Software Techniques for Interactive Performance Systems[1]

Roger B. Dannenberg
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
dannenberg@cs.cmu.edu
(412) 268-3827

ABSTRACT

The CMU MIDI Toolkit supports the development of complex interactive computer music performance systems. Four areas supported by the system are: input handling, memory management, synchronization, and timing (scheduling and sequencing). These features are described and then illustrated by their use in two performance systems.

1. Introduction

Complex interactive real-time systems are among the most difficult to program, but certain techniques can simplify the programming task and increase software reliability. Four aspects of real-time systems that often lead to complexity and errors are: input handling, memory management, synchronization, and timing. Practical approaches to these problems have been developed and incorporated into the CMU MIDI Toolkit, a set of support tools and programming libraries for interactive computer music systems. This paper is a ‘‘report from the trenches’’ on techniques that have proven to be effective. I will also attempt to analyze the techniques to explain why they work.

The techniques are divided into four areas: input management, memory management, synchronization, and timing. These are covered in the next four sections. The paper concludes with a discussion of these techniques and suggestions for future work.

---

[1]Published as: Dannenberg, ‘‘Software Techniques for Interactive Performance Systems,’’ in ***International Workshop on Man-Machine Interaction in Live Performance***, Scuola di Studi Superiori Universitari e di Perfezionamento, Pisa, Italy, 1991, pp 19-28.

2. Input Handling

Languages have traditionally treated input and output with symmetry; that is, both input and output are synchronous operations that take place under program control, and the program stops until the operation is completed. This works fine when you always know when to expect the next piece of input data and when there is nothing else to do while waiting for input. In computer music systems, however, input arrives in an unpredictable order at unpredictable times, so waiting for input is a bad idea. For example, after receiving a MIDI note-on command, it is not possible to simply issue another read operation to get the corresponding note-off command, because other inputs may arrive in the meantime, and there may be other processing to take care of while waiting.

The CMU MIDI Toolkit (CMT) [Dannenberg 86a] uses an approach in which all input is managed by a central dispatcher that parses incoming MIDI and other events. An *event* refers to the arrival of data or the passage of time such that some computation, called an *action*, should be performed. When a complete MIDI message event arrives, a routine (the action) is called; for example, `keydown(3, 60, 100)` is called when a MIDI note-on with pitch middle C and velocity 100 arrives on channel 3. `Keydown()` is not a part of CMT but rather a function that must be defined by each program that uses CMT. This procedural interface eliminates the need to parse or decode input data, which is a common source of programming errors.

In a simple program, a single procedure can handle all note-on messages, but in more complex programs, it is often necessary to provide many different handlers to be used in different situations. Some toolkits have addressed this problem by allowing the programmer to change at run-time what method or routine to call for each event [Boynton 86]. CMT advocates a simpler method: `keydown()` should act as a dispatcher that forwards calls to the appropriate handler, depending upon program variables. The advantage is that the behavior is clearly indicated in the program itself as opposed to being controlled by ‘‘patches’’ that can be changed anywhere in the program.

3. 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. An alternative approach (pioneered by Doug Collinge [Collinge 85, Collinge 88]) is to enqueue a routine name to be called in the future and a list of parameters. This simple operation can be used to simulate most tasks, and it hides all storage management from the application programmer. It is also quite fast.

In CMT, an action can be scheduled for the future by calling the routine `cause()`. For example, the following will turn a note off one second in the future:

```
cause(100, midi_note, 1, 60, 0);
```

The parameters are the time delay (100) before calling a routine, the routine to call (`midi_note`), and the parameters to be passed to that routine (1, 60, and 0). In this case, the routine will send a MIDI note-on on channel 1, with pitch 60, and velocity 0.

It is important that all routines in CMT execute in a short amount of time so that no pending computation will be substantially delayed. By ‘‘causing’’ a call to itself in the future, a routine can extend its behavior over time. For example, the following routine will generate an infinite random melody. Any number of these melodies can run concurrently:

```
melody(channel)
  int channel;
{
    int dur = random(20, 100);
    int pitch = random(48, 72);
    if (melody_halt) return;
    midi_note(channel, pitch, 100);
    cause(dur - 1, midi_note, channel, pitch, 0);
    /* now cause the next melody note */
    cause(dur, melody, channel);
}
```

Notice that no storage is explicitly allocated or deallocated in this example. This is important in making CMT programs reliable and easy to write. Of course, no formalism is perfect for all applications. Since no state is explicitly allocated, it requires special care to terminate a CMT ‘‘process’’. This is usually done by having the ‘‘process’’ check a status indicator each time an action is about to be taken. In the example above, all melody processes can be stopped by setting `melody_halt` to TRUE.

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 pointer to a large structure can be passed, but then it is up to the programmer to allocate and deallocate the structure explicitly.

In summary, all computation in CMT consists of calls on routines that execute for a relatively short period of time without preemption. One of these non-preemptable executions is called an *action*. Actions are initiated in two ways: first, all external events (such as an incoming MIDI message) are mapped into calls on routines by a central dispatcher, and second, the `cause()` routine can be used to schedule a call to a routine at some time in the future. Simple tasks can be simulated with this mechanism, and an advantage is that storage management is implicit and automatic.

4. Synchronization

Synchronization is simplified in CMT by eliminating preemption, which requires locks or semaphores to guard against overlapping shared access to critical variables. In general, this approach is a major simplification, and it eliminates a source of common and elusive bugs. The drawback is that the worst-case response time of the system to an external event will be greater than the longest action anywhere in the system. Referring back to the `melody` example, if `melody` executes in 2 milliseconds, then the worst case timing accuracy and response latency to any input will be at least 2 milliseconds. This is true because if `melody` begins to run just before some other action is ready for execution, the other action will have to wait until `melody`

completes.

In practice, this is usually not a problem. A surprising amount of work can be done in a a few milliseconds, the time scale of interactive music systems. Also, it is often the case that all computations are important, so there may be no basis for guaranteeing a low latency of one computation at the expense of another. Finally, application-specific techniques can often be used to avoid overload conditions. For example, a process might discard any note attacks that are closer than 10 milliseconds to make sure there is enough processing power reserved for other processes.

In spite of these arguments in favor of non-preemptive programming, cases still arise where preemption is unavoidable. One example is handling low-level events such as MIDI, mouse input, and the ascii keyboard. It would be impractical to respond to these hard real-time events without preemption. In CMT, these external events are captured by interrupt handlers that preempt the CMT process. MIDI and other device data is enqueued to be processed later by CMT.

Another example is the use of graphics, where drawing an image non-preemptively would lock up the system for many milliseconds. For the Amiga version of CMT, graphics operations are run in a process that runs preemptively at a lower priority than the music processing. All decision-making runs at the high priority, but graphics rendering is deferred until the main process is idle.

A similar scheme has been used for managing user interfaces. A lisp-based interface manager can be run in a low-priority task so that the relatively slow user-interface operations do not interfere with low-latency music processing. Synchronous interprocess communication operations are used to invoke routines within the music process or to set variables. This communication blocks until the currently executing action in the music process completes so as to preserve the non-preemptive semantics within the music process.

A beat-tracking system constructed with Paul Allen [Allen 90] is the only system where our non-preemptive approach broke down in the context of music processing. This application uses a computationally intensive search to identify downbeats and a ‘‘drummer’’ that plays on the downbeats. In order to achieve precise drum timing, the drummer process is run at a high priority and the search is run at a lower priority. As in the other examples, message passing is used to communicate between the two processes.

All four of these examples illustrate the same principle: when preemption is really necessary, the program can be divided into a number of tasks which communicate by messages. Message passing, as opposed to shared variables, is a structured form of communication and synchronization that preserves the important properties of non-preemptive code. In particular, variables and data structures are not changed during the execution of any action except by that action itself. In contrast, if preemption can occur at any time, then variables and data structures may be changed at any time by some other action, so program analysis and debugging can be much more difficult.

5. Timing

The timing problem is that, in music, it is often necessary to specify elaborate sequences of actions (e.g. musical scores), but these are difficult to express in most programming languages, including the C language in which CMT programs are written. Rather than extend C, the existing text-based score language, Adagio, was adapted for use within CMT programs.

It is now possible to play a score from within a CMT program. This makes it possible to, for example, play a complex musical sequence in response to some MIDI event. Scores can by created using a text editor, by capturing real-time MIDI, or by converting standard MIDI files. The resulting scores are stored in files and are loaded at the beginning of CMT program execution. The scores can then be played at any time under program control. It is much easier to generate MIDI sequences by this method than by writing C code.

In practice, it is often necessary to sequence calls to various processes written in C. For example, one might wish to change over time the way in which input events are processed. This form of timed execution sequence is awkward to express in C, so the Adagio score language was extended so that scores can call functions and set variables. This extended score language can sequence internal program actions as well as external (MIDI) events.

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.

Having described the CMU MIDI Toolkit approach to developing real-time interactive music systems, we will now turn to two specific examples. The first is an extended interface for the VideoHarp MIDI controller, and the second is an interactive composition, *Ritual of the Science Makers*. These examples illustrate useful features of CMT.

6. The VideoHarp in *The Night of Power*

The Night of Power[2], by Reza Vali, is scored for strings, brass, percussion, and VideoHarp. The VideoHarp [Rubine 88] is a MIDI controller, shaped roughly like a harp, 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 Sensor Frame Corporation, 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

[2]Commissioned by the Carnegie Mellon University School of Computer Science in celebration of the 25th Anniversary of Computer Science at Carnegie Mellon

velocity sensitive. Although a pedal can be used, we 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 hitting 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 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 the 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 in that section, the notes can all be played with the right hand on the right surface of the VideoHarp. In this section, 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. 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 the following lines to the note-on handler:

```
/*  If in section 9 and pitch is below 60, send the
 *  note out with the same channel and pitch, but
 *  interpolate the velocity from SEC9LOW to SEC9HIGH
 *  as pitch increases from 24 to 60.  (SEC9LOW and
 *  SEC9HIGH are initialized by reading a file at
 *  startup time.)  Otherwise send the note-on without
 *  alteration.
 */
} else if (section == 9) {
    if (k < 60) {
        midi_note(c, k, SEC9LOW +
            ((k - 24) * (SEC9HIGH - SEC9LOW)) /
            (60 - 24));
    } else midi_note(c, k, v);
```

Another problem encountered was to map areas of the VideoHarp into single notes to make it easier to strike certain pitches. The VideoHarp was pre-programmed to play 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 withing 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.

7. 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.

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.

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.

8. 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.

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 86b] provides a good semantic model, and it would be nice to have an efficient implementation that is integrated with the CMT model.

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.

9. Conclusions

The CMU MIDI Toolkit has been a valuable tool for research, composition, and education. It offers a simplification of the traditional approach to concurrent computation as presented in operating systems textbooks, yet the CMT approach is quite powerful and flexible. One of the nicest properties is that concurrent programs can be realized without managing task control blocks, semaphores, and other paraphernalia, without system-specific calls, and without any special debugging techniques. The integration of scores (sequences) into an otherwise procedural programming system has added a great amount of power, especially because scores can have procedure calls embedded in them.

For further reading, the original CMU MIDI Toolkit manual is available [Dannenberg 93], and this system is also described in a short paper [Dannenberg 86a]. 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 91]. A more detailed list of recent additions to CMT appears in [Dannenberg 90].

10. Acknowledgments

The author would like to thank IBM and the CMU Information Technology Center their support. This work has evolved over a number of years with additional support from the CMU Music Department, the School of Computer Science, Yamaha, and Commodore.

References

[Allen 90] Allen, P. E. and R. B. Dannenberg. Tracking Musical Beats in Real Time. In S. Arnold and G. Hair (editor), *ICMC Glasgow 1990 Proceedings*, pages 140-143. International Computer Music Association, 1990.

[Boynton 86] L. Boynton, J. Duthen, Y. Potard, and X. Rodet. Adding a Graphical Interface to FORMES. In P. Berg (editor), *Proceedings of the International Computer Music Conference 1986*, pages 105-108. International Computer Music Association, 1986.

[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.

[Collinge 88] Collinge, D. J. and Scheidt, D. J. MOXIE for the Atari ST. In C. Lischka and J. Fritsch (editor), *Proceedings of the 14th International Computer Music Conference*, pages 231-238. International Computer Music Association, 1988.

[Dannenberg 86a] 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 86b] 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 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 91] 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 93] Dannenberg, R. B. *The CMU MIDI Toolkit* 1993.

[Rubine 88] Rubine, D. and P. McAvinney. The VideoHarp. In *Proceedings of the 14th International Computer Music Conference*, pages 49-55. International Computer Music Association, 1988.