Freedom: Fast Recovery Enhanced VR Delivery Over Mobile Networks

Shu Shi, Varun Gupta, Rittwik Jana
AT&T Labs Research
Bedminster, NJ, USA
{shushi,vgupta,rjana}@research.att.com

ABSTRACT

In this paper we design and implement Freedom, a mobile VR system that deliver high quality VR content on today's mobile devices using 4G/LTE cellular networks. Compared to existing state-of-the-art, Freedom does not rely on any video frame pre-rendering or viewpoint prediction. We send a latency-adaptive VAM frame that contains pixels around the FoV. This allows the clients to render locally at a high refresh rate of 60 Hz to accommodate and compensate for the user's head movements before the next server update arrives. We demonstrate that Freedom is the first system in the world that can support dynamic and live 8K resolution VR content, while adapting to the real-world latency variations experienced in cellular networks. Compared to streaming the whole 360° panoramic VR content, we show that Freedom achieves up to 80% bandwidth savings. Finally, we provide detailed end to end latency measurements of actual VR systems by running extensive experiments in a private LTE testbed using a Mobile Edge Cloud (MEC).

CCS CONCEPTS

• **Networks** → *Cloud computing*; *Mobile networks*; • **Human-centered computing** → *Ubiquitous and mobile computing systems and tools.*

KEYWORDS

Mobile VR, Remote Rendering, Motion-to-Update Latency, Mobile Edge Cloud, 360 Video

ACM Reference Format:
Shu Shi, Varun Gupta, Rittwik Jana. 2019. Freedom: Fast Recovery Enhanced VR Delivery Over Mobile Networks. In *The 17th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '19), June 17–21, 2019, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307334.3326087

1 INTRODUCTION

The popularity of head mounted virtual reality (VR) headsets in recent years has attracted a lot of enthusiasm from both academia and industry. Some report [34] predicts that VR is about to become mainstream and could surpass $ 40 billion market by 2020. Existing systems fall into two categories: *tethered* VR or *untethered* VR. With current technology, the best quality VR systems such as Occulus Rift [31] and HTC Vive [42] rely on a HDMI cable to stream high resolution video from the PC (equipped with high end GPU cards) to the headsets. On the other hand, mobile VR headsets like Google Daydream [14] and Samsung GearVR [43] perform all the graphics rendering on the mobile hardware chip within the headset. While these untethered VR systems can provide full mobility to users, they cannot match the high resolution quality of tethered VR systems. This performance gap can be difficult to fulfill even in the foreseeable future due to limitations of power consumption and computation on mobile devices [25].

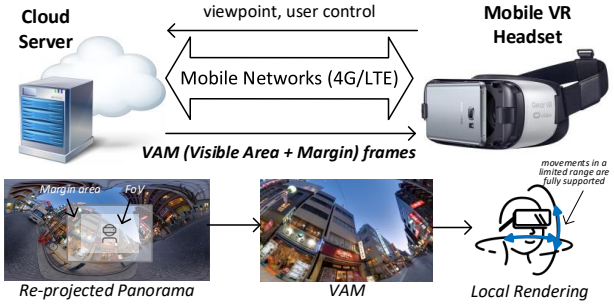


Figure 1: *Freedom* Overview

Recently, several solutions have been proposed to offload the rendering computation of mobile VR headsets to a remote server [1, 9, 25, 27, 40, 41]. However, these existing approaches either depend on the network to stream response visual frames to mobile devices within an extremely low latency for any user motion, or use various techniques of pre-fetching and viewpoint prediction to deliver correct frames to mobile devices in advance. As a result, existing systems usually deploy the rendering server in the same WLAN with mobile devices, consume tremendous amount of network bandwidth, and require the pre-processing the VR content or user viewing trace to achieve a good performance. How to push the rendering server to a remote cloud, support any VR content including gaming and live content that cannot be pre-processed, and stream to mobile VR headsets through cellular networks of today (i.e., 4G/LTE) remains an unsolved problem. In this paper, we propose a new approach: *Freedom*to address these challenges.

Freedom stands for **F**ast **RE**covery **E**nhanced VR **D**elivery **O**ver **M**obile networks. *Freedom* aims to support all types VR content available today, including pre-recorded and live 360° video, games and other graphics content that dynamically generate scenes based

MobiSys '19, June 17–21, 2019, Seoul, Republic of Korea

ACM ISBN 978-1-4503-6661-8/19/06…$15.00
https://doi.org/10.1145/3307334.3326087

on user input. As illustrated in Fig. 1, *Freedom* takes a server-client remote rendering framework but has a key difference compared to other thin-client systems [23, 27]. Instead of sending the rendering result images directly to the client for display, the server generates VAM (Visible Area plus Margin) frames at run-time. A VAM frame includes a portion of carefully selected pixels from the full panorama that support the client device to synthesize images for the viewpoint movement within a certain range. Using VAM frames enables the client device to perform graphics rendering locally and interpolate frames to directly accommodate any head motion at a 60 Hz refresh rate. Our system provides a guarantee of "fast recovery", which means the client will receive a new VAM frame before the viewpoint moves out of the coverage area of an old VAM frame. The combination of VAM and fast recovery brings us two major benefits: (i) *Freedom* can deploy the rendering server further away from the client to a remote cloud and adjust the margin size in the VAM frame to accommodate the actual network latency ; and (ii) *Freedom* acts similarly to Furion [25] but does not need to send full 360° panorama or waste network bandwidth on pre-fetching unused frames.

We have developed a prototype of the *Freedom* system and deployed on an actual MEC (Mobile Edge Cloud) test bed which connects to the mobile client through 4G/LTE radio. We present different design trade offs and system optimization we make to reduce the end to end latency close to 100 ms. Our evaluation results demonstrate that *Freedom* can stream up to 8K (7680x3840) resolution and 30 frame per second live VR content to mobile headsets over current cellular networks. Compared to streaming the whole 360° panoramic content, streaming VAM frames using *Freedom* achieves up to 80% bandwidth savings. We also introduce a concept of using auxiliary frames to boost the efficiency of real-time video encoder by 15%.

The main contributions of our work are as follows: (i) We propose a novel design of *Freedom*. Its components including real-time VAM frame generation, margin size adaptation, auxiliary frame selection, as well as latency optimization techniques can benefit the design and implementation of similar systems; (ii) we successfully integrate and optimize all system components to achieve state-of-the-art performance; and (iii) we evaluate our system in a MEC test bed accessed via 4G/LTE and quantify end-to-end latency of actual VR systems. We believe our latency breakdown surpasses the related work and provides a more realistic view of what the latency numbers are in real-world scenarios. To the best of our knowledge, we are the first to stream 8K live VR content over mobile networks and systematically evaluate an end-to-end mobile VR system using MEC and LTE.

2 BACKGROUND AND MOTIVATION

In this section we take a close look at the challenges of applying remote rendering to mobile VR headsets and identify remaining problems of existing solutions. To clarify, the mobile VR headsets we refer to in this paper usually use a smartphone with a wearable headset accessory (e.g., Google DayDream, Samsung GearVR, etc.). The smartphone serves as both computing and display unit. The system uses the gyroscope sensors to detect user's head movements to update VR rendering accordingly. There is usually a hand-held remote controller accessory available for the user to send other control/interaction signals to move an avatar, or shoot a target.

2.1 Latency and Refresh Rate

Latency is the most challenging requirement in designing a remote rendering VR system. The research [3] shows that the head mounted display is in particular sensitive to any head movement and an adequate user experience requires the *motion-to-photon* latency no larger than 25 ms. We use the term ***motion-to-photon* latency** to refer to the time it takes for a VR headset to respond to a motion and display the response frame.

This requirement has different interpretations in different systems. We show an illustration in Fig. 2. For a representative thin-client system [27] (Fig. 2(a)), the client triggers a rendering request and the server replies with the rendering frame to display directly. In order to meet the latency requirement, the client aims to refresh at 60 Hz, or display a new frame every 16.6 ms, which means the average *motion-to-photon* latency is no more than 25 ms. However, this requires the system to complete all the tasks within 16.6 ms, which include sending the trigger, rendering the scene, streaming the data back, decoding the video, and displaying the frame. Such systems cannot deploy to the cloud where the network Round Trip Time (RTT) between server and client alone is larger than the required 16.6 ms.

Furion [25] (Fig. 2(b)) takes a different approach. The server pre-renders the virtual world from all possible viewpoint positions and generates a 360° panorama for each. The client pre-fetches the panoramas rendered at the target and adjacent viewpoint positions in a batch and performs rendering locally. For the head motion that only changes the viewpoint orientation, the client can simply reuse the existing panorama without requesting further updates. As a result, it makes the *motion-to-photon* latency for any head movement motion independent from the network RTT and meets the requirement as long as the client rendering refreshes no less than 60 Hz.

However, Furion is limited by its own design in several ways: (i) the server has to pre-render the virtual world in advance to support batch pre-fetching. This restriction makes Furion only applicable to a subset of VR applications, such as pre-recorded 360° video, and simple games with a static virtual world. It cannot support live content or dynamically changing scenes (e.g., games with dynamic world or animation in the background scene, live 360° video, or volumetric video a.k.a. free viewpoint video). (ii) The local rendering on the client side only accommodates the head movement motion that changes viewpoint orientation. For other user motion that changes the viewpoint location (for example in Fig. 2(b), the user may operate the remote controller to move the avatar to its right), the client needs to use the pre-fetched panorama to support rendering and trigger a new request immediately to pre-fetch more data. Now the network factors such as RTT and bandwidth play a role again because the application can freeze if the avatar moves too fast to a new location before the pre-fetch frame arrives. (iii) Furion wastes too much network bandwidth pre-fetching panoramas that are not used at all. The network usage increases even more if the system attempts to pre-fetch more frames or support more dimensions of freedom of avatar movements.

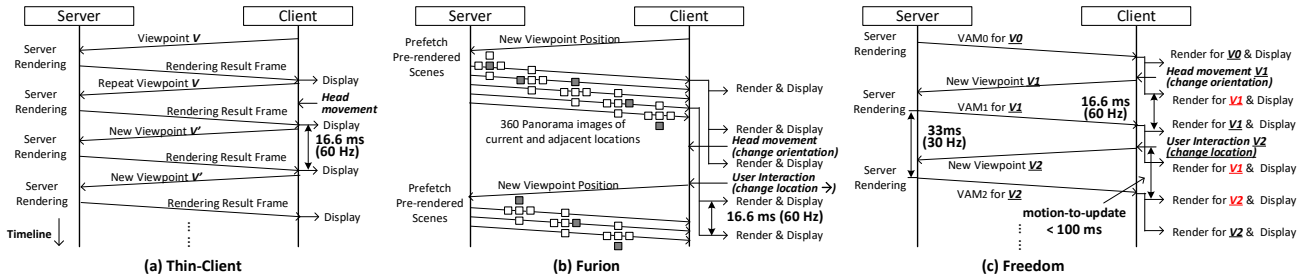


Figure 2: Illustration of system operation and latency: (a) a thin-client system [27], (b) Furion [25], (c) *Freedom*

Our Approach: Mobile VR systems have different latency requirements for different types of user interactions. Due to the nature of wearable devices, VR headsets are more sensitive to any head movements. Note that in contrast to what was previously discussed, the 25 ms *motion-to-photon* latency constraint does not always apply to user interactions made through the remote controller. In fact, there is broad consensus in the community of cloud gaming that users can tolerate delays up to 100ms or even higher [7, 10, 24, 26] (depending on the content and context). We use a hybrid approach to address this challenge. For more latency sensitive head motion that only changes the viewpoint orientation, *Freedom* relies on the mobile client to perform rendering locally at 60 Hz to generate the display frames. Unlike Furion, *Freedom* does not send full panorama to the client. Instead, the server generates a partial panorama customized for a specific viewpoint, which we refer to as VAM (more detalis in §4). For other user interactions through the remote controller, the user waits for a full *motion-to-update* latency to see the response. Here we use the term ***motion-to-update* latency** to refer to the entire time it takes to send a request to the server, generate the response frames, transmit them back to the headset, and finally displayed on the screen. The user experience for those interactions will not be impaired if the system manages to reduce the *motion-to-update* latency to less than 100 ms.

Fig.2(c) shows an example. In the beginning, the server generates a VAM_0 frame customized for the initial viewpoint V_0, and the client performs local rendering using VAM_0 for the viewpoint V_0. Then the user moves the head and changes the viewpoint orientation to V_1. The client immediately sends a request with the viewpoint change to the server. However, before receiving the update VAM_1, the client can continue the local rendering using the existing VAM_0 for the new viewpoint V_1. Note that even though VAM_0 is created for V_0, it contains sufficient pixels to support the client rendering for a new viewpoint V_1 that only changes the orientation. Therefore, the *motion-to-photon* latency for this head movement is less than 16.6 ms since the client rendering has a refresh rate of 60 Hz. Now the user triggers another movement using the remote controller and changes the viewpoint location to V_2. Note that the client is not able to use any existing VAM frames to render for the new viewpoint. Therefore, it continues the local rendering at the previous viewpoint V_1 and only switches to the new viewpoint V_2 after the new VAM_2 frame arrives, which happens after a *motion-to-update* latency.

Our approach supports the content with dynamically changing scenes such as 360° video, volumetric video, and the games with constantly changing background objects. The example above

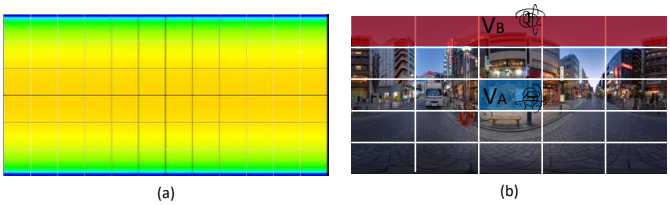


Figure 3: Inconsistent pixel density for equirectangular projection: (a) pixel density heat map; (b) tiles required to cover different viewpoints.

(Fig.2(c)) indicates that the server performs rendering upon the request from the client. However, depending on the content, the server can periodically trigger the rendering to accommodate the source frame rate even when there is no user interaction. For example, when playing a 360° video at 30 fps, the server needs to generate 30 VAM frames per second. The server refresh rate does not affect the refresh rate on the client side. The client always uses the last received VAM frame for local rendering.

2.2 Projection and Representation

A major challenge is determining how does the server prepare and pack the content for the client. For thin-client systems, the server simply extracts the rendering result as image frames. For systems that enable client to perform rendering and accommodate orientation change locally, the server needs to provide an environment map of the scene. The most common formats are equirectangular panorama and cube map [16]. Recent research has proposed different projections, including barrel projection [15] and equi-angular cubemap [11], to improve the pixel efficiency of the conventional formats.

In our design, once the client changes the viewpoint, the new server update that is created for the new viewpoint will arrive after a *motion-to-update* latency. Therefore, the server update may not need to contain the full panorama, but just enough pixels to support the client to perform local rendering for i) the current viewpoint, and ii) the viewpoints that the user may change to within a *motion-to-update* latency. Considering there is a physical limit for the head movement of a human being, our system only sends a partial panorama if the *motion-to-update* latency is reduced to 100 ms, which is the goal to satisfy all other non-head-movement user interactions. However, extracting the appropriate portion of the panorama and packing the pixels to a fixed size rectangular frame is not a trivial problem.

The major difficulty comes from the inconsistent pixel density. For example, under the equirectangular projection, the spherical region near the equator area has a much higher pixel density than the spherical region near the polar area as seen in Fig. 3(a). Most existing 360° video applications take a simple tiling scheme, which divides the panorama into small tiles and only sends the tiles that are within or around the current viewpoint [6, 21, 33]. Fig. 3(b) explains why the tiling scheme does not work for *Freedom*. Assuming only one tile is needed to cover the viewpoint V_A, 5 tiles are needed if the viewpoint moves to V_B. The cloud gaming system Outatime [26] uses a clipped cube map but does not explain how to cope with the pixel density issue. Other methods including pyramid projection [16], POI360 [44] propose to convert the panorama to a different projection or format that takes more bit rates to represent the areas within the viewpoint.

Our Approach: We create the VAM frame by cropping the central region out of a re-projected equirectangular panorama. Compared to other methods, our approach is more suitable for margin adaptation and maintains high pixel density for any viewpoint. We elaborate our design in §4.

2.3 Low Latency Streaming

Streaming real-time VR data over mobile networks is challenging. Encoding the frame data with efficient video codecs saves bandwidth. However, for real-time scenarios, the encoder is unable to look ahead and analyze more frames to make the best coding decisions. In particular, not utilizing B frames (which are dependent on previous and subsequent frames) during encoding decreases the coding efficiency [38]. Moreover, measuring the actual end-to-end system over real-world mobile networks is challenging. Many existing studies do not measure the latency of actual mobile networks, but instead, emulate the mobile networks using Wi-Fi [33] or simulators [40], or simply calculate the network latency with the available bandwidth [25] or ping latency [24].

Our Approach: We propose a novel method of generating auxiliary frames on the server that enables the real-time video encoder to take advantage of B frames without adding extra latency (§4.3). More importantly, we deploy the *Freedom* system in a real MEC test bed and measure the actual end-to-end system latency.

3 *FREEDOM* OVERVIEW

We first present the system framework of *Freedom* in Fig. 4. The server can processes two types of VR content. If the content consists of 360° video, each 360° panoramic video frame is directly passed to the VAM generation module. However, if the content is comprised of games, volumetric video or other graphics content, the system first generates the 360° panoramic frame of the scene and then follow the same path as the 360° video. A real-time video encoder compresses every VAM frame. An auxiliary frame controller manages the generation and encoding of auxiliary frames. The client decodes the received VAM frame, and uses the decoded frame as texture for VR rendering. The client also monitors both the gyroscope and remote controller, analyzes the user's movement, and notifies the server whenever a change happens.

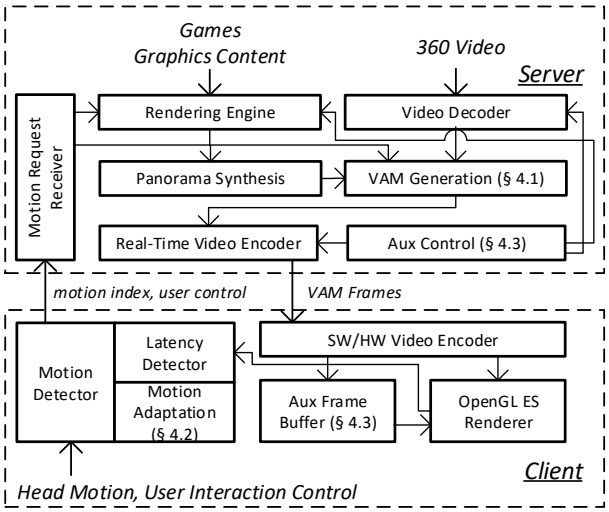


Figure 4: *Freedom* System Framework

We emphasize a few important aspects of *Freedom* design. First, the client VR rendering refreshes at 60Hz. In each rendering cycle, the program uses the latest gyroscope reading to determine the current rendering viewpoint. This makes sure the *motion-to-photon* latency for any head motion is no more than 16.6 ms in the worst case. However, this does not imply that the server should send the motion update 60 frames per second. On the contrary, the server only generates a VAM frame: (i) upon the client request, or (ii) when the source content is updated. In real systems, the server has a rate control to make sure it does not generate more than 30 VAM frames per second to save on network bandwidth usage. Note that our system can also support the 360° video or other VR content that updates at 60 fps. In that case, the client does not need to interpolate but perform rendering on actually received server frame every time. However, increasing frame rate does not decrease, but potentially increase the *motion-to-update* latency because the extra frames add more burden to both network and client devices.

Second, *Freedom* aims to minimize the *motion-to-update* latency to 100 ms, which means for any head movement or controller events, the client will receive, process and display the corresponding server update within 100 ms. The design of VAM and local rendering are used to improve the responsiveness of the system within 100 ms. It also means that even if local rendering fails to generate correct results, these errors will be ultimately fixed by the next VAM frame that comes within 100 ms. Note that the *motion-to-update* latency does not determine or affect the refresh rate of either server or client. It only influences the responsiveness of user experience.

4 SYSTEM DESIGN

4.1 VAM Generation

The VAM generation process takes three steps (Fig. 5). In the first step, the module reprojects the input panorama for a given viewpoint. More specifically, given a user viewpoint V which is denoted with three parameters: (*pitch*, *yaw*, *roll*), we first map the pixels of the original frame to a sphere, apply the reverse rotation matrix constructed by (*pitch*, *yaw*, *roll*), and create a new equirectangular

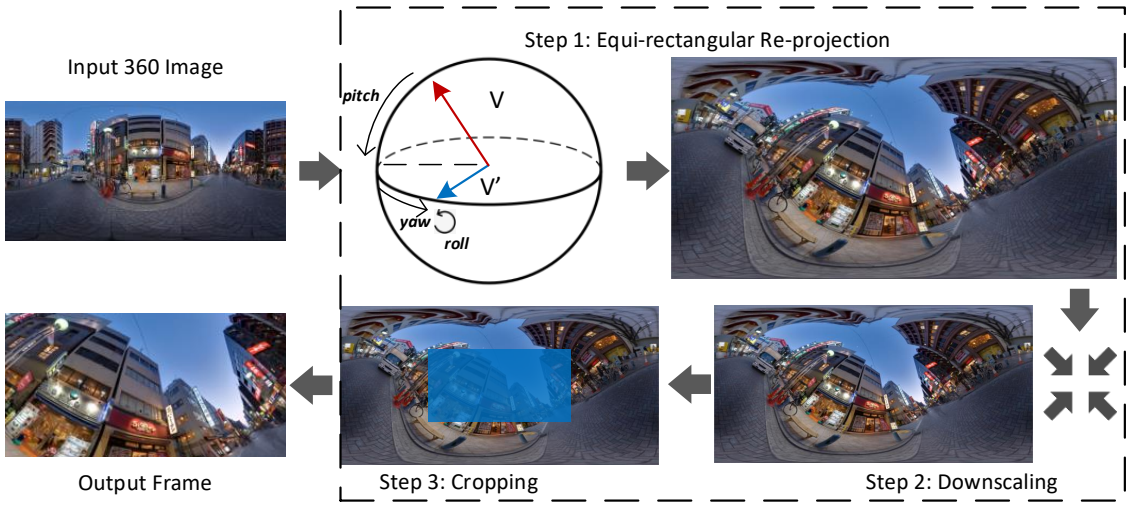


Figure 5: VAM generation procedure

projection of the rotated sphere. Note that the original frame does not have to be an equirectangular projection. In the second step, the filter downscales the reprojected frame according to a *scale* parameter. Finally, the filter crops the center region of the downscaled frame as the output frame. The cropping size is determined during system initialization and remains the same for all frames.

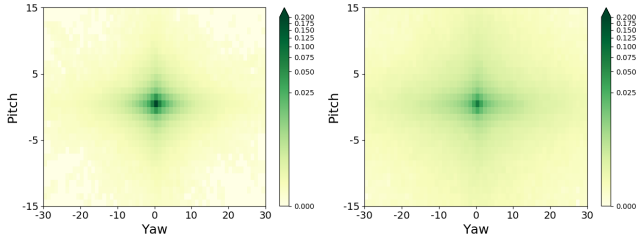
Scale and Zooming: VAM adds a *scaling* feature to optimize streaming ultra high quality VR content to small screen devices. For example, if *Freedom* plays an 8K 360° video, simply cropping the frame at original resolution may generate a VAM frame at 4K resolution or higher. However, due to small screen sizes, current mobile VR headsets may not be able to display 4K video at full resolution. Therefore, downscaling the 8K video first and generating a VAM frame at 1440p or 1080p will have no influence on the final visual quality but save tremendously in computation and network bandwidth. Therefore, *Freedom* utilizes a *scale* parameter to control VAM generation. A $scale = 1$ means no downscaling and a $scale = 0.5$ means the module will downscale the frame to half in width and height before cropping. Note that the scaling operation in VAM generation does not change the frame aspect ratio.

This scaling feature also allows *Freedom* to support zooming easily. Assume *Freedom* is streaming a 8K 360° video, the initial *scale* value is 0.5, and the VAM frame resolution is only 1080p after cropping. If the the user intends to zoom in and see more details of a video object, *Freedom* can correlate the zoom in gesture to the value of *scale*. By increasing *scale* until it reaches 1, *Freedom* puts more pixels within the area of interest in the VAM frame without changing the frame resolution. As a result, *Freedom* allows the user to view all the video details in the original 8K pixel quality but only uses the bandwidth to stream a 1080p video. In theory, *scaling* feature enables *Freedom* to support the content of any high resolution as long as the server hardware has enough processing power.

Motion Index: The generation of each VAM frame is controlled by four parameters: *pitch*, *yaw*, *roll*, and *scale* and all four parameters are critical for this VAM frame to be correctly rendered on the client device. In *Freedom* , we quantize each parameter to an 8-bit integer and pack all four into a 32-bit integer, which we refer to as **motion index**. Each VAM frame has a motion index number and it travels together with the frame packet. In our implementation, on the server side, we modify the the video encoder to append the motion index to the video packet of its corresponding VAM frame after encoding. On the client side, we also need to modify the video player to extract the motion index before passing the video packet to decoder. In this way, the motion index is always in sync with its corresponding VAM frame. An alternative solution is to utilize color codes to embed the meta-data information within image frames as proposed by Facebook [15].

Motion index also allows *Freedom* to accurately measure and breakdown the *motion-to-update* latency. All four parameters are controlled by the client. The viewpoint parameters (*pitch*, *yaw*, and *roll*) are obtained from the gyroscope sensor, and the *scale* parameter is correlated with the zooming gesture. Therefore, the client generates a new motion index number and notifies the server. The server simply forwards the received motion index to VAM generation and sends the same number back to the client with a VAM frame. This allows us to track the life cycle of a motion index and accurately measure the latency of every system component.

The VAM generation technique described above has several advantages. For any viewpoint, it guarantees that all output frames of the same resolution have the same margin size and same pixel density. Compared to other approaches such as the clipped cube map [26], VAM is more convenient in packing the full rectangle frame with effective pixels and converting between angular margin degree and frame resolution size. However, it is computationally expensive to perform reprojection and scaling for every frame and can add extra latency to the overall system. We discuss more details for accelarating VAM computations using GPUs in §5. Note



(a) (b)

Figure 6: (a) The heatmap of user head movement for 100ms, (b) heatmap for user head movement for 300ms

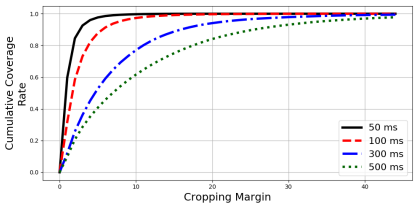


Figure 7: Coverage rate vs. margins [39]

that a server can perform simplified VAM generation by keeping only the last cropping step when GPU acceleration is not available. In this case, the system is limited and only allows the viewpoint orientation to change horizontally along one dimension.

4.2 Margin Adaptation

Margin size adaptation is a key feature that differentiates *Freedom* from previous work. The main drawback of not sending the full 360° panorama is that the client cannot render correctly if the user's viewpoint moves out of the area covered by the partial frame. To address this problem, existing systems choose to improve the accuracy of viewpoint prediction or send the entire 360° frame as a base layer in low resolution [21, 33, 44]. *Freedom* takes a different approach by dynamically setting the margin size just large enough to cover any motion within the actual *motion-to-update* latency. Therefore, *Freedom* not rely on viewpoint prediction or base layer as backup.

We first quantitatively analyze how to select the appropriate margin size for various *motion-to-update* latency values and then elaborate how *Freedom* performs margin size adaptation at run time. Our analysis is based on the head movement trace dataset [12] that consists of 52 users watching six different 360° videos. For the sake of simplicity, we ignore the *roll* parameter of the user viewpoint and use spherical coordinates for the discussion.

We first plot the heat map of change in user head orientations within a period of time (see Fig. 6). We note that most of the head movement is either across the horizontal (*yaw*) and vertical (*pitch*) axis and very few head movements are diagonal. Further, we define the term *coverage rate* as the probability of head movement within a specified region over a given time. For example, the margin region in Fig. 6 is ±30° yaw and ±15° pitch. By adding the value of all pixels in the region, we obtain coverage rates of 99% and 96% for 100 ms and 300 ms, respectively. By searching over all possible region sizes, Fig. 7 shows the CDF of *coverage rate* vs. the margin size for different latency numbers. We observe that 99% coverage rate can be achieved by a margin of 35° for 300ms latency, whereas, only 15° margin is required when the latency is 100ms. We use Fig. 7 to query the appropriate margin size for a given latency value and a desired coverage rate. Note that Fig. 7 was first published in our previous work [39].

At run-time, the latency detector module measures the actual *motion-to-update* latency of the system by monitoring the motion index number. With the accurate latency data, the margin adaptation module queries the mapping table (Fig. 7) and calculates the appropriate margin size. It then adjusts the actual margin size by changing the *scale* parameter. Increasing the value of *scale* reduces the margin size and reducing the *scale* value increases the margin size. Note that the margin adaptation module does not react to the latency increase caused by a sudden jitter. It also avoids making abrupt changes to the margin size.

4.3 Auxiliary Frame Selection

In this subsection, we describe the application of auxiliary frames to improve the efficiency of real-time video coding. Fig.8(a) shows an illustration of using a conventional real-time video coder to encode VAM frames. Each frame is encoded as either I or P frame. All P frames use the previous frame as the reference frame. Thus, failing to deliver a single P frame will lead to a situation where subsequent P frames cannot be decoded. Therefore, *Freedom* system uses TCP based streaming protocol to transmit all frames.

The above problem can be alleviated if the real-time coder can retrieve some future frames in advance. As shown in Fig.8(b), after encoding the first frame as I frame, the encoder can immediately encode a P frame with the original 4th frame, which we refer to as the auxiliary frame. This allows the second and third frames to be encoded as B frames. On the client side, the auxiliary frame is received together with the first I frame. The B frames have their reference frames ready when they arrive and thus do not add extra delay for decoding. The client needs a buffer to hold the auxiliary frame until its target appearance. More importantly, the B frames are not used as reference. In the case a packet of the frame is lost, corrupted, or delayed, the client can simply discard the frame without affecting the decoding process of future frames.

Retrieving future frames is feasible in VR applications. For 360° video, the *Freedom* system simply fetches the future frames from the video data. An auxiliary frame can be generated for live 360° videos as well since live streaming services deliver videos in small chunks. For content such as VR games, the game engine needs to be modified to generate the future frames but these modifications are application specific [26, 38].

It is important to point out that each VAM frame is generated for a specific viewpoint known to the server as the current rendering viewpoint on the client. Since the auxiliary frame is encoded a few frames before its actual appearance, the viewpoint used on the server may change during the period. Fig. 8(c) shows an example. The server chooses to use the same viewpoint V_1 that is used for the first I frame to generate the first auxiliary frame. However, when the user starts to move her head and look to the right, this results in different viewpoints V_2 and V_3 for subsequent B frames. At the time when the auxiliary frame should appear, the viewpoint has

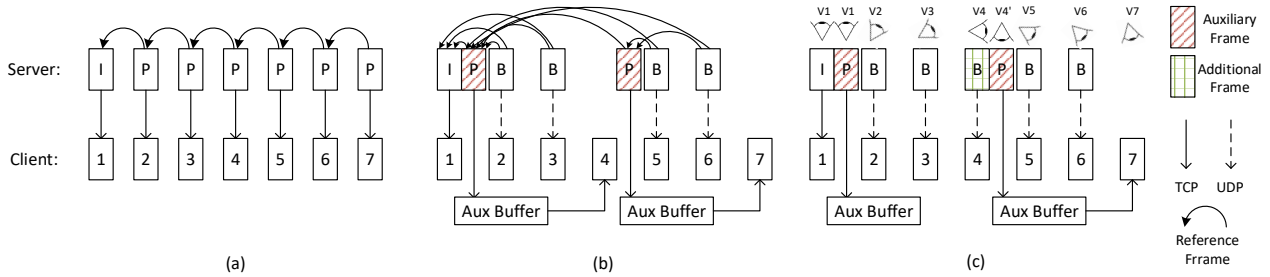


Figure 8: Auxiliary frame selection illustration: (a) Real-time video coding without auxiliary frame; (b) using auxiliary frame to enable B frame coding and hybrid streaming; (c) viewpoint prediction for auxiliary frames

been changed to V_4. In this case, the auxiliary frame generated for V_1 cannot provide enough pixels to support the correct rendering at V_4. Therefore, the server needs to generate an additional frame by using the original panorama and create another VAM for V_4. Note that the additional frame can also be encoded as B frame and transmitted through UDP.

To resolve the above issue, we leverage a user's head motion prediction when generating auxiliary frames (i.e., predicting the future yaw, pitch, roll values). In the same example as above, when the server generates the second auxiliary frame, it does not directly use the currently available viewpoint V_4, but instead predicts what the viewpoint will be when the auxiliary frame is displayed, in this case, V_4'. Assuming the user keeps moving the head and changes the viewpoint to V_7 when the second auxiliary frame is to appear, the predicted viewpoint V_4' is close to the actual viewpoint V_7. Thus the server does not need to generate an additional frame in this case.

Note that the motion prediction described above is different from the viewpoint prediction discussed in other 360° video streaming techniques [33, 44]. In the latter techniques, prediction is usually trained by the user data to predict which part of the video is more likely to be watched by the user at a future time. By contrast, the motion prediction in *Freedom* simply predicts the direction of user motion within a very short period of time. *Freedom* generates no more than 30 VAM frames per second and forces the encoding such that only two B frames are generated between subsequent auxiliary frames. Thus, the window for motion prediction is only 99 ms. In Table 1 we evaluate the impact of prediction on the success rate. The success rate is defined as the percentage of the auxiliary frame that does not need an additional frame to correct error. We consider two techniques for prediction: (i) no prediction, i.e., use the current viewpoint, and (ii) speed-based prediction which calculates the future viewpoint position based on the exponential moving average of speed of head movement. The results indicate that the speed-based prediction yields 2% higher coverage rate than no prediction for small margin size.

Table 1: Success rate of auxiliary frame viewpoint prediction

Prediction Scheme	Margin Size Yaw/Pitch 15/8	30/15	60/30
No Prediction	0.872	0.973	0.997
Speed-based prediction	0.893	0.991	0.996

5 SYSTEM OPTIMIZATION

In this section, we share the experience and lessons learned from optimizing the *motion-to-update* latency of *Freedom*. The goal is to make to the end-to-end latency less than 100 ms. Even though this constraint is relaxed compared to the 25 ms requirement for head movement motion, it remains a challenging task considering that the server and client connect through a 4G/LTE mobile network.

Server Placement: In current LTE networks, a mobile device may exploit compute and storage resources of powerful distant centralized clouds, accessible through an Enhanced Packet Core (EPC) network situated inside a mobile operator and the Internet. Accessing data from a distant cloud introduces long delays since data has to be sent back and forth from servers that are far away from the end users. To reduce this long latency, a new paradigm has recently emerged, namely mobile edge cloud (MEC) whereby compute and network resources are situated very close to the end users. To this end, we have built a MEC that is collocated with a private LTE cell site. The *Freedom* server is situated in the MEC and the *Freedom* client receives LTE packets routed to it. The location of the MEC collocated with the LTE site gives it a unique advantage over cloud servers situated in other distant locations.

GPU vs CPU: The latency on the server side is mainly composed of the heavy computation for VAM generation and real-time video encoding. The VAM generation includes reprojecting and scaling, both being very expensive operations. However, these operations are a great fit for NVidia GPU CUDA acceleration since the same operations apply to every pixel in the frame and there is no dependency between pixels. We have implemented the procedure for both CPU and CUDA execution and compare the results in Section 6.

GPU also includes the hardware video codec to help accelerate the real-time video encoding and reduces the encoding latency significantly. For example, the NVidia GPU card has CUVID for decoding and NVENC for encoding acceleration [30]. We also note that moving the video frame data in and out of GPU memory takes a big hit in latency performance. Thus, it is important to keep all the operations including VAM generation and real-time encoding in GPU for best performance.

Rate Control: The *Freedom* server has a rate control mechanism to make sure that it does not generate more than 30 VAM frames per second. The server sets the pace to generate a VAM frame every 33 ms. When the client sends a request to change the viewpoint, it needs to wait on average 16 ms until the next motion update frame is processed. The server may choose to either reprocess the existing

motion update frame that has already been generated but not yet sent out, or start to process the next motion update frame earlier. This improved rate control mechanism is able to reduce the wait time to less than 5 ms.

FFMPEG Player: On the client side, we use a FFMPEG-based video player that supports both software and hardware codecs [8] and an open source VR player [5]. Compared to some existing approaches [23] which use a simple one thread player loop, FFMPEG player supports both MediaCodec (hardware codec) and AVCodec (software codec). This provides more robustness as well as flexibility to evaluate different streaming protocols and video codecs.

Besides the configurations recommended in [18], we had to make some other changes to the source code to reduce internal queuing delay. FFMPEG ensures that video reading, decoding and displaying are three separate threads to prevent I/O blocking and use system resources maximally. Internal queues are used when passing video data between threads. Increasing the queue size can potentially avoid packet loss and frame drop but it also increases the latency. We observed that the internal queue size of the video player may increase over time, mainly because of network jitter. To resolve this, we modified the rate control module to play the video at a faster rate whenever the internal queue size increases.

Streaming Protocols Latency is also a major concern in selecting streaming protocols. The widely adopted HTTP DASH/HLS [13] protocols for video streaming typically maintain a large buffer (holding multiple seconds worth of data) and are not suitable for *Freedom*. In the *Freedom* prototype, we use point to point streaming [18] with either TCP or UDP protocol. We also evaluated the RTMP [36] protocol designed for real time streaming and got comparable results. It is worth mentioning that the format of video packets also plays a role in latency optimization. Our experimental evaluations comparing FLV [19], MPEG-TS [28], and RTP [37] showed that FLV has the best latency performance. Code analysis shows that FFMPEG player does not release the video frame packed in MPEG-TS and RTP format until the first packet of the next frame arrives, which adds 33 ms to the overall *motion-to-update* latency.

Codecs: We evaluated the performance of both H.264 and HEVC video codecs. Even though HEVC has a higher compression ratio, it does not actually improve the end-to-end latency. On the contrary, the FLV packet format does not support HEVC and we have to pack HEVC frames with MPEG-TS format, which leads to higher latency. When enabling B frame encoding using auxiliary frames, MediaCodec on the client side automatically buffers more than 10 frames internally, which we have not yet found a method to eliminate. In this case, we rely on the software codec to decode the stream with auxiliary frames. Therefore, in our current prototype system, the set up with the lowest *motion-to-update* latency is achieved using H.264 codec without any auxiliary frames.

6 EVALUATION

6.1 Experiment Setup

We have implemented *Freedom* and deployed the prototype system on an experimental MEC with a private 4G/LTE connection.

On the server side, the virtual machines used to host *Freedom* server have eight CPU cores and 16GB memory each. The Nvidia GTX 1050 is used in the scenario when GPU is needed. In order to conduct

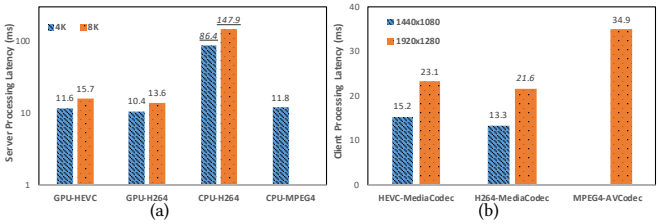


Figure 9: (a) The processing latency on the server, and (b) the processing latency on the client. Some cases cannot support video playback of 30 fps on the client side and the latency numbers of those cases are marked with underscore.

controlled and repeatable experiments with identical conditions, we have the *Freedom* server playing the 360° video files of both 4K (the same files as [12]) and 8K[1] resolution at 30 fps from disk. Even though pre-recorded files were used, we do not pre-process the video but treat them as live content that is retrieved in real-time. We configure the real-time video encoder on the server to encode one VAM frame every 33 ms.

On the client side, we run the client software on a Samsung Galaxy Note 8 phone, which also becomes a VR headset with the Samsung GearVR accessory. We use the Android testing framework Espresso to automatically reproduce the same device motions. We consider a FoV (Field of View) of 120° horizontal and 90° vertical. The margin size is initially set to 60° horizontal (30° in each direction) and 30° vertical (15° in each direction) and dynamically adjusted based on the measured latency using results in Section 4.2.

We list all experiment setups in Table 2.

6.2 Results Analysis

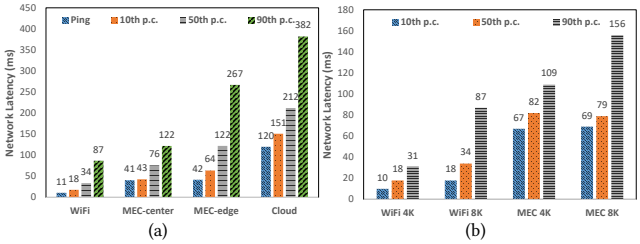
Server and Device Latency: Fig. 9(a) shows the average server processing latency. Our results show that GPU optimization plays the most important role. If the server does not have GPU to accelerate VAM generation and video encoding, it takes 86.4 ms and 147.9 ms on average to generate and encode one VAM frame will full CPU usage for all eight cores for 4K and 8K resolution, respectively. The only option to generate VAM frames at 30 fps is to disable reprojection and scaling steps and encode with MPEG-4, an old but lightweight codec to encode video frames.

Fig. 9(b) shows the processing latency on the client side, which only includes video decoding. Our mobile client is able to use hardware codec (Android MediaCodec) to decode H.264 and HEVC video streams. Since MediaCodec does not support MPEG-4 stream, we opted to use the software codec (AVCodec) to decode. The video player we use [8] calls both Android MediaCodec APIs and AVCodec library in native C code and maintains all the QoS control and sophisticated buffer management from FFMPEG/FFPlay. Note that the number listed is only the decoding latency for one frame. For the MPEG4-AVCodec scenario, the VAM frame is cropped from an original 4K video even though the frame resolution (1920$times$1280) is the same as other VAM frame that are downscaled from 8K videos. Note that the video player processes multiple frames in parallel so that it can achieve 30 fps playback even though the latency of decoding one frame exceeds 33 ms.

[1] YouTube video ID: FOZ5OAoI4Jo

Table 2: Evaluation Setups

Network Setup	**WiFi:** The *Freedom* client connects to the *Freedom* server in the same WLAN through WiFi (802.11ac). **MEC:** the *Freedom* client connects to the *Freedom* server deployed in MEC through our LTE test bed. **Cloud:** the *Freedom* client connects to the *Freedom* server deployed in the central cloud using the commercial LTE network of a major carrier. The cloud server is located in the nearest Microsoft Azure data center, which is approximately 400 kilometers (250 miles) away. The server does not have GPU.
Server Setup	**GPU:** the *Freedom* server uses an Nvidia GPU card to accelerate VAM generateion, video decoding and video encoding. **CPU:** the *Freedom* server runs everything on CPU only. **CPU/c:** *Freedom* server runs everything on CPU only. In particular, the VAM generation only crops the 360° image without reprojecting and scaling.
Device Setup	**MediaCodec:** the *Freedom* client uses Android MediaCodec to decode the received video frames. MediaCodec utilizes the hardware codec available to decode video data. **AVCodec:** the *Freedom* client uses AVCodec, a software codec to decode video frames.
Codec Setup	**HEVC:** HEVC/H.265 standard is used for encoding. **H264:** H.264/AVC standard is used for encoding. **MPEG-4:** MPEG-4 standard is used for encoding.
Auxiliary Frame	**w/ aux:** *Freedom* server generates auxiliary frames, encode 2 B frame for every auxiliary frame and transmit all B frames through UDP. This setting is only used in a few experiments. All other experiments without this setting explicitly marked only generates I/P frames and sends everything through TCP.
Packet Format	**mpegts:** *Freedom* server prepares the video packet in MPEG-TS format. This setting is only used in one experiment. All other experiments without this setting explicitly marked uses FLV as the packet format.
Resolution Setup	**4K:** The original 360° video has a resolution of 4K (3840×2160) and the resolution of the VAM frame is 1440×1080. The initial $scale$ parameter is set to 0.75. **8K:** The original 360° video has a resolution of 4K (7680×3840) and the resolution of the motion update frames is 1920×1280. The initial $scale$ parameter is set to 0.5. **4K/c:** The VAM frame is cropped from the original 4K frame and the cropping window is 1920×1440.



(a) (b)

Figure 10: (a) The network latency for different network setups, and (b) The network latency for different frame size

Network Latency: We highlight the network latency results for different setups in Fig. 10(a). The network latency only includes the network transmission time to send the viewpoint control to server and subsequently sending the VAM frame back to client. Compared with the latency of WiFi, LTE latency data has large variance. Therefore we show 10th and 90th percentile together with the median and ping value as comparison. Obviously, putting the server in the MEC performs much better than central cloud but the strength of LTE signals can also significantly affect the latency. We show two **MEC** setups: **MEC(center)** means the client device is located at the center of the LTE cell with strong signal strength with measured RSRP ≈-75dBm, and **MEC(edge)** means the client device is located at cell edge with weak RSRP signal strength of ≈ -110dBm.

Intuitively, the frame size should play an important role in transmission latency. However, we show in Fig. 10(b) that the frame size does affect the network latency for WiFi set up but does not show much difference when streaming over LTE. Our conjecture is that the LTE network latency is mainly affected by the scheduler and transmission initiation. Once the transmission starts, LTE has sufficient bandwidth and the difference in data size does not play an important role.

End-to-End Latency Breakdown: We break down the end-to-end motion-to-update latency in Table 3. For all seven setups listed in the table, the mobile client decodes and plays the video at an average of 30 fps (the setups that fail to stream at 30 fps are not listed) and runs VR rendering locally at a constant 60 fps regardless how big or small the motion-to-update latency is. We now provide some insights of our results.

First, HEVC is not the best fit for our system in the latency perspective because it requires more time to encode and decode but the saving on bandwidth does not help save in network transmission. On the contrary, because HEVC is incompatible with FLV format, we have to use MPEG-TS that results in worse network transsmion latency. Second, using auxiliry frames forces us to use AVCodec to decode the video because Android MediaCodec automatically adds more than 300 ms latency when decoding streams with B frames. Even though it achives more than 15% saving in bandwidth usage, it does not reflect in latency saving for LTE. Third, we do not see a big portion of the device latency in displaying the frame as presented in [24]. However, the time spent after the video decoding is not negligible. This is mainly because of queuing. Each frame needs to wait for the video player to play at 30 fps and then wait for the VR renderer to refresh at 60 Hz. Last, the motion-to-update latency does not count the time it takes to detect the motion event. Even though we set the gyroscope polling to 100 Hz, it may add another 5 ms on average to the existing results. According to [24], the latency of detecting control events from remote controller can be quite significant.

Bandwidth savings: We consider bandwidth savings as a byproduct of *Freedom* since we only send the VAM frame rather than the full 360° video to the client. Fig. 11(a) shows the throughput for streaming 4K and 8K 360° video with different setups. The client was located at the cell center with a strong signal strength of RSRP approximately −75dBm. For each case, the throughput values are normalized to the throughput required for streaming the whole 360° video. Compared with sending original full 360° video, *Freedom* saves up to 80% of bandwidth even over LTE networks. The results are consistent with what we expect: the latency determines the margin size. The lower latency results indicate more bandwidth savings. Fig. 11(b) shows that the margin size the system selects is

Table 3: Device Latency Breakdown

Test Setup	*motion-to-update* Total (ms)	Server Processing	Network Transmission	Video Decoding	Display Queue	Copy to Display	VR Rendering	Bandwidth (Kbps)
WiFi, GPU, 4K, H264	51.9	10.4	14.2	13.2	5.9	0.8	7.7	3744
WiFi, GPU, 4K, HEVC, mpegts	89.7	11.6	51.3	12.2	5.8	0.2	8.7	3456
WiFi, GPU, 8K, H264	80.7	13.6	25.8	27.8	4.5	0.8	7.2	7942
MEC, GPU, 4K, H264	118.5	10.2	82	11.7	5.7	0.9	8.0	3894
MEC, GPU, 8K, H264	129.9	14.1	84	17	5.9	0.7	8.2	8181
MEC, GPU, 8K, H264, w/aux	147.7	24.6	81	21.1	9.8	8.9	2.3	6778
Cloud, CPU/c, 4K/c, MPEG4	599.6	12	512	15.2	9.1	6.4	4.9	11466

based on the measured motion-to-update latency. If the motion-to-update latency remains high, such as the scenario which we use commercial LTE to stream from a central cloud, the bandwidth saving is very limited because a large margin area is needed. However, the bandwidth saving results may vary significantly depending on the FoV size, initial cropping size, and the *scale* value.

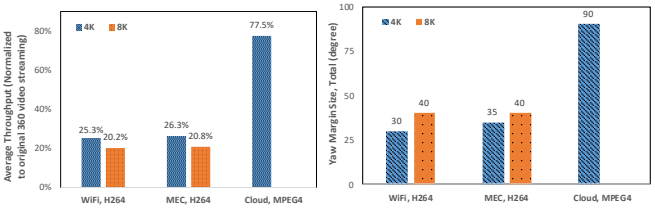


Figure 11: Impact of (a) throughput, and (b) average yaw margin size selected by *Freedom* for different setups

Visual Quality. Next, we evaluate the impact on visual quality of *Freedom* . In Figure 12, we show an example of how *Freedom* can support zooming to allow user to watch the VR content at full 8K resolution.

When the rendering viewpoint on the client side is well covered by the available VAM frame, the visual quality degradation mainly comes from the generation (mainly caused by downscaling) and real-time encoding of VAM frames. Considering the relationship between video rate and distortion has been thoroughly studied in the literature, we omit the analysis of the visual quality loss caused by video coding, but instead, focus on the visual artifact when the rendering viewpoint moves out of the coverage of the VAM frame.

Figure 13 shows two examples of typical artifacts (incorrect renderings) when the rendering viewpoint moves out of the VAM coverage. These artifacts only appear when users change viewpoint abruptly and will be fixed after the new VAM frame arrives within a motion-to-update latency. To quantitatively analyze the artifacts, we measure the size of artifacts (in degree) in every rendered frame when playing 360° video using the pre-recorded head movement trace [12]. For the MEC 4K scenario, this average horizontal artifact is roughly 4.9° which is 5% of the whole FoV width. For Cloud 4K, this is over 12°. The average vertical artifact is less than 2° in each case. Note these numbers are measured with the margin adaptation threshold set to 95%. Increasing the margin adaptation threshold to 99% may further reduce the artifacts but in our case will result in either bandwidth increase or quality degradation.

We identify the time periods when the user's headset does not render the desired FoV.

6.3 Comparison to Related Work

In this subsection, we try to compare the performance of *Freedom* to related work Furion [25]. Since Furion is not open source, we mainly compare the evaluation results reported in its paper and use a simplified test program in our testbed to simulate the performance of running those systems over mobile networks.

Furion does not evaluate an end-to-end system latency, but report a responsiveness delay of 1 ms for rotation (orientation change) and 12 ms for motion (location change). In comparison, our numbers for the WiFi-4K scenario in Table 3 are much higher. Our local rendering rendering time is 7.7 ms and overall *motion-to-update* latency is 51.9 ms. Since Furion does not describe in detail how responsiveness is measured, our conjecture is that the rotation latency does not include the average time spent on waiting for the next refresh cycle, which is approximately 8 ms for 60 Hz refresh rate. The 12 ms motion latency of Furion is mainly used in video frame decoding assuming the video frame has already been pre-fetched. But in our case, even if we remove the server processing delay and all queuing delay that a video player adds to achieve a smooth playback, the lowest latency we can achieve on our mobile device to transmit, decode, render, and display video frames is around 30 ms. It is worth pointing out that our latency is comparable to the results in [24].

The network bandwidth usage of Furion depends on the user behavior. If the user controlled avatar remains static in one spot, the system does not consume any network bandwidth. However, once the avatar starts moving, the system needs to constantly fetch new frames and uses on average 132Mbps. We have tried in our MEC testbed and the maximum bandwidth we can get is around 30 Mbps. Fetching three panoramic frames in a batch over 4G/LTE from MEC takes on average over 100 ms. As a result, the avatar has to move five times slower to avoid stall if we run Furion over mobile networks. As in comparison, *Freedom* only requires 3 Mbps to support the same 4K content. However, *Freedom* requires continuous streaming even if the avatar is not moving.

7 RELATED WORK

Mobile VR: MoVR [1] and TPCast [41] use high bandwidth mmWave technologies (e.g., 802.11ad/WiGig) to replace the HDMI cable. Liu et al. [27] adopt a simple thin-client framework and optimize the system latency to achieve a refresh frequency of 90Hz. LTE-VR [40] proposes improvements on current LTE signaling operations to reduce latency. All these approaches are heavily depend on the ultra low network latency between the server and the client to deliver a satisfying user experience.

Flashback [9] and Furion [25] add the support for viewpoint location change by pre-rendering the whole virtual world from every possible location. In particular, Furion separates the foreground

Figure 12: An illustration of how *Freedom* zooms to support full 8K quality

Figure 13: Typical visual artifacts when the rendering viewpoint moves out of the currently available VAM frame

objects from background scenes. By rendering the lightweight foreground objects on the client devices locally, Furion can support some game actions.

We published a preliminary work named as MEC-VR in [39], which shared similar design as *Freedom* . However, the previous work did not have all components of VAM generation. It supported only one dimension of head movement by cropping the central row of panorama.

Cloud gaming: Cloud gaming has always been the most challenging remote rendering system to design. Most cloud gaming systems [10, 23] take the thin-client approach. These systems are similar to the thin-client and casting VR systems discussed earlier in the paper. Outatime [26] is a state-of-the-art cloud gaming system which mixes speculation execution and image based rendering to hide high network latency. Our work draws some inspiration from Outatime but unlike Outatime focuses on 360 and VR content.

360 Video: 360 video has been used interchangeably with VR in many recent works. Most existing work in 360 video streaming focuses on bandwidth savings. This is usually achieved by predicting user's viewpoint and streaming the visible areas of video frames with higher priority. Existing approaches include: (i) sending only the pixels within and around the user's viewpoint through different tiling schemes [6, 21, 22, 32, 33, 35], or (ii) transforming the video frame to a different projection that has much higher pixel density in the visible area [15, 17], or (iii) encoding the visible area to a higher bit rate [44] or in a separate layer [21, 29]. However, the approaches that require user viewpoint prediction rely on collecting extensive historical training data and learning features of interest in the scene using machine learning algorithms. Additionally, these techniques are not suitable for all content (e.g., live content) and cannot guarantee good performance for every user. *Freedom* uses a similar concept of sending only the pixels within and around the visible area. However, *Freedom* aims to minimize the *motion-to-update* latency and does not rely on viewpoint prediction. Moreover, *Freedom* is designed to support live content and it does not require any pre-processing.

8 DISCUSSION

In this section we discuss the limitations of *Freedom* as well as some issues related to its practical use and deployment.

5G mmWave networks: The performance of *Freedom* is strongly dependent on the network quality. Even with the *Freedom* server deployed on a MEC, there are other factors such as background traffic, interference, mobility, etc. that can significantly affect the latency. Guaranteeing QoS at millisecond level using existing protocols and tools may not be always feasible. The upcoming mmWave radios in 5G networks are expected to provide bandwidths up to a 1 Gbps while further reducing the latency [2]. We hope the 5G networks will provide a low latency channel that is comparable to that of WLAN today. In future we would like to experiment with 5G links in a more realistic scenario using *Freedom*.

Network Jitter: Without a video buffer, low latency streaming is vulnerable to network jitter. Jitter is simply a name of the symptom and can have various causes, such as page fault on the rendering server, congestion in the network switch, or an inefficient LTE scheduler. In our test bed, after analyzing the network logs we observed that most large jitters are caused by TCP re-transmission for error packets. In some worst case scenarios, the TCP re-transmission results in hundreds of milliseconds jitter and causes visible artifacts on the client VR player. However, there are no existing tools or streaming protocols that can effectively address the jitter issue for the streaming applications that requires ultra low latency. In the future, we will explore this in more detail.

Service Deployment and Control: *Freedom* requires a dedicated GPU server for each client which limits its scalability. Currently, the *Freedom* server side application is deployed in a way that is consistent with modern production deployment processes and methods. The applications were run as Docker containers to leverage Docker's ease of deployment with fast startup times (equivalent to starting a process [20]) and minimal runtime performance overhead [4]. The controller for deploying these containers needs to be location and mobility aware to find the most suitable resources for a particular user and application. Deploying the container on different MEC resources on the fly to meet the user request for various VR applications requires further exploration.

9 CONCLUSIONS

We have presented the design and prototype implementation of *Freedom*, a novel mobile VR system that leverages mobile edge cloud and requires no pre-rendering or viewpoint prediction to render high quality VR content on devices using today's cellular LTE networks. We believe that *Freedom* is the first system in the world that can effectively stream live VR content of 8K resolution while achieving up to 80% of cellular bandwidth savings. We provide a thorough latency breakdown analysis for each system component and identify key components to optimize (e.g., reducing client buffers, accelerating frame processing at the server, etc.). However, there exists several limitations of *Freedom*. For example, eliminating the use of buffer makes the system vulnerable to network jitters. The requirement of GPU acceleration on the MEC servers for high resolution content makes the system harder to scale. In future, we will focus on improving the *Freedom* system to address these limitations and adapt to the next generation 5G networks.

REFERENCES

[1] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. 2017. Enabling High-Quality Untethered Virtual Reality.. In *NSDI*. 531–544.

[2] Abari, O. and Bharadia, D. and Duffield, A. and Katabi, D. 2017. Enabling High-Quality Untethered Virtual Reality. In *Proc. of USENIX NSDI'17*.

[3] Michael Abrash. 2014. What VR Could, Should, and Almost Certainly Will Be Within Two Years. (2014). http://media.steampowered.com/apps/abrashblog/AbrashDevDays2014.pdf

[4] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. 2015. Performance Evaluation of Microservices Architectures using Containers. (2015). https://arxiv.org/pdf/1511.02043.pdf

[5] Ashqal. 2016. MD360Player4Android. (2016). https://github.com/ashqal/MD360Player4Android

[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 *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 1161–1170.

[7] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. 2004. The effects of loss and latency on user performance in unreal tournament 2003®. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. ACM, 144–151.

[8] Bilibili. 2016. ijkplayer. (2016). https://github.com/Bilibili/ijkplayer

[9] Kevin Boos, David Chu, and Eduardo Cuervo. 2016. Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *Proc. ACM MobiSys'16*.

[10] Chen, S. and Chang, Y. and Tseng, P. and Huang, C. and Lei, C. 2018. Cloud Gaming Latency Analysis. (2018). http://www.iis.sinica.edu.tw/~swc/onlive/onlive.htm

[11] Chip Brown. 2017. Bringing pixels front and center in VR video. (2017). https://blog.google/products/google-ar-vr/bringing-pixels-front-and-center-vr-video

[12] Xavier Corbillon, Francesca De Simone, and Gwendal Simon. 2017. 360-degree video head movement dataset. In *Proc. ACM MMSys'17*.

[13] DASH. 2018. Dynamic Adaptive Streaming over HTTP. (2018). https://en.wikipedia.org/wiki/Dynamic_Adaptive_Streaming_over_HTTP

[14] Daydream. 2017. Google Daydream. (2017). https://en.wikipedia.org/wiki/Google_Daydrea

[15] Renbin Peng Evgeny Kuzyakov, Shannon Chen. 2017. Enhancing high-resolution 360 streaming with view prediction. (2017). https://code.facebook.com/posts/118926451990297/enhancing-high-resolution-360-streaming-with-view-prediction/

[16] Facebook. 2016. Cubemap Transform Open Source Code. (2016). https://github.com/facebook/transform

[17] Facebook. 2016. Next-generation video encoding techniques for 360 video and VR. (2016). https://code.fb.com/virtual-reality/next-generation-video-encoding-techniques-for-360-video-and-vr/

[18] FFMPEG. 2018. FFMPEG Streaming Guide. (2018). https://trac.ffmpeg.org/wiki/StreamingGuide

[19] FLV. 2018. Video File Format Specification. (2018). https://www.adobe.com/content/dam/acom/en/devnet/flv/video_file_format_spec_v10.pd

[20] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Slacker: Fast Distribution with Lazy Docker Containers. (2016). https://www.usenix.org/system/files/conference/fast16/fast16-papers-harter.pdf

[21] Jian He, Mubashir Qureshi, Lili Qiu, Jin Li, Feng Li, and Han Lei. 2018. Rubiks: Practical 360-Degree Streaming for Smartphones. In *Proc. ACM MobiSys'18*.

[22] Mohammad Hosseini and Viswanathan Swaminathan. 2016. Adaptive 360 VR video streaming: Divide and conquer. In *Proc. IEEE ISM'16*.

[23] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. 2013. GamingAnywhere: an open cloud gaming system. In *Proceedings of the 4th ACM multimedia systems conference*. ACM, 36–47.

[24] Teemu Kämäräinen, Matti Siekkinen, Antti Ylä-Jääski, Wenxiao Zhang, and Pan Hui. 2017. A measurement study on achieving imperceptible latency in mobile cloud gaming. In *Proceedings of the 8th ACM on Multimedia Systems Conference*. ACM, 88–99.

[25] Zeqi Lai, Y Charlie Hu, Yong Cui, Linhui Sun, and Ningwei Dai. 2017. Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices. In *Proc. ACM MobiCom'17*.

[26] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. 2015. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 151–165.

[27] Luyang Liu, Ruigang Zhong, Wuyang Zhang, Yunxin Liu, Jiansong Zhang, Lintao Zhang, and Marco Gruteser. 2018. Cutting the Cord: Designing a High-quality Untethered VR System with Low Latency Remote Rendering. In *Proc. ACM MobiSys'18*.

[28] MPEGTS. 2018. MPEG transport stream. (2018). https://en.wikipedia.org/wiki/MPEG_transport_strea

[29] Afshin Taghavi Nasrabadi, Anahita Mahzari, Joseph D. Beshay, and Ravi Prakash. 2017. Adaptive 360-Degree Video Streaming Using Scalable Video Coding. In *Proc. ACM Multimedia'17*.

[30] NVIDIA. 2017. NVIDIA FFmpeg. (2017). https://developer.nvidia.com/ffmpeg

[31] Occulus. 2017. Occulus Rift. (2017). https://en.wikipedia.org/wiki/Oculus_Rift

[32] D. Ochi, Y. Kunita, A. Kameda, A. Kojima, and S. Iwaki. 2015. Live streaming system for omnidirectional video. In *Proc. IEEE Virtual Reality'15*.

[33] F. Qian, B. Han, L. Ji, and V. Gopalakrishnan. 2016. Optimizing 360 video delivery over cellular networks. In *Proc. ACM All Things Cellular'16*.

[34] Reuters. 2017. Global Virtual Reality Market (Hardware and Software) and Forecast to 2020. (2017). https://www.reuters.com/brandfeatures/venture-capital/article?id=4975

[35] Patrice Rondao Alface, Maarten Aerts, Donny Tytgat, Sammy Lievens, Christoph Stevens, Nico Verzijp, and Jean-Francois Macq. 2017. 16K Cinematic VR Streaming. In *Proceedings of the 2017 ACM on Multimedia Conference*. ACM, 1105–1112.

[36] RTMP. 2018. Real Time Messaging Protocol. (2018). https://en.wikipedia.org/wiki/Real-Time_Messaging_Protocol

[37] RTP. 2018. Real Time Transport Protocol. (2018). https://en.wikipedia.org/wiki/Real-time_Transport_Protoco

[38] Shu Shi, Cheng-Hsin Hsu, Klara Nahrstedt, and Roy Campbell. 2011. Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming. In *Proceedings of the 19th ACM international conference on Multimedia*. ACM, 103–112.

[39] Michael Hwang Rittwik Jana Shu Shi, Varun Gupta. 2019. Mobile VR on Edge Cloud: A Latency-Driven Design. In *Proc. ACM MMSys'19*.

[40] Zhaowei Tan, Yuanjie Li, Qianru Li, Zhehui Zhang, Zhehan Li, and Songwu Lu. 2018. Supporting Mobile VR in LTE Networks: How Close Are We? *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 1 (2018), 8.

[41] TPCast. 2018. TPCast Wireless Adapter. (2018). https://www.tpcastvr.com/product

[42] VIVE. 2017. HTC Vive. (2017). https://en.wikipedia.org/wiki/HTC_Vive

[43] GEAR VR. 2017. Samsung GearVR. (2017). https://en.wikipedia.org/wiki/Samsung_Gear_V

[44] Xiufeng Xie and Xinyu Zhang. 2017. POI360: Panoramic Mobile Video Telephony over LTE Cellular Networks. In *Proc. ACM CoNEXT'17*.