Behavior Research Methods
2006, 38 (2), 280-286

Tscope: A C library for programming cognitive experiments on the MS Windows platform

MICHAËL STEVENS, JAN LAMMERTYN, FREDERICK VERBRUGGEN,
and ANDRÉ VANDIERENDONCK
Ghent University, Ghent, Belgium

Tscope is a C/C++ programming library designed for programming experiments that run on Windows 2000/XP. It is intended for a public of experimental psychologists with moderate programming skills, who are accustomed to writing their own experimental programs for DOS but have not made the step to Windows-based programming yet. It provides molecular functions for graphics, sound, timing, randomization, and response registration. Together with ANSI-C standard library functions and the powerful C syntax, this set of functions gives the experimenter the opportunity to program virtually any experiment one can come up with. Tscope is completely based on free software, is distributed under the GNU General Public License, and is available at expsy.ugent.be/tscope. An integrated development environment for compiling and running Tscope programs is also freely available.

Due to its multitasking nature, the Windows operating system was initially thought to be unsuitable for running time-critical experiments (Myors, 1998, 1999). This concern was caused by the fact that two or more programs running on a single-processor multitasking system are not truly running in parallel. At any given moment, only one program is running, while the other programs are preempted by the operating system. Thus, what is generally called *multitasking* is achieved by switching back and forth at a fast rate between several running and preempted programs. For the human user, the rate at which this switching occurs is too fast to notice, giving rise to the illusion that the programs are running in parallel. For daily use, this type of multitasking is satisfactory, but for experimenters interested in measuring reaction times, it is a nightmare. Consider the situation in which an experiment program is preempted by the operating system while it is waiting for a participant's response. Chances are virtually nonexistent that the experimenter would notice anything strange at runtime, but the reaction time measured would be biased by an unknown factor. Even when inspecting the data afterward, one would not know which reaction times are measured accurately and which are not. As a consequence, cleaning up the data afterward is not possible either. Given this inability to measure reaction times accurately on Windows systems, most researchers have wisely decided to continue working on nonmultitasking environments, such as DOS.

More recently, it has been argued that these concerns about multitasking operating systems raised by Myors (1998, 1999) are valid only when one is running a program written for DOS on a Windows system (Forster & Forster, 2003; MacInnes & Taylor, 2001). When a DOS program is preempted to allow another program to run, its internal timer will not increment until the program is started again. Hence, the reaction time measured by this program will be biased. This does not necessarily apply to programs specifically written for Windows. First, these programs can make calls to the operating system that increase their priority, making it less likely that the program will be preempted in the middle of its operation. Furthermore, Windows programs do not have to keep track of elapsed time themselves but have access to a high-performance hardware clock chip (Forster & Forster, 2003). This clock chip is located on all Pentium-class motherboards and keeps track of time with submicrosecond accuracy and operates autonomously, regardless of processor load. Therefore, by increasing the program's priority and using the hardware clock chip for time measurement, Windows programs are, in principle, capable of millisecond accurate timing.

MacInnes and Taylor (2001) presented the C function calls needed to increase a program's priority and to read out the clock chip. Both sets of functions are part of the Windows API (Application Programming Interface).[1] These authors further presented a program that could test the capability of a specific computer to achieve millisecond accurate timing. This test program used the hardware clock chip to measure how many times and how long it was preempted by the operating system. Runs on different computers, one of them being a low-end Pentium I 133, showed that the test program was not preempted by the operating system when running with increased prior-

This project was made possible by Grants 011D04503, 011D06102, and 10251101 from the Special Research Fund of Ghent University to the first, third, and fourth authors, respectively. The authors thank the undergraduate experimental psychology students at Ghent University who tested earlier versions of Tscope. Correspondence concerning this article should be addressed to M. Stevens, Department of Experimental Psychology, Ghent University, Henri Dunantlaan 2, B-9000 Ghent, Belgium (e-mail: michael.stevens@ugent.be).

ity. On the basis of this result, it was concluded that the computers used in this test were capable of millisecond accurate timing. Furthermore, external tests have been performed to directly compare reaction times measured by Windows programs with reaction times measured by special purpose hardware timers or timers running on DOS computers. These external tests have confirmed the results of the internal software tests performed by MacInnes and Taylor. Although timing accuracy on DOS is even better, Windows is capable of achieving millisecond accuracy (Chambers & Brown, 2003; De Clercq, Crombez, Buysse, & Roeyers, 2003; McKinney, MacCormac, & Welsh-Bohmer, 1999).

However, it should be noted that the total timing precision also depends on the type of response device used. Although Windows programs have an accurate timer chip at their disposal, a variable delay can arise at the level of the input device. For example, mice and keyboards have internal hardware that scan the input less than once per millisecond (Plant, Hammond, & Whitehouse, 2003). Even though no time will be lost by Windows once the signal from the input device has been received, time will be lost within the input device itself—thus, before the input signal reaches the computer. The only input device tested by Plant et al. that transmitted its input to the computer without adding a variable time delay was a response box connected to the parallel port of the computer.

In summary, previous research has shown that Windows programs are capable of achieving satisfactory timing precision, provided that the right precautions are taken. As a consequence, one would expect that most experimenters would have switched to Windows programming by now. Our experience suggests that this is hardly the case. Some experimenters are still running their programs on DOS. Others have moved to Windows but use commercial or noncommercial experiment generators, such as E-Prime (Psychology Software Tools, n.d.) or DMDX (Forster & Forster, 2003). Only a handful of experimenters have adopted Windows programming by now. This is an unfortunate situation, since both DOS and experiment generators have their drawbacks. DOS is not included with recent Windows versions anymore, and the development of DOS drivers for many hardware cards has stopped (see MacInnes & Taylor, 2001, for an elaborate discussion of this topic). Experiment generators also have their specific drawbacks. DMDX, for example, is targeted mainly at language-processing experiments. E-prime supports several types of experiments but is quite an expensive system. Students have access only to a restricted trial version. Dixon (1991) has discussed the pros and cons of programming and using experiment generators in more detail. The main difference to be noted here is that programming languages are more flexible but are also more difficult to learn, whereas experiment generators are easier to work with but often lack flexibility. More often than not, an experimenter will find himself in a situation in which some critical manipulation of his experiment is not supported by his experiment generator of choice. In such situations, most experimenters still need to go back to good old DOS and implement an algorithm that can carry out their manipulation or, even worse, change their manipulation in a way that makes it converge with the experiment generator's possibilities.

The need for experimental psychologists to program their experiments using general purpose languages clearly remains, but for the moment not many experimenters seem to be capable of writing Windows-targeted programs. Although MacInnes and Taylor (2001) have pointed out the functions necessary for programming time-critical experiments on Windows, this has not been the trigger that made the majority of experimenters switch from DOS programming to Windows programming. To achieve this switch, the example programs presented by MacInnes and Taylor were lacking in a number of ways. First of all, the examples were written in C++, the object-oriented descendant of the procedural C language. Although C++ is a logical extension of C, it is not widespread in the experimental community. C itself is not the easiest language to learn; the object-oriented syntax of C++ only adds to this. Furthermore, although object-oriented programming is very useful for developing highly interactive user interfaces, programming an experiment is generally a problem that calls for a procedural programming technique. All events in a typical experiment happen in a relatively fixed succession; that is, an experiment consists of fixed procedures. The extra level of abstraction provided by object orientation thus leads only to overabstraction. A second property of the examples that made them too difficult for many experimenters was the use of threading. This means that within the program, several subprocesses are running in parallel: one responsible for timing, one responsible for the graphics screen, and so forth. This is clearly a technique that would stun a nonprofessional programmer who has just migrated from the nonmultitasking DOS system. A third shortcoming of the examples provided by MacInnes and Taylor was that these authors showed only the function calls needed for timing and for opening a graphics screen. This is hardly enough for programming a complete experiment. To make Windows programming more accessible to the average experimenter, one would also have to provide functions for putting stimuli on the screen or on the speakers and for randomizing lists of stimuli. Finally, the examples did not show how to read input from the parallel port or game port of a computer. They showed only how to get input from the mouse or joystick, two devices known not to be the most accurate (Plant et al., 2003; Shimizu, 2002).

Given this state of affairs, we set out to write a programming library that would meet the following criteria. (1) It should run on recent Windows systems. (2) It should provide millisecond accurate timers and give access to millisecond accurate input devices, such as response boxes attached to the game port or parallel port. Control of stimulus onset should also be guaranteed. (3) It should provide a full set of functions needed by experimental psychologists, including randomizers, graphics, and sound

functions. (4) The syntax of these functions should be as simple as possible; thus, they should not rely on C++ and should have a restricted set of arguments. Threading (forking a program in subprocesses that run in parallel) should be taken care of by the library, not by the programmer. In fact, the programmer should be able to program in the way that he or she was used to on a nonmultitasking system such as DOS. (5) The library should be available at no cost and should come with a freely available, easy-to-use IDE (i.e., an integrated development environment, where the compiler, libraries, and code editor are bundled in a single software package).

The programming library presented here is the result of this effort and meets, in our opinion, the criteria stated above. It is called *Tscope*, after the tachistoscope it is intended to replace. It has no relation with the TSCOP project described in a previous issue of this journal (Norris, 1984). In the following sections, we will describe the software on which Tscope relies and give an overview of the various parts of the library. How the timers are implemented will be discussed in more detail.

Software Used by Tscope

Tscope is not written from scratch. To reduce the writing effort and to increase stability, Tscope relies on a number of existing free software libraries and programs that provide parts of the functionality needed by an experiment-programming environment. In this section, we will first describe the C programming libraries Tscope is built upon and motivate the choice of these specific libraries. Then we will describe the software used in the programming environment developed for Tscope.

Besides standard ANSI-C libraries, Tscope relies on the Allegro game programming library (Hargreaves, n.d.), the Windows API, and a Windows port of the ioperm library. Using the Windows API was not a matter of choice. It is the only way to get access to the computer's hardware timer and to influence a program's scheduling priority. This makes it the basic building block of any Windows timer system that claims millisecond accuracy. Most compilers capable of generating Windows programs have access to this API. One further need for an accurate timing system is the ability to read out the status of the screen refresh signal and of response boxes connected to the game port or parallel port. The ioperm library is, to our knowledge, the only completely free library that provides this functionality for systems based on Windows NT. It has the disadvantage that it cannot be used by other compilers than the Cygwin (Cygnus Solutions, n.d.) version of GCC (Free Software Foundation, n.d.). Thus, the combination of the Windows API and the ioperm library makes millisecond timing and millisecond input precision possible, which are the first two criteria our library should meet. The third criterion, making a complete set of functions for the experimenter, could be met by building upon the Allegro library. This library is aimed mainly at game programming (its name, Allegro, is a recursive acronym for Allegro Low Level Game Routines), but the needs of experiment and game programmers clearly overlap: Allegro provides functions for opening a graphics screen and reading input from popular (but inaccurate) input devices, such as mice, joysticks, or keyboards. It further provides a full set of graphics functions and text output functions. One particular advantage of this library is that it can be compiled so as to run on different systems, two of them being DOS and Windows. The DOS version was already a popular alternative among experiment programmers for doing the graphics-rendering part of their experiments. Therefore, inclusion of this library into Tscope makes it possible for former users of the DOS version of Allegro to port their programs to Windows just by replacing their timer and screen initialization calls with Tscope calls and recompiling their code. One disadvantage of Allegro is that it is a low-level library. This makes it difficult to use for novice programmers (both the required level of C programming skills and the required knowledge about operating systems and hardware are relatively high). Therefore, part of the Tscope library provides simpler, higher level calls to Allegro functions that hide both programming and computer complexities. By including these higher level calls, the fourth criterion—ease of use—could be met.

Finally, our fifth criterion was that an IDE should be available to develop Tscope programs. Given the selection of C libraries used by Tscope, the choice of compilers was restricted to one. Only the Cygwin version of the GCC compiler is currently able to use the Windows API in conjunction with the ioperm and Allegro libraries. Unfortunately, the Cygwin version of the GCC compiler does not come with an IDE. This implies that a separate code editor is needed to write a program, which can then be compiled using commands given at the command prompt. For ease of use, we built an IDE that bundles the GCC/Cygwin system, together with the needed libraries and a high-quality, freely available code editor (Crimson Editor; Kang, n.d.) that is preconfigured with shortcuts to all the necessary calls to GCC. This IDE is available at Tscope's site but does not include the Tscope library itself. Tscope has to be installed separately, since it is more liable to change than is the programming environment. In the next section, we will describe the functions provided by Tscope.

Functionality Provided by Tscope

Tscope consists of several function groups. The logic of each function group is described in the following sections. Three properties are shared by all Tscope functions. (1) They all have the prefix ts, which differentiates them from standard C functions. (2) All the functions take as few arguments as possible. For example, the function to draw a circle takes only two coordinates and a radius as arguments. Whether the circle should be filled and in what color it should appear are controlled by graphics parameter functions. If the user does not specify the graphics parameters, sensible defaults are chosen. (3) All initializations are done automatically. At the first call of a graphics

function, the graphics screen is opened. The first time the user requests a random value, random seeds are initialized on the basis of the clock time and so forth.

On the Tscope site and within the Tscope distribution, there is a reference manual that gives an in-depth explanation of each specific function and function group. For each function group, some examples illustrate their use. On the site, there is also a code base available with full-fledged experiments programmed with Tscope functions. The following sections, therefore, only briefly highlight the most important features of each function group. For more detailed information, the reader is referred to the Tscope site.

Screen setup routines. With this group of functions, the user controls the setup of the graphics screen in which the experiment will run. Tscope functions exist for setting user-defined variables, such as screen resolution, color depth, refresh rate, and screen mode (i.e., full screen or windowed). If the user does not specify any of these values, the experiment is run by default in a small window on the desktop, which is a sensible setup while the experiment is programmed. Before the screen is opened explicitly by the user or automatically by the first call to a graphics function, the screen parameter functions have to be called. If the user specifies a new parameter value while the screen is already running, the screen is not reopened until the user does so with an explicit call. Because of the automatic initialization of the screen and the choice of default settings, most of the time users will not have to bother with these functions until their program is finished and ready to run.

Timing routines. The timing routines work in several steps. At the beginning of the program, the user defines the buttons that will be used as response buttons during the experiment. All of the buttons on the supported response devices (keyboard, mouse, joystick, game port, and parallel port response boxes) have a specific code (e.g., P1, P2, P3, and P4 for the buttons on a parallel port response box). When a button is defined, device drivers are loaded automatically. The first button defined by the user gets response value 1 assigned to it; the second button gets response value 2; and so forth. Only the buttons that are defined in the program will be monitored by the response registration functions; all others will be ignored.

Contrary to standard timing functions, the timers provided by Tscope do not work with the regular start, stop, and reset commands. A first function (called *ts vsync*) waits for the next vertical retrace of the screen to begin and stores the time at which this occurs in a user-supplied variable. This function should be called just before the stimulus is put on the screen, so that the onset of the stimulus is synchronized with the vertical retrace of the screen. A second function (called *ts resp*) waits for the response of the participant. It also stores the clock time at which this occurs in a user-supplied variable and, further, returns the value of the response button pressed. The actual response time can then be derived by subtracting the clock time at stimulus onset from the clock time at response onset. Since these clock times are measured in ticks, they have to be converted to milliseconds to get a humanly understandable value. Macros are provided to do that.

Both the vertical synchronization and the response function are able to detect whether the experiment has been preempted by Windows. Within these functions, the status of the response buttons or the vertical retrace signal is repeatedly read out. Every time the input is read out, the actual clock time is stored and compared with the time of the previous readout. Thus, the timing error of an event is equal to the amount of time that elapses between the last readout at which there was no input and the first readout at which the input is present. This error should always be smaller than a millisecond but will be much larger if the experiment was preempted by Windows while the input occurred. All timing functions store their timing error in a user-supplied variable. At the end of a trial, the user should sum the error of the vertical synchronization function with the error of the response function to know the total timing error of the response time. Note that the error values stored by the timing functions will also be in clock ticks. To get an informative value, this should be converted to microseconds, rather than milliseconds. Most likely, the timing error will be much smaller than a millisecond on most of the trials. Conversion of such small timing errors to milliseconds will give a value of 0, which shows only that timing was "good." Measuring the timing error in microseconds will further show how close timing accuracy is to criterion and how the specific computer is performing.

Some variants of the response functions mentioned above are available. Examples are registering double responses, changing the valid response keys between trials, or timing how long the response key was pressed. There is also a function available that waits for a user-defined number of vertical retraces. This is very useful in priming experiments when a stimulus has to be shown for a fixed number of refresh cycles. Experienced users who need a custom timer loop can define one using the available lower level functions that read out the clock time, the status of the defined response buttons, and the status of the screen refresh signal.

Some additional notes about the vertical synchronization signal are also in place. At lower refresh rates of the screen (i.e., lower than 70 Hz), Tscope's synchronization function can reliably read out the refresh signal from the system registry. At higher refresh rates, the duration of this signal becomes shorter, and chances increase that it will be missed by Tscope's synchronization functions. If the signal is repeatedly missed, the program will run in an endless loop. To avoid such situations, Tscope uses, by default, a retrace simulator, rather than reading out the signal from the registry, if the screen's refresh rate is higher than 70. This is quite a conservative setting, since Tscope will perform well on most computers at reading out the signal from the registry even at much higher refresh rates. Therefore, a program is included with Tscope that tests how capable the system is of detecting the vertical synchroniza-

tion signal at different refresh rates. If a system performs well at refresh rates higher than 70, the limit above which the retrace simulator will be used can be increased by a call to a parameter function. The retrace simulator should be used only if the refresh signal from the registry cannot be read out reliably. Note that the simulator cannot know at which specific moment a retrace is initiated; it knows only how much time has elapsed between two retraces of the screen. Consequently, the simulator can be used to control the duration of stimuli on the screen, but the onset time of the stimuli cannot be controlled as effectively as with the vertical synchronization signal from the registry. Therefore, Tscope will always inform the user when it falls back on the retrace simulator.

With the functions described above, the user is capable of timing responses and of controlling the onset of stimuli. If the experiment program is preempted by Windows at the moment at which the vertical synchronization or response occurs, this is reflected in the timing error reported by these functions. The probability of one's program being preempted can be lowered by increasing the program's priority. Several priority settings are possible. On the one hand, during the development and testing of a program, accurate time measurement is not required. Thus, priority should be left at its default setting. This way, the program can be easily aborted when it runs in an endless loop. For the experimental runs, on the other hand, the program's priority should be increased. Two settings are possible. The HIGH setting is appropriate and recommended for most situations. When this setting is used, the program will not be preempted to allow another regular program to run. The system's resources will be divided between that program and the operating system. Adequate timing accuracy can be achieved with HIGH priority, without bringing the stability of the system into danger. As the operating system is still running at this setting, the program can still be shut down externally when it becomes unstable. There is one higher priority setting, called REALTIME. This should be used only if timing errors are too frequent, even at the HIGH priority setting (to our experience, this is a situation very unlikely to occur). In REALTIME mode, the program's priority is boosted to such a level that the program will not be preempted even for operating system tasks. This also implies that the program cannot be shut down with the task manager: If it runs in an endless loop, the whole operating system hangs. In that case, the only way to get out of this endless loop is by a hard reset of the system. One further drawback of this priority mode is that program threads (subprocesses running in parallel) responsible for reading mouse, joystick, and keyboard input are also preempted. This means that only input from response boxes connected to the game port or parallel port can be read in this mode. As a consequence, the REALTIME priority mode should be used infrequently or, even better, never at all.

There is still one other feature of Tscope's timing functions that further increases accuracy: On multitasking systems such as Windows, timing functions should, in general, be put on a separate thread that is executed only when needed. For example, a timer function meant to read out the status of the parallel port with millisecond accuracy should execute only once a millisecond. After execution, it should give control back to the operating system for the remainder of that millisecond, leaving time for other programs to execute. Sadly enough, one can only hope that the operating system will return focus to the timer thread at the beginning of the next millisecond. Therefore, in order to avoid the risk of not getting control back fast enough, the Tscope timers are implemented as busy loops that iteratively check the input without ever giving control back to the operating system. For normal Windows programs, this would be a bad programming technique, since it reduces the total processing efficiency of the system. However, for experiments, it is defensible, since experimenters are interested only in the efficiency of their program, rather than the efficiency of the entire system.

To conclude this section on timing, we will briefly discuss the different input devices that can be used. The error estimate provided by the timing functions includes only the timing error that arises within the timing loop of the experiment. Other sources of error are the time needed for a response button to travel from its open state to its closed state or the time needed by the hardware within the input device to detect the closing of a button and to transmit the button's status to the system. Recently, Plant et al. (2003) have extensively tested several input devices, and the only really accurate input devices were response boxes connected to the parallel port of the computer. The game port was not tested in this study, but a previous test (Shimizu, 2002) concluded that the game port is also millisecond accurate, at least on DOS. Therefore, if timing is critical, we suggest using custom-made response boxes connected to the game or the parallel port.

Other routines. Although the timer functions are the core of Tscope's functionality, the remaining functions can be seen as complementary functions that are included to make Tscope a complete and easy-to-use experiment-programming library. How they are implemented is of less importance, and most of them are wrappers for lower level Allegro functions. The basic features of these remaining functions will be highlighted here in order to give the reader an idea of the possibilities of the library. For more information, the reader is again referred to the reference manual and the examples included with the Tscope distribution.

To start with, Tscope offers some basic graphics functions. All graphics functions use a Cartesian coordinate system. In this system, the origin (0,0) is at the center of the screen, and coordinate values increase to the right of and above the origin. The units used in Tscope's coordinate system are pixels. Helper functions are available that transform percentages of the screen size into number of pixels. Although this coordinate system has its advantages for stimulus setups that are centered around the center of the screen, some users might prefer the standard computer-style coordinate system with the origin in the

upper left corner of the screen and increasing coordinate values to the right of and below the origin. Therefore, a parameter function exists that switches between the standard computer-style coordinate system and the Cartesian coordinate system.

Graphics parameters such as drawing color, background color and font are not controlled by the graphics and text functions themselves. Tscope provides separate functions that set a specific parameter for all subsequent drawing, or until that parameter is set to a new value. Since most experimental stimulus setups are quite simple and use a specific color or font for more than one drawing operation, this effectively reduces the amount of code needed to put stimuli on the screen. Thus, basic stimulus setups can be drawn to the screen directly, using printf-style text output functions and easy-to-use drawing functions. More complex stimuli that take more than one refresh cycle to draw can be drawn on memory bitmaps before the trial starts and blit to the screen when needed. Bitmap files can also be prepared in an external graphics program and read from disk by Tscope.

A final set of functions discussed here are the randomizers, which an experiment-programming library could not do without. Tscope contains a C implementation of the randomizers described by Brysbaert (1991). With these randomizers, random values can be drawn from uniform, normal, or exponential distributions. A function for producing random lists without replacement is also available. One special random function produces lists where the succession of the random values is controlled. This kind of randomization can be used, for example, in task-switching experiments. Reusing random seeds and, thus, replicating random lists is also possible.

Extensions

An experiment-programming library can never be complete. Any day, an experimenter can have specific needs that are not foreseen by the authors of the library. Several of Tscope's design characteristics make it easily extendable in such situations. (1) Most of Tscope's graphics-rendering functions are based on lower level routines provided by the Allegro game programming library. If a certain functionality is not provided by Tscope, chances are still high that it can be implemented using standard C and Allegro functions. These functions can be called in any Tscope program. For example, an experimenter from our lab needed a special blit function to blend two bitmaps. Although Tscope itself provides blit functions for rotated, scaled, or flipped blits, blended blits are not implemented in the library. In Allegro, several types of blended blits are possible, so an ad hoc solution that made some Allegro calls within a Tscope program was provided. (2) Sometimes, experimenters need only a slight variation on an existing Tscope function. One experimenter needed a function to detect the offset, rather than the onset, of a response. In such a case, experimenters can copy the source of the function of interest and alter it to their discretion. (3) The simplest situation arises when an experimenter needs a procedure that can be achieved by an algorithm that uses only existing Tscope functions. Such a situation reduces to declaring a new function within the program of interest.

Experimenters repeatedly have suggested or asked for extensions to Tscope. Depending on the generality of the function needed, the solution can be included as a new function in the subsequent release of Tscope, or it can be put on a special section of the Tscope site where paradigm-specific functions are collected. These can be downloaded individually and copied in the source of the programs that use them. An example of a now standard extension is a function that produces a beep on the speaker. An example of a specific function that can be downloaded separately is an auditory stop signal timer function. In a standard stop signal task, a visual stimulus is presented on the screen, and after a variable delay, an auditory stop signal is presented. For this purpose, a function was written that allows a flexible and parallel presentation of visual and auditory stimuli.

Full Experiments

Tscope is, in the first place, a function library, not a programming course. Anyone trying to use Tscope will need a basic knowledge of ANSI-C. Several good tutorials are available freely on the Web. Such tutorials will, of course, not cover the design problems specific to writing experiment programs. To illustrate how an experiment program should be structured, the Tscope distribution includes a number of sample programs that implement classic experiments in cognitive psychology. These can be used as examples when writing new experiments.

Conclusion

In this article, we presented Tscope, a C programming library designed for programming experiments that run on Windows systems. We felt the need for such a library, since many experimenters still rely on DOS or on less flexible experiment generators. Two factors may have contributed to this situation. First, initial concerns raised about the suitability of Windows for running time-critical experiments might have led to a reluctancy to use Windows. Second, programming experiments on Windows is currently difficult for experimental psychologists with moderate programming skills, due to a lack of higher level functions that fill their specific needs. The present project targets both problems. Since previous research has shown that Windows is capable of running time-critical experiments, we decided to develop a set of functions and a development environment specific for experimental psychologists. With these functions, any experimetal psychologist with a basic knowledge of the C programming language should be able to program time-critical experiments that run on Windows systems. Tscope is distributed under the GNU General Public License (Free Software Foundation, 1991) and is available at expsy.ugent.be/tscope.

REFERENCES

Brysbaert, M. (1991). Algorithms for randomness in the behavioral sciences: A tutorial. *Behavior Research Methods, Instruments, & Computers*, **23**, 45-60.

Chambers, C. D., & Brown, M. (2003). Timing accuracy under Microsoft Windows revealed through external chronometry. *Behavior Research Methods, Instruments, & Computers*, **35**, 96-108.

Cygnus Solutions (n.d.). Cygwin [Computer program]. For details on Cygwin, see www.cygwin.com/.

De Clercq, A., Crombez, G., Buysse, A., & Roeyers, H. (2003). A simple and sensitive method to measure timing accuracy. *Behavior Research Methods, Instruments, & Computers*, **35**, 109-115.

Dixon, P. (1991). The promise of object-oriented programming. *Behavior Research Methods, Instruments, & Computers*, **23**, 134-141.

Forster, K. L., & Forster, J. C. (2003). DMDX: A Windows display program with millisecond accuracy. *Behavior Research Methods, Instruments, & Computers*, **35**, 116-124.

Free Software Foundation, Inc. (1991). GNU General Public License. See www.gnu.org/copyleft/gpl.html.

Free Software Foundation, Inc. (n.d.). GNU Compiler Collection [Computer program]. Available at gcc.gnu.org/.

Hargreaves, S. (n.d.). Allegro Low Level Game Routines [Computer program]. For details on Allegro, see www.talula.demon.co.uk/allegro/.

Kang, I. (n.d.). Crimson Editor [Computer program]. For details on the Crimson Editor, see www.crimsoneditor.com/.

MacInnes, W. J., & Taylor, T. L. (2001). Millisecond timing on PCs and Macs. *Behavior Research Methods, Instruments, & Computers*, **33**, 174-178.

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.

Myors, B. (1998). A simple graphical technique for assessing timer accuracy of computer systems. *Behavior Research Methods, Instruments, & Computers*, **30**, 454-456.

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

Norris, D. (1984). A computer-based programmable tachistoscope for nonprogrammers. *Behavior Research Methods, Instruments, & Computers*, **16**, 25-27.

Plant, R. R., Hammond, N., & Whitehouse, T. (2003). How choice of mouse may affect response timing in psychological studies. *Behavior Research Methods, Instruments, & Computers*, **35**, 276-284.

Psychology Software Tools, Inc. (n.d.). E-Prime 1.0 [Computer program]. For details on E-Prime for Windows, see www.pstnet.com/e-prime/.

Shimizu, H. (2002). Measuring keyboard response delays by comparing keyboard and joystick inputs. *Behavior Research Methods, Instruments, & Computers*, **34**, 250-256.

NOTE

1. The Windows API provides access to a set of function calls that enable a program to communicate with the operating system. It is made available by most compilers that are capable of producing Windows programs (as opposed to compilers that produce DOS programs; the latter programs can also be run on Windows, but in a so-called DOS box).

(Manuscript received January 31, 2005;
accepted for publication March 29, 2005.)