Low-latency Cloud-based Volumetric Video Streaming Using Head Motion Prediction

Serhan Gül
Fraunhofer HHI
Berlin, Germany
serhan.guel@hhi.fraunhofer.de

Dimitri Podborski
Fraunhofer HHI
Berlin, Germany
dimitri.podborski@hhi.fraunhofer.de

Thomas Buchholz
Deutsche Telekom AG
Berlin, Germany
thomas.buchholz@telekom.de

Thomas Schierl
Fraunhofer HHI
Berlin, Germany
thomas.schierl@hhi.fraunhofer.de

Cornelius Hellge
Fraunhofer HHI
Berlin, Germany
cornelius.hellge@hhi.fraunhofer.de

ABSTRACT

Volumetric video is an emerging key technology for immersive representation of 3D spaces and objects. Rendering volumetric video requires lots of computational power which is challenging especially for mobile devices. To mitigate this, we developed a streaming system that renders a 2D view from the volumetric video at a cloud server and streams a 2D video stream to the client. However, such network-based processing increases the motion-to-photon (M2P) latency due to the additional network and processing delays. In order to compensate the added latency, prediction of the future user pose is necessary. We developed a head motion prediction model and investigated its potential to reduce the M2P latency for different look-ahead times. Our results show that the presented model reduces the rendering errors caused by the M2P latency compared to a baseline system in which no prediction is performed.

KEYWORDS

volumetric video, augmented reality, mixed reality, cloud streaming, head motion prediction

1 INTRODUCTION

Recent advances in hardware for displaying of immersive media have aroused a huge market interest in virtual reality (VR) and augmented reality (AR) applications. Although the initial interest was focused on omnidirectional (360°) video applications, with the improvements in capture and processing technologies, volumetric video has recently started to become the center of attention [21]. Volumetric videos capture the 3D space and objects and enable services with six degrees of freedom (6DoF), allowing a viewer to freely change both the position in space and the orientation.

Although the computing power of mobile end devices has dramatically increased in the recent years, rendering rich volumetric objects is still a very demanding task for such devices. Moreover, there are yet no efficient hardware decoders for volumetric content (e.g. point clouds or meshes), and software decoding can be prohibitively expensive in terms of battery usage and real-time rendering requirements. One way of decreasing the processing load on the client is to avoid sending the volumetric content and instead send a 2D rendered view corresponding to the position and orientation of the user. To achieve this, the expensive rendering process needs to be offloaded to a server infrastructure. Rendering 3D graphics on a powerful device and displaying the results on a *thin* client connected through a network is known as remote (or interactive) rendering [26]. Such rendering servers can be deployed at a cloud computing platform such that the resources can be flexibly allocated and scaled up when more processing load is present.

In a cloud-based rendering system, the server renders the 3D graphics based on the user input (e.g, head pose) and encodes the rendering result into a 2D video stream. Depending on the user interaction, camera pose of the rendered video is dynamically adapted by the cloud server. After a matching view has been rendered and encoded, the obtained video stream is transmitted to the client. The client can efficiently decode the video using its hardware video decoders and display the video stream. Moreover, network bandwidth requirements are reduced by avoiding the transmission of the volumetric content.

Despite these advantages, one major drawback of cloud-based rendering is an increase in the end-to-end latency of the system, typically known as motion-to-photon (M2P) latency. Due to the added network latency and processing delays (rendering and encoding), the amount of time until an updated image is presented to the user is greater than a local rendering system. It is well-known that an increase in M2P latency may cause an unpleasant user experience and motion sickness [1, 3]. One way to reduce the network latency is to move the volumetric content to an *edge* server geographically closer to the user. Deployment of real-time communication protocols such as WebRTC are also necessary for ultra-low latency video streaming applications [10]. The processing latency at the rendering server is another significant latency component. Therefore, using fast hardware-based video encoders is critical for reducing the encoding latency.

Another way of reducing the M2P latency is to predict the future user pose at the remote server and send the corresponding rendered view to the client. Thus, it is possible to reduce or even completely eliminate the M2P latency, if the user pose is predicted for a look-ahead time (LAT) equal to or larger than the M2P latency of the system [6]. However, mispredictions of head motion may potentially degrade the user's Quality of Experience (QoE). Thus, design of accurate prediction algorithms has been a popular research area, especially for the viewport prediction for 360° videos (see Section 2.3). However, application of such algorithms to 6DoF movement (i.e. translational and rotational) has not yet been investigated.

In this paper, we describe our cloud-based volumetric streaming system, present a prediction model to forecast the 6DoF position of the user and investigate the achieved rendering accuracy using the developed prediction model. Additionally, we present an analysis of the latency contributors in our system and a simple latency measurement technique that we used to characterize the the M2P latency of our system.

2 BACKGROUND

2.1 Volumetric video streaming

Some recent works present initial frameworks for streaming of volumetric videos. Qian et al. [18] developed a proof-of-concept point cloud streaming system and introduced optimizations to reduce the M2P latency. Van der Hooft et al. [29] proposed an adaptive streaming framework compliant to the recent point cloud compression standard MPEG V-PCC [22]. They used their framework for HTTP adaptive streaming of scenes with multiple dynamic point cloud objects and presented a rate adaptation algorithm that considers the user's position and focus. Petrangeli et al. [17] proposed a streaming framework for AR applications that dynamically decides which virtual objects should be fetched from the server as well as their level-of-details (LODs), depending on the proximity of the user and likelihood of the user to view the object.

2.2 Cloud rendering systems

The concept of remote rendering was first put forward to facilitate the processing of 3D graphics rendering when PCs did not have sufficient computational power for intensive graphics tasks. A detailed survey of interactive remote rendering systems in the literature is presented in [26]. Shi et al. [25] proposed a Mobile Edge Computing (MEC) system to stream AR scenes containing only the user's field-of-view (FoV) and a latency-adaptive margin around the FoV. They evaluate the performance of their prototype on a MEC node connected to a 4G (LTE) testbed. Mangiante et al. [15] proposed an edge computing framework that performs FoV rendering of 360° videos. Their system aims to optimize the required bandwidth as well as reduce the processing requirements and battery utilization.

Cloud rendering has also started to receive increasing interest from the industry, especially for cloud gaming services. Nvidia CloudXR [16] provides an SDK to run computationally intensive extended reality (XR) applications on Nvidia cloud servers to deliver advanced graphics performances to thin clients.

2.3 Head motion prediction techniques

Several sensor-based methods have been proposed in the literature that attempt to predict the user's future viewport for optimized streaming of 360°-videos. Those can be divided into two categories. Works such as [4, 6, 7, 13, 20] were specifically designed for VR applications and use the sensor data from head-mounted displays (HMDs) whereas the works in [5, 8, 12] attempt to infer user motion based on some physiological data such as electroencephalogram (EEG) and electromyography (EMG) signals. Bao et al. [6] collected head orientation data and exploited the correlations in different dimensions to predict the head motion using regression techniques. Their findings indicate that LATs of 100-500 ms is a feasible range for sufficient prediction accuracy. Sanchez et al. [20] analyzed the effect of M2P latency on a tile-based streaming system and proposed an angular acceleration-based prediction method to mitigate the impact on the observed fidelity. Barniv et al. [8] used the myoelectric signals obtained from EMG devices to predict the impending head motion. They trained a neural network to map EMG signals to trajectory outputs and experimented with combining EMG output with inertial data. Their findings indicate that a LATs of 30-70 ms are achievable with low error rates.

Most of the previous works target three degrees of freedom (3DoF) VR applications and thus focus on prediction of only the head orientation in order to optimize the streaming 360° videos. However, little work has been done so far on prediction of 6DoF movement for advanced AR and VR applications. In Sec. 5, we present an initial statistical model for 6DoF prediction and discuss our findings.

3 SYSTEM ARCHITECTURE

This section presents the system architecture of our cloud rendering-based volumetric video streaming system and describes its different components. A simplified version of this architecture is shown in Fig. 1.

3.1 Server architecture

The server-side implementation is composed of two main parts: a volumetric video player and a cross-platform cloud rendering library, each described further in more detail.

Volumetric video player. The volumetric video player is implemented in Unity and plays a single MP4 file which has one video track containing the compressed texture data and one mesh track containing the compressed mesh data of a volumetric object. Before the playback of a volumetric video starts, the player registers all the required objects. For example, the virtual camera of the rendered view and the volumetric object are registered, and those can later be controlled by the client. After initialization, the volumetric video player can start playing the MP4 file. During playout, both tracks are demultiplexed and fed into the corresponding decoders; video decoder for texture track and mesh decoder for mesh track. After decoding, each mesh is synchronized with the corresponding texture and rendered to a scene. The rendered view of the scene is represented by a Unity *RenderTexture* that is passed to our cloud rendering library for further processing. While rendering the scene, the player concurrently asks the cloud rendering library for the latest positions of the relevant objects that were previously registered in the initialization phase.

Cloud rendering library. We created a cross-platform cloud rendering library written in C++ that can be integrated into a variety of applications. In the case of the Unity application, the library is integrated into the player as a native plugin. The library utilizes the GStreamer WebRTC plugin for low-latency video streaming between the server and client that is integrated into a media pipeline as described in Sec. 3.3. In addition, the library provides interfaces for registering the objects of the rendered scene and retrieving the latest client-controlled transformations of those objects while rendering the scene. In the following, we describe the modules of

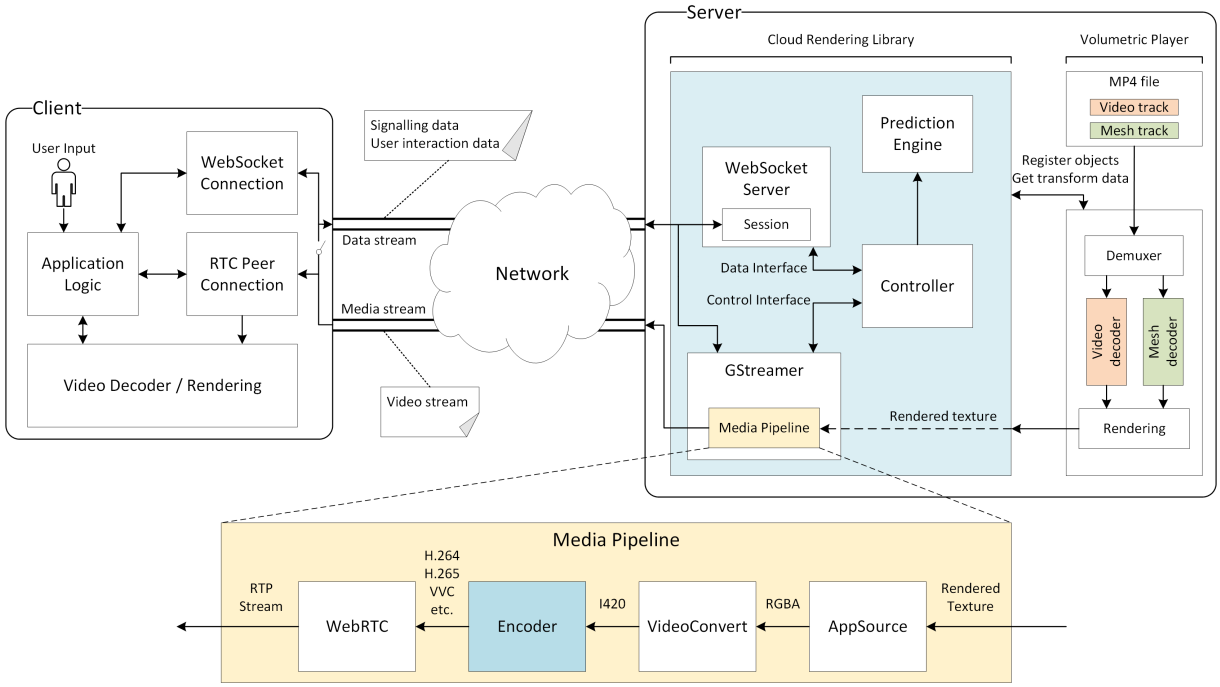

Figure 1: Overview of the system components and interfaces.

our library, each of which runs asynchronously in its own thread to achieve high performance.

The **WebSocket Server** is used for exchanging signaling data between the client and the server. Such signaling data includes Session Description Protocol (SDP), Interactive Connectivity Establishment (ICE) as well as application-specific metadata for scene description. In addition, WebSocket (WS) connection can also be used for sending the control data, e.g. changing the position and orientation of any registered game object or camera. Both, plain WebSockets as well as Secure WebSockets are supported which is important for practical operation of the system.

The **GStreamer** module contains the media processing pipeline which takes the rendered texture and compressed it into a video stream that is sent to the client using the WebRTC plugin. The most important components of the media pipeline are described in Sec. 3.3.

The **Controller** module represents the application logic and controls the other modules depending on the application state. For example, it closes the media pipeline if the client disconnects, re-initializes the media pipeline when a new client has connected, and updates the controllable objects based on the output of the Prediction Engine.

The **Prediction Engine** implements a regression-based prediction method (please refer to Sec. 5) and provides interfaces for usage of other potential methods. Based on the previously received input from the client and the implemented algorithm, the module updates the position of the registered objects accordingly such that the rendered scene corresponds to the predicted positions of the object after a given LAT.

3.2 Client architecture

The client-side architecture is depicted on the left side of the Fig. 1. Before the streaming session starts, the client establishes a WS connection to the server and asks the server to send a description of the rendered scene. The server responds with a list of objects and parameters which the client is later allowed to update. After receiving the scene description, the client replicates the scene and initiates a peer-to-peer (P2P) WebRTC connection to the server. The server and client begin the WebRTC negotiation process by sending SDP and ICE data over the established WS connection. Finally, the P2P connection is established, and the client starts receiving a video stream corresponding to the current view of the volumetric video. At the same time, the client can use the WS connection, as well as the RTCPeerConnection for sending control data to the server in order to modify the properties of the scene. For example, the client may change its 6DoF position, or it may rotate, move and scale any volumetric object in the scene.

We have implemented both a web player in JavaScript and a native application for the HoloLens, the untethered AR headset from Microsoft. While our web application targets VR, our HoloLens application is implemented for AR use cases. In the HoloLens application, we perform further processing to remove the background of the video texture before rendering the texture onto the AR display. In general, the client-side architecture remains the same for both VR and AR use cases, and the most complex client-side module is the video decoder. Thus, the complexity of our system is concentrated largely in our cloud-based rendering server.

3.3 Media pipeline

The simplified structure of the media pipeline is shown in the bottom part of Fig. 1. A rendered texture is given to the media pipeline as input using the *AppSource* element of Gstreamer. Since the rendered texture is originally in RGB format but the video encoder requires YUV input, we use the *VideoConvert* element to convert the RGB texture to I420 format[1]. After conversion, the texture is passed to the encoder element, which can be set to any supported encoder on the system. Since encoder latency is a significant contributor the overall M2P latency, we evaluated the encoding performances of different encoders for a careful selection. For detailed results on the encoder performance, please refer to Sec. 4.1.

After the texture is encoded the resulting video bitstream is packaged into RTP packets, encrypted and sent to the client using WebRTC. WebRTC was chosen as the delivery method since it allows us to achieve an ultra-low latency while using the P2P connection between the client and server. In addition, WebRTC is already widely adopted by different web browsers allowing our system to support several different platforms.

4 MOTION-TO-PHOTON LATENCY

The different components of the M2P latency are illustrated in Fig. 2 and related by

$$T_{M2P} = T_{server} + T_{network} + T_{client} \quad (1)$$

where T_{server}, T_{client} and $T_{network}$ consist of the following component latencies:

$$T_{server} = T_{rend} + T_{enc} \quad (2)$$

$$T_{network} = T_{up} + T_{down} + T_{trans} \quad (3)$$

$$T_{client} = T_{dec} + T_{disp} \quad (4)$$

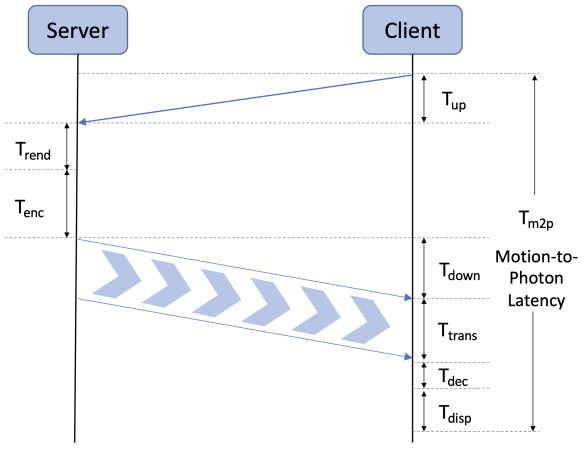

Figure 2: Components of the motion-to-photon latency for a remote rendering system.

In our analysis, we neglect the time for the HMD to compute the user pose using its tracker module. This computation is typically based on a fusion of the sensor data from the inertial measurement units (IMUs) and visual data from the cameras. Although AR device cameras typically operate at 30-60 Hz, IMUs are much faster and a reliable estimation of the user pose can be performed with a frequency of multiple kHz [14, 30]. Thus, the expected tracker latency is on the order of microseconds.

T_{enc} is the time to compress a frame and depends on the encoder type (hardware or software) and the picture resolution. We present a detailed latency analysis of different encoders that we tested for our system in Section 4.1.

$T_{network}$ is the network round-trip time (RTT). It consists of the propagation delay (T_{up} + T_{down}) and the transmission delay T_{trans}. T_{up}, the time for the server to retrieve sensor data from the client, and T_{down} the time for the server to transmit a compressed frame to the client.

T_{rend} is the time for the server to generate a new frame by rendering a view from the volumetric data based on the actual user pose. In general, it can be set to match the frame rate of the encoder for a given rendered texture resolution.

T_{dec} is the time to decode a compressed frame on the client device and is typically much smaller than T_{enc} since video decoding is inherently a faster operation than video encoding. Also, the end devices typically have hardware-accelerated video decoders that further reduce the decoding latency.

T_{disp} is the display latency and mainly depends on the refresh rate of the display. For a typical refresh rate of 60 Hz, the average value of T_{disp} is 8.3 ms, and the worst-case value is 16.6 ms in case the decoded frame misses the current VSync signal and has to wait in the frame buffer for the next VSync signal.

4.1 Encoder latency

We characterized the encoding speeds of different encoders using the test dataset provided by ISO/ITU Joint Video Exploration Team (JVET) for the next generation video coding standard Versatile Video Coding (VVC) [24]. In our measurements, we used the FFmpeg libraries of the encoders NVENC, x264, x265, and Intel SVT-HEVC, enabled their low-latency presets and measured the encoded frames per second (FPS) using FFmpeg *-benchmark* option. The measurements were performed on a Ubuntu 18.04 machine with 16 Intel Xeon Gold 6130 CPU (2.10GHz) CPUs using the default threading options for the tested software-based encoders. For x264 and x265, we used the *ultrafast* preset and *zerolatency* tuning [9]. For NVENC, we evaluated the presets default, high-performance (HP), low-latency (LL) and low-latency high-performance (LLHP). A brief description of the NVENC presets can be found in [28].

Table 1 shows the mean FPS over all tested sequences for different encoders. We observed that both H.264 and HEVC encoders of NVENC are significantly faster than x264 andx265 (both using ultrafast preset and zerolatency tuning) as well as SVT-HEVC (Low delay P). NVENC is able to encode 1080p and 4K videos in our test dataset with encoding speeds up to 800 fps and 200 fps, respectively. We also observed that for some sequences, HEVC encoding turned out to be faster than H.264 encoding. We believe that this difference is caused by a more efficient GPU implementation for HEVC.

All the low-latency presets tested in our experiments turn off B-frames to reduce latency. Despite that, we observed that the picture quality obtained by NVENC in terms of PSNR is comparable to the other tested encoders (using low-latency presets). As a result of

[1]https://www.fourcc.org/pixel-format/yuv-i420

our analysis, we decided to use NVENC H.264 (HP preset) in our system.

Table 1: Mean encoding performances over all tested sequences for different encoders and presets.

Standard	Encoder	Preset	Mean FPS
H.264	x264	Ultrafast	81
	NVENC	Default	353
	NVENC	HP	**465**
	NVENC	LL	359
	NVENC	LLHP	281
HEVC	x265	Ultrafast	33
	SVT-HEVC	Low delay P	74
	NVENC	Default	212
	NVENC	HP	**492**
	NVENC	LL	278
	NVENC	LLHP	211

4.2 Latency measurements

We developed a framework to measure the M2P latency of our system. In our setup, we run the server application on an Amazon EC2 instance in Frankfurt, and the client application runs in a web browser in Berlin which is connected to the Internet over WiFi.

We implemented a server-side console application which is using the same cloud rendering library as described in Sec. 3.1 but instead of sending the rendered textures from the volumetric video player, the application sends predefined textures (known by the client) depending on the received control data from the client. These textures consist of simple vertical bars with different colors. For example, if the client instructs the server application to move the main camera to position P_1, the server pushes the texture F_1 into the media pipeline. Similarly, another camera position P_2 results in the texture F_2.

On the client side, we implemented a web-based application that connects to the server application and renders the received video stream to a canvas. Since the client knows exactly how those textures look like, it can evaluate the incoming video stream and determine when the requested texture was rendered on the screen. As soon as the client application sends P_1 to the server, it starts the timer and checks the canvas for F_1 at every web browser window *repaint* event. According to the W3C recommendation [19], the repaint event matches the refresh rate of the display. As soon as the texture F_1 is detected the client stops the timer and computes the M2P latency T_{M2P}.

Once the connection is established, the user can start the session by defining the number of independent measurements. Since we are using the second smallest instance type of Amazon EC2 (t2.micro), we set the size of each video frame to 512×512 pixels. We encode the stream using x264 configured with ultrafast preset and zerolatency tuning with an encoding speed of ~80 fps. As an example, we set the client to perform 100 latency measurements and calculated the average, minimum and maximum M2P latency. Our results show that T_{M2P} fluctuates between 41 ms and 63 ms, and the measured average M2P latency is 58 ms.

5 HEAD MOTION PREDICTION

One important technique to mitigate the increased M2P latency in a cloud-based rendering system is the prediction of the user's future pose. In this section, we describe our statistical prediction model for 6DoF head motion prediction and evaluate its performance using real user traces.

5.1 Data collection

We collected motion traces from five users while they were freely interacting with a static virtual object using Microsoft HoloLens. We recorded the users' movements in 6DoF space; i.e., collected position samples (x, y, z) and rotation samples represented as quaternions (q_x, q_y, q_z, q_w). Since the raw sensor data we obtained from HoloLens was unevenly sampled (i.e. different temporal distances between consecutive samples) at 60 Hz, we interpolated the data to obtain temporally equidistant samples. We upsampled the position data using linear interpolation and the rotation data (quaternions) using Spherical Linear Interpolation of Rotations (SLERP) [27]. Thus, we obtained an evenly-sampled dataset with a sampling rate of 200 Hz (one sample at each 5 ms).[1]

5.2 Prediction method

We use a simple autoregressive (AutoReg) model to predict the future user pose based on a time series of its past values. AutoReg models use a linear combination of the past values of a variable to forecast its future values [11].

An AutoReg model of lag order ρ can be written as

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_\rho y_{\rho-1} + \epsilon_t \tag{5}$$

where y_t is the true value of the time series y at time t, ϵ_t is the white noise, ϕ_i are the coefficients of the model. Such a model with ρ lagged values is referred to as an AR(ρ) model. Some statistics libraries can determine the lag order automatically using statistical tests such as the Akaike Information Criterion (AIC) [2].

We used the x and q_x values from one of the collected traces as training data and created two AutoReg models using the Python library *statsmodels* [23], for translational and rotational components, respectively. Our model has a lag order of 32 samples i.e. it considers a history window (hw) of the past $32 * 5 = 160$ ms and predicts the next sample using (5). Typically we need to predict not only the next sample but multiple samples in the future to achieve a given LAT; therefore, we repeat the prediction step by adding the just-predicted sample to the history window and iterating (5) until we obtain the future sample corresponding to the desired LAT. The process is then repeated for each frame e.g. each 10 ms for an assumed 100 Hz display refresh rate.

We used the trained model to predict the users' translational (x, y, z), and rotational motion (q_x, q_y, q_z, q_w). We perform the prediction of rotations in the quaternion domain since we readily obtain quaternions from the sensors and they allow smooth interpolation using techniques like SLERP. After prediction, we convert the predicted quaternions to Euler angles (yaw, pitch, roll) and evaluate the prediction accuracy in the domain of Euler angles since they are better suited for understanding the rendering offsets in terms of angular distances.

[1] Our 6DoF head movement dataset is freely available on Github for further usage in research community under: https://github.com/serhan-gul/dataset_6DoF

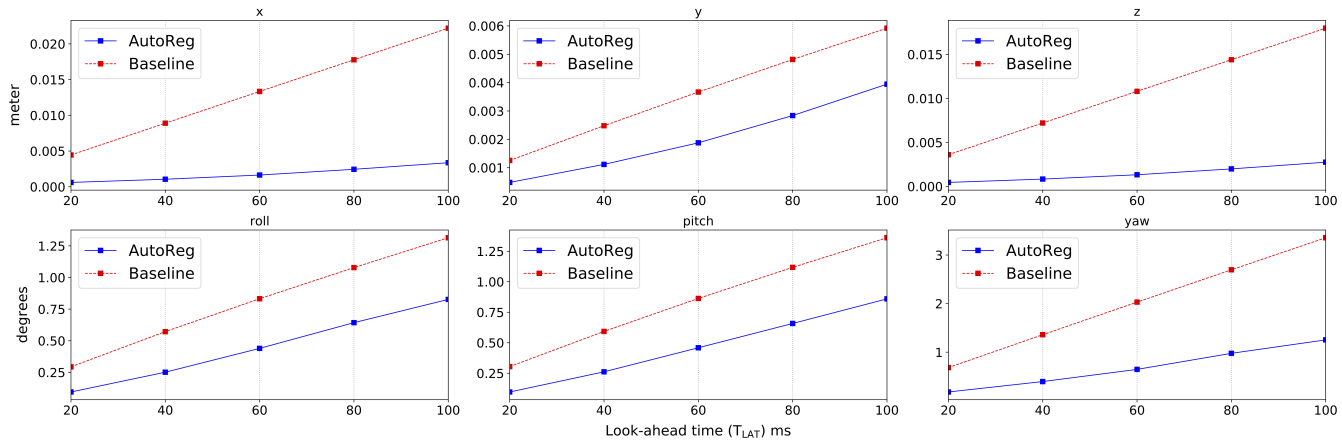

Figure 3: Mean absolute error (MAE) for different translational and rotational components averaged over five users. Results are given for the look-ahead times T_{LAT} in the range 20-100 ms.

5.3 Evaluation

In our evaluation, we investigated the effect of prediction on the accuracy of the rendered image displayed to the user, i.e. the rendering offset. Specifically, we compared the predicted user pose (given a certain look-ahead time T_{LAT}) to the real user pose as obtained from the sensors. As a benchmark, we evaluated a baseline case in which the rendered pose lags behind the actual user pose by a delay corresponding to the M2P latency (T_{M2P}), i.e., no prediction is performed.

For each user trace, we evaluated the prediction algorithm for a T_{LAT} ranging between 20-100 ms. In each experiment, we assume that the M2P latency is equal to the prediction time ($T_{LAT} = T_{M2P}$) such that the prediction model attempts to predict the pose that the user will attain at the time the rendered image is displayed to the user. We evaluated our results by computing the mean absolute error (MAE) between the true and predicted values for the different components.

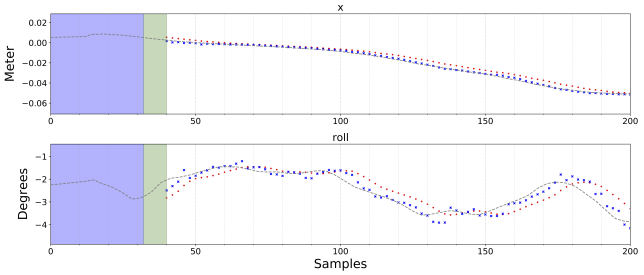

Figure 4: Comparison of the prediction (blue) and baseline (red) results for the *x* and *roll* components of one of the traces (sample time 5 ms; showing the time range 0-1 s) for T_{LAT} = 40 ms. The dashed gray line shows the recorded sensor data.

Fig. 3 compares the average rendering errors over five traces obtained using our prediction method to the baseline. We observe that for all considered T_{LAT}, prediction reduces the average rendering error for both positional and rotational components.

Fig. 4 shows for one of the traces the predicted and baseline (lagged by M2P latency) values for the *x* and *roll* components. At the beginning of the session, the prediction module collects the required amount of samples for a *hw* of 160 ms and makes the first prediction, i.e., the pose that the user is predicted to attain after a time of T_{LAT} = 40 ms (green shaded). We observe that the accuracy of the prediction depends on the frequency of the abrupt, short-term changes of the user pose. If a component of the user pose linearly changes over a *hw* (without changing direction), the resulting predictions for that component are fairly accurate. Otherwise, if short-term changes are present within a *hw*, the prediction tends to perform worse than the baseline.

6 CONCLUSION

We presented a cloud-based based volumetric streaming system that offloads the rendering to a powerful server and thus reduces the rendering load on the client-side. To compensate the added network and processing latency, we developed a method to predict the user's head motion in six degrees of freedom. Our results show that the developed prediction model reduces the rendering errors caused by the added latency due to the cloud-based rendering. In our future work, we will analyze the effect of motion-to-photon latency on the user experience through subjective tests and develop more advanced prediction techniques e.g. based on Kalman filtering.

REFERENCES

[1] Bernard D. Adelstein, Thomas G. Lee, and Stephen R. Ellis. 2003. Head Tracking Latency in Virtual Environments: Psychophysics and a Model. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting* 47, 20 (Oct. 2003), 2083–2087. https://doi.org/10.1177/154193120304702001

[2] Htrotugu Akaike. 1973. Maximum likelihood identification of Gaussian autoregressive moving average models. *Biometrika* 60, 2 (1973), 255–265. https://doi.org/10.1093/biomet/60.2.255

[3] R.S. Allison, L.R. Harris, M. Jenkin, U. Jasiobedzka, and J.E. Zacher. 2001. Tolerance of temporal delay in virtual environments. In *Proceedings IEEE Virtual Reality 2001*. IEEE Comput. Soc, 247–254. https://doi.org/10.1109/vr.2001.913793

[4] Tamay Aykut, Mojtaba Karimi, Christoph Burgmair, Andreas Finkenzeller, Christoph Bachhuber, and Eckehard Steinbach. 2018. Delay compensation for a telepresence System with 3D 360 degree vision based on deep head motion prediction and dynamic FoV adaptation. *IEEE Robotics and Automation Letters* 3, 4 (2018), 4343–4350. https://doi.org/10.1109/lra.2018.2864359

[5] Ou Bai, Varun Rathi, Peter Lin, Dandan Huang, Harsha Battapady, Ding-Yu Fei, Logan Schneider, Elise Houdayer, Xuedong Chen, and Mark Hallett. 2011. Prediction of human voluntary movement before it occurs. *Clinical Neurophysiology* 122, 2 (2011), 364–372. https://doi.org/10.1016/j.clinph.2010.07.010

[6] Yanan Bao, Huasen Wu, Tianxiao Zhang, Albara Ah Ramli, and Xin Liu. 2016. Shooting a moving target: Motion-prediction-based transmission for 360-degree videos. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 1161–1170. https://doi.org/10.1109/bigdata.2016.7840720

[7] Yanan Bao, Tianxiao Zhang, Amit Pande, Huasen Wu, and Xin Liu. 2017. Motion-Prediction-Based Multicast for 360-Degree Video Transmissions. In *2017 14th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 1–9. https://doi.org/10.1109/sahcn.2017.7964928

[8] Yair Barniv, Mario Aguilar, and Erion Hasanbelliu. 2005. Using EMG to anticipate head motion for virtual-environment applications. *IEEE Transactions on Biomedical Engineering* 52, 6 (2005), 1078–1093. https://doi.org/10.1109/tbme.2005.848378

[9] FFmpeg. 2019. H.264 Video Encoding Guide. https://trac.ffmpeg.org/wiki/Encode/H.264. Online; accessed: 2020-03-26.

[10] C. Holmberg, S. Hakansson, and G. Eriksson. 2015. *Web real-time communication use cases and requirements*. RFC 7478. https://doi.org/10.17487/rfc7478

[11] Rob J Hyndman and George Athanasopoulos. 2018. *Forecasting: principles and practice*. OTexts.

[12] Kishor Koirala, Meera Dasog, Pu Liu, and Edward A Clancy. 2015. Using the electromyogram to anticipate torques about the elbow. *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 23, 3 (2015), 396–402. https://doi.org/10.1109/tnsre.2014.2331686

[13] Steve LaValle and Peter Giokaris. 2015. Perception based predictive tracking for head mounted displays. US Patent No. 9348410B2, Filed May 22, 2014, Issued Jun. 6., 2015.

[14] Peter Lincoln, Alex Blate, Montek Singh, Turner Whitted, Andrei State, Anselmo Lastra, and Henry Fuchs. 2016. From motion to photons in 80 microseconds: Towards minimal latency for virtual and augmented reality. *IEEE transactions on visualization and computer graphics* 22, 4 (2016), 1367–1376. https://doi.org/10.1109/tvcg.2016.2518038

[15] Simone Mangiante, Guenter Klas, Amit Navon, Zhuang GuanHua, Ju Ran, and Marco Dias Silva. 2017. VR is on the edge: How to deliver 360 videos in mobile networks. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network*. ACM, 30–35. https://doi.org/10.1145/3097895.3097901

[16] NVIDIA. 2019. NVIDIA CloudXR Delivers Low-Latency AR/VR Streaming Over 5G Networks to Any Device. https://blogs.nvidia.com/blog/2019/10/22/nvidia-cloudxr. Online; accessed: 2020-03-26.

[17] Stefano Petrangeli, Gwendal Simon, Haoliang Wang, and Vishy Swaminathan. 2019. Dynamic Adaptive Streaming for Augmented Reality Applications. In *2019 IEEE International Symposium on Multimedia (ISM)*. IEEE, 56–567. https://doi.org/10.1109/ism46123.2019.00017

[18] Feng Qian, Bo Han, Jarrell Pair, and Vijay Gopalakrishnan. 2019. Toward practical volumetric video streaming on commodity smartphones. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. ACM, 135–140. https://doi.org/10.1145/3301293.3302358

[19] James Robinson and Cameron McCormack. 2015. *Timing control for script-based animations*. W3C Working Draft. https://www.w3.org/TR/2015/NOTE-animation-timing-20150922

[20] Yago Sanchez, Gurdeep Singh Bhullar, Robert Skupin, Cornelius Hellge, and Thomas Schierl. 2019. Delay impact on MPEG OMAFâĂŹs tile-based viewport-dependent 360Âř video streaming. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2019). https://doi.org/10.1109/jetcas.2019.2899516

[21] O Schreer, I Feldmann, P Kauff, P Eisert, D Tatzelt, C Hellge, K Müller, T Ebner, and S Bliedung. 2019. Lessons learnt during one year of commercial volumetric video production. In *2019 IBC conference*. IBC.

[22] Sebastian Schwarz, Marius Preda, Vittorio Baroncini, Madhukar Budagavi, Pablo Cesar, Philip A Chou, Robert A Cohen, Maja Krivokuća, Sébastien Lasserre, Zhu Li, et al. 2018. Emerging MPEG standards for point cloud compression. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 1 (2018), 133–148. https://doi.org/10.1109/dcc.2016.91

[23] Skipper Seabold and Josef Perktold. 2010. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*. inproceedings.

[24] Andrew Segall, Vittorio Baroncini, Jill Boyce, Jianle Chen, and Teruhiko Suzuki. 2017. Joint call for proposals on video compression with capability beyond HEVC. In *JVET-H1002*.

[25] Shu Shi, Varun Gupta, Michael Hwang, and Rittwik Jana. 2019. Mobile VR on edge cloud: a latency-driven design. In *Proceedings of the 10th ACM Multimedia Systems Conference*. ACM, 222–231. https://doi.org/10.1145/3304109.3306217

[26] Shu Shi and Cheng-Hsin Hsu. 2015. A survey of interactive remote rendering systems. *Comput. Surveys* 47, 4 (May 2015), 1–29. https://doi.org/10.1145/2719921

[27] Ken Shoemake. 1985. Animating rotation with quaternion curves. In *ACM SIGGRAPH computer graphics*, Vol. 19. ACM, 245–254. https://doi.org/10.1145/325165.325242

[28] Twitch. 2018. Using Netflix machine learning to analyze Twitch stream picture quality. https://streamquality.report/docs/report.html. Online; accessed: 2020-03-26.

[29] Jeroen van der Hooft, Tim Wauters, Filip De Turck, Christian Timmerer, and Hermann Hellwagner. 2019. Towards 6DoF HTTP adaptive streaming through point cloud compression. In *Proceedings of the 27th ACM International Conference on Multimedia*. 2405–2413. https://doi.org/10.1145/3343031.3350917

[30] Daniel Wagner. 2018. Motion-to-photon latency in mobile AR and VR. https://daqri.com/blog/motion-to-photon-latency. Online; accessed: 2020-03-26.