No more Bricolage! Methods and Tools to Characterize, Replicate and Compare Pointing Transfer Functions

Géry Casiez[1,2,3] *and Nicolas Roussel*[2]
[1]LIFL, [2]INRIA Lille & [3]University of Lille
Villeneuve d'Ascq, France
gery.casiez@lifl.fr, nicolas.roussel@inria.fr

ABSTRACT
Transfer functions are the only pointing facilitation technique actually used in modern graphical interfaces involving the indirect control of an on-screen cursor. But despite their general use, very little is known about them. We present EchoMouse, a device we created to characterize the transfer functions of any system, and libpointing, a toolkit that we developed to replicate and compare the ones used by Windows, OS X and Xorg. We describe these functions and report on an experiment that compared the default one of the three systems. Our results show that these default functions improve performance up to 24% compared to a unitless constant CD gain. We also found significant differences between them, with the one from OS X improving performance for small target widths but reducing its performance up to 9% for larger ones compared to Windows and Xorg. These results notably suggest replacing the constant CD gain function commonly used by HCI researchers by the default function of the considered systems.

ACM Classification: H.5.2 [Information interfaces and presentation]: User interfaces - Graphical user interfaces.

General terms: Documentation, Experimentation, Human Factors, Measurement, Performance, Standardization

Keywords: Pointing, control-display gain functions, CD gain, pointer acceleration, transfer functions, toolkit

INTRODUCTION

Indirect control of an on-screen cursor with a separate device has been the prevalent way of pointing in graphical interfaces for many years. The mouse is undoubtedly the most popular pointing device in this context. As explained by Moggridge [20], it was not chosen simply because Engelbart invented it, but because it turned out to be the device that performed best for pointing and clicking on a display, outperforming everything else that was tried in early tests with users. More than forty years later, the mouse still provides a good match between human performance and the demands of desktop graphical interfaces [13], the touchpad offering a similar match on laptop configurations. Despite the current trend for tactile screens, indirect pointing will thus probably remain the prevalent paradigm for some time.

In indirect pointing configurations, movements in the control space can be mapped to different ones in the display space. We use the term *transfer function* to refer to the relationship between movements in the two spaces. It is very important to note that in order for this relationship to be meaningful, it has to be hardware-independent. All movements must thus be described using standard length and time units (e.g. meters) and not device-specific ones (e.g. mickeys and pixels). In the case of simple linear relations, the term *CD gain*[1] is commonly used to refer to the scale factor between the two spaces, e.g. $CDgain = V_{\text{display}}/V_{\text{control}}$ [8]. To be meaningful, this coefficient must be unitless, which will only be the case if the two factors involved in its computation are expressed using the same length and time units. The CD gain can be constant or dynamically adjusted over time based on control space kinetics or extrinsic information, for example. Whether static or dynamic, CD gain settings involve a trade-off between gross and fine positioning [16]. High gains reduce the time it takes to approach a distant target but make it hard to precisely position the cursor on it. Conversely, low gains support precise positioning but increase the time to cover large distances.

It is generally assumed that in accordance with Meyer et al.'s *optimized initial impulse* model [19], the so-called "pointer acceleration" mechanisms implemented in modern desktops increase the CD gain as the user's hand or finger velocity increases. The transfer functions of Microsoft Windows, Apple OS X and Xorg (the X.Org Foundation server) are actually the only pointing facilitation mechanisms available to all users of these systems. But despite their general use, very little is known about them. Current knowledge on velocity-based transfer functions relies on evaluations of basic ones adapting the CD gain in discrete or continuous ways using low-order polynomials, e.g. [11, 14, 21]. The internal details and design rationales of the functions that we all use are mostly unknown. And with the notable exception of Casiez et al.'s work on Windows XP and Vista functions [8], their impact has never been studied.

This paper reports on efforts we made with the long-term goals of advancing this state of knowledge and bringing it

[1]The term *CD ratio* is also used and corresponds to the inverse of the gain.

UIST'11, October 16–19, 2011, Santa Barbara, CA, USA.

in line with current practices. The paper is organized as follows. We first briefly review the related work, explaining how pointing facilitation research deals with transfer functions and what is known about them. We then explain what is actually involved from a system perspective when pointing on a desktop. We present `EchoMouse`, an HID device we created to characterize the transfer functions of any system, and `libpointing`, a toolkit that we developed to replicate and compare the ones used by Windows, OS X and Xorg[2]. We describe these functions and report on an experiment that compared the default one of the three systems. We conclude with a discussion and implications for future work.

RELATED WORK

Pointing transfer functions can be considered as low-level and general-purpose mechanisms for pointing facilitation. Before explaining what is known about them, it is interesting to look at how research on other pointing facilitation techniques takes them into consideration.

Transfer functions in other pointing facilitation research

As we explained, it is usually assumed that desktop systems dynamically adjust the CD gain based on movement speed. This behavior might well interfere with other pointing facilitation mechanisms, especially those that manipulate the CD gain, e.g. [23, 10, 5, 22]. As a consequence, one would expect researchers working on pointing facilitation to try to disable the system's transfer function, to precisely characterize it, or to systematically investigate its effect.

Transfer functions are usually treated as control variables, meaning they might influence a dependent variable but are not under investigation and are thus held constant from one test condition to another. One would expect a clear description of these control variables to ease the replication of a technique or experiment with different hardware configurations or operating systems, in case the control variable would in fact be a confounding one. However a review of the recent literature on pointing facilitation shows that the level of details provided is often incomplete or unclear.

Numerous authors report using a constant CD gain or ratio as a baseline condition or as a basis for their technique but fail to describe it with sufficient details. In [10], for example, Cockburn & Firth explain that the CD gain "*was set to a constant ratio of approximately 1:1.6*" in experiments running on a Linux system, but do not explain why they chose this particular value nor how they enforced it. The predictive pointing technique of the Delphian Desktop was evaluated by Asano et al. on Windows XP with a CD ratio "*set to a constant value of 0.5*" [2]. Again, the paper does not explain how this ratio was enforced. For the Bubble Cursor on the same system, Grossman & Balakrishnan say that "*mouse acceleration was set to 0, with a control-display ratio of 1/2*" [12]. The exact meaning of "*set to 0*" is unclear considering the Windows XP configuration interface (Figure 4), and we will see that a unitless constant ratio of 0.5 is not achievable through it (Figure 6). In their study of sticky targets, Mandryk & Gutwin said "*Windows pointer acceleration was turned off, and the baseline mouse gain was set to the midpoint*" [18]. The exact definition of this baseline is unknown which is unfortunate since the authors scaled it by 11 values between 0.05 and 1.0, CD gain being one of the experiment factors. In their own study of sticky icons, Worden et al. say that "*normal mouse gain was set at a constant 1 mickey to 3 pixels ratio for all conditions*" [23]. But as the mouse and display resolutions are not specified, this gain can not be expressed in a unitless hardware-independent way, which makes their results difficult to compare with those from other studies.

Enforcing a hardware-independent transfer function, even a constant gain, is actually quite difficult with current systems. Wobbrock et al. had to go to great lengths to disable Windows Vista's pointer acceleration and dynamically control the CD gain for their Angle Mouse study, for example. They acknowledge that "*although some on-line documentation discusses pointer ballistics in Windows, it does not contain sufficient information to establish the slider-to-gain mapping.*" [22]. A good way of enforcing a transfer function is to use a device not attached to the system cursor and an API that provides access to its raw data. As an example, Blanch et al. used the absolute coordinates of a puck on a Wacom tablet as input for their Semantic Pointing technique [5].

An alternative to enforcing a particular baseline is to use the default transfer function of the system. For the Ninja Cursors, Kobayashi & Igarashi say "*the mouse speed and acceleration rate were set to the Windows XP default values (middle speed, no acceleration)*" [15]. For DynaSpot, Chapuis et al. used "*the default X Window acceleration function*" [9]. As it is unclear whether these functions take into account specific characteristics of the devices, extensive details should be provided about them including their resolution (per length unit) and frequency. A problem with this approach is that it will be possible to replicate or reproduce the experiment only as long as the original system can be used.

In some cases, there is simply no way of knowing which transfer function was used. In [17], for example, MacKenzie & Isokoski only report using "*an optical USB Microsoft IntelliMouse with four buttons and a scroll wheel*" and an "*experimental software written in Java*". Mouse data was presumably provided to the Java application by the underlying operating system through its operative transfer function, but none of them is explicitly mentioned in the paper.

What is known about pointing transfer functions

An extensive review of the literature on transfer functions has been recently conducted by Casiez et al. [8]. Prior to their work, research on the effects of these functions on pointing performance had been largely inconclusive. In all constant CD gain studies, the range of gain evaluated was either small or had quantization problems. And the few dynamic transfer functions evaluated poorly resembled the ones used in modern systems, most of them involving a few discrete steps or simplistic low-level polynomials, e.g. [11, 14, 21].

Casiez et al. [8] showed there exists a wide range of constant CD gains for which performance is constant and provided a way to compute that range knowing the hardware and target widths and distances used in a particular context. They

[2]The source code for `EchoMouse` and `libpointing` is available from `http://libpointing.org/`.

also showed that the acceleration mechanisms of Windows XP resulted in faster pointing than constant gain functions: they found an average 3.3% improvement and up to 5.6% for small targets or long distances. In a different study, Casiez and Vogel showed that the impact of transfer functions on performance can be severe in the case of force input [7].

Windows XP mechanisms were re-implemented for the study described in [8] based on information extracted from the Windows registry and documentation publicly available from Microsoft [1]. However, the cursor controlled by this implementation was not in perfect sync with the system one due to missing details in the documentation. Figure 3 of Casiez et al.'s paper shows a plot of four Windows XP functions based on their custom implementation. It also shows a plot of six OS X functions that were estimated from the analysis of publicly available Apple source code, but not re-implemented.

POINTING: A SYSTEM PERSPECTIVE

Most if not all modern pointing devices conform to the *Human Interface Devices* (HID) class of the USB standard. This class covers a variety of equipments including keyboards, mice, touchpads and joysticks but also telephones, remote controls, barcode readers and voltmeters. HID devices are thus required to provide extensive descriptions of their characteristics to be properly recognized and used. Among other things, a pointing device description specifies for each axis whether transmitted values (called *counts*) are absolute or relative, linear or nonlinear, their byte size, their logical range, the corresponding physical range and the unit system and exponent used. The description also specifies the time interval that should be used when polling for data transfers.

The HID specification defines a simple *boot report format* for mice that allows to use them before the operating system is loaded, for low-level system configuration [3, p. 61]. This format describes movements along two axis with relative, linear, unitless counts encoded with one byte per axis, between -127 and +127. Most mice support this format and for many, it is also the one they use to report to the system once it is loaded. This format is also supported by touchpads, although they could provide an absolute location, so they can be used in place of a mouse even at boot time.

The polling interval is typically set to 8 ms for mice, leading to an update frequency of 125 Hz. For devices that specify their logical and physical range and their unit system and exponent, the resolution can be computed in counts per unit with: $Res = LogRange/(PhysRange * 10^{Exp})$. Unfortunately, many report formats including the boot one specify unitless values with no physical range, which makes this formula inapplicable. For these devices, systems usually assume a resolution of 400 CPI[3].

The fact that systems have no reliable way of knowing a pointing device's resolution is becoming more and more problematic as manufacturers not only propose high-resolution ones but also some where it is adjustable[4]. A higher resolution results in more counts reported for the same distance traveled by the device, by sending bigger values at the same rate or by reporting more often. High-resolution mice indeed use two-bytes values for each axis and can set a polling interval as low as 1 ms. A transfer function not aware of the resolution change or not taking time properly into account will inevitably misinterpret the reported counts and produce undesirable effects, the most common one being considerably amplified movements. This problem is so frequent that a lot of people equate the resolution of the device with the cursor speed, as illustrated by this text displayed in the mouse section of a consumer electronics retailer:

> "*Specified in DPI, the resolution corresponds to the speed of the mouse cursor on your screen. The higher it is, the less you will have to move the mouse for the same on-screen distance, though you will be less accurate.*"

Changing the resolution of a pointing device or switching to one with a different resolution should not alter the mapping between movements in the control and display spaces. Changing or reconfiguring the transfer function should be the only way of doing that.

ECHOMOUSE

`EchoMouse` is an electronic device that we designed to measure a system's response to pointing movements received from an HID equipment. Based on a Microchip PIC (18LF14K50) programmed in C using the PICkit 2 development environment, it includes a switch and two LEDs for debugging purposes (Figure 1). Its program uses a mouse firmware provided by Microchip, so it appears as an ordinary HID mouse to the system. It has no motion sensor, though. We instead added a USB endpoint to the firmware to which a program can send an HID report to be echoed by the device on its mouse endpoint. Reports are expected and echoed in HID boot format. They are thus indistinguishable from genuine mouse reports and handled as such by the system.

We have used our `EchoMouse` to look into the transfer functions of Windows, OS X and Xorg with a specific program implementing the following procedure. After placing the

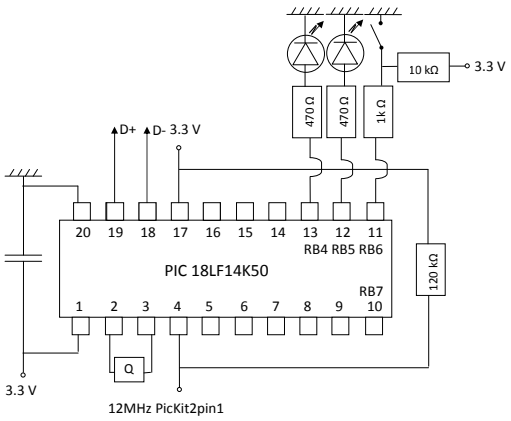


Figure 1: Electronic diagram of EchoMouse. The 3.3V can be easily generated from the 5V of the USB port using a voltage divider or a Zener diode.

[3]We will use CPI (*counts per inch*) and PPI (*pixels per inch*) instead of DPI to make a clear distinction between input and output resolutions.

[4]The resolution of the Logitech Gaming Mouse G500 can be adjusted between 200 and 5700 CPI, for example.

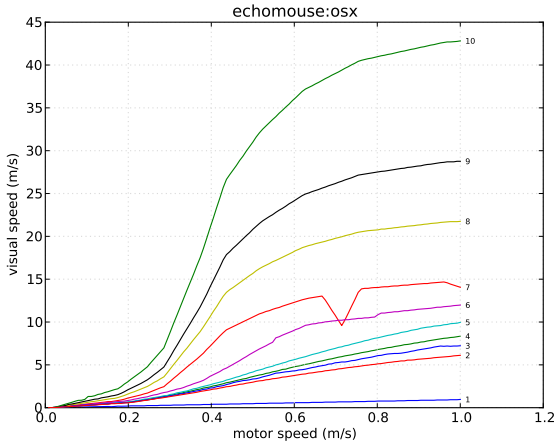


Figure 2: Speed plot for OS X 10.5.6 transfer functions ($n = 10$). Count and pixel values were converted to speed values considering the actual resolution of the monitor used and assuming a 400 CPI resolution for EchoMouse.

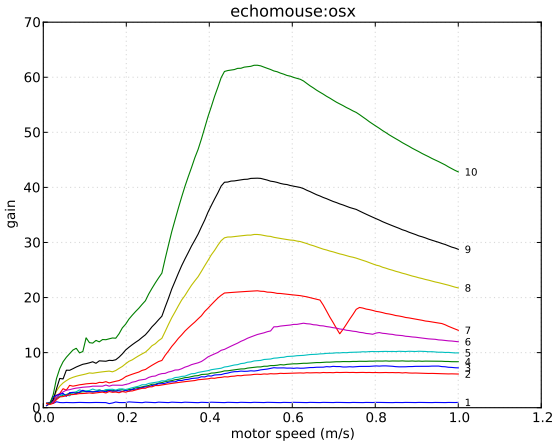


Figure 3: Gain plot of the data shown in Figure 2.

system cursor on the left edge of the screen, the program prepares an HID report describing a $(dx, 0)$ translation. It sends it n times to `EchoMouse` at 125 Hz, waits for a few milliseconds and polls the system for the new cursor position. It then divides the horizontal pixel distance traveled by the cursor by n and stores this number along with dx in a table. This procedure accounts for potential subpixel precision. When repeated for all dx between 1 and 127, it provides an extensional description of the transfer function used by the system. Repeating this for every pointer acceleration setting provides the descriptions of all the functions supported by the system.

Figures 2 and 3 illustrate the 10 transfer functions available in OS X 10.5.6 through the *Tracking* slider of the mouse preference pane shown on Figure 5. Figure 2 shows the speed in display space as a function of speed in control space. One can easily see with this plot that the functions used by OS X are not linear, can not be approximated by a single low-order polynomial and are likely defined in a piecewise fashion. The singularity observed in function 7 is inexplicable, however. And the differences between functions are not easy to perceive, especially at low speeds.

Figure 3 shows the CD gain as a function of speed in control space. This plot makes it easier to compare the functions at low speeds and to relate them to the movements in motor space, a CD gain of 1 corresponding to a horizontal line. The fact that some of the functions strongly decrease after a certain speed is hard to explain. We hypothesize that the functions were designed in a spatial space like the one of Figure 2 and that the designers were actually not aware of this slope change. The singularity observed in function 7 remains inexplicable. At this point, one can only hypothesize that the designers of the functions never tried to plot them.

`EchoMouse` allowed us to investigate the transfer functions used by three systems without spending too much time on their internals. Replicating the device is relatively easy. A similar but pure software approach is also probably achievable by developing fake mouse drivers, e.g. using `uinput` on Linux, or synthesizing low-level mouse motions, e.g. using the `SendInput` function on Windows. The `EchoMouse` approach is however restricted to observation, it can not alter the system functions. One can never be sure that the pseudo motions injected in the system have actually been processed when reading the cursor location. And one can hardly know if and which hardware or movement characteristics are taken into account by the system when applying the functions.

Although valuable for preliminary studies, an outside observation point is not enough to fully understand and compare the transfer functions used by modern desktop interfaces. In addition to `EchoMouse`, we thus developed `libpointing`, a toolkit that allows to replicate and compare them.

LIBPOINTING

The `libpointing` toolkit was designed with several goals in mind. First, we wanted a way of directly accessing HID pointing devices to bypass the system's transfer functions. Second, we wanted to replicate as faithfully as possible the transfer functions of Windows, OS X and Xorg. Third, we wanted the toolkit to run on these platforms to be able to compare our implementations to the genuine ones. And fourth, we wanted to support comparisons between the replicated functions and other ones.

The toolkit consists of about 10,000 lines of C++ developed on OS X 10.6, Ubuntu 10.10 and Windows XP, Vista and 7. Although parts of it use the Qt framework, care has been taken so that its essential components can be used with other GUI frameworks. A key aspect of `libpointing` is that it supports the use of URIs [4] to specify input and display devices as well as transfer functions. Combined with object factories, this makes it possible to (re)define at runtime the instances used by a program and contributes remarkably to the flexibility of the whole. The following summarizes the main other features of the toolkit.

Pointing devices

`PointingDevice` instances are created from URIs using the static `create` method of that class. Other methods allow to

check whether a device is active, to obtain its resolution (in counts per inch), update frequency and URI, and to associate a callback to it. The callback will be executed every time the device has a motion or button event to report, passing it a timestamp, dx and dy values (in counts) and an integer coding the buttons states.

The toolkit provides direct access to any connected HID pointing device through platform-specific subclasses and URIs such as `osxhid:/USB/4600000/AppleUSBTCButtons`. The special URI `any:` matches any supported device and will list the available ones in the console if a debug option is passed on the query string. All HID `PointingDevice` objects support hot (re)plugging of the corresponding device. The toolkit also includes two pseudo-device subclasses for debugging and testing that can be instantiated with URIs such as `noisy:?cpi=400&hz=125` and `dummy:?cpi=800&hz=125`. The first one will execute the callback at the specified frequency to report movements synthesized by a 2D Perlin noise generator. The second one will never execute the callback but will return the specified values when queried for its resolution and update frequency.

Display devices

`DisplayDevice` instances are also created from URIs using a static `create` method. Other methods allow to obtain the horizontal and vertical bounds (in pixels), sizes (in inches or millimeters) and resolutions (in pixels per inch) as well as the refresh rate and the URI of a particular display.

URIs such as `osxdisplay:/69676098` and platform-specific subclasses provide access to the displays connected to the computer. A pseudo-device subclass is also available that will simply store the configuration values passed on the query string, e.g. `dummy:?ppi=96&hz=60`, and return them as expected when requested by the above methods.

Transfer functions

`TransferFunction` instances are created using a static `create` method from a URI, a `PointingDevice` and a `DisplayDevice`. Other methods allow to obtain the URI of a function, to clear its internal state and to apply it to dx_{in} and dy_{in} values (in counts) with a timestamp to produce dx_{out} and dy_{out} values (in pixels). The toolkit provides subclasses that correspond to different transfer functions. Care has been taken so that all implementations are platform-independent, i.e. all the transfer functions proposed by `libpointing` can be used on all the supported platforms. Although it imposes some constraints, we believe that having cross-platform implementations is important: a long-term goal for `libpointing` could be to serve as a living archive of the functions tried and used in research and commercial systems.

Three subclasses replicate the functions used by Windows (`windows:`), OS X (`osx:`) and Xorg (`xorg:`). The next section of the paper will describe these functions with extensive details. The special URI `system:` can be used to create the single appropriate instance of these subclasses that corresponds to the function used by the system. Configuration settings passed on the optional query string are applied to both the created instance and the system function.

Two other subclasses implement constant CD gain in both the naive and the right way. The first one simply multiplies the dx_{in} and dy_{in} values by a specified factor, e.g. `naive:?gain=2`, and returns the nearest integers as dx_{out} and dy_{out}. As this ignores the resolution of the input and output devices and multiplies counts to produce pixels, the effective unitless gain will most probably not be the one requested (it is usually higher, input devices having higher resolutions than displays). The second implementation (`constant:`) takes the resolutions into account to effectively produce a hardware-independent constant gain. It converts counts into distances, multiplies these distances by the specified factor and returns their pixel equivalent.

The toolkit provides various other subclasses, including a `sigmoid:` function and a `composition:` one, the latter allowing to compose an arbitrary number of functions. Adding a new function is simply a matter of creating a new subclass, implementing its `getURI`, `clearState` and `apply` methods and modifying the `TransferFunction::create` method.

Utilities

`libpointing` includes some test and debugging programs that allow to list the available devices and their characteristics, for example. The toolkit also includes a transfer function plotting tool written in Python using matplotlib. This tool proved quite useful as it provided some visual confirmation that our implementations of the Windows, OS X, and Xorg functions matched the data collected using `EchoMouse`. It was also used to plot all the curves shown in this paper.

The toolkit also includes an application that allows to test an arbitrary number of transfer functions at the same time specified by their URI as command-line arguments. The program creates an on-screen cursor (a small square) for each function, a single pointing device being used to control all of them. In addition to supporting informal comparisons between functions or between different settings of the same functions, this application proved again quite useful to compare our implementation of the Windows, OS X and Xorg functions with the system ones.

WINDOWS:, OSX: AND XORG: TRANSFER FUNCTIONS

As explained, one of our goals with `libpointing` was to replicate as faithfully as possible the transfer functions of Windows, OS X and Xorg. We not only wanted cursors controlled by our implementations to follow the system ones as closely as possible, but we also wanted to replicate the controls on the functions available to users from the relevant configuration interfaces. Our work was based on the documentation and source code publicly available from Microsoft, Apple and the Freedesktop community. This section presents the key findings that emerged from it.

The curves shown in the figures below have been plotted assuming the following pointing and display devices: `dummy:?cpi=400&hz=125`, `dummy:?ppi=96&hz=60`.

windows:

The transfer functions used in Microsoft Windows were redesigned for Windows XP, released in 2001. The rationales

for this redesign and its general principles are described in a public document. Together with the configuration interface found in the "*Pointer Options*" tab of the "*Mouse Properties*" dialog (Figure 4), this document served as a starting point for our work. Note however that our experience was similar to that of Wobbrock et al. [22]: the information available was not sufficient to replicate the functions.

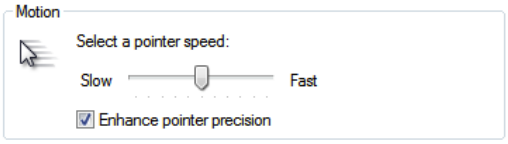


Figure 4: Windows 7 configuration interface with default settings. The same are used by Windows XP and Vista. No tooltip or help text is associated to these controls.

The transfer function code runs in an execution space where floating-point arithmetic is not available. The following equations illustrate how a (dx, dy) displacement in counts is transformed into pixel values when "*Enhance pointer precision*" is checked (Figure 4). Two important scale factors are used to convert count values to input speeds ($inConv$) and output speeds to pixel values ($outConv$). Computations for the y direction are omitted for brevity:

$$mag = max(|dx|, |dy|) + min(|dx|, |dy|)/2 \quad (1)$$
$$v_{in} = mag \times inConv \quad (2)$$
$$gain = lookup(v_{in}) \times pSpeed/10 \quad (3)$$
$$vx_{in} = dx_{in} \times inConv \quad (4)$$
$$vx_{out} = vx_{in} \times gain \quad (5)$$
$$px = vx_{out} \times outConv \quad (6)$$
$$dx_{out} = \lfloor px + rx \rfloor \quad (7)$$

The system computes a fixed-point approximation of the displacement vector magnitude (1). The magnitude is converted into an input velocity (2). A lookup table provides a base CD gain value for that velocity that is scaled by a factor ($pSpeed$) related to the "*pointer speed*" slider (Figure 4) to obtain the actual gain to apply (3). Each direction is then treated separately the following way. The directional input speed is computed (4) and multiplied by the gain to produce the output speed (5), which is then converted to a pixel displacement (6). The function returns the integral part (floor) of the sum of this displacement and the remainder of previous computations (7).

Although $inConv$ should be the quotient of the pointing device update frequency by its CPI resolution, empirical tests showed that it is always $1/3.5$. This constant value might be an approximation of 125/400, these numbers being common for mice. Empirical tests also showed that although $outConv$ should be the quotient of the display resolution by the pointing device update frequency, it is not the case either. Its actual value depends on the display resolution and frequency but also involves hardwired constants (96 PPI, 60 Hz, 150) and varies between system versions, XP and Vista differing from 7. Each of these three versions also uses a different algorithm to handle the x and y remainders: XP clears them when the pointer stops or changes direction, Vista clears them only when the pointer changes direction and 7 never clears them.

The lookup table that returns a base CD gain for a given device speed is stored in the Windows registry, so it should be possible to modify it. The position of the "*pointer speed*" slider (Figure 4) determines the value of the scale factor ($pSpeed$) applied on this base gain. Available values are: (slow) 1, 2, 4, 6, 8, 10 (default), 12, 14, 16, 18 and 20 (fast).

When "*Enhance pointer precision*" is unchecked, a naive constant CD gain is used. Based on the slider position, the available values for this gain are: (slow) 0.03125, 0.0625, 0.25, 0.5, 0.75, 1.0 (default, one pixel for one count), 1.5, 2.0, 2.5, 3.0 and 3.5 (fast). In this mode, no matter the system version, the remainders are never cleared.

The Windows transfer functions are available in `libpointing` through URIs such as `windows:<version>?slider=0&epp=true` where `<version>` can be one of `xp`, `vista` or `7`. The `slider` parameter encodes the slider position between -5 and +5 where 0 corresponds to the default position. The `epp` parameter indicates whether "*Enhance pointer precision*" is checked or not. Figure 6 shows the curves associated to each slider position, with and without enhanced pointer precision. Our implementation of these functions consists of about 200 lines of code. On XP, Vista and 7, a cursor controlled by it remains superimposed with the genuine one whatever movements are made. Our cursor can even respond to pointing device movement before the system cursor when the vertical synchronization of the display is disabled.

osx:

The source code for the internal parts of OS X that deal with pointing transfer functions is publicly available as part of the IOHIDFamily[5] project, the main concerned files being IOHIDSystem/IOHIPointing.cpp and IOHIDSystem/IOHIDSystem.cpp. From the archived versions of this project, it seems that the current pointer acceleration mechanisms first appeared in OS X 10.2, released in 2002. However, although the source code is available, the design rationales and principles of operation of these mechanisms are unknown. Figure 5 shows the related configuration interface, located in the "*Mouse*" pane of the system preferences. Note that from a user-perspective, the acceleration mechanisms are also badly documented, the tooltip associated to the slider being potentially misleading.

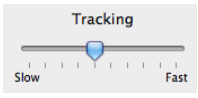
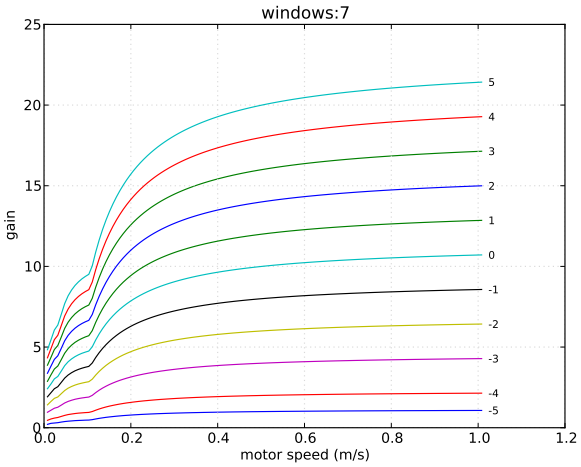


Figure 5: OS X 10.6.7 configuration interface for the mouse. A tooltip associated to the slider says "*Drag to adjust how fast you want the pointer to follow the movement of your mouse*".

[5] `http://opensource.apple.com/source/IOHIDFamily/`



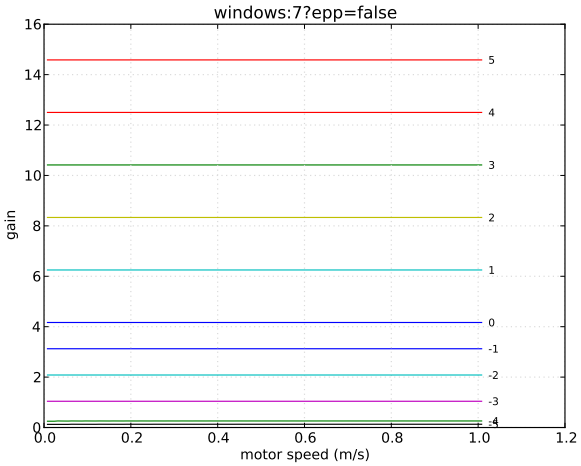
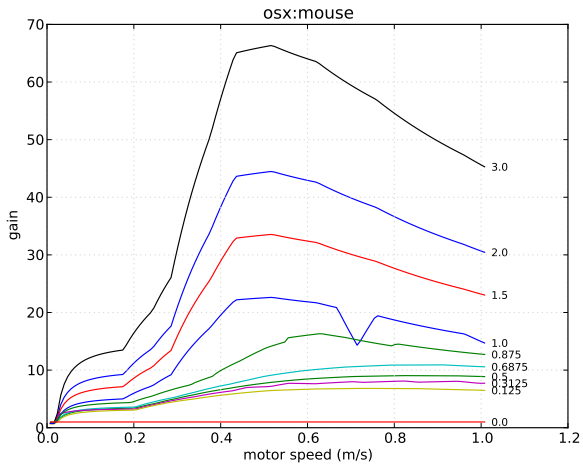


Figure 6: Windows 7 functions available through the interface shown in Figure 4 with and without enhanced pointer precision.



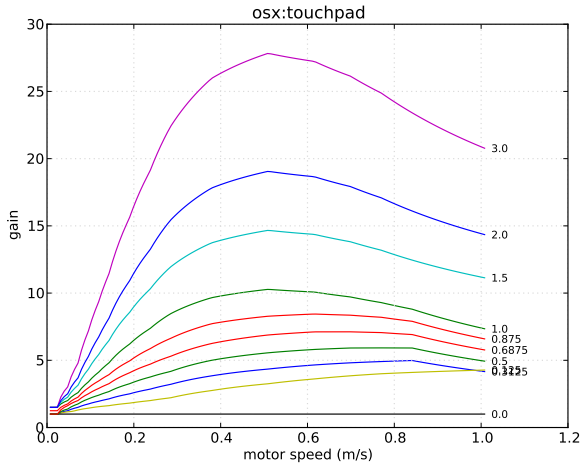


Figure 8: OS X 10.6.7 functions for mice (available through the interface shown in Figure 5) and touchpads.

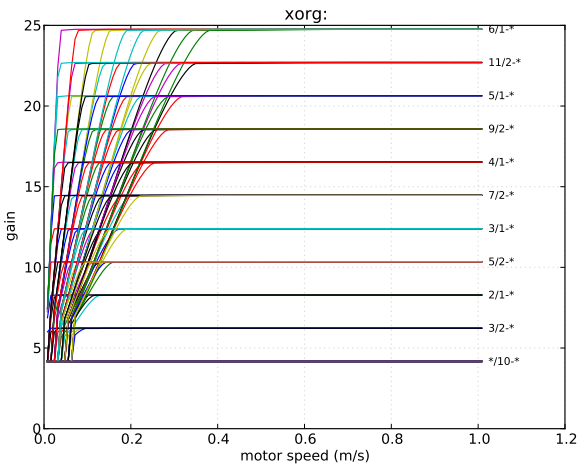


Figure 7: Xorg functions available in Ubuntu 10.10 through the interface shown in Figure 10.

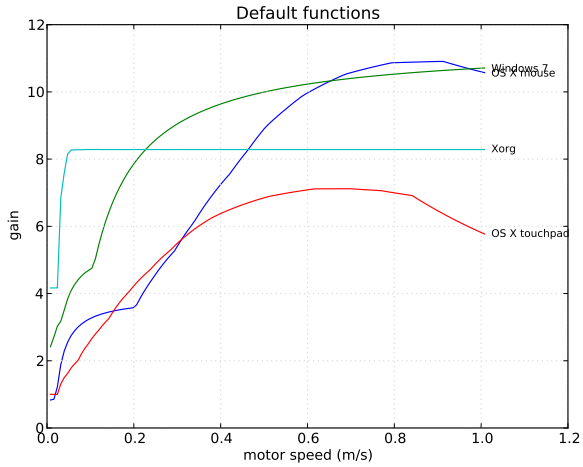


Figure 9: Default functions used by Windows 7, OS X 10.6.7 (mouse and touchpad) and Xorg.

It was relatively easy for us to isolate the portions of code from IOHIDFamily responsible for pointer acceleration (about 500 lines of code, mainly in IOHIPointing.cpp) and to add the necessary wrappers and definitions to make them compile on Windows and Linux. The basic operating principles followed by this code are also relatively simple and somewhat similar to those of Windows.

Each pointing device has an associated acceleration table provided by its driver (some also define a separate table for scrolling). This table specifies one or more curves defined by a series of segments and a *scale* level. The slider shown in Figure 5 allows users to specify a desired scale among the following: (slow) 0, 0.125, 0.3125, 0.5, 0.6875 (default), 0.875, 1, 1.5, 2, and 3 (fast). The system interpolates between the curves provided by the driver to create one that matches the desired scale and maps a vector magnitude to a CD gain value. Then for each (dx, dy) displacement, it computes an approximation of the vector magnitude using the same equation as Windows[6], uses the created curve to find the right CD gain, applies it to dx and dy, adds the previous remainders and returns the integral part of the result after updating the remainders. These remainders are never cleared.

The acceleration curves stored in device drivers are hardware-independent. When it interpolates between them, the system takes the resolution of the input device into account. However, it uses hardwired constants for the resolution of the display (96 PPI) and the frequency of the input and output devices (67 Hz). A detailed inspection of the drivers available on OS X 10.6.7 showed that several of them use the same tables. We identified two particular tables, one for mice and the other for touchpads, that seem to be used by all such devices that rely on Apple's drivers.

OS X transfer functions are available in `libpointing` through URIs such as `osx:<name or path>?scale=0.5`. The names `mouse` and `touchpad` can be used to load one of the generic acceleration tables that we found. A specific table can also be loaded from a file by specifying its path. The `scale` parameter encodes the desired scale. Figure 8 shows the curves associated to each slider position for generic mice and touchpads. Note that contrary to what it may seem in both cases, the unitless gain obtained for a scale of 0 is not always 1 but fluctuates between 0.9 and 1. From what we know, a constant CD gain is not achievable using these generic tables, even a naive one.

Although it can stay close in some situations, a cursor controlled by our `osx:` implementation does not remain superimposed with the genuine one. The reason appears to be some additional control mechanisms implemented in IOHIDSystem.cpp. Our current understanding is that these mechanisms feed the output of the function we just described into a trajectory prediction algorithm that schedules on-screen cursor updates synchronized with display refresh. No correction seems to be implemented in case a prediction was wrong, though. In effect, the whole acts as a low-pass filter on cursor movements. We hypothesize this explains the upward shift between the curves shown in Figure 3 and those in Figure 8 (above plot). Our present inability to schedule calls synchronized with display refresh prevents us from replicating these mechanisms, but we are currently investigating ways to circumvent this problem.

[6]All computations are also made using fixed-point arithmetic.

xorg:
The pointer acceleration mechanisms currently used by Xorg were introduced in 2008. The source code for these mechanisms is publicly available as part of the Xorg source tree[7]. It was again relatively easy for us to isolate the relevant portions of code (about 1500 lines of code, mainly in dix/ptrveloc.c) and to add the necessary wrappers and definitions to make them compile on Windows and OS X. This time, documentation for the design rationales and operating principles was also available[8], although a bit cryptic.

The changes introduced in Xorg in 2008 notably aimed at facilitating the exploration of transfer functions. The current architecture of the code supports 9 different *profiles* implemented within the new "*predictable*" scheme and the older "*lightweight*" scheme "*retained mostly for embedded scenarios*". Profiles can be considered as different transfer functions, although they share some common mechanisms and code. Numerous configuration settings are associated to them. But genericity and flexibility have a price: not only is the Xorg code for pointer acceleration much larger than the one used on other systems, but it is also far less readable.

The "*predictable*" scheme computes the euclidean distance corresponding to each displacement reported by the device and divides it by the time elapsed since the previous one. This instantaneous velocity is stored in a short history list ($n = 16$ by default) that is used to maintain a better estimation of the real pointing device velocity. Two adjustable settings also play an important part: *acceleration*, given as a fraction, and *threshold*. The first one defines a high value for the (naive) CD gain to be applied to displacements, considering a default low value of 1. The second one defines the minimum velocity that needs to be achieved to switch from the low gain to the high one. The active profile specifies how the estimated velocity will be used to determine the actual CD gain within these constraints. All computations are made with floating-point arithmetic. Remainders are preserved and never cleared.

The Xorg "*predictable*" transfer functions are available in `libpointing` through URIs such as `xorg:<profile>?accnum=2&accden=1&thr=4` where `<profile>` names one of the 9 available profiles, `accnum` and `accden` define the acceleration fraction and `thr` the threshold. On Ubuntu 10.10, a cursor controlled by our implementation remains superimposed with the genuine one.

It should be noted that a wide variety of command-line and graphical interfaces exists to configure the different profiles and their settings. Figure 10 shows the configuration interface available in the "*Pointer speed*" section of the "*Mouse preferences*" application of Ubuntu 10.10. Although the code that we use is not functionally limited to it, we will now fo-

[7]`http://cgit.freedesktop.org/xorg/xserver/tree/`
[8]`http://xorg.freedesktop.org/wiki/Development/Documentation/PointerAcceleration`

cus on the default profile used by Ubuntu ("*classic*") and the relevant settings that can be adjusted through this particular interface.

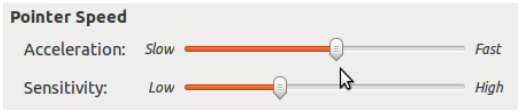


Figure 10: Ubuntu 10.10 configuration interface. A help page says about the first slider: "*Use the slider to specify the speed at which your mouse pointer moves on your screen when you move your mouse*". About the second: "*Use the slider to specify how sensitive your mouse pointer is to movements of your mouse*".

When the threshold is non-null, the "*classic*" profile implements a smooth transition between the low and high gain values. The sliders shown in Figure 10 only allow such configurations. As the label indicates, the upper slider controls the acceleration setting. When dragged, it feels like a continuous control but actually supports only a predefined set of values: (slow) 3/10, 4/10, 5/10, 6/10, 7/10, 8/10, 9/10, 10/10, 1/1, 3/2, 2/1 (default), 5/2, 3/1, 7/2, 4/1, 9/2, 5/1, 11/2, and 6/1 (fast). The bottom slider controls the threshold and actually feels like a discrete control. The available values are: (low) 1, 2, 3, 4 (default), 5, 6, 7, 8, 9, and 10 (high). In total, the interface shown in Figure 10 thus gives access to $19 \times 10 = 190$ configurations of the "*classic*" profile.

Figure 7 shows a plot of these 190 functions. As one would expect, the 90 functions with an acceleration setting lesser or equal than 1, those labeled */10-*, correspond to a naive constant gain of 1 (considering the 400 CPI and 96 PPI used for plotting the curves). Note that this is the only naive constant gain achievable through the interface shown in Figure 10 and that this interface does not allow to achieve a unitless constant gain.

Summary

Figure 9 shows the default transfer functions used by Windows, OS X and Xorg. Overall, despite a few differences, the different families have a lot in common.

The three systems take only partially into account the characteristics of the input and output devices. OS X is the only system that uses the real resolution of the input device (Windows assumes a 400 CPI resolution and Xorg does not use it). Xorg is the only system that takes input event times into account (the two others use harwired constant frequencies). Xorg completely ignores the display frequency and resolution while OS X uses hardwired constants for them (Windows varies on that topic).

All systems use a non linear function by default, but Windows and Xorg also support the use of naive constant gain functions. As the systems fail to properly take into account the resolution and frequency of the devices, none actually supports a unitless constant gain.

Windows and OS X both use fixed-point arithmetic. The three systems work with integer pixel coordinates but preserve the remainders to achieve subpixel precision when pointing. Windows 7, Mac OS X and Xorg never clear these remainders while they are cleared using different strategies on Windows XP and Vista.

The comparison of our custom cursors with the three system ones validated our Windows and Xorg implementations but revealed a slight difference for OS X presumably due to a trajectory prediction algorithm requiring information we are not yet able to provide.

EXPERIMENT

Our initial motivation for this experiment was to compare the performance of real-world transfer functions. Assuming they were probably used by many people and somewhat representative of theses systems, we decided to compare the default functions used by Windows, OS X and Xorg. We also added a constant CD gain function, as they are often used as a baseline for comparing pointing facilitation techniques.

Apparatus

A 400 CPI USB corded Logitech mouse was used as input device. A low-end model was preferred to a high-resolution one as 400 CPI is the default resolution considered by all systems. We used a 23" LCD display at a 1920×1200 resolution (98.5 PPI). The experiment was coded in C++ with the QT framework on a Windows 7 Professional machine with a NVidia GeForce GTX 460 graphics card. Our *libpointing* toolkit was used to get raw input from the mouse and apply the different transfer functions. Vertical synchronization of the display was disabled in order to be able to update our cursor's position at the mouse frequency (125 Hz). In this configuration, our controlled cursor was slightly in advance compared to the system one, which prevented any confounding effect of lag in the experiment.

Task

We used a reciprocal one dimensional pointing task (Figure 11). Each trial began after the previous target was successfully selected and ended with the selection of the current target. After a target was successfully selected, it turned grey and the next one (on the other side of the screen) turned green. If a participant missed a target, a sound was heard and an error was logged. Participants had to successfully select the current target before moving to the next one, even if it required multiple clicks. Participants used the left mouse button to select targets. After each block of trials, a cumulative error rate was displayed and a message encouraged participants to conform to an approximately 4% error rate by speeding up or slowing down.

Participants

Sixteen unpaid participants with a mean age of 30.6 (SD = 7.8, min = 23, max = 46) served in the experiment (15 male and 1 female, 13 right-handed and 3 left-handed). All participants worked most of their time with a computer. Three participants used exclusively OS X, four Windows 7, three Ubuntu 10.10. Two used Ubuntu 10.10 and Windows 7, two OS X and Windows 7, one Ubuntu 10.10 and OS X, and one Windows 7 and Ubuntu 10.10. Four participants used the mouse exclusively, two the touchpad and the remainder both devices. Among the sixteen participants, twelve kept the default settings for the mouse or touchpad while four slightly

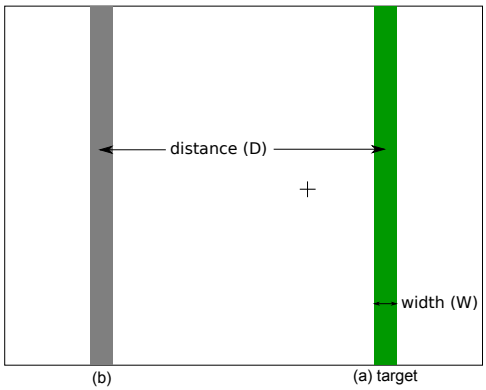


Figure 11: Experimental display. Targets were rendered as solid vertical bars equidistant from the center of the display in opposite directions along the horizontal axis. The target to be selected was colored green (a), and the previous one gray (b). The cursor was represented by a one-pixel-thick black cross 10 pixels wide.

increased the speed of their cursor by moving the slider in their configuration panel one or two ticks to the right.

Design

A repeated measures within-subjects design was used. The independent variables were the transfer function used (TF) and the target width (WIDTH).

Target distance was kept constant at 299.9 mm = 1,163 pixels. We decided for this moderate single distance because Casiez et al. had found stronger differences between constant CD gain and the Windows XP functions for small targets and long distances[9]. This decision was taken to reduce the duration of the experiment and to highlight the effect of WIDTH. The rationale was also that if no effect of TF was found with these settings, it would be likely that no such effect exists.

WIDTH was evaluated with four levels: W_{9pix} = 2.32 mm = 9 pixels, W_{6pix} = 1.55 mm = 6 pixels, W_{3pix} = 0.77 mm = 3 pixels, W_{1pix} = 0.26 mm = 1 pixel. Targets three pixels wide are common when resizing a window or clicking between two letters to position a text cursor. One pixel targets are less frequent but occur for example when selecting adjacent vertices or edges without zooming in vector drawing applications. The index of difficulty ranged from 7.0 to 10.2.

The transfer functions evaluated were constant CD gain of 1.5[10] (*Cst1.5*), the default Windows 7 function (*Win7*), the default OS X 10.6.7 function for mice (*OSX*) and the default Xorg function (*Xorg*). According to Casiez et al.'s method [8] and considering our experimental settings, the minimum gain value to prevent clutching was 30/30 = 1 and the maximum value that could be chosen given quantization problems and human limbs precision was equal to min(400/98.5, 0.26/0.2) = 1.3. The chosen CD gain value of 1.5 represents a good trade-off between these bounds and the CD gain value of 2 often used as a baseline in pointing experiments.

[9]The largest distance in [8] for a similar desktop configuration was 36 cm.
[10]Remainders were handled the same way as the other functions: they were never reset.

Participants were introduced to the task and had about 30 seconds to get used to it. They then completed three successive BLOCKS for each TF. Each BLOCK consisted of 24 trials: 6 repetitions of the 4 WIDTHS. WIDTHS were presented in decreasing order. The presentation order for TF was counterbalanced across participants using a balanced Latin Square design. Participants were encouraged to take a break after every 6 trials. They had to press the spacebar once they felt ready to start a new block. The desk was empty except for the keyboard and screen, and participants were instructed to use as much space as they wished to move the mouse. The experiment lasted approximately 15 minutes.

In summary, the experimental design was: 16 participants × 4 TF × 3 BLOCKS × 4 WIDTH × 6 trials = 4,608 total trials.

RESULTS

The dependent variables were the error rate and the movement time.

Error Rate

Targets that were not selected on the first attempt were marked as errors. Participants followed the instructions with an overall error rate of 4.1%. A repeated measures ANOVA showed a significant effect of WIDTH on error rate, the latter increasing as target width decreases ($F_{3,45}$=14.5, $p<0.001$). Pairwise comparisons showed significant differences between the smallest width and the three other widths ($p=0.001$; W_{1pix}: 9.7%, W_{3pix}: 2.8%, W_{6pix}: 1.7%, W_{9pix}: 2.1%).

Movement Time

Movement time is the main dependent measure and is defined as the time taken to move from a target to the next one and click on it. Targets marked as errors were removed from the timing analysis. We also considered trials at least three standard deviations away from the mean for each TF×WIDTH condition as outliers and removed them from the data analysis (1.6% of the trials).

A repeated measures ANOVA showed that the presentation order of TF had no significant effect or interaction on movement time, indicating that a within-participant design was appropriate. Repeated measures ANOVA showed a significant effect of BLOCK ($F_{2,30}$=14.2, $p<0.001$) on movement time. Pairwise comparisons showed a significant decrease in the movement time between the first block and the two remaining ($p<0.001$; Block 1: 2.05 s, Block 2: 1.93 s, Block 3: 1.94 s). The first block was thus removed from subsequent analysis.

Repeated measures ANOVA showed a significant main effect of TF ($F_{3,45}$=20.7, $p<0.001$)), WIDTH ($F_{3,45}$=244.4, $p<0.001$)) and a significant TF × WIDTH interaction ($F_{9,135}$=3.2, $p=0.023$)) on movement time (Figure 12). Post-hoc analysis showed significant differences between *Cst1.5* and the three other transfer functions ($p<0.001$, *Cst1.5*: 2.22 s, *OSX*: 1.86 s, *Win7*: 1.81 s, *Xorg*: 1.84 s). This shows that *Cst1.5* is more than 20% slower compared to the three default transfer functions. We did not control for clutching but according to the experimenter observation, it was infrequent for all conditions. When it occurred, it was at the beginning of the first block which was removed from the time analysis.

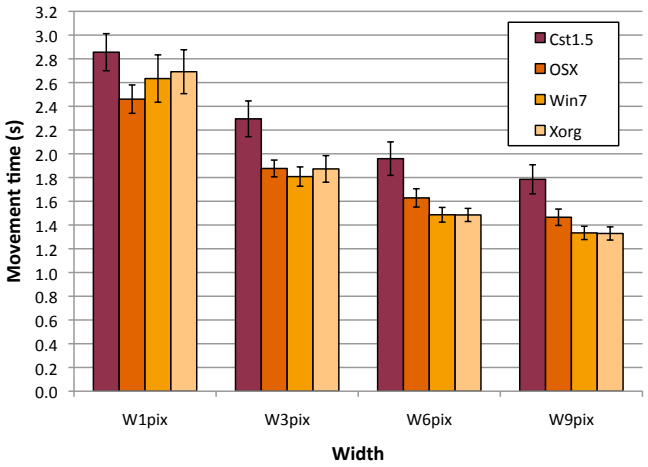


Figure 12: Mean movement time for TF and WIDTH, error bars representing 95% confidence interval.

For W_{9pix}, pairwise comparisons[11] showed significant differences ($p<0.001$) between *Cst1.5* and the three other transfer functions. We also observed significant difference ($p=0.013$) between *OSX* and *Win7* (*Cst1.5*: 1.78 s, *OSX*: 1.47 s, *Win7*: 1.33 s, *Xorg*: 1.33 s). For W_{6pix} pairwise comparisons showed significant differences ($p<0.002$) between *Cst1.5* and the other functions. *OSX* showed again significant difference ($p=0.001$) with *Win7* (*Cst1.5*: 1.96 s, *OSX*: 1.63 s, *Win7*: 1.48 s, *Xorg*: 1.48 s). For W_{3pix}, we again observed significant differences ($p<0.002$) between *Cst1.5* and the other functions. However, the significant difference between *OSX* and *Win7* disappears (*Cst1.5*: 2.29 s, *OSX*: 1.88 s, *Win7*: 1.81 s, *Xorg*: 1.87 s). For W_{1pix}, we observed significant differences ($p<0.048$) between *OSX* and *Cst1.5*, *Xorg* (*Cst1.5*: 2.85 s, *OSX*: 2.46 s, *Win7*: 2.63 s, *Xorg*: 2.69 s).

Our results show there is no single function that is best for all target widths. *Win7* and *Xorg* improve movement time by more than 9% compared to *OSX* and more than 24% compared to *Cst1.5* for widths 6 and 9 pixels. However, the difference with *OSX* disappears for 3 and 1 pixels targets. In contrast, *OSX* improves movement time by 13% compared to *Cst1.5* and 8% compared to *Xorg* for the 1 pixel target when the other functions do not show significant differences.

Qualitative Feedback

At the end of the experiment, participants were asked if they found some differences in the control of the cursor. All noticed there was a condition (*Cst1.5*) where they had to move the mouse over greater distances to reach targets and complained it was less comfortable than the other conditions. None of the participants was able to notice a difference between the three other conditions (*Win7*, *OSX* and *Xorg*).

DISCUSSION

The knowledge acquired by studying, replicating and comparing the transfer functions used by three different systems brings us to the following suggestions.

Choosing a Baseline Transfer Function

The use of a constant CD gain function as a baseline to compare with other techniques should be prohibited unless clearly justified. None of the prevalent systems uses such a function by default and it might not even be obtainable on some, like OS X. As we explained in the previous section, most participants of our experiment declared using the default settings of their system, and results from the experiment show that these default functions outperform a constant CD gain. We thus recommend to use the default transfer function of the considered system as a baseline condition.

Reporting Transfer Functions

To facilitate the replication or reproduction of pointing techniques and experiments, we recommend that researchers provide extensive details concerning the transfer function(s) they used.

If constant CD gains were used, we recommend to report them using unitless values in order to abstract them from hardware specifics. Detailing how a constant gain was achieved might also help detecting potential flaws in the methodology. As an example, it might be important to explain how remainders were handled as it remains unclear if they can affect performance. If a system function was used, the system and its particular configuration settings should be unambiguously described. A screen-shot is probably the most unambiguous way of reporting these settings. For completeness and as we have shown that some systems do not take them into account, we also recommend to report the resolution and frequency of both the input and output devices. For custom non linear functions, we recommend describing them using figures or tables with physical units mapping the device speed to the cursor speed or CD gain.

An alternative for describing transfer functions would be to use a notation based on URIs, similar to what we have started to do in `libpointing`. URIs are interesting because they allow to combine a class description, an instance description and optional parameters, e.g. `windows:vista?setting=2&epp=false`. They can be given fully expanded, with all possible parameters, or in a condensed form specifying only the ones differing from default values. If `libpointing` indeed turns into a living archive for transfer functions, we will certainly need some registry to standardize and officialize these notations.

In addition to the transfer function(s) used, the latency of the system might also be worthwhile to report as it can impact performance and might be a confounding variable. We acknowledge that measuring it is quite difficult. But some of the parameters that affect it can probably be described, such as the characteristics of the communication link between the input device and the computer, or the synchronization characteristics of the display.

Configuration Interfaces and Documentation

As we already stated, transfer functions should be defined in hardware-independent ways. Their implementation should thus be given either hardware-independent data, or the information to do the required conversions. In an ideal world, pointing device manufacturers would take full advantage of the HID specification to put all the necessary information in their device descriptions. Unfortunately, the reality is quite different... As systems are often forced to make educated

[11] Using Bonferroni correction for all post-hoc analysis.

guesses about device characteristics, we believe these should be visible and modifiable in the relevant configuration interfaces. Exposing wrongly estimated values should help raise the level of consciousness of the public about the difference between input resolution and cursor speed, for example.

Considering the different understandings of the current interfaces used for tuning the system transfer functions, even among researchers, we believe these interfaces should at least be properly documented if not completely redesigned.

CONCLUSION AND FUTURE WORK

In this paper, we presented a custom device and a toolkit that helped us characterize, replicate and compare the pointing transfer functions used on a daily basis by millions of people around the world. We showed in a controlled experiment that the default transfer functions used in Windows, OS X and Xorg outperform a constant CD gain similar to those used by most researchers. Our results also show a significant interaction between transfer function and target width suggesting that more work needs to be done to understand how these functions affect performance.

This work represents an important step in the understanding and study of pointing transfer functions. A long term goal is to improve the design of transfer functions by taking more into account the hardware characteristics (i.e. mouse vs. touchpad, desktop display vs. wall size display) and the motor capabilities of the users. This includes the study of management strategies for remainders which we hypothesize can be important for the selection of small targets. We are also interested in the study of the impact of the transfer function in relation with the task: a function performing well for pointing could degrade performance in other tasks like drawing, steering or executing command gestures, for example.

In the short term, we plan to study the interaction of the default transfer functions with pointing facilitation techniques manipulating CD gain, e.g. [5, 10, 23]. To our knowledge, these technique were only implemented on top of a constant CD gain and were also only evaluated against constant CD gains. We also plan to investigate the use of indirect mappings on multitouch interfaces. On this topic, Buxton recently said: "*one of the things that I see most neglected is any consideration of when to use relative vs absolute control and varying, including when and how to effectively and dynamically switch from one to the other, and when and how to dynamically adjust C:D ratio*" [6]. We could not agree more.

ACKNOWLEDGMENTS

We thank Damien Marchal and Mark Cranness for their help in the understanding of transfer functions and their contribution to `libpointing`.

REFERENCES

1. Pointer ballistics for Windows XP. Archived white paper, Windows Hardware Developer Center, Oct. 2002.
2. T. Asano, E. Sharlin, Y. Kitamura, K. Takashima, and F. Kishino. Predictive interaction using the Delphian Desktop. *Proceedings of UIST'05*, 133–141. ACM, 2005.
3. M. Bergman, T. Peurach, T. Schmidt, S. McGowan, J. Crowe, R. Dezmelyk, R. Zimmermann, M. Van Flandern, B. Nathan, M. Davis, and J. Rayhawk. Device class definition for human interface devices (HID). Version 1.11, USB Implementers' Forum, June 2001.
4. T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): generic syntax. RFC 3986, IETF, Jan. 2005.
5. R. Blanch, Y. Guiard, and M. Beaudouin-Lafon. Semantic pointing: improving target acquisition with control-display ratio adaptation. *Proceedings of CHI'04*, 519–526. ACM, 2004.
6. W. Buxton, M. Billinghurst, Y. Guiard, A. Sellen, and S. Zhai. *Human input to computer systems: theories, techniques and technology*. 2011. Working draft, `http://www.billbuxton.com/inputManuscript.html`.
7. G. Casiez and D. Vogel. The effect of spring stiffness and control gain with an elastic rate control pointing device. *Proceeding of CHI'08*, 1709–1718. ACM, 2008.
8. G. Casiez, D. Vogel, R. Balakrishnan, and A. Cockburn. The impact of control-display gain on user performance in pointing tasks. *Human-Computer Interaction*, 23(3):215–250, 2008.
9. O. Chapuis, J.-B. Labrune, and E. Pietriga. DynaSpot: speed-dependent area cursor. *Proceedings of CHI'09*, 1391–1400. ACM, 2009.
10. A. Cockburn and A. Firth. Improving the acquisition of small targets. *Proceedings of HCI'03*, 77–80. BCS, 2003.
11. E. D. Graham. Virtual pointing on a computer display: nonlinear control-display mappings. *Proceedings of GI'96*, 39–46. Canadian Information Processing Society, 1996.
12. T. Grossman and R. Balakrishnan. The bubble cursor: enhancing target acquisition by dynamic resizing of the cursor's activation area. *Proceedings of CHI'05*, 281–290. ACM, 2005.
13. K. Hinckley. Input technologies and techniques. A. Sears and J. A. Jacko, editors, *Human Computer Interaction Handbook (2nd edition)*. CRC Press, Sept. 2007.
14. H. D. Jellinek and S. K. Card. Powermice and user performance. *Proceedings of CHI'90*, 213–220. ACM, 1990.
15. M. Kobayashi and T. Igarashi. Ninja cursors: using multiple cursors to assist target acquisition on large screens. *Proceeding of CHI'08*, 949–958. ACM, 2008.
16. I. S. MacKenzie. Input devices and interaction techniques for advanced computing. W. Barfield and T. A. F. III, editors, *Virtual environments and advanced interface design*, 437–470. 1995.
17. I. S. MacKenzie and P. Isokoski. Fitts' throughput and the speed-accuracy tradeoff. *Proceeding of CHI'08*, 1633–1636. ACM, 2008.
18. R. L. Mandryk and C. Gutwin. Perceptibility and utility of sticky targets. *Proceedings of graphics interface 2008*, GI '08, 65–72. Canadian Information Processing Society, 2008.
19. D. E. Meyer, R. A. Abrams, S. Kornblum, C. E. Wright, and J. E. K. Smith. Optimality in human motor performance: Ideal control of rapid aimed movements. *Psychological Review*, 95(3):340–370, 1988.
20. B. Moggridge. *Designing interactions*. The MIT Press, Oct. 2006.
21. U. Tränkle and D. Deutschmann. Factors influencing speed and precision of cursor positioning using a mouse. *Ergonomics*, 34(2):161–174, 1991.
22. J. O. Wobbrock, J. Fogarty, S.-Y. S. Liu, S. Kimuro, and S. Harada. The Angle Mouse: target-agnostic dynamic gain adjustment based on angular deviation. *Proceedings of CHI'09*, 1401–1410. ACM, 2009.
23. A. Worden, N. Walker, K. Bharat, and S. Hudson. Making computers easier for older adults to use: area cursors and sticky icons. *Proceedings of CHI'97*, 266–271. ACM, 1997.