

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Thomas van Gemert

Dynamic Viewport-Adaptive Rendering in Distributed Interactive VR Streaming:

Optimizing viewport resolution under latency and viewport orientation constraints

Master's Thesis
Espoo, September 19, 2019

Supervisor: Prof. Antti Ylä-Jääski
Advisor: Dr. Matti Siekkinen

Aalto University
 School of Science

 Master's Programme in Computer, Communication and
 Information Sciences

 ABSTRACT OF
 MASTER'S THESIS

Author:	Thomas van Gemert		
Title:	Dynamic Viewport-Adaptive Rendering in Distributed Interactive VR Streaming: Optimizing viewport resolution under latency and viewport orientation constraints		
Date:	September 19, 2019	Pages:	71
Major:	Computer Science	Code:	SCI3042
Supervisor:	Prof. Antti Ylä-Jääski		
Advisor:	Dr. Matti Siekkinen		
<p>In streaming Virtual Reality to thin clients one of the main concerns is the massive bandwidth requirement of VR video. Additionally, streaming VR requires a low latency of less than 25ms to avoid cybersickness and provide a high Quality of Experience. Since a user is only viewing a portion of the VR content sphere at a time, researchers have leveraged this to increase the relative quality of the user viewport compared to peripheral areas. This way bandwidth can be saved, since the peripheral areas are streamed at a lower bitrate. In streaming 360° video this has resulted in the common strategy of tiling a video frame and delivering different quality tiles based on current available bandwidth and the user's viewport location. However, such an approach is not suitable for real-time Interactive VR streaming. Furthermore, streaming only the user's viewport results in the user observing unrendered or very low-quality areas at higher latency values. In order to provide a high viewport quality in Interactive VR, we propose the novel method of Dynamic Viewport-Adaptive Rendering. By rotating the frontal direction of the content sphere with the user gaze, we can dynamically render more or less of the peripheral area and thus increase the proportional resolution of the frontal direction in the video frame. We show that DVAR can successfully compensate for different system RTT values while offering a significantly higher viewport resolution than other implementations. We further discuss how DVAR can be easily extended by other optimization methods and discuss how we can incorporate head movement prediction to allow DVAR to optimally determine the amount of peripheral area to render, thus providing an optimal viewport resolution given the system constraints.</p>			
Keywords:	Virtual Reality, Cloud, Streaming, VR, Optimization, Daydream, Rendering, Latency, Head Movement Velocity, Cloud Gaming, Interactive VR, Mobile, Resolution, Quality of Experience		
Language:	English		

Acknowledgements

I would like to thank the CloudXR¹ team at Aalto University, in particular Matti Siekkinen, Teemu Kämäräinen and Olavi Mertanen for their insight and support during the development of this work. I furthermore extend my thanks to my supervisor, Prof. Antti Ylä-Jääski for his support, Gazi Illahi for several fruitful discussions and Prof. Antti Oulasvirta for his inspiration and support during my education.

Espoo, September 19, 2019

Thomas van Gemert

¹<https://cloudxr.cs.aalto.fi>

Abbreviations and Acronyms

AR	Augmented Reality
DASH	Dynamic Adaptive Streaming over HTTP
DoF	Degrees-of-Freedom
DVAR	Dynamic Viewport-Adaptive Rendering
FoV	Field-of-View
HMD	Head-Mounted Display
JVET	Joint Video Experts Team
MR	Mixed Reality
QEC	Quality Emphasis Center
QER	Quality Emphasis Region
QoE	Quality of Experience
RTT	Round-Trip Time
VR	Virtual Reality
VRE	Virtual Reality Environment
Wi-Fi	Radio technologies for WLAN

Contents

Abbreviations and Acronyms	4
1 Introduction	7
1.1 Problem Statement	9
1.2 Scope of the Thesis	10
1.3 Structure of the Thesis	11
2 Background	12
2.1 Virtual Reality	12
2.2 Virtual Reality Streaming	14
2.3 Head Orientation and Projection Mapping	17
2.4 Related work: 360° Video Streaming	19
2.5 Related work: Real-time Interactive VR Streaming	20
3 System Architecture	23
3.1 Server	24
3.2 Client	26
4 Dynamic Viewport-Adaptive Rendering (DVAR)	28
4.1 Idea	28
4.2 Implementation	30
4.3 Optimization	31
4.3.1 Compensating for System Latency	32
4.3.2 Head Movement	35
5 Experiment Setup	38
5.1 Measurements	38
5.1.1 Unsuccessful Frames	40
5.1.2 Unrendered Area	40
5.1.3 Viewport Resolution	42

5.1.4	Bandwidth, Latencies and System Performance	42
5.2	Materials & Apparatus	43
5.3	Emulating Head Movement	43
5.3.1	Recording the Head Traces	43
5.3.2	Emulating the Head Traces	44
5.4	Reference Conditions & Encoding Parameters	44
5.5	Procedure	45
6	Results	47
6.1	Overall Performance	47
6.1.1	Viewport Resolution	47
6.1.2	Bandwidth	48
6.1.3	Unsuccessful Frames & Effective Bandwidth	49
6.1.4	System Performance	50
6.2	Per-user Analysis	50
6.2.1	User 1	50
6.2.2	User 2	52
6.2.3	User 3	54
6.2.4	User 4	55
6.2.5	User 5	57
6.3	Reference Conditions	58
6.4	Discussion	59
7	Future Work	63
8	Conclusions	65
	Bibliography	65

Chapter 1

Introduction

In recent years Virtual Reality (VR) has seen a significant increase in attention from researchers, market, hardware/software developers and consumers. A recent report predicts that the VR market is expected to reach a value of \$26.89 billion in 2022 [52]: a massive increase from \$2 billion in 2016. Although VR research is not "new" as such, and goes back to the 1980's, recent developments and releases of high-end consumer-grade Head-Mounted Displays (HMD's) such as the HTC Vive and Oculus Rift have made both development and research of VR more accessible. VR content comes in many forms nowadays, but mainly as either video games/computer-generated 3D imagery or 360° video (otherwise known as panoramic or omnidirectional video). In order to view VR content an HMD is required, which is a binoculars-like device that tracks head movement to match the viewport¹ of the virtual environment to the direction a user is looking. Apart from high-end and expensive HMD's such as the Oculus Rift and HTC Vive, low-end HMD's have also been published for mobile devices such as Google Daydream, Google Cardboard or Samsung GearVR. In these HMD's the smartphone is used to take care of the tracking and display of the Virtual Reality Environment (VRE), while the HMD provides the lenses necessary for viewing the VRE in 3D. Both a challenge and a benefit of these mobile solutions is that they are untethered, and thus do not require a separate high-end desktop PC to render VR content. Furthermore, manufacturers such as Oculus and Lenovo have presented stand-alone mobile HMD's such as the Oculus Go,

¹The viewport is the area of a VR image that the user can see at a given moment. In traditional video this is simply all parts of the video visible on screen. In VR video, as the user is watching part of a sphere through the HMD's lenses, this is the part that the user sees through the lenses. The size of the viewport depends on the field-of-view of the HMD, expressed in degrees. Hence, the viewport size does not depend on the resolution of the video or the physical size of the HMD's display.

Oculus Quest or Lenovo Mirage Solo that combine the features of high-end HMD's with full mobility, based on a mobile chipset.

Both mobile smartphone-based and stand-alone HMD's offer benefits in viewing VR content because they do not require expensive supporting hardware, an array of cables or a restricted usage area. These mobile HMD's either render the content locally or receive it over a wireless connection such as Wi-Fi or LTE. Although smart devices are getting increasingly more powerful and are catching up fast to modern PC's in terms of computing power, their performance is still limited and is likely to remain limited compared to high-end PC's since their primary objectives in performance management differ: smart devices generally prioritize power savings and efficiency, while PC's offer a much higher performance. As such, mobile devices are only capable of rendering low-quality 3D images. Streaming VR content over a wireless connection presents a possible solution to provide high quality graphics, as the local device only has to display it on screen and track user and controller position. However, the bandwidth of wireless connections is limited and prone to more or less severe fluctuations.

Due to the nature of VR content the bandwidth requirements of streaming VR content are several factors higher than those of streaming traditional 2D video: in a virtual environment a user's point of view resides inside a sphere, while the VR content is projected to the inside of the sphere. This gives the user a full $360^\circ \times 180^\circ$ Field of View² (FoV). This means that, in order to stream the full spherical content, the transmission of a 360° video consumes 4-6x the bandwidth of a regular video with the same resolution [13], [51], easily reaching 12K pixels resolution for common HMD's. However, the user is only looking at a part of the sphere at a given time (the viewport). In order to present the viewport in sufficient quality to maintain a good Quality of Experience (QoE), the viewport resolution is generally at least the native resolution of the HMD, for example 1600x1440 pixels per eye for the recently released Oculus Quest. Because of the massive resolution of VR content and the limited bandwidth of current wireless connections, it is infeasible to stream any VR content of such a connection. Due to encoding technology such as H.264 (AVC), H.265 (HEVC) or VP9 the bandwidth requirement can be decreased, but it remains significantly higher than that of traditional video. Additionally, VR streaming requires a new video frame to be displayed at a very low latency for a smooth experience, which requires high Frames

²Field-of-View (FoV) refers to the area a user can see, either through his eyes or a device. It is usually measured in degrees. For example, 360° video provides a potential 360° FoV, as the video contains content for each possible viewing direction. However, a typical HMD might offer only a 100° FoV, allowing the user to see 100° worth of content at a time.

Per Second (FPS) rendering, further increasing the bandwidth requirement.

1.1 Problem Statement

A central theme in the VR research community in recent years has been: "How can we further reduce the bandwidth requirements of VR streaming without decreasing the Quality of Experience?". Even with the promise of new Wi-Fi technology, server technology or 5G networking technology, HMD's are expected to also support higher resolutions to further increase the potential visual quality. However much bandwidth becomes available in the future, we can expect the receiving end to match it in demand, so the problem of determining an efficient streaming solution for VR remains (not unlike traditional video streaming optimization). A popular strategy is to leverage the fact that a user only views one portion of a VR video at a time, thus giving content delivery systems the opportunity to deliver higher quality in that area, while limiting the bandwidth requirement for the peripheral areas. Recently many methods for this have utilized the recent MPEG-DASH streaming protocol: by dividing a video into separate tiles and training a content delivery model on head traces from users watching 360° video a quality level of a tile can be determined, based on it's location w.r.t. the user's viewport and the available bandwidth.

Primary downsides of this approach are the required extra storage capacity on the server side, the non-trivial determination of how to generate the tiles and quality levels, or how to implement viewport awareness. Most importantly, however, is that most of these works focus on streaming 360° video, instead of streaming a VR application such as a game (much like in traditional cloud gaming). Streaming Interactive VR requires taking into account the effect of interactivity and requires real-time changes to any dynamic quality variations. This excludes tiling and other pre-generated video based solutions as the additional overhead of creating tens of different versions of tiles of each video frame would violate the latency requirement.

In a lesser way researchers have begun investigating VR streaming systems: due to the strict latency requirements it is non-trivial to set up a VR streaming system that delivers the required QoE. Given that the motion-to-photon latency should be below 10-25ms [7], [35], a server has to render, encode and transmit a frame within, say, 10ms to give the client 6ms to present the frame. Separately, work has been published on improving wireless network connectivity and hardware performance for VR streaming, but the work on combining these new technologies into a VR system that can support fully interactive VR at high resolutions is limited. The task is fur-

ther complicated by the fact that mobile devices such as smartphones, a key target for VR streaming, have limited processing power and are bound by power-saving policies.

1.2 Scope of the Thesis

In this work we present a novel viewport-adaptive rendering solution for VR streaming. By rotating the frontal direction of the VR video sphere with the gaze of the user, we can dynamically increase or decrease the amount of peripheral area to render since this area is not visible to the user. Less peripheral area rendered not only enables lower rendering times, but leaves room on a video frame to increase the proportional resolution of the frontal direction. Our solution optimizes the resolution in the viewport by rendering only the viewport location and a surrounding area based on the current network conditions and head orientation parameters. By utilizing the remaining available bandwidth to increase the viewport resolution we increase the QoE while simultaneously compensating for the effects of system latency. Our solution, which will be introduced in Chapter 4, is designed for streaming Interactive VR content, such as cloud-based VR gaming, but can also support 360° video streaming. The system is designed for live streaming of computer-generated images, such as games or simulations, and streams these at an optimal quality level to a standalone or smartphone-based client device.

The amount of peripheral area to render critically depends on the system's response time and the user's head movement velocity. Our solution dynamically adapts itself to the system's Round-Trip Time (RTT) (time before a requested frame is rendered on the client) and the user's head movement velocity to offer exactly the right amount of peripheral area to avoid showing any unrendered area. This way we can maximize the viewport resolution while offering a completely transparent experience to the user. Because we change the amount of peripheral area to render at render time (i.e. before encoding), our solution can easily be combined with existing approaches such as tiling, improved encoding, head movement prediction, gaze prediction, optimized software algorithms for rendering and coding, etc. Furthermore, our solution comprises a low-impact, high-level software solution which allows for easy improvements as hardware and technology support allow for even higher resolutions and frame rates. We show that Dynamic Viewport-Adaptive Rendering (DVAR) is capable of precisely reacting to changes in system RTT, and that DVAR requires less bandwidth than comparative implementations while offering a significantly higher viewport resolution.

1.3 Structure of the Thesis

This work is organized as follows: we first elaborate on the concept of virtual reality, VR streaming the different factors involved and related works in Chapter 2: Background. We then present a prototype streaming system in Chapter 3: System Architecture. Our Dynamic Viewport-Adaptive Rendering proposal is presented in the likewise-named Chapter 4, after which we set up an experiment to evaluate DVAR's performance. The experiment set-up is discussed in Chapter 5: Experiment Setup and the results of our experiments are presented and discussed in Chapter 6: Results. In the same chapter we furthermore discuss future improvements. Finally, we summarize our work in Chapter 8: Conclusions.

Chapter 2

Background

Virtual Reality is defined by the Oxford dictionary as "The computer-generated simulation of a three-dimensional image or environment that can be interacted with in a seemingly real or physical way by a person using special electronic equipment, such as a helmet with a screen inside or gloves fitted with sensors" [48]. In other words, Virtual Reality can refer to the three-dimensional computer generated environment of a video game, in which the user is immersed as if they were physically present in the virtual environment by means of a (head-mounted) stereoscopic display, binaural audio and simulation of physical hand, head and leg positions in the virtual world. VR revolves around making a user believe they are physically present in a non-physical environment, by substituting real-world input to the senses by artificially created ones.

2.1 Virtual Reality

Although recently VR and its related technologies have seen a massive influx of interest, the concept may be older than the computer itself [49]. The first devices and use cases that are related to how we know VR nowadays come from the 70's and 80's, when NASA's Ames Research Center set up a Virtual Interactive Environment Workstation based on a headset that offered stereoscopic imagery and a convincingly large Field-of-View [49]. During this time also game console manufacturers such as SEGA and Nintendo released VR devices that were capable of displaying stereoscopic images. It should be noted however, that in this time VR did not exclusively refer to the HMD-based idea: in fact, still to this day it encompasses any virtual environment that may be convincing enough to immerse the user. This includes, for example: driving and flying simulators, projection based ("holodeck style")

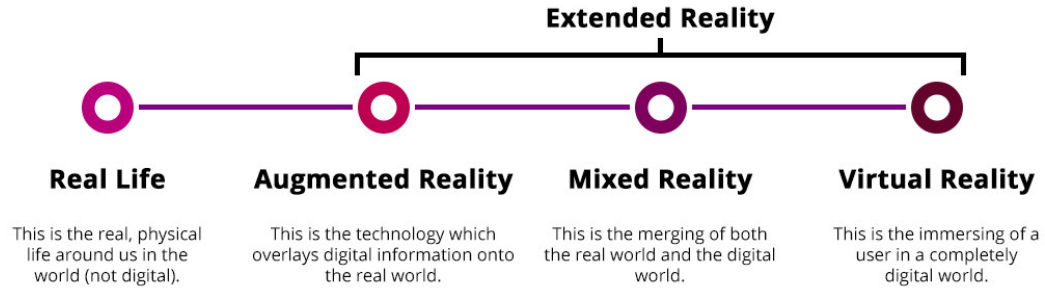


Figure 2.1: An overview of the XR spectrum containing Augmented Reality, Mixed Reality and Virtual Reality technologies in increasing order of immersion into a virtual environment. Image from [37].

VR rooms and telepresence solutions. For a further read on the interesting history of Virtual Reality, see for example [1], [42], [49]. Nowadays VR is part of the umbrella term "eXtended Reality" (XR), which further includes Augmented Reality (AR) and Mixed Reality (MR) technologies (see Figure 2.1). Whereas VR offers full immersion in a completely virtual environment, AR adds virtual elements to the real world, to be viewed using a camera that combines real world images and virtual ones into the same view. MR further combines virtual and real artefacts by allowing them to interact with each other.

The recent influx of interest in Virtual Reality in academia, commercial and consumer markets came around the the introduction of Facebook's Oculus Rift, the HTC Vive and Sony Playstation VR headsets in 2012-2016. These headsets were developed for the consumer market and offered a high quality immersive experience at an affordable cost. Many more VR headsets were to follow, with one of the latest being the Valve Index VR kit released in June 2019. These headsets are all tethered: they require to be connected to a computer at all times to transmit visual data over a multi-Gbps cable and controller and tracking information over USB. However, in 2015 Google introduced Cardboard, a DIY headset that allowed users to place a capable smartphone in the headset and watch stereoscopic content through the headset's lenses. In 2016, Daydream View was released: a more mature and comfortable headset that adopted the same principle and came with a 3-Degrees-of-Freedom¹ (DoF) controller. With Daydream it is possible to not

¹In Virtual Reality, Degrees-of-Freedom (DoF) refers to the representation of the movement of real element (user head or controller) by an avatar in the VRE. 3-DoF means that the element can rotate around it's x-, y- and z-axis in the virtual environment. A 6-DoF element can further translate along these axis, giving a complete movement representation.

only watch stereoscopic content, but interact with the virtual environment as well through the controller. An updated version was released in 2017, which provided a 200° FoV as opposed to the earlier 90° [41]. Similarly, Samsung released their Gear VR headset in 2015, which supports all modern Samsung flagship Galaxy smartphones [46], [47]. An up-to-date overview of recent consumer VR headsets can be found at: [50].

Virtual Reality has many use cases, ranging from simulation, training and evaluation (architecture, design) to gaming and entertainment (omnidirectional video, theme parks, theaters). A recent report estimates that the global VR market will grow from \$2.02 billion in 2016 to \$26.89 billion in 2022 [52]. A particularly interesting development is the rise of 360° video and smart device VR/AR capabilities. YouTube and Facebook, two of the largest social media platforms, offer omnidirectional video that can be watched on any device, but particularly by using a VR headset. A clear benefit of using mobile devices for VR experiences is that the device is not attached to a computer, giving the users much more freedom to move around. Although this is not a factor in 360° video per se, the limited "play area" of tethered devices is a major downside in current VR systems and can easily lead to injury (e.g. by tripping over a cable). However, using mobile devices for VR content often requires the content to be rendered locally at low quality or streamed over a network connection, which is not a trivial task given the massive bandwidth requirements and latency restrictions.

2.2 Virtual Reality Streaming

Without going into the many (specialized) applications of VR, there are a couple of different types of devices and use cases in VR. We're assuming the term VR as used in this work is closer to what we know as the popularized VR: using an HMD to view stereoscopic images and interact with the virtual world. This then excludes the massive HMD's of old, transportation simulators, 3D television, etc. We can divide VR devices into three categories: tethered, standalone and smartphone-based. Tethered devices include popular HMD's such as the Oculus Rift, the HTC Vive and Valve Index. These are characterized by their need to be connected to a computer by cable to receive images and transmit play control and tracking data. One of the main benefits of tethered devices is that due to the high available bandwidth and very low latencies, very high quality VR content can be shown in a way that offers the highest QoE in VR to date.

Standalone devices are relatively new, and include for example the Lenovo Mirage Solo [43], which is a standalone headset based on Google's Daydream,

or the Oculus Quest and Oculus Go [38], [39]. Smartphone based solutions rely on a platform-based service to enable the viewing of content, such as Google Daydream or Samsung Gear VR, and a separate headset with lenses and optionally a controller. Standalone devices are, in simple terms, just dedicated smart devices: they use a mobile chip and have most of the features and connectivity of a smartphone, but with dedicated controllers and displays. This means that standalone devices are also fully mobile, and generally offer a better experience than smartphone based solutions due to for example a more comfortable headset. However, unlike tethered solutions, they can only display very simple graphics due to the limited processing power of the mobile chipset. Smartphone-based VR suffers from the same problem of course, but has the benefit that the display may be of a higher resolution, that smartphones tend to evolve faster than standalone VR headsets in terms of hardware, and that most people nowadays already have a smartphone.

In VR streaming there are two main types of applications: streaming a video such as a 360° video and streaming interactive VR such as a video game. Hybrids exist as well, with minimal interaction in the 360° video (e.g. choosing a new view point in the VRE). Streaming video often relies on pre-rendered content that is either computer generated or comes from a so called 360° camera. In continuation of this work we will refer to streaming 360° video with no or minimal interaction as "360 video" or "VR video streaming" and to VR streaming with full interaction between the user and the virtual world as "Interactive VR". For a more detailed introduction VR video streaming see e.g. [17]. Interactive VR is closely related to Cloud Gaming: in cloud gaming a remote server reads the input from the user on a client device, updates the world state, renders the output and streams this to the client device. The client device then decodes the output, renders it on screen and sends the player controls back to the server to update the world state. For some examples, see: [8], [9], [24], [29]. There are some key differences, however: the visual content of a VRE is generally shaped like a sphere, providing a spherical video. In order to encode such a video this sphere needs to be first projected onto a rectangular texture. Furthermore, the much larger resolution of VR content and the even more stringent latency requirements in VR streaming require specialized software and often specialized hardware. In order to provide a good QoE in an Interactive VR streaming system, [35] posit there are three key requirements: low latency (between 10-25ms) [7], high-quality visual effects (high detail, high FPS [15]), and mobility.

Streaming Virtual Reality relies on the same principles as tethered VR: rendering the virtual environment and displaying it on a client device (HMD). However, instead of the data transfer taking place by cable between the

HMD and the rendering machine, we send a video to the client device over a network connection. This does require us to make some modifications to the VR system. For instance: we cannot just take the raw output from the GPU and send it directly to a client: the video file would be massive and no existing consumer network connection would be fast enough to handle a single frame within the required time frame for delivery. Instead, the video texture is first encoded and then streamed to the client. Second, because encoders are designed for rectangular video formats, and a VR video frame is spherical, we need a way to map the sphere to a rectangular surface. Lastly, the client needs to be able to decode the video file and project the chosen mapping back onto a sphere and distort the output to match the lens parameters of the HMD.

Compared to tethered VR, streaming VR offers a number of benefits of which three stand out: first, untethered VR allows the user full freedom of movement. With wireless connection technology and 3- or 6-DoF tracking built-in any area can be a VR area. This also reduces the dangers of tripping over cables or smashing into your living room TV during a virtual boxing match. Second, By streaming VR content to a thin client we enable very high quality content on devices that would otherwise be incapable of displaying such a level of visual quality. Third, as smartphones evolve and their DPI increases rapidly, they may not suffer from the "screen door effect" as much as older, tethered headsets.

Of course, there are also downsides. For example: to avoid cybersickness, a low system latency is very important (this goes for tethered systems as well). While tethered systems nowadays are able to easily provide such low latencies, mobile devices and network connections in Streaming VR still cannot meet this requirement. [5] reports that cybersickness may affect 60-80% of users. Cybersickness is the apparent feeling similar to motion sickness, which, among others, includes dizziness, nausea, disorientation, sweating or even epileptic seizures. Cybersickness is often caused by discrepancies between a movement in the physical world and the related effect in the virtual world. For example: lag between a movement and the move being visible [5], [12], [22], [27]. Another downside is battery consumption on mobile devices, and although streaming may allow for higher quality, mobile devices are still struggling with very high resolution video (4K+). Tethered solutions may also offer higher refresh rates and higher levels of comfort, depending on the headset.

[35] discusses the possibility of streaming high quality VR to mobile devices, and note that on today's hardware it is unlikely to be successful: they posit that the QoE level of streaming to mobile devices is about 10x lower than acceptable QoE. Furthermore, they predict that newer hardware may

not necessarily be the solution, as higher data rates offered by new network technologies are likely unable to be fully utilized by the still power consumption-bound mobile devices. This means that in order to provide streaming VR to mobile devices with high QoE, smart software solutions are required.

2.3 Head Orientation and Projection Mapping

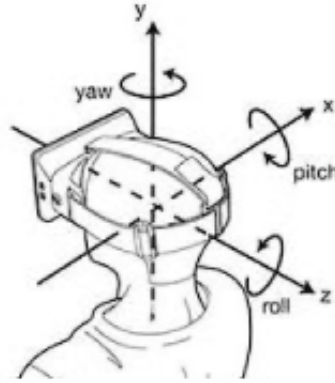


Figure 2.2: A user's head orientation in a 3-DoF system. Horizontal movements (yaw) rotate around the y-axis, Vertical movements (pitch) rotate around the x-axis and Sideways movements (roll) rotate around the z-axis. The origin is generally inside the head, but this may depend on the system. The axis system also represents the different viewing directions, with +z being "forward", -z being "backwards", +y being "right", -y being "left", +x being "up" and -x being "down". Each axis has a range of $[0^\circ, 360^\circ)$. Image adapted from [14].

In any VR system, the rendering device receives the positional information of the head, and optionally that of the controller, from the client device. This includes orientation and position information, as well as button presses or touches in the case of controllers. The orientation is generally represented as a Quaternion, but is more easily visualized and discussed in terms of Euler angles. In Figure 2.2 the axis system with Euler angles is depicted for a user wearing an HMD. Horizontal movements (yaw) rotate around the y-axis, vertical movements (pitch) rotate around the x-axis and sideways movements (roll) rotate around the z-axis. The origin is generally inside the head, but

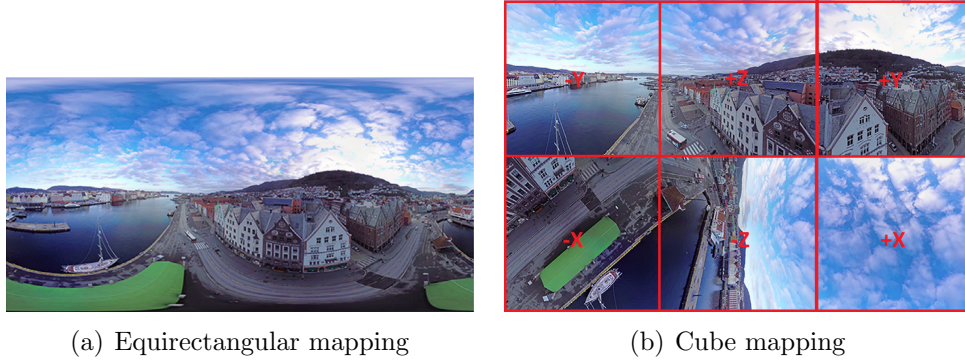


Figure 2.3: (a) An Equirectangular projection mapping. (b) A Cube projection mapping in 3x2 layout with the directional axis overlayed. Some faces are rotated according to JVET recommendations. Image adapted from [26].

this may depend on the system. The axis system also represents the different viewing directions, with $+z$ being "forward", $-z$ being "backwards", $+y$ being "right", $-y$ being "left", $+x$ being "up" and $-x$ being "down". Each axis has a range of $[0^\circ, 360^\circ)$. When receiving the positional information from the client, the rendering device updates the cameras that represent the head/eyes of the user in the virtual world to match the user's position and orientation.

The question of what projection mapping is best for VR streaming (in order to map a sphere to a rectangular surface) is an area of active and ongoing research. At the moment, there is no answer as to what method is best: different authors have invented different methods and each work well in a particular way. However, there are some common methods that are often used nowadays: Equirectangular and Cubemap projection mappings are the most common (see Figure 2.3). In equirectangular projection the vertical direction of the surface corresponds to the latitude of the sphere, and the horizontal direction corresponds to the longitude. Cubemap projection works by assuming a cube circumscribes the sphere, and each point on the sphere is then mapped to the corresponding point on the cube face behind the sphere point. The cubemap projection is natively supported by graphics libraries such as OpenGL, and is thus commonly used in VR games and other computer-generated imagery. Cubemap projection also does not suffer from shape distortion and has less redundant pixels compared to Equirectangular projection [26]. Figure 2.3(a) shows an example of an Equirectangular mapping and Figure 2.3(b) shows an example of the cube mapping. The layout in Figure 2.3(b) is the recommended layout by the Joint Video Experts Team (JVET), however a 4x3 layout is also commonly used. Many more

projection methods exist nowadays. For example: Facebook uses a pyramid shape mapping for their 360° video streaming, which devotes more pixels to the viewport. For an excellent discussion of different projection methods, see [26].

2.4 Related work: 360° Video Streaming

In streaming VR video, one of the main concerns is how to limit bandwidth consumption while maintaining high video quality. Since a user is only viewing a portion of the video at a given time, we can exploit that to provide higher quality in the viewport region while limiting the quality of the peripheral regions. This idea is the core of all current work on optimizing VR video bandwidth consumption, but it requires careful fine-tuning of the used algorithm to make sure quality changes are transparent to the user and the quality level changes accordingly when a user changes their gaze. In [16], the authors develop a viewport-adaptive system for streaming 360° video that prepares multiple representations of Quality Emphasis Centres (QEC's). These representations differ in quality level and bitrate to be able to use higher quality/bitrate video for the viewport and lower for the peripheral area. The video is cut into segments with different QEC locations and for different quality levels. The client then chooses the QEC closest to the gaze location and the quality level befitting its available bandwidth. Their system is somewhat related to ours in that they use an adaptive algorithm to determine when to switch to a different QEC or different quality level. However, their solutions relies on many pre-encoded segments that are stored on a server, which makes it impossible to use this method in real-time streaming systems. Furthermore, their algorithm was trained using a dataset of head movements. The Jaunt Inc. dataset is widely used, but it is generated from users watching 360° video which may comprise very different head movement behaviour compared to Interactive VR, or even different videos. In a follow up paper, [21] augmented the system to include a model of the optimal Quality Emphasis Region quality level distribution. This model was again generated from head traces of users watching a 360° video. In a simplified experiment, they reported up to 45% bandwidth savings, given 4 quality-variable video versions and perfect head movement prediction.

The work of [16], [21] nicely illustrates the main strategies for bandwidth savings in 360° video at the moment: cut a video up into pieces, use some algorithm to match these pieces to the user's viewport, and have multiple quality versions available of each piece to dynamically adjust the quality of different regions. For example, [19] analyzed a different dataset of over

1300 head traces to determine common gaze locations. They then propose a multicast DASH-based solution that streams a higher quality tile based on the likelihood of that tile being in the viewport. Similarly, [23] used these strategies to optimize bandwidth consumption, focusing explicitly on the DASH streaming algorithm and modern encoders such as H.265 and VP9. In [28] the authors realize the multitude of different optimizations for streaming 360° video are becoming increasingly hard to compare due to the lack of standardized tests, and propose a tool to compare their effect on QoE for different optimizations, viewports, encoding and streaming algorithms. In the context of using head trace datasets for determining the most important regions in a VR video, several datasets have been developed. Apart from the ones just mentioned, see also for example [25], [34]. Gaze and saliency analysis may provide us with more useful information of how people control their gaze in a spherical video. For some examples, see [31], [33], [34].

2.5 Related work: Real-time Interactive VR Streaming

The 360° video streaming solutions we discussed above rely on some form of offline representation of the video, usually in the form of multiple tiles with multiple quality representations. Not only does this require more storage space on the server side, but it also cannot be applied to fully live, on-demand streaming of VR content. In this section we discuss some of the works that are designed to optimize live VR streaming, such as live streaming of 360° video and streaming of Interactive VR, such as cloud VR gaming.

In [14] the authors developed a system to predict head movements and, based on the predicted head movement, transmit only the part that corresponds to the predicted viewport location. Using a machine learning algorithm trained on a dataset collected from users watching 360° videos, they were able to accurately predict the new viewpoint on a timescale of 100-500ms. They transmit a circular area for the predicted new viewpoint with a buffer circular area around it sized based on how confident the algorithm is. The authors report a 0.1% failure ratio and up to 45% bandwidth savings. Their results are promising, but based on 360° videos and they do not mention how the partial rendering was implemented. This work is in a way similar to ours, as they exploit the fact that users are only viewing a portion of the VR sphere at a time and stream only that part including a buffer area. They do note that a circular area may not be ideal in terms of bandwidth savings, and do not compare their solution to other head movement

predictions algorithms.

In [32] an untethered VR streaming system is proposed that supports up to 4K resolution with a latency of 20ms. The system consists of multiple encoders and decoders working in parallel, supported by a 60Ghz wireless link. They analyse the component latency of the system and particularly develop a V-Sync driven version of the system to overcome the latency introduced by missing V-Sync calls. Their work serves as an inspiration for the technical implementation of a VR streaming system and is one of the few that actively tackles the issue of latency in VR streaming and the high FPS requirement. It should be noted, however, that they point out that their system used a 60Ghz wireless link which is not commonly available and limits the mobility of the user. Furthermore, they used a laptop for receiving and decoding the video. Although they do not specify the model, the average laptop computer is much more powerful than a smartphone, which is used in this work.

FURION is a system designed by [35] to support Interactive VR streaming to mobile devices. By offloading part of the rendering (background) to a remote server and using the mobile device for foreground rendering and interaction, they manage to achieve a 14, 1 and 12ms latency to controller, rotation and movement interaction respectively. Their system is designed with the same type of mobile device in mind as our work: smartphones such as the Google Pixel or Samsung Galaxy S series combined with household connectivity such as 802.11ac Wi-Fi. Furthermore, it is developed based on Unity3D and Google Daydream and can be relatively easily added to existing content as a plugin. Unlike our VR streaming system, they pre-fetch the entire view for close-by gridpoints, whereas our work and [32] render just the required view on-demand. A key difference between [35] and our work presented here on dynamically rendering and streaming a VRE is that they do not employ any viewport-aware rendering strategies such as ours and thus do not have to consider head movement and latency compensation.

In [36] the authors present a VR streaming system that dynamically adapts a buffer area to compensate for system latencies. Their idea is similar to ours, although they use different methods and posit that it is not necessary to explicitly compensate for head movements. Further similar to our work is their assumption of some VR streaming system either in the cloud or in a Mobile Edge Cloud that streams content to a thin client. The authors present an extensive latency analysis in their work, and show that their adaptive margin area calculations are based on a similar rolling average RTT approach. However, it is unclear how the authors compensate for different head movements. Currently their calculations are based on a head movement dataset and they evaluate their system's performance by streaming 360° video. Compared to their work we explicitly focus on live user head movements and emphasize the

need for reliable head orientation prediction. Furthermore our work focuses on Interactive VR streaming such as gaming, and while [36] supports this in theory, it is not discussed any further in their work. Due to their focus on LTE networks and promising results, combined with our results presented in this work, it seems that this approach of adaptively rendering the user FoV while compensating for network conditions and system latency is a promising way of enabling Interactive VR streaming with high QoE.

Our work is in part based on the work done by [30], who developed **CloudVR**, an Interactive VR system designed to enable VR streaming using cloud-based server instances that stream to thin clients. In particular, the prototype VR streaming system used in this work is based on the prototype system in [30]. Lastly, **Flashback** [15] and **MoVR** [18] further present VR streaming systems that aim to provide high-quality untethered Interactive VR experiences.

Chapter 3

System Architecture

In this chapter we will describe the VR streaming system that is used in the current work. Although the system itself, its development or optimization are not the focus of this work, it will be beneficial to know a VR streaming system works in general. The key part of this work, our solution to the problem stated in Chapter 1, is described in Chapter 4. A working prototype of the VR streaming system described here has been developed in the CloudXR project and is largely based on the CloudVR system presented in [30]. It is within this prototype that we have applied our Dynamic Viewport-Adaptive Rendering solution.

Our Virtual Reality streaming system (henceforth referred to as "System") is implemented as a plugin for a game engine that acts as the server and an application for a mobile client. The server plugin (henceforth "Server") can be implemented during application development in game engines such as Unity3D and Unreal Engine. The client app (henceforth "Client") is a standalone application that connects to a compatible server and renders the video received from the server on the client display while sending user data to the server. The System was designed with cloudification in mind: the plugin can easily be integrated into existing and new projects and facilitates running the entire application in a cloud environment. Furthermore, the system was designed primarily with mobile clients in mind: efficient software and fast hardware on the server side minimize the system latency and require less bandwidth, crucial for streaming to a low-power mobile device. The System utilizes a novel partial cubemap layout to emphasize the quality of the front face of the VR video sphere, while the front face is always kept in the user's viewport. This means that we can dynamically change what part of the VRE to render based on where the user is looking, without to need to render the full 360° FoV at all times.

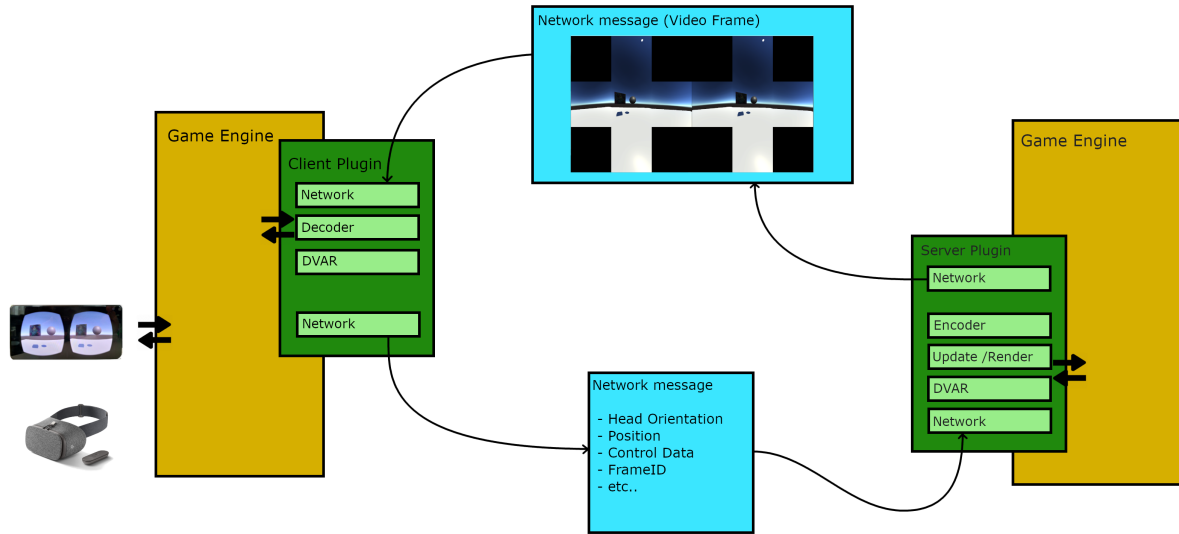


Figure 3.1: Diagram of a typical processing loop in the VR streaming system. Our solution is provided as plugins (green boxes) to an existing game engine. The blue boxes represent data being transmitted over a network connection. The client plugin retrieves data from the client device (left side) through an API such as Google Daydream (not pictured).

3.1 Server

On the server side, a typical render cycle¹ (for one video frame) looks like this:

1. **Receive orientation, position and control data from the client device.** Depending on the implementation this data may have been received earlier or slightly later: in any case the server uses the most recent data to render the new frame when the render cycle is started. The Server sets the cameras to the received orientation and position, so that the VRE on the server matches the user orientation and virtual position on the client. The control data is then used to advance the game logic and update the world state according to the application specifics. Control data includes, but is not limited to: button presses,

¹By render cycle we mean one full iteration in the game engine logic that updates the world state and renders the virtual world to an output, including encoding. Typically one such iteration is done for each video frame.

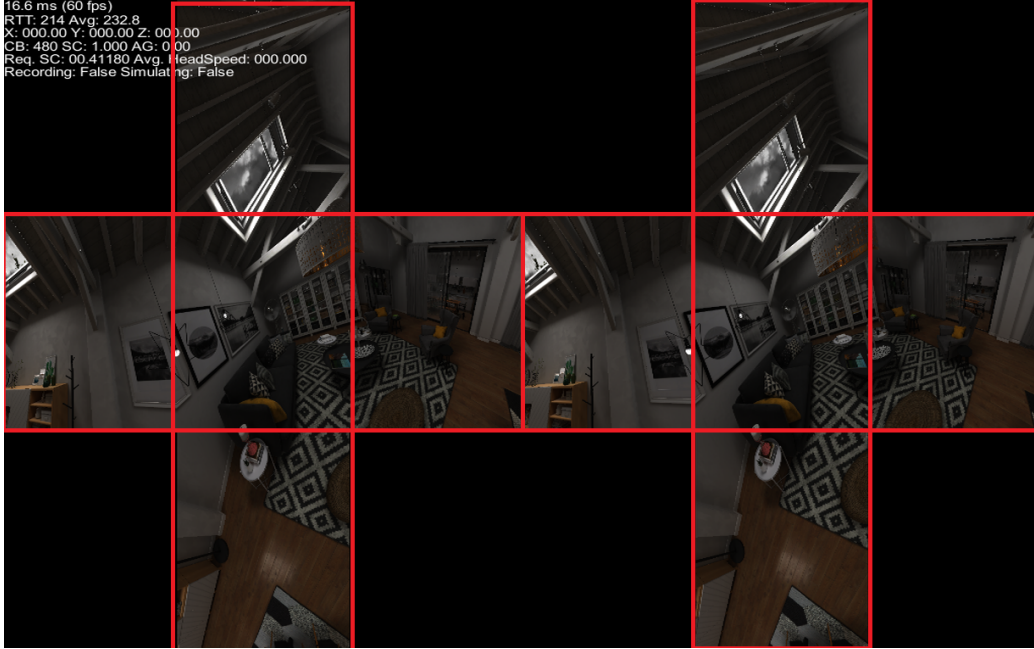


Figure 3.2: The 3x3 cubemap layout used in our system with left and right eye content. The middle rectangle for each eye is the front face of the cube (+z). In this particular example we are rendering all of the peripheral area except for the back face (-z). The cube faces may look slightly distorted in the figure due to rendering the output to a desktop monitor, but in practice each face is square and of equal size.

controller orientation and position, local game object parameters, video frame and system parameters.

2. **Render the virtual scene, using the cameras, to a temporary texture.** The System uses 5 cameras per eye to render the virtual world to the output texture. Each camera has a 90° field of view, and each camera is aligned to a directional axis. Combining the output of all cameras (typically 6), would give us the 360° FoV required. In this case, however, we render using only 5 cameras, ignoring the -z (rear) direction providing a $270^\circ \times 180^\circ$ FoV.
3. **Combine the camera textures into a single rectangular texture for encoding.** The System utilizes a partial cubemap projection method to combine the camera textures into a single texture. In this case, we render the +x and -x, +y and -y and +z axis (vertical, horizontal and frontal views), but not the -z direction (rear direction). This

has a number of benefits: for one, it lowers the amount of rendering needed to compile a full video frame. Second, it allows us to use a 3x3 layout for the partial cubemap, where the front face (+z) is in the middle, adorned by the horizontal (y) and vertical (x) direction faces (See Figure 3.2). This will be crucially important for dynamically scaling the faces to devote more space to the front face in the video texture, which is discussed in Chapter 4.

4. **Encode the texture into a video frame.** The video texture is essentially no different than from what would be rendered on screen in a normal game, or displayed on a tethered HMD. However, because of its large size and the need for streaming it to a client, we need to encode it. The Server uses the FBCapture SDK [40] to capture the rendered texture, and pass it to a hardware-accelerated encoder. In the meantime, the Server can start with its next render cycle.
5. **Stream the video frame to the client, together with control data.** The resulting encoded video frame can then be passed to the network stack and sent to the client. The control data includes, but is not limited to: the orientation that the video frame is based on, as well as video parameters. We use a TCP socket connection to ensure the correct order of the frames being delivered.

3.2 Client

The client application utilizes the HMD manufacturer’s SDK (e.g. Google Daydream, Oculus Quest) to display a received video frame on the device’s display, in such a way to match the layout and properties of the headset’s lenses. Furthermore, the SDK provides API access to the different XR properties of the nodes, such as the head orientation, eye position, controller position, etc., and matches the real-world orientation to the corresponding VRE orientation, so that the user can look and move around in the VRE by physically moving. Optionally, the Client can locally render some game objects and perform part of the game logic locally. The current System utilizes such a hybrid approach, where the controller for example is displayed in the VRE using local rendering [30]. The Client consists of a standalone application (in this case based on Unity3D) and our own native Android decoder plugin. A typical render cycle looks like this:

1. **Receive a video frame and control data from the server.** The decoder plugin is multi-threaded and takes care of receiving NAL units

and control data from the server. The received NALU's and control data are fed to a buffer for processing. The network connection is based on a TCP/IP socket to ensure correct order of both control data and video frames.

2. **Decode the received video frame.** The decoder is based on Google Android's MediaCodec API. It works asynchronously by waiting for a decoder inputbuffer to become available and then feeding it the next NAL unit. When the buffer is processed the output is rendered to an internal texture that can be accessed by the higher-level Client application.
3. **Process the game logic and update the world state.** In case of local rendering and interaction events, the world state is updated accordingly. Furthermore, the new control data (e.g. user orientation, frame ID, performance metrics) are prepared.
4. **Update the shader parameters and render the video frame to the display.** The client displays a spherical video by projecting the received imagery to the virtual environment's skybox. A modified shader provides support for stereoscopic panoramic rendering to the skybox. In order to be able to modify the content outside of the user's viewport without the user noticing, the skybox is oriented along the orientation of the video frame which is extracted from the received control data. This means that a user is always facing the front face (+z direction) of the cubemap/sphere. The received video frame is in rectangular format comprising a non-spherical layout of the faces. The cubemap faces in the video frame are moved to a separate texture for each eye in the common 4x3 cubemap layout, which the shader natively supports. With the shader being updated to the new orientation and having references to the rearranged input textures, the content is rendered to the Client device's display. The HMD's SDK provides the warping of the image to match the lens parameters.
5. **Send the current player control data to the server.** The client sends the server the control data needed to render the next frame. Note that this step does not necessarily happen at the end of the render cycle.

Chapter 4

Dynamic Viewport-Adaptive Rendering (DVAR)

Normally when streaming a VR video frame, each viewing direction is represented by the same amount of pixels. In case of streaming a cubemap layout, each face has an equal amount of pixels, and the six faces together offer 360° of visual content. Notwithstanding technologies that alter the resolution of the video frame, such as adaptive resolution used in video streaming, the size in pixels of a video frame is fixed at a given time. This means that all cube faces are proportionally the same size. However, a user is only looking in one direction at a time, with a limited FoV (generally between $90^\circ - 110^\circ$). As we discovered in the Background section, most companies and researchers have discovered this as well and use it to non-uniformly assign better quality factors to the part of the scene the user is looking at (the viewport). This has been done using frame tiling for example, where tiles in the viewport area of a video frame are given higher quality than the surrounding areas. Many more solutions exist, of course, but the consensus seems to be that streaming all areas of a VR video frame in equal quality is a waste of valuable resources.

In this work we mostly deal with square cube faces and square video frames for a given eye. This means that the height and width of an area will be equal. Because of this we employ a shorthand notation unless otherwise noted: an area with a resolution of 2048×2048 pixels (for example) will be referred to as an area of 2048 pixels.

4.1 Idea

Given a streaming system with a low enough latency, and given that we can orient the frontal direction with the users viewing direction, we can assume

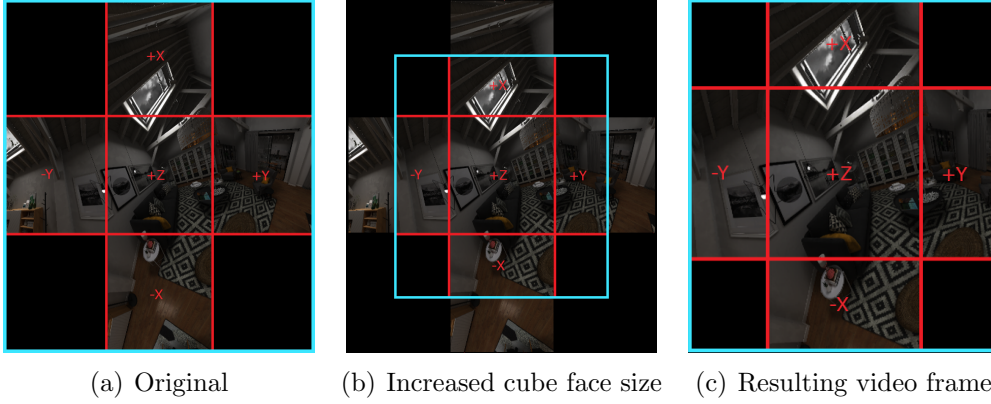


Figure 4.1: Transforming a 3x3 cubemap layout to proportionally increase the size of the front face (+z). In (a) the original proportions are displayed with their axis directions. Each cube face is rendered 100%. The blue border represents the video frame boundaries for one eye. In (b) we have increased the size of all faces, such that the side faces do not fit into the video frame anymore. The video frame size remains the same. In (c), we have adapted the amount of side face to render, such that all faces fit into the video frame, resulting in a proportionally larger area for the front face.

that a user will only be viewing the frontal direction for the majority of the time and any changes in orientation are quickly compensated for in the next video frame. This assumption allows us to ignore the rear direction of the content sphere, because by the time the user will have reached that orientation, a new video frame will have already placed the frontal direction in the way of the previously rear direction. This then, allows us to use the partial cubemap layout described earlier, in which we only utilize 5 front faces of a cubemap and ignore the back face. Our contribution is utilizing this layout to proportionally enlarge the frontal cube face while rendering only part of the side faces (top, bottom, left, right) in their respective direction. This way we end up with a video frame that offers proportionally more pixels to the front face.

For an example, see Figure 4.1: (a) represents the normal layout, with each cubemap face having an equal proportion and equal amount of pixels. Because we disregard the back face, we only need to place the other 5 faces on the video frame in a plus-shape layout (3x3). This square video frame represents one eye, and the other eye is generated correspondingly. Now, we can devote more pixels to the front face by enlarging all cubemap faces to have a desired resolution. However, given that the video frame resolution stays the same, parts of the side faces will now not fit within the frame and

need to either be compressed or cropped as in (b). Instead, our solution is to simply only render the part of each side face that fits within the video frame given the larger front face. We end up with the video frame as in (c), where the front face now contains $\frac{2}{3}$ of the pixels, and we only render $\frac{1}{6}$ of each side face. The resulting video frame offers more pixels in the front face, but limits the available field-of-view in the VRE since only part of the left, right, top and bottom faces are rendered. The non-rendered areas, including the back face, simply appear as black pixels. We can arbitrarily choose the amount of pixels (i.e. the proportion) for the front face by adjusting the amount of side face to render, but there are some caveats related to the user's head movement: when the user moves his head and the new frame (with the "recentered" front face) takes too long to be displayed, the user will see the unrendered area. This is obviously highly detrimental for the user's Quality of Experience, so in the next section we will discuss how to prevent this.

First, however, we'll have a look at how this is implemented in practice, so as to gain a better understanding of the exact workings. Combined with the optimization process described in Section 4.3, we have dubbed this method of rendering DVAR, for Dynamic Viewport-Aware Rendering. The optimizations below will allow us to dynamically adjust the amount of side face to render, based on the current system parameters such as latency, so as to maximize the front face resolution while maintaining good QoE. This method is viewport-adaptive in that we rotate the cube/sphere's front face to the user's viewport every time and because we devote a maximum amount of pixels to the viewport given the system and optimization constraints. Lastly, we do not simply crop or cut the remainder of the side faces that do not fit on the video frame, but rather do not render those areas at all on the Server. Thus DVAR also positively affects rendering times and avoids unnecessarily using resources.

4.2 Implementation

On the server side, the Server plugin has a module that calculates the desired amount of side face to render at every render cycle. From here on out we will refer to the amount of side face to render as the side face percentage ($SF\%$), and the front face resolution and proportional size automatically follow from the side face percentage: if we render 10% side face, and have a face on each side of the front face, the front face occupies 80% of the video frame. In case of a 2048x2048 pixel video frame for one eye, this amounts to 1638x1638 pixels for the front face, 1638x204 pixels for the top

and bottom faces, and 204x1638 pixels for the left and right faces. At each frame, the module determines whether the side face percentage should be changed, based on the system performance and user head movement velocity (see below, Section 4.3).

If the side face percentage is changed, the new resolution for the side and front faces is calculated and helper variables updated. Because the destination texture of each camera, and thus each direction, still all have the same resolution, we can render only part of a direction by modifying the camera projection matrix to obtain the desired side face resolution and content. When the $SF\%$ is changed, the camera projection matrix for each side face camera is adjusted so that it only renders a percentage of its 90° FoV, starting from the edge connected to the front face. DVAR is not content-dependent, so we only need to calculate the new values once, and update the cameras to obtain a new video frame with the new layout. Modifying the projection matrix is essentially the same as cropping the output texture, before rendering. This way we avoid distortion that might arise from either enlarging the front camera FoV or compressing the side face cameras. The rendered areas are then moved to the video texture in the original plus-shape layout, which gives us Figure 4 (c). The video texture is then processed as it would be normally and sent to the encoder. From the encoder we then send the encoded video frame to the client, together with the front face resolution and the $SF\%$.

The client now receives a video frame that is not in any natively supported layout. When the video frame is decoded, the client also knows the front face size in pixels and the $SF\%$. This can be used to calculate the size in pixels of the side faces, since the original resolution of each face is equal, i.e. $50\% \cdot SideFace == 50\% \cdot FrontFace$. This can then be used to determine the location of the faces in the video frame and copy them to a destination texture that is in the 4x3 layout that is supported by the shader.

4.3 Optimization

Now that we have the ability to render and stream any $SF\%$ we need, we need to determine how much to render at any given moment. Recall that the system loops works as follows: the client device sends the head orientation to the server, where the camera position is adjusted accordingly and the new image rendered, encoded and sent back to the client with the sent orientation where it is rendered on screen. Due to the inherent latency in the system and the speed with which users move their heads, it may be possible for the user to observe a non-rendered part of the video before it is updated with

the new content. This is visible as blackness, and it's generally replaced by the proper visual content in the next frame update provided the system is responsive enough. However, a user observing the non-rendered part will have clear implications for their QoE, so we need to be able to render just enough of the side faces that the entire experience of moving one's head to a new orientation is transparent to the user (i.e. the user does not observe any non-rendered areas). The question of how much of the side faces to render, then, is the topic of this section.

4.3.1 Compensating for System Latency

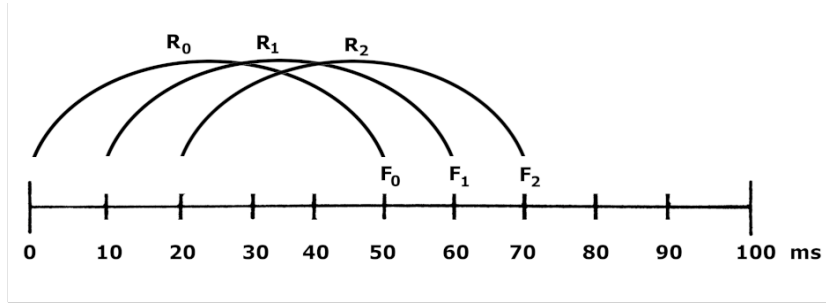


Figure 4.2: Assuming a system RTT of 50ms, a requested frame F_n takes 50ms to be rendered on the client. R_n is the head orientation sent by the client to the server and horizontal axis marks represent time $t = 0..100$ milliseconds.

Observe Figure 4.2. At a given time $t = 0$, the client sends the current head R_0 to the server to render a frame F_0 . Assume the system RTT is a constant 50ms. It thus takes up to 50ms for F_0 to be rendered on the client screen. If the user is moving its head at a constant speed of $50^\circ/\text{second}$, this means that at $t = 50$ the user will have moved 2.5° . Assume further that the client updates the view (i.e. renders a new frame) every 10ms. At $t = 10$ the head rotation R_1 is sent to the server for frame F_1 , which arrives at $t = 60$. If the user's FoV_{HMD} is 90° , the requested frame F_0 should have the HMD's FoV plus the amount the user has moved. Furthermore, at $t = 50$ the viewport will be at the edge of F_0 's FoV, while the next frame still takes 10ms to be rendered. We thus also need to add the user's displacement in those 10ms. Finally, this means that F_0 should have an FoV_{frame} of $90^\circ + 2.5^\circ + 0.5^\circ = 93^\circ$. With this the user's viewport will be at the edge of the frame's FoV at $t = 60$ and the user will not observe any unrendered area. At $t = 60$ the next frame, F_1 , is rendered, providing an extra buffer area of the difference between R_0 and R_1 (i.e. 0.5°), meaning the viewport will again

be at the edge of F_1 's FoV at $t = 70$. It is clear from the example above that if a frame fails to provide an FoV that includes the movement since the request of that frame, the user will observe unrendered area ("blackness") due to the fact that we only render the requested FoV.

If the system RRT drops, incoming frames for the new and lower RTT may arrive at the same time or before an earlier frame for a higher RTT. This means that a frame arriving in between render cycles may get overwritten by a newer frame, essentially skipping the frame. Assume that at $t = 10$ the RTT drops to 45ms, and at $t = 20$ the RTT is 40ms. This means that F_1 will arrive at $t = 55$, but will be skipped since there is no render call at that time. It will get overwritten by F_2 arriving at $t = 60$. F_2 has an FoV of 92° plus 0.5° for the frametime compensation. F_2 is then based on the orientation at $t = 20$, which is 1° from the starting point. At $t = 60$ the head orientation is 3° from the starting point, meaning F_2 has to compensate for $FoV_{HMD} + \Delta R$, which is 92° . Given the calculation above we find that F_2 has an FoV of 92.5° , so the user will not observe any blackness.

If the RTT increases some blackness may be observed: assume that at $t = 10$ the RTT increases to 70ms, causing F_1 to arrive at $t = 80$. At $t = 70$, F_0 will still be rendered, which has an FoV_{frame} of 93° , while the head orientation at $t = 70$ requires 93.5° . At $t = 80$ F_1 is rendered with an FoV of 94° , which removes the unrendered area from the user's viewport. This means that for roughly 10ms, 0.5° of unrendered area will be visible, which in practice is invisible to the user. Of course, with larger changes in RTT these effects will be more pronounced, but a change in RTT of more than 20ms within the time of a single frame is in practice highly uncommon.

Consider another example, presented in Figure 4.3. This top-down view represents either the calculation for the y-axis (horizontal direction) or the x-axis (vertical direction), which are equivalent in this case. The blue lines represent the full FoV in the video frame FoV_{frame} . In this case we are rendering $\frac{1}{4}$ of the side faces, giving us an FoV_{frame} of 135° . At R_1 (green lines) the user's head orientation has moved left 10° , causing the edge of the user's viewport with $FoV_{HMD} = 90^\circ$ to be within the left edge of FoV_{frame} . Thus, at R_1 the user will not see any unrendered area. If, however, the head orientation were at R_2 (red lines), having moved left 40° , the user would be able to see 17.5° of unrendered area.

Given the system specifications that any cube face represents a 90° FoV we need to render the displacement as a factor of $FoV_{render} = 90^\circ$ in the displacement direction to fill the viewport. Since we need to compensate for this movement at $t = 0$, we need to know the angular displacement in degrees per millisecond for $\Delta R = t_i - t_{i-1}$ (RTT; the time before the requested frame

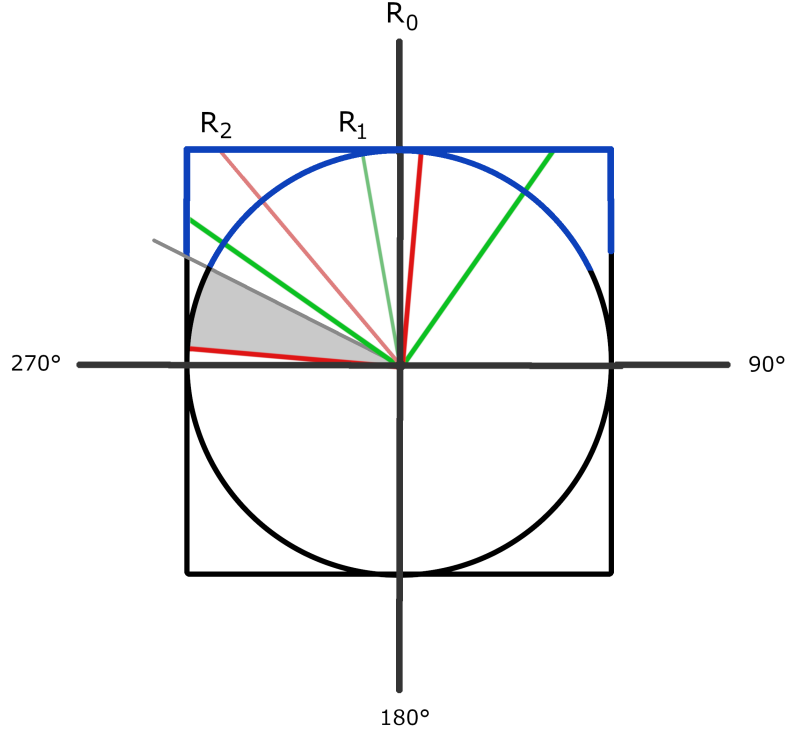


Figure 4.3: A top-down view of the video sphere with circumscribing cube and user head orientation with a 90° FoV. R_0 is pointing in the $+z$ direction. The blue lines indicate the rendered area: in this example we are rendering 25% of each side face, providing an FoV_{frame} of 135° . R_1 represents a head orientation of 10° to the left compared to R_0 , where the user does not observe unrendered area. R_2 however represents a rotation of 40° to the left, causing the user to observe 17.5° of unrendered area (grey area/line).

arrives). This gives us the following equation:

$$s_R = \frac{(f + l) \cdot v}{FoV_{render}} + c \quad (4.1)$$

Where s_R is the required side face percentage, f is the frametime on the client side in ms (i.e. maximum time before the new frame arrives), l is the system RTT in ms, v is the head movement velocity in degrees per ms and FoV_{render} is the cube face FoV in degrees. In other words: the numerator in Equation 4.1 represents how many degrees the user may move in the time to the new requested frame, and the denominator represents the FoV for a cube face.

In practice, the render camera field of view of 90 degrees does not always equal the FoV of a modern HMD ($100^\circ - 120^\circ$), so we add the compensation factor c to render at least the full HMD FoV:

$$c = \frac{FoV_{HMD} - FoV_{render}}{2 \cdot FoV_{render}} \quad (4.2)$$

Since we render all side faces equivalently, the resulting FoV_{frame} is:

$$FoV_{frame} = 2 \cdot s_R \cdot FoV_{render} + FoV_{render} \quad (4.3)$$

For a given eye, we have half the video frame as the width and the video frame height available in pixels. Furthermore, each cube face has an equal amount of pixels. To determine the proportion of the front face in pixels given the amount of side face to render, we need to determine the proportion of the front face in the video frame given the required $SF\%$:

$$r = \frac{m}{2s_R + 1} \quad (4.4)$$

Where m is the resolution of the video frame for one eye (shorthand applies here backwards: 2048x2048 pixels for a video frame means $m = 2048$), s_R is the required $SF\%$ from Equation 4.1 and r is the resolution of the front face in pixels.

4.3.2 Head Movement

The head movement velocity is more challenging to compensate for: although we can record the user's head movement velocity during a movement, as well as the average angular velocity, we don't know this just before the movement starts. Thus we do not know how much velocity to compensate for (term v in Equation 4.1). However, there are some possible solutions:

1. always compensate for some average angular velocity based on a user's average head movement velocity during a session;
2. compensate for some arbitrary amount of angular velocity;
3. always compensate for the maximum head movement speed a user can produce;
4. predict the onset and angular velocity of a head movement before it begins.

It is clear from Equation 4.1 and 5.3 that the more angular velocity we compensate for, the higher the percentage of side face to render is, and thus the lower the resolution of the front face is. This means that option #3 is not ideal, since we want to maintain a high-as-possible front face resolution at all times. The ideal option would be #4, as it would allow the system to know the user’s head velocity during the RTT interval (or, analogously: predict the new head orientation after said interval). Implementing head movement prediction is not trivial, and currently out of the scope of this work. We might use a combination of options #1 and #2: tracking the average angular velocity of a user during a session, and use this as a baseline compensation factor, with a minimum. This means that if a user during a task moves their head generally slowly, the viewport resolution will be higher, while if the user moves their head generally quickly, the viewport resolution will be lower to accommodate the extra visual content rendered.

The average head movement velocity is a moving average measured over some time interval $t_{\bar{v}}$. The main disadvantage of this approach is that the system may take up to $t_{\bar{v}}$ time to adjust itself to the new angular velocity when a user switches from a series of slower movements to a faster one, and vice-versa. When suddenly going from slower to faster movements the user may observe non-rendered areas as the system is still adjusting. For the experiment in Chapter 5 we will be using a fixed, constant head movement velocity compensation, to provide comparable data and avoid the unreliability of using a naive rolling average. In future work this will be replaced by proper head orientation prediction such as done in [14] or [2].

We ran a number of tests that confirmed that the calculation of the required side face percentage to compensate for latency and head movements works. However, the effectiveness crucially depends on the accurate measurement of the motion-to-photon latency and the head movement velocity in the RTT timespan. Because these variables cannot always be measured accurately, it might be well worth sacrificing a little resolution in order to increase the buffer zone.

There has been some interest in analyzing head movements of users in a VR context. However, all works we found were related to 360° video. Although the visual experience is very similar, 360° video offers no real interaction and generally has no task. Since in an interactive scenario such as gaming head movements are highly dependent on the task at hand, player goals, environment variables (saliency, audio, etc.), we cannot directly use these findings in this work. However, [19] found, after analyzing a dataset of 1300 head movement traces from 360° video, that small head movement changes are much more common than large ones. They found that 80% of angular changes were within $[-38.1^\circ, 36.2^\circ]$, $[-14.5^\circ, 15.2^\circ]$, and $[-5.3^\circ, 5.0^\circ]$

for yaw, pitch, and roll angles. This corresponds roughly to our own findings in Section 5, where users were instructed to explore a new VRE. [14] built a VR system roughly similar to ours, but focused on predicting head movements instead of compensating for system RTT. They used a dataset based on 360° videos to predict head movements on a scale of 100-500ms and achieved reasonable accuracy. Other such prediction solutions use for example double exponential smoothing ([2]) on a time scale of 50ms, or motion prediction with constant acceleration/velocity over 20-100ms ([10]). [20] noticed that within a segment duration of 2 s, 95% of the users move less than $\pi/2$ radians. This means that within a 2 s length segment, 95% of the user stay inside the hemisphere centered on the head position of the user at the beginning of the segment. They found this to be true for a video triggering with very few head movements (Roller-Coaster), a video triggering many head movements (Timelapse), and intermediary videos.

Given the work discussed here it seems feasible to implement a head orientation prediction system that would be accurate enough to complement DVAR, and work fast enough to not increase the system latency. However, as we discussed also in Chapter 2 and will see in Chapter 6, head movement characteristics are very personal and may depend highly on the task at hand. This means that apart from the performance considerations, a head movement prediction system should also be task and user agnostic to some degree.

Chapter 5

Experiment Setup

DVAR is designed to maximize the viewport resolution given the constraints of the current network connection and the user’s head movement velocity, thus using the available bandwidth more efficiently in order to provide the highest viewport resolution. In order to assert whether our solution works as intended, we run a series of measurements to determine the performance of DVAR in the current system.

Note that, although video quality and network delay are crucial factors in Quality of Experience in any streaming system, and DVAR aims at providing the best QoE, our current prototype system has some characteristics that make it impossible to evaluate the Quality of Experience directly in a meaningful way. For example, despite new metrics of video quality being developed, currently none are suitable for evaluating the effect of unrendered pixels in a viewport. Furthermore, the current technical limitations are likely to strongly influence a user’s experience during subjective tests, which we are unable to control at this time. However, it is well established that if all other factors remain unchanged, a higher resolution provides a higher video quality and thus a higher Quality of Experience (e.g. [4], [6], [11]). Following this reasoning, we do not assess the QoE as such, but merely assess the increase in resolution due to the use of DVAR. This seems reasonable, as DVAR does not concern encoding or network design choices and these are better covered in other work.

5.1 Measurements

As measurements of DVAR’s performance, we record the following dependent variables:

1. Unsuccessful Frames (UF)

2. Unrendered Area (UA) in an Unsuccessful Frame
3. Viewport resolution

The measurements described here are adapted from [14], who applied these in evaluating a similar system. Our system differs from theirs in some significant ways, and their focus was on evaluating the effect of predicting head movements, instead of compensating for network conditions and optimizing bandwidth usage. As such our results are not directly comparable, but they serve the same purpose in assessing the viewport content’s properties. By Unsuccessful Frame we mean a frame that has some visible amount of unrendered area, and by Unrendered Area in a frame we mean the visible unrendered area in degrees that a user will be able to see in a given frame. The viewport resolution is simply the average amount of pixels the user will observe through the HMD for a given frame.

The experiment consists of emulating a head trace recorded from 5 different users, who are exploring a virtual environment (described in more detail below). The system functions as it would normally, with the exception that real-life head orientations are replaced by the recorded orientations. This way we are able to have the experiment represent real-life use case, while still controlling the head movement velocities of different users. During a trial we record the measurements described in this section. We keep the encoding and streaming settings constant during all trials, while streaming at 30 FPS. We set up DVAR to use a rolling average of the RTT based on the last 60 frames as the latency input. Both server and client are set to render at 60FPS, due to technical limitations. DVAR was configured to use a discrete interval for the side face percentages from 0% to 100% in steps of 5. The closest value to the calculated percentage is chosen for rendering each frame. This can lead to the used value being up to 2.5% off the real value, or 2.25 degrees. We further configured DVAR to use a head movement compensation level of 127 degrees per second. This number is partly based on the results found by [3], who noted that their user’s maximum head movement speed was 382° per second, but 127° per second during natural head movement in a sound localization task.

We end up with 5 unique parameterizations for the 5 users. Each parameterization is played out in the emulation for the duration of the recorded 30 seconds of real interaction. During each trial, we record the dependent variables described below. The unrendered area is recorded before the new frame is rendered based on the current frame and current head orientation, after which we render the new frame and calculate the viewport resolution. Each parameterization is emulated 10 times for each of the 4 conditions (3

references and 1 DVAR) which gives us a total of 200 trials. I.e. we emulate User 1's head trace 10 times using DVAR and 10 times for each of the References 1 through 3. The results from the trials are then combined across users and averaged where applicable to eliminate any minor fluctuations in rendering and network performance. Trials with a Standard Deviation of more than 2x that of the other trials are discarded as outliers. For the unrendered area measurements we maintained a 3° error margin: although the system calculates the required amount of side face to render with high precision, this can cause minute fluctuations in cube face resolution that can make the prototype system unstable. Furthermore, our pilot tests showed that users see unrendered area around 3 degrees later than calculated due to the distortion and quality of the Daydream headset.

5.1.1 Unsuccessful Frames

An unsuccessful frame is defined as a video frame that contains unrendered pixels, which are visible to a user as a black area/black pixel. The percentage of unsuccessful frames is calculated as the fraction of total individual unsuccessful frames rendered on the client display to the total amount of frames displayed on the client display. For this measurement we ignore the size of the unrendered area. This measurement gives us a quick-and-easy measure of how well DVAR performs: the best case scenario is a UF percentage of 0%, while 100% means all frames contained some visible unrendered pixels. The calculation of Unsuccessful Frames and Unrendered Area are closely related, so we discuss the calculation below.

5.1.2 Unrendered Area

The unrendered area is the amount of unrendered pixels in an unsuccessful video frame measured in degrees. At render time of a received frame we know the current head orientation, the orientation of the current video frame, the FoV represented by the frame (FoV_{frame}), and the FoV of the user through the HMD (FoV_{HMD}). We can then calculate how much of the unrendered area is or is not visible in the viewport at that time. Given a side face of the cubemap, with a percentage s rendered, the FoV of that side face is the percentage of the FoV of the entire cube face, which is 90° ($FoV_{render} = 90^\circ$). The total FoV of a given video frame in terms of rendered area is then twice the rendered percentage of a side face plus the FoV of the front face:

$$\begin{aligned}
FoV_{frame} &= s \cdot FoV_{render} \cdot 2 + FoV_{render} \\
FoV_{frame} &= s \cdot 90 \cdot 2 + 90 \\
&= 180s + 90
\end{aligned} \tag{5.1}$$

We then calculate the unrendered area factor P as follows:

$$P = \frac{FoV_{frame}}{2} - \frac{FoV_{display}}{2} - \Delta R + \frac{FoV_{display} - FoV_{HMD}}{2} \tag{5.2}$$

Where ΔR is the displacement between the current frame orientation and the current head orientation in degrees, $FoV_{display}$ is the field of view with which the visual content for a given eye is rendered on screen as calculated from the camera projection matrix¹, and FoV_{HMD} is the field-of-view of the HMD, as reported by the manufacturer.

Equation 5.2 applies to a clockwise turn of the head with a displacement of less than 180° . In order to calculate P for a counterclockwise turn we simply subtract ΔR from 360° first. If any unrendered area would be visible, the sign of P would be negative and its number would represent the amount of degrees of unrendered area visible in the viewport. If P is positive it shows how much rendered area in degrees we have left in this direction before unrendered area would come into the viewport ("buffer area"). This application of the equations above is equivalent for calculating the unrendered area for either the vertical or horizontal direction. Note however that these only apply to a 3-DoF system, or at least a system where the Origin of the frame orientation and camera orientation are equal.

The last term in Equation 5.2 represents the offset between the display's FoV for a given eye and the FoV of the HMD for a given eye. It is important to note here that what is visible to a user greatly depends on the quality and parameters of the client hardware used by the user. For example, Daydream renders the edges of a viewport with some distortion, and renders the display content with a horizontal field of view of 96.35° . The Daydream headset offers only a 90° FoV, meaning the user could have a different experience from the predicted viewport if the predicted viewport content is calculated based on the display contents.

¹The horizontal and vertical fields of view often differ in stereoscopic rendering. In the equations above we refer to the $FoV_{display}$ as either the vertical or horizontal field of view, depending on whether we're calculating the unrendered area in a respective direction.

Source	Name	Description
Client	FT	Frame Time in ms since the last frame was rendered
Client	DT	Decode Time in ms since the last frame was decoded
Server	RTT	System RTT in ms
Client	FPT	Frame Processing Time from receiving a frame to it having been decoded in ms
Client	NRT	NALU Receive Time since receiving the last frame from the network in ms

Table 5.2: The system performance measures we recorded during all trials. We focus on client performance measures as the client device is currently the bottleneck in the prototype system.

5.1.3 Viewport Resolution

The viewport resolution of a given frame is simply the resolution of the front face plus the areas of the side faces in the viewport. Since the width and length in pixels of each face are equal, and each face represents a 90° FoV, the viewport resolution can be calculated as follows:

$$V = \frac{FoV_{HMD}}{90^\circ} \cdot v_{cf} \quad (5.3)$$

Where v_{cf} is the resolution of a cubemap face in pixels. The process is the same for calculating the vertical viewport pixels as well as the horizontal viewport pixels, with the application of the corresponding value of FoV_{frame} . Equation 5.3 does assume that there is no unrendered area visible in the given frame. If there is, we simply subtract the amount of unrendered area from FoV_{HMD} .

5.1.4 Bandwidth, Latencies and System Performance

As dependent variables we further record average bandwidth usage from the packets being streamed to the client, using Wireshark. To get an indication of system performance during the trials we furthermore record the variable in Table 5.2.

The system RTT is calculated in the following way: at each frame that is rendered on the server, this frame is assigned a unique identifier that is then stored on the server with a timestamp of the start of that frame's render cycle. The identifier is sent with the frame to the client. The client catches this identifier and, when the corresponding frame has been rendered,

sends the identifier back to the server. It is there matched to the recorded identifier and the difference between the current timestamp and the recorded timestamp results in the RTT measurement.

5.2 Materials & Apparatus

We used a Dell Precision 5820 workstation as a server. This desktop PC featured an Intel Xeon W-2133 CPU with an NVidia GeForce RTX 2060, using driver package 431.60 and Windows 10 Pro 64-bit fully updated as of 15-8-2019. The client was installed on a Google Pixel 3 XL smartphone, plugged in to a power source, and featured Android 9 at patch level August 1, 2019. All apps and software were fully updated as of 15-8-2019, with Daydream App version 1.20.190204006. The client and server software was built using Unity 2018.3.3f1, using latest Daydream SDK as of 15-8-2019. For the network connection the server was connected with a LAN gigabit connection to a dedicated Asus RT-AC66U B1 router to which the client was connected on the 5Ghz band using WLAN 802.11ac with a peak throughput of 866 Mbps. The headset was a Daydream View 2016. To capture the network packets we used Wireshark v3.0.3, and the server used FBCapture SDK v2.35 to capture and encode the output from Unity. We set up Wireshark to record all packets on the tethered connection to the router, which was disconnected from the internet and set up so that only the server and client have access.

5.3 Emulating Head Movement

5.3.1 Recording the Head Traces

We asked five employees in the Computer Science building at Aalto University to collect head traces. The server was set to render at 60 FPS and capture and stream at 30 FPS. The client always renders at the refresh rate of the display because of Android's limitations, which for the Pixel 3 XL results in 60 FPS. We used the ArchVizPro Interior Vol.6 Demo [45] as the virtual environment. In this demo a detailed, fully furnished house is shown. The demo offer high graphics quality and visual fidelity. We placed the scene camera at a fixed position inside the house, showing the living room and dinner table. After the system had initialized, we asked the participant to stand still while wearing the headset, we re-centered the view to the natural forward direction of the scene using the standard Daydream controls. Then we told the participant to start the task and started the recording. The task

was beforehand explained by asking the participant to look around naturally and freely in the scene while describing what they see. During the task we recorded the head orientation and a timestamp. The task ended after 30 seconds. This gave us 5 separate head trace files with head orientations at a 16.7ms interval, equal to the interval with which head orientation data is sent to the server normally.

5.3.2 Emulating the Head Traces

The recorded head trace was loaded at the start of the program. After the system has initialized, we place the device in a holder to maintain a steady orientation and re-center the view using the Daydream controls. The device's orientation now matches the starting orientation of the head traces. We start the experiment by reading each recorded head orientation from the file in memory at each render cycle, and replacing the use of the head orientation in the program on both the client and server side with the recorded orientation. The emulation ends automatically when the last head orientation is used and the system then records the dependent variables to disk.

5.4 Reference Conditions & Encoding Parameters

In Table 5.1 the different testing conditions and their main parameters are presented. R1 represents a high bandwidth, high resolution, full field of view setting that should always offer a comparatively high quality of experience. Other solutions may offer up to 16K video resolution, but our current prototype system is unable to handle such high resolution video. However, the results are easily extended to higher resolution references. R2 represents the system's worst-case scenario in which the entire field of view needs to be rendered, resulting in lower bandwidth usage but also a much lower viewport resolution. R3 represents a naive setting in which the viewport resolution is very high, but the rendered field of view is only equal to the HMD's field of view (90° for the Daydream View 2016 headset). This means that un-rendered areas will be easily visible except for very low system latencies. In R1 we stack both eyes on top of each other in the video frame, whereas in the other conditions the left and right eye are represented in their respective halves of the video frame (horizontal stacking). As a final note we remind the reader that because we do not render the $-z$ direction of the cube map and instead rotate the $+z$ face with the user's gaze, the $270^\circ \times 180^\circ$ is essentially equivalent to a full FoV as offered by a typical 360° video.

Condition	Video Resolution	Cube Face Res.	FoV	Layout
R1	4096x4096	1024	270° x 180°	3x2
R2	4096x2048	682	270° x 180°	3x3
R3	4096x2048	2048	90° x 90°	1x1
DVAR	4096x2048	Variable	Variable	3x3

Table 5.3: Conditions tested in the experiment with their main parameters. Video resolution refers to the resolution in pixels of one video frame containing content for both eyes. Cube Face Res. refers to the resolution in pixels of one cube face of the cubemap for one eye using the shorthand notation. FoV refers to the rendered area provided by one video frame one eye (FoV_{frame}). Layout refers to the layout of the cube faces in the video frame for one eye.

We used the standard encoding settings from the FBCapture SDK for live streaming. The FBCapture SDK used the NVEncoder SDK for encoding. We set the encoder to use the **Low Latency High Performance** preset with the h.264 codec, a QP of 28 and infinite GOP length resulting in a constant QP encoding setup. For more information, see the official NVEnc documentation: [44].

5.5 Procedure

1. Build an executable of the client and server with the required parameterization;
2. repeat for 10 trials:
 - (a) start Wireshark and prepare to record on the connection to the client;
 - (b) start the server software;
 - (c) start the client software;
 - (d) wait for the system to initialize, and another 5 seconds for the system to stabilize;
 - (e) place the client device in the holder;
 - (f) reset the orientation to the scene's natural forward using the Daydream controller;
 - (g) start the recording in Wireshark;
 - (h) start the emulation;

- (i) upon finishing, save the recording in Wireshark and close all software.
- 3. change the parameterization of the client and the server to the next;
- 4. repeat from step 1.

Chapter 6

Results

In the following we present the results from the experiment. The viewport resolution and bandwidth measurements are in the following section, after which we discuss the unrendered area/unsuccessful frames measurements for each user separately. Finally, we quickly discuss the results presented.

6.1 Overall Performance

6.1.1 Viewport Resolution

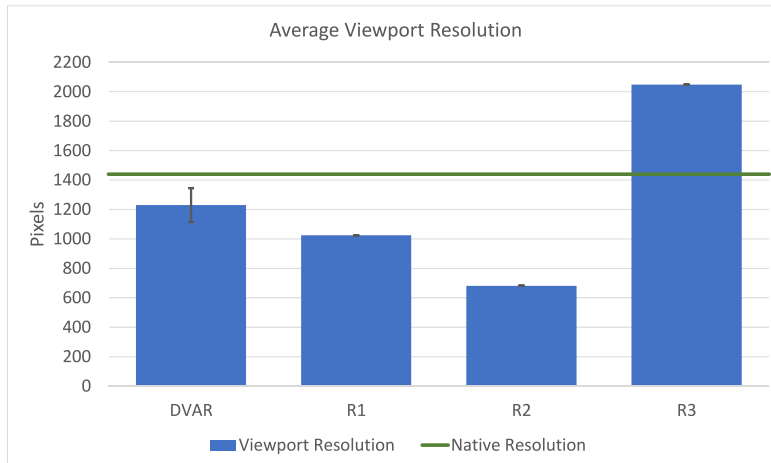


Figure 6.1: The average viewport resolution of DVAR and the three reference conditions. The green line represents the native resolution of the Pixel 3 XL for one eye (1440 pixels). The error lines on the left bar represent the average resolution ± 1 SD.

In Figure 6.1 the average viewport resolution is plotted for DVAR and the three reference conditions. The average viewport resolution is the resolution visible through the HMD with a 90° FoV. For R1 through R3 this resolution is fixed, while DVAR optimizes the viewport resolution based on the system RTT and head movement velocity. We can see that DVAR provides a viewport resolution close to the device's native display resolution for one eye (green horizontal line), with a mean of 1227 pixels and a mode of 1204 pixels. To get the average viewport resolution we took the mean of the viewport resolution of all users and all trials minus the outliers. R1, R2 and R3 provide a constant viewport resolution of 1024, 682 and 2048 pixels respectively.

6.1.2 Bandwidth

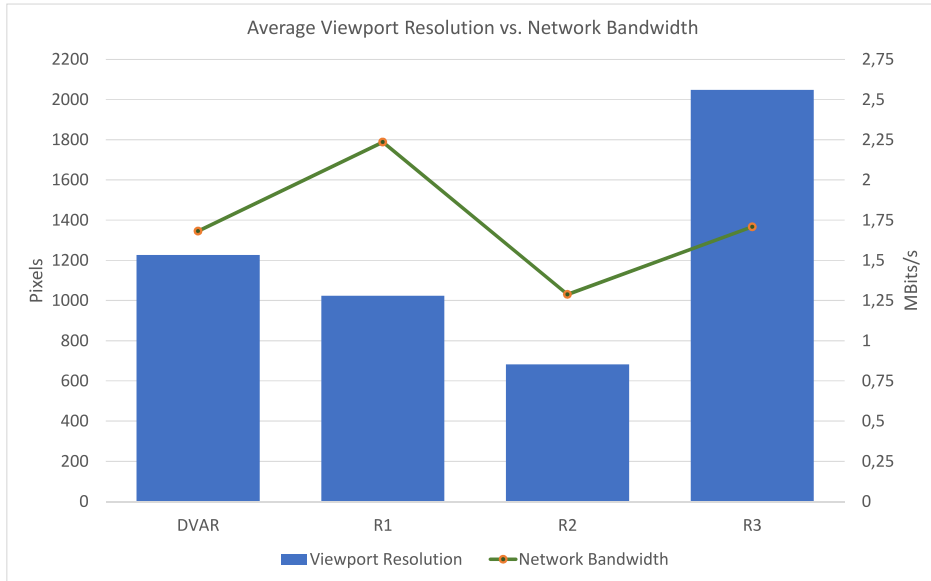


Figure 6.2: The average resolution as in Figure 6.1, combined with the network bandwidth used by streaming the video to the client in average megabits per second (right axis).

Figure 6.2 shows the average network bandwidth used by streaming the video from the server to the client in MBit/second. The average was obtained from all trials by all users minus the outliers. The resolution data is the same as in Figure 6.1. We can see that R1, which uses a 4096×4096 pixels video with 1024 pixels per cube face, requires the most bandwidth at an average of 2.24 ± 0.04 Mbit/s. R2 uses cube faces of only 682 pixels, and thus contains

a lot of black areas in the video. These black areas are likely to be encoded very efficiently, so R2 unsurprisingly requires the least bandwidth (1.29 ± 0.01 MBit/s). R3 uses the same 4096x2048 video resolution as DVAR, which is reflected in the results: DVAR used 1.68 ± 0.03 MBit/s on average. R3 only uses slightly more bandwidth (1.71 ± 0.02 MBit/s), likely due to the small black areas in DVAR's video which are efficiently encoded. As DVAR works by scaling the resolution of the front face, it's worst case scenario is equal to R2, with it's best case scenario equal to R3. We can thus expect DVAR's bandwidth usage to be between that of R2 and R3 in Figure 6.2.

6.1.3 Unsuccessful Frames & Effective Bandwidth

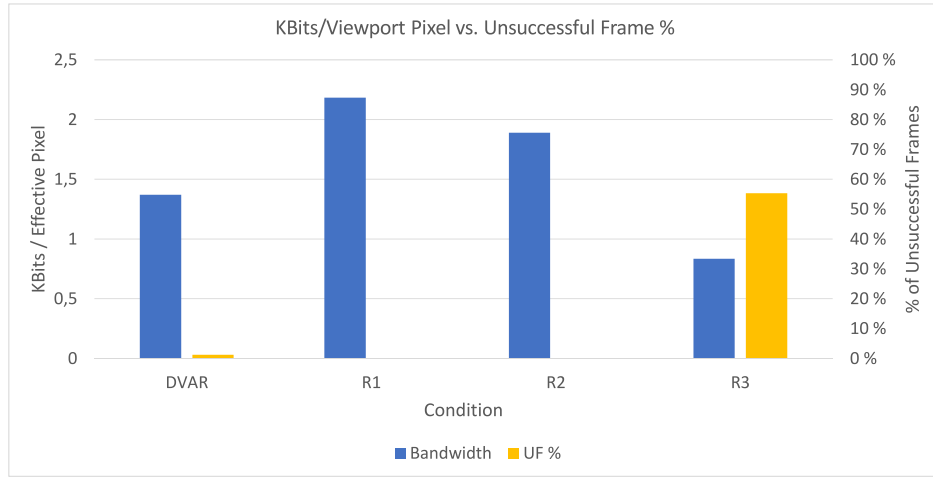


Figure 6.3: The bandwidth usage as average KBit/s per effective pixel (i.e. pixels in the viewport) for all of the conditions. The secondary axis represents the percentage of Unsuccessful Frames (frames containing more than 3° of unrendered area).

In Figure 6.3 we can see the average bandwidth usage per effective pixel, where an effective pixel is a pixel visible in the viewport. We can see that R1 and R2 have a much higher bandwidth usage as they stream the entire FoV and only a small area of that is visible in the viewport at a time. R3 uses it's bandwidth most efficiently, because we only stream the viewport. However, this leads to large amount of unrendered area being visible. Where DVAR has less than 1% of Unsuccessful Frames over all trials by all users, R3 has 55%. Furthermore, R3 actually did not have a single successful frame if we do not consider the 3° error margin. It then seems that DVAR offers a better trade-off in terms of Bandwidth/Effective Pixel and Unrendered Area.

	DVAR	R1	R2	R3
FT	33.02 ± 6.37	36.41 ± 6.67	33.02 ± 5.19	32.96 ± 5.73
DT	33.29 ± 6.02	36.76 ± 6.69	33.34 ± 4.86	33.33 ± 5.54
RTT	218.07 ± 36.31	3179.4 ± 615.9	197.90 ± 38.33	216.64 ± 37.13
FPT	50.96 ± 7.68	148.38 ± 5.83	50.08 ± 5.22	50.42 ± 6.07
NRT	33.32 ± 2.02	36.76 ± 6.67	33.33 ± 1.49	33.33 ± 2.50

Table 6.1: Average system latency measurements for User 1 in all conditions. FT = FrameTime, the time in milliseconds since the last frame was rendered on the client display. DT = DecodeTime, the time since the last decoder buffer became available (measure of decoding performance). RTT = Round-Trip Time, a measure of the total system RTT. FPT = Frame Processing Time, a measure of the time it takes from receiving a video frame to it being decoded. NRT = NALU Receive Time, the time since the last NALU packet was received. Since we’re streaming at 30 FPS the ideal value, except for RTT, is 33.33ms.

6.1.4 System Performance

In Table 4.1 we present the average values for the different system latencies we measured. All values are in milliseconds. It is clear from the data that DVAR does not significantly decrease or increase decoding, rendering or network performance. The slightly higher average RTT for DVAR compared to R2 can be explained by the fact that DVAR uses almost the entire video frame, thus increasing network load compared to R2 which contains around 44% black area. This is further supported by the fact that the RTT for DVAR and R3 are statistically equal, as they both use the entire video frame. Regarding R1: as we mentioned before the prototype system is not suitable for high resolution video at the moment and this shows in the measurements in Table 4.1. However, R1 is mainly used for the bandwidth calculations, where these values do not diminish its validity. Lastly, Table 4.1 only contains the results for User 1: we have omitted the results of the other users because the values for the other users present the same pattern and have no significant differences to User 1’s values.

6.2 Per-user Analysis

6.2.1 User 1

In Figure 6.4 the average system RTT as used by DVAR is plotted against the viewport resolution during the DVAR condition for User 1. This figure

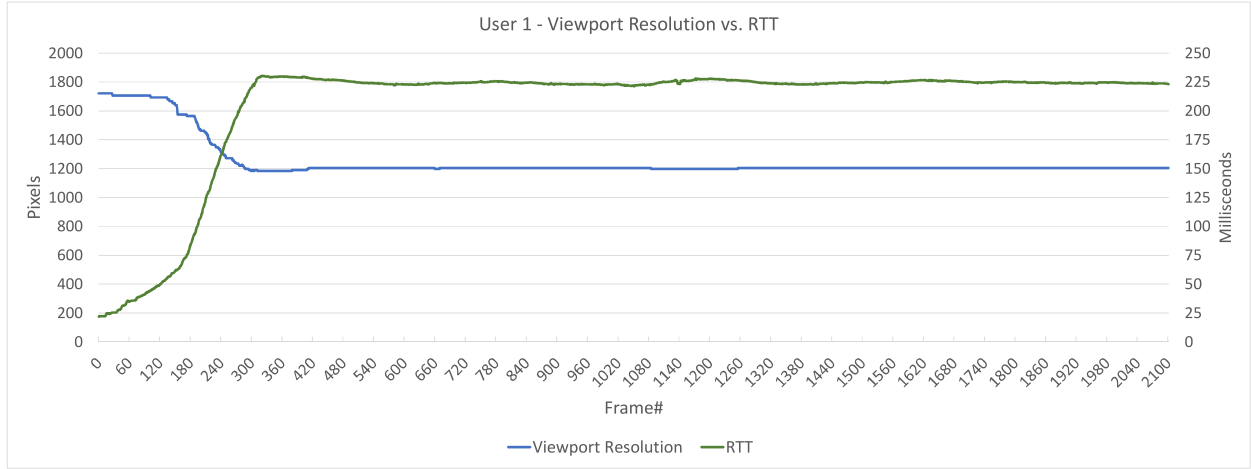


Figure 6.4: The average Round-Trip Time and viewport resolution of User 1 in the DVAR condition. The green line represents the RTT (right axis) in milliseconds, while the blue line represents the viewport resolution (left axis) in pixels. The resolution is for a single eye in either the width or the height, since the video area for one eye is square.

provides us with an overview of how DVAR responds to changing RTT values. We can see that early in the trails the system is adapting quickly to the increasing RTT, while during the remainder of the condition the RTT is relatively stable and thus also the viewport resolution. The distinct steps in resolution in the first couple of frames, as opposed to the smooth RTT increase, are due to the 21 discrete side face percentages we used. This figure shows us that DVAR works as expected with no surprises. In the other conditions, R1 - R3, the viewport resolution is fixed and does not react to changes in the network conditions.

Figure 6.5 shows the amount of unrendered area in degrees at a given time (frame) against User 1's head movement velocity in the DVAR condition. The yellow line represents the "blackness" threshold: values above this threshold will be visible as unrendered area (i.e. black pixels). Values under this threshold represent how much "visual buffer" is available, i.e. how much extra FoV is available in the video frame that is not visible to the user. The green lines represent the amount of unrendered area. The values in the x-axis are generally less than that in the y-axis, because users tend to move their head in the horizontal direction more, while both the horizontal and vertical direction get an equal amount of compensation in the current system. The red line represents the velocity compensation level. In general we expect blackness to be visible if the user moves their head faster than the velocity

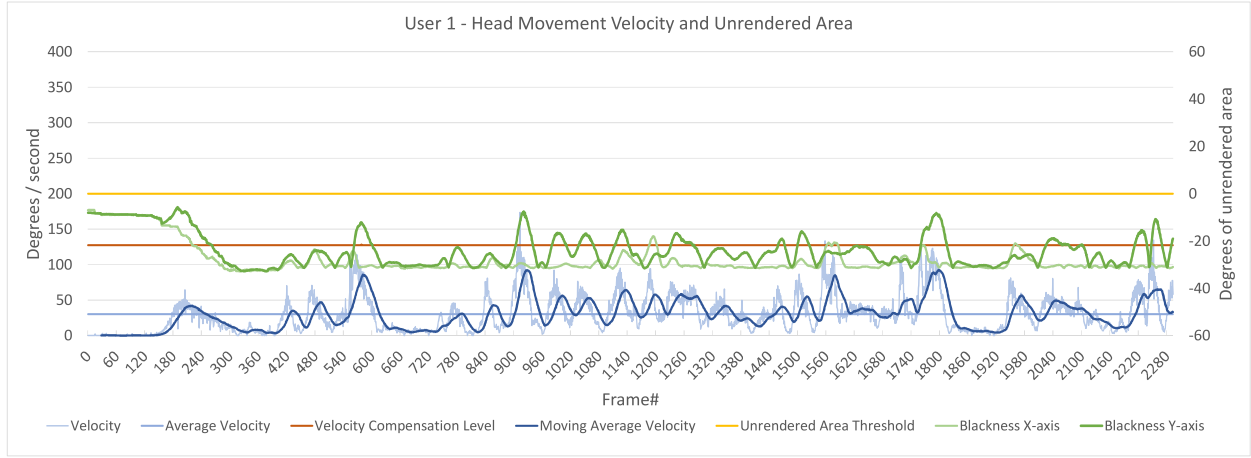


Figure 6.5: The unrendered area in both the y-axis and the x-axis, combined with the magnitude of User 1’s head velocity. The tested condition is DVAR. The blue lines represent the head velocity in degrees per second (left axis), while the green lines represent the amount of unrendered area in degrees in their respective axis. The yellow line is the threshold for when unrendered area becomes visible: blackness values above the threshold are visible, while values under the threshold represent the amount of ”visual buffer” that is still available for the given velocity compensation level and the current RTT.

compensation level.

User 1 had a relatively low overall average head movement velocity of $30.17^\circ \pm 25.62^\circ$ per second, and if we count individual peaks above the average line in the moving average velocity, User 1 has 22 movements. The moving average (over the last 30 frames) shows that separate movements are also relatively slow with all peak velocities staying well under 100° per second. This means that DVAR shouldn’t have any problem compensating for User 1’s head movements, given the compensation level of $127^\circ/s$. The data in Figure 6.5 confirms this, with the blackness level (the amount of unrendered area) in both the X- and Y-axis staying well under the threshold. Consequently, User 1 does not have any Unsuccessful Frames, i.e. 0%. The mean amount of FoV that is not visible (i.e. ”visual buffer”) is $23.86^\circ \pm 6.93^\circ$ for the y-axis, and $27.99^\circ \pm 6.14^\circ$ for the x-axis.

6.2.2 User 2

Figure 6.6 shows the viewport resolution during User 2’s trials in the DVAR condition versus the RTT in those trials. There are no surprises here, as the pattern is very similar to User 1. At this point the reader may notice

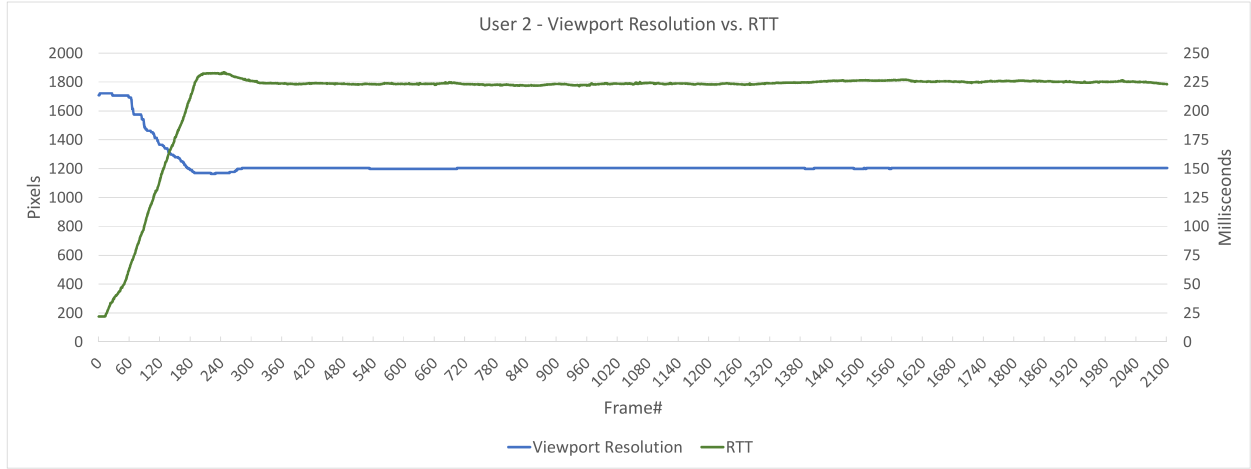


Figure 6.6: The average Round-Trip Time and viewport resolution of User 2 in the DVAR condition. The green line represents the RTT (right axis) in milliseconds, while the blue line represents the viewport resolution (left axis) in pixels.

that the Resolution vs. RTT figures all feature a steep increase in RTT and thus a decrease in viewport resolution in roughly the first 200 frames. This is because the user starts the trial from a stationary moment, where the orientation and position are the same. This increases the overall system efficiency and thus causes the RTT to decrease. As soon as the user starts moving, the RTT quickly jumps back to "normal", however, because we are measuring a rolling average here, it takes a couple of frames to stabilize. For the purposes of analyzing the unrendered area and head movement velocity, the reader may ignore the values for the first 200 frames. In the bandwidth sections these values do not represent normal usage, but do show nicely how DVAR reacts to changing RTT values.

In Figure 6.7 User 2's head movement velocity and unrendered area are plotted. The properties of this figure are similar to Figure 6.5. User 2 has the aforementioned early blackness peak due to incorrect RTT information and a strong early movement, but in the remainder of the session there is no unrendered area. User 2 has an average head velocity of $28.37^\circ \pm 29.04^\circ$ per second, and 15 movements. Similarly to User 1, User 2 has no unrendered area if we do not count the start-up peak at Frame # 70. We feel confident excluding this peak because a very similar movement occurs at 960 frames, where there is no unrendered area. Again the data shows that DVAR performs well in compensating for the system latency and different head movements. The visual buffer is $23.86^\circ \pm 6.93^\circ$ for the y-axis and $27.99^\circ \pm 6.14^\circ$

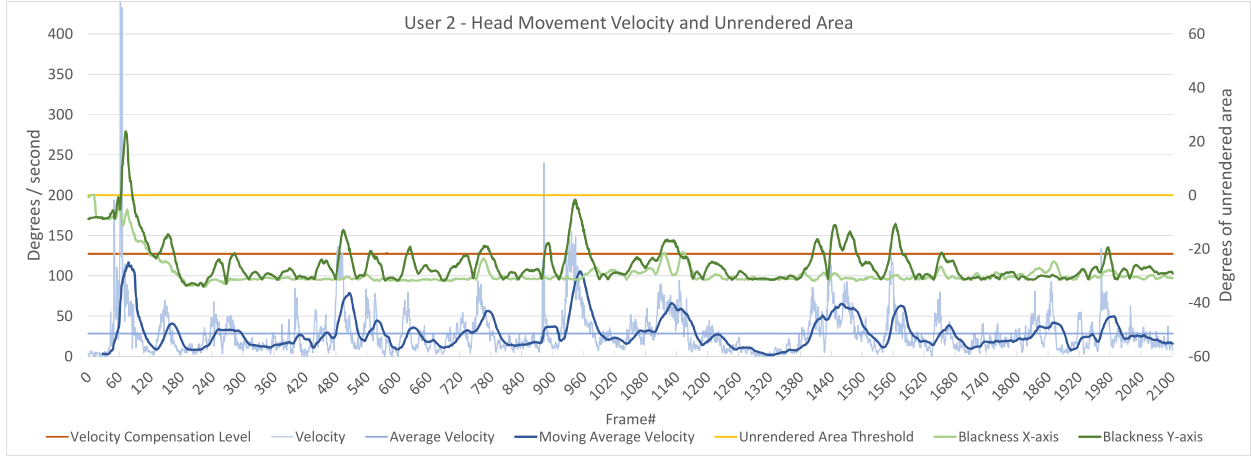


Figure 6.7: The unrendered area in both the y-axis and the x-axis, combined with the magnitude of User 2’s head velocity. The tested condition is DVAR. The blue lines represent the head velocity in degrees per second (left axis), while the green lines represent the amount of unrendered area in degrees in their respective axis. The yellow line is the threshold for when unrendered area becomes visible: blackness values above the threshold are visible, while values under the threshold represent the amount of ”visual buffer” that is still available for the given velocity compensation level and the current RTT.

for the x-axis. Note that from these numbers it seems that the amount of (un)rendered area is more stable than the actual head velocity. This is partly explained by the noisy head velocity data. When we look at the moving average velocity values and the blackness values, we see that the amount of unrendered area is closely related to the head movement velocity, as expected.

6.2.3 User 3

Figure 6.8 shows User 3’s viewport resolution and RTT during the DVAR session. The network conditions are slightly less stable here, so we can see that the resolution changes with slight increases/decreases in RTT. Otherwise this figure shows no surprises and is very similar to Figures 6.4 and 6.6.

In Figure 6.9 User 3’s head movement velocity and unrendered area are plotted. The properties of this figure are identical to Figures 6.5 and 6.7. Due to the lack of a strong movement in the beginning, the start-up phase does not produce any unrendered area this time. User 3 has 19 relatively slow movements, with a couple even slower ones (small peaks below the average line). The average head velocity is $23.84^\circ \pm 23.16^\circ$ per second which is the lowest of all users. In this case the velocity compensation level of $127^\circ/s$

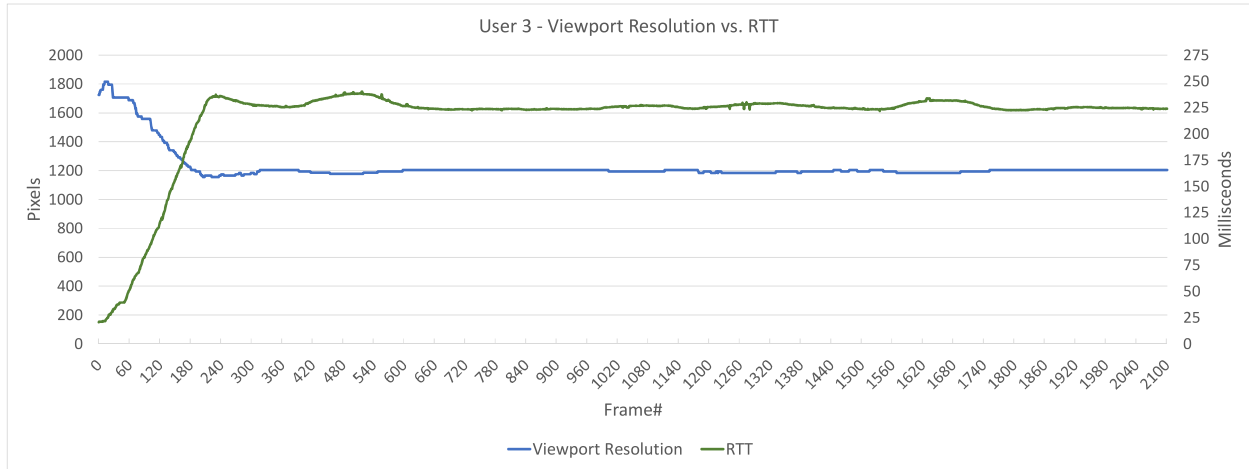


Figure 6.8: The average Round-Trip Time and viewport resolution of User 3 in the DVAR condition. The green line represents the RTT (right axis) in milliseconds, while the blue line represents the viewport resolution (left axis) in pixels.

might be overkill, but we can still see some movements coming within 10° of the threshold. Another interesting thing to note here is that User 3 has many small movements, as opposed to the fewer but larger movements seen earlier and in the following users. This highlights again that head movement patterns are inherently personal and predicting them requires taking into account personal differences in behaviour even within the same scene and same task. The average "visual buffer" is $26.62^\circ \pm 6.25^\circ$ for the y-axis and $29.46^\circ \pm 5.20^\circ$ for the x-axis.

6.2.4 User 4

Figure 6.10 shows User 4's viewport resolution and RTT during the DVAR session. We see a slight dip in RTT around 350 frames, which might not have been compensated properly in terms of resolution scaling as we saw previously. However, the lowest point in the dip is only 212 ms, for which 1200 pixels viewport resolution is still more than adequate. If we take a sneak peek at Figure 6.11, we see that this was a moment of User 4's slowest head movements, which could explain the dip as the system efficiency increases when dealing with similar frames and orientations. At the end of the session we see a sudden increase in RTT, of which the cause is unknown. However, DVAR responds as expected, even given the noisy dips. Remember that these values are (rolling) averages, which means that these discrepancies are

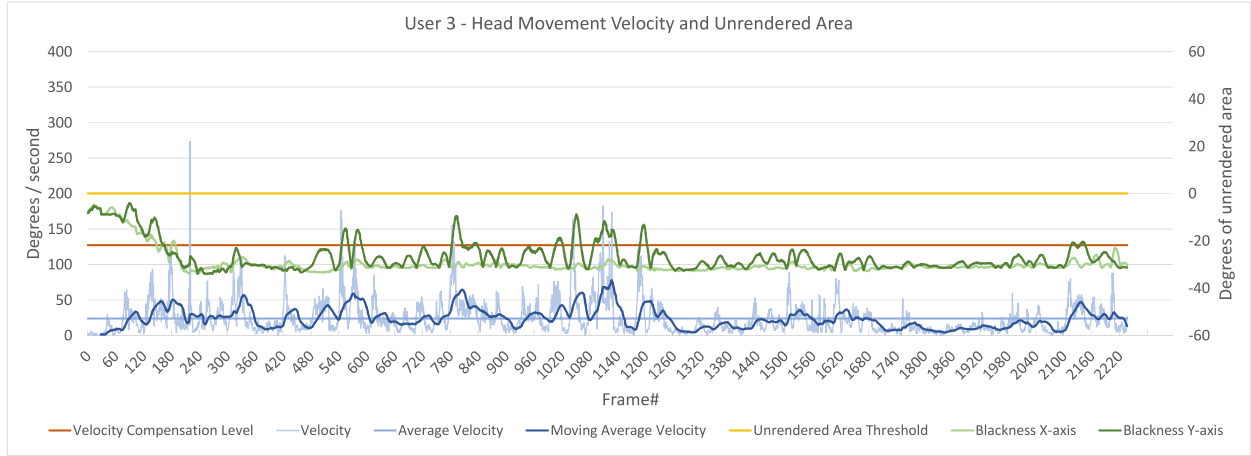


Figure 6.9: The unrendered area in both the y-axis and the x-axis, combined with the magnitude of User 3’s head velocity. The tested condition is DVAR. The blue lines represent the head velocity in degrees per second (left axis), while the green lines represent the amount of unrendered area in degrees in their respective axis. The yellow line is the threshold for when unrendered area becomes visible: blackness values above the threshold are visible, while values under the threshold represent the amount of ”visual buffer” that is still available for the given velocity compensation level and the current RTT.

not merely coincidences but were present in all trials.

User 4’s head movement velocity and unrendered area values are then plotted in Figure 6.11. This figure is more interesting than the previous ones, as User 4’s head movements cause it to show unrendered area at around 1920 frames. The amount of unrendered area is only present in the y-axis (horizontal direction) and reaches 19.76° of unrendered area. The average unrendered area was $12.47^\circ \pm 4.81^\circ$ and lasted for 33 frames. The cause of this unrendered area is easily determined, as the head velocity goes well over the velocity compensation level and the RTT remains steady. The average head velocity of User 4 was relatively higher than the previous users and had higher peaks at $33.59^\circ \pm 35.91^\circ$ per second and 20 movements. The average amount of visual buffer (counting the blackness peak) was $23.72^\circ \pm 9.05^\circ$ for the y-axis and $28.23^\circ \pm 6.10^\circ$ for the x-axis. The similar averages confirm that User 4’s head movements were still well under the velocity compensation level for the majority of the session, while the slightly higher SD corresponds to the stronger movements.

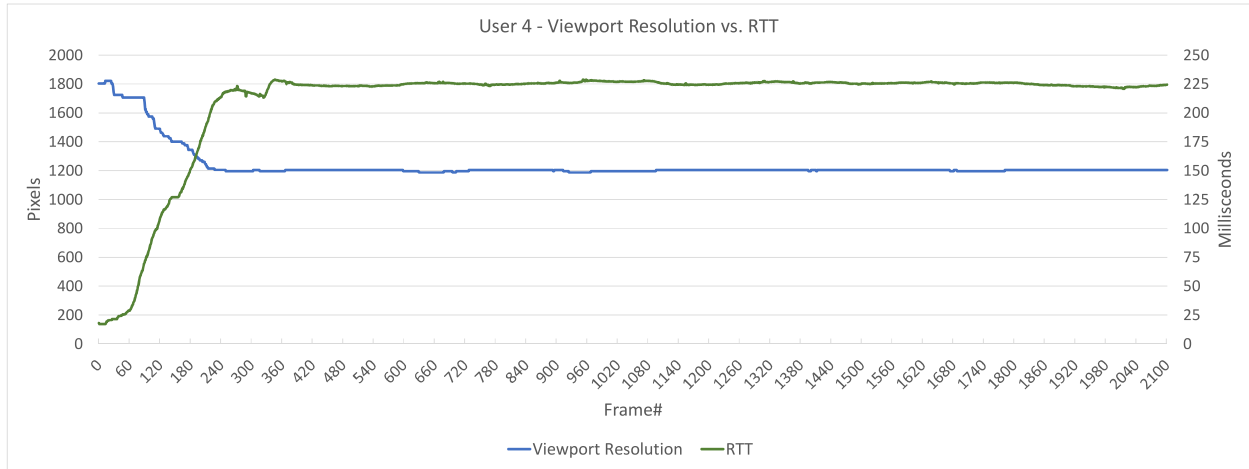


Figure 6.10: The average Round-Trip Time and viewport resolution of User 4 in the DVAR condition. The green line represents the RTT (right axis) in milliseconds, while the blue line represents the viewport resolution (left axis) in pixels.

6.2.5 User 5

Figure 6.12 shows us again a very steady RTT and viewport resolution, similar to Figures 6.4, 6.6 and 6.8. The fact that the viewport resolution remains steady at 1204 pixels throughout the majority of the session indicates that the system was relatively stable throughout the User 5's trials. Interestingly, User 5 also has the highest head velocity, strongest movements and largest amount of unrendered area.

In Figure 6.13 we see the head movement velocity and unrendered area amount plotted over the DVAR condition session. User 5 may be the most interesting case so far, given the strong and plenty head movements and the resulting amount of unrendered area. First off we can see again a peak in the first 200 frames due to the start-up effect and an early strong head movement. To remain consistent with User 3's results, we will ignore this first peak. Overall, User 5 has an average head velocity of $47.69^\circ \pm 54.82^\circ$ per second which is much higher than the previous users. User 5 also has 24 head movements (not counting the one at 60 frames). User 5's high head velocity results in 3 occasions of unrendered area being visible in the y-axis, and 1 in the x-axis. The last peak in "Blackness Y-axis" at 1570 frames stays just under the error margin of 3° . The first peak lasts 15 frames with an average blackness of $9.10^\circ \pm 2.41^\circ$ maximum 12.48° . The second and largest peak lasts 23 frames with an average blackness of $32.82^\circ \pm 16.47^\circ$ and a maximum of 55.15° . The last peak would have been barely visible at a duration of 11

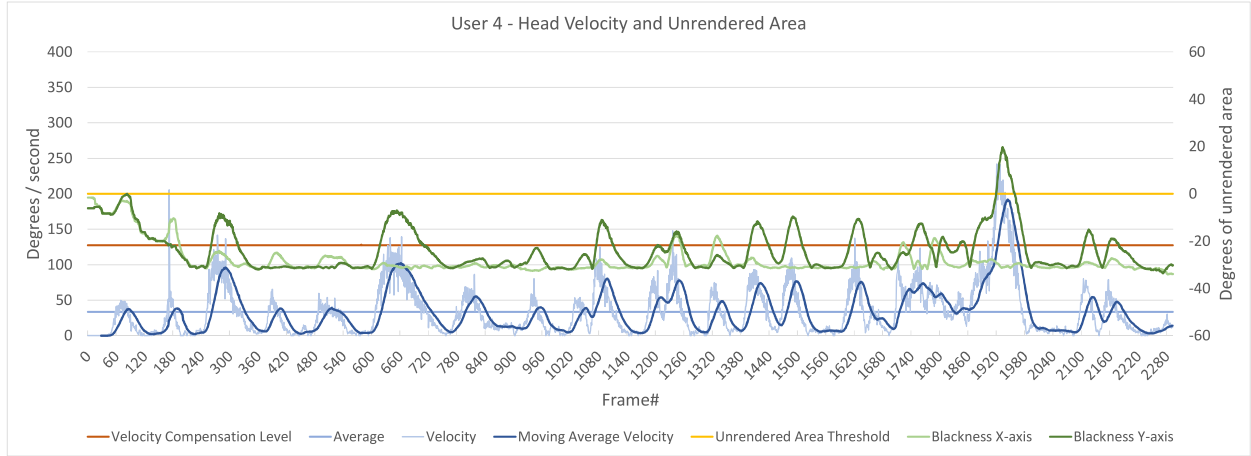


Figure 6.11: The unrendered area in both the y-axis and the x-axis, combined with the magnitude of User 4’s head velocity. The tested condition is DVAR. The blue lines represent the head velocity in degrees per second (left axis), while the green lines represent the amount of unrendered area in degrees in their respective axis. The yellow line is the threshold for when unrendered area becomes visible: blackness values above the threshold are visible, while values under the threshold represent the amount of ”visual buffer” that is still available for the given velocity compensation level and the current RTT.

frames with an average of $5.99^\circ \pm 1.77^\circ$ and maximum of 8.77° , which results in roughly 5° of blackness given the error margin. The x-axis peak (vertical direction) also barely stays under the error margin and we can thus ignore it here.

6.3 Reference Conditions

The attentive reader will have noticed that we have mainly discussed the DVAR condition so far. The reason for this is that it is by far the most interesting condition for determining whether DVAR works as it’s supposed to. In order to compare it to other implementations we care mainly about the bandwidth and resolution changes, which we have covered in an earlier section (Section 6.1). R1 and R2 both offer a full field-of-view, hence unrendered area measurements are not applicable here. Also, the viewport resolution stays constant, nullifying the need for comparison. For completeness’s sake, however, we will have a quick look at the unrendered area performance of R3 for User 5, which only renders the front face. This is visualized in Figure 6.14. The results are as expected and consistent with what we saw in Figure

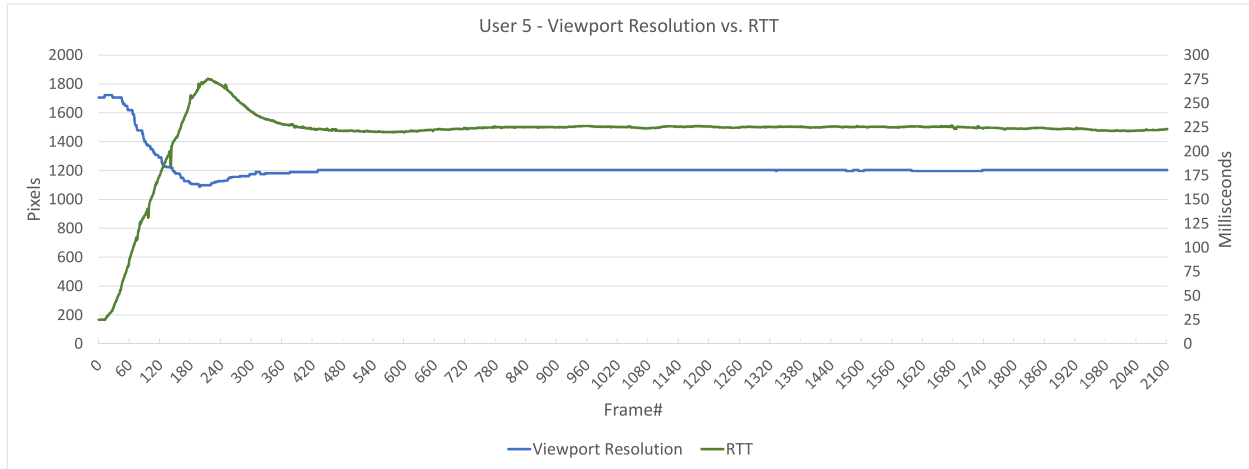


Figure 6.12: The average Round-Trip Time and viewport resolution of User 5 in the DVAR condition. The green line represents the RTT (right axis) in milliseconds, while the blue line represents the viewport resolution (left axis) in pixels.

6.3, with constantly some amount of unrendered area during the session, correlated with the head movement velocity. A change in head orientation will immediately lead to a view outside of the current viewport, while the new frame does not render immediately. Note that this is unrelated to the viewport resolution, but relies solely on the amount of FoV that is rendered unless we have a system that provides extremely low latencies, such as a tethered system.

6.4 Discussion

The results just presented provide us with a couple of key insights. First, DVAR is able to provide a much higher viewport resolution than other implementations while not using more bandwidth and not showing a significant amount of unrendered area. R3 does offer a higher viewport resolution due to the fact that we render only the HMD FoV and thus can dedicate the entire video frame to the front face. However, due to the system latency this also causes unacceptable levels of unrendered area being visible for even small movements (see Figure 6.14 for example). Furthermore, the figures showing the viewport resolution (Figure 6.1), bandwidth usage (Figure 6.2) and Unsuccessful Frames (Figure 6.3) show us that DVAR can offer a near-native viewport resolution even at high RTT values without compromising the QoE.

Given further optimizations of the current VR streaming system, or by

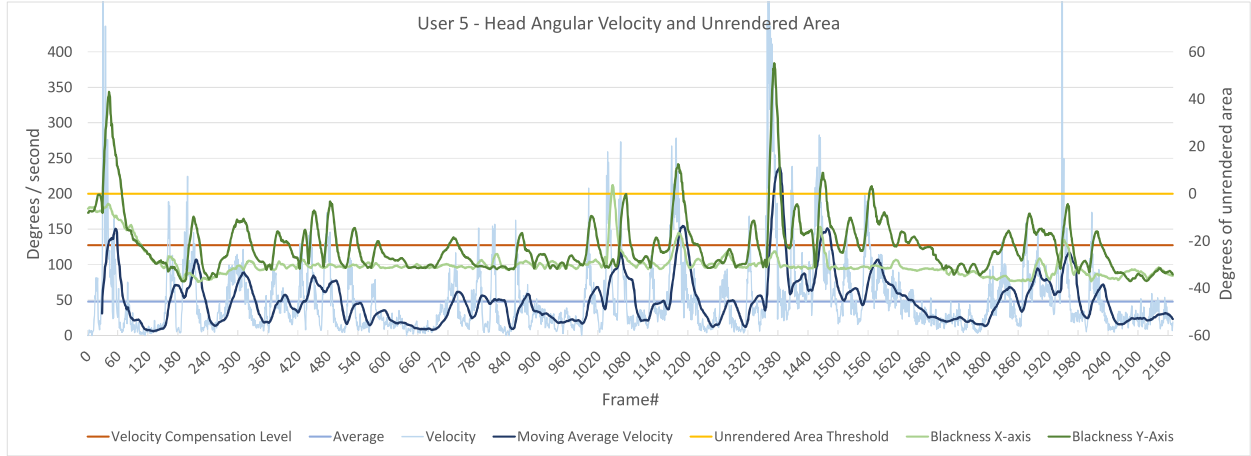


Figure 6.13: The unrendered area in both the y-axis and the x-axis, combined with the magnitude of User 5’s head velocity. The tested condition is DVAR. The blue lines represent the head velocity in degrees per second (left axis), while the green lines represent the amount of unrendered area in degrees in their respective axis. The yellow line is the threshold for when unrendered area becomes visible: blackness values above the threshold are visible, while values under the threshold represent the amount of ”visual buffer” that is still available for the given velocity compensation level and the current RTT.

applying DVAR in an existing high-performance system the RTT could be decreased significantly, leading to even higher viewport resolutions while still maintaining a good QoE by avoiding unrendered areas in the viewport. This is exemplified in Figure 6.15: if we assume that the system RTT is stable around 50ms, DVAR will provide an average viewport resolution of 1660 pixels by rendering only 12% of the side faces. Given that the other parameters remain unchanged and we have established that DVAR avoids showing unrendered area as long as the head velocity is lower than the compensation level, DVAR can provide higher-than-native viewport resolution while maintaining a good QoE. Furthermore, with lower head velocity compensation of higher video frame resolution, DVAR can provide even higher viewport resolutions. As R3 represents the best-case scenario for DVAR resolution wise (i.e. using the maximal viewport resolution supported by the video frame) and R2 represents the worst-case, we can expect DVAR’s bandwidth usage to be in between R3 and R2’s values for other viewport resolutions. Lastly, we have shown that DVAR does not increase render times significantly. In fact, with further system optimizations, DVAR is expected to have a lower performance impact than other implementations due to not having to render

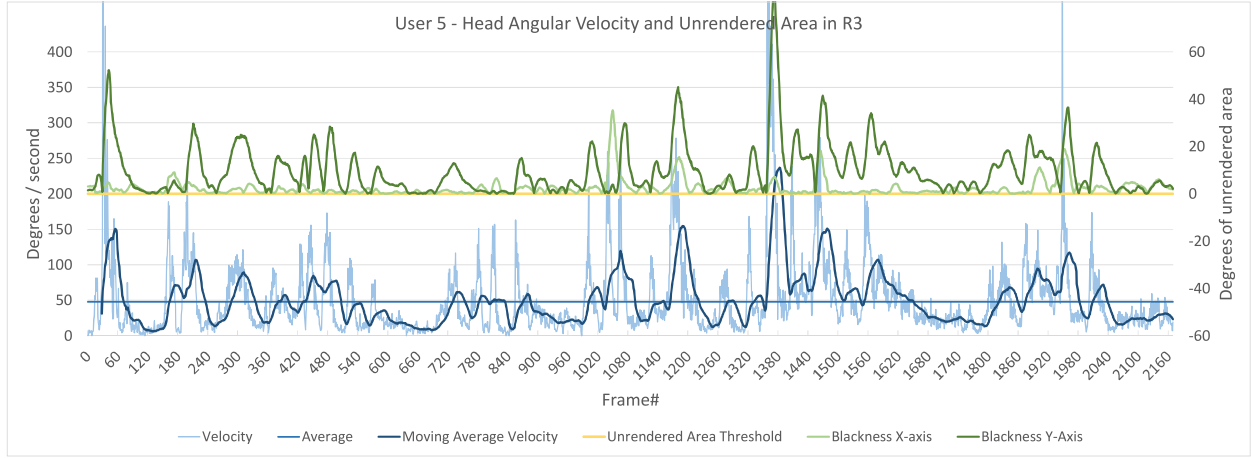


Figure 6.14: The unrendered area in both the y-axis and the x-axis, combined with User 5’s head velocity in the R3 condition. The blue lines represent the head velocity in degrees per second (left axis), while the green lines represent the amount of unrendered area in degrees in their respective axis. The yellow line is the threshold for when unrendered area becomes visible: blackness values above the threshold are visible, while values under the threshold represent the amount of “visual buffer” that is still available for the given velocity compensation level and the current RTT.

the entire VRE.

The second insight is that the amount of unrendered area critically depends on what values are used for the head movement compensation. In the results shown here we have maintained a constant head movement compensation level, which is sufficient for the task used (i.e. exploring a new VR environment): most users did not encounter any unrendered area, while User 4 and 5 did because of their fast head movements. However, in some cases, such as User 5’s fast head movements, a constant compensation level is inadequate as it cannot respond to movements that are faster than the compensation level. In other cases, such as for User 3, the compensation level may have been too high: by using a lower compensation level we could have offered a higher viewport resolution while still keeping the unrendered area to a minimum. This further emphasizes the need for a dynamic and reliable algorithm to predict head movement velocities/future head orientations in order to use DVAR to provide the optimal viewport resolution at any given time for any given scenario. One can imagine the yellow threshold line in the Unrendered Area figures being lower or higher to get an idea of the blackness in such scenarios. Ideally, the yellow line would be perfectly

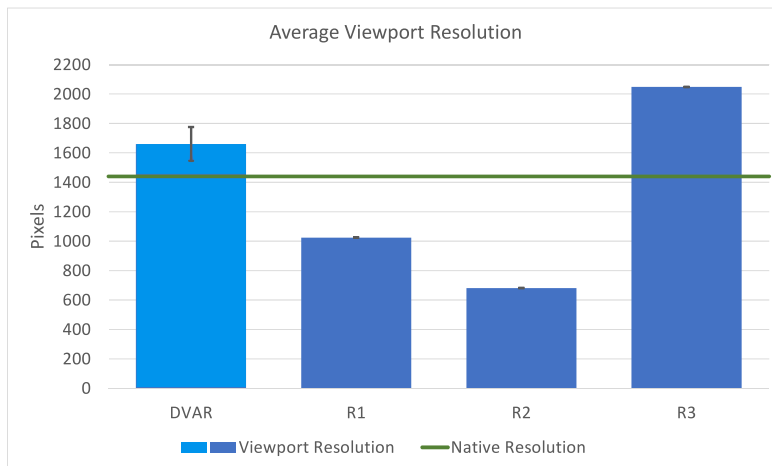


Figure 6.15: The average viewport resolution for each of the conditions, assuming the system RTT is 50ms. The DVAR conditions in that case provides an average viewport resolution of around 1660 pixels. All other parameters are as in Figure 6.1.

correlated with the head velocity (as the green and blue lines are now) to avoid any superfluous visual buffer and any unrendered area, for any head velocity.

Chapter 7

Future Work

In order to provide even higher viewport resolutions in future applications, there are three main avenues to consider: reducing the system latency, increasing the supported resolutions on the server and client side, and further optimizing the head movement compensation. Additionally, one could also further decrease the bandwidth requirements or further investigate the impact of other projection mappings on the visual quality to improve the overall QoE.

Currently the prototype system only supports 4K video frames, resulting in a maximum viewport resolution of 2K pixels per eye for stereoscopic content. By further optimizing the encoder and decoder, as well as upgrading the system's hardware higher resolutions are within reach. DVAR is system-agnostic in a way that it does not limit the maximum or minimum resolutions of the viewport resolution. Care should be taken however to ensure that higher resolutions do not further increase the system latency. In Chapter 2 we have seen a couple of interesting approaches that support higher resolutions. For example, by utilizing parallel coders or using a hybrid rendering approach between client and server. Furthermore, it is clear from DVAR's description and analysis that a lower system RTT would result in higher viewport resolutions without significantly increasing the bandwidth requirement. Similarly, we discussed several possible approaches to more precisely determine a head movement compensation level in real-time. If the head movement compensation level is lower, we again have to render less of the peripheral area and can thus provide a higher viewport resolution.

In order to provide DVAR as a complete and reliable solution for Interactive VR streaming future work should provide a way to accurately determine future head orientations to lower the compensation level. Furthermore, future work should attempt to implement DVAR in a system that enables a much lower system RTT, to profit from the increased viewport resolutions.

As we've only examined objective system performance measurements in this work, it is hard to predict what the impact of DVAR on user's QoE will be. It would be a good idea to examine the performance of DVAR, including the resolution changes, buffer area and possible resulting side effects in a subjective user study. We have also not discussed in this work how DVAR might work with other projection mappings than a cube mapping. Although a cube map is natively supported by graphics hardware and provides an especially convenient way of employing DVAR, other mappings may provide a better visual quality, more efficient bandwidth usage due to less redundant pixels, or even higher viewport quality by natively assigning more pixels to the viewport.

Lastly, an interesting possibility for future work is collaboration with existing solutions. In Chapter 2 we mentioned several VR streaming systems, some of which embrace similar goals as our work. It would be worthwhile to attempt to combine our strategies in order to provide the best possible performance in mobile Interactive VR streaming. Furthermore, many factors influence the overall performance of an Interactive VR system, such as network configuration, server and client hardware, projection mappings, encoding, and many more. All of these factors have been studied individually, to more or less extent in the context of VR streaming. However, to our knowledge there is currently no system that discusses and considers the best option for each of these factors.

Chapter 8

Conclusions

In this thesis we have presented a novel solution for rendering VR content in an adaptive way to optimize the viewport resolution given system constraints. Our solution, Dynamic Viewport-Adaptive Rendering, comprises a method that can very precisely compensate for the system latency and user head movement such that the highest possible viewport resolution is provided to the user. Simultaneously, DVAR prevents to user from observing any unrendered area resulting from the update delay by optimally rendering the peripheral area.

We have first discussed the background of virtual reality, and the particulars of streaming virtual reality. In the system architecture chapter we have elaborated on the cloud-based VR streaming system that is used in this work. We have shown how using a 3x3 cube map layout allows us to dynamically alter the proportional size of the front face of the cube map. Furthermore, by rotating the front face with the user's head orientation, we can avoid rendering all peripheral areas.

We have presented the idea for our solution, DVAR, in Chapter 4 and elaborated on it's implementation. We have especially presented the two main factors involved in optimizing such adaptive rendering: system latency and head movement velocity. We have discussed how to optimize these parameters and provided a working solution. Finally, we have presented our experiment to evaluate whether and how well DVAR actually works and presented our results. We have shown that DVAR works as expected and provides a significantly higher viewport resolution to the user, while not increasing the bandwidth requirement or decreasing QoE. Due to DVAR's design, it can be easily integrated into existing projects or enhanced with e.g. improved encoding or streaming technologies.

Bibliography

- [1] T. Mazuryk, “Virtual reality history , applications , technology and future”, 1999.
- [2] J. J. LaViola, “Double exponential smoothing: An alternative to kalman filter-based predictive tracking”, in *Proceedings of the workshop on Virtual environments 2003*, ACM, 2003, pp. 199–206.
- [3] I. Toshima, H. Uematsu, and T. Hirahara, “A steerable dummy head that tracks three-dimensional head movement: TeleHead”, *Acoustical Science and Technology*, vol. 24, no. 5, pp. 327–329, 2003, ISSN: 1346-3969. DOI: 10.1250/ast.24.327. [Online]. Available: <http://joi.jlc.jst.go.jp/JST.JSTAGE/ast/24.327?from=CrossRef>.
- [4] T. Zinner, O. Hohlfeld, O. Abboud, and T. Hossfeld, “Impact of frame rate and resolution on objective qoe metrics”, in *2010 Second International Workshop on Quality of Multimedia Experience (QoMEX)*, Jun. 2010, pp. 29–34. DOI: 10.1109/QOMEX.2010.5518277.
- [5] S. Bruck and P. A. Watters, “The factor structure of cybersickness”, *Displays*, vol. 32, no. 4, pp. 153–158, Oct. 2011, ISSN: 01419382. DOI: 10.1016/j.displa.2011.07.002. [Online]. Available: <http://dx.doi.org/10.1016/j.displa.2011.07.002%20https://linkinghub.elsevier.com/retrieve/pii/S014193821100059X>.
- [6] G. Cermak, M. Pinson, and S. Wolf, “The relationship among video quality, screen resolution, and bit rate”, *IEEE Transactions on Broadcasting*, vol. 57, no. 2, pp. 258–262, Jun. 2011, ISSN: 0018-9316. DOI: 10.1109/TBC.2011.2121650.
- [7] M. Abrash, “What VR could, should, and almost certainly will be within two years”, Valve, Tech. Rep., 2014, p. 44. [Online]. Available: <https://steamcdn-a.akamaihd.net/apps/abrashblog/Abrash%20Dev%20Days%202014.pdf>.

- [8] C.-Y. Huang, K.-T. Chen, D.-Y. Chen, H.-J. Hsu, and C.-H. Hsu, "Gaminganywhere: The first open source cloud gaming system", *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 10, no. 1s, p. 10, 2014.
- [9] T. Kamarainen, M. Siekkinen, Y. Xiao, and A. Yla-Jaaski, "Towards pervasive and mobile gaming with distributed cloud infrastructure", in *2014 13th Annual Workshop on Network and Systems Support for Games*, vol. 2015-Janua, IEEE, Dec. 2014, pp. 1–6, ISBN: 978-1-4799-6882-4. DOI: 10.1109/NetGames.2014.7008957. [Online]. Available: <http://ieeexplore.ieee.org/document/7008957/>.
- [10] S. Lavalley, A. Yershova, M. Katsev, and M. Antonov, "Head tracking for the oculus rift", English (US), *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 187–194, Sep. 2014, ISSN: 1050-4729. DOI: 10.1109/ICRA.2014.6906608.
- [11] S. Möller and A. Raake, *Quality of experience: advanced concepts, applications and methods*. Springer, 2014.
- [12] J. Jerald, *The VR book: Human-centered design for virtual reality*. Morgan & Claypool, 2015.
- [13] U. Pal and H. King, "Effect of uhd high frame rates (hfr) on dvb-s2 bit error rate (ber)", in *SMPTE15: Persistence of Vision-Defining the Future*, SMPTE, 2015, pp. 1–11.
- [14] Y. Bao, H. Wu, T. Zhang, A. A. Ramli, and X. Liu, "Shooting a moving target: Motion-prediction-based transmission for 360-degree videos", in *2016 IEEE International Conference on Big Data (Big Data)*, IEEE, Dec. 2016, pp. 1161–1170, ISBN: 978-1-4673-9005-7. DOI: 10.1109/BigData.2016.7840720. [Online]. Available: <http://ieeexplore.ieee.org/document/7840720/>.
- [15] K. Boos, D. Chu, and E. Cuervo, "Flashback: Immersive virtual reality on mobile devices via rendering memoization", in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '16, Singapore, Singapore: ACM, 2016, pp. 291–304, ISBN: 978-1-4503-4269-8. DOI: 10.1145/2906388.2906418. [Online]. Available: <http://doi.acm.org/10.1145/2906388.2906418>.
- [16] X. Corbillon, G. Simon, A. Devlic, and J. Chakareski, "Viewport-Adaptive Navigable 360-Degree Video Delivery", in *Proceedings of the 5th ACM Multimedia Systems Conference on - MMSys '14*, ACM Press, Sep. 2016, pp. 271–282, ISBN: 9781450327053. DOI: 10.1109/ICC.2017.

7996611. arXiv: 1609.08042. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=2557642.2557652%20http://arxiv.org/abs/1609.08042%20http://dx.doi.org/10.1109/ICC.2017.7996611>.
- [17] T. El-Ganainy and M. Hefeeda, “Streaming Virtual Reality Content”, pp. 1–8, Dec. 2016. arXiv: 1612.08350. [Online]. Available: <http://arxiv.org/abs/1612.08350>.
- [18] O. Abari, D. Bharadia, A. Duffield, and D. Katabi, “Enabling high-quality untethered virtual reality”, in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’17, Boston, MA, USA: USENIX Association, 2017, pp. 531–544, ISBN: 978-1-931971-37-9. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3154630.3154674>.
- [19] H. Ahmadi, O. Eltobgy, and M. Hefeeda, “Adaptive Multicast Streaming of Virtual Reality Content to Mobile Users”, in *Proceedings of the on Thematic Workshops of ACM Multimedia 2017 - Thematic Workshops '17*, ACM Press, 2017, pp. 170–178, ISBN: 9781450354165. DOI: 10.1145/3126686.3126743. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3126686.3126743>.
- [20] X. Corbillon, F. De Simone, and G. Simon, “360-Degree Video Head Movement Dataset”, in *Proceedings of the 8th ACM on Multimedia Systems Conference - MMSys’17*, ACM Press, 2017, pp. 199–204, ISBN: 9781450350020. DOI: 10.1145/3083187.3083215. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3083187.3083215>.
- [21] X. Corbillon, A. Devlic, G. Simon, and J. Chakareski, “Optimal Set of 360-Degree Videos for Viewport-Adaptive Streaming”, in *Proceedings of the 2017 ACM on Multimedia Conference - MM '17*, ACM Press, 2017, pp. 943–951, ISBN: 9781450349062. DOI: 10.1145/3123266.3123372. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3123266.3123372>.
- [22] A. M. Gavvani, K. V. Nesbitt, K. L. Blackmore, and E. Nalivaiko, “Profiling subjective symptoms and autonomic changes associated with cybersickness”, *Autonomic Neuroscience: Basic and Clinical*, vol. 203, pp. 41–50, 2017, ISSN: 18727484. DOI: 10.1016/j.autneu.2016.12.004. [Online]. Available: <http://dx.doi.org/10.1016/j.autneu.2016.12.004>.
- [23] M. Graf, C. Timmerer, and C. Mueller, “Towards Bandwidth Efficient Adaptive Streaming of Omnidirectional Video over HTTP”, in *Proceedings of the 8th ACM on Multimedia Systems Conference - MMSys’17*, New York, New York, USA: ACM Press, 2017, pp. 261–271,

- ISBN: 9781450350020. DOI: 10.1145/3083187.3084016. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3083187.3084016>.
- [24] T. Kämäräinen, M. Siekkinen, A. Ylä-Jääski, W. Zhang, and P. Hui, “A Measurement Study on Achieving Imperceptible Latency in Mobile Cloud Gaming”, pp. 88–99, 2017. DOI: 10.1145/3083187.3083191.
- [25] Y. Rai, J. Gutiérrez, and P. L. Callet, “A dataset of head and eye movements for 360 degree images”, in *Proceedings of the 8th ACM on Multimedia Systems Conference*, 2017, pp. 205–210.
- [26] Z. Chen, Y. Li, and Y. Zhang, “Recent advances in omnidirectional video coding for virtual reality: Projection and evaluation”, *Signal Processing*, vol. 146, pp. 66–78, 2018, ISSN: 01651684. DOI: 10.1016/j.sigpro.2018.01.004. [Online]. Available: <https://doi.org/10.1016/j.sigpro.2018.01.004>.
- [27] A. Doumanoglou, D. Griffin, J. Serrano, N. Zioulis, T. K. Phan, D. Jimenez, D. Zarpalas, F. Alvarez, M. Rio, and P. Daras, “Quality of Experience for 3-D Immersive Media Streaming”, *IEEE Transactions on Broadcasting*, vol. 64, no. 2, pp. 379–391, Jun. 2018, ISSN: 0018-9316. DOI: 10.1109/TBC.2018.2823909. [Online]. Available: <https://ieeexplore.ieee.org/document/8373005/>.
- [28] J. Heyse, M. Torres Vega, T. Wauters, F. De Backere, and F. De Turck, “Effects of adaptive streaming optimizations on the perception of 360 virtual reality video”, *Proceedings of the 30th International Teletraffic Congress, ITC 2018*, pp. 89–92, 2018. DOI: 10.1109/ITC30.2018.00021.
- [29] G. Illahi, T. Van Gemert, M. Siekkinen, E. Masala, A. Oulasvirta, and A. Ylä-Jääski, “Cloud Gaming With Foveated Graphics”, Sep. 2018. arXiv: 1809.05823. [Online]. Available: <http://arxiv.org/abs/1809.05823>.
- [30] T. Kämäräinen, M. Siekkinen, J. Eerikäinen, and A. Ylä-Jääski, “Cloudvr: Cloud accelerated interactive mobile virtual reality”, in *Proceedings of the 26th ACM International Conference on Multimedia*, ser. MM ’18, Seoul, Republic of Korea: ACM, 2018, pp. 1181–1189, ISBN: 978-1-4503-5665-7. DOI: 10.1145/3240508.3240620. [Online]. Available: <http://doi.acm.org/10.1145/3240508.3240620>.
- [31] S. Knorr, C. Ozcinar, C. O. Fearghail, and A. Smolic, “Director’s cut: A combined dataset for visual attention analysis in cinematic vr content”, in *Proceedings of the 15th ACM SIGGRAPH European Conference on Visual Media Production*, 2018.

- [32] L. Liu, R. Zhong, W. Zhang, Y. Liu, J. Zhang, L. Zhang, and M. Gruteser, “Cutting the Cord: Designing a High-quality Untethered VR System with Low Latency Remote Rendering”, *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys '18*, pp. 68–80, 2018. DOI: 10.1145/3210240.3210313. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3210240.3210313>.
- [33] M. Startsev and M. Dorr, “360-aware saliency estimation with conventional image saliency predictors”, *Signal Processing-image Communication*, vol. 69, pp. 43–52, 2018.
- [34] Y. Xu, Y. Dong, J. Wu, Z. Sun, Z. Shi, J. Yu, and S. Gao, “Gaze prediction in dynamic 360 immersive videos”, in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, pp. 5333–5342.
- [35] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, N. Dai, and H.-S. Lee, “Furion: Engineering High-Quality Immersive Virtual Reality on Today’s Mobile Devices”, *IEEE Transactions on Mobile Computing*, pp. 1–1, 2019, ISSN: 1536-1233. DOI: 10.1109/tmc.2019.2913364.
- [36] S. Shi, V. Gupta, M. Hwang, and R. Jana, “Mobile vr on edge cloud: A latency-driven design”, in *Proceedings of the 10th ACM Multimedia Systems Conference*, ser. MMSys '19, Amherst, Massachusetts: ACM, 2019, pp. 222–231, ISBN: 978-1-4503-6297-9. DOI: 10.1145/3304109.3306217. [Online]. Available: <http://doi.acm.org/10.1145/3304109.3306217>.
- [37] Dan Simpson, *The Ultimate Extended Reality (XR) Glossary*. [Online]. Available: <https://www.tvr1p.com/ultimate-extended-reality-glossary/> (visited on 09/12/2019).
- [38] Facebook Technologies, LLC, *Oculus Go Product Page*. [Online]. Available: <https://www.oculus.com/go/> (visited on 08/31/2019).
- [39] —, *Oculus Quest Product Page*. [Online]. Available: <https://www.oculus.com/quest/> (visited on 08/31/2019).
- [40] *FB Capture SDK on GitHub.com*. [Online]. Available: <https://github.com/facebook/360-Capture-SDK> (visited on 09/02/2019).
- [41] *Google Daydream Wikipedia Entry*. [Online]. Available: https://en.wikipedia.org/wiki/Google%7B%5C_%7DDaydream (visited on 08/31/2019).
- [42] *History of VR (VRS)*. [Online]. Available: <https://www.vrs.org.uk/virtual-reality/history.html> (visited on 08/31/2019).

- [43] Lenovo, *Lenovo Mirage Solo Product Page*. [Online]. Available: <https://www.lenovo.com/us/en/virtual-reality-and-smart-devices/virtual-and-augmented-reality/lenovo-mirage-solo/Mirage-Solo/p/ZZIRZRHVR01> (visited on 08/31/2019).
- [44] NVidia Video Codec SDK documentation. [Online]. Available: <https://developer.nvidia.com/nvidia-video-codec-sdk> (visited on 09/05/2019).
- [45] OneirosVR, *ArchVizPRO Interior Vol.6*. [Online]. Available: <https://assetstore.unity.com/packages/3d/environments/urban/archvizpro-interior-vol-6-120489%20https://oneirosvr.com/portfolio/archvizpro/> (visited on 04/01/2019).
- [46] Samsung Gear VR Product Page. [Online]. Available: <https://www.samsung.com/global/galaxy/gear-vr/> (visited on 08/31/2019).
- [47] Samsung Gear VR Wikipedia Entry. [Online]. Available: https://en.wikipedia.org/wiki/Samsung%7B%5C_%7DGear%7B%5C_%7DVR (visited on 08/31/2019).
- [48] Virtual Reality Definition. [Online]. Available: https://www.lexico.com/en/definition/virtual%7B%5C_%7Dreality (visited on 08/31/2019).
- [49] Virtual Reality Wikipedia Entry. [Online]. Available: https://en.wikipedia.org/wiki/Virtual%7B%5C_%7Dreality (visited on 08/31/2019).
- [50] What Is VR? An overview of VR and VR headsets. [Online]. Available: <https://www.marxentlabs.com/what-is-virtual-reality/> (visited on 08/31/2019).
- [51] Youtube's Ready to Blow Your Mind With 360-Degree Videos. [Online]. Available: <https://gizmodo.com/youtubes-ready-to-blow-your-mind-with-360-degree-videos-1690989402> (visited on 08/31/2019).
- [52] Zion Market Research, *VR Market report*. [Online]. Available: <https://www.globenewswire.com/news-release/2019/07/12/1882002/0/en/Report-Virtual-Reality-VR-Market-Size-Revenue-To-Surge-To-US-26-89-Billion-by-2022.html> (visited on 08/31/2019).