Behavior Research Methods, Instruments, & Computers
2001, 33 (2), 174-178

Millisecond timing on PCs and Macs

W. JOSEPH MACINNES and TRACY L. TAYLOR
Dalhousie University, Halifax, Nova Scotia, Canada

A real-time, object-oriented solution for displaying stimuli on Windows 95/98, MacOS and Linux platforms is presented. The program, written in C++, utilizes a special-purpose window class (GLWindow), OpenGL, and 32-bit graphics acceleration; it avoids display timing uncertainty by substituting the new window class for the default window code for each system. We report the outcome of tests for real-time capability across PC and Mac platforms running a variety of operating systems. The test program, which can be used as a shell for programming real-time experiments and testing specific processors, is available at http://www.cs.dal.ca/~macinnwj. We propose to provide researchers with a sense of the usefulness of our program, highlight the ability of many multitasking environments to achieve real time, as well as caution users about systems that may not achieve real time, even under optimal conditions.

As operating systems progress to match faster hardware and higher user expectations, they seem to become less compatible with the needs of experimenters whose research requires real-time capabilities. Although some researchers have stated that this is a reason to continue utilizing nonmultitasking environments (e.g., DOS; Myors, 1999), this strategy prevents the use of many advances, such as 32-bit graphic libraries and video hardware acceleration. Furthermore, although graphic libraries exist for DOS, most displays must be written directly to the video card (often in assembly language) to achieve the performance required of experimental software. By contrast, and owing in part to the popularity of computer games, 32-bit graphical libraries for Windows (such as OpenGL) have become viable, offering an impressive combination of fast speed, easy programming, and high quality displays. OpenGL can be used to create experimental displays that would have needed specialized equipment a few years ago. These displays range from simple two-dimensional presentations to complex three-dimensional (3-D) representations (see, e.g., Shore, Stanford, MacInnes, Klein, & Brown, 2001). Moreover, new video cards are being released without any thought given to DOS drivers or support, leaving the DOS programmer to fall back to the lowest common denominator—VGA. This denies the use of any of the advanced features that are commonly found on even the least expensive of today's video cards.

Software is quickly becoming another problem for researchers using DOS systems. Newer (usually better) software packages will not run on DOS systems, and some new compilers (e.g., Microsoft Visual C) can no longer compile DOS programs. Other compilers (e.g., Metrowerks CodeWarrior) offer advanced graphical support for Windows' conventions such as MMX and AMD's 3DNOW!. Working on a Windows system and experimenting on a DOS system can lead to other compatibility issues related to the need to convert to short file names for DOS and the use of different file systems and default palettes.

Despite the difficulties associated with using nonmultitasking environments, they have sometimes been favored over newer systems owing to concerns over timing issues. Windows 95 and, later, MacOS and Linux are all multitasking operating systems. This means that the operating system is responsible for allocating time to each of the applications. Even if only a single application is running, the operating system will periodically "steal time" from the application to ensure that OS errors do not crash the entire system. Although the degree of multitasking varies between operating systems, *all* will steal time from a program under normal running conditions. So long as this is the case, an experimenter cannot achieve true real time.

Many solutions (both free and commercial) have been proposed to deal with the conundrum of optimizing modern operating systems for real-time operation, but most are expensive, difficult to verify, or fall short of true real time. Some of these solutions require specialized hardware (see, e.g., McKinney, MacCormac, & Welsh-Bohmer, 1999) or center around arguments that time-critical data can be collected without a true real-time system (Hecht, Oesker, Kaiser, Civelek, & Stecker, 1999; Stevenson, Francis, & Kim, 1999). Finally, it is difficult to judge commercial software packages, because many do not advertise real-time statistics and most do not include source code for independent testing. Readers should also be warned

This research was funded in part by the National Science and Engineering Research Council of Canada, the Canadian Space Agency, and Dalhousie University. Thanks to William Schmidt, Steven Finney, and four anonymous reviewers for comments on this and earlier versions of this paper. The testing software used in this article is owned and copyrighted by the first author and is available for free download under the GNU public license. Individuals using this software in published articles are asked to cite the source of the code. Correspondence concerning this article should be addressed to W. J. MacInnes, Department of Psychology, Dalhousie University, Halifax, NS B3H 4J1, Canada (e-mail: macinnwj@cs.dal.ca).

that packages claiming to use millisecond timers are not necessarily claiming real time. This is akin to claiming that prices are measured in pennies without reporting the actual price.

The purpose of this report is to present a software solution that is capable of achieving and testing for real time. This software solution is designed to be easily ported to any system capable of 32-bit (OpenGL) graphics and can be used on a variety of systems, including MacOS, Windows, and Linux. It may be used as a base for developing real-time experiments or used "as is" to test the timing capabilities of experimental machines. To demonstrate the utility of this program as a test for real-time operation and to inform the reader of which modern computer systems are capable of achieving real time, we examined the results of timing tests run on a variety of hardware and software combinations.

METHOD

All software for this timing test was written and compiled using Metrowerks CodeWarrior 5.0 (for Windows and Mac) and a standard "make" file for Linux. Although the code could be easily converted to other compilers, Metrowerks has the advantage of being able to optimize graphics for both MMX and 3DNOW! calling conventions and is available for all three operating systems. The executable included in the PC download is optimized for an AMD (K6-2 or Athlon) computer, but settings are also available for Intel-based CPUs.

Apparatus

OpenGL version 1.1 was used as the graphics library for all tests to take advantage of graphic hardware acceleration and cross-platform compatibility. Performance tests were run on a variety of PCs, ranging from low-end Pentiums with no graphics accelerator to high-speed Athlons with 3DNOW!, MMX, and OpenGL acceleration. Mac systems also ranged from low-end with no acceleration to new systems with OpenGL acceleration. Operating systems included Windows 95/98/2000, MacOS 7.5/8.6/9.0, and Mandrake Linux 7.1. The Linux systems were installed with the XFree86 4.0 X window distribution to take full advantage of any OpenGL hardware acceleration. See Table 1 for a full list of the systems tested.

The testing program, written in C++, utilized a full-screen, OpenGL window. Although code is included for stimulus display and keyboard /mouse responses, these features were not used during the timing tests. For the duration of the timing test, the computer screen was in full-screen mode (a completely black screen). The screen was not updated, and input was not recorded for the duration of the experiment. This was to ensure a measure of system timing that was not confounded with the measurement of delays that were due to display or input devices. The only class that differed between operating systems was that used for creating the window (GLWindow

Table 1
Performance of Machines Tested Using Real-Time Software

System	Number of Excessive Loop Durations	Average Loop Duration (msec)	Average Miss Duration (msec)
Linux			
Athlon 650 Xwindows	51	<0.001	7.04
Athlon 650 'bash'	34	<0.001	6.76
Athlon-Duron 600 Xwindows (with scheduler)	0	<0.001	0.00
Macintosh			
OS7.6-8500	0	0.055	0.00
OS8.1-8500	0	0.059	0.00
OS8.6-G3Tower(1)	165	0.011	2.32
OS8.6-G3Tower(2)	327	0.002	1.41
OS8.6-6500	1,666	0.042	2.13
OS9.0-G4-No Zip	0	0.006	0.00
OS9.0-G4-With Zip	0	0.006	0.00
OS9.0-G3Desktop	1,666	0.019	1.72
PC-Win2K			
W2K-Athlon-Duron600	0	0.002	0.00
W2K-K6-2-500	0	0.009	0.00
W2K-P133	0	0.013	0.00
W2K-Celeron400	ERR*	ERR*	ERR*
W2K-Athlon500	ERR*	ERR*	ERR*
W2K-Athlon650	ERR*	ERR*	ERR*
PC-Win95/98			
W98-Athlon500	0	0.007	0.00
W98-K62-350	0	0.006	0.00
W95-K62-400	0	0.005	0.00
W95-P125	0	0.007	0.00
W95-P166/MMX	0	0.006	0.00
W98-PII-233w 2 video cards	1	0.006	3.85
W98-PII-233w 1 video card	0	0.006	0.00
W98-Athlon650	0	0.006	0.00

Note—Number of excessive loop durations refers to the total number of loops that exceeded 1 msec. Average miss duration is the average length of those excessive loops. Average loop duration is the length of all loops, including those that were missed. All results are generated by the sample program in the file time.dat. ERR* refers to Windows 2000 systems that were unable to complete the test owing to a bug in the multimedia timer.

class). Functions used in the GetTime( ) method of this class were the following: Linux, gettimeofday( ); Mac, Seconds( ); and PC, QueryPerformanceCounter( ). All other code was platform independent.

Procedure

All timing tests were accomplished using a simple loop that did nothing but track the time between iterations by calling GetTime( ) from the window class (see below).

By simulating 500 discrete trials with a duration of 10 sec each, each machine underwent over 83 min of continuous testing. A "sample" was defined as the "LoopDuration" of a single iteration of the timing loop, and a missed sample was any single iteration that took longer than 1 msec to complete. An "ideal" system, would, therefore, complete this entire test without a loop duration exceeding 1 msec. Even though 1 msec is the maximum allowable loop, the average loop duration should be small enough (a fraction of our 1-msec maximum) to allow for the code and calculations that typically fall within a typical experiment's timing loop. All errors were buffered in memory and written to the file at the end of the experiment.

It is important to note that this was a customized timing loop, and not the messaging loop commonly used with standard "window" programming. Message loops on Windows, Mac, and Linux allow the operating system to pass information between the hardware and the individual applications, as well as between applications. Because the standard messaging loop was bypassed on all of these platforms, alternative ways were found to interact with such devices as the keyboard, the mouse, and the display.

To facilitate cross-platform programming, all of the hardware-specific code was kept within the window class (GLWindow). Window setup, rendering, timing, and input were all controlled through methods in this class. Changing the program to support new operating systems meant simply changing the code internal to GLWindow and leaving the method names the same; the remainder of the code needed no modification.

Wherever possible, the priority of the experiment was boosted to the maximum supported by that system. SetPriorityClass(h, REALTIME_PRIORITY_CLASS) (Microsoft Developer Network On-Line Library, 2001) was used for all the Windows systems. For the Linux system, SetPriority(-20) was used, and then, prompted by success in achieving real time reported by Finney (2001), sched_setscheduler( ) was added to improve the Linux test. For MacOS, there is currently no official priority setting. Although there are implementations (Pelli, 2001) of an older, unsupported, priority call, this was not used in these tests. Since the code is not supported by Mac and is not PowerPC native, there was no guarantee of how it would perform on current and future systems. Macs, however, were run without nonessential extensions and with virtual memory disabled.

It should be noted that although real-time priority (process priority, in particular) in Windows can have a large effect on performance, it does so at the expense of all other processes, *including* the operating system. Although real-time priority does not completely shut out the operating system, it will completely shut out the messaging loop, as well as cause the disk cache not to flush immediately. When in doubt, researchers should consider using HIGH_PRIORITY_CLASS or ABOVE_NORMAL_PRIORITY_CLASS, both of which are higher priority than most other programs, but not the operating system.

RESULTS AND DISCUSSION

Table 1 summarizes the outcome of the timing tests that were performed. The number of excessive loop durations refers to the total number of loops that exceeded 1 msec, the average miss duration is the average time (in milliseconds) of those missed loops, and the average loop duration is the average length of all the loops, including those that were missed. This information is the same as researchers will find in the time.dat file produced by the downloadable software http://www.cs.dal.ca/~macinnwj.

Windows

All of the PC systems achieved true real time while running Windows 9x. The only exception was a Pentium II, which consistently had a loop duration that exceeded a millisecond within the first second of the experiment. This was likely due to the overhead associated with the use of two video cards for graphics acceleration; no single-card solution had this problem, including the same system with the second card removed. All other tests on Windows 9x PCs completed the entire 83 min (500 iterations of 10,000 msec) without skipping a single millisecond.

The results for Windows 2000 were not as clear-cut. Some systems were capable of consistent real-time performance, whereas others behaved as if the drivers were incomplete or contained bugs. Both low-end Pentium systems (P133 and a PII 233), an Athlon 650, and an Athlon (Duron) 600 achieved real time without a single loop duration exceeding 1 msec. An Athlon 500 system, however, had consistent long loops every 10 sec, and a Celeron 400 had compatibility problems with the multimedia timer that was used. The Celeron finished the entire 83-min experiment in under 20 sec, despite the fact that the timer and compatibility test provided by Microsoft for this timer returned positive results. In an attempt to determine whether this represented a bug in Windows 2000, the latest service pack[1] was installed on the Celeron 400, the Athlon 500, and the Athlon 650. This caused all three machines to report incorrect times with the multimedia timer, despite passing the compatibility test. Although it seems possible to achieve true real time on Windows 2000, tests on all machines will not be conclusive until bug-free drivers are available.

One of the most critical time-bandits in Windows, surprisingly, was displaying the mouse cursor. Failing to disable the cursor display actually had a greater effect on timing than setting the priority. Although it was possible to achieve real time without the cursor in normal priority,

```
CurrentTime := GetTime( )                    //Set up Timer; resolution = 1 msec//
FinalTime := CurrentTime + 10,000            //10 sec per test//
PreviousTime := CurrentTime
While(FinalTime—CurrentTime > 0)
        CurrentTime := GetTime( )
        LoopDuration[i]:= CurrentTime—PreviousTime    //save loop duration for test//
        [Test to see if LoopDuration < 1 ms, and report errors]
        PreviousTime := CurrentTime
        i := i + 1
```

samples were lost with the standard cursor displayed even on real-time priority.

Macintosh

Results for Mac systems also varied according to a number of factors. Machine type, operating system, and extension set played a large role in determining the outcome of the timing tests. Some systems were capable of producing real time, whereas others were not. Every system tested with Version 7 of the operating system produced real time without any difficulty; some systems tested with OS 8.x achieved real time, whereas others did not; and finally System 9.0 achieved real time on G4 machines, but not on any other. It should be noted that the speed of the machine does not always determine performance. For example, an 8500 running System 8.1 achieved real time, whereas a G3 running System 8.6 did not.

It is interesting to note that the extensions on a given machine had a large impact on the performance in these timing tests. By disabling the internal Zip drive alone, most of the missed samples were eliminated (all the tests in Table 1 are *after* the Zip was disabled, unless otherwise stated). In fact, only the G4s tested (OS 9.0) were able to achieve real time with the Zip drive enabled. All times reported for Macs in this paper are for tests run with minimal system extensions and with virtual memory disabled. On a final note, it is difficult to separate the performance of the G4 from that of OS9. Machine and operating system were designed to work together, and although OS9 did not achieve real time on a G3, it is impossible to test a G4 on an earlier operating system.

Linux

None of the original Linux systems examined was capable of pure real time, using this test. Although the Xwindows code presented is not quite ready for full experimentation (mouse cursor can only be hidden, not removed, and Xwindows must be restarted after running), it is sufficient for real-time tests (see Finney, 2001, for another solution to real-time data acquisition and control on Linux). To ensure that time loss was not due to improper implementation, a similar timing loop was run without Xwindows, using only the Bourne Again Shell (bash)—the Linux command line DOS equivalent. Although bash did perform better than the Xwindows implementation, it was also incapable of performing the test without loop durations exceeding 1 msec. Finney has also measured loop performance in Linux and has also been able to improve this performance by setting the scheduling priority, in addition to the methods presented in this paper. He shows that when the sched_setschedular call in Linux is used to give a process high priority, Linux provides timing resolution (100 μsec or better) that is more than sufficient for millisecond-level control. This result was replicated by adding the scheduling calls to this timing test (see Table 1, Athlon-Duron 600 Xwindows [with scheduler]).

IMPLEMENTATION AND EXTENDABILITY

The program used to run the timing tests is available at http://www.cs.dal.ca/~macinnwj and is currently capable of system timing tests on a variety of machines. The executable files included with the downloads should be entirely compatible with AMD (Windows), PowerPC (MacOS), and Kernel 2.2.16 (Linux). Other systems (Intel based PCs, etc.) should be able to use the software after a simple recompilation of the included source code.[2]

The code is written to run 500 trials. For a shorter test, researchers may change the first number in the included stimulus file (stim.in) from 500 to any number of 10-sec loops. Timing results from the program are output to a file called time.dat. Each line of the output file represents a summary of the data from a 10-sec loop. Output includes the number of samples that exceeded 1 msec in duration, the total number of loops that were run in each 10-sec trial, the sample frequency, the average loop duration, and the average time (in milliseconds) of those loop durations that exceeded 1 msec.

The software package can also be used as a base for developing fully-functioning experiments.[3] A simple demonstration experiment[4] is included in the code and can be activated by changing a few comments before recompiling. The method Capp::Run() in the file App.cpp contains the main running loop of the code. The loop that begins with while(((CurrTime... is executed once in each 10-sec trial and contains code to test the timing, as well as code to display two 3-D wire frame objects. A researcher may simply comment out the timing code and uncomment the drawing code to get a sample experiment that tests the additive effect of various 3-D cues (see note 4). As with the timing test, the first number in the file stim.in may be changed to alter the number of trials that the program will run. Each trial awaits a response from 1 to 10 on the computer keyboard, representing the relative difference in depth between the two objects (see note 4) that are presented. Results from the trials are output to the file stim.out.

The following is a short description of the classes included in this project and of the program flow. It should be noted that this is not meant to be an introduction to object-oriented or OpenGL programming; it is intended only to demonstrate the use of these principles in this particular application. GLWindow.cpp is responsible for the creation of the full-screen window, as well as for keyboard input and tracking system time. All platform-specific code is encapsulated in this class for greater portability. Main.cpp is not a class, but an entry point for the program (WinMain). It has only one purpose: to create a single instance of the application class (Capp) and ask it to run. Note that the windows messaging loop is not entered until *after* the entire program has finished running. The application class (App.cpp) controls the majority of the program flow; it has methods to load data from input files and save data to output files, as well as

the main program loop. The trial class (Trial.cpp) is an abstract class that defines the behavior of any class that inherits from it. A number of such classes can be found in Depthtrial.cpp and reflect the different types of trials that are available in this sample experiment.

Program flow is as follows. The application class, through its Run( ) method, reads in the stimulus file (stim.in) and begins filling an array of trials (Tlist[]) with instances from the classes found in Depthtrial.cpp. This selection is currently random, based on the percentages found in the input file. Because all of these classes inherit from the Trial class, they are interchangeable in this array and are capable of displaying completely different visuals through their Render( ) method. Tlist[i]-> Render( ) is called before the timing loop and renders the appropriate display for that trial to an off-screen buffer. The timing loop begins and monitors two possible events: (1) a keyboard response, which ends the trial, or (2) onset time for the trial display, which tells the window class to swap the off-screen buffer to the front display. This continues for the preprogrammed number of trials.

CONCLUSION

All the machines, regardless of speed and graphics card, were able to maintain an average of about 100 loop iterations per millisecond (slightly less for some Macs). Of course, the most basic experiments will do much more than check a timer in the main loop, but with loop overhead this low, there is plenty of room to add whatever calculations are needed. In addition, Windows 9x and Linux (after modifications) were able to maintain optimum performance without losing any time to the operating system. Mac, although not true real time under all conditions, still managed to keep the time loss to a minimum. Finally, it is important to realize that every combination of hardware, operating system, and experimental software is potentially different and must be tested to ensure that it meets the needs of the experimenter. A capacity for real time does not guarantee real-time performance, as our results demonstrate. It is the responsibility of the researcher to perform real-time tests on their systems and of the makers of experimental software to include such tests with their software.

REFERENCES

Finney, S. A. (2001). Real-time data collection in Linux: A case study. *Behavior Research Methods, Instruments, & Computers*, **33**, 167-173.

Hecht, H., Oesker, M., Kaiser, A., Civelek, H., & Stecker, T. (1999). A perception experiment with time-critical graphics animation on the World-Wide Web. *Behavior Research Methods, Instruments, & Computers*, **31**, 439-445.

McKinney, C. J., MacCormac, E. R., & Welsh-Bohmer, K. A. (1999). Hardware and software for tachistoscopy: How to make accurate measurements on any PC utilizing the Microsoft Windows operating system. *Behavior Research Methods, Instruments, & Computers*, **31**, 129-136.

Microsoft Developer Network On-Line Library [On-line]. Available: http://msdn.microsoft.com/library/default.asp. (Retrieved March 1, 2001)

Microsoft Windows 2000 Service Pack 1 Download [On-line] Available: http://www.microsoft.com/windows2000/downloads/recommended/sp1/default.asp. (Retrieved March 1, 2001)

Myors, B. (1999). Timing accuracy of PC programs running under DOS and Windows. *Behavior Research Methods, Instruments, & Computers*, **31**, 322-328.

Pelli, D., *Video Toolbox* [on-line]. Available: http://vision.nyu.edu/VideoToolbox/. (Retrieved March 1, 2001)

Shore, D. I., Stanford, L., MacInnes, W. J., Klein, R. M., & Brown, R. E. (2001). Of mice and men: Using virtual Hebb–Williams mazes to compare learning across gender and species (*Homo sapiens* and *Mus musculus*). *Cognitive, Behavioral, & Affective Neuroscience*, **1**, 83-89.

Stevenson, A. K., Francis, G., & Kim, H. (1999). Java experiments for introductory cognitive psychology courses. *Behavior Research Methods, Instruments, & Computers*, **31**, 99-106.

NOTES

1. Service Pack 1.0 (2001) can be downloaded from the Microsoft Home Page.

2. In CodeWarrior the target processor setting can be found under Edit->Settings->Code Generation->x86 Processor.

3. It should be noted that the following description of the code is not needed to run or understand the timing capabilities of the software. Anyone not interested in modifying the code should skip ahead to the Conclusion section.

4. This experiment is only provided as a sample of what can be accomplished. The authors make no claims as to its scientific validity.

(Manuscript received November 8, 2000;
revision accepted for publication March 21, 2001.)