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. Also discussed are issues of asynchronous input, concurrency, memory management, scheduling, and testing. Two examples are described.

Keywords: music, interactive, composition, MIDI, multimedia, software

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. Computers, as we shall see, offer an alternative.

Unfortunately, real-time computer systems are among the most difficult to develop due to concurrency, the use of special purpose input/output devices, and the fact that time-dependent errors are hard to reproduce. Much has been written about real-time control, process control, scheduling and synchronization. So much has been written, in fact, that it is difficult to know what techniques to apply to a given problem. Often, in an effort to squeeze out every last drop of performance, researchers (including this one) have designed complex strategies which are

[1]Published as: Dannenberg, ‘‘Software Design for Interactive Multimedia Performance,’’ ***Interface - Journal of New Music Research***, 22(3) (August 1993), pp. 213-228.

primarily of academic interest. In this paper, I will present practical advice for building real-time systems for computer music and animation. This paper is a ‘‘report from the trenches’’ on techniques that have proven to be effective. This advice is based on experience building real systems and in many cases runs counter to common assumptions. I will also attempt to analyze the techniques to explain why they work.

1.1. Overview

Section 2 describes the artistic context that motivates this work. From this context, a number of technical problems arise, and these are described in Sections 4 through 10 along with some solutions and practical advice. Section 11 describes two examples of performance systems I have implemented, Section 12 describes future work, and Section 13 presents conclusions.

2. Interactive Composition

Composition by computer at the time of performance allows the composer to respond to live improvisation and vice versa, creating a 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 compositional strategies and which *do* run in real time. One might call these programs *meta-compositions* because the programs themselves are composers. The end result is a balanced mixture of composition and improvisation, where composer and improviser influence one another in real time.

2.1. 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 given that these real-time interactive computer music and animation programs are interesting, and I will describe a number of techniques that were 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.

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:

- **Input.** It is often necessary to handle multiple streams of input, but stopping to wait for input is usually not possible.
- **Concurrency.** Music generation typically requires many independent threads of control, but traditional approaches add program complexity.
- **Synchronization.** Concurrent tasks must communicate and share data.
- **Memory Management.** Data objects are often allocated in response to input and must be carefully deallocated when no longer needed.
- **Sequencing.** Programming is essential for interactive systems, but event sequences such as musical scores 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.

The following sections consider these issues in greater detail. These problems are all addressed in the CMU MIDI Toolkit, which will be used to illustrate solutions. The CMU MIDI Toolkit is a collection of software written in the C programming language and available from the author.

4. Handling Input

Traditionally, input has been handled in programming languages by calling procedures (or executing special *read* statements) which return data when it becomes available. But stopping to wait is not possible in a real-time system, where *time dependent* as well as *data dependent* computation takes place. One solution is an inverted structure where instead of programs calling ''in'' for input, the runtime system calls ''out'' to the program whenever input arrives. In real-time systems, programs typically then decode the input to determine an appropriate action to take. A standard decoding procedure can be provided by the language or run-time system, avoiding the need to reimplement input decoding in each application program.

As an example, the CMU MIDI Toolkit handles input from MIDI and the console (typewriter keyboard). The MIDI input is parsed into a number of message types and a C routine is called when a complete message is received. The routines are `keydown`, `keyup`, `pitchbend`, `prgmchng`, `aftertouch`, `ctrlchng`. In addition, `asciievent` is called when a character is typed at the console. The programmer can provide application-specific definitions for any or all of these routines.

5. Concurrency

Interactive music programs often require several musical lines or simultaneous notes. 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.

5.1. Event-Based Programming

While concurrent real-time programs are ordinarily implemented as multiple processes, each with its own stack, I believe that this approach is more complex and troublesome than necessary. An alternative, first described by Douglas Collinge [Collinge 85], is an event-based organization that uses only one stack and supports interleaved execution of many tasks.

The principal idea is to think of the program as consisting of many execution events, each of which consists of calling a short-lived procedure at a particular time with a particular set of operands. Input data give rise to events (as described in Section 2), and events may also cause other events, either immediately or some time in the future.

For example, in the CMU MIDI Toolkit, one can write: `cause(100, echo, pitch, loud)`. This says to call the routine named `echo` after a delay of 100 time units, passing the values of `pitch` and `loud` as operands. This event is saved in a time-ordered queue until the current event completes and the indicated time has elapsed. Other events may execute in the meantime. Extended computations that produce output, wait for some time, produce more output, wait again, etc. can be implemented by using `cause` within events to generate future events:

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

This simple organization is efficient because it is not necessary to save all of the processor's registers in order to create or execute an event. Executing an event amounts to calling a procedure, and this is normally faster than switching processes. Storage management is very simple and mostly automated by the system: the `cause` operation allocates an event record from a free list and stores the procedure address, operands and event time in the record. The record is then inserted in a queue. When the time arrives the event procedure is called with the saved operands and the event record is returned to the free list. Since this all happens automatically, it is much less error-prone then explicitly allocating, initializing, and freeing storage for processes.

Another advantage of this approach is that it is compatible with standard debuggers which often do not work properly with multiple processes and stacks. Also, debugging is simplified by having events called from the main program so that they can print to the console, be paused by a debugger, or single-stepped. In interrupt-driven, multiple-process systems debuggers are typically (but not inherently) of limited use. My main point here is that a simple system that can use an off-the-shelf debugger and even simple ‘‘print’’ statements is generally easier to program than a complex system requiring special debugging tools.

As an aside, it is worth mentioning that this event-based strategy is compatible with the polling of input devices as opposed to interrupt-driven input. Since interrupts are almost always more difficult to implement and debug than polling-based systems, interrupts should only be used when necessary for performance.

5.2. Preemption

Preemption occurs when one process is stopped at an arbitrary instruction in order to run another process. Preemptive systems are useful when it is important to run a high-priority process as soon as data arrives or when one wants to implement a ‘‘fair’’ scheduler that prevents a single long-running process from taking all of the processor time.

While these sound like nice properties, preemptive systems also have disadvantages. The main disadvantage is that a process may leave a data-structure in an intermediate state at the time of preemption. For example, suppose processes A and B each read a shared variable V, add 1, and store the result back into V. Now, suppose V is 0 and A reads V but is preempted before it stores the result (1). Process B might then increment V to 1. When A resumes, it also stores 1. This is one example of a classic operating systems problem: V was incremented twice, but due to preemption, the final value is 1! The standard solutions to this problem [Andrews 83] all involve making updates to shared data structures mutually exclusive. In this case, process B would be denied access to V until process A finished its operations. This requires extra processing to take place before and after updating the data structure. Notice, however, that this problem only arises if process A can be preempted. If we disallow preemption, data structure accesses will run without interruption, and no extra precautions need to be taken.

There are at least three advantages of non-preemptive systems. First, non-preemptive systems are usually the simplest and most efficient to implement. Second, non-preemptive systems do not incur as much overhead to lock and unlock data structures to achieve mutual exclusion. Finally and most importantly, non-preemptive systems obtain mutual exclusion by default, avoiding timing-dependent programming errors which are difficult to debug.

On the other hand, non-preemptive systems can only switch tasks when a process explicitly blocks waiting for input, delays, or requests that the system run another process. Therefore, a long-running computation can seriously degrade real-time performance by delaying other processes. Since we are interested in interactive, low latency systems, there are rarely any long-running computations. If there were, then the assumption of low latency would be false with or without preemption. In short, the problems solved by preemption do not normally exist in interactive real-time music and animation software. (I will discuss some exceptions later.) Another problem with non-preemptive systems is that special care must be taken so that other processes can run when a process waits for input. This situation does not occur when input is handled as described in Section 4.

6. Memory Management

Memory management is often a problem in multi-tasking systems or object-oriented systems. The problem is that tasks or objects must be created dynamically to handle input events or to launch new activities. The creation and initialization of objects can be expensive, and it is usually the programmer's task to free storage when it is no longer needed. Freeing storage to which there are still outstanding references is a common error.

Using the event-based programming scheme outlined in Section 5.1, notice that no storage is explicitly allocated or deallocated in the `lots-o-notes` example. The `cause` function automatically allocates space to save parameters until the function is called. After the call, the space is automatically reclaimed. This is important in making CMU MIDI Toolkit programs reliable and easy to write. Of course, no formalism is perfect for all applications. Since no state is explicitly allocated, there is no name or reference to the future action generated by `cause`. This makes it difficult to cancel an action.[2]

Another drawback of this system is that a ''process'' does not retain any state information from one action to the next. Usually, state information can be passed in the parameter list to `cause`. In the example above, the per-process state information consists of a single integer channel. Note how channel is passed as a parameter from each activation of melody to the next through the parameter list of `cause`. If the amount of state is large, a reference to a large structure can be passed, but then it is up to the programmer to allocate and deallocate the structure explicitly.

Another approach to memory management is to use garbage collection to free storage. Unfortunately, real-time garbage collection requires language and compiler support and is difficult to implement, so it is rarely used. Another option is reference counting, which can also benefit from language and compiler support. Explicitly coded reference counting is error prone, but quite useful as a last resort in real-time systems.

A few more practical issues are worth mentioning: When possible, preallocate memory before starting to run. When memory must be allocated dynamically, preallocate large chunks and then allocate from the chunks. This avoids time-critical calls to the language run-time system, which may not provide fast memory allocation. Often, system calls to free memory are also slow, so it is safer to place freed memory on linked lists (one list per object size) and have future allocations reuse these freed objects.

7. Sequencing

Programming languages make it awkward to express arbitrary sequences of notes and other events. Facilities such as `cause` are excellent when sequences are to be computed according to some algorithm, but few programming languages are convenient for predetermined (composed) sequences, especially polyphonic ones. 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.

[2]The usual solution is to have the action check a flag before doing anything else. To cancel the action, you simply set the flag.

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 `cause` and the C programming language, but conventional music sequences are written in Adagio and ''conducted'' from within the 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 support timing and sequencing even further, scores operate according to virtual time systems that allow the timing of scores and programs to vary with respect to real time. This facility makes it possible to select starting and stopping points in a score and to synchronize scores with external events. For example, a score can be stopped temporarily by making its virtual clock run infinitely slow. In addition, we have developed MIDI synchronization, conducting, and computer accompaniment systems, all of which maintain synchronization by manipulating the virtual time of the score with respect to real time.

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.)

8. 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.

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.

One 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, 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 are easy to program, e.g.

```
    midi_note(ch,pitch,100+vel_offset[ch]);
```

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. [Loy 85, Pope 91, Anderson 90]. 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 [Chadabe 78], Mabel [Bartlett 85], and Max [Puckette 88] achieve efficiency through pre-compiled modules that are interactively connected, but do not provide the generality of ordinary programming languages.

9. Scheduling

As discussed above, event-based programming provides a simple approach to computing multiple independent streams of events. Routines (function calls) are scheduled 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.

In my experience, it is better to separate long-running computations into another process and use preemption to minimize the latency of computations. This brings us back to the problems of synchronization and mutual exclusion discussed in Section 5. An effective way to avoid these problems is to use a message-based client/server interface between processes. The idea is that a client wanting some service (for example, a graphics operation) sends a request message to a server process. The server reads one request at a time and uses a reply message to return results to the client. The message queue(s) are the only shared data structures and otherwise, both the client and server can be programmed as if they are never preempted. This organization is easy to manage and leads to reliable programs.

Circular buffers are especially good data structures to implement client/server message interfaces. Circular buffers have the advantages that they require no dynamic storage allocation and in cases where there is only one reader and one writer, no locking of the data structure is necessary to prevent interference, even with preemption. Interrupt-driven device handlers should also use circular buffers to communicate with processes that use the devices.

This organization retains most of the simplicity of the single-process non-preemptive model presented in Section 5. The only interprocess synchronization (a common source of complexity) is in the message queue. Furthermore, the interface to allocate and send messages can be an easy-to-master set of procedure calls which hide most of the details.

10. Testing

All of these techniques encourage well-structured and reliable programs, but programming is inherently error prone, so careful testing is essential. Interactive systems present some interesting testing problems because live input is required. Simulation of performances using a sequencer can be invaluable for composing and adjusting parameters as well as debugging software.

In my work, simulation is supported by multitasking 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 [Apple 90] and the MIDI Share software system [Orlarey 89], but the Macintosh lacks a true multitasking capability.

11. Examples

In this section, I will give two examples of interactive music systems that illustrate some of the recommendations given above. The first is a relatively simple system for customizing a special controller, the VideoHarp, for a specific performance. The second is a composition for 3 performers, computer music, and computer animation.

11.1. The VideoHarp in The Night of Power

The Night of Power, by Reza Vali, is scored for strings, brass, percussion, and VideoHarp. The VideoHarp [Rubine 88] is a MIDI controller, shaped roughly like a harp (see Figure 11-1), whose sensors detect the positions of fingers on two playing surfaces. The VideoHarp supports a number of playing gestures, including strumming, bowing, and keyboard emulation. In spite of this flexibility, the features implemented by the VideoHarp developers were not a perfect match to the demands of The Night of Power, making it very difficult to perform the piece. (In defense of the manufacturer, most of the difficulties were to be addressed in the next software release, but a shorter-term solution was required.)

The first problem was velocity control: the VideoHarp, when emulating a keyboard, is not velocity sensitive. Although a pedal can be used, I found it difficult to control with precision. Another problem was that ''keys'' on the VideoHarp, although marked with an etched pattern, are difficult to hit precisely, and touching between two keys can lead to double attacks. This might not be a problem with an experienced VideoHarp performer, but I wanted some computer assistance.

The approach I took was to write a CMU MIDI Toolkit (CMT) program that would be inserted between the VideoHarp and the synthesizers it was to control. The desired action of the CMT program is different in different sections of the piece, so the computer's RETURN key is used to step through 14 sections. Stepping to the next section automatically sends a program change to the synthesizers. For rehearsal purposes, the SPACE bar backs up one section.

For velocity control, no universal approach was found to be adequate, so different techniques were used in different parts of the piece. In some sections, a constant velocity is appropriate for an entire section. Advancing to one of these sections sets the velocity to be used. In another section, there are diminuendi, but the notes can all be played with the right hand on the right surface of the VideoHarp. There, the left hand moves up and down to control volume, as if moving a large fader. Of course, the VideoHarp generates notes when the left hand moves, so the CMT program captures these notes and computes a velocity based on the pitches it receives.

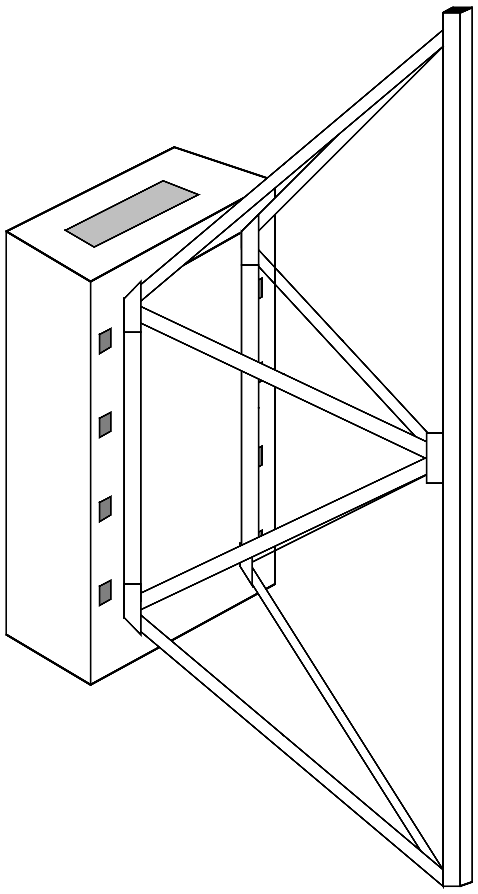

Figure 11-1: The VideoHarp MIDI controller. Light strip at right provides edge lighting along two transparent trapezoidal playing surfaces supported by a metal frame. Optical sensors and a computer in the enclosed base track fingers and output MIDI data to synthesizers.

In the final case, a crescendo is required while the right hand plays chords and the left hand plays arpeggios. Since the arpeggio rises throughout the crescendo, the velocity is tied to the pitches of the left hand and scaled to achieve the desired effect. This effect was implemented in a few minutes during a rehearsal by adding just a few lines of code.

Another problem encountered was to map areas of the VideoHarp into single notes to make it easier to strike certain pitches. The VideoHarp normally plays scales, but it was a simple matter to map pitch ranges into single pitches. Unfortunately, with just pitch mapping, it is still possible to accidentally hit two pitches, producing a double attack, or to hit in between two pitches, producing nothing at all! The solution was to use a strumming or wiping gesture within a mapped region to insure that at least one note was played by the VideoHarp. Software then filters out the multiple attacks, using a timeout mechanism.

The software for Night of Power is straightforward, but it would be difficult or impossible to achieve the same effects without programming. CMT made the task easy and the result is reliable. A facility like CMT for customizing controllers is very important.

11.2. Interactive Composition: Ritual of the Science Makers

Ritual of the Science Makers is an interactive composition for flute, violin, and cello. All three instruments are connected via pitch-to-MIDI converters to a computer, which generates music in response to incoming MIDI notes. In addition to MIDI, the computer also generates graphical animations that respond to the live performers. The CMU MIDI Toolkit was used to implement the piece, which relies on many of the features discussed above.

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.

Ritual uses preemptive scheduling so that relatively slow graphics operations can be called without affecting the critical timing of musical actions. The interface to the graphics process is via messages to eliminate the complexity of concurrent access to shared variables. If the music task is not run at a higher priority than the graphics task, the degradation in performance is quite noticeable.

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 than abrupt changes.

Sequences are used extensively in the piece, not so much for sequencing notes as for scheduling parameter changes. For example, in one section, incoming notes are transposed to form chords. To avoid monotony, the transpositions applied to each of the three instruments are changed over time by a sequence. In transitions between sections, parameters of a digital signal processor are slowly changed according to a stored sequence. In some sections, parameters that control the interactive composition are changed at designated times according to a score. The use of scores to store pre-planned sequences of actions makes it easy to adjust parameters and to edit the exact timing of various actions.

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 11-2). 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. Later, the Adagio versions of the instrumental parts were used to drive simulated performances for testing as described in Section 10.

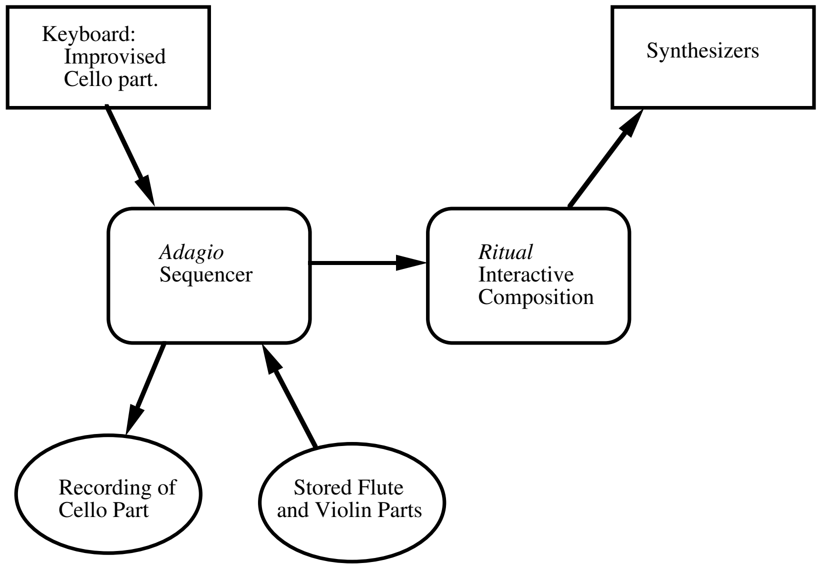
Keyboard: Improvised Cello part.
Synthesizers
Adagio Sequencer
Ritual Interactive Composition
Recording of Cello Part
Stored Flute and Violin Parts

Figure 11-2: 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.

12. Future Work

So far, my focus has been on areas where the CMU MIDI Toolkit offers valuable support for real-time interactive performance systems. However, no system is perfect, and there are areas where more support is needed. One area where CMT is not particularly helpful is in graphical user interfaces for real-time systems. An attempt was made to build some tools in Lisp: a message interface to CMT allows the user to set parameters and invoke functions in response to adjusting sliders, clicking on buttons, or typing in values. Although this system did become operational, there were several problems.

- While it is possible to set values from the graphical interface, the interface has no facilities for monitoring values of variables as they are changed under program control (monitoring requires care to maintain the low-latency characteristics of the music process);
- The system is unwieldy, requiring many processes and windows to be started and configured;

- There is no automatic way to save and restore parameter settings.

We hope to rectify these limitations in a future system.

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. The Explode system [Puckette 90] is an example of a graphical sequencer designed for interactive compositions.

CMT offers little support for continuous functions of time, even though it is quite common to use MIDI continuous controller commands to smoothly change some synthesis parameter. For example, pitch bend can be used to produce a vibrato, and digital signal processor parameters like reverberation time can be controlled with a time-varying function. The language Arctic [Dannenberg 86a] provides a good semantic model, and it would be nice to have an efficient implementation of Arctic integrated with the CMT. With faster processors, even signal processing is possible in software [Dannenberg 91a, Dannenberg 92]. Another problem is keeping consistent state, especially with MIDI.

The areas of interactive composition and mixed media compositions have raised many artistic questions as well as technical ones. There are many possibilities waiting to be explored.

13. 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.

I have attempted to de-mystify the art of writing reliable and efficient interactive real-time music and animation software. I have argued for simple organizations that are amenable to traditional debugging techniques and which avoid error-prone synchronization and storage management. The resulting approach has been successfully used to implement a number of music and animation programs.

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.

For further reading, the original CMU MIDI Toolkit manual is available [Dannenberg 93], and this system is also described in a short paper [Dannenberg 86b]. The issues of preemption and high-latency graphics operations are discussed in [Dannenberg 89], and a longer discussion of CMT and Ritual of the Science Makers appears in [Dannenberg 91b]. A more detailed list of recent additions to CMT appears in [Dannenberg 90].

14. Acknowledgments

The Carnegie Mellon School of Computer Science has made this work possible. Ritual of the Science Makers 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.

Portions of this work were also supported by IBM and the CMU Information Technology Center, the Studio for Creative Inquiry, and the Music Department. Special thanks are due to Noel Zahler, whose Arts and Technology Symposia provided the initial motivation to organize and present these ideas, and to Leonello Tarabello, whose symposium led to this article.

References

[Anderson 90] Anderson, D. P. and R. Kuivila. A System for Computer Music Performance. *ACM Transactions on Computer Systems* 8(1):56-82, February, 1990.

[Andrews 83] Andrews, G. R. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys* 15(1):3-43, March, 1983.

[Apple 90] Apple Computer, Inc. *MIDI Management Tools, Version 2.0* Apple Programmers and Developers Association, 1990.

[Bartlett 85] Bartlett, M. The Development of a Practical Live-Performance Music Language. In B. Truax (editor), *Proceedings of the International Computer Music Conference 1985*, pages 297-302. International Computer Music Association, 1985.

[Chadabe 78] Chadabe, J., and R. Meyers. An Introduction to the Play Program. *Computer Music Journal* 2(1):12-18, 1978.

[Collinge 85] Collinge, D. J. MOXIE: A Language for Computer Music Performance. In W. Buxton (editor), *Proceedings of the International Computer Music Conference 1984*, pages 217-220. International Computer Music Association, 1985.

[Dannenberg 86a] Dannenberg, R. B., P. McAvinney, and D. Rubine. Arctic: A Functional Language for Real-Time Systems. *Computer Music Journal* 10(4):67-78, Winter, 1986.

[Dannenberg 86b] Dannenberg, R. B. The CMU MIDI Toolkit. In *Proceedings of the 1986 International Computer Music Conference*, pages 53-56. International Computer Music Association, San Francisco, 1986.

[Dannenberg 89] Dannenberg, R. B. Real Time Control For Interactive Computer Music and Animation. In N. Zahler (editor), *The Arts and Technology II: A Symposium*, pages 85-94. Connecticut College, New London, Conn., 1989.

[Dannenberg 90] Dannenberg, R. B. Recent Developments in the CMU MIDI Toolkit. In C. Sandoval (editor), *Proceedings of the International Seminar Ano 2000: Theoretical, Technological and Compositional Alternatives*, pages 52-62. Universidad Nacional Autonoma de Mexico, Mexico City, 1990.

[Dannenberg 91a] Dannenberg, R. B., C. L. Fraley, and P. Velikonja. Fugue: A Functional Language for Sound Synthesis. *Computer* 24(7):36-42, July, 1991.

[Dannenberg 91b] Dannenberg, R. B. Software Support for Interactive Multimedia Performance. In D. Smalley, N. Zahler, and C. Luce (editor), *Proceedings of The Arts and Technology 3*, pages 85-94. Connecticut College, New London, Conn., 1991.

[Dannenberg 92] Dannenberg, R. B. Real-Time Software Synthesis on Superscalar Architectures. In *Proceedings of the 1992 ICMC*, pages 174-177. International Computer Music Association, San Francisco, 1992.

[Dannenberg 93] Dannenberg, R. B. *The CMU MIDI Toolkit* 1993.

[Loy 85] Loy, G. Musicians Make a Standard: The MIDI Phenomenon. *Computer Music Journal* 9(4):8-26, Winter, 1985.

[Orlarey 89] Orlarey, Y., H. Lequay. MidiShare, A Real Time Multi-tasks Software Module for Midi Applications. In T. Wells and D. Butler (editor), *Proceedings of the 1989 International Computer Music Conference*, pages 234-237. International Computer Music Association, 1989.

[Pope 91] Pope, S. T. Introduction to MODE: The Musical Object Development Environment. *The Well-Tempered Object: Musical Applications of Object-Oriented Software Technology.* In S. T. Pope, MIT Press, Boston, 1991, pages 83-106.

[Puckette 88] Puckette, M. The Patcher. In C. Lischka and J. Fritsch (editor), *Proceedings of the 14th International Computer Music Conference*, pages 420-429. International Computer Music Association, 1988.

[Puckette 90] Puckette, M. EXPLODE: a user interface for sequencing and score following. In S. Arnold and G. Hair (editor), *ICMC Glasgow 1990 Proceedings*, pages 259-261. International Computer Music Association, San Francisco, 1990.