B.Comp. Dissertation

Hardware Ray-Tracing Assisted Hybrid Rendering Pipeline for Games

By

Tan Chong Han, Alden

Department of Computer Science

School of Computing

National University of Singapore

2021/2022

B.Comp. Dissertation

Hardware Ray-Tracing Assisted Hybrid Rendering Pipeline for Games

By

Tan Chong Han, Alden

Department of Computer Science

School of Computing

National University of Singapore

2021/2022

Project No: H1351930

Advisor: Prof. Anand Bhojan

Deliverables:

Report: 1 Volume

Abstract

The rise in smartphones and virtual or augmented reality technologies has resulted in a shift away from traditional video games. More and more players are playing games on devices that does not have much graphics processing power. Coupled with the advent of 5G networks, there is great potential to improve the rendering techniques for games that are available on these devices by offloading some rendering tasks to a remote server. We implement a novel method of distributed and hybrid rendering where real-time ray tracing is performed on the server to augment the client's rasterised image. Using simulated conditions of 5G edge computing networks in the laboratory, our method manages to provide ray-traced graphics without compromising the interactive frame rates that are required for quality gaming experiences.

Subject Descriptors:

(1) Applied computing → Computers in other domains → Personal computers and PC applications → Computer games
(2) Computer methodologies → Computer graphics → Rendering → Ray tracing
(3) Computer methodologies → Computer graphics → Rendering → Rasterization
(4) Computer systems organization → Architectures → Distributed architectures → Client-server architectures
(5) Computer systems organization → Architectures → Distributed architectures → Cloud computing

Keywords:

Distributed rendering, hybrid rendering, real-time ray tracing, cloud gaming, edge computing

Implementation Software and Hardware:

C++17, Clumsy, DirectX 12, High-Level Shader Language, Microsoft Visual Studio, NVIDIA Falcor, NVIDIA GeForce RTX 2060, Windows 10, Winsock API

Acknowledgements

I would like to express my appreciation for Dr. Anand, for hosting our weekly meetings and providing the team with timely guidance and advice. Also, I am grateful for Yu Wei, who contributed to most parts of the project, reviewing my changes to the source code and authoring multiple papers about the project. Lastly, I would like to thank the other students who worked on the project, namely Nicholas, Sharadh and Tianzuo. The project has been a group effort, and the accomplishments that I had made would not have been possible without their insights and assistance.

Table of Contents

List of Figures

1. Introduction

1.1 Background

Internet speeds are increasing rapidly around the world, and responsive online gaming over fifth-generation (5G) networks is becoming more and more feasible. Based on independent research conducted by her company, marketing executive Bita Milanian (2022) believes that widespread use of cloud gaming services is imminent. Also, graphics processing units (GPUs) that can handle real-time ray tracing such as NVIDIA's RTX series have paved the way for new algorithms, and it is continuing to shape the future of graphics rendering with innovations such as real-time path tracing (Pharr, 2021).

Some key beneficiaries of these technological developments are mobile devices such as smartphones and virtual reality headsets. Currently, mobile devices can only perform the standard graphics rendering pipeline known as rasterisation, as battery-powered devices like them do not have enough power to run conventional chips designed for personal computers (PCs) (Mims, 2020). Our project aims to augment the graphics produced by mobile devices with additional effects from a remote ray-tracing server, and I explore how this can be done without impacting rendering speeds or responsiveness of the pipeline to user input. Such a combined rasterisation and real-time ray-tracing pipeline is also known as a hybrid pipeline, and it is an ongoing research effort in the field. Our graphics pipeline is also a distributed one, since it relies on two different computers communicating over a network.

Our current implementation makes use of ray tracing to generate the visibility data of each fragment, which is then rasterised to produce shadows for the image. I have also experimented with and implemented various strategies to deal with network issues such as latency or packet loss. Since our intended use case involves client mobile devices communicating to a server with real-time ray-tracing capability, in the report the client is associated with the rasterisation part of the pipeline, and the server with the ray-tracing part of the pipeline. To briefly summarise the rendering pipeline, the client starts the rendering process by sending the server information about the location of the camera in the scene, and the server replies with the visibility buffer information for the client to generate the final shaded image.

In Section 2 of the report, I will discuss more about some alternative rendering techniques that could be applied to our target use case, namely interactive video games on

mobile devices, and compare them to our current implementation. Section 3 of the report explains the current implementation of the rendering pipeline, while Section 4 of the report focuses on evaluation of our pipeline's performance. Lastly, Section 5 of the report talks about recommended extensions to the pipeline for future study.

1.2 Factors of Evaluation

For evaluation of our project, we focus on three key measurable variables, namely the responsiveness of the application to user input, the number of frames per second that the pipeline is able to produce, as well as the accuracy of the produced frames as compared to a fully offline rendering solution.

Responsiveness of the application is important for an interactive gaming experience, and it is also linked to the rate at which frames are displayed on the screen. Even in the face of network issues, our rendering pipeline continues to function on the client and to generate images, therefore ensuring that the application can be as responsive as a fully offline rendering pipeline.

Since the technique that we have developed is targeted towards games, high frame rates are crucial for a successful implementation, specifically a value of at least 60 frames per second (FPS). Prior research has found out that gamers are able to perform 14% better in moving target selection at 60 FPS as compared to at 30 FPS (Janzen & Teather, 2014). Currently, we have achieved a frame rate of more than 110 FPS, and this will be discussed in the Analysis section, Section 4.

Also, at negligible levels of network latency, our technique would ideally produce images of similar quality as those generated by the established offline real-time rendering pipeline. As network latency increases, the image quality should decrease proportionally up to a certain minimum level, where the image would be fully generated by rasterisation without any contribution from the remote server. To achieve this, several issues with image quality that were present in the previous implementation of our project have been resolved. The pipeline now executes a calibrated switching between different modes of rendering and extrapolates received data.

1.3 Project Objectives

Besides striving for visual quality, our project aims to determine the best way to send graphics pipeline data between two computers. Compression of the data is a must, since the standard 1080p resolution that is adopted by the majority of display devices today contains over 2 million pixels, and the visibility data that real-time ray tracing produces in our pipeline is per-pixel in nature. Without compression, each pixel generates 4 bytes of visibility data, which sums up to a total of 8.29 megabytes (MB) per frame. At 60 frames per second, we would need an incredibly large network bandwidth of 497.66 megabytes per second (MBps), or 3,981.31 megabits per second (Mbps) to run our rendering pipeline successfully, far exceeding the capabilities of normal hardware today. For comparison, in December 2021, the median download speed of internet users in Singapore was measured to be merely 192.17 Mbps (Low, 2022), and it was found to be the highest in the world for that month.

$$4\ B \times 1920 \times 1080 = 8.2944\ MB$$
$$8.2944\ MB \times 60\ Hz = 497.664\ MBps$$
$$497.664\ MBps \times 8 = 3981.312\ Mbps$$

In our current implementation, we have adopted LZ4 compression, which is a Lempel-Ziv type lossless compression algorithm known for its high compression and decompression speeds (Collet, 2020). The results are quite good, and we have achieved an average compression ratio of 97% for our current test scene. Most notably, the reduced data footprint allows for faster data transfer over a network assumed to have constant bandwidth, accelerating the rate at which frames can be produced by the client. We found that the decrease in the time taken for sending a frame's rendering data from the server and receiving the frame's rendering data at the client far outweighs the additional time required for compression at the server and decompression at the client.

On the networking side, we aim to provide a robust system that continues to function well despite possible packet losses or packet corruption. To achieve that, I have modified the program to use the User Datagram Protocol (UDP) to send data over the network, rather than the Transmission Control Protocol (TCP) that had been used in the original implementation. Our application makes use of the Winsock API for Windows.

The use of UDP increases the speed of sending and receiving packets, since it does away with various TCP features that increase reliability at the cost of speed. However, UDP is unreliable, and using UDP means that our program has to handle missing data in the application layer. This could open up some possibility of novel error correction methods that are sensitive to the rendering context, although it is unlikely. Then again, reliability may not be so important when handling frame-based data.

We aim to determine the best way of transmitting data from the GPU to the central processing unit (CPU), for eventual transmission over the network. This may be more of a hardware bottleneck as the current GPU that we have used in the implementation, the NVIDIA GeForce RTX 2060, does not allow for a direct interface between the GPU and the network. NVIDIA has developed technologies such as GPUDirect that allows GPU data to bypass the CPU entirely and access network adapters (Potluri et al., 2013), but currently this is only available for NVIDIA data centre GPUs and not for commercial graphics cards like RTX 2060 (NVIDIA Corporation, 2015).

Lastly, we seek to fix issues with image quality that arise from poor network communication, as briefly mentioned in Section 1.2. One technique that has been implemented is switching between sequential and concurrent modes of operation of the client rendering pipeline on the fly, delaying the final image until the correct data is received from the server, if it is feasible to do so. I have also implemented a prediction pass that extrapolates received data on the client to match the currently rendered frame, producing images that better match the ideal accurate image produced by ray tracing, which we take to be the ground truth.

To simulate issues with the network in a laboratory setting, we have used Clumsy to test our rendering pipeline. Clumsy is an open-source application that makes use of the WinDivert library (Tao, 2021) to redirect packets. It allows us to easily configure an exact amount of network latency or packet loss. Initially, we had implemented an additional pass to suspend frames that were received at the client and simulate a delay, but this was phased out in favour of the use of Clumsy, which was seen as a less intrusive way to achieve this.

2. Literature Review

2.1 Cloud Streaming Services

One way to obtain a better-quality image on a device with limited rendering capabilities is to simply perform all the rendering on a separate server and send the images back to the device as a video stream. There are various commercial products that offer such a rendering service for video games, and they have gained popularity in recent years. We will look at one example of this by Google called Stadia.

Google Stadia is a classic example of a cloud gaming service, where a game is executed remotely on the server and the rendered display output is streamed over the network to the user's device. Inputs such as mouse movements and keyboard presses are recorded and sent to the game application running on the server. However, the drawback of this approach is that it would take one round-trip time on the network for a user's input to affect the image on the screen. Indeed, upon release, Stadia was panned for significant lag issues and the feeling of non-responsiveness (Jess, 2019). Since then, the networking issues that plagued Stadia have been fixed and the delay has been reduced, but the visual quality of the final displayed images may be reduced instead when network issues are encountered (Andronico, 2021).

As for our distributed rendering approach, the client produces a rasterised image immediately while waiting for the server to perform its ray-tracing pass. We avoid the problem of non-responsiveness with our approach, as we are able to display the rasterised image promptly even if we have not received the latest data from the server yet. The quality of the image may be affected slightly due to the use of or the extrapolation of old data, but interactivity of the application will not be affected. Furthermore, our rendering pipeline is always able to provide images at full resolution, whereas the majority of cloud gaming services produce a lower resolution image adaptively in response to a drop in network throughput (Di Domenico et al., 2021).

2.2 Hybrid Rendering Algorithms

Hybrid rendering refers to the use of both ray tracing and rasterisation in the graphics pipeline. Our current approach to hybrid rendering is straightforward, we perform deferred shading with rasterisation on the client while the server performs ray tracing to determine the visibility of each fragment of the scene. The server sends the visibility data to the client, which performs shading and illumination of the scene to produce the final image in a second rasterisation pass. There are many other hybrid rendering approaches that we may consider, however. These approaches are helpful in various ways. They may teach us ways to optimise our own algorithms, they may allow us to add more effects to the image for more physical accuracy, or they may serve as an alternative which we can compare our algorithm to.

The first hybrid rendering approach is one described by Parhizgar and Svensson (2021), where refractions are performed using a combination of both rasterisation and ray-tracing pipelines. For the post-rays approach, rasterisation is first used to produce the initial refracted rays based on Wyman's approach for image-space refractions (Wyman, 2005), and ray tracing is used afterwards to follow these initial rays. The pre-rays approach is similar, but ray tracing will be done for the initial refraction, and rays that terminate too early when the fixed recursion depth limit is reached will be used to sample a texture that is produced by rasterisation using the image-space refractions technique. The post-rays approach is slightly faster as it always saves on rays traced, but less accurate. Overall, there is a reduction in rendering speed as compared to the purely ray-traced alternative as there is a reduction in the number of rays shot for both methods.

Currently, our implementation only produces shadows, so this hybrid rendering approach would be useful for adding refraction effects to our pipeline. Both the post-rays method and the pre-rays method are potentially useful, however we would need to modify their approach to account for networking between the client and the server in our pipeline. To apply the post-rays approach, we would need to send vector data from the client to the server, and the server will need to reply with final image data, however this will only have to be done for the fragments in the scene that contain the colour of a refracted object. As for the pre-rays approach, we would need to send vector data of the final rays from the server to the client, and the client would use these rays to access the rasterised refraction textures. The pre-rays approach may also produce the final colour of the pixel if the ray tracing terminates before the recursion depth is reached, which we can send to the client to output directly to the screen.

Secondly, we have a different approach by Jhang and Chang (2021), who describe an algorithm where the rasterisation pass sets up the information required for ray tracing instead. Their approach is inspired by voxel cone tracing where the scene is initially split up into voxels in a process known as voxelisation. The final output images are produced by the ray tracer using the path-tracing algorithm. First, rasterisation is used to calculate the incoming light directions for each voxel in the scene. This information is passed on to the ray tracer, which will shoot out rays to perform path tracing, and use the voxel data to guide the rays. This algorithm results in less noisy path-traced images at the cost of a slight drop in frames per second.

To give some context, path tracing has conventionally been an algorithm that either produces grainy images due to the random nature of the tracing of rays, or produces images at a high time-cost due to the need to average the samples of a large set of rays to make up for the inherent randomness. However, path tracing is able to give us global illumination, adding the full set of real-world lighting effects such as reflection, refraction, and shadows within a single framework. Although Jhang and Chang's method produces more accurate path-traced images by eliminating noise, it is an inversion of our current pipeline as the images would be produced at the end of the ray-tracing portion of the pipeline rather than at the end of the rasterisation portion.

Also, the voxel cone-tracing approach used incurs additional cost for voxelisation, requiring a large memory usage for the voxel information (Crassin et al., 2011). If it is implemented as a distributed algorithm for our use case, there would be a high load on the network because the voxel information will have to be sent to the server, and the server would have to stream image data back to the client for display. Granted, the voxel information needs only to be updated for the same scene if there are moving objects or the camera moves, so the full voxel octree does not have to be sent for every frame.

2.3 Further Ideas

A new technology that has debuted on NVIDIA GPUs is deep learning super sampling (DLSS), which makes use of artificial intelligence to enhance an input image (Edelsten et al., 2019). This method leverages machine learning techniques to up-sample an image, increasing an image's resolution. However, the computation for up-sampling is non-trivial, and it has to be performed on the ray-tracing GPUs themselves. Furthermore, the program has to be trained with a sample set of images that will be specific to the application. There are also possible failure cases where the algorithm fails to produce a correct image as a result of a lack of information from the original input.

NVIDIA has kept implementation details about the DLSS algorithm under wraps as a trade secret, so it would be difficult to use it as a comparison with our rendering algorithm. Nevertheless, Facebook has debuted their own method of super-sampling an image which claims to be comparable to NVIDIA's in image quality (Xiao et al., 2020), and details are publicly available. This area of research might be useful to our project as it is also a hybrid approach where an image is rendered with traditional techniques, and then more processing is done on the image to improve its quality.

The last alternative technique that we may consider is to perform ray tracing on the mobile device itself. Although this is not possible now, hardware advancements may make real-time ray tracing possible on chips designed for battery-powered devices. Advanced Micro Devices Inc (AMD) has hinted that they are working on ray-tracing capabilities for Samsung's Exynos chip, which is used for Samsung Galaxy smartphones, although no specifics have been revealed (Duckett, 2021). Even if such a feat was achieved, the final processor may be more costly or more power-intensive than the current ones, so our distributed rendering approach may still be relevant.

3. Implementation

3.1 Change of Network Protocols

Initially, the hybrid pipeline made use of TCP to send and receive data. To change the pipeline to use UDP and ensure that the rendering still functions well, I had to account for packet losses and out-of-order receiving. In the end, I decided to implement a custom protocol for the UDP transmission. The following diagram displays the fields of the header for this custom protocol.

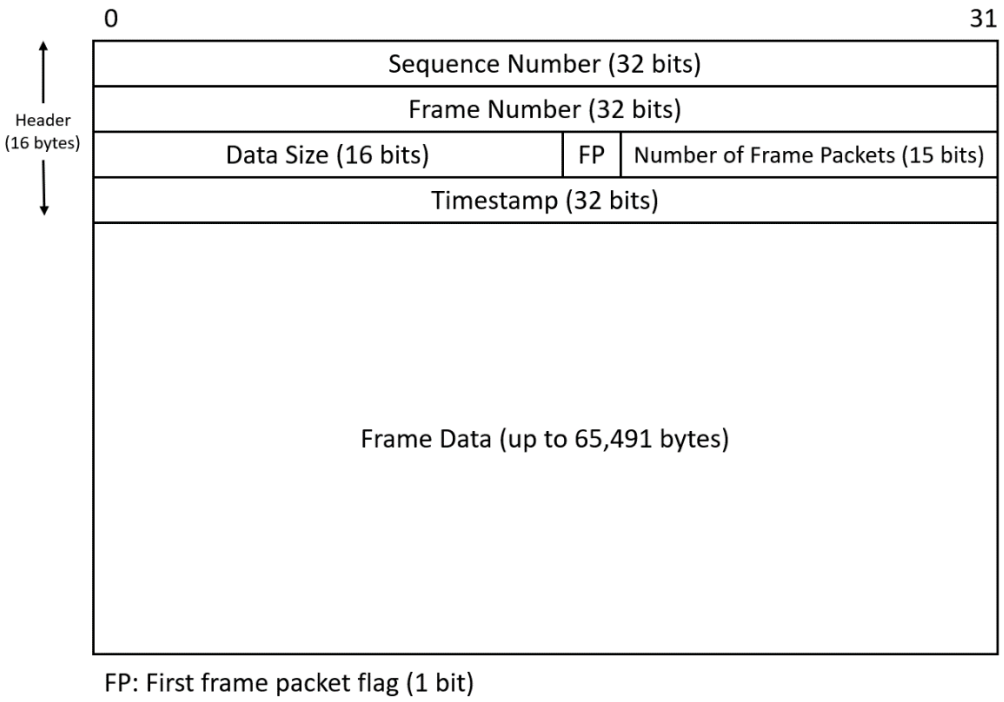


Figure 1: Structure of a UDP Custom Protocol packet.

There are 6 header fields, 3 of which consist of 4-byte integers, 1 of which is a 2-byte integer, 1 of which is a 15-bit integer, and 1 which is a 1-bit flag. The first integer, the sequence number, is used to identify whether the packet arrived out-of-order or not. The sequence number is always increasing and starts at 0 for the first packet. The third integer, the data size, stores the number of bytes in the frame data section of the packet, and it is used to delineate the packets so the program can tell where the next packet begins. Due to the limitations of UDP, one UDP packet may only contain 65,507 bytes, so the packet size field only requires 16 bits.

The UDP Custom Protocol is used for the transmission of the camera data from the client to the server, as well as the transmission of the visibility buffer from the server to the client. The second, fourth and fifth integer contain information that are only used for the visibility buffer sending, and their bits will be filled with a placeholder value if they are not required. More specifically, the frame number and the timestamp will be filled with 1s, and the number of frame packets will be filled with 0s. This is because 0 may be a valid value for the frame number and the timestamp, however 0 number of frame packets would not be a sensible value.

Based on a fixed window size of 1920 x 1080, the uncompressed visibility buffer has a size of $1920 \times 1080 \times 4 = 8{,}294{,}400$ bytes. 4 bytes or 32 bits are allocated for each pixel to allow for potentially 32 lights in the scene. Note that the actual screen size is slightly smaller than 1920 x 1080 due to the presence of Windows interface elements such as the taskbar. Hence, without compression, we can expect the visibility buffer of each frame to be sent in $8{,}294{,}400 \div 65{,}487 = 127$ UDP Custom Protocol packets. Also, note that each UDP Custom Protocol packet will be further split into smaller Internet Protocol (IP) packets based on the maximum transmission unit (MTU) of the network.

The fields of frame number, number of frame packets, and timestamp will contain the same values for all of the packets that hold a different piece of the data for the same frame. The idea is so that even if only one packet of that frame makes it to the destination, we still know about the metadata for the entire frame.

The frame number is a unique numerical label for each frame that is rendered by the server and the client, and similar to the sequence numbers, we start at 0 for the first frame, and we add 1 to the frame number each time to get the frame number of every subsequent frame.

To explain the meaning of the other two integer fields and the one-bit flag, we will take a hypothetical frame with frame number *i*. For a packet that is produced for frame number *i*, the field for the number of frame packets will encode the total number of UDP Custom Protocol packets that was sent by the server to encode the information for frame *i*. Due to compression, the number of packets for each frame may vary. Hence, we will need to know the number of frame packets to reconstruct the original data for the frame *i*.

If the packet produced is the first packet for frame *i*, which is also to say that the packet has the lowest sequence number out of all the packets for frame *i*, the first frame packet (FP) flag will be set to 1. Otherwise, the FP flag will be set to 0. This is used by the client to identify cases of packet loss, of which the exact mechanism will be described in the following section, Section 3.2.

Lastly, the timestamp field allows the program and the user to know about the exact amount of network latency between the client and the server. The timestamp field encodes the number of milliseconds elapsed since the server has rendered the visibility buffer of the first frame, at the time when frame *i* is rendered.

3.2 Handling of Network Issues

Some problems that we need to handle with UDP are the possible loss of packets, the reordering of packets, and corrupt packets. The program takes a greedy approach to this, assuming that because frames are being rendered so fast, there is a very high chance that it would receive all of the packets for a certain frame in order. Hence, at the sign of any error with the individual packets, the entire frame is discarded, and the program tries to receive the next possible frame. Figure 2 contains a flowchart explaining the receiving algorithm, which is designed to maximise the chances of receiving an entire frame correctly with minimal copying of data.

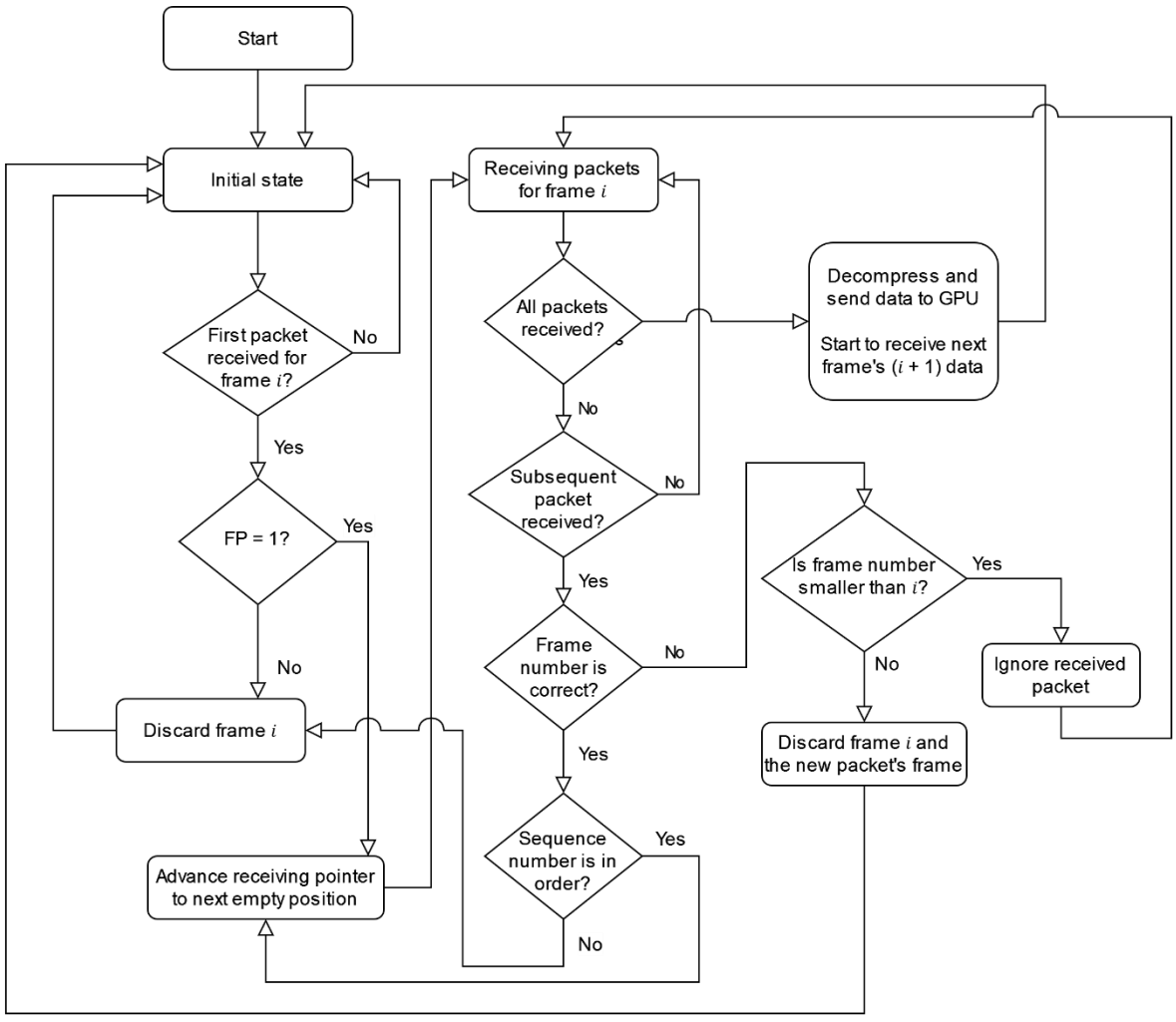


Figure 2: Flowchart of the receiving algorithm on the client.

At the start, the receiving thread is in the initial state. The receiving thread tries to receive a packet with the FP bit set to 1, indicating that it is the first packet for a certain frame

i. It will ignore all other packets with FP bit set to 0. Once it has found the first packet, it retrieves the frame information from it, including the frame number *i*.

For subsequent packets, the receiving thread relies on the sequence numbers of the packets to ensure that the packets are received in consecutive order. If a packet is received for the same frame *i* and it is not in order, the entire frame is discarded, and the receiving thread returns to the initial state. If a packet is received for an earlier frame *h* where $h < i$, it can be safely ignored. Otherwise, if a packet is received for a later frame *j* where $j > i$, the receiving thread returns to the initial state and waits for packets from the frame $j + 1$. Below, figure 3 illustrates what happens in the receiving buffer for the various cases.

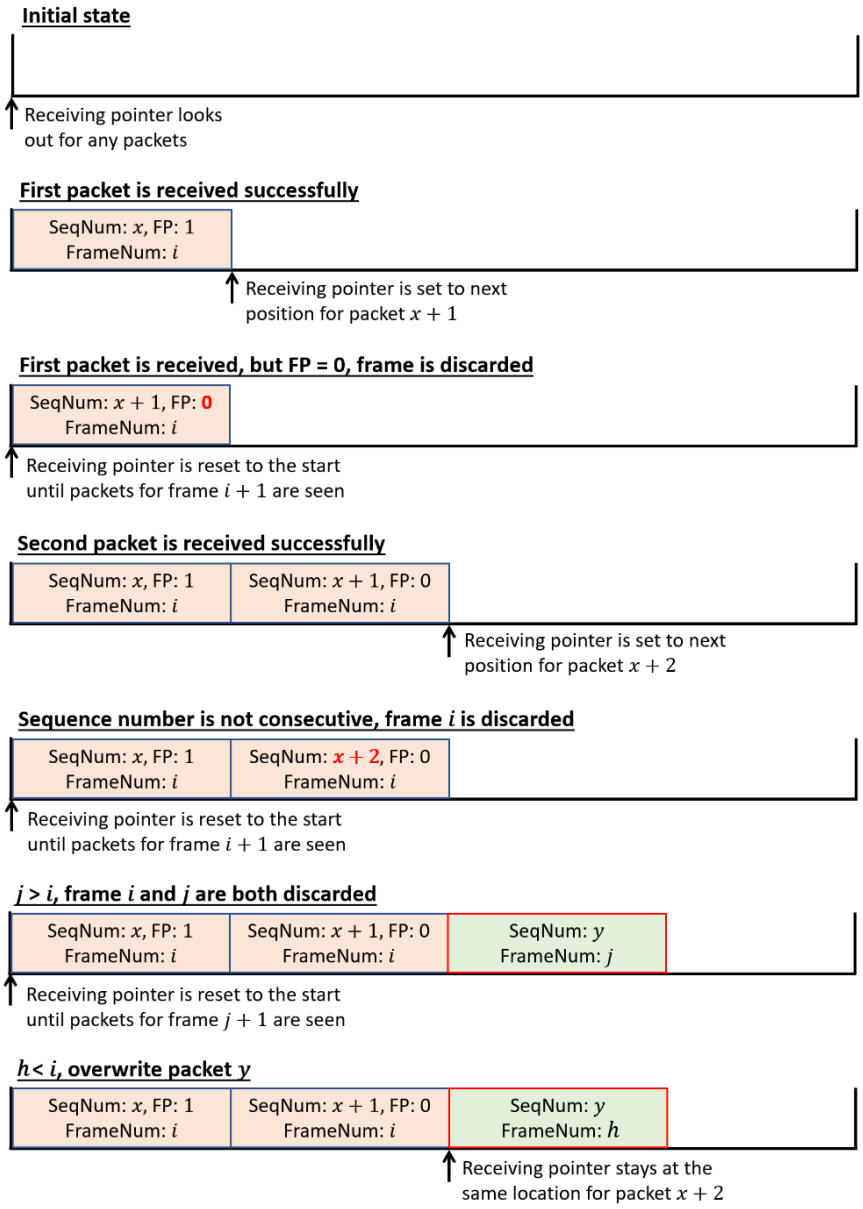


Figure 3: Diagram illustrating the various states of the receiving buffer.

To reduce the copying required, the packets are received into the same buffer that will be used for decompression. There is still a little copying required to remove the packet headers, which is shown in Figure 4. The assumption made is that the packet headers are likely to be smaller than the packet data, which is almost always the case for the packets sent by our program.

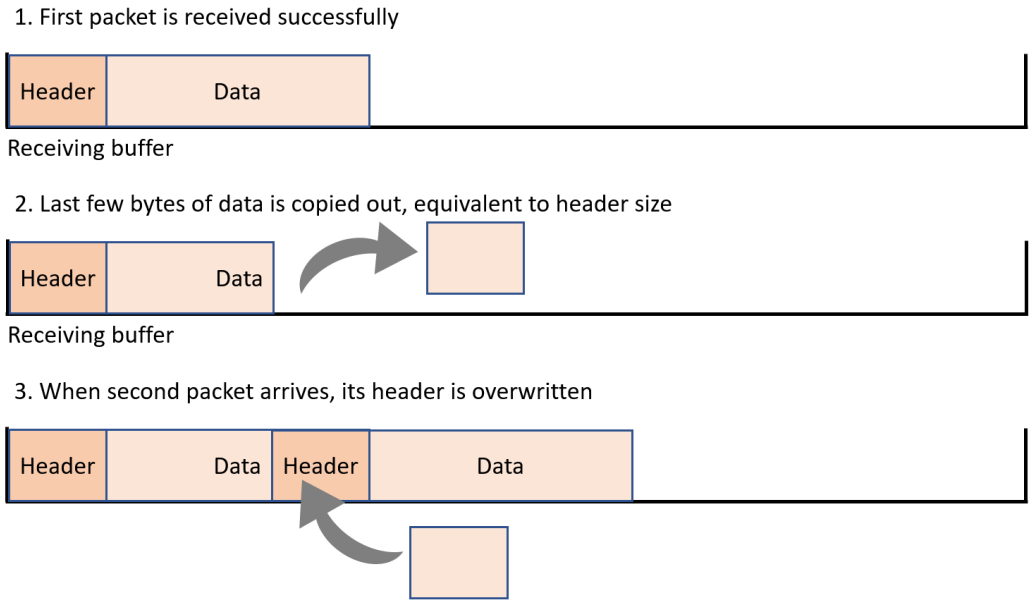


Figure 4: Illustration of the copying required in the receiving buffer to remove packet headers.

Previously, the program attempted to detect for and fix out-of-order packets. However, I found that this was not possible without frequent copying of the data, because the data needs to be put into a contiguous buffer for decompression and eventual transfer to the GPU. In the previous implementation, when a newer-than-expected packet was received, it would always be copied to a hash table so the program could continue to listen for the expected packet. Then, when the expected packet number increased eventually, whether due to the expected packet being received or due to a socket timeout, the correct packet will be copied out of the hash table and placed in the correct location.

Figure 5: A stuck image where the shadow is incorrect for a number of frames, from a previous version of the rendering pipeline.

Since the visibility data is quite large in size, whenever there was a skip in the packet sequence numbers, the copying of packets slowed down the receiving thread and caused even more copying because the receiving thread could not receive the older packets in time. This resulted in a positive feedback loop, which manifested on the screen as a bug where the shadows would get stuck for a couple of frames, shown in Figure 5. This also explained why adding a delay to the server or running the program in Debug mode solved the issue, as in those situations, the server would send at a slower pace and allow the client to catch up with it if packets were lost.

Hence, the current approach was implemented which avoids all copying of the visibility buffer data to improve the receiving speed. The only copying takes place after the data is fully received, to decompress the data and to subsequently transfer it to the GPU for use in rendering. Note that for the UDP Custom Protocol, no acknowledgement packets or acknowledgement numbers are required as we do not do any retransmission.

3.3 Parallelisation of Client and Server Passes

After implementation of UDP and compression, we had achieved frame rates of more than 30, which is the minimum required for a feeling of interactivity in users. However, it was still much lower than what we had expected or desired. We had expected a higher frame rate similar to the case where frames are fully rendered on a single machine, which produces more than 120 frames per second with our current hardware.

Hence, we started looking into the ordering of the rendering passes. I identified a bottleneck with the process of sending and receiving of the visibility data, as I realised that in the original pipeline, the client would wait for the visibility buffer to be fully received before beginning the rasterisation pass to display the image on the screen, while the server would wait for the visibility buffer to be fully sent before beginning to perform ray tracing. However, we could potentially overlap the network operations with the rendering operations as they are performed on different devices (CPU for networking, GPU for rendering), resulting in more frames per second produced.

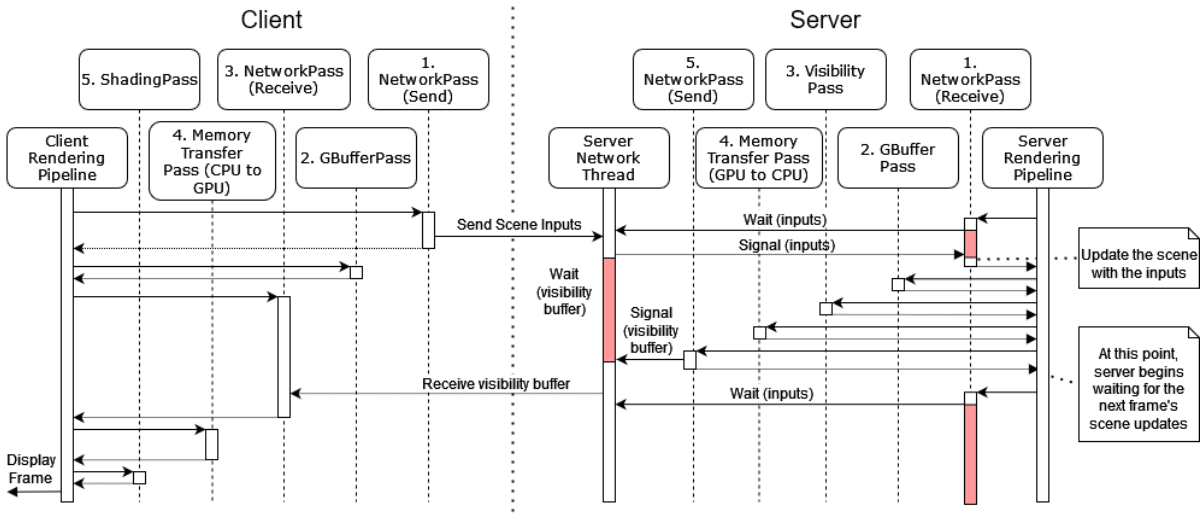


Figure 6: Original ordering of the distributed pipeline.

Above, Figure 6 shows the original sequence of the pipeline operations, with waiting threads highlighted in red. Although there are multiple threads in the server, the existence of two different conditional variables causes the server to effectively run in a sequential manner. One variable is set up to ensure that the Server Rendering Pipeline (SRP) waits for the Server Network Thread (SNT) to receive the camera data from the client in NetworkPass (Receive) before starting on rendering. Meanwhile, the other conditional variable ensures that before

sending the visibility buffer to the client, the SNT needs to wait for a signal from NetworkPass (Send), a pass that is only executed after the SRP completes its rendering tasks. Immediately after signalling, the SRP goes to NetworkPass (Receive) to wait for the SNT again while SNT is still sending. Also, the client has no parallelisation at all.

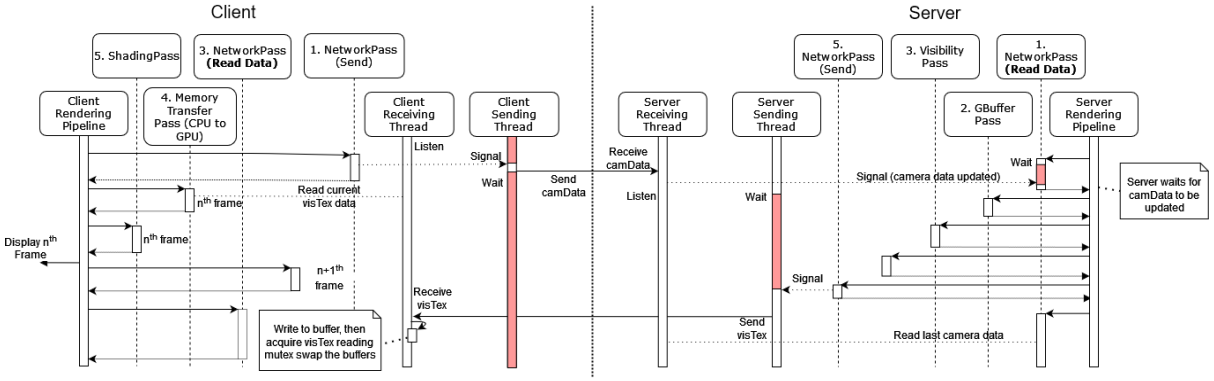


Figure 7: New ordering of the distributed pipeline.

Hence, I fully parallelised the client by adding a sending and receiving thread. I also added a separate receiving thread for the server and eliminated the extra conditional variable. For the client, to support the parallel threads, we had to create another buffer for the visibility data, such that the rasterisation pipeline could read from the buffer and draw to the screen while the receiving thread is writing to the other buffer. Once the receiving thread has fully received the visibility data, the receiving thread will acquire a mutex for the reading buffer, before swapping the buffers around. This allows the client to perform rendering uninhibited by the prevailing network conditions, but the client may not be always using the latest visibility data.

The reason why sending and receiving were split into different threads was to allow sending and receiving to occur in an arbitrary order. Previously, when sending and receiving was done in the same thread in the server, the server was always forced to listen for camera data before sending its visibility buffer. Then, the camera data would be lost if it arrives while the server is still sending, which is a situation that occurs more frequently after the pipeline was changed to use UDP instead of TCP.

As of now, the client and server of our implementation run at similar rates because the server waits for the client's response every time before starting the rendering. If we decouple the server and the client however, the server is able to produce much more frames per second as compared to the client. The server may be running faster because it does not draw anything to the screen, but the client does so. It may then be possible to provide the server with additional

rendering work to reduce the workload of the client, resulting in an overall increase in frames per second shown on the client's screen, or to run the server with multiple clients without impacting performance.

3.4 Prediction of Received Data

Figure 8: An image produced by the old rendering pipeline. The difference in the shadows and the scene geometry is visible as a white line in the right corner of the room.

With a parallel pipeline, I found that there was a small lag between the shadows and the rest of the scene when the camera was moving. In Figure 8, we can see evidence of the lag between the ray-traced shadows and the rest of the scene in the right corner of the room where the walls meet. In this screenshot, the camera is moving towards the back wall, so the back wall is becoming larger. There are 1-2 pixels which do not appear to be in shadow on the edge of the wall, which is because the server has not responded to the changes in the position of the camera yet, so the shadow for the back wall is still the smaller one for the previous frame. These incorrect pixels quickly disappear when the camera stops.

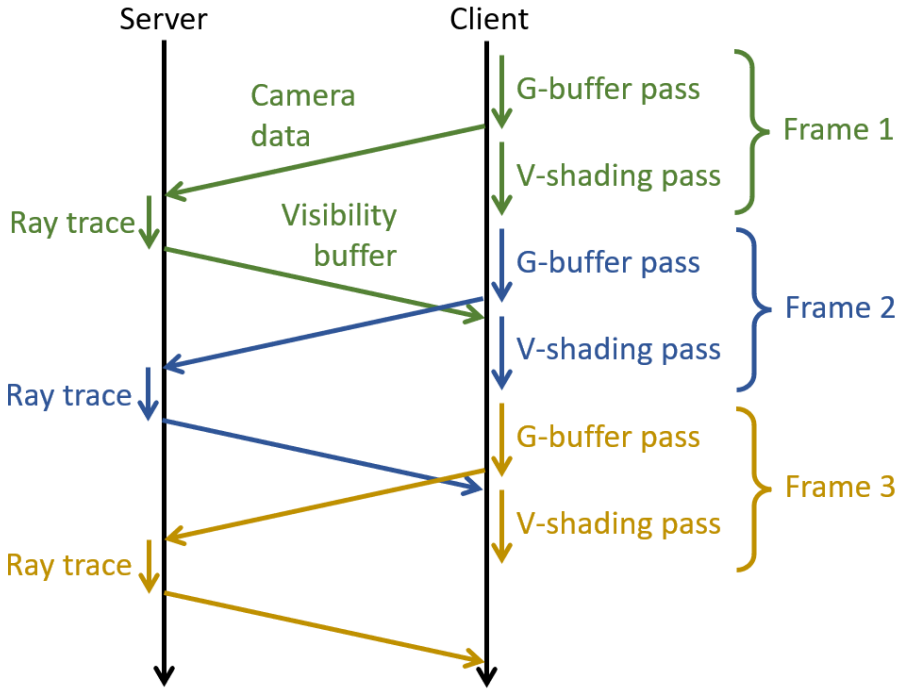


Figure 9: Client-server diagram illustrating the effects of network latency on the rendering pipeline. Frame 2 will only have the outdated visibility data meant for frame 1.

This lag can be attributed to network latency, as shown in Figure 9. After parallelisation, the client no longer waits for the server to reply with the correct visibility buffer before proceeding with rendering, which increases the frame rate but results in inaccurate images.

To compensate for the incorrect shadow, we have implemented two strategies in the rendering pipeline. One of them is switching between a sequential rendering and a concurrent rendering, and the other is extrapolating the received data to predict the data for the current frame. I worked on the latter, and it has been implemented in the form of an additional prediction pass in the client rendering pipeline that runs immediately after the received visibility buffer data from the server is transferred to the GPU.

To perform prediction, the client stores the view transformation matrix of the camera in a circular buffer for every frame that it renders. The view transformation matrix contains information about the camera's position and rotation in world space, and this information is stored in a format that is easy to use in a shader. For moving scenes, this circular buffer may be expanded to hold the information of moving objects as well, so that it contains all of the changes in the scene. A small modification is made to the pipeline such that when the client

sends its camera data to the server, it also sends the frame number of the current frame that the client is rendering in the ongoing cycle of the render pipeline. This number is placed in the frame number field of the UDP Custom Protocol, which was previously unused when camera data was being sent. When the server replies to the client with the ray-traced visibility buffer, the server will send this frame number back to the client, corresponding to the camera data that the visibility buffer was rendered with. The server may take a while to reply though, so the client will not wait and proceed with rendering.

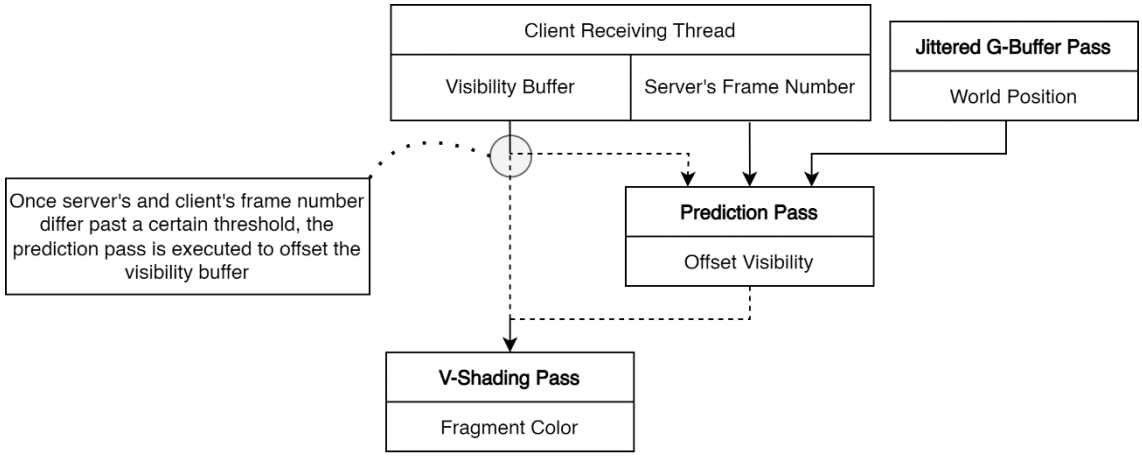


Figure 10: Data flow diagram for colour computation at the client. The prediction pass is not applied at negligible levels of latency.

Back at the client, the visibility buffer is received from the server along with the corresponding frame number, and the client passes this number as an input to the prediction pass. The prediction pass is only applied when the difference between the client's and the server's frame number exceeds a certain threshold, as seen in Figure 10.

In the latest implementation of the distributed rendering pipeline, I have set the threshold to 3 frames of difference or more. Then, the prediction pass only runs when the difference is 3 or greater, which translates to 2.5 ms of delay at 120 FPS. This value was obtained after testing the pipeline with different threshold values for prediction, and I found that at 1 or 2 frames of difference, the sequential rendering was able to correct the error in the image with minimal disruption to the interactivity of the program. At 3 frames or more, correcting the error by sequential rendering led to a small but noticeable lag in the frames produced, while the visibility data obtained by prediction still appeared quite accurate. As network latency increases, the prediction becomes less accurate as the client does not have enough data to shade the entire scene.

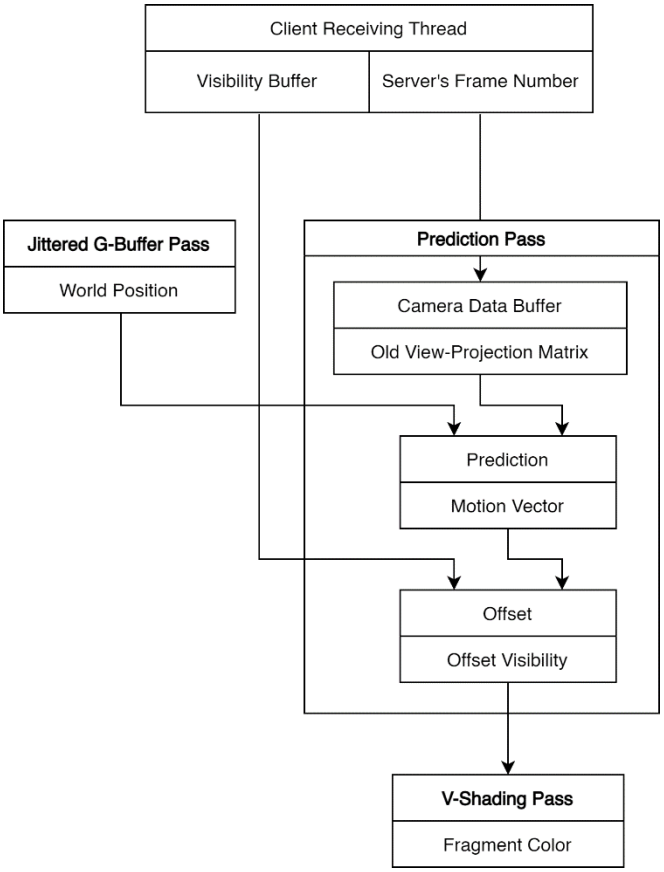


Figure 11: Internal structure of prediction pass, showing the required inputs and produced outputs.

Figure 11 shows the individual components within the prediction pass, as well as the inputs and outputs for each component. The prediction pass uses the received frame number to retrieve the older camera data corresponding to the received visibility buffer. Together with the newer camera data of the current frame being rendered, the prediction pass calculates a 2-dimensional motion vector for every fragment of the image. These motion vectors store the difference between the screen-space position of the same world-space points as seen by the old camera and the new camera. Finally, each fragment's visibility information in the received visibility buffer will be offset by the individual motion vectors, forming a new visibility buffer which we call the offset visibility buffer. The offset visibility buffer will be used for the shading of the scene instead.

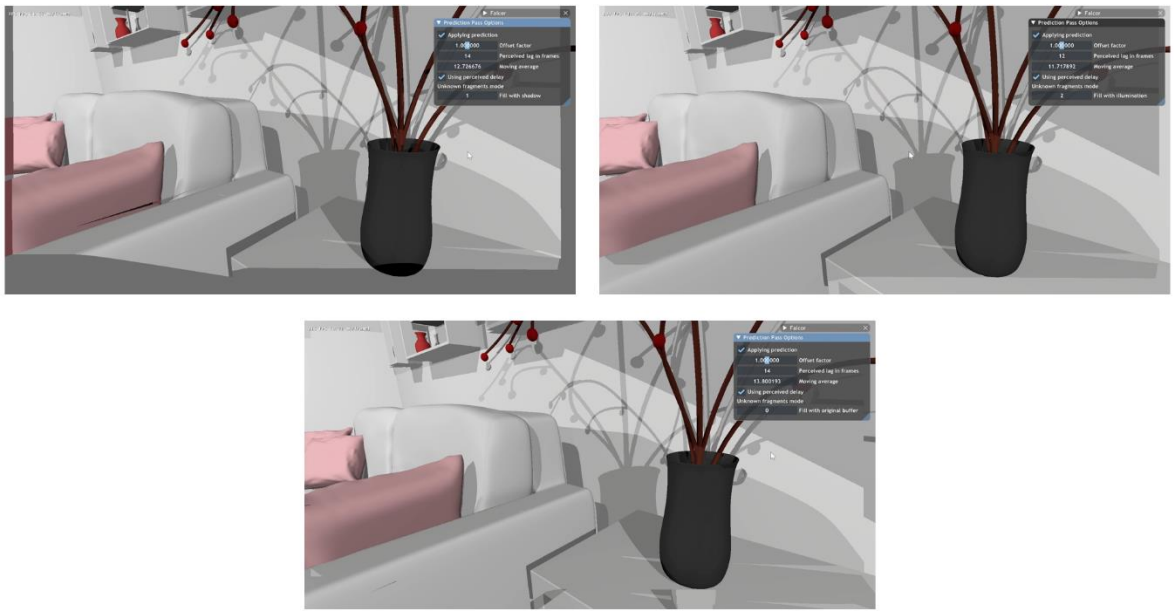


Figure 12: Comparison of different modes of handling of unknown fragments. Top-left: In complete shadow. Top-right: In complete illumination. Bottom: Copy from received visibility buffer.

Since the old visibility data is rendered by the server for a different camera orientation as compared to the current camera orientation at the client, the client will not know about the visibility data for certain fragments that are revealed from camera movement. Such fragments can be identified based on their motion vector values. Whenever the motion vector for a certain fragment points to a location that is outside of the screen, it is an unknown fragment because the visibility data for that fragment does not exist, and the program can be configured to fill in the data for that fragment. I tried out three different modes for specifying the visibility data for these unknown fragments, namely setting their visibility data to be fully in shadow or fully in illumination, or copying their visibility data from the original received visibility buffer. Images of these modes are shown in Figure 12 for comparison.

For our current test scene, I found that having the unknown fragments fully illuminated gives rise to the best image, and it has been set as the default mode. The mode of direct copying from the received buffer causes obvious inaccuracies in the unknown fragments, which are especially noticeable when the camera is moving. As for the mode of being in complete shadow, the dark fragments contrast starkly with the other parts of the image that are mostly lit, and the fragments being in complete shadow is generally incorrect when compared to the ground truth produced by ray tracing.

3.5 Other Additions

This section mentions the modifications to the pipeline which were done by a teammate Nicholas. Although I did not contribute directly to the implementation of these features, I was involved in discussions about them, and I assisted in resolving conflicts during merging of the code branches. The features are briefly described here before the Analysis section, Section 4, so that the reader may understand all of the factors that contributed to the subsequent findings of the research.

As mentioned in the earlier segment, each frame produces a total of 8,294,400 bytes at the server, which is a huge amount of information to send over the network. Through timing the various passes and network operations, we found that the sending and receiving of the visibility buffer of the frame was taking up the most amount of time in the pipeline, so we sought to reduce the data that we had to send. To draw a comparison, the camera data of the frame which is sent from the client to the server only consists of 3 vectors of 3 floats each, which consists of merely $3 \times 3 \times 4 = 36$ bytes.

Hence, various compression methods were experimented with, and the compression of the entire visibility buffer data for a single frame with the LZ4 algorithm was eventually implemented. Some redundant copying of data in the pipeline was also eliminated, namely in the CPU to GPU data transfer phase. We found that the version with compression was much faster than the version without. LZ4 compression was able to reduce the size of the visibility buffer immensely, specifically by 97% on average. Previously, we had required 127 UDP Custom Protocol packets to send the data, but after compression we only needed 6.

Apart from client-side prediction, another strategy of switching between sequential and concurrent modes was implemented to reduce the perceivable error between the shadows and the geometry of the scene. Sequential refers to the use of older geometry-buffer data that matches the currently received data from the server, such that the frames rendered is equivalent to one produced by the non-distributed pipeline, but the frames will also be older than what is supposed to appear on the screen. Concurrent refers to the use of the latest visibility buffer from the server and the latest geometry buffer from the client, leading to a rendered image that may look incorrect, but it would be more responsive to changes in input.

When the network delay is shorter than the time it takes to render one frame on the client, or when the camera movement is large and abrupt, the sequential mode is used. Otherwise, the concurrent mode is used with the prediction of received data that I implemented. This switching strategy was implemented together with a refactor of the code, to allow for easier reordering of the various passes in the pipeline, as well as to separate the client and server network code which was originally located in the same files.

4. Analysis

4.1 Framerate

all values measured in Debug Mode of Visual Studio unless specified	Frames Per Second (frame/s)	Time Per Frame (ms/frame)
Original TCP	7.7	130
UDP	6.9	144
UDP and LZ4 Compression	35.5	28.2
Parallel	87.1	11.5
Parallel with Bugs Fixed and Prediction *(Release Mode)*	114.6	8.7

Table 1: Comparison table for the number of frames produced.

The implemented changes have resulted in an increased frame rate as compared to the initial version. Frame rate is defined as the number of frames per second that are rendered and displayed on the screen by the distributed rendering pipeline. The frames per second value and time per frame value were measured at the client side, and the values are taken directly from the reading shown in the user interface overlay by NVIDIA's Falcor framework. The values were taken after waiting around 30 seconds after the program launches, allowing for the initialisation phase to be completed and for the values to stabilise.

Note that these FPS values are constantly changing for every frame rendered, so it is difficult to get a value of high precision. The actual time per frame value that is displayed tends to vary from the table values by ± 3 ms. Also, a different frame-rate value will be obtained for different positions of the camera, especially when the camera is moved far away from the geometry of the scene, and the camera mostly renders the environment map. This phenomenon is caused by a reduced data size of the visibility buffer generated by the server, leading to faster transmission of the data. Hence, the camera was set to the same position when the readings were taken, where the scene would occupy the entire screen.

Most of the values were taken when the program was compiled with Debug Mode, because the Release Mode suffered from bugs like the stuck frames issue mentioned in Section 3.2. Now, I have fixed the bugs, and we can observe that compiling in Release Mode causes an additional speed-up in the frame rate. This is likely due to certain parts of the code not being optimised out when compiled in Debug Mode, for the purposes of adding breakpoints and tracing the code.

Although the LZ4 compression reduces the size of the visibility buffer by more than 20 times, when it was first implemented, the pipeline only improved in speed by 5 times. The exact reason why this happened is possibly due to the bugs that were still present in the system at that time. When the parallel pipeline was implemented, some inefficiency in the network passes was eliminated as well. Interestingly, we found that on the parallel pipeline, the sending and receiving rate of the frames were identical to the actual frame rate displayed on the screen at the client, despite the fact that the network threads were allowed to run independently of the main rendering thread. Specifically, the frame rate of the parallel pipeline was measured to be 87.1 FPS as shown in the table. Meanwhile, the server's sending rate of the visibility buffer was measured to be 87.39 FPS by averaging over 10 randomly selected frames, while the client's receiving rate for the visibility buffer was 90.63 FPS, averaging over the same 10 frames.

The current frame rate of around 115 FPS is adequate for video games, which typically have a frame rate of 60 FPS on mobile phones or consoles. It is also adequate for virtual reality headsets, which have a slightly higher frame rate of 90 FPS. This higher frame rate reportedly prevents motion sickness in the users of virtual reality headsets, although there does not seem to be much quality evidence backing up this claim. Of course, frame rates that are greater than 90 FPS may be desirable for our pipeline as well, as according to an internal study by NVIDIA, gamers perform better in competitive first-person shooters running at higher frame rates (Gerardo, 2019).

It is interesting to note that the initial UDP implementation actually achieved a lower frame rate than the corresponding TCP one. This could be explained by three reasons. Firstly, some of the inbuilt capabilities of TCP such as checking whether the packets arrived in order are done at the operating system level or at the hardware level. When UDP is used, we need to implement the same checking ourselves, and it ends up being slower because it is executed at the application level.

Next, the programs are being tested in the laboratory environment over a local network, and the computers are physically very close to each other. The chance of packet loss in this environment is quite low, and we have observed that packet loss, packet reordering, and packet corruption almost never happen, even when running the application for extended periods of time of 10 minutes or more. In this scenario, sending packets through TCP takes a similar amount of time as sending packets through UDP, because no retransmissions would have to be done.

Finally, the initial implementation that made use of UDP was not fully optimised, and there was a lot of copying to reorder packets even when it was not needed. I have fixed this redundant copying in the latest version, as explained earlier in Section 3.2.

4.2 Image Quality

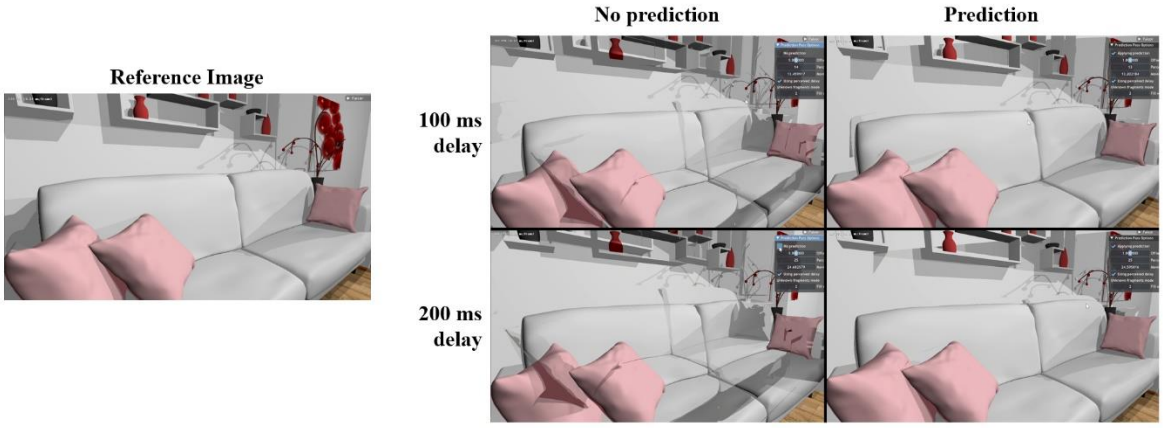


Figure 13: An example of the images produced by the prediction pass when the camera is moving. Far-left: Reference image at 0 ms delay.
Top-left: No prediction with 100 ms delay. Top-right: Prediction with 100 ms delay.
Bottom-left: No prediction with 200 ms delay. Bottom-right: Prediction with 200 ms delay.

In Section 3.4, the prediction pass in the client was mentioned as a way to update the received visibility buffer to fit the currently rendered scene. Figure 13 shows the results of the prediction at different levels of simulated network latency using Clumsy. It can be observed that as the network latency increases, the prediction becomes more and more inaccurate. Hence, we intend for our prediction method to only be used when the network latency is low. It is not meant to be a full replacement of the server-side rendering if network latency is very high, or if the connection to the server is lost completely. The prediction pass is currently limited to only predicting 50 frames, after which it will signal to the user that their connection is too poor to support distributed rendering.

Figure 14: Prediction does not work well for objects that are close to the camera, such as the objects on the side table. In the background, the shadows for the sofa and the cushion are more accurate.

Also, we find that the prediction results in more accurate shadows for large objects that are far away from the camera, but for small objects close to the camera, the inaccuracy in prediction is highly visible. An example of this is shown in Figure 14. This is likely because objects close to the camera would move a greater distance across the screen when the camera moves as compared to objects that are further away, hence the older visibility data would offer a worse prediction for them.

Definitely, more effects are warranted to support the full range of graphics that are used in video games today. Nevertheless, our current implementation acts as a strong proof of concept that distributed rendering is a feasible mode of rendering for thin-client devices. In the future, depth-of-field effects, motion blur effects, reflection effects, and more may be added to the pipeline using novel algorithms.

4.3 Measurements with Visual Metrics

To objectively compare our prediction method with the expected ground truth, we rely on various video and image quality metrics. This section deals with our obtained measurements with several visual metrics and explains how the results may be interpreted.

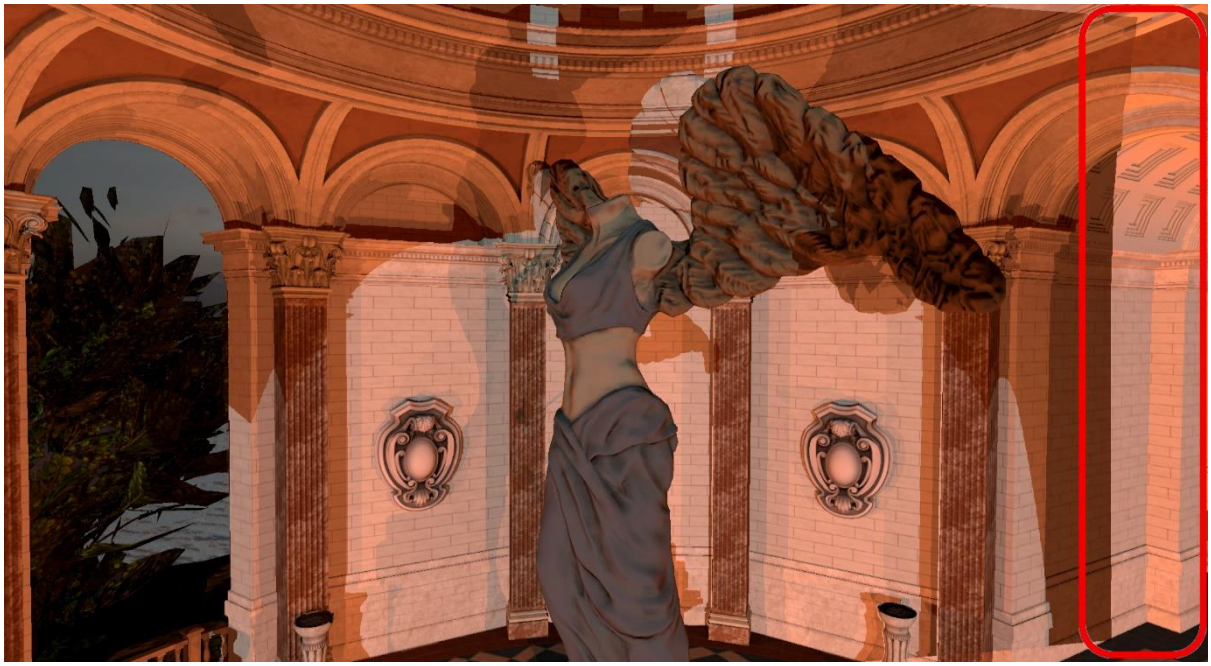


Figure 15: Prediction cannot occur for the revealed scene geometry circled in red at the right side of the image, so the pipeline assumes that it is fully illuminated.

As the camera moves, it reveals more parts of the scene that previously could not be seen. Since the prediction extrapolates from the visibility data for an older frame, we find that our rendering pipeline produces inaccurate shading for the revealed geometry in the newer frame at the side of the image that is opposite to the camera movement direction. As mentioned in Section 3.4, the fragments that contain the revealed geometry are taken to be fully illuminated. Figure 15 shows an example of this.

We hypothesised that rendering a larger visibility buffer at the server would help to resolve this issue, as there would be a higher chance that the client will receive visibility data for the revealed geometry at the sides of the image.

In Figure 16, we measure the error of the visibility buffer created at different levels of prediction for different sizes of visibility buffer. We use the expected visibility buffer computed by the server pipeline running at 0 frames of delay from the client as the ground truth. The

default visibility buffer is the exact same size as the image at 1920 x 1080 pixels, and the rest of the buffers shown in the figure are larger than the default. We take 64 * 36 to refer to an additional 64 pixels for the width of the visibility buffer, and also an additional 36 pixels for the height of the visibility buffer, for a total buffer size of 1984 x 1116 pixels. Similarly, the 128 * 72 pixel buffer refers to a buffer size of 2048 x 1152, and 640 * 360 pixels refers to a buffer size of 2560 x 1440.

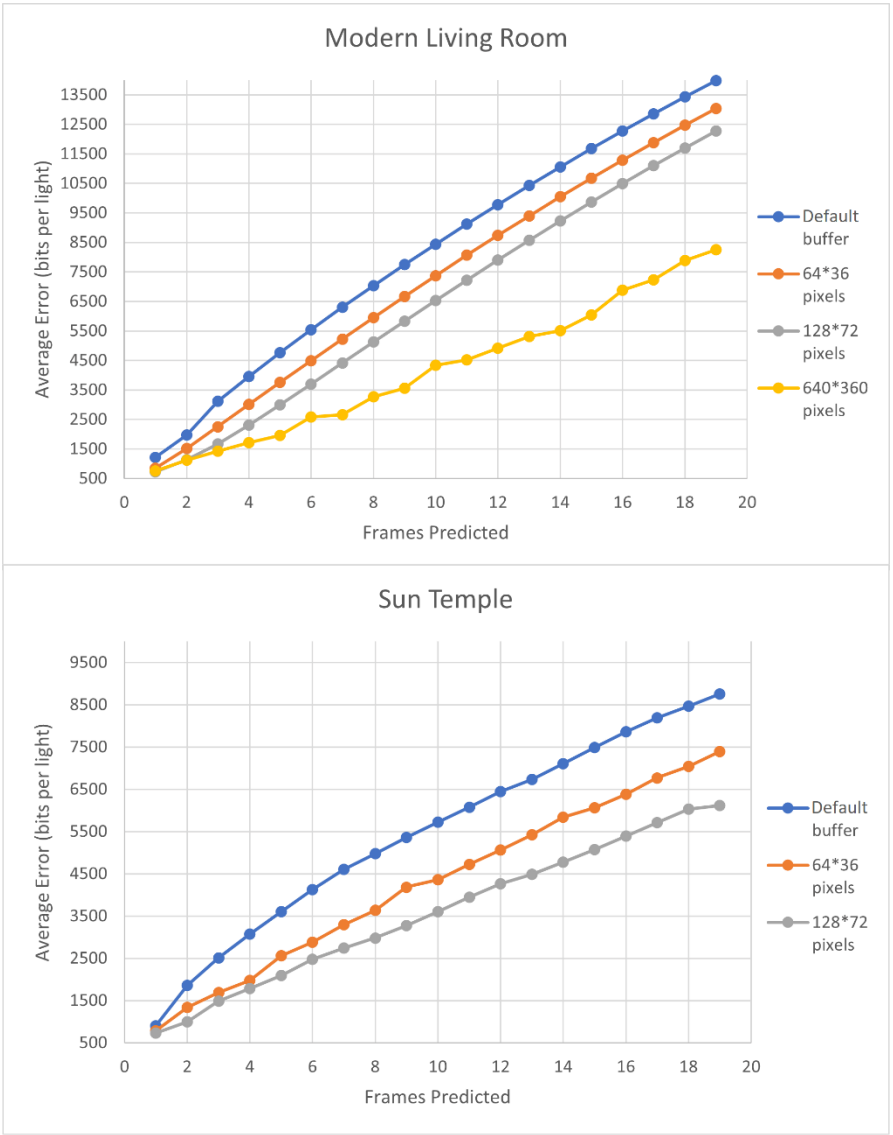


Figure 16: Bitwise error of prediction with different sizes of visibility buffer.

As the number of frames predicted increases, we can see that the average error increases, which would be expected as there is a larger deviance between the data received by the client and the ground truth visibility buffer data. From Figure 16, we can also see that the error decreases when a larger visibility buffer is used, and this result supports our earlier hypothesis.

In Figure 17, we measure the video quality of the rendering pipeline with two different sizes of visibility buffer. The visual metric used is the Video Multi-Method Assessment Fusion (VMAF) developed by Netflix, which is a metric based on machine learning that correlates strongly with subjective scoring done by real humans (Zhi et al., 2018).

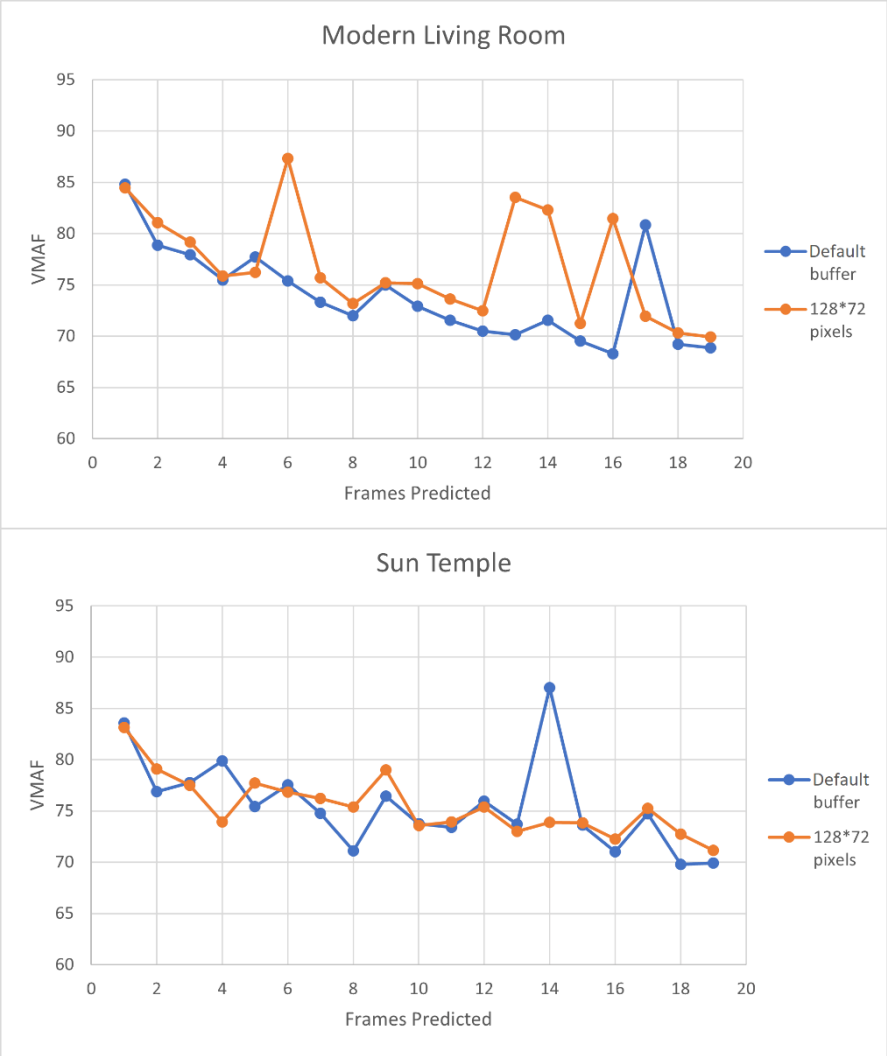


Figure 17: VMAF scores for videos rendered with two different visibility buffers.

We can see a general downward trend in the VMAF scores as the number of frames predicted increases, which indicates that video quality is falling. However, the data is noisy, and there appear to be a few outlier data points. We can also observe that the larger 128 * 72 pixel buffer fares slightly better than the default buffer. Both of these observations match with the results from the bitwise comparison.

As for image quality, we rely on two metrics, the structural similarity index measure (SSIM) and the peak signal-to-noise ratio (PSNR). We rendered images with a larger 128 * 72

pixel visibility buffer for different numbers of predicted frames, and compare them with the ground truth image that would be obtained if the server's and client's frames were exactly the same. The results are shown in Figure 18.

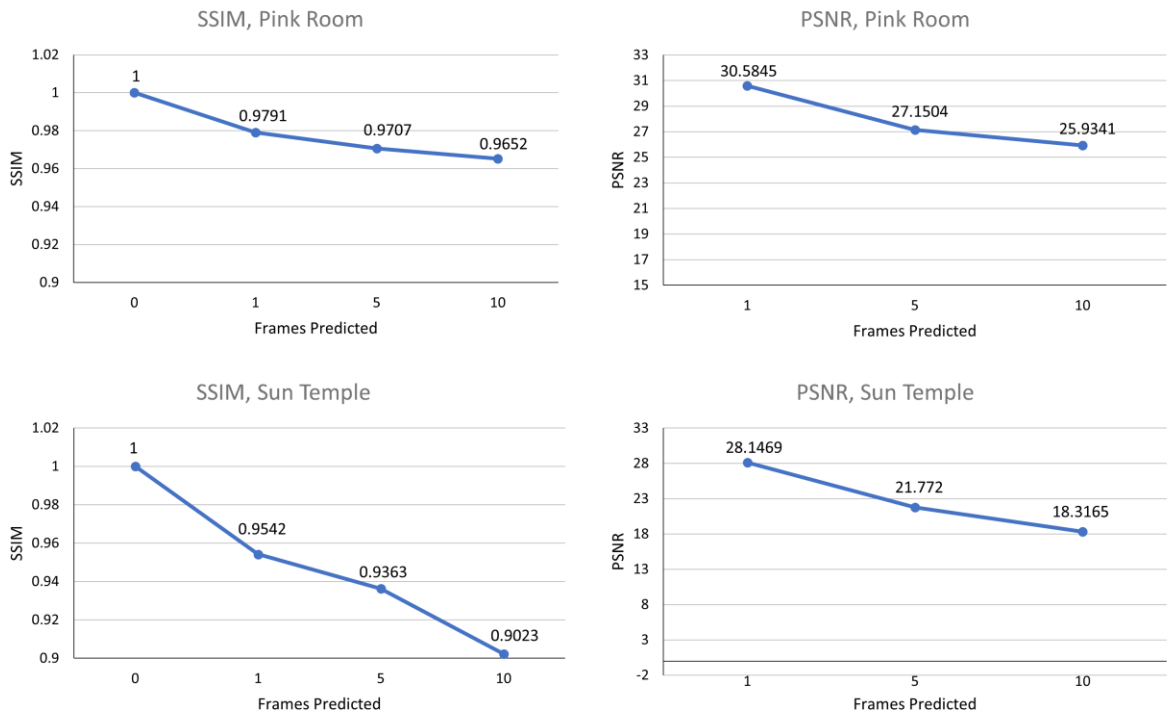


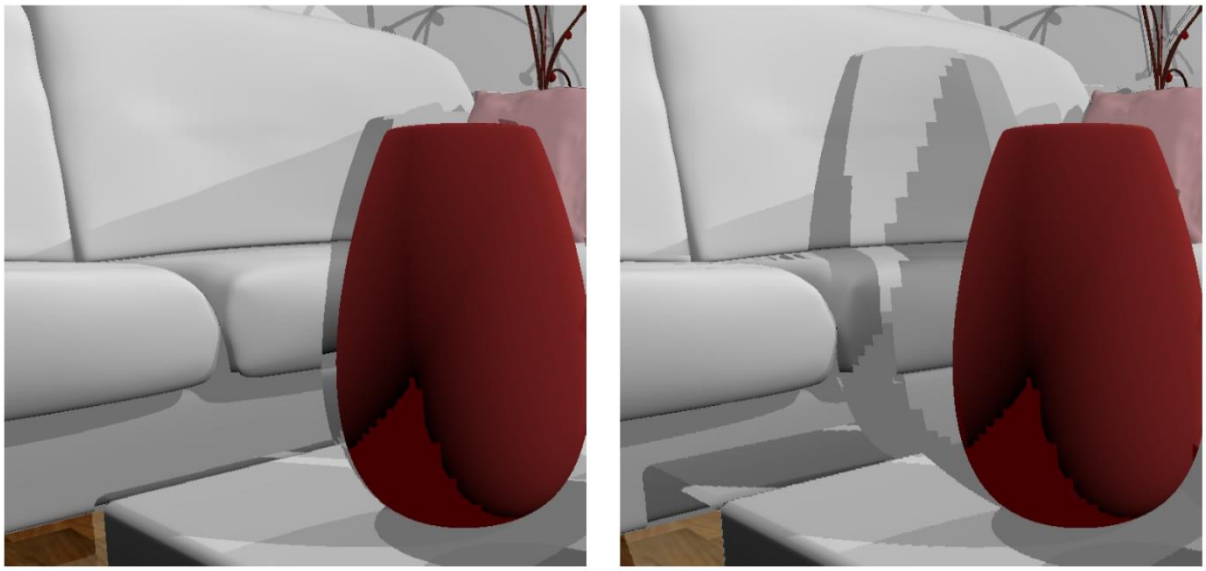
Figure 18: SSIM and PSNR scores for sample rendered images.

We can see that when the number of frames predicted increases, the SSIM and PSNR scores fall, indicating that the visual quality of the images is decreasing. Again, this finding lines up with the findings from the bitwise error measurement and the VMAF scores.

The next two figures contain samples of images obtained by prediction, which we use to provide additional observations about the visual quality of the image. For ease of comparison, we only look at two images for each scene at x frames predicted, one where $x = 1$ and one where $x = 10$.

(a) $x = 1$ **(b)** $x = 10$

Figure 19: Close-up of statue wing in Sun Temple scene.



(a) $x = 1$ **(b)** $x = 10$

Figure 20: Close-up of red vase in Pink Room scene.

Figure 19 and Figure 20 both show the inaccuracies in the illumination of the scene. The incorrect shadow beside the wing of the statue in Figure 19 is more obvious when the number of frames predicted is higher. Additionally, behind the red vase in Figure 20, we notice the presence of jagged shadow edges. These jaggies are the result of using the old visibility buffer for prediction, as the old buffer was rendered when the camera was further away. As a result, the visibility buffer generated has to be enlarged, leading to multiple pixels querying the visibility buffer at the same position, as a result sharing the same visibility buffer value. This may be resolved in the future by linear interpolation instead of taking a single point sample.

5. Conclusions

5.1 Summary

We have created a distributed rendering pipeline where rasterisation is performed on the target device and ray tracing is performed remotely to add shadows to the image. The pipeline produces images at a high interactive frame rate, making it suitable for use in video games. Although network latency will always be an issue, we have implemented various strategies to address it so that the image quality can be maintained. This work is of interest to the gaming industry, the mobile device, and the virtual reality headset industries, as it represents a new way of delivering interactive visual experiences to thin clients that may not support their rendering innately.

5.2 Limitations

Firstly, our implementation is still quite simple and only adds shadows to the rasterised scene. Adding other illumination effects may require us to change the networking portion of the program, as we had made several assumptions while designing it. For example, we assumed that the client will always send less data to the server than what the server sends to the client, which may not be the case if we implement algorithms where rasterisation would help in ray tracing.

If we were to send more data between the client and the server, we may need to apply different kinds of compression, as LZ4 compression may not work well on other kinds of data like RGB colour data, for instance. One alternative to our current compression method is to make use of motion vectors to send our information, similar to the way that video streams are compressed by using P-frames in H.264/AVC encoding. The downside of this is that errors would accumulate in the pipeline as it runs, and they would only be fixed when key frames or I-frames are sent.

Also, the frame rate of the pipeline is still not as high as the purely offline ray-tracing rendering pipeline, which operates at a frame rate of 200 FPS and above. Theoretically, because the client has been parallelised, it should be able to achieve such speeds as well. Timing of the various parts of the program will have to be done to find out the underlying reason for this,

whether it is due to unnecessary overhead in the networking part of the system, or whether it is due to inefficient implementations of the rendering passes.

Our application has only been tested with the Clumsy application, and it has not been subjected to real-world network conditions. One way to do this would be to route packets through a server located far away in a different country before sending it over to the recipient. Ideally, we would also want to test our rendering pipeline on the target devices that we have designed it for, like mobile phones and virtual reality headsets, and gather feedback from users of these devices.

5.3 Recommendations for Further Work

To improve the accuracy of the images, one possible modification to the pipeline could be to perform server-side prediction in addition to the current client-side prediction. The server would render a few different visibility buffers based on the possible future locations that the camera may move to. These visibility buffers would be almost identical to each other, containing much duplication of data, so they should be able to be highly compressed when they need to be sent to the client. The client will then select the visibility buffer that corresponds most closely to the actual camera motion and use that for the client-side prediction.

Different methods of data compression that are more applicable to the nature of the data should be explored, such as H.264/AVC encoding that was mentioned briefly in Section 5.2. Currently, we are only compressing the data within a single frame, but since consecutive frames would contain highly similar data, compressing the data across different frames would be expected to lead to a more effective compression. The advantage of using video compression techniques is that they are so common in applications nowadays, that there are now hardware encoders and decoders available for use on GPUs to accelerate the process.

We would also need to compare our current method to pure ray tracing and see if there is any benefit to sending only the visibility data back to the client rather than the full image data. Due to the coherence and cache-friendliness of the buffers in the ray-tracing pipeline, it may be faster to continue to use the visibility buffer to render the final pixel colour at the server and send that to the client to be displayed instead. This was highlighted in the review of our submission to the 2021 Pacific Graphics conference.

Our current pipeline only supports a still scene and a moving camera, but in video games, there are often multiple objects moving in the scene at any one time. Moving forward, we will need to support the rendering of such scenes with moving objects, and this will require some changes to the pipeline. For example, the world-space position data of every object may need to be sent to the server, along with the camera data, to allow it to generate accurate visibility buffers for the client.

References

Andronico, M. (2021, August 17). *Can't find a PS5 or Xbox Series X? Here's why Google Stadia is a solid alternative*. CNN Underscored. https://www.cnn.com/2021/08/17/cnn-underscored/google-stadia-review/index.html

Collet, Y. (2020). *LZ4—Extremely fast compression*. https://github.com/lz4/lz4/blob/9cf52b315192d4c66fde47cc8217b1cccfd19dc7/README.md

Crassin, C., Neyret, F., Sainz, M., Green, S., & Eisemann, E. (2011). Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum*, *30*(7), 1921–1930. https://doi.org/10.1111/j.1467-8659.2011.02063.x

Di Domenico, A., Perna, G., Trevisan, M., Vassio, L., & Giordano, D. (2021). A network analysis on cloud gaming: Stadia, GeForce Now and PSNow. *Network*, *1*(3), 247–260. https://doi.org/10.3390/network1030015

Duckett, C. (2021, June 2). *Nvidia CEO eschews mobile RTX in favour of GeForce Now*. ZDNet. https://www.zdnet.com/article/nvidia-ceo-eschews-mobile-rtx-in-favour-of-geforce-now/

Edelsten, A., Jukarainen, P., & Patney, A. (2019, March 21). *Truly next-gen: Adding deep learning to games & graphics*. Game Developers Conference, Moscon Center, San Francisco. https://www.gdcvault.com/play/1026184/Truly-Next-Gen-Adding-Deep

Gerardo, D. (2019, March 7). *Unlock your full potential—How higher frame rates can give you an edge in battle royale games*. NVIDIA GeForce. https://www.nvidia.com/en-us/geforce/news/geforce-gives-you-the-edge-in-battle-royale/

Janzen, B. F., & Teather, R. J. (2014). Is 60 FPS better than 30? The impact of frame rate and latency on moving target selection. *CHI '14 Extended Abstracts on Human Factors in Computing Systems*, 1477–1482. https://doi.org/10.1145/2559206.2581214

Jess, G. (2019, November 18). *Review: Google Stadia*. Wired. https://www.wired.com/review/google-stadia/

Jhang, J.-W., & Chang, C.-F. (2021). An unbiased hybrid rendering approach to path guiding. *Eurographics 2021 - Posters*, 2 pages. https://doi.org/10.2312/EGP.20211032

Low, D. (2022, February 15). Singapore tops in global ranking of median fixed broadband speeds. *The Straits Times*. https://www.straitstimes.com/tech/tech-news/singapore-tops-in-global-ranking-of-median-fixed-broadband-speeds

Milanian, B. (2021, January 7). Game on: As 5G deployments speed up, so will cloud gaming. Forbes. https://www.forbes.com/sites/forbescommunicationscouncil/2021/01/07/game-on-as-5g-deployments-speed-up-so-will-cloud-gaming/

Mims, C. (2020, December 12). Intel not inside: How mobile chips overtook the semiconductor giant. *Wall Street Journal*. https://www.wsj.com/articles/intel-not-inside-how-mobile-chips-overtook-the-semiconductor-giant-11607749203

NVIDIA Corporation. (2015, October 6). *GPUDirect*. NVIDIA Developer. https://developer.nvidia.com/gpudirect

Parhizgar, D., & Svensson, M. (2021). Hybrid ray traced and image-space refractions. In A. Marrs, P. Shirley, & I. Wald (Eds.), *Ray Tracing Gems II* (pp. 457–467). Apress. https://doi.org/10.1007/978-1-4842-7185-8_29

Pharr, M. (2021, August 9). *Leading lights: NVIDIA researchers showcase groundbreaking advancements for real-time graphics*. The Official NVIDIA Blog. https://blogs.nvidia.com/blog/2021/08/09/siggraph-research-real-time-graphics/

Potluri, S., Hamidouche, K., Venkatesh, A., Bureddy, D., & Panda, D. K. (2013). Efficient inter-node MPI communication using GPUDirect RDMA for Infiniband clusters with NVIDIA GPUs. *2013 42nd International Conference on Parallel Processing*, 80–89. https://doi.org/10.1109/ICPP.2013.17

Tao, C. (2021). *Clumsy, an utility for simulating broken networks for Windows Vista / Windows 7 and above*. https://jagt.github.io/clumsy/

Wyman, C. (2005). An approximate image-space approach for interactive refraction. *ACM Transactions on Graphics*, *24*(3), 1050–1053. https://doi.org/10.1145/1073204.1073310

Xiao, L., Nouri, S., Chapman, M., Fix, A., Lanman, D., & Kaplanyan, A. (2020). Neural supersampling for real-time rendering. *ACM Transactions on Graphics*, *39*(4), 142:142:1-142:142:12. https://doi.org/10.1145/3386569.3392376

Zhi, L., Bampis, C., Novak, J., Aaron, A., Swanson, K., Moorthy, A., & De Cock, J. (2018, October 26). *VMAF: The journey continues*. Netflix Technology Blog; Medium. https://netflixtechblog.com/vmaf-the-journey-continues-44b51ee9ed12