Evaluation of Communication Latencies of Extracting Heterogeneous Context Information Instances from Virtual Reality Headsets

Mathias Ooms
Master student
University of Antwerp
Email: mathias.ooms@student.uantwerpen.be

***Abstract*—Over the last years, Virtual Reality (VR) systems are gaining in popularity and applications for both individuals and industry practices. In order to achieve full VR immersiveness, a better understanding and proper measurement tools are needed for reduced latency, reliability, network structures and throughput. Current off-the-shelf virtual reality systems are able to generate different types of context information. This context information consists of the user's location, head orientation (rotation), eye tracking, etc. Unfortunately, extracting this information is typically bounded to manufacturer software, incomplete, head-mounted display (HMD) specific or worse: not available at all due to the lack of PC drivers (e.g. PSVR). In this paper, we propose a general, future-proof design that exports multiple types of context information that are sent over a network interface. This design works for basically all HMDs, with the help of community-supported tools that support driver implementation. The performance is then shown, by measuring the latencies from the moment of physical motion to the detection in a subscriber. This system can than act as a gateway for building measurement tools, analytics and visualization of context information. The result is an easy to use system that works cross-platform and is functional. Finally, we demonstrate the benchmarked communication latencies of roughly 90 ms, suggesting that improvements will be needed for truly enabling full-immersive VR applications.**

***Index Terms*—Latency, context-extraction, network distribution, VR, context-aware-VR, OpenVR, OpenHMD**

I. Introduction

Over the last years, Virtual Reality (VR) systems are gaining at popularity and applications for both individuals and industry practices. In order for VR to really be useful and broaden the range of industry related applications, it demands for better immersiveness and practical use such as wireless VR. This is achieved by a better understanding of current VR and proper measurement tools.

Current off-the-shelf virtual reality systems (e.g. HTC VIVE Pro) are able to generate different types of context information. Context information is defined as the collection of: positional coordination, rotational coordination and eye-tracking. This results in a mobility pattern that is necessary to determine the eventual video feed for a VR user. The more accurate this mobility pattern is, the more accurate the video feed, the better the experience of the user. However, only an accurate mobility pattern is not enough, all context information and video feed needs to be send with very low latency in order to not nauseate the user and create an immersive experience.

If these context information instances can be made readily available at the network level, they could be leveraged for improving the performance of the VR-supporting wireless network (e.g. proactive handovers, load balancing, traffic prioritization). One could reduce latencies, increase throughput (increased resolution and frame-rate video feed) and increase reliability. All this would lead to an enhanced immersiveness and optimizations of supporting wireless infrastructures such as 5G [4], [25].

To realise this, there is a need for proper benchmarking the aspects of context information, in terms of latencies, throughput, reliability and computational burden [36], [37]. Through this benchmarking, one can have better understanding where current limitations are, where performance can be gained and make valid comparisons of different systems. Unfortunately, this is not the case for current VR systems. In addition, there is a lack of understanding of the level of importance of various types of context information, hence the prioritization of different context information instances at the moment cannot be done.

In this paper, we demonstrate, using multiple HMDs, how context information can be extracted easily over a network interface. This results in easy access and distribution of this context information to possible third party analytical and graphical tools (subscribers) based on this information. We do this by developing a method for extracting these different types of context information from COTS VR systems, like the HTC VIVE and PlayStation VR HMD, which gives us a comparison in different hardware. We implement this system and foresee two different networking protocols, ZMQ and MQTT, for later comparison in performance. This performance is then shown, by measuring the latencies from the moment of physical motion to the detection in a subscriber, thus including the context-extraction as well as the delivery to subscribers. This results in a cross-platform system that is ready to use and for everyone available.

We start by going over related works in terms of context-extraction and latency benchmarking. Followed by the

background research III, where we consider possible HMDs to provide the context information and the possible SDKs to extract that information. Next, based on the background research, the design and implementation of the system is shown, explained and justified IV. After this, the system is benchmarked for its latency to extract the information and the observations are elaborated V. Towards the end we give potential directions for future work and conclude the paper VI.

II. Related Work

Measuring latency in VR is already been done before, but typically it is based on rendering. In [2] the measured the delay between the physical movement and the rendering on screen, by attaching light-sensors to the physical object and the screen. The measurement is than obtained by moving the tracked object repetitively by hand. A similar more recent approach was taken in [22], measuring the different delays of the Oculus rift in comparison with V-SYNC enabled and to different smartphones. In the works of [10], they used a robotic arm and a optoelectronic motion tracking system, in order to determine the precision of the tracking of an Oculus Rift S HMD. The main difference with this work in terms of benchmarking, is that the latency isn't the round-trip delay of the VR rendering, but the latency between the physical movement and the detection of this movement for a third party subscriber.

Recently, wireless HMD benchmarking has gained way more interest with the arrival of 5G [4], [25], as immersive VR over cellular networks can serve as an opening for more practical applications. They state the trade-off of the three most important aspects: latency, throughput and reliability, continuing with the challenges and possible solutions. This work builds a distribution platform for VR context information, in order to address the underlying problems for these challenges.

III. Background Research

We provide an overview of the important features of different HMDs and outline the possibilities and limits of context-extraction in different SDKs. These will serve as the basis for the criteria for choosing appropriate HMDs in the final discussion III-C. Table I provides the full overview of the properties of the VR systems. In the first column, we see the (widely) varying prices, the second column shows whether base stations are required to determine position and/or rotation. The third column shows whether an external device, such as a PC or PlayStation, is needed for content generation, otherwise we refer to it as a standalone (SA) HMD (thus wireless). Virtual Reality of course needs to offer rotational tracking (and preferably positional tracking) as this is key to realise VR, but this does not mean we can easily access this information. The next three columns state if we can access this type of information through one of the mentioned SDK's in the last column. At last, we also added the availability of eye tracking for each HMD.

Table II provides a full overview of the SDK properties. Concerning the context information they can extract, on which operating systems are compatible and the platform that they use. For each SDK, we go over the possibilities and properties of the different SDK's that were mentioned during HMD research. This will be an important choice as these SDK's determine the limits for our context-extraction. As we want as much freedom as possible, we consider the following criteria: support for multiple cross-vendor HMDs, access to all information at once and it needs to be future-proof.

A. Hardware

1) Phone-driven VR: A possible approach is to use your phone as HMD, as most modern phones have the necessary hardware to provide simple but functional VR experience. This enables the tracking of rotation, but how about location? Then we are obligated to use an external device for location tracking, that also doesn't suffer from "drift" [13]. For eye-tracking, the front-facing camera comes to mind, but those are ill-positioned. Not to mention, the phone needs to fit in a case, which would already block the view for the front-facing camera to track the eyes.

Of course there are many other problems in terms of using different phones. Things as performance and precision are very dependent from phone to phone. These concerns, combined with the expected implementational burden, mean that phone-driven VR is not a viable approach for this work.

2) HTC VIVE Pro Eye: The HTC Vive Pro Eye [29] is the latest and greatest of HTC HMDs, providing accurate position-, rotation- and eye-tracking. Parts of the information we need could be extracted using: Vive SDK (Vive SRanipal SDK), Tobii SDK or OpenVR (Valve). While this HMD provides all the necessary information we want, it was not an option due to cost concerns.

3) HTC VIVE original: The HTC Vive original [30] is also a HMD that provides very accurate positional and rotational tracking. Eye-tracking can be added using add-ons presented in Section III-A6. Possible SDK's are: Vive SDK (Vive SRanipal SDK), Tobii SDK or OpenVR (Valve). The only drawback is that no built-in eye-tracking is provided.

4) PlayStation VR: The PSVR HMD [21] is officially only meant for use with PlayStation. However, as it is a popular and affordable HMD that has been on the market for some time, there exists software to provide functionality with computers for both Windows and Linux. The only option to extract the context information is through OpenVR (Valve). Because we already are in the disposal of a PSVR, makes it an excellent additional to try out the final system. The main drawbacks of this HMD however are that it (for now) only can provide rotational information, as positional information

HMD	Price	Base stations	Device	Orientation	Positional	Eye tracking	SDK
Phone driven	€0	No	SA	?	?	No	?
PlayStation VR [21]	±€300	No	PS	Yes	Not yet	No	OpenVR
Oculus Go [15] †	±€378	No	SA	Yes	Yes	No	No
Oculus Quest [16] †	±€350	No	PC & SA	Yes	Yes	No	OpenVR
Oculus Quest 2 [17]	€350	No	PC & SA	Yes	Yes	No	OpenVR
Oculus Rift S [18]	€450	No	PC	Yes	Yes	No	OpenVR
HTC Vive (original) [30] †	±€250 + €300*	Yes*	PC	Yes	Yes	Pupil Labs	OpenVR
HTC Vive Cosmos [31]	€800	No	PC	Yes	Yes	Pupil Labs	OpenVR
HTC Vive Focus Plus [33]	€870	Yes	SA	Yes	Yes	Pupil Labs	OpenVR & Tobii sdk
Pico Neo 2 Eye [20]	€945	No	PC	Yes	Yes	Yes	Tobii sdk
HTC Vive Cosmos Elite [32]	€1030	Yes	PC	Yes	Yes	Pupil Labs	OpenVR
HTC Vive Pro Eye [29]	€1400	Yes	PC	Yes	Yes	Yes	OpenVR & Tobii sdk

TABLE I: Overview of popular VR systems with their considered properties. (†: end-of-life)

is required by an external proprietary PlayStation Camera and there is no support for eye-tracking.

5) Oculus Quest 2: The Oculus HMDs [17] (now owned by Facebook) differ somewhat in the way they are operated. Previous HMDs are only operational wired to a computer. The Oculus Quest (1 & 2) are able to operate as standalone or wired to a computer. When the HMD is wired up to the computer, it is possible to access the SteamVR library, which will later on be crucial for our system. Being it one of the cheapest models, it would be an excellent option. The only drawbacks are: no support for eye-tracking and the lack of Linux support at the time of writing.

6) Eye-tracking as Add-on: The majority of current HMDs does not yet provide eye-tracking as this is considered to be a premium feature and is very expensive. However, there does exist some add-on hardware for this. Pupil Labs [11] released an add-on device that provides the eye-tracking information through it's own API. This device fits in the HTC Vive, Epson and HoloLens, but is unfortunately very expensive (€1400). A cheaper competitor called the Droolon F1 [8] also exists, retailing at around €150. However, support and documentation for the device appear to be scarce.

B. SDK

1) Tobii SDK: The Tobii XR SDK [27] enables cross-platform development of eye tracking by providing simple methods to access eye tracking data as well as tools and libraries to speed up development. They support a handful of HMDs, but through three main methods: the Unity game engine, Unreal Engine and natively.

The Unity [7] method is the most extensive option in terms of details in information. It is currently compatible with four HMDs: the HP Reverb G2 Omnicept Edition, the Pico Neo 2 Eye, the HTC VIVE Pro Eye and the Tobii HTC VIVE Devkit. This allows for the design of 3D environments and interaction through eye information like: gazing, blinking, etc. and the possibility of storing all this data. Additionally to this method, they provide an advanced API called Tobii Ocumen, to provide easy access to this data. The next method is through Unreal Engine [23]. The principle is the same as before, but this method is still in Beta and for example does not yet provide support for the HTC VIVE Pro Eye. The last method is called the Tobii XR Native SDK and is the most fitting solution when using the SDK only for raw data extraction. Unfortunately, it is still in Alpha and only supports: the Tobii HTC VIVE Devkit and the Tobii AR4 integration. The Qualcomm Snapdragon VR 845 and Tobii Pro Glasses are only available upon request and still a "work in progress". The HTC VIVE Pro Eye is not supported at all.

Unfortunately, the main drawback of this SDK is that this data only relates to eye-tracking. Although the eye information is very extensive, it is not what we are looking for, as we still need positional and rotational context information. This information would upon its turn, need to be provided by yet another SDK, which would be inconvenient. Also, the limitation in the number of HMDs is not future-proof and the features would make us too dependent of this SDK.

2) Vive SDK: The provided SDK by Vive [35] themselves is split up in three parts: Vive Wave, Vive Sense and VivePort. The Vive Wave offers an open interface, enabling interoperability between numerous all-in-one VR HMDs and accessories, supporting mainstream game engines. This allows players with different VR devices easy access to your con-

SDK	Orientation	Positional	Eye tracking	OS	Platform
Tobii SDK [27]	No	No	Yes (extensive)	Linux & Windows	Unity/Unreal Engine/Native †
Vive SDK [35]	Yes	Yes	HMD dependent	Windows	Unity/Unreal Engine/Native †
Oculus SDK [14]	Yes	Yes	No	Windows & Mac	Unity/Unreal Engine/Native †
OpenVR [28]	HMD dependent	Yes	HMD dependent	Linux & Windows	Native

TABLE II: Overview SDK properties. (†: limitations apply)

tent. Vive Sense provides a framework for providing more detailed data like: eye-, facial- and hand-tracking. Viveport is a platform to publish your VR designs (app-store). The SDK provides easy integration with Viveport to utilize the store features.

Not all SDK's are fully compatible with all HTC HMDs, for a full overview see [34]. The most interesting SDK for our use-case is the Vive Sense SDK, as we want to access this type of information. Unfortunately, much like Tobii SDK, this works as a plugin with Unity and is designed for Windows only. Additionally, this would limit us to HTC-only HMDs, which strikes against our criteria.

3) OpenVR: The OpenVR API by Valve [28], is open-source software that provides a way to interact with Virtual Reality displays without relying on a specific hardware vendor's SDK. It works by communication through SteamVR (Valve), which is a property of uttermost importance. The reason is that HMDs still serve primarily to provide immersive gameplay, with Steam (Valve) as the number one PC-game distribution-platform, HMD manufacturers need to provide (almost) always compatibility with this platform. This will make our context-extraction tool cover numerous current- and future-HMDs. The open-source aspect makes it also cross-platform, as we can compile it ourselves and makes it future-proof as it evolves together with SteamVR.

We mentioned that this solution would make it cross-platform, but therefore we need the VR-drivers, which are typically only Windows-optimized. Well, fortunately Valve designed SteamOS a Debian-based Linux distribution, originally intended for their Steam Machines. This is why they provide excellent support for Linux distributions like Ubuntu and designed Linux compatible drivers. However, since Windows is still the most used OS for gaming, Linux VR-drivers tend to come deployed much later, but OpenVR also provides a way to implement your own drivers, making it the most unlimited option we have and ticks off all our criteria.

C. Discussion

Concerning the selection of the HMDs, many of the discussed options are not viable, due to cost concerns. For example the HTC Vive Pro Eye provides all context information, but at a very high cost, just as the eye-tracking add-ons. We opt to use the HTV Vive and PSVR headsets, without any eye-tracking add-ons. This choice will make the comparison interesting as PSVR was not intended to use in combination with a computer and does not have optimized drivers for it.

In terms of SDKs, The OpenVR SDK is by far the most robust option to choose. It is fully open-source, future-proof, cross-platform and allows implementation of extra headsets. From all SDKs, OpenVR provides the most freedom, as the other SDKs are typically manufacturer-dependent and limited support. The combination of the chosen headsets and SDK will lead to interesting comparison, between hardware and software.

IV. Context-Extraction

After the extensive research, we can now implement the total design and justify our choices to realise an easy-to-use, cross-platform, future-proof distribution system that is compatible with a wide range of HMDs.

A. Goal

The final goal of our system is easy access to the extracted context information (for multiple HMDs now and in the future) at the best possible performance. To realise this, we split the design into two parts: acquiring the context information and publishing this over a network interface. The performance is then shown by the measurement of the latency.

For the first part, An SDK is needed that works with multiple HMDs, is plug-and-play and needs to be easy to setup. For the second part, we opt to use publish-subscribe mechanisms. Other possibilities were REST or single-writer mechanisms, but typically these offer more features, which typically translates to complexity and overhead. We want to reduce this in order to minimize the communication latency. Publish-subscribe mechanisms are very simple and typically scale better as they just publish the information on a network port. This way one can use a desired computer-language to access the information and will make it possible to implement programs (subscribers) to utilize with the context information.

B. Design

The theoretical design is shown in figure 1. As the complete system is designed for use with external analytical tools, it is important to know which latency they can expect and where it comes from. An important additional aspect of this design is the modularity, as the the messaging protocol can easily be switched an thus also the communication with subscribers.

The complete implemented design is shown in figure 2. We opt to use OpenVR as this gives us no theoretical limits: we

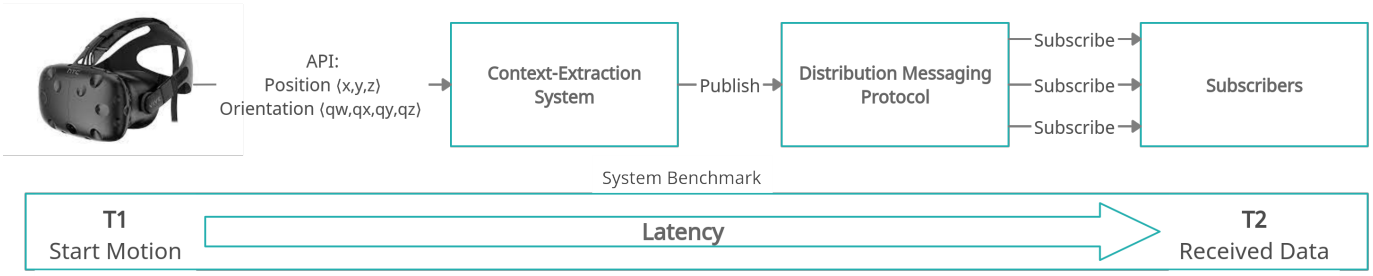

Fig. 1: The design.

can use SteamVR drivers, implement our own, works cross-platform and access all context information provided by the HMDs. Using the OpenVR C++ library, we implement our own C++ program that extract positional and rotational coordinates and publishes them on a port, either using the ZMQ-library or the Paho MQTT library. This way a subscriber, can easily access the information and use it to store data, build real-time applications or analytical tools.

With this design, the complete framework is highly modular an easy to use. SteamVR provides tutorials to ease the setup of multiple HMDs. Our C++ program comes with easy to run scripts and only needs to be imported by its `CMakeLists.txt` and select the preferred publish-subscribe mechanism (own add your own). From this point forward, the context information is published and only needs to be processed by a subscriber. Our subscriber serves as example of a real-time application showing the rotational orientation, but can easily be replaced by more advanced applications.

C. Implementation

The implementation of this design 2 is theoretically perfectly possible, but in practice was hard to realise. The way we started to implement this, is by looking at the included OpenHMD samples, where they can customize the VR-display and access all tracking information. Two excellent introductions that displayed the positional & rotational coordinates, were already implemented and showed (on Windows) by [1], [12]. These are excellent starting points to realise the first necessary steps to acquire positional & rotational coordinates.

The final result is an C++ application, that integrates all necessary third-party libraries nicely together in one directory, where installation is as easy as running a few scripts, importing the rewritten `CMakeLists.txt` and choosing your protocol (ZMQ or MQTT), working on both Linux and Windows.

D. Setup

In order to use our program, we first need to establish a correct setup of our devices. They need to be detected by SteamVR and only when the detection succeeds, we can guarantee the working of our system. We provide the setups for our two chosen HMDs.

1) HTC Vive: The HTC Vive provides the most flawless experience under both Linux and Windows. We setup the base stations diagonally, which determines the room in which we can move and we hook up all cables from the computer to the HMD. Next, we need to install Steam and SteamVR (within Steam). On both Linux and Windows, after launching the SteamVR client the HTC Vive will automatically be detected. From there we can install and compile our own software.

2) PS VR: For the PS VR we need to implement a driver, as steam does not provide drivers out-of-the-box. There exist programs like iVRy [24], but work only under Windows. Fortunately, OpenVR gives us the possibility to develop our own driver. There exist already open-source examples for this like OpenHMD [5] and OpenPSVR [19]. For our use case we had the most success using OpenHMD, as it also provides compatibility with the HTC Vive. Once the HMD is detected by SteamVR, we're able to apply our design.

V. Benchmarks

In order to measure the performance of this system, we perform some experiments to determine the latency (visible in figure 1) between the actual movement of the HMD versus the virtual detected movement.

A. Initial Goals

In order to measure this latency we envisioned 2 possibilities. The first being a robotic arm, that moves the HMD at a certain time and detect time of the positional change in the subscriber of our system. If we run both programs on the same computer, we use the same system times and give us an accurate result of the delay. Additionally, the robotic arm can only be controlled under Linux, which is fortunately no problem for our system.

The second approach would be to use a camera and record the movement of the HMD and computer detection of this movement, in the same shot. When the camera is capable

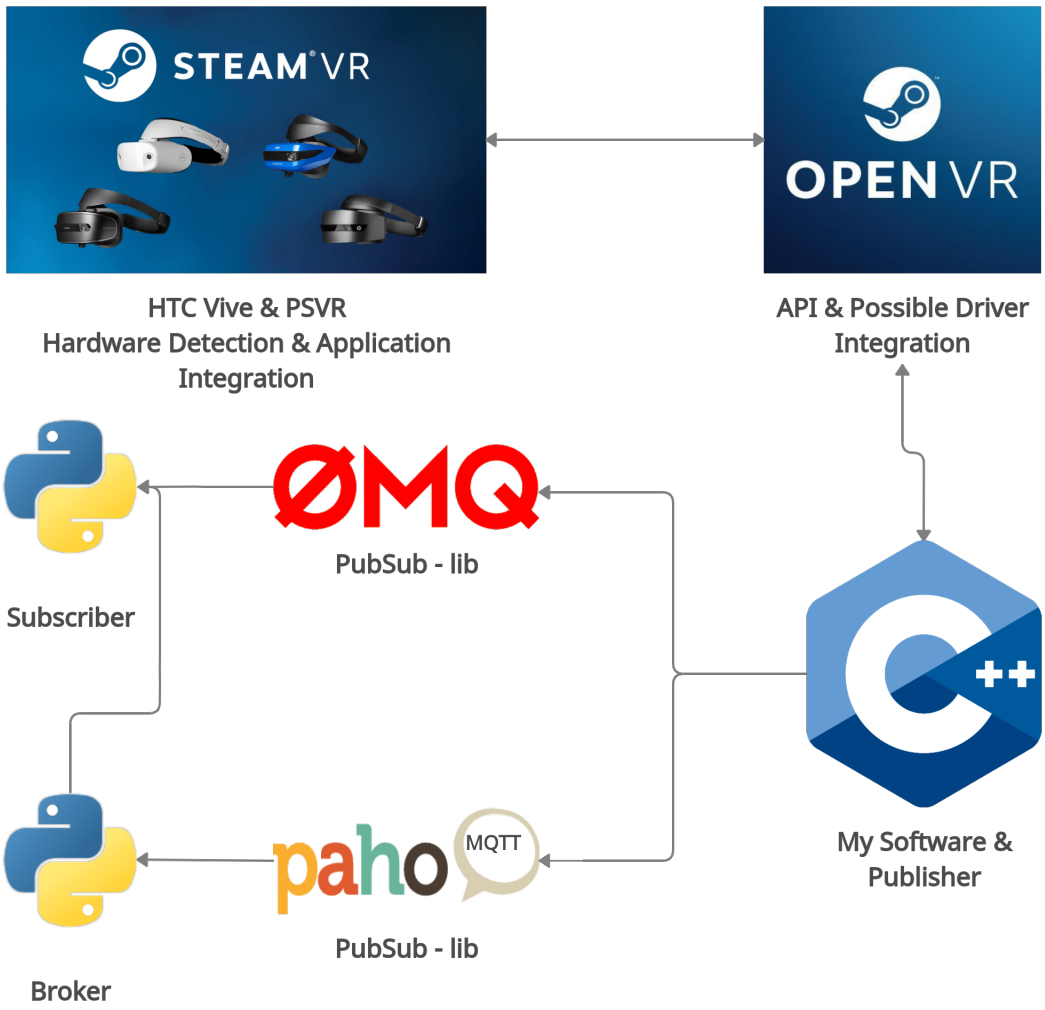

Fig. 2: The implemented design.

of recording at high frames per second, we can measure the delay by counting the frames it takes to detect the movement on the display of the computer.

B. Problems

As we had access to a UR10 robotic arm, we first attempted this approach. Unfortunately, we ran into practical issues with the setup. The robotic arm needs to perform a steady and quick motion holding the HMD in order to reach accurate results. However, typically HMDs are bulky objects and the robotic arm has only a small clamp to hold the HMD. Both the HTC Vive and PSVR do not feature a suitable mounting point for the robotic arm.

C. Solutions

Proposed solutions for the robotic arm could be: buying an larger clamp or a 3D-printed attachment, as done in [10]. For now, we are constrained to use a high FPS camera, but by lack of having one, we try the next best thing: a smartphone. Of course this solution will not have exact precision, but it will provide an order of magnitude in latency.

D. Setup

The setup is quite simple: we display all context information on the screen (positional and rotational coordinates), together with the system clock in milliseconds. This way we can see which position the HMDs take at which time. Next, we use our smartphone to record video at 240 FPS, this means that between each frame is a delay of 4.167 ms. Normally, this would determine the precision and thus define the fault margin and unit. However, one could argue that, if the experiment is repeated enough times, the additional delay introduced by this effect averages out to half of the inter-frame time and could be eliminated as such. The camera and the instant the rotation starts are in no way synchronized, so with enough repetitions, the extra delay should be distributed uniformly in [0, 4.167]

ms with an average of half that. For each combination we repeat the process six times.

HMD	Computer	Smartphone
Model	/	Pixel 4a 5G
Record speed	/	240FPS
OS	Ubuntu 20.04	/
Processor	AMD Ryzen 5 3600x	/
Memory	32 GiB	/
Graphics	GeForce GTX 1660 Ti	/

TABLE III: Hardware

The experiment then consist of making a quick rotational motion with the HMD (because the PSVR does not provide positional coordinates), displaying these coordinates together with the system time and record both the screen and HMD motion in one shot in slow motion, as shown in figure 3. Then, we determine the delay by analysing all frames of the video. The start time is the frame where we first see the start of the motion of the HMD and the stop time is the frame where we see an increase in rotational coordinates. The hardware that is used is listed in table III.

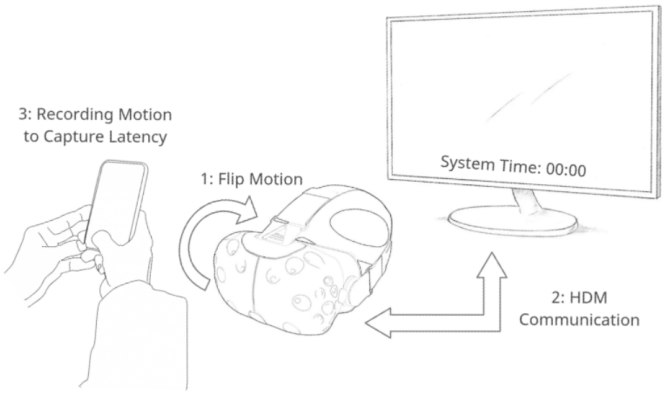

Fig. 3: The setup for benchmarking. Src: [3]

E. Results

We repeat the experiment between different HMDs (HTC Vive & PSVR) as well as different publishers (ZMQ & MQTT), to see if there are any significant differences in delay. The results are shown in table IV.

HMD	ZMQ (in ms)	MQTT (in ms)
HTC Vive	131	128
PS VR	134	150

TABLE IV: First results of measuring delays (1 example).

These first results are a raw estimate of the delay, because of several unexpected issues that occurred during the experimentation. The first issue being that the sensors experience the notion of 'drift', meaning while the HMD is perfectly laying still, the sensors still detect quite a lot of change, which makes it very hard to determine the exact point/frame when the motion is shown on the display.

Secondly, the screen itself acts as a bottleneck of our recording. Despite recording at 240FPS, the screen has only a refresh rate of 60FPS. This means that in between two frames of our recording the same system times and coordinates are shown on the display and between two frames of the display multiple coordinates are shown. All this make it very hard to determine the exact start- and stop times.

Fortunately, this 'drift' that has been observed, changes in (almost) a constant way. This means, if we show the absolute difference between each coordinate change (basically the absolute of the derivative function), the difference will become more clear as the motion will clearly differ from the constant values by a significant increase. Note that dividing this value by the time interval, becomes the velocity in e.g. radians per second.

Now, we can determine the exact time the motion is detected by our subscriber. To make it even more easy to determine the exact stop time, we can plot this difference for each system clock time to get an accurate result. Now, only the start time needs to be determined by the slow motion recording and we can determine our delay way more precise than before.

Additionally, we need to change the motion we perform on the HMD. First, we placed the HMD on its side and turned it as quickly as possible to the table. If we analyse the first graph 4, we see it is still hard to determine the exact system time at which the motion occurs. We solve this by changing our motion from a quick turn to a flip. We put the HMD on an edge of the table and flip it. This way we can hit the HMD in one quick and fast motion. The result is visible in figure 5. Now the exact moment in time the motion is detected is indisputable and very accurate. With this new setup we show the final results in table V.

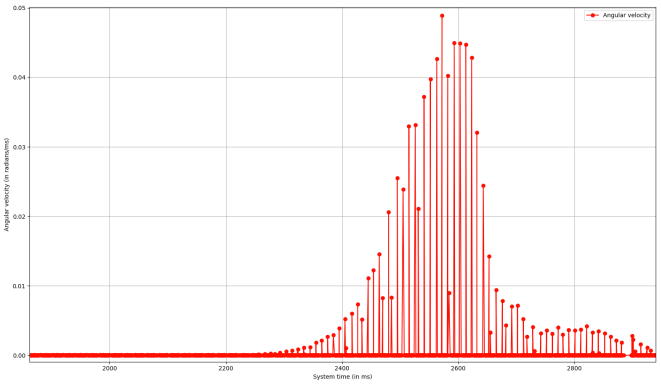

Fig. 4: Quickly turning the HMD (one example).

Now that we consist of more accurate results, we can analyze the results with confidence. The first important aspect to assess is the interval rate at which the C++ publisher sends coordinates. For MQTT, we were obligated to slow down the rate to 10 ms, as the broker in the MQTT protocol could not

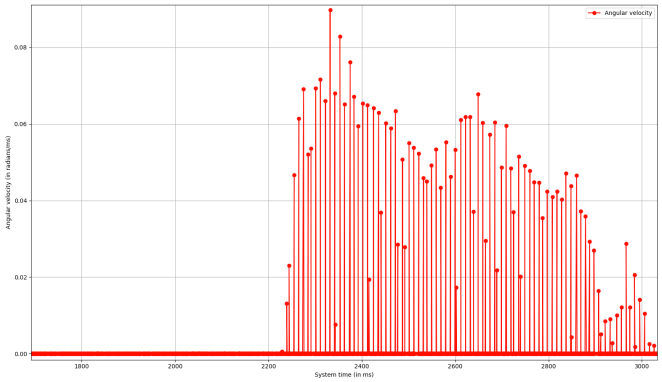

Fig. 5: Flipping the HMD (one example).

HMD	ZMQ (in ms) (MIN/MAX/AVG)	MQTT (in ms) (MIN/MAX/AVG)
Interval / update rate	±0.4	±11
HTC Vive	77 / 99 / 90	96 / 130 / 111
PS VR	72 / 102 / 89	87 / 118 / 107

TABLE V: Final results of measuring delays (5 time average).

handle such a high rate of 0.4 ms. Possible explanations are the use of a broker that is run on the same system, that needs to share network resources. Of course this has influence in the precision at which the motion is detected, causing a worst case delay of 11 ms. So it seems that the use of a broker is the reason why MQTT performs (around 20 ms) slower than ZMQ. More interesting is that the latency between the different HMDs is very similar, indicating that the OpenHMD driver doesn't introduce any extra latency compared to the more optimized drivers of the HTC Vive.

Unfortunately, the observed latencies in the 100 ms range are arguably not sufficient for enabling truly immersive VR systems. The main reason comes from the fact that the latencies are too high, for immersive VR we need at least 10-20 ms round trip delay as shown in [22].

Adversely, the benchmarked communication latencies are sufficient for enabling several context-based network optimization procedures. For example, the characterized latencies suffice for supporting VR setups in which the transmit communication beams are always aimed in the LoS directions of the VR users [26]. Such optimizations can operate well even with second-large communication delays [9].

VI. Conclusions & Future Work

In this research project, we have designed and developed a future-proof framework that can provide context information of the rotational and positional coordinates. The framework is highly modular, as publish-subscribe mechanisms can easily be replaced. Additionally, the program is plug-and-play to a variety of HMDs (depending on its popularity and novelty) that are manufacturer independent and works cross-platform, provided that the appropriate drivers are available. Drivers for Windows are more commonly available and up-to-date compared to their Linux counterparts. Nevertheless using the OpenHMD driver, we can access a variety of extra HMDs out-of-the-box even under Linux [6].

This provides a way to access real-time data to be used by any other program and is showed off by two demos. This latency is benchmarked in a simple but efficient way and gives us an order of magnitude of the delay, with a fault margin of 5 ms. Our system currently features time resolution accuracy at the level of 90 ms, which is a bit disappointing when round trip delays of 10-20 ms provide full immersive VR [22]. To further improve this accuracy, one could consider one of the following experiments.

In order to determine more accurately the latency, we need a more exact measurement of the start time of the physical movement. A possible solution is attaching a small additional high refresh rate screen to show the system time for the high FPS camera. Another solution is attaching a LED that somehow shows the clock at higher granularity (e.g. long blink at every 1s, short blink at every 0.1s). This way a more accurate latency could be acquired. Other methods to improve the precision of the measurements are using higher FPS cameras or a robotic arm with better mounting points (greater clamp, 3D-printed holders). Second, the high latency could be improved by better understanding its origin. This is possible by repeating the same experiment but in a Windows environment. This could exclude any unoptimized driver issues under Linux. Another approach is to repeat the experiment without publish-subscribe mechanisms and directly in the C++ application. This would give us the latency introduced by the publish-subscribe mechanisms. At last we can also try to substitute the Python subscriber for a C++ equivalent, as typically Python is considered to run slower than C++. All these methods drive towards a better understanding of where the source of the increased latency is located. When one reduces this latency further, the complete framework would act as a gateway for building interesting measurement tools.

VII. Demonstrations & Software

We show the workings of the system by two video-demos. The demos shows that a Python Subscriber can easily access this information and display a 3D-model following the rotational coordinates of the VR user in real-time. The first demo displays the workings with the HTC-Vive [1] and the second one with the PSVR [2]. The software is open-source and provided by a GitHub repository [3] with step by step instructions for how to install the software, so anyone can build their own subscribers or improve performance.

[1] https://www.youtube.com/watch?v=3XEQG9NMxoM
[2] https://www.youtube.com/watch?v=SxlgeaNkngw
[3] https://github.com/Mathias-Ooms/Context-Aware-VR

REFERENCES

[1] Kamran Bigdely-Shamloo. How to get raw (positional) data from htc vive? https://www.codeproject.com/Articles/1171122/How-to-Get-Raw-Positional-Data-from-HTC-Vive (visited: 06/2021).

[2] Massimiliano Di Luca. New method to measure end-to-end delay of virtual reality. *Presence: Teleoperators and Virtual Environments*, 19(6):569–584, 2010.

[3] SteroeLab DrawingForall, Stockio. Drawings. https://www.stockio.com/free-clipart/cell-phone-2 & https://www.stereolabs.com/zed-mini/setup/vive/ & https://www.drawingforall.net/how-to-draw-a-monitor/ (visited: 06/2021).

[4] Mohammed S Elbamby, Cristina Perfecto, Mehdi Bennis, and Klaus Doppler. Toward low-latency and ultra-reliable virtual reality. *IEEE Network*, 32(2):78–84, 2018.

[5] Christoph Haag. Openhmd. https://github.com/OpenHMD/OpenHMD (visited: 06/2021).

[6] Christoph Haag. Openhmd list of compatible headsets. https://github.com/OpenHMD/OpenHMD/wiki/Support-List (visited: 06/2021).

[7] John Haas. A history of the unity game engine. *Diss. WORCESTER POLYTECHNIC INSTITUTE*, 2014.

[8] Julian Horsey. Droolon f1 vr eye tracking adapter. https://www.geeky-gadgets.com/vr-eye-tracking-25-09-2019/ (visited: 06/2021).

[9] Xueshi Hou, Jianzhong Zhang, Madhukar Budagavi, and Sujit Dey. Head and body motion prediction to enable mobile vr experiences with low latency. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2019.

[10] Tyler A Jost, Bradley Nelson, and Jonathan Rylander. Quantitative analysis of the oculus rift s in controlled movement. *Disability and Rehabilitation: Assistive Technology*, pages 1–5, 2019.

[11] Pupil Labs. Pupil labs vr-ar. https://pupil-labs.com/products/vr-ar/ (visited: 06/2021).

[12] Matias Nassi. Introduction to openvr 101 series: What is openvr and how to get started with its apis. https://skarredghost.com/2018/03/15/introduction-to-openvr-101-series-what-is-openvr-and-how-to-get-started-with-its-apis (visited: 06/2021).

[13] Niels C Nilsson, Stefania Serafin, and Rolf Nordahl. Unintended positional drift and its potential solutions. In *2013 IEEE Virtual Reality (VR)*, pages 121–122. IEEE, 2013.

[14] Oculus. Get started developing with oculus. https://developer.oculus.com/get-started/ (visited: 06/2021).

[15] Oculus. Oculus go. https://www.oculus.com/go/?locale=nl_NL (visited: 06/2021).

[16] Oculus. Oculus quest. https://www.oculus.com/quest/features/ (visited: 06/2021).

[17] Oculus. Oculus quest 2. https://www.oculus.com/quest-2/ (visited: 06/2021).

[18] Oculus. Oculus rift s. https://www.oculus.com/rift-s/ (visited: 06/2021).

[19] Stephen O'Hair. Openpsvr. https://github.com/alatnet/OpenPSVR (visited: 06/2021).

[20] Pico. Pico neo 2. https://www.pico-interactive.com/us/neo2.html (visited: 06/2021).

[21] PlayStation. Playstation vr. https://www.playstation.com/nl-be/ps-vr/ (visited: 06/2021).

[22] Kjetil Raaen and Ivar Kjellmo. Measuring latency in virtual reality systems. In *International Conference on Entertainment Computing*, pages 457–462. Springer, 2015.

[23] Andrew Sanders. *An introduction to Unreal engine 4*. CRC Press, 2016.

[24] Mediator Software. ivry driver for steamvr demo (psvr lite edition). https://store.steampowered.com/app/1005970/iVRy_Driver_for_SteamVR_DEMO_PSVR_Lite_Edition/?l=dutch (visited: 06/2021).

[25] Beatriz Soret, Preben Mogensen, Klaus I Pedersen, and Mari Carmen Aguayo-Torres. Fundamental tradeoffs among reliability, latency and throughput in cellular networks. In *2014 IEEE Globecom Workshops (GC Wkshps)*, pages 1391–1396. IEEE, 2014.

[26] Jakob Struye, Filip Lemic, and Jeroen Famaey. Millimeter-wave beamforming with continuous coverage for mobile interactive virtual reality. *arXiv preprint arXiv:2105.11793*, 2021.

[27] Tobii. Tobii/xr sdk - documentation. https://vr.tobii.com/sdk/develop/ (visited: 06/2021).

[28] Valve. Openvr documentation and source code. https://partner.steamgames.com/doc/features/steamvr/openvr?l=dutch (visited: 06/2021), https://github.com/ValveSoftware/openvr (visited: 06/2021).

[29] Vive. Htc pro eye. https://www.vive.com/eu/product/vive-pro-eye/overview/ (visited: 06/2021).

[30] Vive. Htc vive. https://en.wikipedia.org/wiki/HTC_Vive (visited: 06/2021).

[31] Vive. Htc vive cosmos. https://www.vive.com/us/product/vive-cosmos/features/ (visited: 06/2021).

[32] Vive. Htc vive cosmos elite. https://www.vive.com/us/product/vive-cosmos-elite/overview/ (visited: 06/2021).

[33] Vive. Htc vive focus plus. https://business.vive.com/eu/product/focus-plus/ (visited: 06/2021).

[34] Vive. Vive-overview. https://developer.vive.com/resources/vive-sense/sdk-compatibility-overview/ (visited: 06/2021).

[35] Vive. Vive-sdk. https://developer.vive.com/us/ (visited: 06/2021).

[36] Cedric Westphal. Challenges in networking to support augmented reality and virtual reality. *IEEE ICNC*, 2017.

[37] Qinping Zhao. 10 scientific problems in virtual reality. *Communications of the ACM*, 54(2):116–118, 2011.