ASSESSING IMAGE QUALITY IMPACT OF VIEW BYPASS IN CLOUD RENDERING

by

Alex Stamm

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science

Department of Computer Graphics Technology

West Lafayette, Indiana

May 2019

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

I dedicate this thesis to Bo, the cat, whom always made sure I was awake on time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

SSIM	Structural Similarity Index Method
FPS	Frames per Second
3D	Three Dimensional
ND	Network Delay
PD	Processing Delay
OD	Output Delay
3DOF	Three Degrees of Freedom
6DOF	Six Degrees of Freedom

ABSTRACT

Author: Stamm, Alex. MS
Institution: Purdue University
Degree Received: May 2019
Title: Assessing Image Quality Impact of View Bypass in Cloud Rendering
Committee Chair: Tim McGraw

The accessibility and flexibility of mobile devices make them an advantageous platform for gaming, but there are hardware limitations that impede the rendering of high-quality graphics. Rendering complex graphics on a mobile device typically results in a delayed image, also known as latency, and is a great discomfort for users of any real-time rendering experience. This study tests the image stream optimization View Bypass within a cloud gaming architecture, surpassing this imposing limitation by processing the high-quality game render on a remote computational server. A two sample for means test is performed to determine significance between two treatments: the control group without the View Bypass algorithm and the experimental group rendering with the View Bypass algorithm. A SSIM index score is calculated comparing the disparity between the remote server image output and the final mobile device image output after optimizations have been performed. This score indicates the overall image structural integrity difference between the two treatments and determines the quality and effectiveness of the tested algorithm.

CHAPTER 1. INTRODUCTION

1.1 Statement of the Problem

Graphics hardware is converging into two categories: powered desktops and lightweight mobile devices. Each are designed for their intended purpose, yet each have a drawback. Desktops render console-quality images at speedy frame rates, but do not provide the convenience given by the mobile device. Those who wish to combine the two and play desktop-ready games on their mobile device might find their experience lacking due to the lowered frame rate and image quality. In an immersive virtual environment, latency is a noticeable and undesirable feature that can cause great discomfort and motion sickness for users (Regan & Pose, 1994). Many people currently experience this delay when watching a video buffer online, but this effect in virtual environments may cause greater uneasiness for a user due to need for an even faster display response time. Reducing this delay between images in the display to minimize these feelings of motion sickness has been a challenge for scholars. Applying image stream optimizations such as View Bypass may help in reducing the measurable latency in virtual environments (Carmack, 2014).

1.2 Research Question

To what extent does the View Bypass image stream optimization affect image quality for immersive virtual environments within a mobile cloud gaming framework?

1.3 Significance

The results of this study may benefit society by contributing to the advancement of high-fidelity graphics for future immersive simulations and virtual reality rendering. Latency is a major factor in providing a high-quality experience (Carmack, 2014). Enhancing real-time rendering methods may lead to advances in the field of computer graphics and virtual reality. Studies have shown that skillsets can be improved as a result of training inside highly immersive virtual reality simulations (Knerr, 2007), and may become a leading tool for enhancing training and education. Increasing overall immersion graphically may lead to increased feelings of presence inside a virtual environment, providing a more engaging experience for users interacting with virtual content. As mobile devices become more common all around the world, finding ways to enable high quality and low latency graphics on these devices may create more accessibility, enabling more potential players for the gaming market, and increasing the overall gaming experience for everyone.

1.4 Scope

This study tests the View Bypass algorithm, seeking to mitigate latency in the context of a mobile cloud gaming environment. The experiment compares two treatments: mobile frames streamed by the server with and without the View Bypass algorithm. Each treatment is composed of 100 trials, and each trial lasts approximately 30 seconds. The study is conducted on the Android operating system for mobile and Windows operating system for the powered server. The server uses the Unity 3D game engine (Unity Technologies, 2019) and C# for scripting. The virtual scene is a simple environment with instanced foliage and trees utilizing depth textures. The complexity of the environment is balanced: simple so that the desktop may quickly render and encode the video frames, yet complex enough that it cannot be rendered by the mobile device alone. Vertex count is consistent, and textures have a resolution of 1024x1024. One GLSL shader handles writing textures for the server camera and the other GLSL shader handles the mobile display playback. SSIM values are gathered in FFmpeg by comparing the experimental frames at fixed frame intervals against a control video frame rendered and stored on the server. Render frame processing speed is calculated within the Unity game engine on the server side, and display processing delay is determined by the mobile frame rate. The WebRTC protocol is utilized to stream the video data over the network.

1.5 Definitions

Immersive Virtual Environments – lets the user experience a computer-generated world as if it were real producing a sense of presence, or “being there” in the user’s mind. (Bowman, 2007)

Latency – the time delay between the display image and the user’s movement (Regan & Pose, 1994)

Address Recalculation Pipeline – “A graphics display architecture which will detach user head orientation from the rendering process” (Regan & Pose, 1994, p. 1)

View Bypass – An image process that uses the delta in user input to calculate a new display frame by updating the view matrix in the graphics rendering pipeline (Carmack, 2014)

RTP Packet – a Real-time Transform Protocol (RTP) packet containing media data (Holmer & Alvestrand, 2011)

Frame - a set of RTP packets transmitted from the sender at the same time instant, a video texture at a single moment in time (Holmer & Alvestrand, 2011)

1.6 Assumptions

The assumptions of this study include assuming a reliable connection to the server throughout the duration of the trial. The network speed is assumed to be consistent for the duration of each individual trial. The view position at each location in time is assumed to be consistent across trials as they derive from the same seed animation. The rendered immersive virtual environment is assumed to be graphically demanding enough to be virtually unplayable on a mobile-only solution. Finally, the test mobile device has enough reserve graphics processing power to decode a standard WebRTC video stream as well as consistently send transform update data to the server.

1.7 Limitations

Limitations to this study include the network speed available for this testing, averaging around 10Mbps. Processing latency is another factor primarily dependent on the game engine of choice as the rendering abilities differ between commonly used game engines. Processing latency may only be minimized in this context by reducing the complexity of the virtual environment. Other limitations include the final frame output is affected by the WebRTC video streaming algorithm, which may generate unwanted noise or compression artifacts.

1.8 Delimitations

The application developed for this study is not built to run on multiple mobile operating systems. Choosing Android over iOS seems to be the proper choice as it is more cost-effective, the operating system is open source, and can more easily be edited.

Another delimitation in this study includes a singular immersive virtual environment for consistency. Utilizing a standardized game scene from the Unity Asset Store such as the Nature

Starter Pack 2 (Shapes, 2018) is simple for a desktop to process, yet complex enough for performance issues to occur on a mobile only setup.

The target resolution is set to be 1024x1024 as the frames are recorded on square render textures with size 2^n for ideal memory usage. This resolution was chosen as the minimum width and height that does not surpass the maximum texture size for the mobile renderer. The target frame rate for the cloud gaming framework is set to 30fps.

1.9 Summary

Reducing latency and improving image display has constantly been a challenge for graphics scientists and engineers who are often limited by the available hardware resources to build a given system. Having a software solution to optimize this type of mobile cloud gaming architecture could lead to more immersive and accessible gaming methods. This solution utilizes the Unity3D game engine to maintain scope; however, the underlying algorithm may be applied to any game engine system using a game loop and render camera. This rendering method, by utilizing WebRTC, prioritizes packet speed over loss, making it a potential future solution for view-agnostic rendering for graphically demanding immersive virtual environments.

CHAPTER 2. LITERATURE REVIEW

The following chapter discusses the background literature used to illustrate this study. This review discusses immersion and how it can be quantified, how latency is defined and mitigated in previous studies, the various common video streaming methods, the cloud gaming method, and measurements used to break down total system latency. Finally, SSIM, the metric for image quality, is described as the method for determining frame quality output.

2.1 Immersion

As computer graphics evolves the gaming experience by enhancing image quality and frame rates, the level of immersion required by the user grows, along with expectations. Game developers are constantly innovating to meet these expectations, leading to advances in real-time rendering techniques in order to maintain immersion and give the player a sense of agency inside the virtual world. Steuer (1992) described this experience as ‘telepresence,’ or the sense of locality between parties in virtual communication. He described a situation where two individuals talk to each other on the phone, both aware the other is not actually in the same room. The feeling that the person on the other end is locally present from this communication is telepresence. Steuer (1992) described two dimensions that enhance telepresence: vividness and interactivity. Vividness is described as the ability for the technology to produce a sensory rich environment. It is enhanced through redundancy, or reinforcing, reality through multiple different senses. Other metrics to describe vividness include speed of the display, quality of the resolution, and breadth of available sensory outputs. Interactivity is defined as the degree to which users can influence the virtual environment. This includes the ability to pick up and manipulate objects and to interact with virtual or real players in the simulation. Interactivity also includes the range, or number of given possibilities for action, and the mapping consistency of the motion control inputs. Enhancing the speed at which the display is received by the user directly enhances the vividness experienced by the user, thus enhancing the total telepresence for the user in a given system. Increasing image display rate leads to more total vividness in a system, which further enhances immersion in the simulation. Maintaining interactivity in this same system allows for optimal telepresence in the context of cloud gaming.

The words immersion and presence are often used interchangeably when describing virtual simulations, so it is important understand the nuanced differences in the definitions of these two words. Bowman, Mcmahan, and Tech (2007) described immersion as the quantitative element and presence as the qualitative. These authors went into further detail to quantify immersion by describing it as the systems and devices that add sensory input. Immersion is objective in nature and easily measurable. Enhancing display response times can quantitatively contribute to enhanced immersion in the simulation. Presence can be described as subjective in nature. According to Heeter (1992) presence refers to the perception a user experiences by their interaction with the virtual simulation. Presence may not be measured as easily, but enhancing immersion has been shown to also contribute to enhanced presence in a virtual system. Steuer (1992) suggested that better ‘narratization’ contributes to better perceived experiences. He even suggested that multiple users experiencing the same virtual environment leads to enhanced social presence, the idea that the existence of other people in your virtual world gives it more validity.

As immersive virtual simulations become more common in society and begin to develop into a practical medium for self-improvement and entertainment alike, users will require more realistic, lower-latency, and higher resolutions to increase their level of immersion. Studies have shown cognitive skills may be developed as a result of training inside these highly immersive virtual reality simulations (Knerr, 2007). Users in these tests self-reported improvement in coordination, communication, and planning. The willingness to participate with the graphical simulation led to increased preoccupation inside the virtual reality environment, providing a channel for users to more easily suspend their disbelief to engage more with the content. Recognizing the value that a highly immersive environment may have on a user's subjective experience gives purpose to finding methods and developing latent strategies to strive towards achieving fully immersive simulations.

2.2 Latency

In immersive virtual environments, delays between the user's head rotation and the display image can cause great discomfort in users. According to Regan and Pose (1994), this delay, known as latency, is a noticeable and undesirable feature in many systems. Minimizing the latency perceived by a user during head rotation increases the realism of the virtual world. While the sourced images are supplied to the display at a rate faster than what the user can visibly perceive, the irritation from latency is greatly diminished. A reduction to this type of latency is one of the greatest common factors in improving immersion in virtual reality.

One of the earliest latency mitigation strategies was developed by Regan and Pose with their Address Calculation Pipeline method. This method calculates the image at the last possible moment before the image reaches the display. First, the renderer generates an image based on the user's view position. Then, the processor temporarily stores the given image to be presented to the user. Before display, the frame is recalculated using the motion vector generated by the user movement. Because the pixel differences are across small time increments, small pixel adjustments are made using the updated player view matrix.

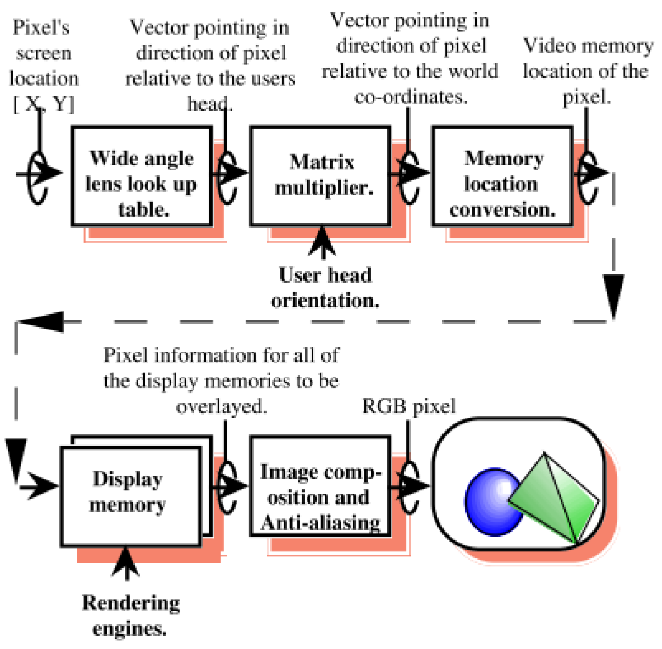
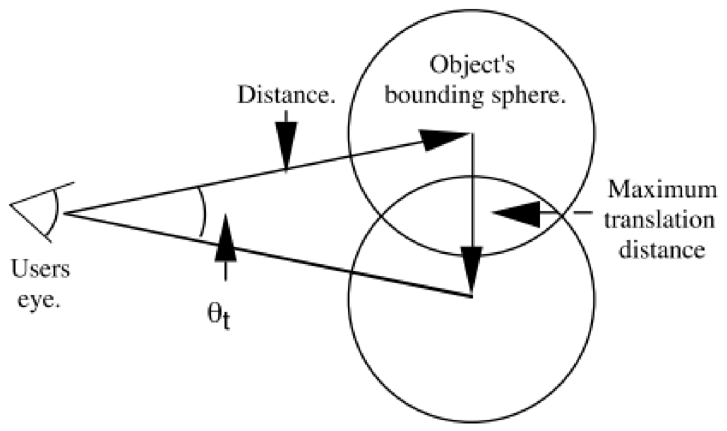

Pixel's screen location [X, Y]
Vector pointing in direction of pixel relative to the users head.
Vector pointing in direction of pixel relative to the world co-ordinates.
Video memory location of the pixel.
Wide angle lens look up table.
Matrix multiplier.
Memory location conversion.
User head orientation.
Pixel information for all of the display memories to be overlayed.
RGB pixel
Display memory
Image comp-osition and Anti-aliasing
Rendering engines.

Figure 2.1. The address recalculation pipeline. (Regan & Pose, 1994, p.2)

Additional mitigation strategies to reducing this delay have been presented by Carmack (2014). He suggested methods such as Alternate Frame Rendering which uses two separate graphics processors, each dedicated to a singular eye view of a virtual display. Considering the fact that two eye displays are required to render stereoscopic virtual reality environments, streamlining the data into two separate jobs reduces the total load. The parallel nature of the graphics rendering allows for more scalable models as well.



$$\tau_{translation}(i) = \frac{\text{distance}(i) * \text{sqrt}(2*(1-\cos(\theta_t)))}{\text{relative\_speed}(i)}$$

Where i = object number.

Figure 2.2. An object's translational validity period (Regan & Pose, 1994, p.5)

Further optimizations suggested by Carmack (2014) include using View Bypass, a method where the rendered image displayed on the screen is not reliant on user input, so the time it takes for input to travel to the processor is not included in final latency times. The method described by Carmack is a similar approach to the Address Recalculation Pipeline by Regan and Pose (1992). Both solutions approached the problem by rendering the final image by transforming the view matrix by a delta calculated between the current and previous transform. Carmack approached the problem in the context of virtual reality rendering. He noted the sensitivity of the human eye makes small delays in image even more noticeable for the user, making latency even more of a challenge.

Carmack (2014) proposed a game processing model that should be analyzed as a potential graphics pipeline for low-latency displays. His model can be described as a sequential flow of control states. First the user input is read, the simulation is run, then rendering commands are issued, followed by graphical draw calls, waiting for v-sync, and displaying the final output. This method uses linear programming practices and better optimizes the flow state of the rendering program (Steinbach, Farber, & Girod, 2001).

According to Carmack (2014), achieving latency reduction to graphical applications with the described rendering methods provided improved computational performance of the system. These algorithms may increase the speed at which the GPU can display an image to the screen in order to decrease total latency; however, these techniques are speeding up a process that already outputs in under 100 milliseconds (ms). Lowering this number becomes more difficult as it approaches zero. Carmack suggested using these latency reduction strategies in the context of cloud gaming as a way of filling a current gap in research. Framing the topic from this perspective has potential for greater impact due to the tendency for networked applications to exhibit high amounts of latency. Signal speed becomes a bottleneck when trying to reach dynamic display speeds of less than 100 ms using a transmission that must travel roundtrip to a remote server for processing. Modern applications that require the Internet may also migrate between servers, further contributing to the total signal travel time. These realistic limitations must be considered when trying to develop solutions for reducing latency in applications with large data flows.

2.3 Video Streaming

Video streaming requires large amounts of data to be transferred quickly from a server to a receiving client. Many video streaming protocols have been developed that compress video frames into packets that get sent over the network.

One commonly known type of video streaming is the HTTP Live Streaming (HLS) protocol. This method, used by most media and content providers today, sends video content as small segments that are processed in sequential order by the graphics processor. This method prioritizes reliable packets but can result in the video requiring buffer time with a low network connection speed. A low-latency HLS solution was developed by Kalman, Davies, Hill, and Pracht (2017) that loads these small video segments as an advertised playlist consisting of many segments at once. These ‘chunks’ allow for a playlist of videos segments for the client to have buffered and ready to display if there happens to be a request error or a temporary drop in connection. This method has resulted in much less noticeable buffering time. Having adaptive control of data packet size by calculating an error retry rate for the data request can also help mitigate some apparent latency (Steinbach et al., 2001).

Another evolving video streaming protocol is DASH, or Dynamic Adaptive Streaming over HTTP. DASH is a MPEG standard, with a protocol that dynamically adapts to the incoming bitrate of the stream, and alters the decoding algorithm based on the network bottleneck. DASH is described as an advancement to HLS streaming, which also utilizes HTTP, but has a focus on resulting in a pliant output based on the available network bandwidth (Fund, Wang, Liu, Korakis, Zink, & Panwar, 2013).

WebRTC, or HTML5 Web Real Time Communication, is another common video streaming protocol that plays a role in browser-to-browser communication. Developed at Google by Holmer and Alvestrand in 2011, this protocol has a unique congestion algorithm that adjusts image quality based on input bandwidth. Adopted as a World Wide Web Consortium (W3C) standard, this method dynamically adapts by estimating the available bandwidth and using this calculation to estimate the bitrate of the target encoder (Fund et al., 2013). The design of WebRTC focuses on real-time communication of large data packets utilizing User Datagram Protocol (UDP) based network signaling instead of the normally used Transmission Control

Protocol (TCP) with HTTP. With the emphasis on real-time communication, the drawback of this method is reducing latency as opposed to preventing packet loss. The result is faster frame rates and display times but a potential loss of video quality. According to Nurminen, Meyn, Jalonen, Raivio, and Garcıa Marrero (2013), an assumption we can make with the WebRTC protocol, in regard to mobile performance, is a single incoming video stream should not be an intensive process for a mobile device using WebRTC for decoding. Nurminen et al. (2013) went on to say that handling two streams, an upstream and downstream is also quite manageable for current desktops and newer mobile devices.

Multicasting is the method of streaming audio and video content to a group of users. Liu, Guo, and Liang (2008) have surveyed multiple forms of multicasting and image streaming that focus on reducing in latency. Most content delivery networks deal with a server to client communication downstream that can be classified as a live streamed video or video on-demand (VoD). Live streamed video handles content within a video playback ‘window of time’ when outputting data. VoD is accessing a video with asynchronous runtimes across the board for users and are known to have more complications with data storage. Peer-to-peer networks have also been proposed to deliver video content on a more distributed platform. Instead of having the server send the entire segment of data by itself, a peer-to-peer network will partition the data among the active users so that the load is greatly reduced, resulting in ultimately lower bandwidths and latencies. Liu et al. (2008) described the most notable early peer-to-peer network types being tree based. These methods achieved low latencies, but suffered from a problem known as peer churn, where a user supplying a section of the segmented data leaves the session, resulting in a data gap. A mesh-based type of peer-to-peer network has been proposed to reduce this type of error. Rather than relying on a singular user instance for data transfer, multiple users have copies of the same data segment, and the requesting user has many possible caches to connect to in the case of an ungraceful closure from the data pipeline. These programming architectures have yet to be mass adopted due to current technological limitations such as modern internet speeds and data storage. Building these types of networks requires a deviation from typical data storage and transfer methods, which generally follow a linear approach to its intrinsic architecture.

These video streaming protocols could be applied to cloud gaming and satisfy the image streaming requirement for the framework. Many groups have been working towards optimizing this practice for the sake of enhancing the virtual reality experience through low-latency. Zhihan, Yin, Han, Y. Chen, and G. Chen. (2011) discussed a peer-to-peer model for rendering networked virtual environments, but the research leaves a gap in developing an application with this network type. Their study discusses WebVR, an open API that utilizes web browsers to compute high-end virtual reality simulations inside the browser. Due to its peer-to-peer nature, the more people connected to a website, the more interchange of data that can be shared for optimizing the application. Other groups such as Friston, Steed, Tilbury, and Gaydadjiev (2016) have developed a custom frameless renderer to minimize work in rendering a virtual environment. This approach renders pixels in an “arbitrary, application-defined order”, that segments the frame into varying resolutions based on factors such as field of view. This method contrasts the traditional “painter’s algorithm” approach in graphics where an entire frame is filled with pixels of the same resolution before being shown to the display in a sequence. The potential use of the frameless renderer method is optimizing based on where a user might be looking, sparing processing from rendering more peripheral pixels. The downside to a frameless renderer is the need for a raycasting approach instead of a rasterized full-frame approach typically seen in most virtual reality devices, games, and applications today.

2.4 Cloud Gaming

Cloud gaming is the conventional application of these image streaming models in the context of rendering immersive virtual environments. Shea, Liu, Ngai, and Cui (2013) provided an explanation for why technology is expected to advance in this direction. The authors described leveraging a remote graphics rendering server to perform a procedure known as computational offloading. This method lessens the processing burden of a device such as a mobile phone by redistributing the rendering load to an offsite remote server. The application may then access the image data through a local terminal, only needing to process the received video stream.

Cloud gaming as a process has methods that are similar to live media streaming: both must compress and encode video in addition to handling distribution of the content. Shea et al. (2013) concisely explained the approach taken by a cloud gaming system:

> *Cloud gaming, in its simplest form, renders an interactive gaming application remotely in the cloud and streams the scenes as a video sequence back to the player over the Internet. A cloud gaming player interacts with the application through a thin client, which is responsible for displaying the video from the cloud rendering server as well as collecting the player's commands and sending the interactions back to the cloud.* (p. 1)

Cloud gaming is an effective system holistically, but it consists of a complex architecture in terms of design. Dissecting the definition given by Shea et al. (2013), this architecture is a remote, Internet based system that requires two components: the server and the client, or in this case, a thin client. The role of the thin client is to be as limited as possible in terms of processing and should be perceived as a local terminal. The duties of the client include the ability to send user given inputs, as well as the ability to display the resulting output image from the complete game rendering process.

In addition to needing to reserve processing power to encode the video sequence, a cloud gaming application must also be able to render the entire virtual scene. Depending on the scene complexity and resolution, this virtual scene may utilize most of the available processing power and leave little remaining for rendering and encoding the resulting video data. This is the primary limitation to modern cloud gaming that faster graphical processors are trying to address.

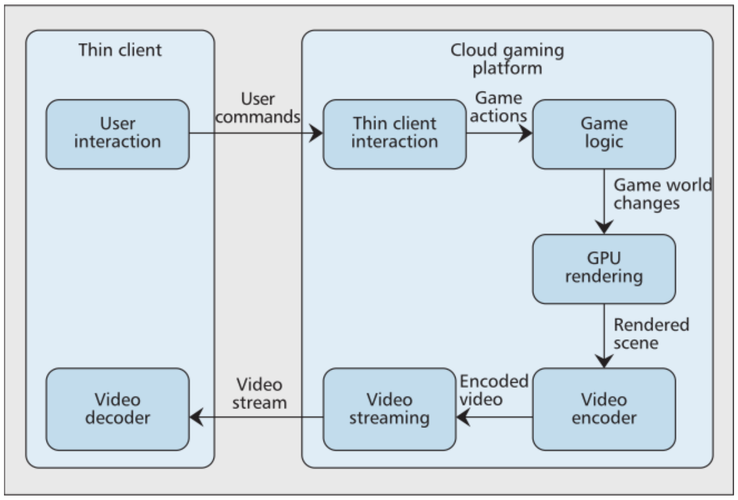


Figure 2.3. Cloud gaming architecture. (Shea et al., 2013, p. 18)

Huang, Hsu, Chang, and Chen (2013) developed an open-source cloud gaming system to stimulate research and innovation in the field of cloud gaming. The solution prioritizes minimizing the frame processing delay for a smooth render. Their solution utilized FFmpeg, a C++ library responsible for encoding and streaming audio and video (FFmpeg Developers, 2018). This software is a powerful video encoding and decoding tool, and its use is consistent across other cloud gaming solutions such as Flashback (Boos, Chu, & Cuervo, 2016) and Furion

(Lai, Hu, Cui, Sun, & Dai, 2017). The FFmpeg library is a common video encoding and decoding scheme utilized in various cloud gaming architectures. Criticisms of Flashback (Boos et al., 2016) include the lack of available interaction in application. The user is limited to only watching a video, as opposed to being able to interact within the scene itself. This cuts out latency delay from the pipeline, but the lack of interaction leads to a minimized immersive experience and lowered telepresence.

Furion (Lai et al., 2017) utilized a method called cooperative rendering, which selectively chooses which items in the scene should be given processing priority. Objects that require less interaction delay, those that are up closer to the viewer, are rendered on the mobile thin client. Considering the entirety of the virtual scene typically will require more power to render, the remaining non-priority objects are rendered via the server and return to the thin client as a panorama image. The thin client then processes the received image. This method is practical and simple; however, a criticism of the work is the requirement to be connected to a 150 Mbps network or greater to have latency free playback. At the time of this writing, this speed is not easily attainable by the average person, especially those on a typical connection. The average internet connection speed in the U.S. around 25 Mbps according to United States Speedtest Market Report (2017).

In general, one of the appeals of cloud gaming is its accessibility. Using a thin client, a user without local processing power available may have access to a remote immersive virtual environment of a much higher fidelity. An advantage of this technology is that it can access big data set through a lightweight mobile device, not powerful enough to handle the entirety of the processing, but just powerful enough to ping the remote server and display the rendered or calculated image.

In order to improve the computational offloading component present in most cloud gaming systems, utilizing a mobile device for this type of data handling is a logical solution for practical application. Moving computing power onto the cloud while offloading data from the mobile device, known as mobile cloud computing, is a beneficial way of bridging the hardware limitation gap imposed by the less powerful mobile device (Ma, Zhao, Zhang, Wang, & Peng, 2013). Accessing data via a mobile terminal and letting the remote server process the larger data, such as higher resolution graphical renders, is a potential software approach to a more practical distribution of the rendering data.

Performing real-time video encoding for mobile cloud gaming is not without its complications. One group has found the main goal of building these systems is to reduce interaction latency, the perceived delay between when a user inputs a control and expects to receive the response. Many gaming experiences require real-time interactions at fractions of seconds, so on-demand rendering methods seems to be the sensible approach. Shi, Hsu, Nahrstedt, and Campbell (2011) built a mobile cloud gaming system that considers rendering viewpoint and pixel depth of objects. The original view matrix is used to generate the first image, and pixel depth is tested to determine required resolution, where closer objects appear higher quality, and objects in the background require less frequent display updates. The authors also used camera input motion and head speed to generate additional frames using motion interpolation between the fully rendered true frames.

Interaction delay must be kept as short as possible for an ideal experience. The shorter the player’s tolerance for interaction delay, the less time that can be spent on game rendering and video compression. Fast paced action games that require quick responsive interactions demand even less time that can be spent processing the image. The higher network latency can negatively affect a player's experience of interaction (Shea et al., 2013).

A mobile device is typically less powerful in terms of rendering ability when compared to high-end stationary computer hardware; however, utilizing this concept of mobile cloud computing becomes more realized as more people around the world begin to utilize smartphones in their everyday life.

2.5 Measurement Tools

Chen, Chang, and Tseng (2011) developed a simplified method for measuring latency in terms of total response time delay in the context of cloud gaming. While the perceived delay in the display is typically described as latency, this methodology breaks latency apart into three parameters to measure the total additions from multiple types of latency sources. The delay metrics include network delay, processing delay, and playout delay. The total response delay (RD) measurement consists of the total contribution of these individual sources of latency.

Network delay can be described as the delay caused by the limitations in network signal speed. The total network delay includes the round-trip transmission signal time to send the client interaction to the server and back. It is commonly referred to network round-trip time (RTT). This time measurement can be taken using tools like the Internet Control Message Protocol (ICMP), a common tool for measuring response times for networked applications.

Processing delay is the difference in time between receiving the user's command and processing the corresponding frame. In computer graphics, this processing delay is typically considered for all standard 3D rendering including baking meshes and lighting. In a cloud gaming system, this delay occurs within the server, as the remote machine requires time to calculate all the graphical components that make up a 3D virtual environment. This delay also includes the time taken to encode the video content and prepare video packets to be sent back to the client device.

Playout delay is the time taken from when the server signal is received by the client, to the time it takes to display the image to the screen. This delay includes the time taken to decode the received video packet, and then display the raw pixel color information to the screen. In terms of the Address Recalculation Pipeline by Regan and Pose (1994) and View Bypass by Carmack (2014), this is the only delay that should be attributed to total latency in a pipelined rendering system. The two graphics algorithms provide methods for superseding the first two parameters, which is a solution for bypassing these forms of latency; however, all three parameters should be considered when calculating response delay (RD) in final latency determinations. The sum of these elements is the total RD of the system.

The difference in image quality between the control and experimental states can be calculated using the structural similarity measurement metric (SSIM), an index measurement from 0 to 1 that can be calculated by measuring the difference between synced video frames. This procedure is based on means and variances within pixel neighborhoods to determine a quantifiable value for effectiveness. If this index remains high, the image structural difference between the two renders will be small, indicating a similar displayed image classification. A quantitative quality assessment of the image can be inferred by measuring the visibility of errors between two image frame datasets (Wang et al., 2004). A SSIM value of 0.9 or greater indicates good visual quality. Any value below 0.86 is where the human eye begins to notice a decrease in quality (Cuervo, Wolman, Cox, Lebeck, Razeen, Saroiu, & Musuvathi, 2015).

By quantifying immersion, there is a rationale for increasing display speed and reducing latency for users playing games. Latency has been studied extensively over the last few decades.

Even with improvements to graphics, the described latency mitigation techniques still are relevant to rendering today. As for video streaming, three different protocols are discussed, expressing benefits and drawbacks of the different systems. For cloud gaming, finding a way for a mobile device to offload heavy rendering load seems to be a natural bridge for leveraging the mobile capabilities of a mobile device but still achieving console quality rendering. Finally, the measurement tool SSIM has been described as a metric for measuring perceptual difference in image quality.

CHAPTER 3. METHODOLOGY

In this chapter, the methodology of the study is presented including the main hypothesis. The following section will detail the immersive 3D virtual environment used for this study. This section also describes some graphically intensive aspects of the scene that make a mobile-only render solution impossible for viable playback with the given environment. The cloud gaming framework used in this study is described within the context of the Unity game engine. Finally, the data collection and analysis sections depict the way the data is gathered, stored, and processed for this study.

3.1 Hypothesis

The output frames from a cloud gaming framework utilizing the View Bypass algorithm will have a greater image structural integrity value compared to output frames of the same virtual environment without the View Bypass algorithm.

3.2 Immersive Virtual Environment

The experimental immersive virtual environment exhibits a rendering demand greater than what is possible for modern mobile devices; however, the environment is easily rendered by the powered desktop. The test environment includes a single 3D scene constructed in the Unity3D (Unity Technologies, 2019) game engine using the Unity Asset Store asset, Nature Starter Kit 2 (Shapes, 2018). The chosen environment resembles a small forest, including foliage such as grass, trees, and bushes. The ability to instance foliage on the GPU is much more achievable with a powered desktop compared to a mobile graphics card. Lighting is processed in real-time on the server. Shadows are soft but are set to low resolution to increase the rendering processing rate. Processing demands are introduced by the overlapping, alpha-masked tree branches and leaves that require depth textures to render the proper object outline, another process not as easily attainable by the more lightweight mobile graphics card. All animations and movement in this scene are entirely controlled by camera motion, not including object animations to minimize the variability of the final rendered image. A drawback of this scene is it does not include user interaction, though not doing so may minimize the amount of CPU overhead from a user control scheme.

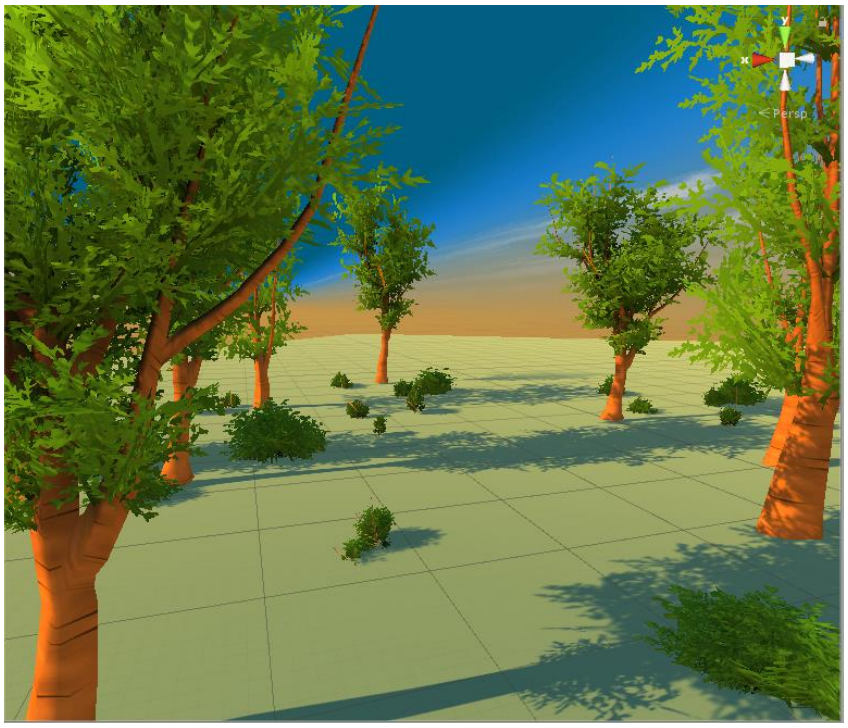


Figure 3.1. 3D Environment used for this study.

An Android build was produced for the mobile device that attempts to render the given scene locally. Due to the graphically intensive nature of the scene, the mobile build was only able to display a few frames before stalling, crashing, and forcing the application to quit. Some factors such as per-object depth textures, real-time lighting and shadows, and the instanced foliage contributed to an unplayable application for the mobile-only renderer. Thus, if the scene is rendered on a remote powered desktop, the client can display the video stream so that the environment can be viewed on the less powerful mobile device.

3.3 Cloud Gaming Framework

The cloud gaming framework utilized in this study resembles the pipeline as shown in Figure 2.3. For the information held by the server, it includes environmental data such as lighting, mesh, and material information, standard for a game render and does not deviate much from how a game engine normally functions. The server stores a proxy camera, to be the main render camera, at origin until the client connects and the test begins. The client's starting information includes the character animation and starting location, as well as a transform sender that is attached to the head transform of the character in motion. This transform sender sends the position and rotation information to be picked up by the server proxy camera. The client's display is a 3D quad mesh that occupies screen space and receives the server frame updates.

Before the connection is made, the server listens for the client. Once the client connects to the server, the server proxy camera remains stationary but begins to feed frame data over the network to the video texture displayed on the client.

In a single frame, the client-stored current camera transform including position and rotation is encoded into a byte array to be sent over to the server. Once the server receives the transform, the information is decoded and used to set the current transform of the server proxy camera. From this updated transform, the server proxy camera renders a single frame using the materials and shaders supplied for the environment and is saved to a render texture. A byte array stores the raw texture data read from this texture and is encoded with the VP8 codec. Finally, the encoded texture gets sent as a packet over the network via WebRTC to be received by a frame update event handler on the client. Once the frame is received, the client quickly decodes the texture and updates the client display. The shader attached to the client display renders colors in ARGB32 format to maintain the texture format rendered on the server and displayed on the mobile client. Utilizing textures with lower color bit depth may introduce additional processing savings in the system; however, for this study, a higher bit depth texture format was chosen to maximize the processing load for texture rendering on the powered desktop and contributing to the overall color precision. Back face culling is also enabled for optimization. After this, the next frame begins.

For each frame captured onto the 1024x1024 render texture, the raw texture data is read and stored in a byte array. The uncompressed texture size per frame at this resolution is 4.19 MB, meaning the cloud gaming application must be able to handle 125.7 MB of data every second to maintain a 30 frame per second rate. If this data were streamed uncompressed across the network, an internet speed of approximately 1Gbps would be required to effortlessly stream 1024x1024 uncompressed at 30 fps. If a resolution of 2048x2048 were used, a speed of 4Gbps would be required, and this value doubles to 8Gbps at 60 fps. The choice to use WebRTC as the video streaming protocol is informed by these currently unattainable internet speeds. Due to the nature of the WebRTC algorithm, bandwidth and network congestion affects the encoding bitrate for the VP8 codec. Due to the WebRTC’s compression algorithm, bitrate is not explicit or controllable outside the native code; however, even with large uncompressed textures being passed through the WebRTC protocol, the actual bandwidth of the system occupies around 200 kbps, accounting for the size of the VP8 audio and video codecs at each packet update. With this internet speed discrepancy, pure raw textures are not actually sent over the network, and the outputs will be affected by the WebRTC congestion algorithm to determine the final image (Holmer & Alvestrand, 2011).

The compression applied to images streamed over the network may be a factor in determining final SSIM output values as the image size must be reduced to fit effectively in a packet. The final image compression is determined by the congestion algorithm and may be a notably lower quality than the control frame captured within the game engine. This is expected when streaming high resolution video content at low bandwidths and should be taken into consideration when analyzing the final data output.

Design choices were made for the purposes of optimizing the application. Most notably, the main application class was split into three classes: camera, display, and event handling. This follows the separation of concerns design principle and allows for asynchronous calls where tasks must work independently of one another. The camera class affects the server proxy camera only and is separate from the client. Event handling controls call information, handles the sending and receiving of data, and listens for the right time to update the server rendered frame. Display handles the client update loop and frame rate, which is not always congruent with the server frame rate. Decoupling these two processes greatly decreased the rendering demand of the client graphics processor as part of the exploration process.

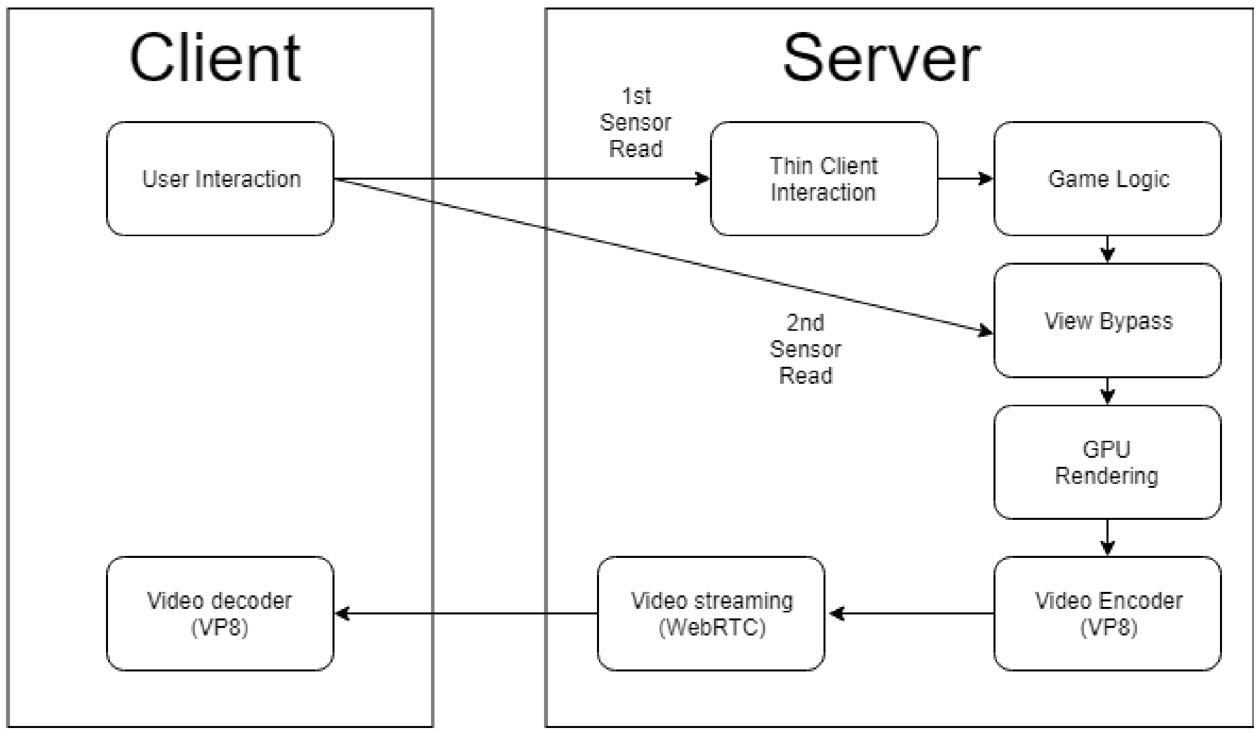
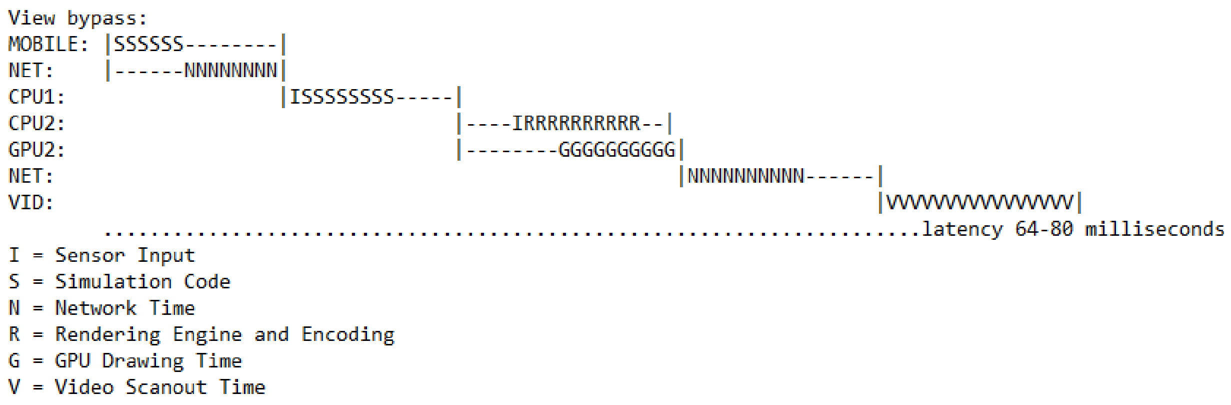


Figure 3.2. Customized cloud gaming framework with View Bypass.

An additional optimization includes solving a bottleneck introduced by the nature of this cloud gaming system. As the client updates the animation frame in the update loop, any increased display latency impacts the playback speed of the animation. Thus, if the frame rate decreased locally, it would proportionally decrease the animation frame rate received by the proxy server camera. To circumvent this issue, the animation update was placed in the fixed update loop to ensure equidistant transform updates for the server proxy camera, creating a more consistent video stream flow from the game engine. Placing the animation update in the fixed update loop also allows for asynchronous user input, so the thread responsible for sending the transform data is not impacted by any frame rate changes in the final display.

Enabling and disabling View Bypass in this system is controlled by a simple flag. When it is disabled, the image frames are rendered and packaged normally. When it is enabled, the

View Bypass algorithm performs the appropriate calculations. Due to the animation updates being sent from the fixed update loop, a fixed time step of 16.67 ms sends two independent sensor reads within a single frame at a target display rate of 30 fps. The first sensor input, generated by the animation at the beginning of the frame loop, is sent from the mobile client to the remote server to initialize the render. After the initial rendering code processes simulation events and game code, a second sensor input is sent from the client to generate the delta. The following algorithm is called using Unity’s OnPreRender, an event function called just before the camera begins the frame render. The position delta is calculated from the difference between the second sensor position and the first sensor position from the beginning of the frame. Similarly, a rotation delta is calculated by multiplying the second input quaternion by the inverse of the first input quaternion. The position delta and rotation delta are used generate a transformation matrix. The camera view matrix is updated by multiplying the camera world-to-local matrix by the generated transform matrix. The result is a transformed image using the most updated parameters for the view matrix.



```
View bypass:
MOBILE: |SSSSSS--------|
NET:    |------NNNNNNNN|
CPU1:                  |ISSSSSSSS-----|
CPU2:                                 |----IRRRRRRRRRR--|
GPU2:                                 |--------GGGGGGGGGG|
NET:                                                     |NNNNNNNNNN------|
VID:                                                                      |VVVVVVVVVVVVVVV|
        ......................................................................latency 64-80 milliseconds
I = Sensor Input
S = Simulation Code
N = Network Time
R = Rendering Engine and Encoding
G = GPU Drawing Time
V = Video Scanout Time
```

Figure 3.3. ASCII Diagram showing View Bypass rendering model.

3.4 Data Collection

The experiment will be broken into 2 treatments: testing the cloud gaming framework with and without the View Bypass algorithm. Both treatments collect frame data and application data per stored frame including client frame rate, server frame rate, and network roundtrip time. In this case, network round trip time can be defined as the time between sending user input from the mobile device to the time the signal is received again (Chen, Chang & Tseng, 2011). Due to the nature of the cloud gaming framework, the processing delay occurs during network roundtrip time, thus subtracting the processing latency from this value gives the total travel time across the network per recorded frame, which is a less obtrusive way for determining network delay.

The experiment utilizes a C# script written to automate a virtual camera moving through the virtual environment, simulating a user's head movements through the world. This motion resembles a camera dolly typically used in film and was chosen to help simulate six degrees of freedom (6DOF), including positional and rotational information. The difference in supporting 6DOF as opposed to three degrees of freedom (3DOF), which only supports rotational movement, means the user is introduced to an environment with objectively higher immersion. Though users are not being observed in this experiment, simulating 6DOF is important to maintain for the significance of virtual environment rendering as the medium evolves to more commonly support this level of immersion.

The virtual camera animations are determined by using human seeded motion tracking data of a person walking, so that the movements through the environment reflect natural movement through a virtual space more accurately. The prerendered animation set was taken from the CMU Graphics Lab Motion Capture Database (2018) utilizing a simple walk cycle with a captured framerate of 120 fps but down-sampled to 30 fps once imported into the Unity game engine. This motion tracking dataset is continually sent to the powered desktop on a fixed frame interval from the mobile device in real-time to simulate the upstream of user head tracking data. The virtual camera starting world position is consistent throughout each trial, at the center of the forest environment.

The virtual camera will not begin to operate and record until a valid connection has been made between the server and client so that the rendering process is communicating between devices properly. Simulated user movement speed is randomized into a speed factor to generate variability within each treatment. This speed factor will be randomized by discrete values from 0.1 meters per second (m/s) to 2.0 m/s, incrementing by 0.1 m/s, to incorporate values slower and faster than typical user walking speed of 1.0 m/s (Regan & Pose, 1994). This speed factor affects the translational speed of the camera, as well as the playback speed of the camera animation. As the speed affects the camera animation playback, some animation interpolation may take place in the Unity game engine as slower speeds may need to extrapolate frames to extend the motion. The speed factor variable is randomly sampled and adjusted for each trial until each discreate speed has been recorded for both treatments at least 5 times.

At each discrete speed, a control image set is prerendered by the server for each treatment. These control images are used to compare the quality of the mobile images produced to determine SSIM values. The method for collecting these images will be the same as the treatment image collection process, but the output frames are stored on the powered desktop before it travels through a network. Due to utilizing the same animation path and starting point, the renders correspond precisely between the control and experimental frames at the congruent moment in time. Comparing the output of a trial at a given discrete speed to this control image set at the same discrete speed will determine appropriate SSIM values.

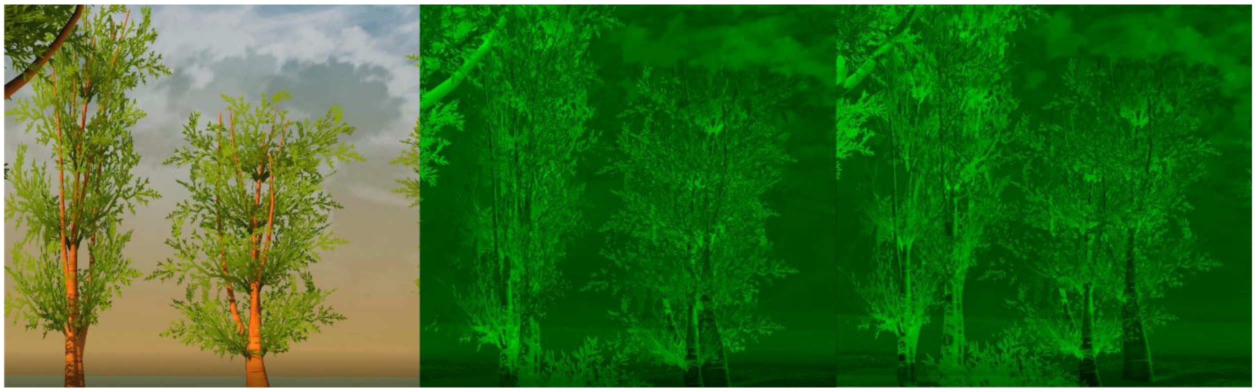


Figure 3.4. Corresponding frame outputs at a speed factor of 1.0.

Each corresponding image frame will generate a SSIM value by using FFmpeg. Each trial should last approximately 30 seconds and will result in 30 SSIM values with an average SSIM score calculated for each trial at each discreate speed.

To remove rendering overhead, frames output for data will not be calculated every frame, rather frames will be stored at a fixed interval, one per second. Once the application starts on the mobile device, blank textures with the same resolution as the video frames are generated. Rather than write these files as .PNG during the trial and risking a decreased frame rate while the image writes to disk, the textures are stored in an array and the color information is updated periodically based on the most recent sample of the video texture. Once the trial concludes, the mobile device writes the stored textures to the internal device as .PNG files, and the application outputs the above-mentioned metrics as a .TXT file on the mobile device. The .PNG image sets and .TXT corresponding to each trial are stored respectively to be used for data analysis.

Total system latency is determined using the method proposed by Chen, Chang, and Tseng (2011) stating: Total Response Delay (RD) = Network Delay (ND) + Processing Delay (PD) +

Playout Delay (OD). ND is determined by using the Unity ping function, which is an asynchronous operation that polls for a connection status with a given IP address. For this study, the IP address of the server is directly given to the client for the connection. PD is calculated and determined by the Unity Game Engine and recorded for each trial. PD represents the framerate of the powered desktop as it renders the game. OD represents the local frame rate of the mobile device, independent of the streamed server frame rate. Client fps, server fps, and ND are output at the end of each trial.

3.5 Data Analysis

For each trial, each set of .PNG files correspond to a pre-rendered control .PNG set. This file format is used to construct a .PNG sequence of the collected frames in FFmpeg. Instead of analyzing each frame individually, time is saved by combining the 30 frames into a single sequence that can be analyzed against the control sequence in a single command. Using FFmpeg with command line, each experimental and control frame at each respective moment in time are compared and output a SSIM value. This value is an index between 0 and 1. Each trial generates 30 SSIM values, one for each frame comparison made. Each value is recorded to a spreadsheet and the mean value of each trial contributes to determining the overall image structural integrity.

For hypothesis testing, a two sample for means test is used to determine significance, comparing the View Bypass data to the control data with n = 100. If SSIM values are significantly greater in the View Bypass treatment compared to the control, the null hypothesis is rejected.

The ideal server frame rate is set to 30 fps. If this value drops, it is not indicative of the client frame rate, also ideally set to 30 fps. Drops in server frame rate indicate render processing or network bottleneck issues. Drops in client frame rate indicate display processing issues. Both frame rates are recorded at the end of the trial and should be analyzed as a support metric to determining the effectiveness of this experimental cloud gaming framework.

This chapter stated the main hypothesis, described the 3D environment used for this study, and depicted the cloud gaming framework used for the application. The data collection process of frame gathering during playback goes into detail about an optimized method for capturing frames without slowing down the device or making unnecessary disk writes. Finally, the data analysis section described using SSIM as a quantifiable image quality measurement tool.

CHAPTER 4. RESULTS

The following chapter shows the captured data from this study. SSIM values show the image quality results. Each table displays mean values of the 5 trials tested per given discrete speed.

4.1 Data

The following tables show results from the experiment including mean SSIM values across all speed factors, mean server and client frame rate, and mean latency factors determining total system latency. Table 4.1 shows mean SSIM values across all trials for each treatment at each discreate speed factor. The mean SSIM value for the No View Bypass treatment is 0.6446. The mean SSIM value for the View Bypass treatment is 0.6327, a mean difference of .0009.

Table 4.1 Mean SSIM Values

Speed Factor	View Bypass	No View Bypass	Difference
0.1	0.5909	0.6007	-0.0098
0.2	0.6505	0.6589	-0.0085
0.3	0.6779	0.6973	-0.0194
0.4	0.6815	0.5936	0.0878
0.5	0.6618	0.7120	-0.0502
0.6	0.6041	0.6259	-0.0217
0.7	0.6016	0.6057	-0.0041
0.8	0.6514	0.6770	-0.0256
0.9	0.6380	0.6180	0.0200
1.0	0.6346	0.6449	-0.0103
1.1	0.6424	0.6451	-0.0026
1.2	0.6188	0.6445	-0.0257
1.3	0.6273	0.6498	-0.0225
1.4	0.6229	0.6350	-0.0121
1.5	0.6512	0.5990	0.0522
1.6	0.6130	0.6008	0.0122
1.7	0.6125	0.6253	-0.0128
1.8	0.6203	0.6133	0.0070
1.9	0.6270	0.6163	0.0107
2.0	0.6256	0.6089	0.0167

Table 4.2 shows mean server and client frame rates. The mean client frame rate for the No View Bypass and View Bypass treatment is 29.69 fps and 29.76 fps, respectively. The client fps is the local frame rate of the mobile device. The mean server frame rate for the No View Bypass is 25.18 fps, and the mean server frame rate for View Bypass is 25.97 fps. The server fps is the rate of the streamed video frames over the network.

Table 4.2 Mean FPS of Client and Server

Treatment	Client FPS	Server FPS
No View Bypass	29.69	25.18
View Bypass	29.76	25.97

Table 3.3 shows mean latency factors such as network delay (ND), processing delay (PD), output delay (OD), and the summation of the three factors. Total system latency for this cloud gaming framework for No View Bypass is 74.12 fps and 74.11 fps for View Bypass. ND shows the results of the Unity ping function between the mobile client and desktop server, in this study performed in a local arena network. PD shows draw speed taking about 6.88 ms on average for each treatment which is equivalent to 145 fps for the powered desktop rendering the game. OD is determined by calculating the inverse of Client FPS from Table 4.2.

Table 4.3 Mean Latency Factors

Treatment	ND (ms)	PD (ms)	OD (ms)	Total (ms)
No View Bypass	33.5477	6.8887	33.6874	74.1238
View Bypass	33.6168	6.8833	33.6072	74.1073

The justification for using this level of precision is determined by the small differences in delay amounts within the results. As the results showed the same number of milliseconds, it is important to have more decimal precision to see the differences in the data. Having additional precision does not seem necessary as the numbers begin to describe microsecond differences in timing, which ultimately may not yield a perceptual difference in the display. With less precision, the values would not show any difference. While the differences are not vast, there are minor differences that should be highlighted and shown by this level of decimal precision.

Figure 4.1 depicts the distribution of data for View Bypass and No View Bypass. The graph shows that most of the data points fall under the 0.60 to 0.65 range for SSIM across both treatments. A notable outlier with a value of about .712 occurred in the No View Bypass treatment with a speed factor of 0.7.

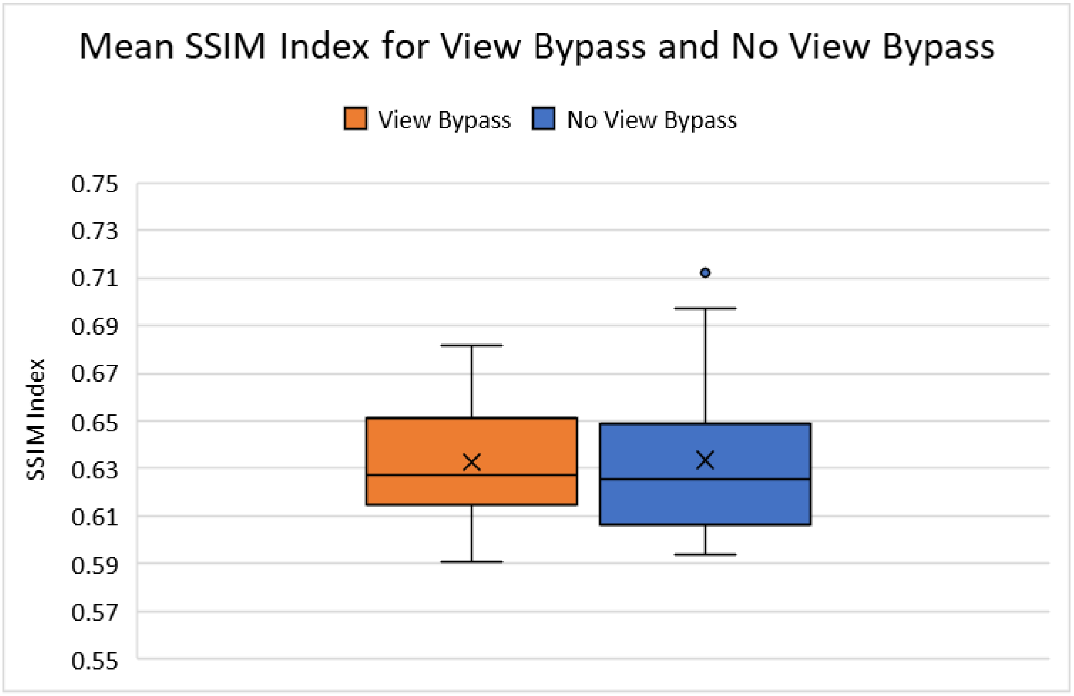


Figure 4.1. Mean SSIM Index for View Bypass and No View Bypass Treatments.

Figure 4.2 is a histogram showing the distribution of the data between treatments. The graph shows the two treatments have a similar distribution, both skewed to the right. Figure 4.3 shows mean SSIM values over each discrete speed factor. It depicts higher values and variability at lower speed factors and shows more consistent results at higher speed factors. The figure represents the mean of 5 trials run at each discrete speed factor.

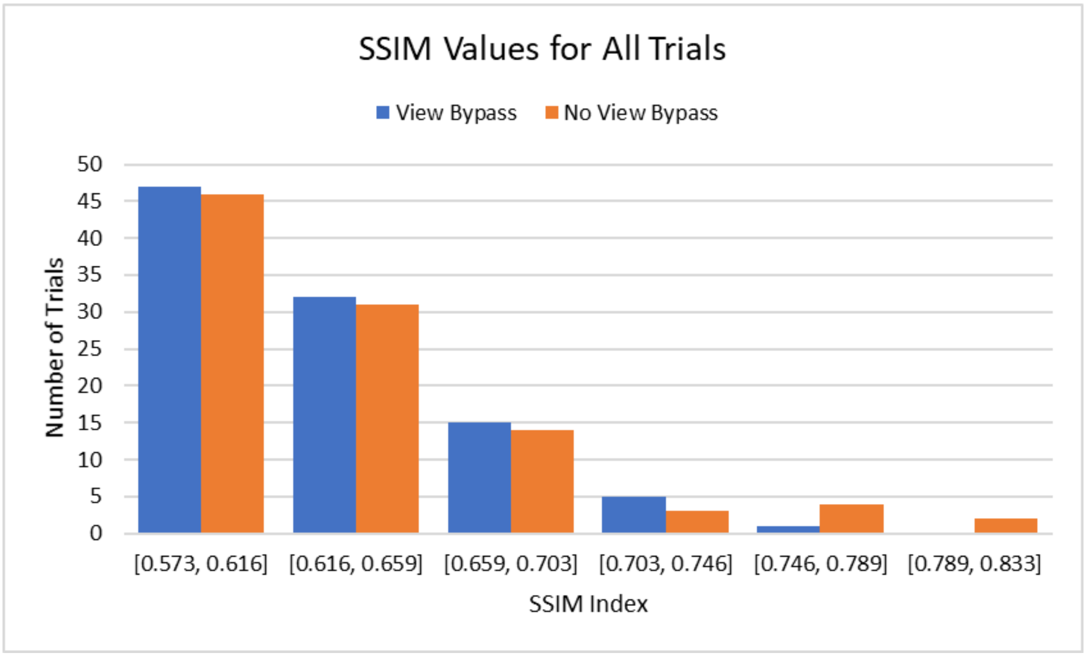


Figure 4.2. Histogram of SSIM Values for All Trials.

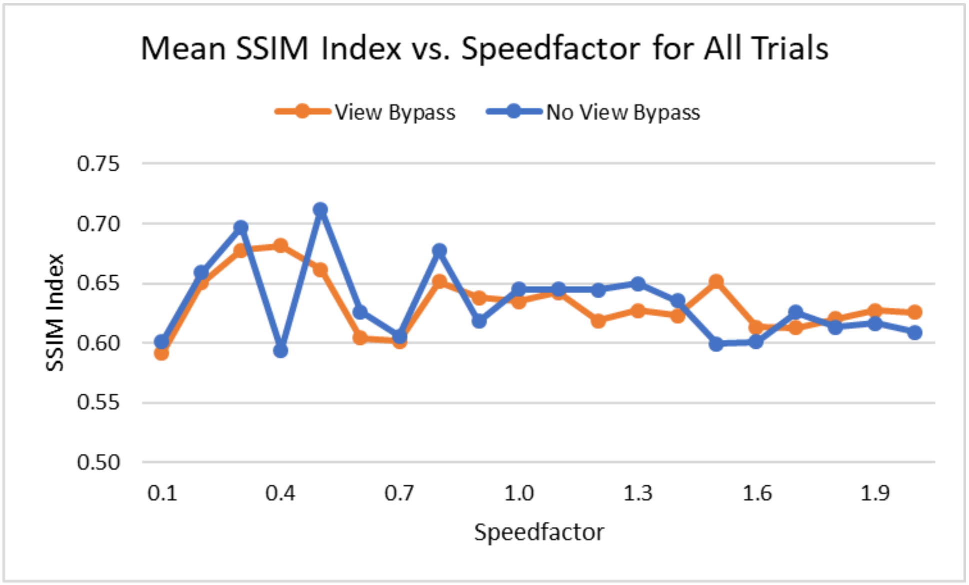


Figure 4.3. Mean SSIM Index vs. Speedfactor for All Trials.

Figure 4.4 expresses the mean server frame rate across all speed factors showing higher frame rate consistency for the View Bypass algorithm. The server frame rate is the speed of the streamed video data and is dependent on factors contributed by the server and network connection. Figure 4.5 depicts mean client frame rate across all speed factors and shows less variability between treatments. View Bypass maintains a slightly higher client frame rate over the No View Bypass treatment, despite both methods processing approximately the same amount of video data.

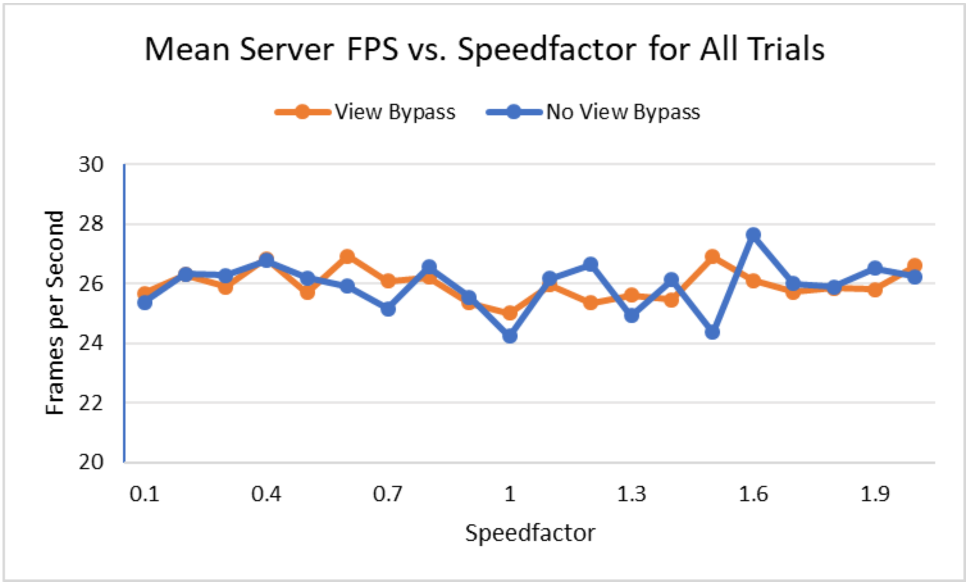


Figure 4.4. Mean Server FPS vs. Speedfactor for All Trials.

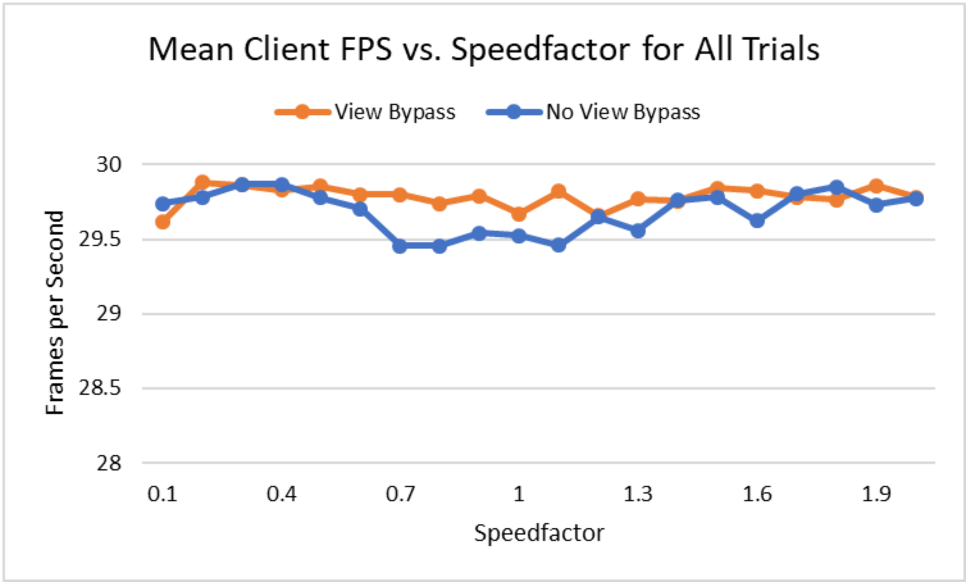


Figure 4.5. Mean Client FPS vs. Speedfactor for All Trials.

For the two sample for means test, the average of all trials at each discrete speed factor are determined and used to test a significant difference between the means for each treatment. An alpha value of 0.05 is used ($\alpha = 0.05$) for a confidence level of 95%. With 19 degrees of freedom, variance is calculated to be .0011 for No View Bypass and 0.0006 for View Bypass. With these values the Z critical value is determined to be 1.645 with a test statistic of .101, resulting in a p-value of .4596.

CHAPTER 5. DISCUSSION

The following chapter discusses the results of this study. The analysis section describes the statistical analysis and the resulting significance determined from the data. The conclusion section depicts the benefits and drawbacks of the cloud gaming framework in the context of WebRTC video streaming. Finally, future work tells of potential ways this work may be expanded or improved later in time.

5.1 Analysis

Overall the results show minor SSIM differences in means suggesting a small change in image quality between the two treatments. With a mean difference of .009, the No View Bypass treatment performed slightly better than the View Bypass treatment for the final SSIM value. As the test statistic of .101 falls below the Z critical value of 1.645, the calculated p-value of 0.4596 does not below the alpha value of ($\alpha = 0.05$). Therefore, the null hypothesis is not rejected. The output frames from a cloud gaming framework utilizing the View Bypass algorithm does not have a greater image structural integrity value compared to output frames of the same virtual environment without the View Bypass algorithm.

The box plot depicted in Figure 4.1 shows how similar the data is between the two treatments with a congruent distribution. Figure 4.2 depicts that across discrete speed factors, there seems to be an effect on final image quality; however, between the two tested treatments, the SSIM values correlate strongly and do not differ at each corresponding speed factor. At lower discrete speed factors, values fluctuated more and yielded higher values compared to values yielded in the higher speed factor range. An explanation for this could be at slower speeds, the video stream rendering the final output makes less drastic pixel deltas, allowing for available bandwidth to be used to improve the overall image quality. The SSIM values fluctuated less at higher discrete speed factors but does not decrease in image quality with increasing speed.

While View Bypass does not have a significant effect on final image quality, it does seem to affect server frame rate. The mean server frame rates between treatments only differ by about .02 fps, so there is not a practical increase to frame rate with the View Bypass algorithm. The overall server frame rate is mostly dependent on the speed of the powered desktop; however, trials using the View Bypass algorithm yielded more consistent frame rates for the frames rendered by the server. Figure 4.3 depicts the server frame rate consistency compared to the more variable frame rate for the No View Bypass treatment.

Client frame rate is largely independent of the factors of this study and dependent on the output speed of the given mobile device. Figure 4.4 shows that client frame rate was close to reaching the mobile device limitation of 30 fps across all trials. Client frame rate was unaffected by changes made to the server and managed to render on average within a single frame of the maximum frame rate for all trials due to the nature of the cloud gaming rendering method. Similar to client frame rate, Figure 4.5 shows that network delay did not deviate between treatments or speed factors. The network delay value remained consistent, about 33.5 ms, because the testing occurred on a local area network and did not need to travel to a remote server. Outside of the controlled experimental environment, this value would fluctuate more depending on the connected server and may net values much higher and more variable than the output from this experiment.

5.2 Conclusion

The software system created for this study functions appropriately as a cloud gaming framework, but optimization could be extended, and improvements may result in a more powerful system. The drawbacks to using this system concern sacrificing quality for speed. The rationale is the priority for players interacting with games is to receive and display game information at the appropriate time, as opposed to reaching ideal quality with latency and risking frame skips. Using WebRTC as the video streaming protocol is beneficial to supporting this priority, maintaining a consistent frame rate over ideal resolution; however, due to the data congestion algorithm inherit to the WebRTC architecture, quality loss occurs with low network bandwidths. Additionally, as the WebRTC API is high level, there is no open access to adjusting the encoding bitrate manually, and the render quality is altered dynamically depending on the available bandwidth. The VP8 codec responsible for encoding the stream requires about 200 Kbps for encoding, and due to code protections, higher bitrates could not be set manually and were dependent on the given network speed. Ideally the available network bandwidth would be notably higher to achieve images with a more ideal resolution while maintaining frame rate.

Overall, the compression algorithm that makes WebRTC efficient may have been the greatest contributing factor to quality loss within this study. As the underlying algorithm expects low bandwidths for rendering, one might expect much higher bandwidths to have almost no compression artifacts or pixeled frames for the final streamed image result.

Some limitations affecting the cloud gaming system include deciding which data should be sent over the network. Transform data is sent to generate camera movements. Video frames are sent back to the client for the display. Ideally, the video texture sent back to the client contains a depth texture with motion vectors that may be used for motion estimation of objects; however, not all modern phones have the graphics capability of properly utilizing depth textures for this purpose. Due to this current technical limitation, view bypass must be performed on the renderer with access to the vertex buffer to achieve accurate and undistorted frame results.

Some technical advancements must be made to further reduce the latency of cloud gaming systems. Primarily, the largest contributor to total latency delay is the frame rate of the display device. For this study, utilizing a mobile device that renders with a 30 FPS display means that the output delay may not be reduced below around 33.33 ms. As device frame rates continue to improve, the quantity of this delay will be reduced considerably. The other major contributor to total latency delay is network delay. For the results of this study, the network delay is consistent and precise, as the data collection occurred over a local area network, as opposed to connecting to a distantly remote server. Additionally, the speed of Wi-Fi must be improved to reduce the overall latency of the system as the client device must talk to the local router, resulting in additional steps for the large data packets to take. Mobile phones generally are not suitable for a wired connection and rely on Wi-Fi for connection. Wireless data transmission must improve to handle larger data communication, achieving better quality streams and reducing overall latency.

5.3 Future Work

Future work may involve testing to find the ideal video streaming protocol for cloud gaming. The best solution for streaming cloud games is not necessarily the best solution for streaming standard video. In addition to testing WebRTC on higher bandwidth networks, future studies might look at utilizing DASH as a video streaming method. A future study that tests the VP8 codec against VP9, H.264, H.265 or future video compression standards may show the ideal encoding method for cloud gaming. While not observed in this particular study, it is worth examining the ability of using LTE or 4G in comparison to Wi-Fi to see if there is a noticeable affect on congestion on different types of networks. Additionally, as display resolutions improve, cloud gaming systems will render with higher resolutions and faster frame rates. Future studies might investigate more optimized compression algorithms that produce higher quality images with smaller data packet size.

Future studies should also look at determining the quality of experience as perceived by users engaged with a cloud gaming system. For this study, the 74 ms delay might be quite noticeable by a user accustomed to faster frame rates. This delay could be usable for slow-paced games, but first-person shooters or virtual reality games may not be enjoyable with this latency. Adding motion estimation and predicting images based on movement might improve the quality perception of the user. Motion estimation may also interpolate the playback of the server frame rate potentially creating a smoother experience for the user.

A more pragmatic approach to improving future cloud gaming systems is by utilizing additional frameworks such as asynchronous time warp and cooperative rendering. Carmack (2014) describes View Bypass as a method that works best in tandem with asynchronous time warp and could lead to more impactful latency reductions when put together into the same system. Cooperative rendering should be potentially incorporated into this system for more rendering savings. Mobile phones, while not as powerful as a desktop, generally have available processing power in addition to decoding and displaying a video stream. If interactable objects or objects close to the user were rendered locally and were supported by a set of streamed panoramas for the remaining parts of the scene, the rendering load would be shared. Future studies should look at user focus and determining if a combined solution of compositing local renders with streamed panoramas would be noticeable for a user. This study could be tested in virtual reality to test the impact of 6DOF within the system. This combined system might also offload the rendering bottleneck from graphically demanding virtual reality.

Additionally, this framework may be improved by allowing a more direct sensor read for the given view matrix, instead of using values generated by the chosen game engine. For a VR application, directly accessing the head mounted display or motion controls may lead to minute latency reductions by limiting the number of steps in between the input sensor and the renderer.

Cloud gaming is becoming more viable as network speeds increase and graphics rendering improves. With additional improvements, gaming may no longer be tied to local hardware. Disassociating gaming from hardware removes the burden of part cost, space, and the inevitable failure of an aging machine breaking down. As cloud gaming begins to integrate into our technological landscape, video compression methods and optimized network solutions combined together will yield better quality resolutions, faster frame rates, and better user experiences.

REFERENCES

Boos, K., Chu, D., & Cuervo, E. (2016). FlashBack: Immersive Virtual Reality on Mobile Devices via Rendering Memoization. International Conference on Mobile Systems, Applications, and Services, 291–303. https://doi.org/10.1145/2906388.2906418

Bowman, D. a, Mcmahan, R. P., & Tech, V. (2007). Virtual Reality: How Much Immersion Is Enough? (Cover story). Computer, 40(7), 36–43. https://doi.org/10.1109/MC.2007.257

Carmack, J. (2014). Latency Mitigation Strategies. Retrieved from https://web.archive.org/web/20140719053303/http://www.altdev.co/2013/02/22/latency-mitigation-strategies/

CMU Graphics Lab Motion Capture Database. (2018). Retrieved from http://mocap.cs.cmu.edu/

Chen, K.-T., Chang, Y.-C., & Tseng, P.-H. (2011). Measuring the latency of cloud gaming systems. Proceedings of the 19th …, 1269–1272. https://doi.org/10.1145/2072298.2071991

Cuervo, E., Wolman, A., Cox, L. P., Lebeck, K., Razeen, A., Saroiu, S., & Musuvathi, M. (2015). Kahawai: High-Quality Mobile Gaming Using GPU Offload. MobiSys, 121–135. https://doi.org/10.1145/2594368.2601482

FFmpeg Developers. (2018). Retrieved from http://ffmpeg.org/about.html

Friston, S., Steed, A., Tilbury, S., & Gaydadjiev, G. (2016). Construction and Evaluation of an Ultra Low Latency Frameless Renderer for VR. IEEE Transactions on Visualization and Computer Graphics, 22(4), 1377–1386. https://doi.org/10.1109/TVCG.2016.2518079

Fund, F., Wang, C., Liu, Y., Korakis, T., Zink, M., & Panwar, S. S. (2013). Performance of DASH and WebRTC video services for mobile users. 2013 20th International Packet Video Workshop, PV 2013. https://doi.org/10.1080/02681219380000131

Heeter, C. (1992). Being There: The Subjective Experience of Presence. Presence: Teleoperators and Virtual Environments, 1(2), 262–271. https://doi.org/10.1162/pres.1992.1.2.262

Holmer, S., & Alvestrand, H. (2011). A Google Congestion Control for Real-Time Communication on the World Wide Web. https://tools.ietf.org/html/draft-alvestrand-rtcweb-congestion-00

Huang, C., Hsu, C., Chang, Y.-C., & Chen, K.-T. (2013). GamingAnywhere: An Open Cloud Gaming System. Proceedings of the 4th ACM Multimedia Systems Conference on - MMSys '13, 2(3), 36–47. https://doi.org/10.1145/2483977.2483981

Kalman, M., Davies, G., Hill, M., & Pracht, B. (2017). Introducing LHLS Media Streaming – Periscope Code – Medium. Medium. Retrieved from https://medium.com/@periscopecode/introducing-lhls-media-streaming-eb6212948bef

Knerr, B. W. (2007). Immersive Simulation Training for the Dismounted Soldier for the Behavioral and Social Sciences A Directorate of the Department of the Army Deputy Chief of Staff , G1.

Lai, Z., Hu, Y. C., Cui, Y., Sun, L., & Dai, N. (2017). Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices. ACM International Conference on Mobile Computing and Networking, 409–421. https://doi.org/10.1145/3117811.3117815

Liu, Y., Guo, Y., & Liang, C. (2008). A survey on peer-to-peer video streaming systems. Peer-to-Peer Networking and Applications, 1(1), 18–28. https://doi.org/10.1007/s12083-007-0006-y

Ma, X., Zhao, Y., Zhang, L., Wang, H., & Peng, L. (2013). When mobile terminals meet the cloud: Computation offloading as the bridge. IEEE Network, 27(5), 28–33. https://doi.org/10.1109/MNET.2013.6616112

Nurminen, J. K., Meyn, A. J. R., Jalonen, E., Raivio, Y., & Garcıa Marrero, R. (2013). P2P media streaming with HTML5 and WebRTC. 2013 IEEE Conference on Computer Communications Workshops, 63–64. https://doi.org/10.1109/INFCOMW.2013.6970739

Regan, M., & Pose, R. (1994). Priority rendering with a virtual reality address recalculation pipeline. Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH ’94, 155–162. https://doi.org/10.1145/192161.192192

Shapes (2018). Nature Starter Kit 2. Unity Asset Store. Retrieved from https://assetstore.unity.com/packages/3d/environments/nature-starter-kit-2-52977

Shea, R., Liu, J., Ngai, E., & Cui, Y. (2013). Cloud gaming: Architecture and performance. IEEE Network, 27(4), 16–21. https://doi.org/10.1109/MNET.2013.6574660

Shi, S., Hsu, C.-H., Nahrstedt, K., & Campbell, R. (2011). Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming. Proceedings of the 19th ACM International Conference on Multimedia - MM ’11, 103. https://doi.org/10.1145/2072298.2072313

Steinbach, E., Farber, N., & Girod, B. (2001). Adaptive playout for low latency video streaming. International Conference on Image Processing, 1, 962–965. https://doi.org/10.1109/ICIP.2001.959207

Steuer, J. (1992). Defining Virtual Reality: Dimensions Determining Telepresence. Journal of Communication, 73-93.

Unity Technologies (2019). Unity3D. Retrieved from https://unity3d.com/

United States Speedtest Market Report. (2017). Retrieved from http://www.speedtest.net/reports/united-states/

Wang, Z., Bovik, a C., Sheikh, H. R., & Simmoncelli, E. P. (2004). Image quality assessment: form error visibility to structural similarity. Image Processing, IEEE Transactions on Image Processing, 13(4), 600–612. https://doi.org/10.1109/TIP.2003.819861

Zhihan, L., Yin, T., Han, Y., Chen, Y., & Chen, G. (2011). WebVR-web virtual reality engine based on P2P network. Journal of Networks, 6(7), 990–998. https://doi.org/10.4304/jnw.6.7.990-998