Real Time Control For Interactive Computer Music and Animation[1]

Roger B. Dannenberg
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

Real-time systems are commonly regarded as the most complex form of computer program due to parallelism, the use of special purpose input/output devices, and the fact that time-dependent errors are hard to reproduce. Several practical techniques can be used to limit the complexity of implementing real-time interactive music and animation programs. The techniques are: (1) a program structure in which input events are translated into procedure calls, (2) the use of non-preemptive programs where possible, (3) event-based programming which allows interleaved program execution, automatic storage management, and a single run-time stack, (4) multiple processes communicating by messages where task preemption is necessary, and (5) interface construction tools to facilitate experimentation and program refinement.

These techniques are supported by software developed by the author for real-time interactive music programs and more recently for animation. Although none of these techniques are new, they have never been used in this combination, nor have they been investigated in this context. Implementation details and examples that illustrate the advantage of these techniques are presented. Emphasis is placed on software organization. No specific techniques for sound and image generation are presented.

1. Introduction

Much has been written about real-time control, process control, scheduling and synchronization. So much has been written 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 primarily of

[1]Published as: Dannenberg, ‘‘Real Time Control For Interactive Computer Music and Animation,’’ in ***Proceedings of The Arts and Technology II: A Symposium***, Connecticut College, (February 1989), pp 85-94.

academic interest. In this paper, I will present practical advice for building real-time systems for computer music and animation. This advice is based on experience building real systems and in many cases runs counter to common assumptions.

I will begin with a description of the sort of ‘‘real-time’’ system to which the subsequent recommendations apply. First, these systems are *interactive*; that is, they respond to input as opposed to producing a sequence of predetermined outputs. This implies that *low latency* is an important goal. The desired maximum latency is on the order of milliseconds. These systems are *soft* real-time systems in that outputs are still important even if the output cannot be produced within the desired maximum latency. Most activity is asynchronous and occurs at unpredictable times. Finally, these systems are *not overloaded* except for brief periods; that is, the average workload does not exceed the processing capability, so all scheduled computations can be completed.

In the following sections, I make five recommendations for designing real-time software of this sort and explain the advantages and disadvantages of each recommendation. The first is to handle input data by mapping input into procedure invocations. The second is to avoid preemption whenever possible. The third is to use a particular event-based programming strategy that handles interleaving, storage management, and executes on a single stack. The fourth recommendation is to use multiple processes that communicate through messages when preemption is necessary, and the fifth is to use interface construction tools as an aid to testing and program development. All of these principles are embodied in a recent version of the CMU Midi Toolkit [8], which is used as a source of examples.

2. Handling Input

Traditionally, input has been handled in programming languages by calling procedures (or executing special statements) which return data when it becomes available. In real-time systems, programs typically then ‘‘decode’’ the input to determine an appropriate action to handle the new data. This is clumsy and inappropriate in two respects. First, it is inappropriate to block waiting for some particular input. Input may arrive from another device first, or it may become time to run a previously scheduled action before any data arrives. Second, if input is almost always decoded to determine an action to take, then a standard decoding procedure should be provided by the language or run-time system rather than each application. This avoids the need to reimplement input routines for 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()`, `pitchblend()`, `prgmchng()`, `aftertouch()`, and `ctrlchng()`. In addition, `asciievent()` is called when a character is typed at the console. Default behaviors for all of these routines are provided in a library, and programs are linked so that the programmer can provide application-specific definitions for any or all of these routines.

This same technique is often used for building display-oriented user interfaces where an application must handle interleaved events generated by the mouse, menu system, and console keyboard [4]. This organization helps in writing software that is always ready to handle any input event as opposed to always being in some mode expecting a particular type of input.

3. 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 are adding data to an array by assigning a value to `V[i]` and then incrementing index `i`. Suppose A assigns a value to `V[i]` but is preempted before it increments `i`. Process B might then overwrite `V[i]`. This is one example of a classic operating systems problem. The standard solutions to this problem [1] all involve making updates to shared data structures mutually exclusive. In this case, process B would be denied access to array `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 the previous section.

4. Event-Based Programming

While concurrent real-time programs are ordinarily implemented as multiple processes, each with its own stack, I will argue here that this approach is more complex and troublesome than necessary. An alternative, first described by Douglas Collinge [6], 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.

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 the queue. When the time arrives, the event procedure is called with the saved operands and the event record is returned to the free list. In practice this is much less error-prone than 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 stacks. Although it is possible to execute events within an interrupt handler, 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. Multiple Processes

The organization described above is perfectly adequate for a variety of real-time programs; however, there are cases where some long-running computations are necessary. User interfaces that perform expensive graphics operations such as scrolling or image generation are examples of long-running computations. One may also want to combine low-latency (fast) music processing with relatively slow graphic operations. These can always be handled by breaking them up into a sequence of shorter computations, but this is error-prone and it can make the program unreadable. Input device handling, Midi in particular, requires very low latency responses to avoid losing data. This hard real-time constraint makes preemption almost essential.

In these cases, 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 3. In my experience, the most 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.

6. Interface Construction Tools

Following the previous advice makes it possible to quickly put together rather elaborate interactive programs. Typically, programs that produce music and graphics have many parameters that must be adjusted interactively to achieve the desired results. This implies that user interface construction will be a major component of the overall programming task. This effort can be minimized through the use of visual programming tools that allow the user to interactively place software buttons, switches, sliders, and other controls onto a control panel and then define the action which is to take place when the control is manipulated. Once designed, the control panel can be saved to a file for use by an application or for further editing. This technique was first demonstrated by Jean-Marie Hullot.

This approach can dramatically reduce the implementation time and increase the quality of user interfaces. Since interfaces are easy to construct in this way, the implementer is encouraged to use visual interfaces for debugging and testing as well as for the final product. This is extremely valuable for exploratory programming which is typical in the development of artistic music and animation software.

7. Putting It Together

An experimental version of the CMU Midi Toolkit was produced to incorporate all of these recommendations. Time critical software is written in C and runs non-preemptively in a single high-priority ‘‘music’’ process. Within this music process, an event-based structure as described in section 4 is used so that this single process actually supports many interleaved operations. All Midi input is decoded by this process and each Midi message results in a call to a C routine as discussed in section 2. Figure 7-1 illustrates connections among processes and interfaces.

A user interface is provided by another ‘‘graphics’’ process which runs at a lower priority. This process provides an interactive screen-oriented editor to create control panels consisting of sliders, buttons, switches, and text labels. The editor is written is Lisp so that arbitrary actions can be attached to controls and used immediately without compilation. In keeping with the recommendation to use a message-based interface, the control panel process sends messages to the high-priority music process to effect changes. The messages are handled exactly like Midi and console input and do not preempt running events. Messages can be used to set variables and cause events within the music process.

The issues of garbage collection and execution speed are always raised in the context of Lisp-based real-time systems. Garbage collection is the way Lisp systems reclaim and reuse memory

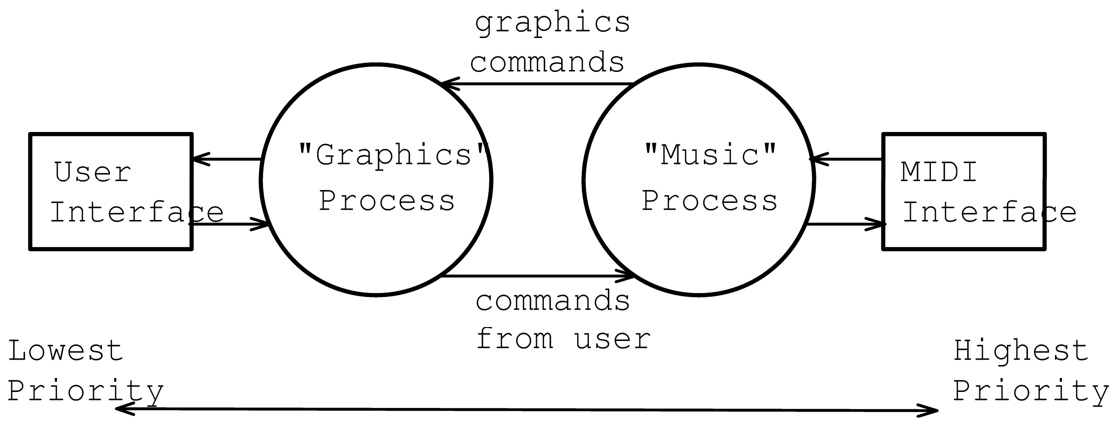

Figure 7-1: Processes, priorities, and interfaces in the CMU Midi Toolkit.

that is used only temporarily by a program. Execution normally stops while garbage collection takes place. The question of execution speed is raised because executing Lisp programs often involves some amount of interpretation as opposed to the faster technique of running compiled programs. The music process is written in C for speed and to avoid garbage collection. The graphics program is written in an interpreted Lisp which occasionally stops for garbage collection.

To avoid speed problems with the Lisp interpreter I have made extensive use of graphics and user interface primitives provided by the operating system, and in some cases I have added my own primitives written in C. This leaves relatively little work for the Lisp program, which is therefore quite responsive. Garbage collection is fairly fast (about 1 second) since the program is small, and this is not usually noticeable because time-critical performance input data is read (via Midi) directly by the music program. This approach would be unsuitable for continuously reading the mouse position and updating some music parameter, but it works well for discrete events caused by buttons, switches, and sliders (the system ignores intermediate slider positions while the slider is being moved).

Others have avoided the garbage collection problem by explicitly freeing garbage or by using only staticly allocated structures. The first of these approaches is error-prone and the second significantly reduce the benefits of using Lisp in the first place. This led to my decision to use Lisp only for the less time-critical user interface code where Lisp seemed to have a real advantage over C and where garbage collection was tolerable.

The music process can also send messages to the lower priority graphics process. Currently, these are all requests to perform graphics operations. Sending graphics operations to a lower-priority process allows the music process to respond more rapidly to time critical Midi input. Note, however, that the graphics operations can fall behind the music if too much processing time is taken by the music process.

Midi input and output is interrupt-driven and runs at the highest priority. Midi data is transmitted to the music process through yet another message interface in keeping with the recommendations.

This particular implementation runs on a Commodore Amiga computer with 1 MByte of memory. The Amiga was chosen because its operating system provides multiple tasks with priority-based scheduling.

8. An Example

To illustrate how this all works in practice, I will describe the implementation of the final section of ‘‘Assuming that you wish to dump data from module 1 ...’’, a work for live trumpet, computer, and synthesizer. The music of the final section is played by four independent voices, each of which plays in a different octave. The first voice plays a note every 5 beats, the second plays every 7 beats, the third plays every 9 beats, and the fourth plays every 11 beats. This creates a constantly changing pattern. Each note is accompanied by the appearance of an image which then fades to blackness. The code to implement these voices is the following:

```
do_voice(i, multiplier)
{
    /* turn off previous note */
    midi_note(basechan+i, pitches[i], 0);
    if (!running) return;        /* stop */
    /* get a new pitch only if a new note was played */
    if (note_count > last_count) {
        pitches[i] = 24 + (12 * i) + note_pitch % 12;
    }
    last_count = note_count;
    /* give time for decay, then play note: */
    cause(10, midi_note, basechan+i, pitches[i], vel);
    cause(11, new_figure, i, rate * multiplier);
    cause(rate * multiplier, do_voice, i, multiplier);
}
```

This `do_voice` routine is invoked once for each voice, and each execution of `do_voice` uses `cause` to schedule another execution for the next note of that voice. The choice of pitch class for each voice is based on pitches played by the trumpet. Every time a new trumpet pitch arrives, it is saved by the input handler routine (not shown) into a global variable (`note_pitch`), and a counter called `note_count` is incremented. The next time `do_voice` plays a note, it looks at `note_count` and compares it to `last_count`, the value of the `note_count` last seen when a voice sounded. If the note is new (`note_count > last_count`), then the voice takes the pitch class of the trumpet note in an octave based on the voice number (the parameter `i`).

Thus each trumpet pitch can be used by at most one voice. This gives the effect of selecting and echoing pitches performed on trumpet. The code example shows how each execution of `do_voice` turns off the currently sounding note, tests to see if the performance has ended by testing a global variable, calculates a new pitch, plays the note via Midi, computes an image (by causing `new_figure`) to coincide with the performance of the note, and finally schedules another call to `do_voice` to end this note and start a new one.

The graphics routine `new_figure` finds a location for an image, generates an irregular image and draws the image on the screen. The image is a solid color that corresponds to a particular entry in a (hardware) color table. To create the fade to black, the color table entry is modified in 16 successive steps. Each step corresponds to an event scheduled by `cause`, and each step event schedules the next step, just as each execution of `do_voice` causes the next one. When the color has faded to black, the image is redrawn using the background color (also black). This step is invisible, but it frees the color table entry just in time for reuse in conjunction with the sounding of a new note.

9. Evaluation

No controlled study has been made to test these recommendations in a scientific manner; in fact, an unbiased study would be very difficult to design because programming methodologies tend to influence software design.

Another way to evaluate a methodology is to see how it is used. The CMU Midi Toolkit, which embodies most of these recommendations has been in use for several years. It appears that this software has enabled at least several composers and a number of students to create programs which would not have been possible otherwise [5, 7, 10]. Some Lisp-based Midi software, which provides an alternative manifestation of many of these same design principles, has been similarly successful [2, 3].

Of these recommendations, the use of multiple processes and interface design tools are the ones with which I have the least experience. These were implemented as tools for a recent work involving interactive music and graphics generation. As it turned out, I have never bothered to raise the priority of the music processing over graphics processing, so at least in this case, the extra work to offload graphics to another process was unnecessary. The control panel design software was not as useful as expected. Most of the changes and adjustments required reprogramming in addition to simply changing the values of some parameters or calling existing routines, so having a reconfigureable control panel in conjunction with a compiled real-time system did not eliminate compilations. Also, since the screen was normally occupied by graphics output from the program, it was awkward to also use the screen for a control panel. Nevertheless, with some additional improvements to make this software easier to use, I believe it will become an essential tool for future program development.

Two areas which have not been addressed are dealing with continuous controls and reducing the computation time required for interesting animation. The event-oriented strategies described here do not make it convenient to deal with continuous data such as amplitude envelopes or trajectories of animated objects. Arctic [9] and FORMES [11] are languages designed specifically for this kind of computation, but neither language has a real-time implementation yet. This is an important area for future research.

Computing images in real-time requires a combination of special-purpose hardware, fast processors, and various image- and program-specific tricks. As with sound generation, the most advanced image generation techniques do not run in real-time, and the graphics community is actively developing new hardware and software techniques to get better and faster results. Regardless of the approach, the techniques presented here are useful for the generation of control information in real time. Describing all the possibilities for real-time sound generation and image production in response to this control information is beyond the scope of this paper.

10. Conclusions

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.

11. Acknowledgments

I would like to thank Cherry Lane Technologies, Commodore-Amiga, Inc., Yamaha, and the Computer Science and Music Departments of Carnegie Mellon for supporting the various projects which led to the ideas presented here.

References

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

[2] Boynton, L., P. Lavoie, Y. Orlarey, C. Rueda and David Wessel.
MIDI-LISP: A Lisp Based Music Programming Environment for the Macintosh.
In P. Berg (editor), *Proceedings of the International Computer Music Conference 1986*, pages 183-186. International Computer Music Association, 1986.

[3] 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.

[4] Buxton, William.
Lexical and pragmatic considerations of input structures.
Computer Graphics 17(1):31-36, 1983.

[5] Chabot, Xavier, Roger Dannenberg, and Georges Bloch.
A Workstation in Live Performance: Composed Improvisation.
In P. Berg (editor), *Proceedings of the International Computer Music Conference 1986*, pages 57-59. International Computer Music Association, 1986.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Pennycook, Bruce W.
PRAESCIO-II: AMNESIA Toward Dynamic Tapeless Performance.
In C. Lischka and J. Fritsch (editor), *Proceedings of the 14th International Computer Music Conference*, pages 383-391. International Computer Music Association, 1988.

[11] Rodet, X. and P. Cointe.
FORMES: Composition and Scheduling of Processes.
Computer Music Journal 8(3):32-50, Fall, 1984.