CloudVR: Cloud Accelerated Interactive Mobile Virtual Reality

Teemu Kämäräinen
Aalto University
Espoo, Finland
teemu.kamarainen@aalto.fi

Jukka Eerikäinen
Aalto University
Espoo, Finland
jukka.eerikainen@aalto.fi

Matti Siekkinen
Aalto University
Espoo, Finland
matti.siekkinen@aalto.fi

Antti Ylä-Jääski
Aalto University
Espoo, Finland
antti.yla-jaaski@aalto.fi

ABSTRACT

High quality immersive Virtual Reality experience currently requires a PC setup with cable connected head mounted display, which is expensive and restricts user mobility. This paper presents CloudVR which is a system for cloud accelerated interactive mobile VR. It is designed to provide short rotation and interaction latencies through panoramic rendering and dynamic object placement. CloudVR also includes rendering optimizations to reduce server-side computational load and bandwidth requirements between the server and client. Performance measurements with a CloudVR prototype suggest that the optimizations make it possible to double the server's framerate and halve the amount of bandwidth required and that small objects can be quickly moved at run time to client device for rendering to provide shorter interaction latency. A small-scale user study indicates that CloudVR users do not notice small network latencies (20ms) and even much longer ones (100-200ms) become non-trivial to detect when they do not affect the interaction with objects. Finally, we present a design of CloudVR extension to multi-user scenarios.

CCS CONCEPTS

• **Computing methodologies** → **Virtual reality**; • **Computer systems organization** → **Cloud computing**;

KEYWORDS

Virtual Reality, rendering, Unity, optimization, cloud, edge computing

ACM Reference Format:

Teemu Kämäräinen, Matti Siekkinen, Jukka Eerikäinen, and Antti Ylä-Jääski. 2018. CloudVR: Cloud Accelerated Interactive Mobile Virtual Reality. In *2018 ACM Multimedia Conference (MM '18)*, October 22–26, 2018, Seoul, Republic of Korea. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3240508.3240620>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

MM '18, October 22–26, 2018, Seoul, Republic of Korea

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5665-7/18/10.

<https://doi.org/10.1145/3240508.3240620>

1 INTRODUCTION

Virtual Reality (VR) has been rapidly gaining ground in recent years. The current method for providing a high quality immersive VR experience for human users requires a dedicated head mounted display (HMD), such as Oculus Rift or HTC Vive, that is cable connected to a PC that operates a relatively powerful graphics card. GPU computing is necessary for rendering high quality graphics and local PC deployment with tethered HMD ensures short enough latency to avoid "VR sickness" which is provoked by a mismatch between the sensory inputs of human vestibular and visual systems (visual input lags vestibular input when user rotates head). Tethered PC deployment is however relatively expensive for the user and restricts user mobility due to cabling constraints.

Recent smartphones also have dedicated platforms for VR. They consist of a separate headset to which the phone can be mounted as well as some software and hardware capabilities, such as front buffer rendering to reduce display latency. Mobile phone based VR can provide untethered experience with relatively low cost (assuming the user already possesses the smartphone). Unfortunately, the computational capacity of smartphones to render graphics is far from that of PC deployments having dedicated graphics cards.

Our goal in this work is to explore cloud-based remote graphics rendering to accelerate mobile devices in providing a high quality visual VR experience. We want to alleviate the need for local PC-based rendering setup without sacrificing quality of graphics. Obviously, providing short enough latency with such a distributed rendering system is a major challenge. One way to mask latency is to do speculative rendering and prefetching of content (e.g., [6, 16]) but that approach can be prohibitively expensive from computational and/or network traffic point of view. Hence, the other major challenge is to curb the server-side computational load as well as bandwidth demand, while dealing with latency.

We present in this paper a system called CloudVR. By default, CloudVR clients are thin so that graphics rendering is offloaded to remote servers that stream panoramic video frames to the client device, hence perceived latency from head movements to display update i.e., rotation latency, is on par with locally rendered setups. The system enables to dynamically migrate the rendering of a specific object from a remote server to the client device when deemed appropriate, in particular to reduce interaction latency with an object that the user wants to interact with. We made a decision not to support speculative rendering. In contrast, our system is designed to reduce the server-side computational load as well as

network traffic through two optimizations in graphics rendering. First, we judiciously limit the number of rendered objects that are outside of the user's current field of view (FoV). Second, we render only the current FoV in high resolution and the rest of the scene in low resolution. Our system design also addresses multi-user situations.

We present the design and prototype implementation of CloudVR and quantify the impact of the rendering optimizations on computation and traffic. The results suggest that the two render time optimizations together allow doubling the framerate of the server and halving the bandwidth usage with the same expected perceived quality by the user. In addition, small objects can be quickly moved and instantiated at run time to client device so that they can be rendered locally for shorter interaction latency. We also present results from a small-scale user study on perceivable network latency with CloudVR. They indicate that CloudVR users cannot notice small network latencies (20ms) and even much longer ones (100-200ms) become non-trivial to notice when they do not affect the interaction with objects but only translation motion.

2 BACKGROUND AND RELATED WORK

Remote rendering of graphics for interactive applications has been mostly adapted in the concept of cloud gaming. There exists multitude of cloud gaming providers already on the market and several research papers on the design and optimization of such cloud gaming systems [8, 13, 17]. In cloud gaming, user input is sent from the thin client device to the server where the input is fed into the game logic with corresponding frames being sent back to the client. The end-to-end delay requirement for cloud gaming varies between games [18]. Cloud gaming has been visioned to boost the popularity of VR games and enable infinite virtual worlds although high response delay has been identified as an open issue [7]. In CloudVR, we show that offloaded rendering is feasible with VR even with the stringent latency requirements.

While latency is a big challenge with VR, not all latencies are equal. The most stringent latency requirement in VR pertains to head movements. In other words, the *rotation latency* needs to be extremely short, less than 20ms by some estimates [19], in order to avoid the nasty physiological symptoms caused by a lag between the sensory inputs from visual and vestibular systems [12]. Another kind of latency, which we term *translation latency*, is commonly present in scenarios where a separate handheld controller is used to move the avatar forward, backward, or sideways, i.e., induce translation motion. It is experienced differently because the virtual motion is not generated by equivalent motion of the human user. Hence, there is already a sensory mismatch by design, which is not worsened by added latency. In addition, we identify separately the delay between moving a game object with a controller and observing that movement in the virtual world and term it *interaction latency*.

We are not the first to consider remote graphics rendering for VR. The closest to our work is Furion[16] by Lai et al., which, similar to us, renders panoramic frames that client can navigate locally. Different from us, the authors propose speculative rendering and pre-fetching of frames to also drive down the translation latency.

The downside is that both server side rendering load as well as network traffic gets multiplied. We conjecture that translation latency affects the usability of the VR application in a similar way than in cloud gaming. Therefore, we assume that, compared to rotation latency, substantially longer translation latencies can be tolerated and do not even try to mask it. In contrast, we design optimizations to minimize the server side computational load and network traffic. Furion also takes advantage of collaborative rendering so that the developer decides at implementation time whether a particular game object is to be rendered at server or client. CloudVR includes a dynamic object placement mechanism, which allows deciding at run time where an object is rendered.

Also others have studied remote rendering of VR content. Boos et al. developed Flashback[6] which is a mobile VR system that relies on aggressive pre-rendering and caching of frames. It is unsuitable for interactive VR applications that require real-time rendering. Reinert et al. proposed an image based rendering (IBR) system for VR [21]. Relying on IBR, it is also designed for static scenes although the authors also propose some fallbacks to handle dynamic scenes in the paper. Abari et al. designed untethered VR by using a very high bandwidth mmWave wireless link [2]. They tackle the challenges posed by signal blockage with mmWave radio links and the directional antennas combined with moving human user.

In addition, a number of papers have been published on streaming static pre-rendered 360°/panoramic video. Usually HTTP-based streaming protocols, such as DASH[1] compliant protocols, are used. HTTP follows request-response behavior (save for HTTP/2 with push) because of which video frames are grouped into chunks having duration of one or multiple seconds. This makes them unsuitable for real-time rendering where latency is critical. Streaming the entire 360° view is wasteful as the user will only watch a small portion of it. Therefore, several of these papers try to optimize the streamed panorama with some model or prediction of users' head motions [5, 10, 20]. We take a similar approach in defining a custom frustum size at render time.

3 CLOUDVR

3.1 Overview

CloudVR enables high quality graphics on mobile devices through remote rendering of graphics. The idea is that the mobile client application transmits control information to a cloud server that renders graphics and streams it as video back to the client. Obviously, this approach adds some amount of delay to the end-to-end (motion-to-photon) latency. As explained in Section 2, comfortable VR experience requires that the system reacts extremely quickly to head movements. Therefore, simply offloading the rendering of each individual frame to a remote server is not a feasible solution. Instead, CloudVR server renders and transmits panoramic video which is then navigated locally by the client, which makes the latency from head movements to display update, i.e., rotation latency, as short as with locally rendered VR content (or shorter because the mobile device only needs to decode video).

When a user wants to interact with an object, e.g., pick up a tool and use it, the game object to be interacted with (i.e., the tool) is transmitted to the client application, which begins rendering it locally. In this way, we reduce the interaction latency. The rationale

Figure 1: The thin-client - server architecture of cloud-based VR streaming (CloudVR).

is that the game objects that users typically directly interact with are relatively few, small, and lightweight to render compared to rest of the visible scene.

With the above mechanisms, translation latency is still mandated by network latency. Hence, the scene that the user sees is also somewhat delayed all the time. We consider masking this latency using speculative rendering and prefetching to be too computationally expensive and to generate too much network traffic. We also believe that this latency is nowhere nearly as critical as rotation latency for the user experience. Instead, we take an opposite direction and design a number of rendering optimizations to *reduce* the computational load of the server and the network traffic.

The roles of the server and thin client applications can be seen in Figure 1. We next describe the different parts of the system in more detail and dissect the prototype implementation built using the Unity game engine.

3.2 Rendering server

The rendering server runs the VR application and renders the VR application scene. A single rendering server can render one or more 360 player views. The player game object (including a 360 degree encoding camera) is handled by the rendering server. The role of the rendering server is thus similar to the client PC in current immersive VR applications.

The prototype implementation of the CloudVR server application consists of a Unity prefab and an accompanying plugin that encodes video. We utilize a modified version of the 360-Capture-SDK[11] published by Facebook in the video encoder. The prefab is an asset type of Unity which contains all the components and properties stored in a game object and can thus be easily imported to any scene created with the Unity game engine. Besides a powerful enough graphics card, the server does not require virtual reality-specific hardware or software.

Our implementation replaces the standard player game object which usually holds the main camera. In CloudVR the camera renders the 360 panoramic player view instead of only the traditional camera view frustum. The modified camera captures the player surroundings using square views with an edge-to-edge field of view of 90 degrees. The square views are stored to a cube map texture, a data structure for capturing full surroundings as viewed from an object in the scene. The cube map is projected to a single, flattened panoramic equirectangular video frame that is then passed to the DLL plugin for encoding and transmission. The plugin uses Nvidia's NVENC SDK to encode the video frames using a dedicated SOC available on modern GPUs. The panoramic video frames are encoded to h.264 and streamed to the client over a TCP connection. The server capture frame rate is controllable, and the rate at which the server captures the view is decoupled from the rate at which client projects the view.

In addition to rendering and transmitting video, the server listens for controller input from the client. Player's gameobject's position and orientation is changed according to user input. In the prototype Daydream implementation, the user controls the application through a single controller equipped with a touch pad and a gyro.

3.3 Thin client

The CloudVR client application is a thin client projecting a 360 degree skybox where the rendering server streams the complete 360 degree player surroundings. The client rotates the skybox according to the user's head movement. This way the rotational latency is only dependent on the client device even in cloud-based VR rendering. Translation latency is, however, dependent on the latency between the client and the rendering server.

Our prototype client implementation is an Android application created with Unity with an additional native plugin for receiving and decoding the incoming video stream. A panoramic skybox shader is used to project the equirectangular video stream. A real-time VR system, such as described here, needs to read, decode and project frames with low latency. Hence, our system buffers only a single h.264 encoded frame directly from the TCP stream and passes that to the GPU decoder. The application uses the Google VR SDK for reading the user's head orientation and controller inputs which are sent to the server using a separate RPC channel.

Even though the server is built with Unity, its interface is technology agnostic. It is possible to implement clients for different platforms depending on needs and use cases. Even though the system has been designed with mobile phone clients in mind, use of desktop clients has not been precluded in the design. A possible desktop client implementation would entail a Unity scene built with the suitable VR SDK for the headset, such as SteamVR plugin for

HTC Vive or OVR for Oculus Rift, and a DLL plugin for decoding the video stream.

3.4 Network protocols

Most existing implementations of video stream players, even in the context of virtual reality, rely on HTTP-based protocols, such as MPEG-DASH or HLS, which makes them unsuitable for a real-time system (Section 2). As existing libraries for protocols designed for transmission of video were not directly usable, the system is implemented in the simplest possible way: Raw h.264 stream transported over a TCP socket connection. For production use, the video stream from the server to the client should ideally be transmitted over, for example, RTP over UDP.

3.5 Extensibility and customization

The system is inherently extensible and customizable. The client architecture is not tied to the technological choices made in the design of the server, nor to any platform. Rather, any implementation that is able to decode panoramic or cube map-projected 360 video stream can be used to view the content produced by the server. To participate actively, the client needs to implement also control transmission scheme, but this is not directly tied to the inner workings of Unity; rather, it is enough that the client is able to send its position and control actions.

Most of the server's rendering parameters can be changed on a per-user basis to flexibly adapt to different use cases. The original rendering resolution of the cube map camera array can be adjusted for the requirements and capabilities of any particular client implementation. For example, rendering each of the cube map faces in 4096 x 4096 resolution provides no perceptible benefits when viewed with a full HD smartphone with a corresponding display FOV of 90 degrees. In a case like this, the rendering resolution can be reduced to 2048 x 2048 pixels with no perceptible detrimental effects to image quality. Furthermore, the parameters of h.264 encoding can be changed in response to, for example, network status.

The current implementation defaults to monoscopic rendering, but the system itself supports stereoscopic rendering, transmission and decoding. In the case of stereoscopic rendering, each of the cameras are replaced by a pair of cameras, each representing one of the eyes when looking along the direction the pair of cameras are pointed. The client will then have two skyboxes which are displayed for different eyes. Decoded frames need to be split and passed to skyboxes after decoding. In a naive implementation the depth perception will be perfect only for objects lying precisely on the axis, and progressively deteriorating towards the edges. In practice, though, the artifacts caused are rather tolerable. Furthermore, should stereoscopic rendering be emphasized, the client could transmit orientation data in addition to position, and the rendering camera array could be rotated so that the best prediction for future orientation is prioritized. In the next section we further improve the system by introducing rendering optimization methods to alleviate computational requirements of CloudVR.

4 RENDERING OPTIMIZATIONS

Rendering the complete 360 degree surroundings of the virtual environment can be computationally expensive and partly diminish

the benefits of remote rendering. In addition the different latencies involved may hinder the quality of experience for the user. In this section, we introduce two methods to optimize the rendering performance, custom frustum culling and multi-resolution rendering. In addition, we introduce a dynamic object placement strategy to compensate for the object interaction delay added in CloudVR.

4.1 Custom frustum culling

In CloudVR, the surroundings are initially rendered to a cube map which is rotated according to user's head orientation. It is not necessary to render the rear face of cube at all as the delay would have to be extremely high for the user to be able to change his/her head orientation before the rendering cube face is rotated on the server side to compensate.

The side and top facing cube map faces might also include unnecessary rendered objects depending on the speed of the user's head orientation change and the prevailing network delay conditions. Game engines automatically apply a method called view frustum culling where objects that are outside of the player's current field of view are not rendered. However, in 360°streaming the resulting frames would by default have all the surrounding objects rendered as the view frustum culling is applied once per cube face. We propose a custom frustum culling method where the front facing view is rendered fully, back facing view is not rendered at all and the other faces are rendered such that only the objects that are closest to the front face are rendered. In practice this is achieved by slicing the culling frustum of the camera by a percentage depending on the prevailing latency conditions. The technique allows the client to slightly adjust the user's view according to head movements in between received frames from the server. At the same time the workload of the server is kept to a minimum as only the objects that are or can become visible are rendered.

Figure 2 shows the effect of custom frustum culling on the equirectangular texture streamed to the client. The user is oriented towards the pier where the houses are rendered both with custom frustum culling enabled or disabled. Additional houses in the left and right cube map faces are not rendered when the optimization is enabled. The black area in both pictures is the rear face of the cube map which is not rendered at all. We evaluate the performance of the custom culling method in Section 5.3.2.

4.2 Multi-resolution rendering

In addition to not rendering objects that are unlikely to be viewed by the user before the next frame (with updated orientation), CloudVR also introduces a multi-resolution rendering scheme where the front face of the cube is rendered with a higher resolution than the rest. We argue that the side, bottom and top faces of the cube map are mostly viewed by the user's peripheral vision and can be rendered with a lower quality than the front face of the cube map which is the main view seen by the user. This is due to the fact that in CloudVR the initial cube map rendering is rotated according to the user's head movements, only lagging behind by the end-to-end delay. We evaluate the multi-resolution rendering scheme in Section 5.3.1.

Figure 2: Custom frustum culling disabled (left), enabled (middle) and with multi-resolution rendering (right).

4.3 Dynamic object placement

In order to reduce the interaction latency, we propose a dynamic object placement strategy where the control of game objects in the scene can be transferred between the client and server as required: When an object on the server is going to be interacted with by a player, a similar object can be instantiated on the client to be locally rendered for shorter latency interactions. The detection of such an event can be a simple game logic where a player grabs an object or a more sophisticated heuristic, such as a distance measurement, that suggest near-future interactions with a player. The server's instance of the object is hidden from the client that has taken control of the object and is now rendering the object locally. The server's instance is also set to update its position from the client's instance. When the client releases control of the object, the server's instance is freed and rendered for all clients and the client's instance is destroyed or hidden.

In our prototype implementation we utilize the Asset bundle feature of the Unity game engine to create downloadable binaries of object's placed in the scene. We created an RPC-message based logic where the server commands the client to instantiate a copy of an object when the user picks up for example a sword. As both the server and the thin client software utilize the same game engine, we can download the specific asset bundle (object) in the client, instantiate it, attach it to be followed by the controller orientation and start streaming the object's orientation to the server. We measure the time needed to transfer an object's control from the server to the client in Section 5.4.

5 PERFORMANCE EVALUATION

5.1 Setup

We built the Viking Village test scene directly for the Android platform in addition to the CloudVR desktop prototype to compare the performance between the two. We used four mobile phones in the comparison: more recent Google Pixel 2 XL and Huawei Mate 9 Pro and two older models Huawei Nexus 6P and Samsung Galaxy S7. The desktop computer is equipped with an Nvidia GTX 1080 GPU and an i5-2500K CPU. In addition to the native performance comparison of Section 5.2, we measured the effect of the rendering performance optimizations on framerate and bandwidth in Sections 5.3.1 and 5.3.2 and finally the time needed for dynamic object placement in Section 5.4.

Figure 3: Framerate when running the Viking Village scene directly on the Android phone or using the CloudVR system. For supported phones we also show the VR (Daydream) performance.

5.2 CloudVR vs. Standalone Mobile Client

Modern mobile phones can render relatively high quality 3D content. However, the performance gap between desktop and mobile GPUs still remains. We use the Unity technology demo scene Viking Village in our experiments to showcase the benefits of remote rendering. For the CloudVR system to be feasible, it needs to outperform the local rendering use case. For this purpose we build the Viking Village scene for Android and run a set 80 second benchmark path while measuring the framerate of the application.

Figure 3 shows how all the four mobile phones tested struggle to render the scene. The two older phones Huawei Nexus 6P and Samsung Galaxy S7 both achieve roughly 6 frames per second performance. The newer top of the line mobile phones Huawei Mate 9 Pro and Google Pixel XL 2 achieve 8.5 and 10.9 frames per second. These two mobile phones also support the Daydream VR platform for which the builds drop the fps even further down to 4.3 and 6.1 respectively as the Daydream platform uses stereoscopic rendering.

The CloudVR client is able to decode and show the remotely rendered scene with a steady target frame rate of 60. Thus the performance of CloudVR is solely dependent on the performance of the rendering remote server. The cloud server can be scaled based on the requirements of the application which is not the case with mobile client hardware.

5.3 Rendering Performance Optimizations

In Section 4, we introduced methods to optimize the rendering performance of the CloudVR server. In order to evaluate their feasibility

Figure 4: Framerate improvement with Multi-resolution rendering and Custom frustum culling in the Viking Village scene.

Figure 5: Bandwidth use of CloudVR with and without Multi-resolution rendering (MRR) and Custom frustum size (CFC). (Resolution: 2880x1440, Fixed framerate: 40)

we measured a baseline FPS of 91 running the Viking Village scene with the graphics preset *Beautiful* to stress the GPU. The baseline was calculated using a cubemap face resolution width and height of 2048 for all cubemap faces. The rendering performance tests were run without the encoding component to find out the effect of the optimizations on the rendering performance. In the bandwidth tests we fixed the FPS to 40 and the video size to 2880x1440 using constant Quantization Parameter (QP) rate control set to 28.

5.3.1 Multi-resolution rendering. Figures 4 and 5 show the effect of the optimizations to the rendering performance. The multi-resolution methods shows a clear framerate improvement. For example rendering the non user-facing sides of the cubemap with a resolution width and height of 512 increases the framerate from the original 90 to 160. In the bandwidth measurements (Figure 5) where the framerate was fixed to 40, multi-resolution rendering dropped the bandwidth requirement from 20 to 14 Mbit/s. In our tests, we also observed that the non-user facing resolution of 512x512 is still feasible for the peripheral vision not to be disrupted by clearly a lower level of detail. The reason for this tolerance stems from the limitations of human visual system (HVS). When a user rotates head, thereby also refocusing gaze, there is a short period of time when the HVS omits some of the visual input. For example, Albert et al. conclude from their study that an end-to-end latency of 50–70ms is tolerable in foveated rendering[3].

5.3.2 Custom frustum size. The rendering performance can be further improved by introducing the custom frustum culling method introduced in Section 4.1. To first understand how large the frustum needs to be so that the user does not notice its limited size, we analyzed head movements from two VR usage datasets: dataset1[24] and dataset2[9]. We calculated the maximum rotation around y-axis (pitch) and z-axis (yaw) within a specific time window. From this we inferred the fraction of peripheral faces that need to be rendered so that the user's head movements according to the datasets would not exceed the frustum before it gets updated. The results for 100ms time window (i.e., time from head movement to update on display) are shown in Figure 6. They suggest that rendering another 10% of the side, top, and bottom faces should be sufficient for 99% of the time if the end-to-end latency is at most 100ms.

The performance improvement caused by custom frustum culling (Figure 4) is most visible together with the lower resolution settings of the multi-resolution rendering method where an additional 20-30 fps improvement is achieved. With custom frustum size we were also able to reduce the bandwidth use an additional 2-3 Mbit/s. Overall using the two rendering optimization methods geared towards performance, we were able to double the framerate of the CloudVR server. Alternatively with a fixed framerate, the optimizations are capable of reducing the bandwidth usage by 40% while still using the sensible 512x512 resolution for non user-facing cubemap sides.

5.4 Dynamic object placement

In addition to the rendering performance optimizations, we introduced a dynamic object placement in Section 4.3 to reduce the interaction latency. We tested the dynamic placement with two objects using our prototype implementation. To evaluate the time needed to transfer a object for the client we created two asset bundles, one for a larger object (a house in the village) in the scene and another for a small object (a sword).

Table 1 shows the asset bundle file sizes, measured web request times and instantiation times after download. The measurements show that the download times can be relatively high compared to the actual instantiation. However, the downloaded objects can be cached and instantiated later without the web request time.

Table 1: Dynamic object placement file sizes and times for a small and large object.

Object	Object Size (verts, tris)	Asset Size (Mbytes)	WebRequest time (ms)	Instantiation time (ms)
Sword (small)	123, 156	0.18	232	23
House (large)	24k, 33k	20.48	3692	242

6 USER STUDY

In order to get a preliminary understanding of how CloudVR users perceive latency, we conducted a small-scale user study.

6.1 Setup

We setup Google Pixel 2 XL with Daydream View as the client device. The server was a PC equipped with Nvidia Geforce GTX

Figure 6: Required normalized lengths for peripheral faces of custom frustum with 100ms end-to-end latency based on two VR usage datasets.

1080 graphics card. It was configured to output graphics to the client using 1440x720 resolution and framerate 100 to minimize the base latency. Test subjects were asked to use the Viking Village scene where they could move forward and backward, look around, and swing a sword. We used two versions of the scene, one where the sword was rendered by the server and another one where it was rendered locally by the client. All the other graphics were rendered by the server in both versions.

The study was designed as two-alternative forced choice (2AFC [15]) task where the subjects were instructed to move, look around, and swing the sword at their own will, while we presented them with two alternative configurations, one immediately after the other. One of the configurations did not have any added latency, i.e., only baseline latency, whereas the other one was setup with added network latency of 20-200ms using clumsy[22]. The subjects were to choose the one that in their opinion appeared to have longer latency, felt more sluggish or somehow slower than the other one. The subjects knew neither the difference between the two different versions of the application (locally vs. remotely rendered sword) nor how exactly the longer latency would manifest itself. Ten subjects were recruited to the study.

6.2 Baseline Latencies

Before the user study, we wanted to understand the baseline latencies. Hence, we measured the end-to-end latency from control to photon by moving the Daydream's Bluetooth controller while the VR scene was playing, and meanwhile shooting a slow motion video with another smartphone (Figure 7). Through manual inspection of the video, we identified the approximate latency between the motion of the controller and the motion of the sword in the VR world. In addition, we measured the delay from head movement to screen update in a similar fashion with the difference that now the the edge of the high resolution front face was detected in the video analysis and compared to the physical rotation of the device.

As a result, we obtained a latency of roughly 100ms and 140ms for the Daydream control with local and remote rendering of the sword, respectively, which highlights the relatively long delays caused by the controller and the mobile device display update pipeline [14]. The end-to-end latency from head movement to updated frame drawn on display was 85-90ms. Note that this latency is not the same as rotation delay, which is directly perceived by the user when

Figure 7: Baseline latency measurement test with both a local and a remote object interaction (sword).

Table 2: Percentage of correct answers from the 2AFC study.

added latency	20ms	50ms	100ms	200ms
w/ remotely rendered sword	56%	60%	90%	100%
w/ locally rendered sword	67%	78%	78%	78%

rotating head, but it corresponds to the latency that should be used to calculate the frustum size according to estimated head motion (Section 5.3.2).

6.3 Results

We show in Table 2 the percentage of samples where the test subject correctly chose the longer latency among the two alternatives. Although the sample size is relatively small, it gives clear indication of two interesting phenomena. First, with only 20ms of added latency, the results are practically the same as with random guess meaning that such latency difference is unnoticeable. In contrast, with 100ms and 200ms of added latency, the test subjects almost invariably were able to detect the difference when the sword was rendered at the server. The other interesting observation is that the results are more indecisive specifically with the longer latencies when the sword is locally rendered, which suggests that a noticeable difference in latency is shorter with interaction latency than with translation latency. This is a positive result because it is relatively "easy" to reduce the interaction latency through dynamic object placement.

7 MULTI-USER CLOUDVR

So far, we have only considered single user scenarios. In this section, we discuss how CloudVR could be extended to support multi-user scenarios.

7.1 Multi-User Scenes with Masters and Slaves

Interactive VR applications and games are usually divided to separate logical areas, which can be called levels or scenes depending on the implementation. The scenes can be completely independent parts of the world or in case of completely open worlds, artificially divided parts of the game world.

We design the extension of CloudVR to multi-user scenarios so that each scene is served by one master server and multiple rendering servers that we call slaves. The slave servers are physically located close to the users in order to provide low network latency between them and the client devices. The master server keeps track of the game objects in the scene and acts as a relay between the RPC calls from the rendering servers. Its role is similar to that of a traditional multiplayer game server.

The slave servers need to be launched rapidly on-demand when new players enter the system or move between level/area borders in the game. We envision to deploy them as containers which can be launched, suspended, and resumed considerably faster than traditional virtual machines without sacrificing application-level performance. In addition, the computational requirements for rendering multiple users simultaneously inside a level may vary significantly over time, for which the system needs to be able to scale the rendering in almost real-time for different scenarios.

7.2 Cluster Rendering

Cluster rendering has been successfully used in dividing the rendering workload in multi-screen environments such as video walls, CAVEs and Domes [4]. We propose to use the same architecture in the multi-user extension of CloudVR. Existing game engine implementations of cluster rendering can scale up to 50 displays [23]. Traditionally, the network latency limits the distance between the servers in one cluster. However since the render results of the different CloudVR slaves are not viewed side-by-side, the network latency between the master and slaves can be larger and lock-step synchronization is not needed.

In contrast to traditional cluster rendering, our proposed system can render multiple player views on one server if the machine has available resources. This approach is possible as the player views are only panoramic capture and encode objects in the game world, not traditional game scene cameras with a connection to a physical display. Server sharing can save costs for the service provider since the mapping between the application servers and players does not have to be one-to-one.

7.3 Client to Server Mapping

Client to server mapping in multi-user CloudVR where some clients may be geographically close and others distant from each other has a direct impact on perceivable latencies. As in the single user scenario, client devices may start rendering game objects locally when the user interacts with them. In this case, the slave server that the client is connected to starts to receive updates of the object position from the client. Other slave servers that render scenes showing that object receive the position from this slave and render the object for their clients. As a result, the perceivable latencies pertaining to that game object may be vastly different between users. Latency between the rendering server and a master server has a similar effect than in traditional online multiplayer games. It can however be masked to some extent using interpolation and dead-reckoning.

Figure 8 shows the hierarchical structure of an example rendering cluster. Clients 1 and 2 are geographically located close to each other and the cluster assigns them to the same render server s1. Client c1

Figure 8: The render server cluster architecture.

is currently interacting with object 3 and controls the object, while the rendering server s1's instance of object 3 follows c1's instance. All other objects in the scene visible to client c1 are rendered on the server s1 and only the video is transmitted. Object 2 is a potential future candidate for interactions with either client c1 or c2, and thus it is owned and controlled by server s1. Object 1 is further away, and hence owned by the master server s0. Both clients c2 and c3 only see the transmitted video.

As the majority of objects in most game scenes tend to use mostly-diffuse materials, an optimization possibility emerges: As a diffuse material fragment's color does not depend on the view angle, the color for a particular fragment will be the same for all viewers of the scene. Thus, a rendering cluster can render the diffuse materials only once, essentially baking light maps onto the textures themselves. Then, the cluster needs to only render the surfaces with some specularities per each client individually. The effectiveness of this technique is naturally dependent on the scene.

8 CONCLUSION

This paper presents a design and prototype of a cloud accelerated mobile VR system, which offloads most of the rendering to a remote server in order to enable high quality VR on smartphones. We designed two rendering optimizations to reduce the server side workload and a mechanism for dynamic object placement. The system is designed to provide short rotation and interaction latencies, but does not try to mask the latency of translation motion. Performance evaluation shows that the rendering optimizations reduce significantly the server side load and bandwidth demand and that dynamic placement and instantiation of small objects from server to client device is fast. A small-scale user study indicates that small added network latencies (20ms) are hardly noticeable even when it affects interaction latency. In addition, even a much longer latency of up to 200ms becomes more difficult to notice when it does not affect interaction latency but only translation latency.

As future work, we intend to extend the prototype implementation to include the designed multi-user extension. In addition, we plan to conduct a more thorough user study of the perceived latency and include the impact of graphics quality dynamics (multi-resolution rendering) in the study.

ACKNOWLEDGMENTS

This work has been financially supported by the Academy of Finland (grant number 297892) and the Nokia Center for Advanced Research.

REFERENCES

- [1] ISO/IEC 23009. 2012. Dynamic adaptive streaming over HTTP (DASH). (2012).
- [2] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. 2017. Enabling High-quality Untethered Virtual Reality. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, Berkeley, CA, USA, 531–544. <http://dl.acm.org/citation.cfm?id=3154630.3154674>
- [3] Rachel Albert, Anjul Patney, David Luebke, and Joohwan Kim. 2017. Latency Requirements for Foveated Rendering in Virtual Reality. *ACM Trans. Appl. Percept.* 14, 4, Article 25 (Sept. 2017), 13 pages. <https://doi.org/10.1145/3127589>
- [4] Giovanni Avveduto, Franco Tecchia, Marcello Carrozzino, and Massimo Bergamasco. 2016. A Scalable Cluster-Rendering Architecture for Immersive Virtual Environments. In *International Conference on Augmented Reality, Virtual Reality and Computer Graphics*. Springer, 102–119.
- [5] 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.
- [6] Kevin Boos, David Chu, and Eduardo Cuervo. 2016. FlashBack: Immersive Virtual Reality on Mobile Devices via Rendering Memoization. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '16)*. ACM, New York, NY, USA, 291–304.
- [7] Wei Cai, Ryan Shea, Chun-Ying Huang, Kuan-Ta Chen, Jiangchuan Liu, Victor CM Leung, and Cheng-Hsin Hsu. 2016. The future of cloud gaming [point of view]. *Proc. IEEE* 104, 4 (2016), 687–691.
- [8] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. 2012. The Brewing Storm in Cloud Gaming: A Measurement Study on Cloud to End-user Latency. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games (NetGames '12)*. IEEE Press, Piscataway, NJ, USA, Article 2, 6 pages. <http://dl.acm.org/citation.cfm?id=2501560.2501563>
- [9] Xavier Corbillon, Francesca De Simone, and Gwendal Simon. 2017. 360Degree Video Head Movement Dataset. In *Proceedings of the 8th ACM on Multimedia Systems Conference (MMSys'17)*. ACM, New York, NY, USA, 199–204. <https://doi.org/10.1145/3083187.3083215>
- [10] Xavier Corbillon, Alisa Devlic, Gwendal Simon, and Jacob Chakareski. 2017. Optimal Set of 360-Degree Videos for Viewport-Adaptive Streaming. In *Proceedings of the 2017 ACM on Multimedia Conference (MM '17)*. ACM, New York, NY, USA, 943–951. <https://doi.org/10.1145/3123266.3123372>
- [11] Facebook. 2018. 360-Capture-SDK. (2018). <https://github.com/facebook/360-Capture-SDK>
- [12] PA Howarth and PJ Costello. 1997. The occurrence of virtual simulation sickness symptoms when an HMD was used as a personal viewing system. *Displays* 18, 2 (1997), 107–116.
- [13] Chun-Ying Huang, Kuan-Ta Chen, De-Yu Chen, Hwai-Jung Hsu, and Cheng-Hsin Hsu. 2014. GamingAnywhere: The First Open Source Cloud Gaming System. *ACM Trans. Multimedia Comput. Commun. Appl.* 10, 1s, Article 10 (Jan. 2014), 25 pages.
- [14] Teemu Kämäräinen, Matti Siekkinen, Antti Ylä-Jääski, Wenxiao Zhang, and Pan Hui. 2017. Dissecting the End-to-end Latency of Interactive Mobile Video Applications. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications (HotMobile '17)*. ACM, New York, NY, USA, 61–66.
- [15] FAA Kingdom and N Prins. 2010. Psychophysics: a practical introduction. (2010).
- [16] 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 *Proceedings of the 23rd International Conference on Mobile Computing and Networking (MobiCom'17)*. ACM, Snowbird, Utah, USA.
- [17] 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 (MobiSys '15)*. ACM, New York, NY, USA, 151–165.
- [18] Yeng-Ting Lee, Kuan-Ta Chen, Han-I Su, and Chin-Laung Lei. 2012. Are All Games Equally Cloud-gaming-friendly?: An Electromyographic Approach. In *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games (NetGames '12)*. IEEE Press, Piscataway, NJ, USA, Article 3, 6 pages. <http://dl.acm.org/citation.cfm?id=2501560.2501564>
- [19] OculusRift-Blog.com. [n. d.]. John Carmack's Delivers Some Home Truths On Latency. ([n. d.]). <http://oculusrift-blog.com/john-carmacks-message-of-latency/>.
- [20] Feng Qian, Lusheng Ji, Bo Han, and Vijay Gopalakrishnan. 2016. Optimizing 360 Video Delivery over Cellular Networks. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges (ATC '16)*. ACM, New York, NY, USA, 1–6.
- [21] Bernhard Reinert, Johannes Kopf, Tobias Ritschel, Eduardo Cuervo, David Chu, and Hans-Peter Seidel. 2016. Proxy-guided Image-based Rendering for Mobile Devices. In *Computer Graphics Forum*, Vol. 35. Wiley Online Library, 353–362.
- [22] Chen Tao. 2018. clumsy. (2018). <https://jagt.github.io/clumsy/>
- [23] Unity. 2017. Unity Documentation. Cluster Rendering. (Nov. 2017). Retrieved Nov. 19, 2017 from <https://docs.unity3d.com/Manual/ClusterRendering.html>
- [24] Chenglei Wu, Zhihao Tan, Zhi Wang, and Shiqiang Yang. 2017. A Dataset for Exploring User Behaviors in VR Spherical Video Streaming. In *Proceedings of the 8th ACM on Multimedia Systems Conference (MMSys'17)*. ACM, New York, NY, USA, 193–198. <https://doi.org/10.1145/3083187.3083210>