# Measuring the Latency of Graphics Frameworks on X11-Based Systems

Andreas Schmid
University of Regensburg
Regensburg, Germany
andreas.schmid@ur.de

Raphael Wimmer
University of Regensburg
Regensburg, Germany
raphael.wimmer@ur.de

## ABSTRACT

Latency is an intrinsic property of all human-computer systems. As it can affect user experience and performance, it should be kept as low as possible for real-time applications. To identify the source of latency, measuring partial latencies is necessary. We present a new method for measuring the latency of graphics frameworks on X11-based systems. Our tool measures the time between an input event arriving at the kernel until a pixel is updated in graphics memory. In a systematic evaluation with 36 test applications, we found that our method delivers consistent results for most tested frameworks, and does not add a significant amount of additional end-to-end latency. Even though further investigation is required to explain inconsistencies with Qt-based frameworks, our method measures the latency of graphics frameworks reliably and accurately in all other cases.

## CCS CONCEPTS

**• Computing methodologies → Graphics systems and interfaces**; **• Human-centered computing → User interface toolkits**; **• Software and its engineering** → *Empirical software validation.*

## KEYWORDS

latency measurement, graphics frameworks

**ACM Reference Format:**
Andreas Schmid and Raphael Wimmer. 2023. Measuring the Latency of Graphics Frameworks on X11-Based Systems. In *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23), April 23–28, 2023, Hamburg, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3544549.3585779

## 1 INTRODUCTION

Latency, the delay between a user's action and the system's response, is an intrinsic property of all interactions between human and computer. As it affects task difficulty and overall user experience, it is important to minimize it as much as possible, particularly in real-time applications, such as video games [5, 12, 18]. Furthermore, latency can confound the outcome of psychological experiments, as well as replication studies [13, 16].

All software and hardware components contribute to a system's end-to-end latency, including input devices, operating system, applications, network connection, and output devices. There are many different ways to measure end-to-end latency, such as using high-speed cameras. However, to identify bottlenecks and reduce latency, it is necessary to measure the partial latencies of individual components.

Measuring the latency of software components, such as graphics frameworks, may seem straightforward at first glance – one could just log timestamps before and after an operation. However, access to an application's source code is required for this approach. Additionally, this method does not work if the used graphics framework handles rendering asynchronously.

In this paper, we present a method for measuring the latency of graphics frameworks on X11-based Linux distributions. Latency is determined by measuring the time between an input event arriving at the operating system's kernel and a pixel being updated in graphics memory. We use XShm to retrieve the pixel's color because of its direct access to graphics memory. This approach is independent of the application under test, as long as it reacts to an input event by updating the displayed content. As no modifications of the application's source code are required, our method can also be used with proprietary software, such as video games. We measured the latency of 18 graphics and UI frameworks and validated our measurements by comparing measured framework latencies to simultaneously measured end-to-end latency.

Our measuring method can help select an appropriate graphics framework for the development of real-time applications, as well as measure and validate the latency of finished applications before deployment. Furthermore, even though the measured absolute latencies we report in this paper are only valid for our test computer, relative differences can be generalized to other systems.

## 2 RELATED WORK

The most common approach to measuring latency is to use a high speed camera [9, 10, 14, 15, 20–22]. Most current mobile phones can record slow-motion video with up to 240 frames per second. However, this method has several disadvantages. First of all, this method's accuracy is limited by the camera's frame rate. Accuracy is further reduced as it is oftentimes hard to determine at which exact video frame a button press or a change in display color happens. Additionally, for camera-based latency measurement, manual annotation of video material is necessary [8]. However, as latency is rarely constant, measurement series are necessary to determine its distribution [2], which makes this method very time consuming.

When sub-millisecond accuracy and long measurement series are required, microcontrollers or SoC computers can be used to trigger or sense input events at a system and capture its output, for example using photo sensors attached to a display. This method can for example be used to measure the end-to-end latency of devices with a capacitive touch screen. Deber et al. [6] use a brass contact placed on the touch screen and connect it to electrical ground with a relay to trigger a touch event. An application on the device under test reacts to this event by changing the color of an area on the screen from black to white. This change in brightness is detected with a photo diode. Time between triggering the touch event and the screen changing its color is measured with a microcontroller. Kämäräinen et al. [11] use a similar approach to measure the latency of mobile devices. They added several functions, such as simulating external input devices, and measuring network latency. Casiez et al. [4] use a vibration sensor attached to a user's finger to sense touch or button press events. Similar to Deber et al. and Kämäräinen et al., they use an application changing the screen's color, and a photo diode to determine the end of the latency measurement. In addition to end-to-end latency, they also measured partial latencies by logging timestamps at several points in the tested device's software, such as when the input event is first detected by the operating system, or when the test application finishes rendering.

Measuring such partial latencies is crucial to finding and improving performance bottlenecks of a system. Wimmer et al. [23] use a model of input device latency to show how slight differences in polling rates can lead to vastly different distributions of overall latency. With their *LagBox*, Wimmer et al. [1, 23] measured the latency of different mice, keyboards, and gamepads. They used a Raspberry Pi to electrically trigger the button of an input device with an optocoupler, and then measured the time it takes for an input event to arrive at the operating system. However, as this method requires electric contacts attached to a button, the device under test has to be disassembled before measurement.

Casiez et al. [3] propose a non-invasive approach for measuring the end-to-end latency in graphical user interfaces. An optical mouse is placed on the computer's screen, which displays a special texture. When the texture is moved, a movement event is triggered by the mouse. The time between moving the image on the screen and the mouse's movement event arriving at the application is considered as the computer's end-to-end latency. Additionally, partial latencies can be measured by logging timestamps on the system under test. This way, Casiez et al. could find differences in latency caused by factors such as operating system, system load, and different graphics frameworks.

In an extensive blog article, Pavel Fatin [7] illustrates which factors contribute to latency when typing text. They also measured and systematically compared the latency of different text editors on different operating systems. For their latency measurements, Fatin implemented a Java application that simulates a virtual keyboard, triggers a key event, and measures the time until a pixel at a predefined position changes color due to a character appearing in the text editor. They found that several factors, including the choice of editor, operating system, and window manager, influence text input latency.

The latency of computer displays, also known as display response time, is standardized by the International Display Measurement Standard[1]. Response time is defined as the time it takes from image data being sent to the monitor until the display's center point reaches 50% brightness. A low-cost implementation of this measuring method was published by Stadler et al. [19]. Additionally, *Leo Bodnar Electronics*[2] distributes dedicated devices for measuring display response time.

As latency is an important aspect throughout the field of Human-Computer Interaction, numerous methods for measuring and reducing latency have emerged. There are several accurate methods for measuring a system's end-to-end latency, as well as partial latencies contributed by input devices and displays. However, to the best of our knowledge, there are no scientific publications proposing a precise method for measuring the latency of graphics frameworks. Such a method could be an important step towards gaining a thorough understanding of end-to-end latency and its building blocks.

## 3 IMPLEMENTATION AND VALIDATION

In this section, we describe the implementation and validation of a method to measure the latency of graphics frameworks and other applications that react to an input event by updating screen content. We validated our method by simultaneously measuring framework latency and end-to-end latency of different test applications on a powerful computer with low latency input and output devices. By subtracting the resulting framework latency from the corresponding end-to-end latency, the remaining system latency should be consistent as nothing except the test application was changed between measurements (Eq. 1).

$$\begin{aligned} L_{EtE} &= L_{input} + L_{framework} + L_{display} + L_{unknown} \\ \Rightarrow L_{EtE} - L_{framework} &= L_{input} + L_{display} + L_{unknown} \end{aligned} \tag{1}$$

Additionally, we measured our measuring program's influence on the system's end-to-end latency by running each measurement series twice, once with and once without the measuring program running.

### 3.1 Latency Measuring Program

Our tool for measuring the latency of graphics frameworks is written in C++ and uses the evdev[3] library to capture input events, and the XShm extension[4] to access graphics memory. The evdev library provides an interface for accessing input devices such as keyboard, mouse, touch pad, and joystick. The library interfaces directly with the device driver, bypassing the input subsystem of the Linux kernel, and is therefore one of the fastest ways to capture input events on Linux. The XShm extension provides shared memory transport between an X client and the X server and allows for fast data transfer by eliminating the need for data to be serialized and sent over a socket. Because of those requirements, our tool only works on X11-based systems.

Our latency measuring tool is initialized with the path to an input device in `/dev/input/` which is then monitored for input events using evdev. Once an input event occurs, the latency measurement is started by recording a timestamp in microseconds from `chrono`'s

[1]https://www.sid.org/Standards/ICDM
[2]http://www.leobodnar.com/shop/index.php?products_id=212
[3]https://linux.die.net/man/4/evdev
[4]https://linux.die.net/man/3/xshm

`steady_clock`. The program then enters a loop continuously querying a specified pixel from graphics memory using `XShmGetImage`. Once the pixel's color changes, a second timestamp is recorded and the measurement is stopped. The resulting difference between both timestamps is sent to standard out so the measuring program can be easily included into application pipelines.

## 3.2 Test Applications

Even though our tool can be used to measure the latency of any application that reacts to an input event and then updates the displayed content, we implemented standardized test applications with 18 different graphics and UI frameworks in different programming languages. To compare the lowest achievable latencies for each framework, we implemented simple test applications. With each framework, we wrote a program that displays a $1920 \times 1080$ pixel full screen window with a black background. Once a mouse button is clicked, the entire window's color is changed to white until the button is released. Those clicks are handled by the individual frameworks' input libraries as the time it takes to capture input events is part of a framework's latency.

Additionally, we implemented similar applications that display a more complex scene. For those applications, instead of changing the entire window's color to white, 1000 randomly sized rectangles with random color are rendered. A white rectangle is rendered at the top left corner as the additional end-to-end latency measurement described in the next section needs maximum contrast to function properly.

We implemented test applications for the following graphics and UI frameworks: FLTK (C++), GTK3 (C, with Cairo backend), Java 2D, Java Swing, GLEW (C++, using SDL2 for input handling), GLUT (C++), pygame, pyglet (Python), PyQt5, PyQt6, Qt5 (C++), SDL2 (C++, with software, OpenGL, and OpenGLES2 backend), tkinter (Python), wxPython, Xlib (C), and XCB (C).

## 3.3 End-to-End Latency Measurement

As described earlier in Eq. 1, the accuracy of framework latency measurements can be evaluated by comparing them with simultaneous end-to-end latency measurements, given all system components except the measured framework remain constant. For our end-to-end latency measurements, we used a modified version of Schmid and Wimmer's *Yet Another Latency Measuring Device* (YALMD) [17]. The measuring process works as follows: First, an *Arduino Micro* microcontroller starts a timer and closes an optocoupler connected to the button of a computer mouse. Then, a test application on the PC under test reacts to this induced mouse click by changing its background color from black to white. The microcontroller senses this change in brightness with a photo sensor attached to the system's display. Once a brightness threshold is reached, the timer is stopped and the difference between timestamps – the measured end-to-end latency – is sent to the PC via USB. Before each measurement, the microcontroller waits for a randomized time span between 500 and 1500 milliseconds to prevent synchronization with USB polling or display refresh rate.

Our modified version of YALMD is controlled with serial signals sent via USB. A calibration process to determine the brightness threshold for stopping the measurement can be started by sending

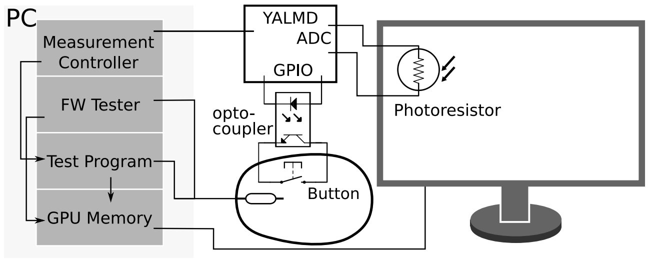

**Figure 1: Measurement apparatus and procedure. The measurement controller starts the test application, the framework latency tester, and the measurement process. YALMD triggers the mouse button by closing an optocoupler and starts the end-to-end latency measurement timer. The framework latency tester receives the click event an starts the framework latency timer. The test application reacts to the click event by changing the screen's color. Once pixels have changed in GPU memory, the framework latency tester stops its measurement. Once YALMD detects a change in brightness on the display, it stops the end-to-end latency measurement.**

the character `'c'` and a single measurement can be conducted by sending `'m'`. By triggering individual measurements from the system under test, we can assign each measured framework latency to its corresponding end-to-end latency.

## 3.4 Procedure

We conducted all measurements on a desktop computer[5] running Debian Buster 5.10. An *Asus XG248Q* monitor was used at a resolution of $1920 \times 1080$ pixels and a refresh rate of 240 Hz. YALMD was connected to a Logitech G15 gaming mouse with an input device latency of 2.17 ms (SD: 0.3 ms) [23] to trigger mouse clicks. Its photo sensor was attached to the monitor's top left corner to minimize delay added by display refresh.

A bash script starts the test application, triggers YALMD's calibration process, and starts the measurement controller – a Python program – to start a measurement series with 500 individual measurements. This Python program communicates with YALMD and starts the framework latency test program as a subprocess. Furthermore, it reads framework latency from the test program and end-to-end latency from YALMD for each measurement and stores them in a *pandas* data frame which is saved in CSV format after a measurement series is completed. For each test application, we conducted one measurement series with and without the framework latency measurement program running. This way, we measured whether the framework latency measurement program influences the system's end-to-end latency. The whole measuring process is schematically depicted in Fig. 1.

We repeated the whole measuring process for all test applications on different desktop environments: pure Xorg without a window manager, KDE Plasma with enabled or disabled KWin compositor, and on Xfce4 with and without compositing. Vertical synchronization was turned on only in conditions with an active compositor.

[5]Intel i7-8700 @ 3.2 GHz, Nvidia GTX 1080 (Driver: Nvidia 470.103.01), 16 GB DDR4 RAM

**Table 1: Difference between median of measured end-to-end latency for when the framework tester was running and when it was not. Negative values mean a lower end-to-end latency with an active framework latency tester. All values are in milliseconds. For all frameworks excluded from this table, the absolute error was below 1.5 milliseconds in all conditions.**

| **Desktop** | **Xfce4** | | **Plasma** | | **Xorg** |
|---|---|---|---|---|---|
| **Compositor** | **on** | **off** | **on** | **off** | **off** |
| GLUT | -3.57 | -0.79 | -0.89 | -0.89 | -5.05 |
| PyQt5 | -3.01 | -2.02 | -2.02 | -1.85 | -2.24 |
| PyQt6 | -3.07 | -2.62 | -1.8 | -1.91 | -2.02 |
| Qt5 | -3.14 | -1.79 | -2.02 | -1.84 | -2.13 |
| tkinter | -1.69 | -1.11 | -1.12 | -1.01 | 0.23 |
| pyglet | -2.24 | -1.9 | -0.67 | -1.17 | -0.73 |

## 4 VALIDATION AND MEASUREMENTS

In the following, we present the results of our framework latency measurements and our validation by comparing those measurements with the system's end-to-end latency.

### 4.1 Influence on End-to-End Latency

First, we report the measuring program's influence on the system's end-to-end latency. For accurate measurements of framework latency, it is critical that the measuring program does not introduce additional latency. By conducting each measurement series twice, once with the latency measuring program running and once without, we can find the measuring program's influence on the system's end-to-end latency by calculating the difference between both measurement series.

Results show that the absolute difference is below 1.5 ms in most cases. However, for some frameworks the difference is significantly higher with up to five milliseconds in one case (Table 1). Surprisingly, in many cases, measured end-to-end latency was lower when the framework latency tester was running. One possible explanation for this behavior could be that XShm forces rendering of the content when reading the graphics memory.

### 4.2 Measurement Validation

To validate our method for measuring the latency of graphics frameworks, we compare framework latency measurements to corresponding end-to-end latency measurements. If our method measures framework latency accurately, the difference between measured end-to-end latency and measured framework latency should be the same regardless of the test application. We calculated the latency difference for each individual measurement. An example can be seen in Fig. 2. Latency difference distributions for all permutations of desktop environment, compositor, and type of test application can be seen in Fig. 3.

In some conditions with Qt and Java-based frameworks, our validation revealed unexpected behavior. For Qt-based frameworks, the measured framework latency was oftentimes higher than the system's end-to-end latency. To this point, we have no reliable

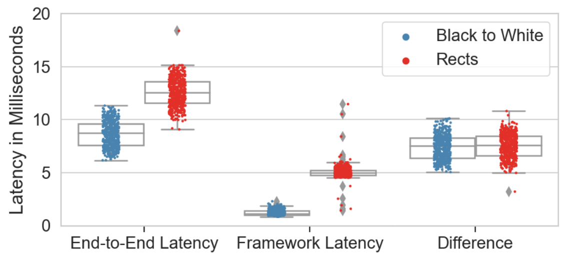

**Figure 2: Example of our validation process for the GLUT framework on KDE Plasma with compositing turned off. Blue dots represent measurements in the simple condition with the screen only changing color from black to white. Red dots represent measurement in the complex condition displaying 1000 rectangles on click. The latency difference is calculated by subtracting framework latency from end-to-end latency for each individual measurement. In this case, the latency difference is almost equal for both conditions.**

explanation for this behavior. One possible reason would be that Qt blocks the graphics memory longer than necessary, so XShm can not access it to read pixels. However, this issue needs further investigation in future work.

Of the 180 permutations of framework, desktop environment, compositor, and type of test application, in 26 cases, at least one negative latency difference could be found among the 500 measurements (see Fig. 3). This was exclusively the case for measurements of Java (5) and Qt-based (21) frameworks. Therefore, we exclude those permutations for following observations regarding our entire data set and assume our framework latency measurements are incorrect in those cases. The aggregated mean latency difference over all remaining permutations is 7.9 ms with a standard deviation of 3.7 ms and a 95% confidence interval of 7.3 ms – 8.5 ms.

### 4.3 Framework Latency Measurements

Fig. 4 depicts measured framework latencies for the 15 remaining graphics frameworks. Detailed measurement results for each framework with each desktop environment can be found in the paper's appendix (Table 2 and 3). For some frameworks, an active compositor adds a small amount of latency. This is probably due to double buffering which delays rendering by one frame ($\approx$ 4 ms with a 240 Hz display). While active compositors can affect latency, their influence does not increase at higher overall latency. Also, the influence of the used desktop environment on framework latency is low.

Many frameworks were very performant in the simple condition, contributing only around one millisecond to the system's end-to-end latency – especially with a disabled compositor. However, most Python-based frameworks performed clearly worse in this condition.

Unsurprisingly, the more complex condition rendering 1000 rectangles leads to higher latency with all measured frameworks. Furthermore, system-near and OpenGL-based frameworks – while not always the fastest with the simple test application – handle

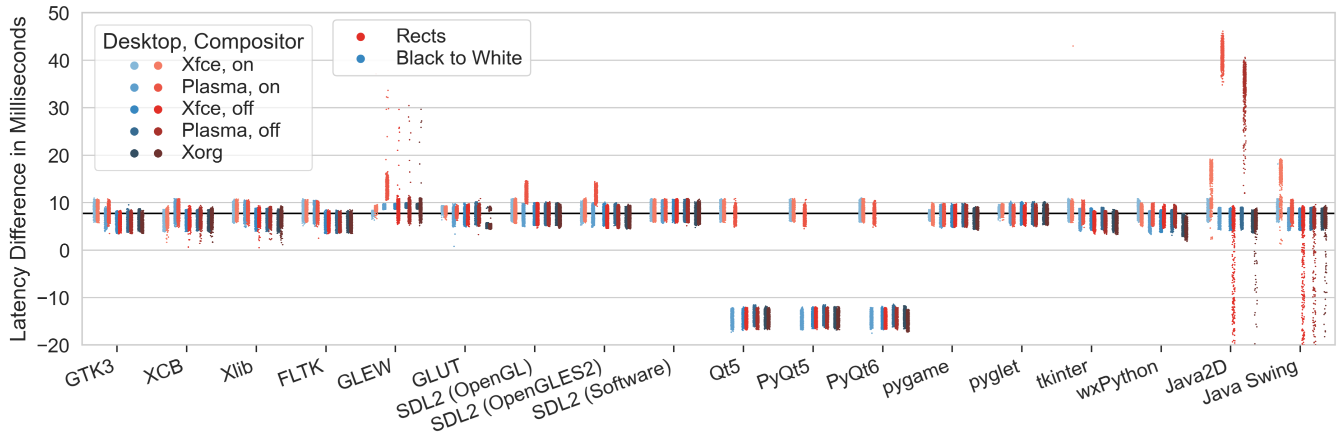

**Figure 3: Latency difference for all frameworks. The black horizontal line shows the aggregated mean of 7.9 milliseconds. Ideally, all measurements would align. Even though the error is in an acceptable range in most cases, there are still some outliers, especially for both Java frameworks. The negative values for Qt-based frameworks are caused by the measured framework latency being higher than the end-to-end latency.**

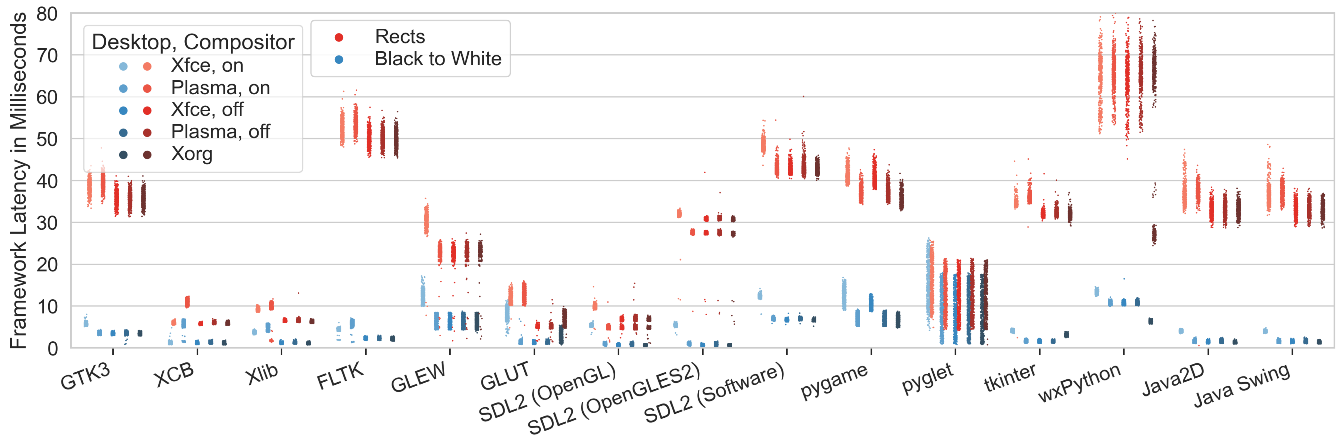

**Figure 4: All measured framework latencies at a glance. Note that extreme outliers for both Java frameworks are cut off by the limited Y axis and Qt-based frameworks were excluded due to the measurement's unrealistic results compared to end-to-end measurements.**

the complex condition better than other frameworks. For example, Xlib, XCB, and GLUT, as well as SDL2 with an OpenGL backend, could achieve latencies as low as five milliseconds with a disabled compositor. This is an order of magnitude lower than the latencies of the worst performing frameworks, such as FLTK or wxPython.

Those findings imply that when choosing a graphics framework for a real-time application, it is important to consider the complexity of the rendered scene.

## 5 DISCUSSION, LIMITATIONS, AND FUTURE WORK

In this paper, we presented a new method for measuring the latency of graphics frameworks, including a thorough validation process. Even though our method for measuring the latency of graphics frameworks did not work with the rather popular Qt-framework, it delivered consistent results in all remaining cases. The absolute effect of our measuring program on the system's end-to-end latency was below 1.5 milliseconds with most tested frameworks. We therefore regard the current state of our method as usable and accurate in most cases, but recommend an additional end-to-end latency measurement as a sanity check.

For a fully reliable and accurate measuring process, further research, development, and validation is required. Even though the XShm library we used is designed for quick and direct access to graphics memory, it is not available on all systems and, as our validation has shown, it seems to be incompatible with some graphics frameworks. Therefore, future research should compare different methods for accessing graphics memory in terms of speed and compatibility to find the best candidate for framework latency measurements.

Despite the numerous permutations of test applications and desktop environments taken into consideration, our implementation and validation of a framework latency measuring tool is still very limited. All of our measurements were run on the same computer. However, choice of graphics card, driver, and operating system could influence results significantly.

Despite those limitations, our work is an important step towards measuring – and therefore understanding – the different partial latencies that make up a system's end-to-end latency. Once we can accurately measure all major factors contributing to end-to-end latency, it becomes possible to create theoretical models and simulate the latency of different systems.

Furthermore, even at the current state, our method can be used to validate experimental setups for user studies without needing to build or purchase additional hardware. The source code of our measuring program, as well as data of our measurements, can be found on GitHub[6]. Additional information can be found on our project website https://hci.ur.de/projects/framework-latency.

## REFERENCES

[1] Florian Bockes, Raphael Wimmer, and Andreas Schmid. 2018. LagBox – Measuring the Latency of USB-Connected Input Devices. In *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*. ACM Press, Montreal QC, Canada, 1–6. https://doi.org/10.1145/3170427.3188632

[2] François Bérard and Renaud Blanch. 2013. Two Touch System Latency Estimators: High Accuracy and Low Overhead. In *Proceedings of the 2013 ACM International Conference on Interactive Tabletops and Surfaces (ITS '13)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/2512349.2512796

[3] Géry Casiez, Stéphane Conversy, Matthieu Falce, Stéphane Huot, and Nicolas Roussel. 2015. Looking Through the Eye of the Mouse: A Simple Method for Measuring End-to-end Latency Using an Optical Mouse. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. ACM, New York, NY, USA, 629–636. https://doi.org/10.1145/2807442.2807454 event-place: Charlotte, NC, USA.

[4] Géry Casiez, Thomas Pietrzak, Damien Marchal, Sébastien Poulmane, Matthieu Falce, and Nicolas Roussel. 2017. Characterizing Latency in Touch and Button-Equipped Interactive Systems. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 29–39. https://doi.org/10.1145/3126594.3126606

[5] Mark Claypool and Kajal Claypool. 2006. Latency and player actions in online games. *Commun. ACM* 49, 11 (Nov. 2006), 40–45. https://doi.org/10.1145/1167838.1167860

[6] Jonathan Deber, Bruno Araujo, Ricardo Jota, Clifton Forlines, Darren Leigh, Steven Sanders, and Daniel Wigdor. 2016. Hammer Time!: A Low-Cost, High Precision, High Accuracy Tool to Measure the Latency of Touchscreen Devices. ACM Press, 2857–2868. https://doi.org/10.1145/2858036.2858394

[7] Pavel Fatin. 2015. Typing with pleasure. https://pavelfatin.com/typing-with-pleasure/

[8] Sebastian Friston and Anthony Steed. 2014. Measuring Latency in Virtual Environments. *IEEE Transactions on Visualization and Computer Graphics* 20, 4 (April 2014), 616–625. https://doi.org/10.1109/TVCG.2014.30 00063.

[9] Ding He, Fuhu Liu, Dave Pape, Greg Dawe, and Dan Sandin. 2000. Video-Based Measurement of System Latency. *International Immersive Projection Technology Workshop* 111 (July 2000), 6. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.1123&rep=rep1&type=pdf

[10] Topi Kaaresoja and Stephen Brewster. 2010. Feedback is... late: measuring multimodal delays in mobile device touchscreen interaction. ACM Press, 1. https://doi.org/10.1145/1891903.1891907

[11] Teemu Kämäräinen, Matti Siekkinen, Antti Ylä-Jääski, Wenxiao Zhang, and Pan Hui. 2017. Dissecting the End-to-end Latency of Interactive Mobile Video Applications. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*. ACM, Sonoma CA USA, 61–66. https://doi.org/10.1145/3032970.3032985

[12] Ulrich Lampe, Qiong Wu, Hans Ronny, André Miede, and Ralf Steinmetz. 2013. To Frag or to Be Fragged - An Empirical Assessment of Latency in Cloud Gaming. SciTePress - Science and and Technology Publications, 5–12. https://doi.org/10.5220/0004345900050012

[13] Xiangrui Li, Zhen Liang, Mario Kleiner, and Zhong-Lin Lu. 2010. RTbox: A device for highly accurate response time measurements. *Behavior Research Methods* 42, 1 (Feb. 2010), 212–225. https://doi.org/10.3758/BRM.42.1.212

[14] Jiandong Liang, Chris Shaw, and Mark Green. 1991. On temporal-spatial realism in the virtual reality environment. In *Proceedings of the 4th annual ACM symposium on User interface software and technology (UIST '91)*. Association for Computing Machinery, New York, NY, USA, 19–25. https://doi.org/10.1145/120782.120784

[15] Andriy Pavlovych and Wolfgang Stuerzlinger. 2009. The tradeoff between spatial jitter and latency in pointing tasks. ACM Press, 187. https://doi.org/10.1145/1570433.1570469

[16] Richard R. Plant, Nick Hammond, and Tom Whitehouse. 2003. How choice of mouse may affect response timing in psychological studies. *Behavior Research Methods, Instruments, & Computers* 35, 2 (May 2003), 276–284. https://doi.org/10.3758/BF03202553

[17] Andreas Schmid and Raphael Wimmer. 2021. *Yet Another Latency Measuring Device.* preprint. Open Science Framework. https://doi.org/10.31219/osf.io/tkghj

[18] 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 on - SA '19 (SA '19)*. ACM Press, New York, NY, USA, 110–113. https://doi.org/10.1145/3355088.3365170 00003.

[19] Patrick Stadler, Andreas Schmid, and Raphael Wimmer. 2020. DispLagBox: simple and replicable high-precision measurements of display latency. In *Proceedings of the Conference on Mensch und Computer (MuC '20)*. Association for Computing Machinery, New York, NY, USA, 105–108. https://doi.org/10.1145/3404983.3410015

[20] Anthony Steed. 2008. A simple method for estimating the latency of interactive, real-time graphics simulations. In *Proceedings of the 2008 ACM symposium on Virtual reality software and technology (VRST '08)*. Association for Computing Machinery, New York, NY, USA, 123–129. https://doi.org/10.1145/1450579.1450606

[21] Colin Swindells, John C. Dill, and Kellogg S. Booth. 2000. System lag tests for augmented and virtual environments. In *Proceedings of the 13th annual ACM symposium on User interface software and technology (UIST '00)*. Association for Computing Machinery, New York, NY, USA, 161–170. https://doi.org/10.1145/354401.354444 00052.

[22] Robert J. Teather, Andriy Pavlovych, Wolfgang Stuerzlinger, and I. Scott MacKenzie. 2009. Effects of tracking technology, latency, and spatial jitter on object movement. IEEE, 43–50. https://doi.org/10.1109/3DUI.2009.4811204

[23] Raphael Wimmer, Andreas Schmid, and Florian Bockes. 2019. On the Latency of USB-Connected Input Devices. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems - CHI '19 (CHI '19)*. ACM Press, Glasgow, Scotland Uk, 1–12. https://doi.org/10.1145/3290605.3300650

[6] https://github.com/PDA-UR/FrameworkLatencyTester/

## A MEASURED FRAMEWORK LATENCIES IN TABULAR FORM

**Table 2: Results of framework latency measurements for the simple condition in tabular form. In the simple condition, upon a mouse click, a white rectangle was drawn on top of a black full screen window. Depicted results are medians of a measurement series with 500 individual measurements. All values are in milliseconds.**

| **Desktop** | **Xfce4** | | **Plasma** | | **Xorg** |
|---|---|---|---|---|---|
| **Compositor** | **on** | **off** | **on** | **off** | **off** |
| GTK3 | 5.68 | 3.39 | 3.47 | 3.45 | 3.39 |
| Java2D | 4.01 | 1.32 | 1.48 | 1.43 | 1.18 |
| Java Swing | 3.97 | 1.35 | 1.38 | 1.43 | 1.14 |
| GLEW | 12.21 | 6.44 | 6.34 | 6.42 | 6.42 |
| GLUT | 8.12 | 1.12 | 1.19 | 1.15 | 3.11 |
| pygame | 12.63 | 10.43 | 6.89 | 6.79 | 6.49 |
| SDL2 (OpenGL) | 5.28 | 0.56 | 0.85 | 0.82 | 0.51 |
| SDL2 (OpenGLES2) | 5.32 | 0.54 | 0.89 | 0.84 | 0.51 |
| SDL2 (Software) | 12.2 | 6.58 | 6.8 | 6.79 | 6.42 |
| tkinter | 3.99 | 1.43 | 1.45 | 1.42 | 3.07 |
| wxPython | 13.51 | 10.66 | 10.75 | 10.74 | 6.21 |
| FLTK | 4.52 | 2.12 | 5.77 | 2.09 | 2.02 |
| pyglet | 15.73 | 8.38 | 9.44 | 9.11 | 9.21 |
| XCB | 1.07 | 0.94 | 5.47 | 1.08 | 0.89 |
| Xlib | 3.71 | 1.12 | 4.98 | 1.12 | 0.88 |

**Table 3: Results of framework latency measurements for the complex condition in tabular form. In the complex condition, upon a mouse click, 1000 randomly colored rectangles were drawn on top of a black full screen window. Depicted results are medians of a measurement series with 500 individual measurements. All values are in milliseconds.**

| **Desktop** | **Xfce4** | | **Plasma** | | **Xorg** |
|---|---|---|---|---|---|
| **Compositor** | **on** | **off** | **on** | **off** | **off** |
| GTK3 | 38.18 | 35.86 | 39.33 | 35.66 | 35.58 |
| Java2D | 36.54 | 33.2 | 37.11 | 33.01 | 33.07 |
| Java Swing | 36.34 | 33.36 | 37.07 | 33.11 | 33.01 |
| GLEW | 30.39 | 22.87 | 22.83 | 22.78 | 22.93 |
| GLUT | 12.43 | 4.92 | 12.91 | 5.0 | 6.76 |
| pygame | 40.96 | 40.55 | 36.81 | 37.03 | 35.52 |
| SDL2 (OpenGL) | 9.73 | 5.79 | 4.75 | 6.31 | 5.24 |
| SDL2 (OpenGLES2) | 32.0 | 27.99 | 27.46 | 28.35 | 27.8 |
| SDL2 (Software) | 47.76 | 42.43 | 42.25 | 42.34 | 42.09 |
| tkinter | 34.36 | 31.8 | 36.0 | 31.95 | 31.66 |
| wxPython | 65.49 | 64.61 | 65.56 | 65.64 | 64.33 |
| FLTK | 52.35 | 50.04 | 53.6 | 49.94 | 49.76 |
| pyglet | 16.21 | 13.11 | 13.13 | 12.41 | 12.76 |
| XCB | 5.86 | 5.65 | 10.77 | 5.89 | 5.75 |
| Xlib | 9.21 | 6.39 | 9.74 | 6.37 | 6.19 |