technique is needed to insure that all events are computed at the proper time.

As described in an earlier paper [7], the solution I have adopted is to divide computations into two groups: (1) time-critical, low-computation music event generation, and (2) less critical graphical music computations. The graphical computations are off-loaded to a separate, low-priority ‘‘graphics’’ process that is preempted whenever a time-critical event becomes ready to run in the primary ‘‘music’’ process. To coordinate graphical and musical events, messages are sent from the music process to the graphics process to request operations.

This organization retains most of the simplicity of the single-process non-preemptive Moxc model. The only interprocess synchronization (a common source of complexity) is in the message queue. Furthermore, the interface to allocate and send messages is an easy-to-master set of procedure calls which hide most of the details.

3.5. Testing

Ritual presented some interesting testing problems because in performance, there are three independent MIDI sources providing data in real time. I decided to simulate these performances using a sequencer to drive and test *Ritual*. This simulated performance proved invaluable for composing the piece and adjusting parameters as well as debugging the software.

The simulation is supported by multitasking in the Amiga computer and by a special MIDI driver designed and implemented in my lab. The MIDI driver allows a process to designate one of many *ports* as the destination for MIDI messages. The default port corresponds to the hardware MIDI Out port, but others are available. Similarly, processes accept input from any subset of the ports. The default is to receive input from the MIDI In port, but any other port or set of ports can be selected. The CMU MIDI Toolkit allows ports to be specified on the command line, making it easy to start and interconnect several processes in an arbitrary fashion. A similar facility is provided in the Apple MIDI Manager and the MIDI Share software system [8]. For the simulation, the Adagio sequencer is started with its ouput port connected to the input port of the *Ritual* process. The output from *Ritual* is sent to synthesizers via MIDI Out.

Parts of *Ritual* were composed by improvising one of the instrumental parts while running a simulation. The Adagio sequencer was set up to record (from MIDI In) and play at the same time. The merged MIDI In and sequenced data were passed on to the *Ritual* process so that I could hear immediately how the system would respond in a performance to the same input. (See Figure 3-1). Parts could be built up in layers this way. After I was satisfied with the simulation, I transcribed the sequencer data to music notation by hand.

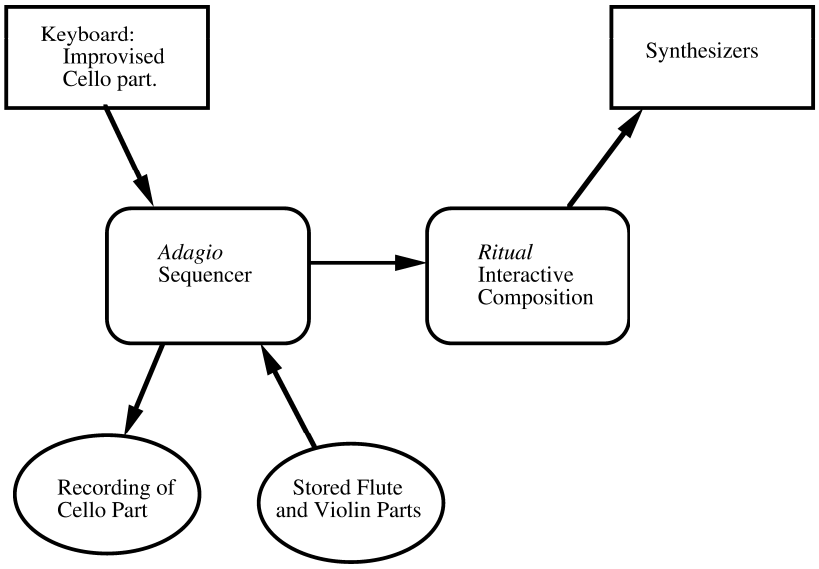

Figure 3-1: The *Adagio* sequencer simulates live performers by generating real-time input to the *Ritual* program, which runs concurrently. Live input from a keyboard or other MIDI source can be merged with pre-recorded sequences and forwarded to *Ritual*. Live input is also captured and saved to an editable score file.

4. Future Work

There are still many areas in which progress can be made. The Adagio sequencer is flexible, but has no graphical interface, and Adagio is also inadequate to support non-MIDI software synthesis (where notes have an arbitrary number of parameters) although to some extent this is supported by calls to arbitrary C routines.

Another area in need of attention is the representation of time-varying parameters. These are typically represented in MIDI systems as discrete control changes, but this does not support development and rehearsal where it is necessary to jump to some time point in the score and determine what various parameter values should be. This is complicated by the fact that time-varying parameters may be computed and may even depend upon external real-time input.

The whole area of multitasking seems ripe for rethinking. We need an approach that permits preemption without sacrificing the simplicity of non-preemptive systems. I have argued in favor of non-preemptive systems, but the need for a separate graphics task in my work is an indication that the non-preemptive approach has its limits. Two likely situations will cause even more problems than graphics. First, music understanding capabilities such as score following [3] and

beat tracking [1] can require a fairly large amount of computation to handle a single input event such as a MIDI Note On message. This will disrupt the timing accuracy of other music processing. Second, if digital signal processing is controlled directly from within the interactive composition [10], response times measured in microseconds may be necessary, again calling for a preemptive approach.

Finally, this paper has addressed technical issues of real-time interactive systems, but the esthetic issues will present the most difficult and interesting problems in the long run.

5. Summary

Interactive computer music and animation systems pose a number of problems. This paper has presented practical solutions to a number of problems that arise in the implementation of a particular composition. The multitasking problem is solved by using an event scheduler that enqueues timed procedure calls and their arguments for future execution. Event scheduling is very efficient and can be used to simulate most activities that would otherwise require true tasks. The event sequence problem is solved by extensions to the Adagio score language. These extensions allow scores to be invoked from within an interactive program and allow procedures to be called from within scores. Thus, time-driven and event-driven programs can be intermixed. The parameter adjustment problem is also solved with the Adagio score language, which allows the values of program variables to be set to any value at any time. Scores are interpreted rather than compiled, supporting rapid refinement. The scheduling problem is solved by moving expensive graphics operations to a low-priority task that can be preempted by a higher priority music processing task. The testing problem is also solved using multiple tasks: a sequencer provides real-time input via a MIDI driver that supports the connection of MIDI streams from one process to another.

6. Conclusions

Interactive computer music software demands much more support than a conventional programming language. Various programming abstractions that support multiple tasks, sequencing, and graphics, all with easy-to-use interfaces, are necessary. A good programming environment enables the composer to explore ideas rapidly and to produce reliable software for use in live performance.

There is much to be explored in musical applications of this technology. We have barely begun to explore ways in which improvisers and composers can interact. The role of animation in music performances is an open-ended question. It is hoped that this paper will encourage others to explore this new territory.

7. Acknoledgments

This Carnegie Mellon School of Computer Science has made this work possible. *Ritual* was written in honor of the twenty-fifth anniversary of Computer Science at Carnegie Mellon. The author would like to thank Yamaha Music Technologies and Commodore Amiga, Inc. for their generous support and contributions to this research.

References

1. Allen, P. E. and R. B. Dannenberg. Tracking Musical Beats in Real Time. ICMC Glasgow 1990 Proceedings, International Computer Music Association, 1990, pp. 140-143.

2. Bartlett, M. The Development of a Practical Live-Performance Music Language. Proceedings of the International Computer Music Conference 1985, International Computer Music Association, 1985, pp. 297-302.

3. Bloch, J. J. and R. B. Dannenberg. Real-Time Computer Accompaniment of Keyboard Performances. Proceedings of the International Computer Music Conference 1985, International Computer Music Association, 1985, pp. 279-290.

4. Chadabe, J., and R. Meyers. "An Introduction to the Play Program". *Computer Music Journal 2*, 1 (1978), 12-18.

5. Collinge, D. J. MOXIE: A Language for Computer Music Performance. Proceedings of the International Computer Music Conference 1984, International Computer Music Association, 1985, pp. 217-220.

6. Dannenberg, R. B. The CMU MIDI Toolkit. Proceedings of the 1986 International Computer Music Conference, International Computer Music Association, San Francisco, 1986, pp. 53-56.

7. Dannenberg, R. B. Real Time Control For Interactive Computer Music and Animation. The Arts and Technology II: A Symposium, Connecticut College, New London, Conn., 1989, pp. 85-94.

8. Orlarey, Y., H. Lequay. MidiShare, A Real Time Multi-tasks Software Module for Midi Applications. Proceedings of the 1989 International Computer Music Conference, International Computer Music Association, 1989, pp. 234-237.

9. Puckette, M. The Patcher. Proceedings of the 14th International Computer Music Conference, International Computer Music Association, 1988, pp. 420-429.

10. Viara, E., and M. Puckette. A Real-Time Operating System for Computer Music. Proceedings of the 1990 International Computer Music Conference, International Computer Music Association, 1990, pp. 270-272.