Listen to Your Hack

Bernard Bernstein and Chris DiGiano

Department of Computer Science
University of Colorado at Boulder
Boulder, Colorado 80309-0430
bernard@cs.colorado.edu
digi@cs.colorado.edu

Abstract

When writing code in an event-oriented environment, many things may happen at any given time. Debugging environments provide the programmer feedback about the execution of the code, but the information is often limited to textual snapshots of the program state. With the use of audio debugging points, one can "listen" to the code as it executes and hear when something wrong happens. This paper describes what a sonic debugger might sound like and how it would work.

Introduction

In the 1960's intrepid hardware hackers rigged a speaker to some of the machine's hardware registers. Programmers used the sounds emitted from the speaker to monitor their code as it compiled and executed. Those with enough experience could identify compile stages by their sounds and distinguish between good and bad sounds for their running code. Today it seems we have all but forgotten those computer audio pioneers. Debugging is now almost exclusively a textual activity, relegating the speaker to arcade games and SysBeeps.

In most respects today's debugging techniques are far more sophisticated than those of thirty years ago. Debuggers provide useful snapshots of the state of the machine at various points during the execution of a program. Where traditional debuggers fail is in monitoring the ongoing activities of executing code. Debuggers provide little overall sense of how the program is progressing. A profiler provides feedback about what sections of code were executed, but this is only available as a post-mortem dump.

By contrast, when driving an automobile we are almost bombarded by a constant stream of feedback pertaining to the performance of the vehicle. Much of this information takes the form of sound. If a "klunk" or a "digadigadiga" or a "ticktickick" occurs, the driver knows immediately something is wrong with the car. When a good mechanic hears the sound, he or she can often identify the exact problem.

Audio feedback has the potential of being equally valuable for software developers. With sound, a programmer might be able to identify when the code is "puttering along" correctly, or when something seems wrong about the program.

This paper explores the idea of a low level sonic debugger to facilitate Macintosh hacking. We first postulate what a program would sound like using a sonic debugger. We then describe some specific examples of when a sonic debugger could help programmers identify otherwise hard to locate problems. The paper presents a list of some features that a useful sonic debugger should have, and concludes by proposing some methods of hacking a sonic debugger.

Previous Work

Earlier work with sound and programming suggests that audio indeed has an important role to play in the software development cycle. Jackson and Francioni (1992) demonstrated that carefully designed "auralizations" of parallel programs could improve programmers' understanding of an algorithm and even help identify problems. Brown and Hershberger (1992) produced a meaningful sound track for animations of algorithms which highlighted the power of the combined modalities. Finally, user studies of LogoMedia (DiGiano, 1992) indicated that sound could in fact be used to find bugs in programs and verify fixes.

The Procedural Orchestra

Audio feedback is useful for listening to repetitive patterns. Our ears are extremely sensitive to repetitive sounds and can recognize an imperfection easily.

Continuous iterations of the event loop offer one useful source of rhythmic patterns in any Macintosh program. Consider the possibilities if each event loop iteration forms a measure in the musical score of a program. The down-beat will be the WaitNextEvent call. Of course, WaitNextEvent gives time to other processes in the machine, so there will be a possibility of confounding sonic events. If the user would prefer not to hear these extraneous sounds, then the sonic debugger will need to be deactivated before the call to WaitNextEvent and reactivated afterward.

After the event is received, several things can happen in the next phase of the musical measure. In the case of an update event, a potentially huge number of Quickdraw calls may be executed. We will call this the Update Symphony (in D minor). During this symphony, the windows will be redrawn with their UpdateWindow commands, which will, in turn, call many Quickdraw functions to draw the windows. In addition, the program-specific drawing will occur with another potentially large number of calls. Normally, many of the Quickdraw calls are quite fast, because they may be asked to draw in regions that are still "valid" (not asked to be redrawn). In those cases, one may hear many more functions get called than are necessarily causing actions.

In the case of a mouse-down event, something will happen to a window or a menu, or perhaps the context will switch to one of the background applications. While the mouse is down, an interlude will be played which is under the user's direction. We will call this user solo the Cadenza. This user solo will likely have some Quickdraw calls giving the user feedback in some graphical way to the status of the user's action. For example, if a menu is selected, then the Menu Manager will be actively displaying and hilighting the various menu items while the cursor is over them.

When the mouse is released, something more complex will probably happen as a result of the mouse operation, such as a menu selection. This will be the full orchestra pouncing on the score immediately following the user solo. We hear the standard event loop rhythm return, most likely beginning with the Update Symphony.

By listening to the execution, the programmer will become familiar with certain auditory patterns associated with sections of code. It is this familiarity with the sounds that gives the listener the ability to distinguish between correct and incorrect execution of the code.

What Sounds Can Help Identify

Using a sonic debugger, one can listen for patterns and inconsistencies in patterns. A common sequence is generated by the initialization, execution and cleanup of a function. If a function is missing any of these characteristic sound elements, then there may be something wrong with it.

A common pair of sounds to listen for are allocation and deallocation of memory. If a function allocates memory, then there should be some matching deallocation of the same block somewhere in the program or else a memory leak may exist. Alternately, allocation can trigger a sustained non-obtrusive background sound which can only be turned off by its matching deallocation. Memory leaks would stand out as sounds that continue to play after the program should have deallocated all memory. When programming in an object-oriented language such as C++, one may ask the debugger to begin a sound with an object's constructor and stop it when its destructor is executed.

Another class of sounds one can listen for are functions that are called at the wrong time or the wrong number of times. For example, when drawing takes place, it is common for a program to draw shapes multiple times when it is only necessary to draw something once. This can impede the performance of the program since the drawing is sometimes a bottleneck in the speed. The repeated auditory signature of a function could make this problem explicit. Thus, an improper calling of a function can have the equivalent effect as a “klunk” in a car engine.

Continuous sonic debugging means that even if the machine bombs because of an error in their code, hackers still have auditory impressions of the execution trace in their own human memory. This may be the only record of the events leading up to the crash if the failure corrupted the stack.

We cannot expect sounds to accurately convey absolute information such as just how much memory was allocated or exactly what parameter were passed to a function. Nonetheless, sonic debugging offers a rich new set of methods for observing general behavior. It may take time to become sonically aware of your code, but once the auditory associations are mastered, its feedback can likely increase the speed of finding certain bugs within code.

Since our goal is to find bugs, and not to spend time learning how to use the debugger, we need to suggest some features that would make the sonic debugger practical.

Features of a Sonic Debugger

A sonic debugger should be able to trigger sound commands for most A-Traps in the Macintosh system as well as for any location within a program. Sound commands can start or stop sounds, or adjust a sound’s properties such as pitch, volume, or duration.

A sonic debugger should allow audio points to be arbitrarily enabled or disabled to allow programmers to focus their listening attention. The user should be able to restrict sound generation to particular functions or specific operating system managers. For example, a user may only want to hear sounds for Memory Manager calls, or perhaps just for NewHandle and DisposeHandle. This will allow hackers to debug and optimize specific components of the program.

A serious concern is the cacophony which might result if every trap in the Macintosh were continually associated with sound. The hacker would not know if the sounds were coming from his or her program or from another process. WaitNextEvent, therefore, may need to disable the auditory feedback before other processes are given time, and enable them when the program continues.

Since the production of the sound may interfere with the execution of the program, an unobtrusive way of generating sound would be to send it to an external processor. A solution to this problem would be to send messages to an external Macintosh or to a Musical Instrument Digital Interface (MIDI) (IMA, 1983) instrument. All three systems mentioned in the introduction use MIDI for sound generation.

Sound events used by a sonic debugger should be triggered synchronously by the real-time execution of code. Naturally, this will cause the execution of code to be slowed. But, the advantage is that the sounds are heard in context with the appropriate sections of the actual running program.

The programmer would have the choice of whether to pause execution while a sound is played or else simply wait long enough for a sound command to be issued. Pausing execution would allow sounds of arbitrary length to be heard without interference from other audio points. This would facilitate non-musical sampled sounds and even synthetic speech. Voice could be used to say the name of the function being executed, or perhaps a message like “allocating handle” can be uttered by the machine.

These features would make the system feasible, but we still have not discussed what the user interface to the debugger will be.

Making the Sonic Debugger

A sonic debugger could take the form of a Macintosh control panel which maintains a list of traps. Each trap could be associated with any sound. Each trap could also have its sound enabled or disabled.

Another method would involve a `dcmd` for MacsBug or the equivalent in another debugger. Like the control panel, the `dcmd` could assign sounds to traps. But, in addition, it could assign sounds to specific memory addresses so that when the address was executed, the debugger could generate sound commands. An additional benefit of this method is that programmers could provide custom auditory associations for specific parts of their code using calls to the debugger extensions to make the sounds.

The common thread between both the control panel and `dcmd` approaches is that the sonic debugger needs to patch every trap that may trigger a debugging sound command. The patch must send a message to a sound player module which dispatches the appropriate sound command.

The sound module could either pass the appropriate commands to a MIDI device, or it could generate the sounds itself. We already discussed the problems with generating the sounds on the same machine as the debugger, so we will most likely use a MIDI processor to execute the auditory feedback.

Conclusions

Using traditional Macintosh debuggers we are limited to a single textual feedback mode. This representation allows us to see a fine grained representation of the current state of the machine and how it changes into another state. With the added dimension of sound, we gain several benefits. We obtain feedback at a high level, from which programmers can listen to any operation, and one can use the additional sense of audition to experience their programs execution.

Sound expands our repertoire of debugging techniques in new and exciting ways. Unlike meaningless textual display points which might fly across the screen, audio points can provide an acoustic gestalt, or overview of the program execution. Problems such as excess redraws and memory leaks can suddenly become apparent through auditory feedback.

By combining traditional textual methods with sonic debugging we can only expect hacking productivity to increase. The Macintosh is a multimedia machine. Let’s take advantage of its capacity for sound to elevate the art of hacking to a new level.

References

Brown, Marc H., and John Hershberger (1992). Color and Sound in Algorithm Animation. *IEEE Computer* 25 (12): 52-63.

DiGiano, Christopher J. (1992). Visualizing Program Behavior Using Non-Speech Audio. MSc. Thesis, University of Toronto.

IMA (1983). *Musical Instrument Digital Interface Specification 1.0*. IMA.

Jackson, Jay Alan, and Joan M. Francioni (1992). Aural Signatures of Parallel Programs. *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, 218-229.