Hardware-Embedded Pointing Transfer Function Capable of Canceling OS Gains

Seonho Kim
Yonsei University
Seoul, Republic of Korea
cogbrain18@yonsei.ac.kr

Munjeong Kim
DGIST
Daegu, Republic of Korea
moondoong@dgist.ac.kr

Jonghyun Kim
Yonsei University
Seoul, Republic of Korea
truejong1@yonsei.ac.kr

Donghyeon Kang
Yonsei University
Seoul, Republic of Korea
car991231@yonsei.ac.kr

Sunjun Kim*
DGIST
Daegu, Republic of Korea
sunjun_kim@dgist.ac.kr

Byungjoo Lee*
Yonsei University
Seoul, Republic of Korea
byungjoo.lee@yonsei.ac.kr

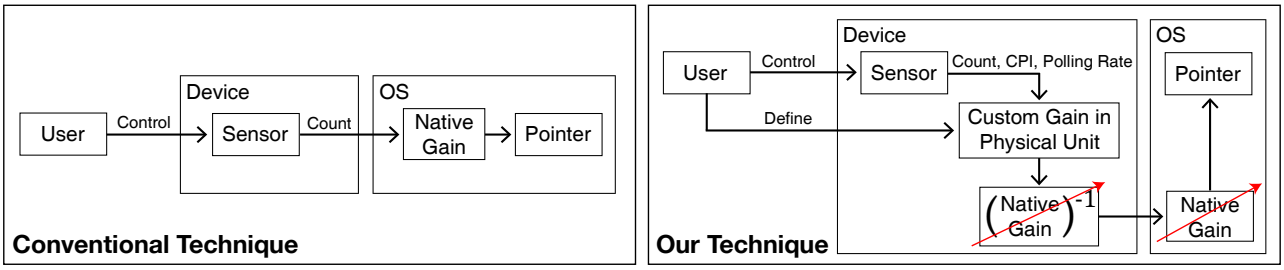

Figure 1: (Left) conventional pointing transfer function method, (Right) hardware-embedded transfer function method

Abstract

When using indirect pointing devices in modern operating systems (OS), users' perception of the pointing transfer function is easily influenced by the device's hardware or OS-native transfer function settings. This could hinder users from finding and fully adapting to the transfer function that is optimal for them. We propose a novel hardware-embedded transfer function technique that is expected to allow users to consistently experience the desired function even when device hardware or OS settings change. The technique (1) allows users to define the desired function within the device firmware in physical units and (2) enables the firmware to cancel out the influence of OS-native functions and hardware setting perturbations, so that the uploaded function can persist regardless of the external environment. Through technical evaluation including transfer functions of various shapes, we showed that the proposed technique has comparable robustness and accuracy to the conventional approach.

CCS Concepts

• **Human-centered computing** → **Pointing devices**.

Keywords

Esports, Pointing, Gain Function, Fitts' Law

*Co-corresponding authors

CHI '25, Yokohama, Japan

ACM ISBN 979-8-4007-1394-1/25/04
https://doi.org/10.1145/3706598.3714076

ACM Reference Format:
Seonho Kim, Munjeong Kim, Jonghyun Kim, Donghyeon Kang, Sunjun Kim, and Byungjoo Lee. 2025. Hardware-Embedded Pointing Transfer Function Capable of Canceling OS Gains. In *CHI Conference on Human Factors in Computing Systems (CHI '25), April 26–May 01, 2025, Yokohama, Japan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3706598.3714076

1 Introduction

When using indirect pointing devices such as a mouse or trackpad, a pointing transfer function determines the relationship between the physical speed of the device s_d and the speed at which the pointer moves on the screen s_p [2, 12]. If we compute the ratio of two speeds at a particular time t, that is the pointing gain G ($=s_p/s_d$), and pointing transfer functions are therefore also called gain functions [7, 19]. An acceleration gain function increases the gain as the device speed increases, while a constant gain function maintains the gain regardless of the device speed [6].

Pointing transfer function design has a significant impact on on-screen target acquisition performance [6]; providing a gain that is neither too high nor too low [1, 6] and providing an acceleration function rather than a constant function can result in higher performance [6, 12]. Meanwhile, pointing behaviors can vary significantly depending on users' cognitive and physical characteristics and motivational states (e.g., speed-accuracy bias) [3], so there may not be a single transfer function that satisfies all interaction scenarios and all users [12]. Accordingly, most operating systems (OS) today allow users to select from a number of transfer function presets (accelerated or constant) with different levels, rather than providing a single fixed function (see Figure 2).

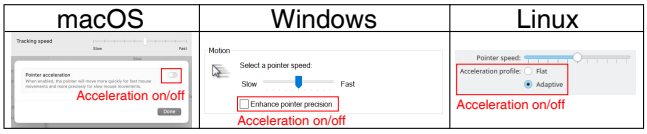

Figure 2: Pointing transfer function customization options in modern OSs

While previous studies [2, 6, 12, 15] have mainly focused on the design of transfer functions in general office interaction scenarios with moderate performance pressure, in this study we critically examine how modern transfer functions can support users who aim for *extreme* level of performance. Consider professional athletes in desktop competitive video games (i.e., esports) [5, 11, 13, 17, 22]; competitions between those athletes are highly active today, with millions of dollars in prize money and millions of viewers around the world. Unlike everyday interactions in the office, even a slight performance deficit can result in a critical defeat in competitive video games, so they are looking for a more suitable game setting that can boost their performance even a little [9, 13, 17].

For example, recent studies [14, 21] showed that competitive video game players turn off most of the graphical options because it allows them to reduce latency through a small increase in frame rate [20], and improve enemy recognition through a small reduction in visual clutter [18]. Our main question in this study, then, is: What pointing transfer function features should be available to users who seek maximum input performance and welcome even the smallest performance improvements?

The first desired feature is probably the maximum level of *customizability* to accommodate the different physical and cognitive characteristics and motivational states of users. The pointing transfer function is a continuous function of the device speed s_d and can therefore be determined in infinitely many different ways. If the user could draw the function shape directly or define it as a cubic spline based on a few control points, it would be easy for the user to explore and find the function that best suits them. However, no OS or device provides such a feature yet. Currently, users can only passively select one of about 20 transfer functions in a form predefined by the OS [2]. This may have led to a culture where serious

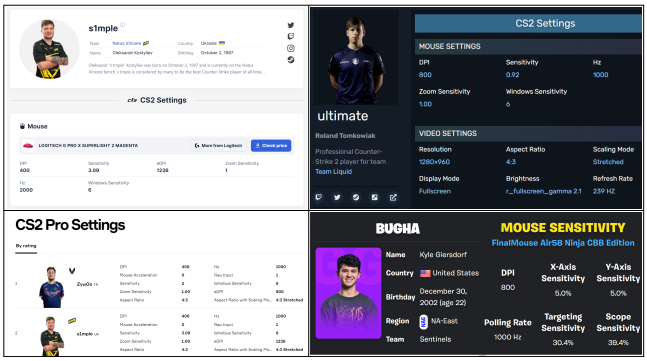

Figure 3: Websites that allow anyone to reference the mouse settings of professional esports athletes

gamers today simply follow the recommendations or settings of other players [17] rather than spending enough time exploring the transfer function that suits them (see Figure 3).

The second desired feature is that the transfer function designed by users should be *sustainable* over a long period of time without being disturbed by external factors until users are sufficiently trained on it. However, a feature of modern computing environments [7] makes this feature difficult to provide: the OS-native transfer functions available today are defined as functions of logical counts rather than physical speed units (e.g., m/s). This causes users to experience a completely different transfer function in *physical* units if the hardware settings of the device, such as sensor sensitivity (i.e., counts per inch, CPI) or polling rate (in Hz), change for any reason (see Figure 4). Figure 6 illustrates how the gain function that users actually experience in physical units changes under each condition of OS-native constant, linear acceleration, and nonlinear acceleration gain function when the CPI and polling rate of the indirect pointing device are changed. In particular, if the OS native function is set as an accelerated one, the overall scale and shape of the perceived gain function is disrupted by both CPI and polling rate [7], which we speculate may be the reason why professional video gaming athletes do not use the accelerated function [1, 4] despite its performance benefits observed in previous studies [6, 12].

We could wait for modern OSs to provide transfer function features that can support high-performance users [7], but in this study, we would like to preemptively suggest a solution that is more immediate and independent of existing OSs (see Figure 1). More specifically, we propose to upload the following two components to the firmware of the device: (1) a transfer function defined in physical units that can be fully customized by users, and (2) an algorithm that cancels out the influence of OS-native transfer functions and hardware setting perturbations, so that the uploaded transfer function can persist regardless of the external environment. With input devices equipped with this solution, users can experience the desired transfer function by simply plugging in the input device into most modern computing environments, regardless of OS, CPI, or polling rate settings. Our proposed solution is realized in this study based on a custom computer mouse equipped with a fully programmable chip[1]. In a full-factorial technical evaluation performed on two CPIs, two polling rates, and six OS-native transfer functions, we demonstrate that the proposed technique can realize a variety of targeted transfer functions while maintaining robustness and accuracy comparable to the conventional system.

2 Background and Related Work

2.1 Comparison with Existing Solutions

Several previous studies [2, 7] in HCI have also criticized the way pointing transfer functions are defined and used today. Those studies have focused on *scientific replicability* issues in HCI research rather than considering the needs of high-performance users. According to those studies, in order for user experiments using OS-native transfer functions (defined in logical units) to be replicated,

[1]Our proposed solution is lightweight enough to be integrated into the firmware of existing gaming mice, but in this study we chose not to hack existing devices in order to achieve a transparent and controllable implementation.

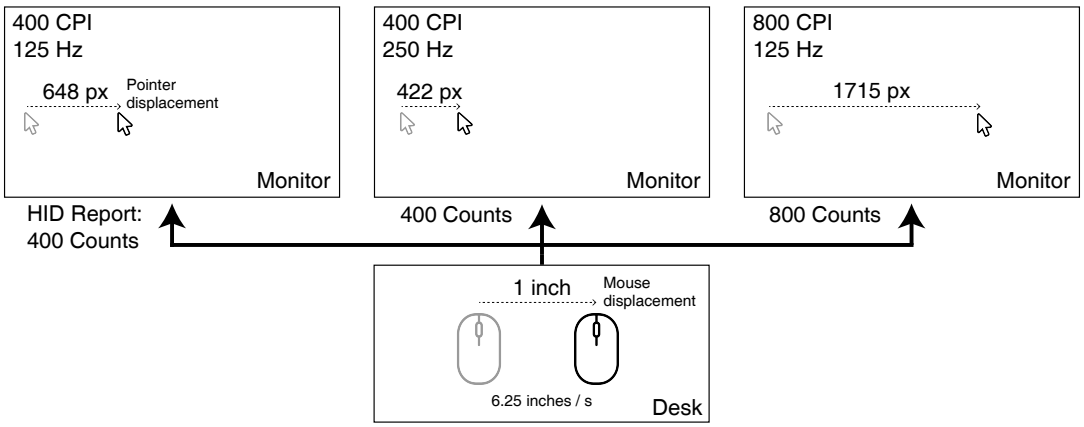

Figure 4: As noted by Hanada et al. (2021) [7], even with the same physical mouse movement, the resulting pointer displacement can vary significantly depending on mouse device settings. In this case, a mouse moved 1 inch at a constant speed of 6.25 inches per second over 160 ms (the mouse sensitivity slider set to tick 6 with "Enhance Pointer Precision" enabled in Windows)

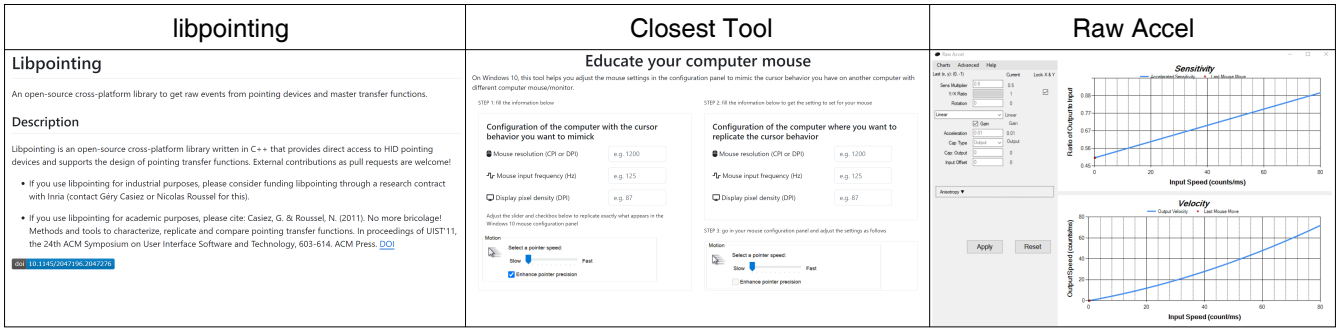

Figure 5: Existing transfer function solutions: (left) `libpointing`, (middle) `Closest` tool, (right) `Raw Accel`

the resolution of the display and hardware settings must all be explicitly stated in the paper, which is not common. This is essentially the same story as our concern that the transfer function experienced in physical units is easily disturbed by external factors [7], making it difficult for high-performance users to sufficiently adapt to the desired function.

To address the replicability problem, two solutions have been proposed. (see Figure 5). The first is to bypass the OS-native transfer function by using system APIs such as RAWINPUT which allow intercepting the native mouse events from the device before the system transfer function is applied. Through this method, `libpointing` library released in 2011 [2] successfully implemented various custom transfer functions independently from the OS-native transfer functions. However, this method still does not completely prevent unintended disturbances in hardware settings from significantly affecting users' transfer function experience because the device's CPI or polling rate must be manually notified to the software. It also has limited control over the position of the pointer because in Windows, it alters `SmoothMouseXCurve` and `SmoothMouseYCurve` registry values to apply the customized transfer function, but it only has four control points, which cannot faithfully apply the desired shape of the function. Instead, it may be possible to fully control the pointer using an OS API like `SetCursorPos` [12], but many games block software-based pointer control APIs due to security issues or allow only internal raw input channels[2]. Therefore, we believe that the solution is difficult to use for high-performance users. The hassle of having to pre-install the library also makes it difficult to use widely.

The second solution, called `Closest` tool [7], is to suggest to users how the OS-native function should be set (e.g., which tick on the slider to select) to get the most similar experience to a desired transfer function defined in physical units. However, this approach still burdens users with installing separate software and manually notifying the system of changes in their computing environment or device settings. Furthermore, it does not allow for sufficient customization of transfer functions to satisfy high-performance users who cannot find what they want among the small number of predefined OS-native functions.

Meanwhile, outside of the realm of HCI research, in the video game community, a tool called `Raw Accel`[3] has been developed with a goal closer to ours: supporting high-performance users. The tool manipulates HID reports coming from input devices through a low-level kernel driver to implement a desired transfer function.

[2]Popular games like: Apex Legends, PUBG, Call of Duty, Call of Duty: Warzone
[3]https://github.com/a1xd/rawaccel

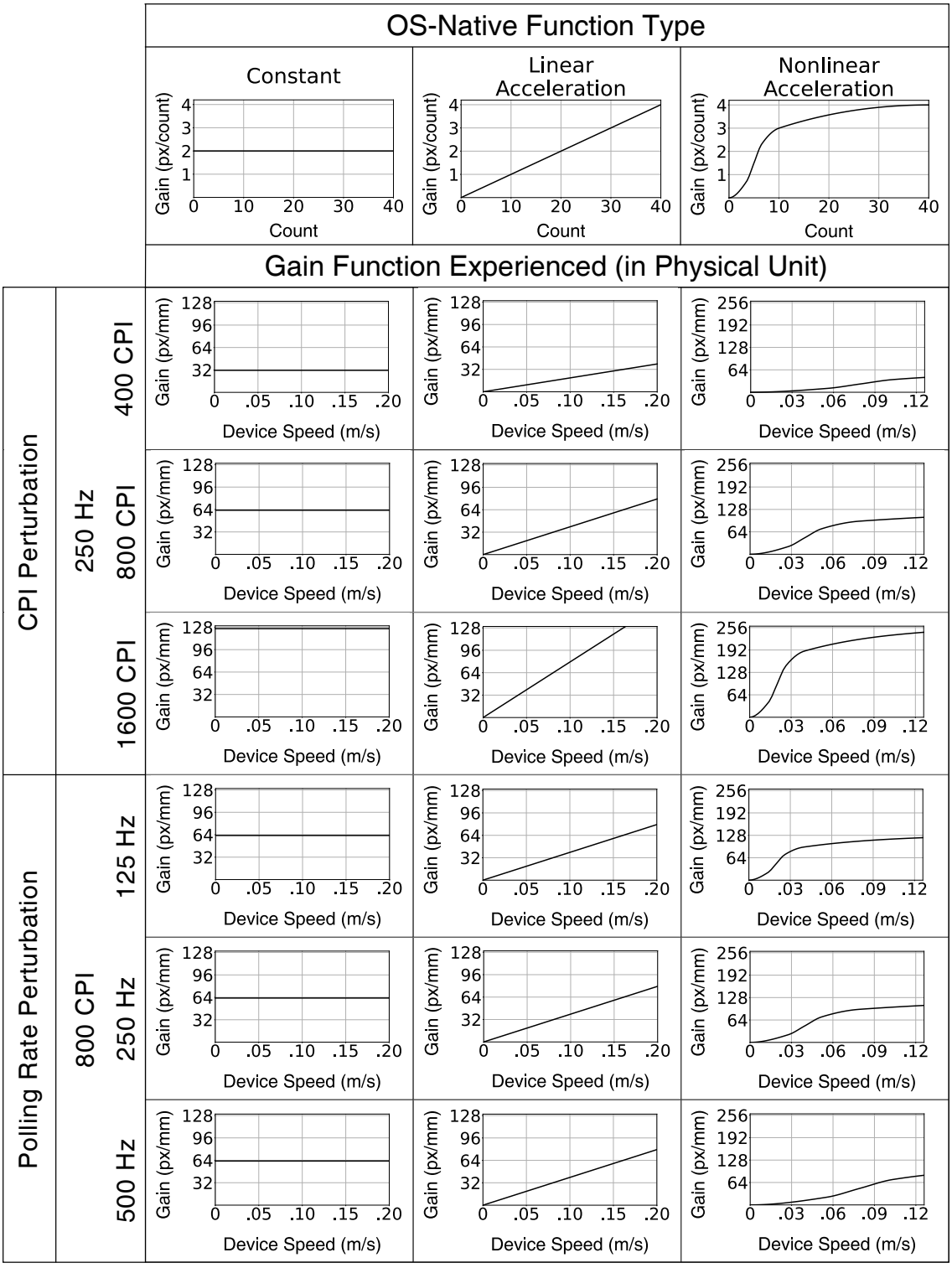

Figure 6: We calculate and illustrate how the gain function in physical units changes under different OS-native gain functions—constant, linear acceleration, and nonlinear acceleration—by varying the CPI and polling rate of the indirect pointing device. For the constant type, the scale changes with CPI. For the linear acceleration type, both scale and shape change with CPI. For the nonlinear acceleration type, both scale and shape change with CPI, and they also change with polling rate.

Table 1: Comparison of the transfer function technique proposed in this study with existing solutions

Technique	Hardware Setting Compensation	Customizability	Additional Software	Anti-Cheat Block Possibility
`libpointing [2]`	Manual	High	Needed	High[4]
`Closest [7]`	Manual	Low	Needed	Low
`Raw Accel`	Manual	High	Needed	Medium
Ours	Automatic	High	Unneeded	Low

This allows users to be relatively free from security blocking issues compared to tools that rely on high-level system APIs. However, the tool still requires users to install separate software, perform manual calculations to compensate for the effects of input device settings, and may be blocked by more sophisticated anti-cheat mechanisms. Additionally, the fact that `Raw Accel` cannot work on OSs other than Windows prevents it from being a universal solution for high-performance users.

Unlike previous solutions, the transfer function technique proposed in this study can automatically respond to hardware setting changes because all processes are implemented within the hardware firmware. Furthermore, it allows full customization of transfer functions regardless of the type of OS without installing separate software or libraries. Finally, since the signals from the input device equipped with our solution are indistinguishable from the signals from conventional devices from the system's perspective, our solution can be widely adopted in applications that impose some restrictions on the pointer control mechanism. Table 1 further highlights the differences between ours and the existing ones.

2.2 Preliminary Survey with Serious Gamers

The main motivation for this study is that today's OS-native transfer functions do not satisfy high-performance users. To support this claim, we conducted a preliminary survey of 17 serious gamers (μ=15.94 years, σ=2.39). All survey participants were players of first-person shooters (FPS, N=12) or multi-player online battle arena (MOBA, N=5) games whose primary input device was a computer mouse, and were randomly recruited from a local esports academy. One of the participants was a current professional esports athlete, and two were trainees aiming for a professional debut. The in-game tiers of the remaining participants were one in the top 0.2%, four in the top 2%, eight in the top 12%, and one in the top 35%. Participants were compensated with a gift worth 4 USD. The survey consisted of 44 questions that investigated the overall experience related to mouse use, of which seven questions were directly related to the motivation of this study. The survey results for those questions are:

- Q1. Are you using an acceleration pointing transfer function? (Yes: 3 / 17.6%)
- Q2. Do you think your gaming skills deteriorate if you use a mouse that you don't normally use? (Yes: 15 / 88.2%)
- Q3. Do you think different game genres require different mouse settings? (e.g., CPI) (Yes: 13 / 76.5%)
- Q4. Do you need different mouse settings (e.g., CPI) depending on the type of character or weapon you control in the game? (Yes: 5 / 29.4%)
- Q5. Have you ever tried to mimic another player's mouse settings? (Yes: 11 / 64.7%)
- Q6. Have you ever wanted more control over the transfer function of your mouse? (Yes: 9 / 52.9%)
- Q7. Please select all the features below that you think are necessary but not present in today's gaming mice (transfer function optimization: 13, 76.5%)

In summary, these results support our claim that modern pointing transfer functions need to be significantly improved to satisfy high-performance users. Serious gamers want to be able to fine-tune their pointing transfer function (Q3, Q4, Q6) and seem to be highly interested in finding the optimal device settings that best suit them (Q2, Q5, Q7). Despite the general advantages of the acceleration function, we confirm once again that it is hardly used (Q1), probably because an acceleration function can be more severely disturbed by changes in CPI or polling rate than a constant function.

2.3 Pointing Transfer Function for Gamers

For enthusiastic gamers, especially in FPS games, the general consensus of mouse settings is probably to "turn off mouse acceleration", which refers to a constant CD gain setting [1, 4]. The reason for this is probably to get rid of an extra variable – see Figure 6 for how easily nonlinear acceleration functions can be perturbed by external variables – to develop consistent muscle memory, which is crucial for better aiming performance. In particular, competitive FPS players have avoided the non-linear transfer function, who often use open-loop targeting strategies and rely on muscle memory for better repeatability of the action [1]. For precise movement, low gain is preferred [17], but because of the constant gain, they require a large mouse pad mousepad and large motion for specific actions such as a quick 180 degree turn in FPS games.

While constant gain is preferred, attempts to apply nonlinear transfer functions have been advocated by a subset of gamers. If gamers could make a consistent open-loop action with the acceleration transfer functions, they could have the best of both worlds: slow and precise movement, and fast and large sweep. The community has shown significant interest in a customizable transfer function. For example, `Raw Accel` GitHub repository has 200+ forks and 1.8k stars (as of late 2024), and a YouTube video[5] about how to set up proper transfer function for Valorant gameplay gained nearly 1 million views in less than two years. A variety of custom transfer

[4]Since `libpointing` has limited ability to apply a custom OS-level transfer function, application-level mouse message injection is required to have full control of the cursor, which has a higher chance of being filtered by an anti-cheat detector.

[5]https://www.youtube.com/watch?v=u9auSOa3Hb0

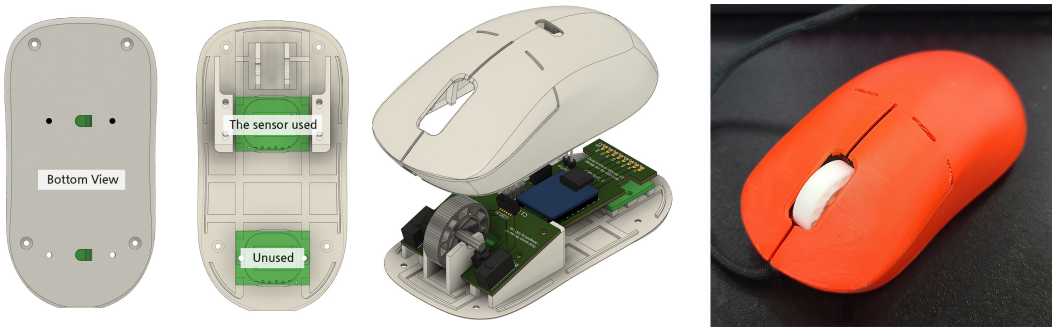

Figure 7: The custom-built mouse hardware design: Only the signal from front sensor was used in this study.

functions have been tried, such as jump curve, linear curve, power curve, and exponential curves [16]. The most prominent approach may be a two-step approach, which maintains a constant gain at low speeds and applies an increased gain after a speed threshold.

In summary, while academic research has demonstrated the performance benefits of non-linear transfer functions, particularly mouse acceleration (increased gain at higher speeds) in general office scenarios [3, 6, 12, 15, 19], their advantages have not been thoroughly evaluated in high-performance gaming contexts. We believe that the absence of a sustainable and sufficiently customizable transfer function technique has contributed significantly to this, and we hope that this study's technique can be useful in future studies on transfer function design for high-performance users.

3 Technique Implementation

3.1 Overview

Our transfer function technique is implemented directly in the firmware of an input device hardware. Among various indirect pointing devices, we demonstrate our technique on a computer mouse, the most widely used input device in competitive video games today [8]. To do this, rather than hacking the firmware of an existing commercial mouse, we decided to create a mouse hardware that we can fully control ourselves. This allows for a more rigorous technical evaluation that minimizes the influence of any possible confounding variables. The following sections describe the mouse hardware we implemented in detail, followed by the firmware uploaded to it.

3.2 Custom Mouse Hardware

The custom mouse used in this study was created based on 3D printer drawings and `Arduino` source code that were released as open source on the web[6]. The resources are basically for those who want to implement a dual-sensor mouse, a special mouse with two optical sensors, and have been usefully utilized in recent HCI studies [8, 10, 17]. We implemented the dual-sensor mouse according to the guide and used only one of the two sensors, which is closer to the fingertip (see Figure 7). The total cost of producing one dual-sensor mouse in this study was about 100 USD [8]. Considering that gaming mice costing several hundred dollars are popular on the market today, this is affordable for serious gamers to make one themselves.

[6]https://github.com/SunjunKim/DualSensorMouse

The following sections describe the hardware specifications of the mouse in more detail.

3.2.1 Mouse Shell. The mouse shell shape mimics the Logitech G Pro Wireless, which is one of the most popular model for gamers. The shell was printed using a low-cost FDM printer (Bambu Lab X1-Carbon) in PLA material.

3.2.2 Optical Sensor. For the sensor, we used PixArt PMW3389DM-T3QU optical mouse sensor. The sensor has a maximum resolution of 16,000 CPI (count per inch), a tracking rate of 12,000 frames per second, and a tracking speed of up to 400 IPS (inch per second) with 50G acceleration. The PMW3389 sensor is featured in high-end gaming mice such as the Razer DeathAdder Elite and the Coolermaster MM710, and has been widely praised by competitive gamers for its performance.

3.2.3 Microprocessor. As the brain of the mouse, the Espress ESP32-S3 microprocessor handles sensor reading and communication with the host computer. The ESP32-S3 is a powerful microcontroller with a dual-core Xtensa LX7 CPU clocked at 240 MHz, 512 KB SRAM and 4 MB flash memory. It also features a native USB interface, which is ideal for a mouse application. This is a 32-bit RISC architecture microprocessor, similar to other MCUs for gaming mice on the market today in 2024 (e.g., Nordic nRF52840, nRF52833, etc.).

3.2.4 Firmware. We implemented the mouse driving firmware on Arduino IDE. The MCU reads the sensor displacement values from PWM3389 devices via SPI communication, processes sensor data, and sends the USB HID (human interface device) report to the host computer. To facilitate an advanced function, the USB HID descriptor was extended to report the X and Y displacements in 16-bit (instead of the standard 8-bit), covering the range from -32,768 to 32,767 counts. Debug and log messages are sent separately to the host computer via the `USBCDC` library, which acts as a virtual serial device. The device could run up to 1,000 Hz polling rate (`bInterval=1` in a full-speed USB device), and we simulated the other polling rates by adding artificial delay in the loop.

3.3 Hardware-Embedded Pointing Transfer Function Defined in Physical Units

In our solution, let us denote the pointing gain function that users want to experience as $G()$, where $G()$ is defined as a function of the physical speed v (i.e., in m/s unit) that the mouse translates

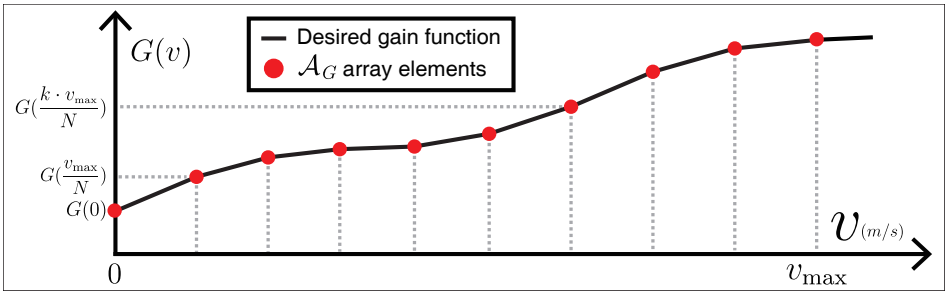

Figure 8: The way a custom desired gain function $G(v)$ is defined and uploaded to the firmware as an array $\mathcal{A}_G$ in our technique

over the desk surface. Then, the speed of the pointer movement on the screen s (i.e., in px/s unit[7]) at the instant when the mouse is moving at speed v should be determined as follows:

$$s = G(v) \cdot v \tag{1}$$

Note that the unit of gain $G(v)$ in this formulation is px/m. We assume that this desired gain function is discretized and uploaded to the firmware via serial communication. More specifically, if the theoretical maximum physical speed of the mouse is v_{max}, the gain function discretized into N points is uploaded to the firmware as the following array:

$$\mathcal{A}_G = [G(0),\ G(\frac{v_{max}}{N}),\ G(\frac{2v_{max}}{N}),\ \cdots,\ G(v_{max})] \tag{2}$$

In this study, v_{max} was determined to be 1 m/s, referring to previous studies [2, 12]. N was set to 100 so that the transfer function could be customized in sufficient detail (see Figure 8).

The physical speed of the mouse v can be precisely estimated by analyzing the optical sensor readings. If the polling rate of the mouse is expressed as P (in Hz), then mouse sensor readings are obtained every $1/P$ seconds. If the values of the i-th sensor reading are assumed to be cx_i and cy_i in the x and y axes of the sensor (unit: counts), respectively, then the estimated speed of the mouse $\hat{v}_i$ at that moment is calculated as follows:

$$\hat{v}_i = 0.0254 \cdot \frac{P\sqrt{cx_i^2 + cy_i^2}}{CPI} \tag{3}$$

Here, CPI is the sensitivity setting of the mouse at the time the sensor reading was created, and 0.0254 is a proportional constant introduced to convert inches to meters. Finally, the gain G_i to be applied to the i-th sensor reading (cx_i, cy_i) is the linearly interpolated value at the fractional index corresponding to $\hat{v}_i$ in the array of gains in Equation 2. If $\hat{v}_i$ is greater than v_{max}, G_i is simply set to $G(v_{max})$.

According to the definition of the gain function in Equation 1, the pointer speed on the screen s_i that should be generated correspondingly from the i-th sensor reading is as follows:

$$s_i = G_i \cdot \hat{v}_i \tag{4}$$

[7]We use px/s here because the size of screen elements is defined in pixels and manipulated by a pointer with pixel-level precision. In this case, we believe that defining the pointer speed in physical units is less meaningful because the physical screen size can vary while maintaining a field of view (FoV) similar to the eyes (e.g., the same screen content rendered on a laptop screen, desktop monitor, and projector at different distances). Further rationale will be presented in the Discussion and Limitation section.

Since each sensor reading occurs over a period of $1/P$, if we want to move the pointer at an average speed of s_i as above, the corresponding displacement of the pointer d_i on the screen must be:

$$d_i = \frac{s_i}{P} \tag{5}$$

From the equations presented above, we were able to understand how much the pointer should be displaced on the screen for each sensor reading to realize the gain function G desired by users. In particular, since the influence of hardware settings such as CPI or polling rate is automatically compensated internally in the firmware (Equation 3), if the above equations can be actually implemented, users can always experience the same transfer function in physical units regardless of hardware setting perturbations.

3.4 OS Gain Cancellation

The process by which the pointer is moved in today's OSs is as follows. First, for the i-th sensor reading step, the input device sends an Human Interface Devices (HID) report in the form of an integer vector $h_i = (hx_i, hy_i)$ to the OS through USB Bus. Next, the OS calculates the magnitude of the report $|h_i|$ and multiplies it by the corresponding OS-native gain G_{OS} to determine the required pixel displacement $(\Delta x_i, \Delta y_i)$ of the pointer. Finally, the displacement is reflected in the pointer position on the screen. The relationship between the pointer displacement vector and the HID report vector is expressed as follows:

$$(\Delta x_i, \Delta y_i) = G_{OS}(|h_i|) \cdot (hx_i, hy_i) \tag{6}$$

One thing to note here is that depending on the OS, the native gain function may compute the magnitude of the HID report it takes as input in a way other than the familiar Euclidean norm. For example, on Windows, $|h_i|$ is computed in a unique way:

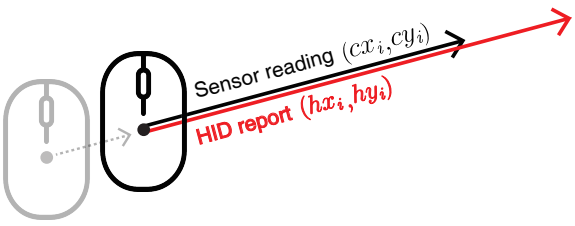

Figure 9: Sensor reading and HID report should be parallel to avoid jittering

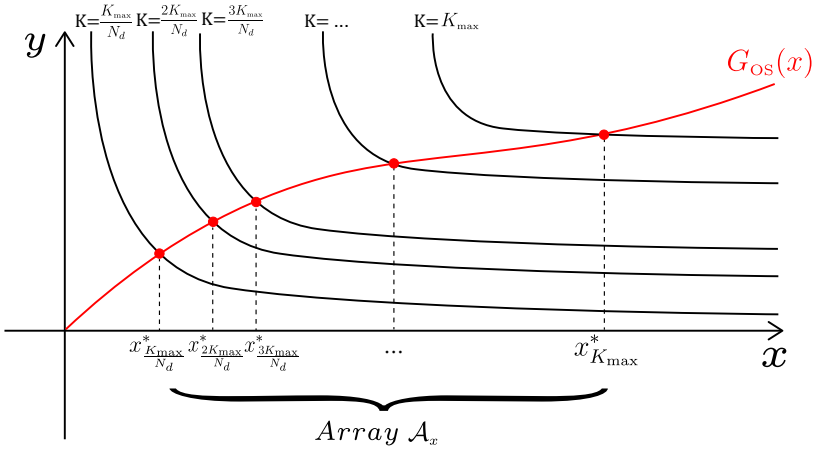

Figure 10: The process of of obtaining the array $\mathcal{A}_x$

$\max(hx_i, hy_i) + \min(hx_i, hy_i)/2$ [2]. In this study, we assume that all magnitude expressions written in the G_{OS} function represent magnitudes computed according to the method of the target OS.

We assume that G_{OS} is uploaded to the mouse in the form of an array, just like the user-defined gain G. This is a possible assumption, since the native transfer functions of most OSs have already been precisely measured in previous studies and made public [2]. If we denote the maximum HID report magnitude we consider as h_{max}, the G_{OS} array is expressed as follows:

$$\mathcal{A}_{OS} = [G_{OS}(0),\ G_{OS}(\frac{h_{max}}{N_{OS}}),\ G_{OS}(\frac{2h_{max}}{N_{OS}}),\ \cdots,\ G_{OS}(h_{max})] \quad (7)$$

The divisor N_{OS} is set to 1,000, $G_{OS}(0)$ is simply assumed to be 0, and h_{max} was set to 500.

In the previous section, we derived the on-screen pointer displacement d_i that should occur from the i-th sensor reading, which should be related to $(\Delta x_i, \Delta y_i)$ and G_{OS} as follows:

$$d_i = \sqrt{\Delta x_i^2 + \Delta y_i^2} = G_{OS}(|h_i|) \cdot |h_i| \quad (8)$$

If we can find h_i that satisfies the above equation, we can realize the desired pointer movement d_i.

To avoid users noticing directional jittering in pointer movement, (hx_i, hy_i) should always be set parallel to the sensor reading $c_i = (cx_i, cy_i)$. As a result, (hx_i, hy_i) should always be determined by multiplying (cx_i, cy_i) by a constant k_i, as follows (see Figure 9):

$$(hx_i, hy_i) = k_i \cdot (cx_i, cy_i) \quad (9)$$

Here, (cx_i, cy_i) are known values measured from the sensor and k_i is an unknown value that we need to determine for each i-th sensor reading. According to the Equation above, if the magnitude of (cx_i, cy_i) computed by the OS-specific method is $|c_i|_{OS}$, and the Euclidean norm of (cx_i, cy_i) is $|c_i|_E$, Equation 8 can be reformulated as follows:

$$d_i = G_{OS}(k_i \cdot |c_i|_{OS}) \cdot k_i \cdot |c_i|_E \quad (10)$$

If we substitute $k_i \cdot |c_i|_{OS}$ with x, the solution to Equation 10 is basically the x-coordinate of the intersection point between the following two functions on the xy coordinate plane:

$$y = f_1(x) = \frac{K}{x} \quad \text{and} \quad y = f_2(x) = G_{OS}(x) \quad \text{where} \quad K = \frac{|c_i|_{OS}\, d_i}{|c_i|_E} \quad (11)$$

To solve this, we apply a computational method. First, we gradually increase x from 0 in small intervals and find the point where the sign of $(f_1(x) - f_2(x))$ changes. In this process, G_{OS} for x was linearly interpolated if it did not exist in the array of Equation 2.

If x before the sign change is x_1 and x after the sign change is x_2, we obtain the x-coordinate of the intersection point between the line segment connecting the points $[x_1, f_1(x_1)]$ and $[x_2, f_1(x_2)]$ and the line segment connecting the points $[x_1, f_2(x_1)]$ and $[x_2, f_2(x_2)]$ as the final solution to Equation 11 (see Figure 10).

If the solution of Equation 8 obtained for a particular K is x^*_K, we upload the following array to the firmware:

$$\mathcal{A}_x = [x^*_0,\ x^*_{\frac{K_{max}}{N_d}},\ x^*_{\frac{2K_{max}}{N_d}},\ \cdots,\ x^*_{K_{max}}] \quad (12)$$

Here, K_{max} is the maximum value of the desired K, which is set to 500, sufficiently large. The divisor N_d is also set to 1,000. However, if the overall magnitude of the OS-native gain is very low, the intersection may occur at very large x, and as a result, K_{max} may have to be increased further. The x^*_0, which represents the intersection of $y = 0$ and $y = G_{OS}(x)$, was assumed to be 0 because $G_{OS}(0)$ was also assumed to be 0 (see Equation 7).

Once we have uploaded the three arrays $\mathcal{A}_G$, $\mathcal{A}_{OS}$, $\mathcal{A}_x$, we are ready to cancel out the impact of that OS-native gain and move the pointer as we want, simply by controlling the HID report sent by the device firmware, without installing any additional client software. We assume that we have precomputed $[\mathcal{A}_{OS}, \mathcal{A}_x]$ array pairs for all native function settings of a particular OS. In this study, we have verified that the memory size of our custom mouse (512 KB) allows us to store in the firmware the $[\mathcal{A}_{OS}, \mathcal{A}_x]$ array pairs computed for all 22 native settings of Windows. The next section presents the final algorithm on how the firmware actually generates HID reports and manages remainders based on all the uploaded information.

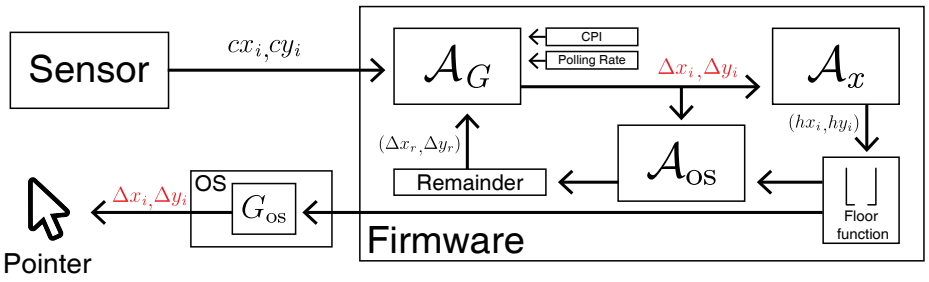

Figure 11: Block diagram of the proposed algorithm

3.4.1 HID Reporting Algorithm. This section describes in chronological order the process by which the firmware creates a HID report (hx, hy). First, the firmware calculates the desired pointer displacement d_i from the i-th sensor reading using Equation 3 to 5. Then, the firmware calculates the desired pointer movement vector $(\Delta x_i, \Delta y_i)$ parallel to the sensor reading vector (cx_i, cy_i) as follows:

$$(\Delta x_i, \Delta y_i) \leftarrow \frac{d_i}{|(cx_i, cy_i)|_{\mathrm{E}}} (cx_i, cy_i) \tag{13}$$

If the remainder of the pointer movement vector that could not be processed in the previous sensor reading is $(\Delta x_r, \Delta y_r)$, it is added to Δx_i and Δy_i to be processed in the current step:

$$(\Delta x_i, \Delta y_i) \leftarrow \frac{d_i}{|(cx_i, cy_i)|_{\mathrm{E}}} (cx_i, cy_i) + (\Delta x_r, \Delta y_r) \tag{14}$$

The firmware then updates the desired pointer displacement variable d_i as follows, to include remainder:

$$d_i \leftarrow |(\Delta x_i, \Delta y_i)|_{\mathrm{E}} \tag{15}$$

Next, the firmware computes K and retrieves the x_i^* value corresponding to its fractional index from $\mathcal{A}_x$ array (via interpolation):

$$K \leftarrow \frac{|(cx_i, cy_i)|_{\mathrm{OS}}\, d_i}{|(cx_i, cy_i)|_{\mathrm{E}}} \quad \text{and} \quad x_i^* \leftarrow \mathcal{A}_x[\frac{KN_d}{K_{\max}}] \tag{16}$$

Following the original definition of x, the variable k_i is computed as follows:

$$k_i \leftarrow \frac{x_i^*}{|(cx_i, cy_i)|_{\mathrm{OS}}} \tag{17}$$

Finally, according to Equation 9, the HID report that the firmware needs to send to the OS is calculated as follows:

$$(hx_i, hy_i) \leftarrow k_i (cx_i, cy_i) \tag{18}$$

However, since today's OSs only accept HID reports with integer components, the HID report vector computed above is passed through the floor function $\lfloor x \rfloor$ and being transmitted the OS:

$$(hx_i, hy_i) \leftarrow (\lfloor hx_i \rfloor, \lfloor hy_i \rfloor) \tag{19}$$

The loss of pointer displacement caused by the above flooring can be calculated using array $\mathcal{A}_{\mathrm{OS}}$ based on Equation 6 as follows and it is set as remainder for the next sensor reading:

$$(\Delta x_r, \Delta y_r) \leftarrow (\Delta x_i, \Delta y_i) - \mathcal{A}_{\mathrm{OS}}[\frac{N_{\mathrm{OS}}|(hx_i, hy_i)|_{\mathrm{OS}}}{h_{\max}}]\,(hx_i, hy_i) \tag{20}$$

The series of algorithms from Equations 13 to 20 are performed for each sensor reading (see Figure11 for a block diagram).

4 Technical Evaluation

In this section, we rigorously evaluate whether our proposed technique has comparable robustness and precision to conventional technique. In particular, we follow a purely quantitative approach rather than relying on subjective evaluations from users. We collect the physical speed of the device, HID reports, and the pointer position changes on the screen during random mouse movements to evaluate how accurately and precisely our mouse implements the desired gain function.

4.1 Method

4.1.1 Design. The experiment is conducted independently for both the baseline (or `Baseline`) and our technique (or `Ours`). Both experiments are performed with the same mouse, and in the `Baseline` condition, the mouse's sensor readings are directly sent to the OS as HID reports, just like a conventional mouse. In `Ours`, the algorithm proposed in this study is implemented in the mouse firmware.

The `Baseline` experiment followed a 6×2×2 full factorial design, and the independent variables and their respective levels are:

- Native Function: `C2`, `A2`, `C6`, `A6`, `C10`, `A10`
- CPI: `400` or `800`
- Polling Rate (in Hz): `125` or `250`

Native Function refers to the native gain function setting that the OS (Windows) has while the experiment is running, `C` refers to the constant function, `A` refers to the acceleration function, and the following number refers to the slider position in the control panel (see Figure 12).

The experiment for `Ours` follows a 4×3×6×2×2 full factorial design, and the independent variables and their respective levels are as follows:

- Shape: `Constant`, `Sigmoid`, `Sine`, or `Zigzag`
- Scale: `Low`, `Mid`, `High`
- Native Function: `C2`, `A2`, `C6`, `A6`, `C10`, `A10`
- CPI: `400` or `800`
- Polling Rate (in Hz): `125` or `250`

Shape refers to the shape of the gain function that we want to realize with our technique. Scale represents the amplitude of the desired gain function. `Low`, `Mid`, and `High` conditions were determined with reference to the range of `A2`, `A6`, and `A10`, respectively. All desired gain functions resulting from each Shape-Scale combination are plotted in Figure 13.

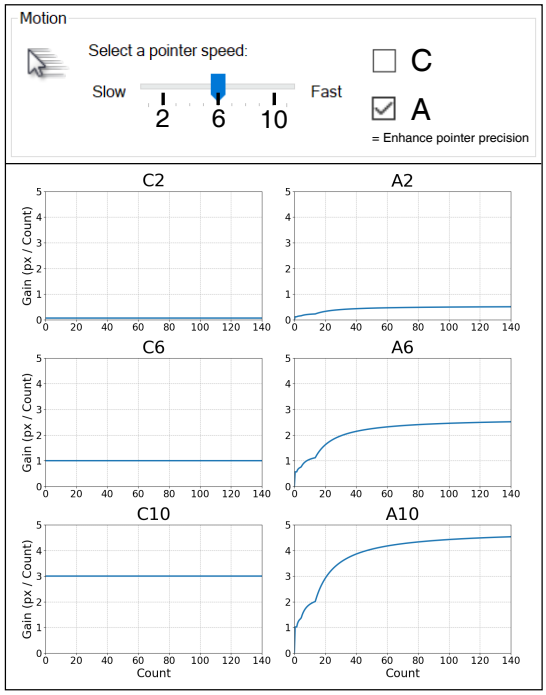

Figure 12: Six levels of `Native Function` used in `Baseline`

In addition to the above experiments, we also conducted additional experiments on `Ours` technique, where the device's CPI and polling rate were randomly changed once per second during mouse movement under one of the following four conditions:

[CPI, Polling Rate (Hz)]=[`400`, `125`], [`400`, `250`], [`800`, `125`], or [`800`, `250`]

In this additional experiment, the Shape and Scale of the desired gain function were fixed to `Sigmoid_Mid`, respectively, and the Native Function was fixed to the `A6` condition. This experiment allows us to evaluate whether the desired gain function can be robustly reproduced by our technique under unintended perturbations of hardware settings, such as CPI or polling rate.

4.1.2 Apparatus and Data Acquisition. The same custom mouse described in Section 3.2 was used in the experiments. Arrays $\mathcal{A}_{\text{OS}}$ and $\mathcal{A}_x$ for each Native Function were prepared in advance and uploaded to the custom mouse firmware. For significantly low Native Gains (`C2` and `A2`), $K_{\max}$ and N_d of $\mathcal{A}_x$ array was increased to 10,000 and 20,001, respectively. The experiments were performed on a single desktop PC (AMD Ryzen 5 7500F, 3,701 MHz, 32GB RAM, Windows 10), equipped with a 4K display (LG 27UP850N, 3840 × 2160, 69.7×39.2 cm) to observe pointer displacement over as wide a range as possible.

The mouse reads the sensor values at 125 Hz or 250 Hz (matched to the Polling Rate of each condition) and sends the information to the PC in two paths: via serial and via USB HID mouse report. Through the serial communication, we collected three pieces of mouse data for each sensor reading, (1) the physical speed of the mouse $\hat{v}$, (2) the HID report sent to the OS (rx_i, ry_i), and (3) the microprocessor timestamp. With the USB HID mouse report, OS translated the reported data to the cursor movement. A Python

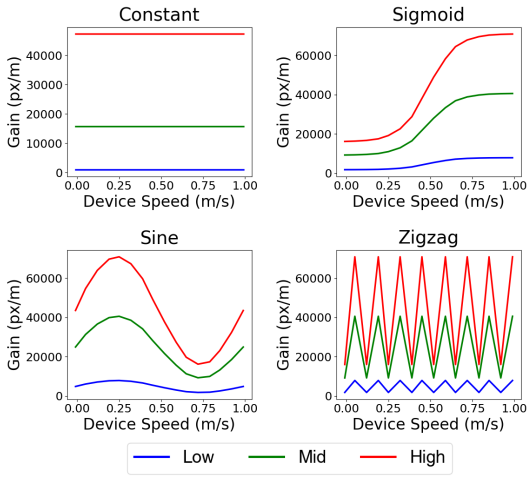

Figure 13: Twelve `Scale`×`Shape` combination gain functions used in `Ours`

script based on the `pynput.mouse` library detected the coordinates of the pointer on the screen on cursor move events and recorded the PC timestamp at the event. The baseline implementation took an average of 155 µs data processing time from sensor data acquisition to HID report generation, and our technique took an average of 280 µs, with an additional 130 µs of gain cancellation algorithm computation time.

4.1.3 Procedure. The first author of this paper conducted all the experiments. The experimenter randomly moved the mouse on the desk during data collection for each experimental condition of `Baseline` and `Ours`. To ensure that the mouse moved similarly across all conditions, a simple visualization was provided to the experimenter in real time. The visualization consisted of two progress bars. One bar showed whether the experimenter's movements sufficiently covered a wide range of mouse speeds. While most conditions had a desired speed range of 0 to 1 m/s, some conditions were measured only up to speeds lower than 1 m/s, as shown in Table 2. This is because, for gain functions with a large average scale, the pointer continues to hit the screen edge above a certain device speed. Furthermore, this can reduce experimenter fatigue by minimizing the time spent in an excessively fast speed range. The second bar showed whether the experimenter's movements sufficiently covered a wide range of mouse movement directions (0 to 360°). Data collection was automatically terminated only when a sufficient number of data points (N=300) were collected for each speed and direction bin, such that all progress bars reached 100%.

This experiment took 0.8 hours for `Baseline` and 9.6 hours for `Ours` to complete, during which the mouse actually moved for a total of 0.15 hours and 1.6 hours, respectively. The separate experiment for the `Ours` condition, where hardware settings were changed randomly, took 5 minutes to complete, during which the mouse actually moved for a total of 0.85 minutes.

4.1.4 Performance Evaluation. As a result of the experiment, we quantify how accurately the desired gain function is implemented as intended. We first multiply the measured mouse speed $\hat{v}$ in the

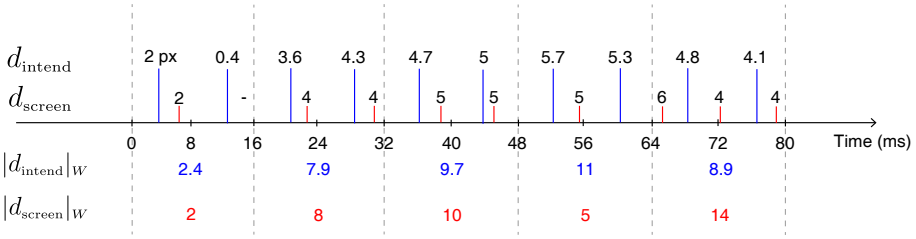

Figure 14: This figure shows how the values of d_{intend} and d_{screen} are accumulated within a time window of 16 ms.

firmware by the true desired gain $G(\hat{v})$ to obtain the intended pointer displacement, d_{intend}. For `Baseline` conditions, d_{intend} is obtained by multiplying the magnitude of the measured sensor reading count $|(cx, cy)|_{\text{E}}$ by the matching OS native gain $G_{\text{OS}}(|(cx, cy)|_{\text{OS}})$.

Then, we obtain the actual displacement of the pointer d_{screen} on the screen at the instant and compare it to d_{intend}. Next, we also tried a method that sacrificed some external validity: we computed the predicted pointer displacement d_{predict} on the screen by multiplying each raw HID report from the mouse by a known OS gain, rather than measuring it directly from the movement of the pointer on the screen: $d_{\text{predict}} = |(rx, ry)|_{\text{E}} \cdot G_{\text{OS}}(|(rx, ry)|_{\text{OS}})$. Ideally, the pairs d_{intend} and d_{screen}, and the pairs d_{intend} and d_{predict} should show high correlation. In `Baseline` condition, since the sensor reading (cx, cy) and the HID report (rx, ry) are always the same, d_{predict} and d_{intend} are also always the same.

4.2 Result

4.2.1 Estimating d_{screen}. There may be a delay between the pointer coordinate data and the data sent from the mouse firmware (such as mouse speed and HID reports). The delay for each condition was estimated from the lagged cross-correlation between the pointer coordinates and the HID report data after resampling. On average, a time delay of 2.3 ms (σ = 2.2) was observed (the firmware data was lagging) and the two data were synchronized for each condition. When divided into two groups, `Baseline` conditions and `Ours` conditions, the average time delay was 3.3 ms (σ=1.9) and 2.2 ms (σ = 2.2), respectively. The pointer coordinate data is then converted into pointer displacement (d_{screen}) data by subtracting adjacent rows. Meanwhile, when the pointer contacted the edge of the screen, the pointer displacement due to the HID report may not have been fully achieved, which could act as significant noise in our technical evaluation. Therefore, data measured while the pointer was contacting one of the screen edges and data measured adjacent to and before and after those contacts were considered outliers and were excluded from the analysis. In `Baseline`, 1,029 rows out of 32,874 rows were removed (3.13%), and in `Ours`, 16,640 rows out of 350,454 rows were removed (4.61%).

Table 2: In some conditions, a maximum device speed lower than 1 m/s was considered for measurements.

Condition	Gain Function	Maximum Device Speed (m/s)	
		400 CPI	800 CPI
Baseline	C6	1.00	0.82
Baseline	A6	0.66	0.36
Baseline	C10	0.55	0.28
Baseline	A10	0.41	0.24
Ours	Constant-High	0.55	0.55
Ours	Sigmoid-Mid	0.69	0.69
Ours	Sigmoid-High	0.53	0.53
Ours	Sine-High	0.91	0.91
Ours	Zigzag-Mid	0.72	0.72
Ours	Zigzag-High	0.45	0.45

4.2.2 Agreement Between $|d_{intend}|_W$ and $|d_{screen}|_W$. Since the pointer coordinate data and the data transmitted from the firmware are measured at different sampling rates, comparable (d_{intend}, d_{screen}) pairs may not always exist. Therefore, instead of doing an element-wise comparison of d_{intend} and d_{screen}, we focus on the fact that the d_{intend} data sampled at a high frequency is accumulated to determine the d_{screen} sampled at a low frequency. We divided the d_{intend} and d_{screen} data into W-ms long time windows evenly and then calculated $|d_{\text{intend}}|_W$ and $|d_{\text{screen}}|_W$, which are the sums of d_{intend} and d_{screen}, within each time window (see Figure 14). Considering the sampling rate at which pointer coordinates are collected (125 Hz or 250 Hz), W is set to 16 ms to ensure that at least two pointer coordinates can be included per time window to obtain the displacement.

Two metrics are introduced to evaluate the agreement between $|d_{\text{intend}}|_W$ and $|d_{\text{screen}}|_W$. The first is the correlation between the two values. Linear regression is performed on all ($|d_{\text{intend}}|_W$, $|d_{\text{screen}}|_W$) points for each condition, and the regression equation and coefficient of determination (R^2) are analyzed. The second is the difference between $|d_{\text{intend}}|_W$ and $|d_{\text{screen}}|_W$, i.e. the error in implementing the desired pointer displacement. The mean absolute error (MAE, unit: pixels) is computed for all ($|d_{\text{intend}}|_W$, $|d_{\text{screen}}|_W$) pairs.

As a result, the R^2 and MAE of `Baseline` and `Ours` did not show significant differences, as follows: `Baseline` R^2=0.9666 (σ=0.0400), MAE=5.0020 (σ=5.8621), `Ours` R^2=0.9692 (σ=0.0341), MAE=4.6514 (σ=4.4401). The mean slope and intercept of the regression equation were as follows for each technique: `Baseline` slope 0.9644 (σ=0.0361) and intercept 3.6276 (σ=4.6703), `Ours` slope 0.9436 (σ=0.0388) and intercept 4.7642 (σ=4.4972). Figure 15 show the main effects of each independent variable on R^2 and MAE. For `Baseline` and `Ours`, we randomly sampled the same number of ($|d_{\text{intend}}|_W$, $|d_{\text{screen}}|_W$) pairs, uniformly across all conditions, and plotted their scatter plots in Figure 16, together with the additional results when W is increased to 256 ms.

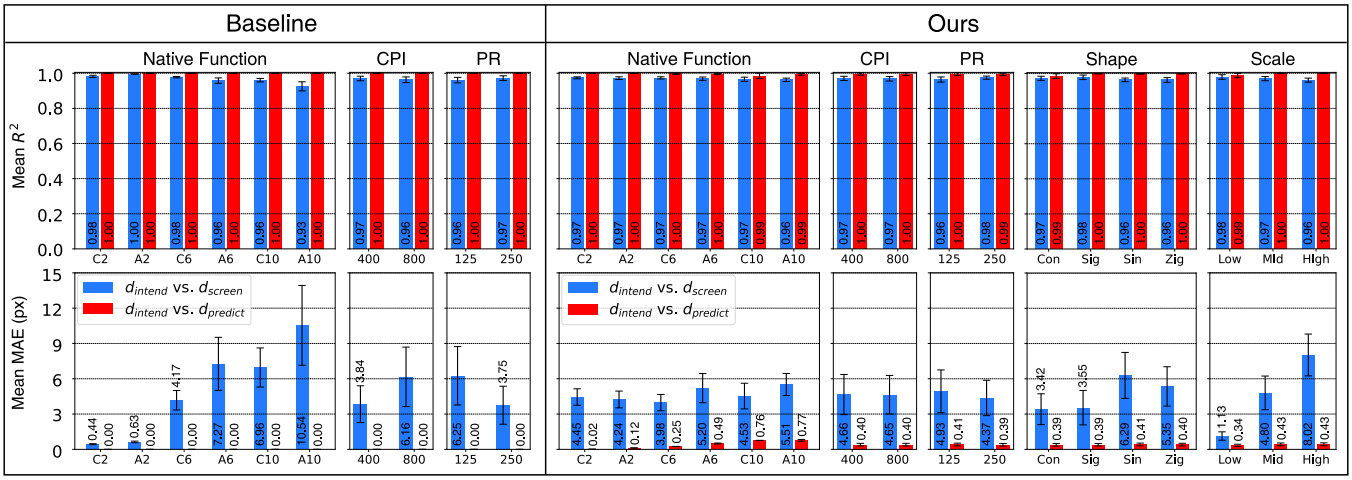

Figure 15: R^2 and MAE of `Baseline` and `Ours` conditions obtained as a result of technical evaluation

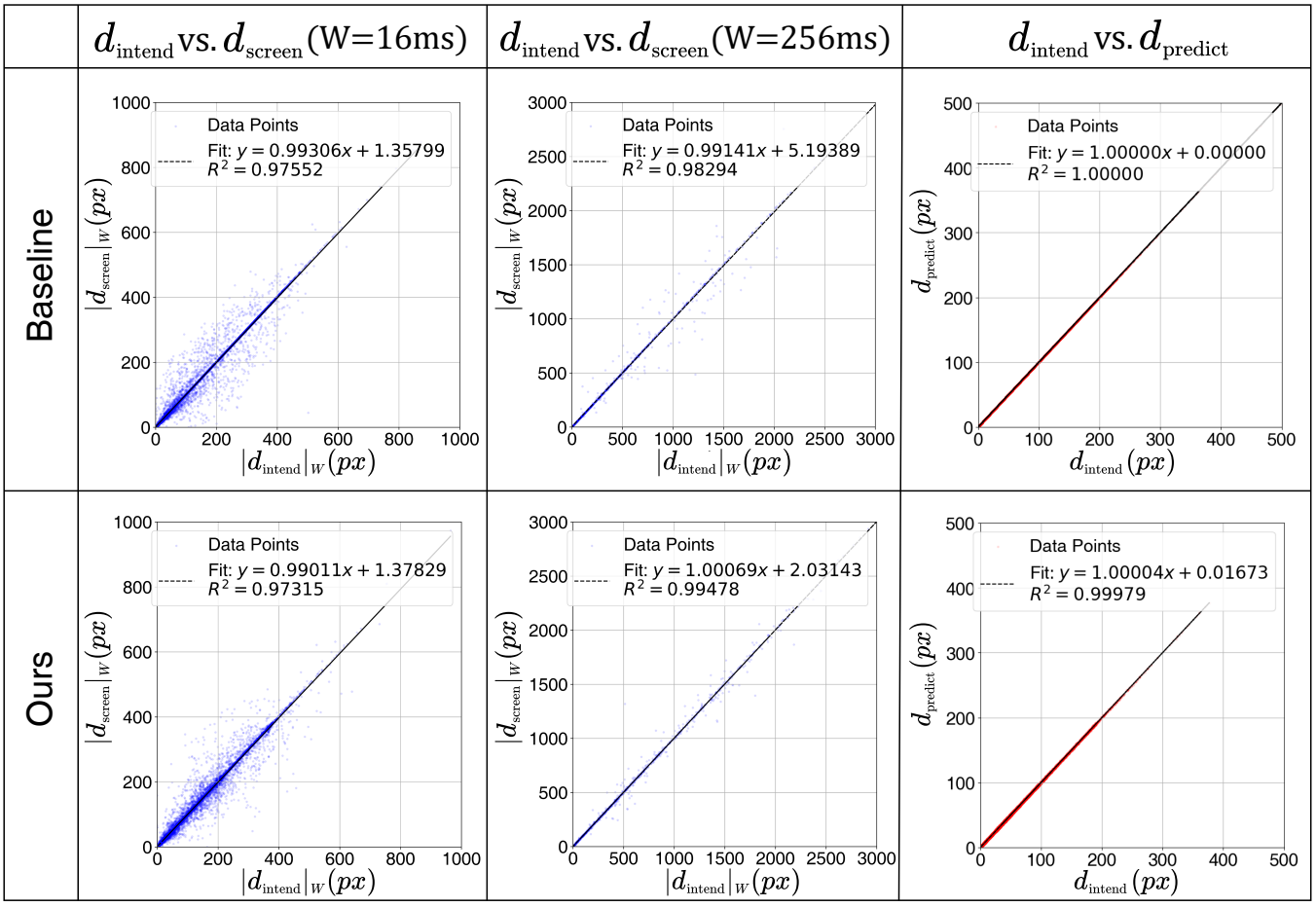

Figure 16: Scatter plots drawn by randomly sampling the same number of ($|d_{\text{intend}}|_W$, $|d_{\text{screen}}|_W$) or (d_{intend}, d_{predict}) pairs for `Baseline` and `Ours` conditions

Using the same method, we also analyzed data measured under the condition where the CPI and polling rate were randomly changed every second. As a result, the R^2 and MAE were still comparable to the `Baseline`, as follows: R^2 =0.9731 (σ=0.0190) and MAE=5.6236 (σ=3.2210). There was no noticeable anomaly in pointer movement at the moment when the hardware settings were automatically changed.

4.2.3 Agreement Between d_{intend} and $d_{predict}$. To examine the correlation between d_{intend} and d_{predict}, timestamp synchronization was not required, as both were calculated within the same serial port data packet. Therefore, we compared individual (d_{intend}, d_{predict}) pairs without a window-based accumulation. Additionally, since HID reports are unaffected by whether the pointer reaches the screen edge, no post-processing was performed for the calculation of d_{predict}. For each condition, a linear regression analysis was performed on the (d_{intend}, d_{predict}) data points, and MAE between d_{intend} and d_{predict} was also calculated. Note that these analyses were not conducted for `Baseline`, as d_{intend} and d_{predict} are always identical in that case, making the analyses irrelevant (i.e., $R^2 = 1.0$ and MAE=0).

As a result, R^2 and MAE in `Ours` conditions were obtained as follows: R^2=0.9951 (σ=0.0196), MAE=0.4060 (σ=0.3155). The average slope and intercept of the linear regression equations were R^2=1.0004 (σ=0.0046) and MAE=0.0192 (σ=0.0502), respectively. The results are also shown in Figures 15 and 16, alongside the d_{intend} vs. d_{screen} results. We also analyzed data measured under the condition where the CPI and polling rate were randomly changed every second. The R^2 and MAE of `Ours` condition were: R^2=0.9998 (σ=0.00005) and MAE=0.4891 (σ=0.0197).

5 Discussion and Limitations

In the technical evaluation, we found that the performance of `Ours` in implementing the desired gain function was indistinguishable from `Baseline` in terms of pointer displacement measured directly on the screen (d_{screen}). We speculate that the common noise observed in the scatter plots of both `Baseline` and `Ours` conditions (Figure 16) is because the measurement delay between d_{intend} and d_{screen} slightly varies in real time during a single measurement even after pre-synchronization. Such stochastic noise can cause the number of data points included in the 16 ms time window to exceed or fall short by one, which can lead to inaccurate calculations of $|d_{\text{intend}}|_W$ and $|d_{\text{screen}}|_W$, thereby lowering R^2 and increasing MAE. In particular, such noise can be amplified proportionally to the pointer speed, and we actually observed that an increase in the overall magnitude of the gain handled in both `Baseline` and `Ours` conditions leads to a decrease in R^2 and a increase in MAE (see `A10` condition and `High` condition in Figure 15). When the window size was increased to 256 ms, the noise in the scatterplot spread less in proportion to the displacement magnitude, which also supports that it originates from the measurement delay.

The results of the correlation analysis between d_{intend} and d_{predict}, which are relatively free from delay issues because the analysis was performed only with data transmitted from the device firmware, showed that `Ours` condition had additional sub-pixel level errors compared to `Baseline` condition. In `Ours` condition, the error between d_{intend} and d_{predict} showed an increasing trend as the overall scale of the gain function grew, while still maintaining sub-pixel level. One important point to note is that this sub-pixel error does not imply the existence of probabilistic jitter in the pointer's movement trajectory. This error simply means that the gain function realized is not exactly the same as the intended gain function due to the approximation error present in the tables uploaded to the device firmware ($\mathcal{A}_x$, $\mathcal{A}_{\text{OS}}$). We expect that the ability to implement any custom gain function with sub-pixel level pointer displacement errors will satisfy most high-performance users. Moreover, the errors in our technique can be further reduced by increasing the resolution of the tables being uploaded (i.e., increasing N_d or N_x), although this would require additional memory in the device firmware. However, we acknowledge that future research needs to more closely explore what level of sub-pixel error is sufficient to provide users with adequate subjective satisfaction.

The most important breakthrough in this study is the computation of a lookup table, $\mathcal{A}_x$ (see Equation 12), uploaded to the firmware that tells us which HID reports to send to generate the desired pointer displacement while canceling out the influence of the OS-native gain. $\mathcal{A}_x$ is constructed by finding the intersection of two functions, $y = \frac{K}{x}$ and $y = G_{\text{OS}}(x)$. One might wonder if there are cases where it is impossible to compute $\mathcal{A}_x$. The answer is yes, and there are two possible cases: when $G_{\text{OS}}(x)$ is 0 in some region, or when $G_{\text{OS}}(x)$ is defined to only accept inputs below a certain maximum. As far as we know, modern OS-native gain functions do not fall into either of these cases. Rather, we think that the most realistic and critical challenge in computing $\mathcal{A}_x$ comes when $G_{\text{OS}}(x)$ has a very large dynamic range. Then, the intersection point x^* of the two functions will also vary over a large range, and as a result, the size of the table to be uploaded to the firmware may become excessively large. If such a case occurs, instead of uploading the raw table as it is, we should try to approximate it by uploading only the coefficients of the polynomial fit function or by reducing the size of the table and relying on interpolation. Fortunately, we think that such problems rarely occur under the dynamic range of the common OS-native gain functions today.

Even if the native gain functions are not known in advance, it is not difficult to accurately measure them [2]. By repeatedly sending a specific HID report to the OS, the displacement of the pointer can be measured, and the corresponding OS gain can be determined. The HID reports can be generated using a separate microprocessor [2] or, for a specific OS (e.g., Linux), can be emulated by software. In our technique, it is possible to add a feature to the firmware of the custom mouse that allows it to switch into a so-called "gain measurement mode", which repeatedly sends HID reports to the OS. Our pilot implementation of this idea showed that for a single OS gain function setting, it took a total of 3.6 minutes to accurately measure the gain function up to a maximum magnitude range of 512 in the HID report (17 repetitions per magnitude, with steps of 7). Since the OS gain function only needs to be measured once per setting, we believe that this will not be a big burden for high-performance users.

In this study, we defined the transfer function as a function of physical mouse speed (m/s) and on-screen pointer speed (px/s), so the gain unit we used is px/m. Unlike us, most previous studies defined the gain in the transfer function as unit-less [2, 6], using both the device speed and the pointer speed in physical displacement

units. The concept of unit-less gain makes sense when considering unified physical contexts where the display and interaction space are tightly coupled to the user's physical environment, such as direct manipulation and immersive VR setups. In the context of gaming on an indirect display, however, the interaction elements on the display are primarily defined in pixel terms, controlled by pixel-precise pointers. In addition, the apparent angular size of the displayed element is subject to change by the distance between the eyes and the screen. Therefore, we believe that the absolute size of objects on screen is less significant. For example, Kim et al. [9] conducted a study to measure first-person shooter game scores while varying the display size, while maintaining equivalent FoV (field of view, unit:°), latency, brightness, and display resolution. Except for one condition where the display size is too small (13") and too close to the eyes, the game score remained at the same level (~1% difference) while the display size varied in the range from 26" to 65". This is equivalent to tripling the gain in physical size, but the effect was minimal. This result suggests that the unit of the transfer function in this context should be $^\circ/m$ as the function of the user's FoV and mouse displacement, rather than the unitless gain. For future reference, one pixel in our study corresponds to 0.01486° (3,840×2,160 resolution, 69.7×39.2 cm display placed at 70 cm from the user's eye). Our results in px/m can be easily converted to $^\circ/m$ by multiplying the factor of 0.01486.

On the other hand, readers may wonder how our technique can be applied to some commercial games that bypass the OS transfer function and force the use of their own in-game transfer function. In such games, input device control sometimes leads to changes other than pointer movement. For example, in FPS games, moving the input device usually changes the character's first-person view camera orientation, not the position of a pointer on the screen. In order to utilize our technique in such cases, the game system should be considered as a separate independent OS and the following two modifications should be made: (1) re-define the units of gain function as the units of in-game changes caused by input (e.g., °/m in FPS games)[1], and (2) precisely measure the transfer functions embedded in the game system. Since the in-game transfer functions of popular commercial games are generally already measured and known by players[8], we expect that these modifications can be made easily in most cases. Except for these modifications, all steps in Section 3 can be performed in the same way.

Although our technique significantly reduces the burden of transfer function customization and maintenance compared to existing solutions, there is still one thing that users must do manually to use our technique: informing the device firmware of the OS gain function setting (i.e., the position of the slider in the control panel). For example, this can be done by pressing buttons on the device to adjust the firmware settings, aligning them with the OS settings. Alternatively, to minimize manual effort, we can pre-define standard OS settings to use the device and require users to adhere to them (e.g., set the slider to the center position). If users are willing to install additional software, a simple client application could read the OS setting registry values when the device is connected and automatically send them to the device via the serial port.

Finally, we believe that the technique we proposed could be commercialized in the near future. Compared to existing mouse implementations on the market, our technique additionally requires about 177 KB of memory to store the entire lookup table pre-loaded and ≈130 μs of computation time. In our case, the MCU we used was sufficient to embed the entire lookup table on the chip and easily achieved the 1,000 Hz rate in our test. Alternatively, it is also possible to connect an additional memory chip that costs only a fraction of a dollar to support an existing microprocessor with limited on-chip memory. Or, it is possible to store only one lookup table at a time (using only 5 KB of SRAM) on demand when implementing the technique on existing mouse hardware. Meanwhile, note that we built the mouse hardware following Dual Sensor Mouse project [8, 10], and the modified 3D model and gain-cancelling source code are available online[9] This allows interested researchers and users to build the mouse themselves without waiting for mouse manufacturers to incorporate our transfer function technique.

6 Conclusion

The way pointing transfer functions are implemented in today's operating systems makes it difficult to satisfy users' demands for extreme high performance due to the following two problems: (1) the transfer functions cannot be customized in detail, and (2) unintended perturbations of hardware settings or OS-native transfer function settings cause significant disruptions in users' transfer function experience. The hardware-embedded pointing transfer function technique proposed in this study solves both problems by directly defining the transfer function in physical units in the device's firmware and embedding a special algorithm in the firmware that cancels out the influence of the OS-native function. In particular, our technique does not require separate software installation, unlike existing solutions, and supports all OS types. In a rigorous technical evaluation study using a custom-built mouse hardware, we evaluated and compared the conventional technique and our technique to see whether the intended pointer displacement based on the desired transfer function was actually accurately realized on the screen. As a result, we demonstrated that our technique has comparable accuracy and reliability to the conventional method in realizing custom transfer functions of various shapes and scales under various CPI and polling rate settings.

Acknowledgments

This study was funded by National Research Foundation of Korea (RS-2023-00223062, RS-2023-00211872), and Institute of Information and Communications Technology Planning and Evaluation (RS-2020-II201361). We thank anonymous reviewers for constructive feedback.

References

[1] Ben Boudaoud, Josef Spjut, and Joohwan Kim. 2023. Mouse sensitivity in first-person targeting tasks. *IEEE Transactions on Games* (2023).

[2] Géry Casiez and Nicolas Roussel. 2011. No more bricolage! Methods and tools to characterize, replicate and compare pointing transfer functions. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology* (Santa Barbara, California, USA) *(UIST '11)*. Association for Computing Machinery, New York, NY, USA, 603–614. https://doi.org/10.1145/2047196.2047276

[8] Such as Counter-Strike: Global Offensive (CS:GO), Rainbow Six Siege, and Valorant

[9] https://github.com/SunjunKim/esp32_pmw3389_dual.git

[3] Seungwon Do, Minsuk Chang, and Byungjoo Lee. 2021. A simulation model of intermittently controlled point-and-click behaviour. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–17.

[4] Ian Donovan, Marcia A Saul, Kevin DeSimone, Jennifer B Listman, Wayne E Mackey, and David J Heeger. 2022. Assessment of human expertise and movement kinematics in first-person shooter games. *Frontiers in Human Neuroscience* 16 (2022), 979293.

[5] Jessica Formosa, Nicholas O'donnell, Ella M Horton, Selen Türkay, Regan L Mandryk, Michael Hawks, and Daniel Johnson. 2022. Definitions of esports: a systematic review and thematic analysis. *Proceedings of the ACM on Human-Computer Interaction* 6, CHI PLAY (2022), 1–45.

[6] Ravin Balakrishnan Géry Casiez, Daniel Vogel and Andy Cockburn. 2008. The Impact of Control-Display Gain on User Performance in Pointing Tasks. *Human–Computer Interaction* 23, 3 (2008), 215–250. https://doi.org/10.1080/07370020802278163 arXiv:https://www.tandfonline.com/doi/pdf/10.1080/07370020802278163

[7] Raiza Hanada, Damien Masson, Géry Casiez, Mathieu Nancel, and Sylvain Malacria. 2021. Relevance and Applicability of Hardware-independent Pointing Transfer Functions. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 524–537.

[8] Donghyeon Kang, Namsub Kim, Daekaun Kang, June-Seop Yoon, Sunjun Kim, and Byungjoo Lee. 2024. Quantifying Wrist-Aiming Habits with A Dual-Sensor Mouse: Implications for Player Performance and Workload. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–18.

[9] Joohwan Kim, Arjun Madhusudan, Benjamin Watson, Ben Boudaoud, Roland Tarrazo, and Josef Spjut. 2022. Display size and targeting performance: Small hurts, large may help. In *SIGGRAPH Asia 2022 Conference Papers*. 1–8.

[10] Sunjun Kim, Byungjoo Lee, Thomas Van Gemert, and Antti Oulasvirta. 2020. Optimal sensor position for a computer mouse. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

[11] Erica Kleinman, Reza Habibi, Garrett B Powell, Brent Reeves, James Prather, and Magy Seif El-Nasr. 2024. "Backseat Gaming" A Study of Co-Regulated Learning within a Collegiate Male Esports Community. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–14.

[12] Byungjoo Lee, Mathieu Nancel, Sunjun Kim, and Antti Oulasvirta. 2020. AutoGain: gain function adaptation with submovement efficiency optimization. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–12.

[13] Hanbyeol Lee, Seyeon Lee, Rohan Nallapati, Youngjung Uh, and Byungjoo Lee. 2024. Characterizing and Quantifying Expert Input Behavior in League of Legends. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–21.

[14] Arjun Madhusudan and Benjamin Watson. 2021. Better frame rates or better visuals? An early report of Esports player practice in Dota 2. In *Extended Abstracts of the 2021 Annual Symposium on Computer-Human Interaction in Play*. 174–178.

[15] Mathieu Nancel, Emmanuel Pietriga, Olivier Chapuis, and Michel Beaudouin-Lafon. 2015. Mid-air pointing on ultra-walls. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 5 (2015), 1–62.

[16] N.R.K#7525. 2020. Mouse Accel 101. https://docs.google.com/document/d/1wuQln99lQVBU9L8_QbpifrarpJ1xjPuKsKD2FY026Hc/edit?usp=sharing [Accessed Sep 12, 2024].

[17] Eunji Park, Sangyoon Lee, Auejin Ham, Minyeop Choi, Sunjun Kim, and Byungjoo Lee. 2021. Secrets of Gosu: Understanding physical combat skills of professional players in first-person shooters. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–14.

[18] Ruth Rosenholtz, Yuanzhen Li, and Lisa Nakano. 2007. Measuring visual clutter. *Journal of vision* 7, 2 (2007), 17–17.

[19] Shaishav Siddhpuria, Sylvain Malacria, Mathieu Nancel, and Edward Lank. 2018. Pointing at a distance with everyday smart devices. In *Proceedings of the 2018 CHI conference on human factors in computing systems*. 1–11.

[20] Josef Spjut, Ben Boudaoud, Kamran Binaee, Jonghyun Kim, Alexander Majercik, Morgan McGuire, David Luebke, and Joohwan Kim. 2019. Latency of 30 ms benefits first person targeting tasks more than refresh rate above 60 Hz. In *SIGGRAPH Asia 2019 Technical Briefs*. 110–113.

[21] Benjamin Watson, Josef Spjut, Joohwan Kim, Byungjoo Lee, Mijin Yoo, Peter Shirley, and Rulon Raymond. 2024. Is Less More? Rendering for Esports. *IEEE Computer Graphics and Applications* 44, 2 (2024), 110–116.

[22] Benjamin Watson, Josef Spjut, Joohwan Kim, Jennifer Listman, Sunjun Kim, Raphael Wimmer, David Putrino, and Byungjoo Lee. 2021. Esports and high performance HCI. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–5.