B.Comp. Dissertation

DHR: Distributed Hybrid Rendering using Edge Computing for Thin-Client Games and Metaverse Experiences

By

Nicholas Nge Jing Shou

Department of Computer Science

School of Computing

National University Singapore

2021/2022

B.Comp. Dissertation

DHR: Distributed Hybrid Rendering using Edge Computing for Thin-Client Games and Metaverse Experiences

By

Nicholas Nge Jing Shou

Department of Computer Science

School of Computing

National University Singapore

2021/2022

Project No: H113680

Advisor: Dr Anand Bhojan

Deliverables:

Report: 1 Volume

Abstract

This project explore a distributed hybrid-rendering approach of incorporating real-time ray tracing into thin-client games and metaverse experiences by performing the expensive ray tracing computations on powerful cloud hardware and raster-based rendering on user access devices. Specifically, it works towards a complete hybrid rendering game engine using a distributed approach. It further explores adjustments made in order to improve and balance the three metrics of performance, response time, and graphics quality.

Subject descriptors:

Computer Graphics
Game Development

Keywords:

Visual Computing, Computer Graphics, Computer Games

Implementation Software and Hardware:

C++, Falcor, NVIDIA RTX Series GPUs

Acknowledgments

I thank Dr Anand Bhojan, my supervisor, for his insight, guidance, understanding and support throughout this project, and for accepting me into this project initially even though I was inexperienced. I thank Yu Wei, for her continued encouragement and help on so many things. I thank Mr Chow for helping me to set up the lab machines at Visually Immersive Studio, and I thank my teammate Alden.

Table of Contents

Chapter 1

Introduction

1.1 Hybrid Rendering

A hybrid renderer combines the speed of the rasteriser and the context awareness of the ray tracers by rasterising a G-buffer representing all the visible surfaces, before using its ray-tracing capabilities to compute the lighting and reflections for those surfaces. This allows it to generate high-quality graphics while maintaining frame rates required for real-time applications. If full ray tracing is performed, geometrically complex scenes or scenes with a lot of lights may take a long time to compute, hence we only apply ray tracing partially for selected lighting effects or pixels etc where the visual quality of rasterization falls short. Selective usage of shadows computation, ambient occlusion or global illumination are common techniques selectively employed in hybrid rendering.

Figure 1 Left: Hybrid ray tracing in PICA PICA, Right: Hybrid rendering pipeline

However, on mobile devices and current VR headsets, even hybrid rendering with selective ray tracing struggles to meet the tight performance constraints of games. Unlike real-time ray tracing dedicated GPUs such as NVIDIA GeForce RTX series cards or AMD Radeon RX series cards, mobile device and metaverse system hardware catered for the consumer market cannot perform real-time ray tracing fast enough to meet the requirements of 60 fps for non-metaverse games and 90 fps for metaverse applications, the common respective benchmarks. For VR/AR games, a higher fps is needed to prevent motion sickness and to suspend belief, since the user/player is expected to believe to some extent the reality of the created scene. Nonetheless,

ray tracing-based rendering can generate more advanced lighting and camera effects as compared to traditional rasterization for more photorealistic and immersive graphics.

1.2 Distributed Rendering

Rendering graphics can require massive computational resources for complex scenes that arise in scientific visualization, medical visualization, CAD applications, and virtual reality.

In offline rendering, distributed rendering is a rendering technique where multiple machines across a network render a single frame of a scene or image. The frame is divided into smaller regions, and each machine receives some of them to render. Once each region has been rendered, it is returned to the client machine and combined with other rendered regions to form the final image.

In real time rendering, the same concept of leveraging the graphics capability of remote servers for rendering is applied. The most common application of real time distributed rendering is in cloud gaming, where the user's access device (e.g. gaming console, PC) handles player inputs and communicates them to the cloud, which helps with the generation of output game contents in the form of video frames.

In current cloud gaming systems, server performs the full load of rendering, and the output to the client is an encoded video stream, which the client receives and decodes. The experience bears similarity to simply streaming a video, since the local GPU does not participate in the rendering. Image-based streaming is the standard for cloud gaming, being able to remove any dependency on the client hardware since client is only responsible for sending inputs and reading and receiving the output image feed. High-fidelity graphics can thus be achieved, regardless of the client side device. Popular cloud gaming services like Google Stadia, NVIDIA GeForce NOW and Amazon Luna, all employ image-based streaming.

1.3 Distributed Hybrid Rendering

In image based streaming, there is no dependency on the client hardware since the image feed is simple to work with. If we instead assume *some* level of competence in the client hardware, we can leverage this limited graphics capability for rasterization. Hence, combining the approaches of hybrid rendering and distributed rendering, we choose to perform ray tracing on cloud servers, leveraging ray tracing-accelerated GPUs, while the client performs the rasterization steps. This distributed rendering approach to hybrid rendering can give us not only interactive frame rates through raster-based rendering on the client, but also high-quality graphics through ray tracing on the server.

Chapter 2

Study of Existing Work

This FYP builds upon previous work of the hybrid rendering team led by Dr Anand Bhojan, hybrid rendering engine *hrender*. Hence it is necessary that this section covers both the literature to understand the project and elaboration on the existing, inherited infrastructure of the *hrender* project codebase.

2.1 **Shading Model**

The existing work utilises deferred shading, where rasterization is first performed to generate a G-Buffer containing data required for shading computation. Next, light visibility information is obtained via ray tracing for every light per pixel. The G-Buffer and light visibility information are then used to obtain the final pixel color I as shown.

$$I = \frac{I_d}{\pi} \sum_i k_i \cdot saturate(N \cdot L_i) I_i$$

In the equation, $\frac{I_d}{\pi}$ refers to the pixel's diffuse BRDF with I_d as its material diffuse color. N denotes its surface normal, and k_i, L_i and I_i refer to the relative visibility, direction and intensity of light i respectively in relation to the pixel. A shadow ray for each pixel is traced from its world position to every light in the scene and a light is deemed visible if the ray reaches the light without hitting any other object.

The pixel world positions can be obtained from rasterization or ray casting. For rasterization, both the client and the server perform it to obtain the world positions. This is to minimize data transfer since by both parties performing rasterization, the world positions do not have to be sent over the network. This is also made possible because rasterization is very fast in the server.

In the original non-distributed ray-traced shadow algorithm, the color contributions of lights are computed and accumulated while looping through every light and tracing rays to it. DHR

separates this color computation and the ray tracing loop so they can be performed on different hardware for better parallelism and performance. As such, the ray tracing process only stores the visibility boolean k_i of every light in a visibility bitmap buffer such that every pixel is assigned a compact bitmask representing all the lights in the scene. For instance, if light 2 is visible from a particular pixel, querying the bitmap at its coordinates will produce a bitmask with bit 1 at position 2. The number of bitmaps and size of per-pixel bitmasks can be tuned based on the number of lights in the scene.

Figure 2. The output of the shading model on the scene The Modern Living Room.

The images on the left represent the visualization of the ray-traced visibility buffer encapsulating information for 3 scene lights where the material color of each pixel is shown if the respective light is not obstructed from its world space position. The right image is the final result of combining the ray-traced shadow information in the visibility buffer with Lambertian shading at the client.

2.2 Falcor

The core rendering framework used for this project is Falcor, an open source real-time rendering framework built by NVIDIA on top of rendering APIs DirectX12 / Vulkan. Critically, Falcor supports DirectX ray tracing (DXR), and has helpful UIs and abstractions of the render pipeline and passes for rapid prototyping.

Figure 3 Demo of scenes rendered in Falcor

Falcor is widely used from home projects and prototyping work in NVIDIA itself, but the target audience of Falcor is the researcher. As of its latest update two weeks ago, Falcor is now shipping with other NVIDIA RTX SDKs out of the box: DLSS, RTXGI, RTXDI, NRD, attesting to the central piece it plays in the development process.

2.3 Existing Framework

2.3.1 Optimization – UDP and compression

The project utilises UDP and compression of the visibility buffer to speed up the performance. UDP was implemented using winsock API and compression is available in parallel LZ4 compression or NVENC, a hardware encoder in selected NVIDIA GPUs. Due to difficulty in

implementing the H264 decoder, the current implementation continues to use LZ4 compression.

2.3.2 Sequential model

In the initial sequential model, client sends the camera data of a frame and waits for the corresponding visibility buffer to be received from the server. This approach was quickly scrapped as it produced very low frame rates since a significant portion of the time was spent idle and waiting for server to complete rendering and send the result over. However, the visual quality of the output was guaranteed in the sequential model.

2.3.3 Non sequential model

The non-sequential model (shown in the next page) was subsequently adopted where the frame rendered by the client was a result of the GBuffer of the current frame and the visibility buffer most recently received. This idea allowed for a significantly faster fps and no latency.

Two network threads were used in the client and two network threads in the server. Client's sending thread sends camera data to server at the start of every frame. The receiving thread waits to receive the visibility buffer from the server. Likewise, the server sending and receiving thread sends the visibility buffer and receives the camera data respectively. This allowed the rendering loop to be faster and more independent of network activity (Though it was not completely independent as semaphores were used for signalling).

However, this model caused misalignment of the visibility buffer with the GBuffer, and created a distorted image, especially when camera is moving. Hence, prediction is used to predict a future visibility buffer with a recently received one. The details of prediction of the visibility buffer will not be elaborated on here. In prediction, the number of frames ahead to predict can be specified. The higher the number, the more error prone the prediction is.

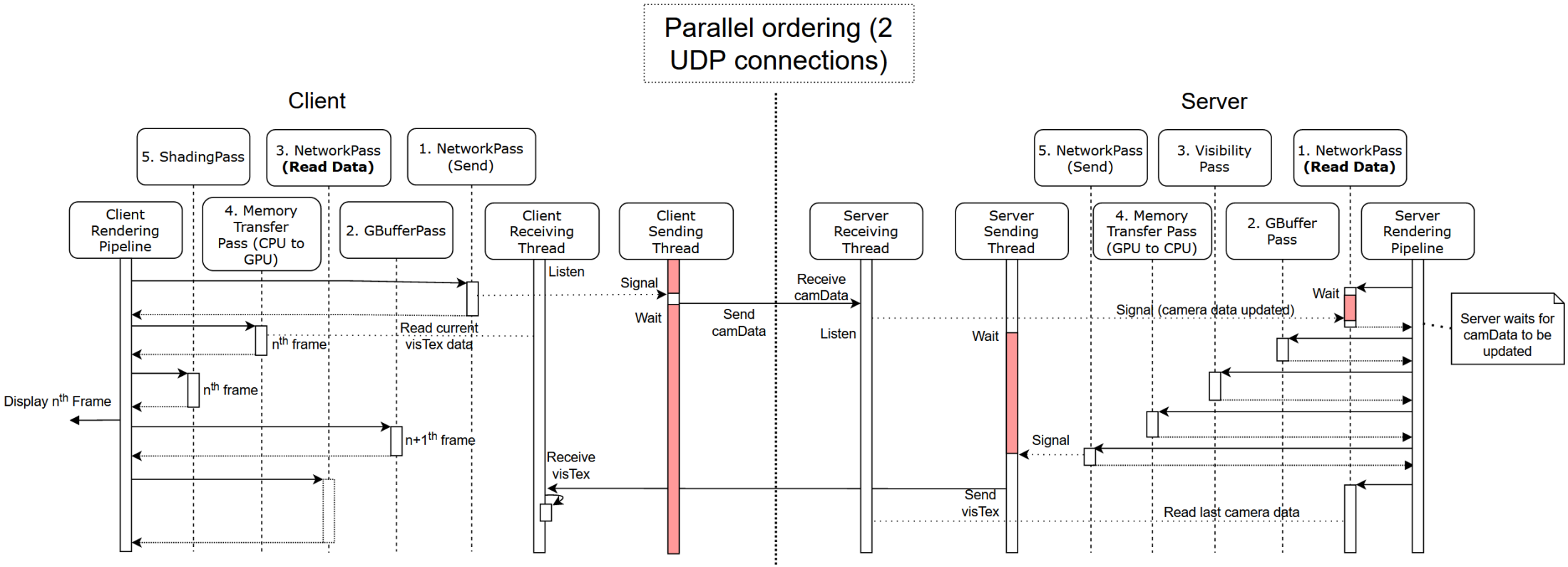


Figure 4 Retired non-sequential model

Chapter 3

Design

Section 3.1 highlights the three resources to balance for the project. Sections 3.2 to 3.4 highlight the suite of three solutions to improve performance, response time, and graphics quality respectively.

Sections 3.6 and 3.7 focuses on support features. Section 3.6 develops a modular system which allowed changes to the control logic to be made more conveniently and quickly. In Section 3.7 we extend the server's capabilities to support multiple clients.

3.1 Balancing performance, response time, & graphics quality

Performance, response time & graphics quality are three metrics to improve in the existing system, but they each come at a cost of each other. Performance is measured by frames per second (FPS) or how long rendering one frame takes.

We use graphics quality to refer to the artefacts or errors arising from the misalignment of the visibility buffer and the G-buffer in a non-sequential setting. For example, if the visibility buffer is 1 frame behind, the graphics quality will suffer since the two buffers do not belong to the same frame. How we measure this metric is explored in Section 4.1.

We refer to response time as the time from when a frame's camera data is sent by the client to when that frame's visibility buffer data is ready (ie. it is ready for the shading pass and subsequent output/blit). This response time consists time in the network, server rendering time. For example, if the server RTT is 100ms and server rendering takes 8ms, the response time will be around 108ms (typically slightly more). If however we predict 3 frames, we reduce the response time by $8 * 3 = 24$ms to 84ms because that frame will be ready 3 frames earlier.

3.2 Framework revisions – Improving performance & graphics quality

The original fully sequential approach in Section 2.3 achieves great visual quality, but only works in an environment with extremely low latency, since we have to wait for the server's texture before starting the next frame. Additionally, it suffers from very low frame rates.

The fully non-sequential approach mentioned in Section 2.3 on the other hand shades the latest GBuffer of the client (rendered locally) with the latest visibility buffer received from the server. While this resulted in good frame rates and a low response time since the most updated version of the scene is immediately rendered, the visual result was often not aligned, worsened still when network latency is high.

The sequential approach gives an accurate image while the non-sequential approach achieves lower latency and less reliance on the quality of the network. We settle on a revised framework shown in the diagram on the next page which achieves visual quality while maintaining a high frame rate.

Considering network latency, while the client updates the server with the camera data for every frame, the frame it displays to the user is dependent on the latest visibility buffer received from the server. As such, although the client can compute the most updated G-Buffer locally, its latest received visibility buffer will be around p frames behind, where p is the response time divided by the total time taken to render and display one frame.

The client continues to update the scene with the user inputs, and send the most updated camera data at the start of each frame to the server. However, it also internally stores a queue containing camera data for which visibility buffer has not been received (from the server). Upon receiving visibility buffer, it pops the relevant camera data from the queue to use for render the GBuffer and subsequently perform shading together with the received visibility buffer. This way, we can maintain high frame rates while keeping visual quality good.

However, this approach is still not great in network conditions with high latency, hence need to incorporate the prediction ideas from our non-sequential framework into our solution.

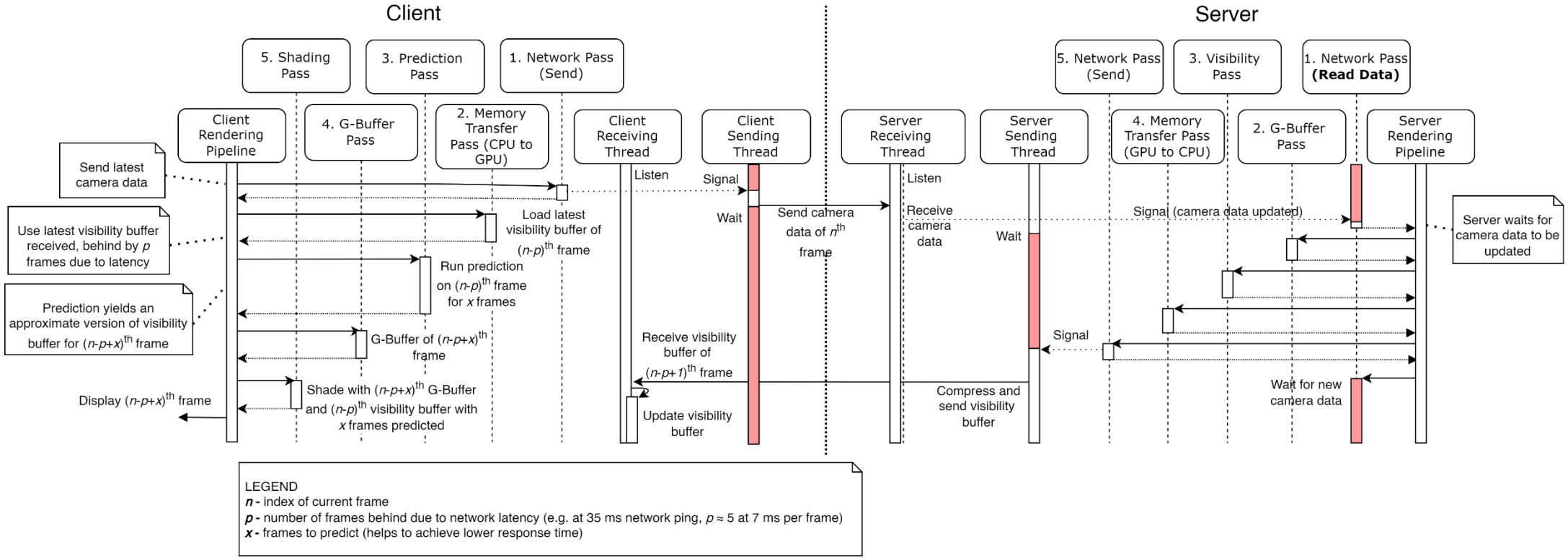


Figure 5 Revised framework - Control flow diagram for client and server

3.3 Predicting only *some* frames ahead – Reducing response time

While the revised framework improved performance especially when network latency was high, it still suffered from high network latency since the response time would be long. In AR/VR systems, acceptable response times are typically around 50ms minimally with around 20ms considered ideal, due to the tendency for users to experience motion-sickness.

Before we revised the framework, we made use of prediction of the visibility buffer to minimize errors arising from the scene misalignment of the G-buffer and the visibility buffer. We can now re-incorporate this idea into our approach but allow ourselves to vary the intensity of the prediction.

Predicting not all but only some frames ahead creates an interesting trade-off between response time and graphics quality. For every frame ahead that we predict, we reduce the response time by the time needed for one frame since we will receive the visibility buffer needed 1 frame earlier. On the other hand, the graphics quality suffers since prediction becomes less accurate.

We refer to the legend of the diagram on the previous page for the variables n, p, and x. n refers to the index of the current frame in the client rendering, p refers to the network latency measured in number of frames – ie. if one frame takes 7ms and network ping is 70ms, $p = 10$. x refers to the number of frames ahead we choose to predict. Every rendering loop, the client will send out the camera data for frame n, and expect to receive visibility buffer for frame n-p, since it is lagging by around p frames. Then, it will predict visibility buffer n-p+x using visibility buffer n-p, and use the result to shade and output frame n-p+x. The response time then is given by p-x times the time for one frame. Following the above example, if $x = 3$, even though network ping is 70ms, client experiences only 70 – 21 = 49ms response time. This allows the client to experience a faster "response time" than the actual response time while keeping the amount of approximation used to a pre-defined minimum so as to maintain the level of visual quality for the user.

The higher the value x chosen, the more frames ahead we predict, leading to better response times but lower graphics quality due to having to predict more frames ahead (prediction error). This establishes the trade-off when choosing the value x.

3.4 Prediction buffer - Improving graphics quality

During prediction, we do not know the visibility information of scene points revealed by camera movement, so their corresponding pixels are taken to be fully illuminated with respect to all scene lights. Doing so helps to prevent adding false shadows to the scene with camera movement. However, we see prominent borders of illuminated pixels along the edges of the frame in the direction of camera movement as the offset positions given by the motion vectors fall outside the boundary of the original visibility buffer. In the image below, the camera is tilting towards the top right direction, and while prediction for the rest of the shadows is good, the borders on the left and top reveal non-existent shadow data.

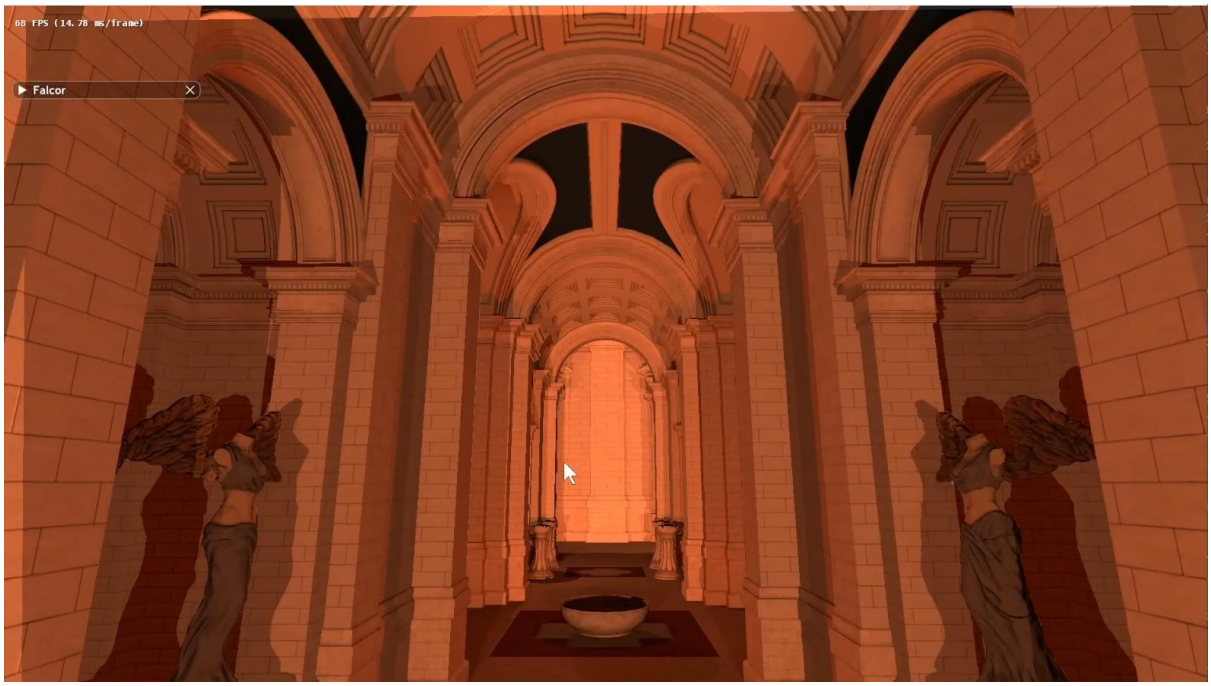


Figure 6 Prediction error at borders of frame

We minimize this error at a small performance cost by working with textures slightly larger so that some visibility information is available past the edges of the frame. For example, taking a display resolution of 1920×1080, we work with a larger 1984×1116 visibility buffer texture with an additional 64×36 of pixels resolution. This allows shadow data to still be available for some amount of space outside the display resolution, which prediction can use to fill areas revealed by camera movement. We will call the size of the textures we work with the size of the prediction buffer. The larger the prediction buffer, the more prediction can withstand sudden camera movements without errors at the borders.

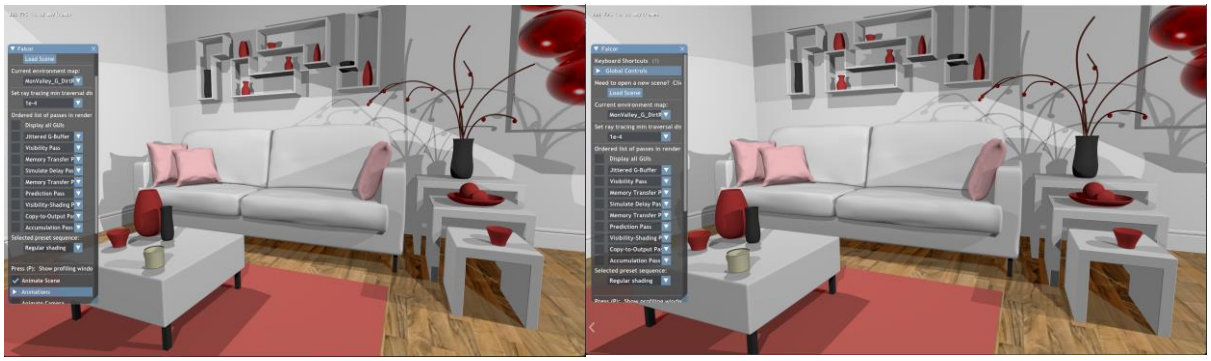


Figure 7 Before correction, right picture appears more zoomed in since it is using a larger prediction buffer

By working with larger textures, the image displayed will change and appear more zoomed in. To counteract this side effect, the focal length of the camera has to be adjusted to maintain the same displayed region of the scene for different numbers of additional pixels to the display resolution.

$$New\ focal\ length = Old\ focal\ length * \frac{1920}{prediction\ buffer\ width}$$

We choose to test with three prediction buffer sizes, 0 (1920×1080), 64×36 (1984×1116) and 128*72 (2048×1152). We choose to use 64-pixel increments in the width for optimization reasons. D3D12_TEXTURE_DATA_PITCH_ALIGNMENT, a value relating to the pitch of the GPU in use, is 256 bytes in our set up. This means the bytes needed for one width line (ie. 1920, 1984 or 2048) should be in a multiple of 256. Given that the visibility information is 4 bytes per pixel, this means that we have to increment the width size in 64-pixel increments. The size of the width then is set to maintain the aspect ratio of 16:9.

3.5 Automatic switching

This section covers a feature developed to automatically switch between sequential and non-sequential frameworks before the revised framework mentioned in Section 3.2 was introduced. Even though this section has since been turned off as we migrate to the revised framework, I believe the core idea can still be recycled to determine dynamically the parameters used in the framework, particularly value *x*, the number of frames ahead to predict. Below is a description of the idea, which was applied to the old sequential and non-sequential frameworks.

Automatic switching looks at the 1. Current network latency and 2. Current frame's camera movement data to determine if client should utilise a non-sequential or a sequential framework. If client camera movement is sharp and sudden, we expect great error in the subsequent frames in the non-sequential framework as prediction will not be accurate and we should switch to sequential framework. If client camera movement is gentle, we can stay in non-sequential mode. Similarly, when network latency is high, we should prioritise non-sequential to improve frame rates.

To do this, the 3 (x, y, z) coordinates of the camera's position, up, and target vector are taken (total 9 coordinates) together with the network latency *('Time for one frame'* refers to the network latency, though it may have been worded improperly in the UI) are taken and multiplied by their corresponding weights to produce a single value. The weights have been taken to make sure each value has a fair contribution to the final sum, but regression techniques can be employed in the future to fine-tune these values.

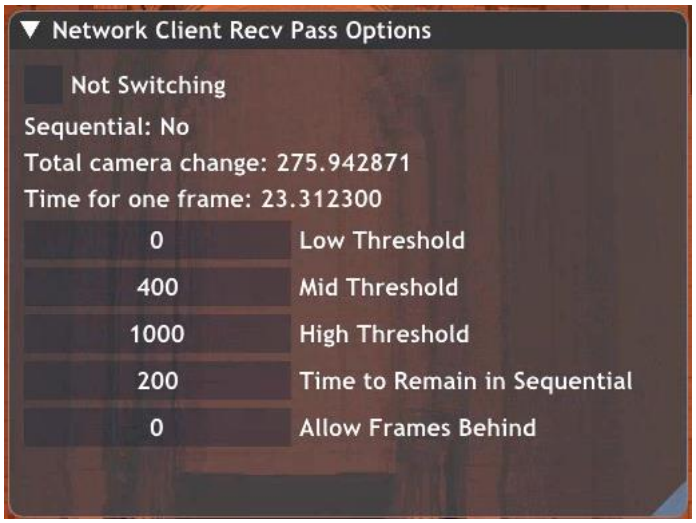


Figure 8 UI for switching

Threshold values could be set to adjust the smoothness of the switching. There are three threshold values, high mid and low. The client is in non-sequential mode by default. Upon crossing the high threshold, the client enters into sequential mode for a number of frames corresponding to the value set for *'Time to Remain in Sequential'*. Only after those number of frames, client checks again where our assessed value lies.

1. If it is above the mid threshold when we are already in sequential, we remain in sequential and reset *'Time to Remain in Sequential'*.
2. If we are in between the low and mid threshold, we remain in sequential but do not reset *'Time to Remain in Sequential'* (ie. we will keep monitoring every frame)
3. If we are below low threshold, we immediately switch back to non-sequential.

3.6 A Modular System

I found that the existing infrastructure of the project resulted in highly coupled code. Because the render passes often needed to access each other's resources/textures, trying a small change in the code resulted in having to change multiple parts of the code in different places, which slowed down development.

Eg. The compression pass would need to access the visibility buffer, the network manager would have to access the output compressed texture of the compression pass. However, if compression was turned off, or if new passes were added in between in the ordering, the passes would have to retrieve their input textures / place their output from and to a different place.

An existing solution was Falcor's ResourceManager, a part of the Falcor API that allows resource sharing. However, this still does not solve the problem, since the render passes still have to know what resources to retrieve from the ResourceManager based on the render ordering.

```
266  RenderConfiguration renderConfiguration = {
267  1920, 1080, // texWidth and texHeight
268  0, // sceneIndex
269  5,
270  { // Array of RenderConfigPass
271      NetworkServerRecvPass,
272      JitteredGBufferPass,
273      VisibilityPass,
274      MemoryTransferPassGPU_CPU,
275      NetworkServerSendPass
276  }
277  };
```

Figure 9 Setting RenderConfig

A RenderConfig file was created, storing the rendering ordering, scene used (pink room/sun temple), and display resolution (1920×1080). Based on only the information specified RenderConfig file, the paths and input/output for each of the render passes will be set up before rendering begins, and developers only have to edit that file to try new configurations, speeding up development time. This RenderConfig file can also be easily implemented to be communicated by the client to the server at the start of the connection, in which case it will only have to be set in the client code once.

3.7 Multiple clients

As the performance of the system was improved, frame rates exceeded 150 fps when building on Release mode. As an experimental idea, we wanted to test if a single server was capable of supporting multiple clients. The server code was extended to include capabilities to accept more than one client. In the current implementation, when multiple clients are connected to the server, the server naively alternates between clients, rendering a visibility buffer and sending the respective client.

The metric used to measure the change in performance however, would not be fps. In the current implementation, fps of the client is completely independent of how many frames the server sends it per second, ie. a client can achieve 150 fps while server only sends 140 new visibility buffers per second, hence only having 140 unique frames. Instead, unique visibility buffers per second (vbps) sent by the server would be the metric to measure, since with more clients the server's time per client is reduced and will send less buffers to the client.

Chapter 4

Discussion

We test our DHR implementation on a client and a server with ray tracing-accelerated GeForce RTX 2080 GPU. Both the client and server have an Intel Core i7-7700K CPU and 16GB RAM. Results from Section 4.1 Graphics Quality was tested separately on my home computer

4.1 Graphics Quality

There were two approaches in measuring graphics quality, ie. how well our prediction works. The first approach (bitwise error) quantitatively measures the error while the second approach (VMAF) includes an element of how the human will perceive the error.

4.1.1 Methodology – Bitwise Error

The first approach is through bitwise comparison of the actual visibility buffer and the predicted visibility buffer. This was done for each frame for the in-built animation sequences of the two scenes pink room and sun temple, lasting 14 seconds and 50 seconds respectively. For Pink Room, we fixed the frame rate at 60 fps and let the animation run twice, resulting in exactly 14*2*60=1680 frames. For Sun Temple, we ran the animation once at 58 fps for a total of 50*58=2900 frames. This was done for a varying number of prediction buffer sizes. The mean error was then divided by the number of lights in the scene to find the error (bits per light). Pink Room has 3 lights while Sun Temple has 8 lights.

*Additional notes: The constant frame rate was to ensure that animation proceeded consistently across measurements since the individual frames will be different if fps differs. Sun temple was run at 58 fps as 60 fps was not stable at 128*72 prediction buffer with the bitwise error calculation on.

4.1.2 Results – Bitwise Error

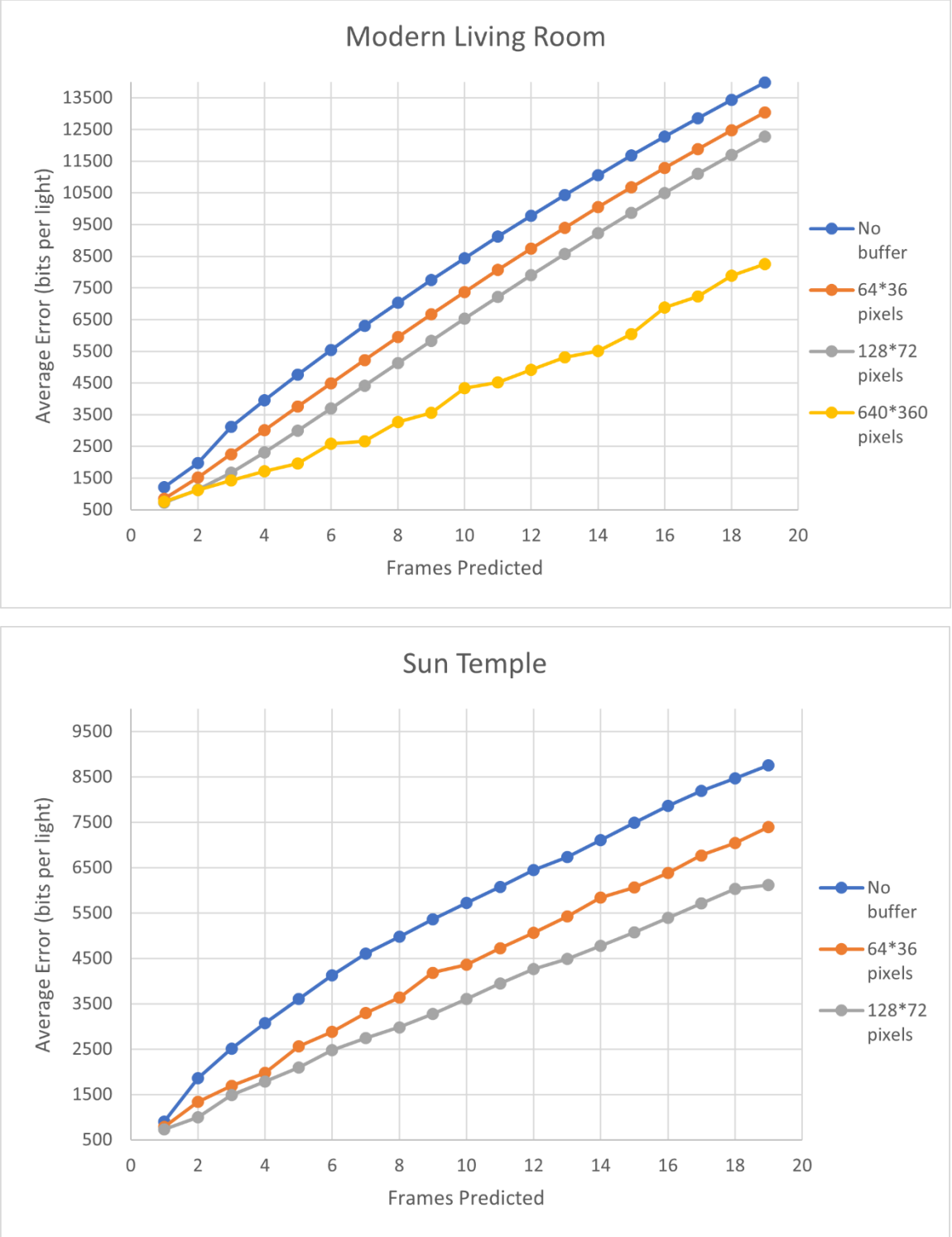


Figure 10 Average prediction error per light measured using bitwise comparison of correct visibility buffer with the predicted version at fixed frame rate of 60 FPS.

From objective visual metrics measurements as shown above, we see that visual quality decreases as more frames are predicted for the same frame n. This is because as x increases

while *n* remains constant, *n-p* decreases which means that an older visibility buffer is used for the prediction, resulting in a higher prediction error.

Increasing the size of the visibility buffer lowers the prediction error by roughly a constant amount, which can only arise from minimizing errors at the edges of the frame since increasing buffer size does not affect anything else. This has a large impact and is visually very noticeable at lower values of frames predicted, where error can be almost halved with the use of 128*72 additional pixels. We included a result of using an unreasonably large buffer of 640*360 pixels (i.e. 2560*1440) on Pink Room to showcase that a large portion of the error continues to arise from the edges of the frame. The same result was not applied on Sun Temple due to difficulty in maintaining the fixed 58 FPS while performing the error calculation.

4.1.3 Methodology – VMAF

The second approach is using Video Multi-Method Assessment Fusion (VMAF), a perceptual video quality assessment algorithm developed by Netflix. VMAF rates a "distorted video" against a "source video" and outputs a number from 0-100 corresponding to video quality of the distorted video, with 0 being very poor and 100 being perfect.

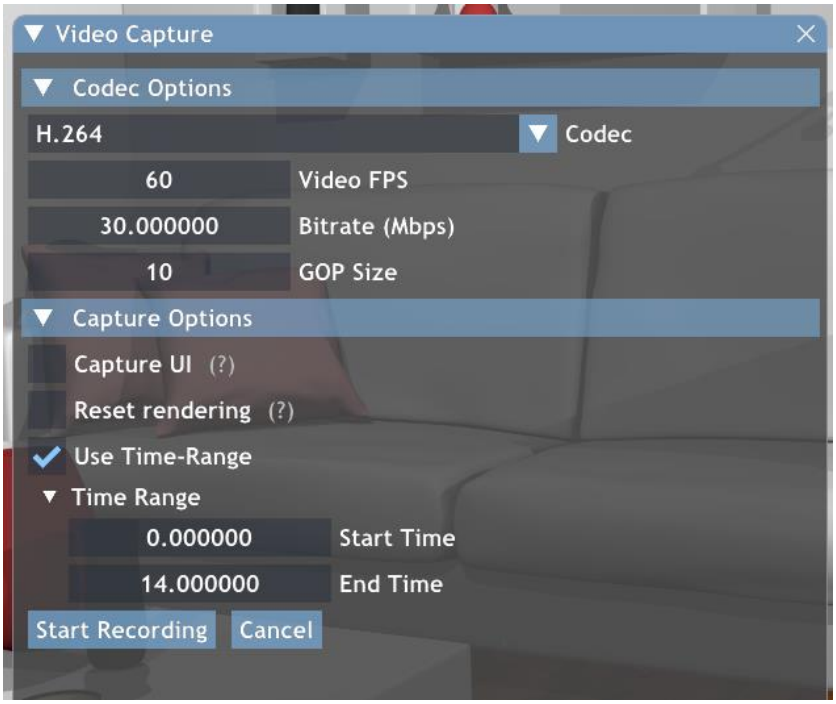


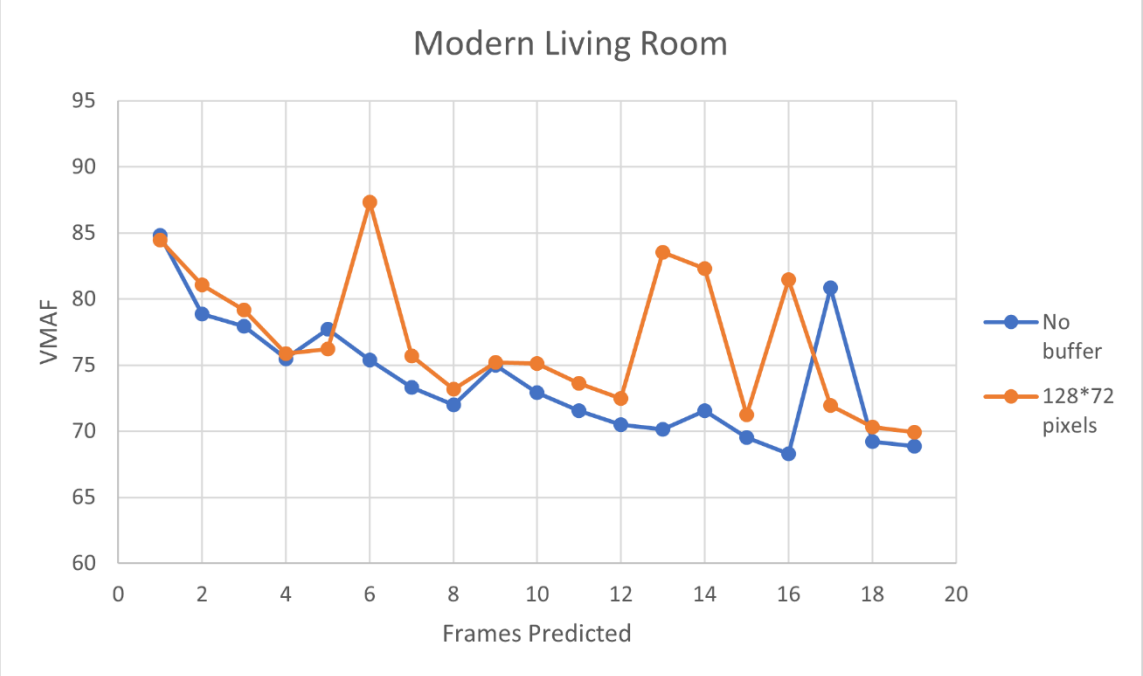
Figure 11Video Capture UI within Falcor

We utilise Falcor's in-built video capture to record the animation sequences into a video which we feed to VMAF, with the video running the animation without any prediction as the source video and videos using prediction from 1-19 frames ahead as the distorted videos.

The full 14 second animation sequence was used for Pink Room, whereas an excerpt of duration 20 seconds was taken from the Sun Temple 50 second long animation. Pink Room and Sun Temple were both recorded at 30 frames per second to reduce deviation from VMAF's recommendation of 24 fps samples. A discrepancy from the recommended would not affect the reading significantly, but it will be interpreted as a 24 fps video, which means the motion feature of VMAF will generate lower scores and the overall VMAF will be slightly lower.

4.1.4 Results – VMAF

For VMAF, we can see a general downward trend at a low number of frames predicted, showing that human perception of the prediction errors can be felt. However, the data has anomalies and also does not highlight the impact of the visibility buffer size on the accuracy of the prediction. There is room for improvement in finding a better metric to measure the perception of error arising from the misalignment of the visibility buffer, but to the best of my knowledge VMAF is already the most suitable video quality metric (VQM) for our use case.



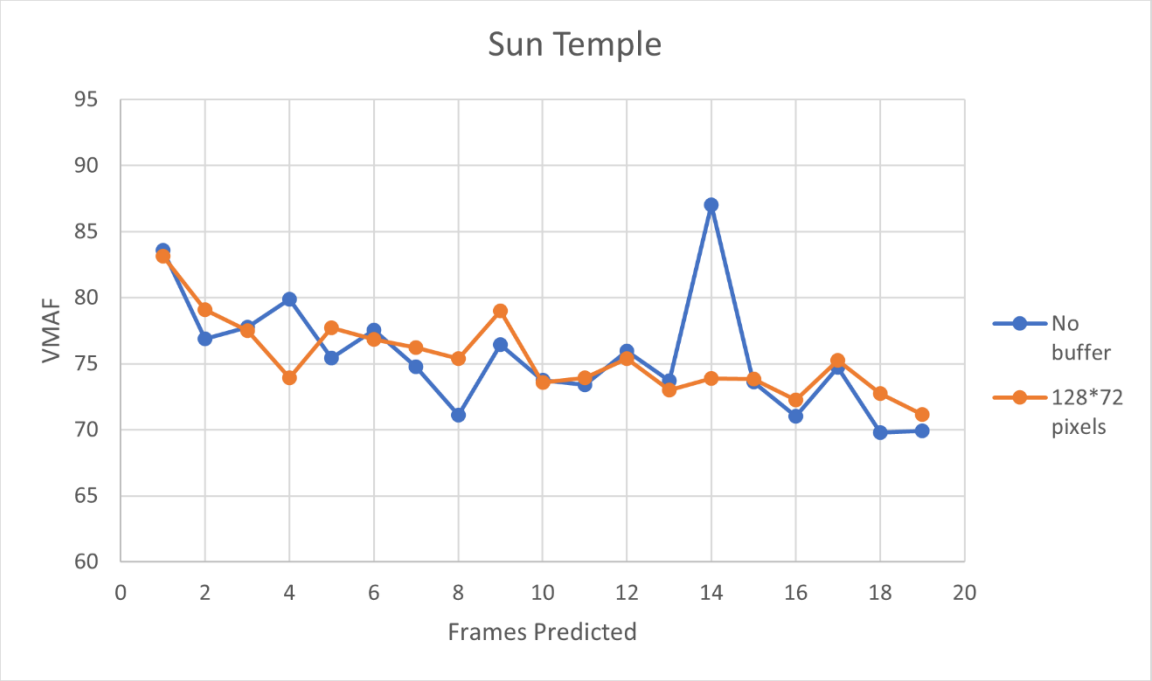


Figure 12 VMAF of videos rendered using varying numbers of predicted frames against reference video with no prediction at fixed frame rate of 30 FPS.

One possible reason for the VMAF values not being consistently in the downward direction is that VMAF is mostly trained to identify and evaluate effects of artefacts and blurriness arising from lossy encoding. While the artefacts caused by the prediction error may resemble such artefacts when small, VMAF may be untrained to accurately evaluate the video when prediction error becomes significant and the image significantly distorted. This would explain why the anomaly data points become more common when we are predicting more frames ahead. Regardless, even in the anomaly data points in Pink Room, we can observe a downward trend.

4.2 Performance

4.2.1 Methodology

A single server and client were set up on the lab computers in the Visually Immersive Studio, in COM 1, NUS via remote access. Frames rates for both scenes as frames ahead predicted as well as size of the prediction buffer were varied were captured on the client. We also measure the unique visibility buffers per second (vbps) that the server sends out. The vbps has a max value equal to the client fps since the server will not render new visibility buffers unless client

sends new camera data, but can be less than fps if client is faster than the server. The fps, together with the vbps help us to identify whether the server or client is bottlenecking.

Additional note: Varying number of frames ahead predicted did not seem to have an impact on the frame rates or visibility buffer rates of the client, and that axis has been omitted as well.

4.2.2 Results

The results in the table are as expected as they show that frame rate changes with the geometric complexity of the scene (Sun Temple is more complex than Pink Room) number of extra pixels in the visibility buffer. In general, we achieve interactive frame rates for our scene setups even when rendering large visibility buffers.

Size of prediction buffer	Pink Room (fps)	Pink Room (ms per frame)	Sun Temple (fps)	Sun Temple (time per frame)
0	154	6.49	133	7.52
64 × 36	148	6.76	127	7.87
128 × 72	142	7.04	125	8.00

Figure 13 Frame rates of client across scenes and buffer sizes.

Overall, the fps of the system has improved greatly since a year ago when we were still using the old framework with TCP and no compression. Then, fps of the system was 7.7 fps on Pink Room, which was increased to 87.1 fps by the mid report (November 2021) and now currently 154 fps.

Size of prediction buffer	Pink Room (vbps)	Pink Room (ms per frame)	Sun Temple (vbps)	Sun Temple (time per frame)
0	154	6.49	80	12.49
64 × 36	148	6.76	77	12.98
128 × 72	142	7.04	74	13.51

Figure 14 Visibility buffer per second rates of server across scenes and buffer sizes.

On this single client set up, unique visibility buffers per second sent by the server were exactly equal to the frame rate of the client in the Pink Room where there are only 3 lights, but in Sun Temple where there are 8 lights the ray tracing dependent step to generate visibility buffer takes longer and unique visibility buffers sent per second is less than client fps. This bottleneck of the server is not a big issue because the current set up for the client is very powerful and does not fully represent a thin device in the real world for which this setup is meant. In Section 4.4, we further explore the vbps and fps in two client one server set ups.

4.3 Latency

4.3.1 Methodology

To test for the response times as frames ahead predicted varied in different network conditions, we put the system through different network latencies. To do this, network condition simulator *clumsy* was used to simulate network latency in the system, since the lab computers used for testing had a 1ms RTT. We also fixed the network latency at 100ms and varied the size of the prediction buffer.

4.3.2 Results

On the first graph, the response time for different network ping values corresponding to Google Cloud regions from Singapore according to (https://www.gcping.com) is shown, including the ping for a 5G edge node which is ideal in terms of the number of frames predicted for our DHR setup. Under different network conditions, the impact of predicting more frames ahead remains consistent, with each additional frame ahead predicted corresponding to a reduction in the response time by approximately the amount taken for one frame.

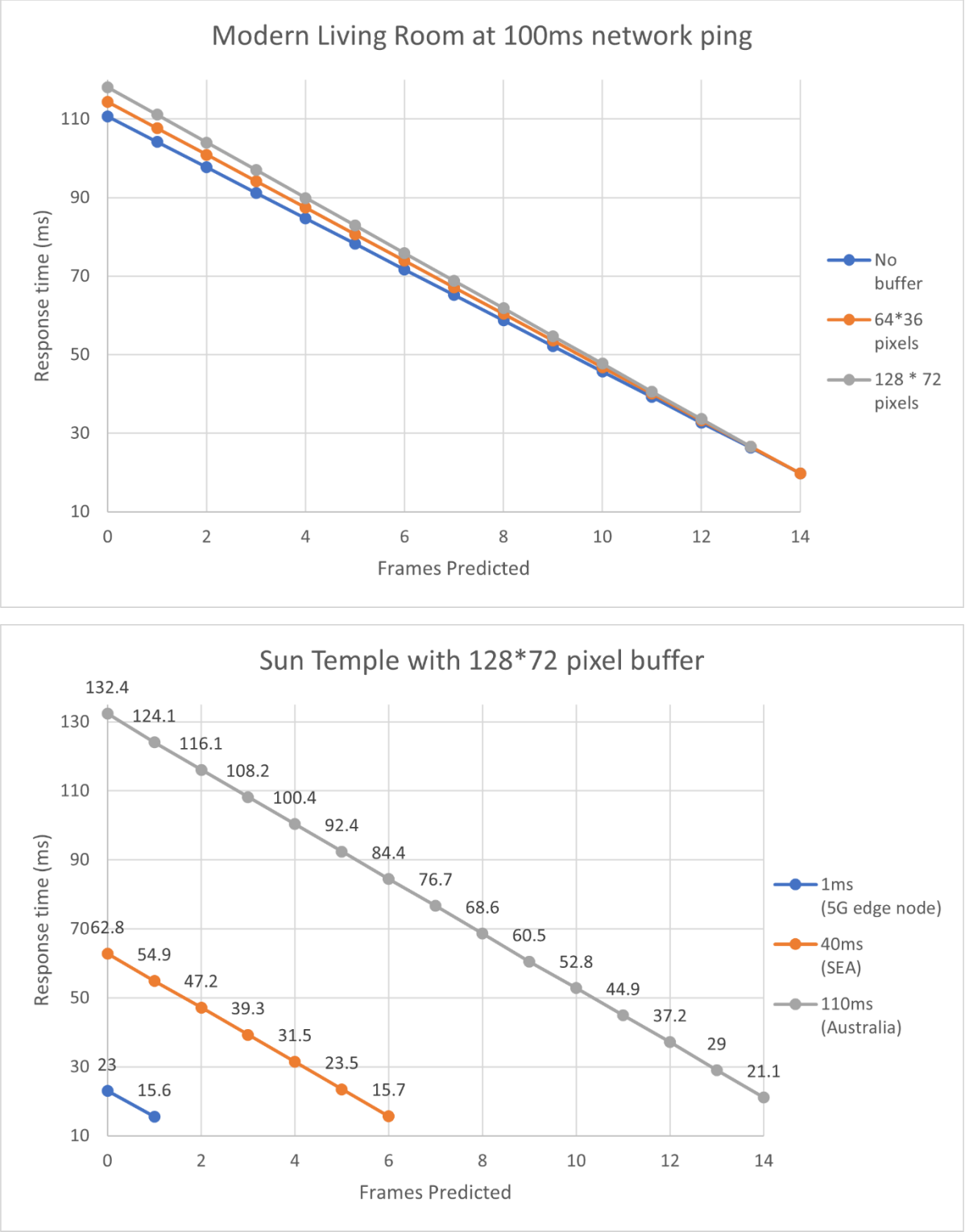


Figure 15 Graphs of response time against no. of frames predicted.

As for the second graph, we fix the network ping and vary the number of extra pixels. Initially at no frames predicted, the response time is higher with a larger buffer as it takes slightly longer for the server to transfer between device and host memory, render, compress and send the texture. Response time then decreases as expected with more frames predicted. We also note

that response time experienced when using different numbers of extra pixels converges as we predict more frames since each frame predicted reduces the response time more due to a lower FPS or longer time to render each frame.

4.4 Multiple clients

In Section 3.7 we introduced the capability of the server to accept multiple clients. Here we test fps and vbps in two client, one server set ups. Consistent with the results in Section 4.2.2, the vbps remains at the same values in the single client system in Sun Temple, since it is the server that is taking the longer time to render the visibility buffer.

However, in Pink Room, we see that with two clients, the vbps of the server is improved to 213 fps (no buffer). This is likely because the Pink Room scene with 3 lights was much easier for the server, which had spare capacity in the Pink Room scene.

Size of prediction buffer	Pink Room (vbps)	Pink Room (ms per frame)	Sun Temple (vbps)	Sun Temple (time per frame)
0	213	4.70	80	12.49
64 × 36	204	4.91	77	12.98
128 × 72	196	5.09	74	13.51

Figure 16 Visibility buffer per second rates of server across scenes and buffer sizes in a two-client setting

Chapter 5

Conclusion & Future Work

5.1 Summary

Section 4 showed the results of applying the three main changes mentioned in Sections 3.2 through 3.4, with an included short section on multiple clients and visibility buffers per second in Section 4.4. We were able to balance the three resources of performance, response time, and visual quality.

From my own observations, I found that while prediction error results suggest a linear relationship between the number of frames predicted and the error, our human perception suggests more a exponential relationship with the number of frames predicted. Visual quality is very good at less than 4-5 frames of prediction but quickly becomes very poor past that. Given for example a target FPS of 90 and a target response time of 20ms, I found that 4-5 frames of prediction looks the best and allows us to support up to 20 + 11.1 * 4.5 = 70ms of network ping while maintaining very good visual clarity.

5.2 Limitations

Measuring perception of the prediction error can be improved with a better metric, or existing video quality metrics (VQM) modified to be able to better capture its artefacts. As the scene complexity or rendering pipeline complexity grows, the measurement technique for bitwise error might have to be improved as it is computationally intensive. It is currently conducted on the CPU on a separate thread every frame. It is also difficult to test VMAF in a server client setting since the video expects every frame to be the same for both videos. Ie. lag or ping spikes would cause the frames to be different, which means the VMAF reading would not be accurate.

The client hardware used for testing included an Intel i7 7700K and RTX 2080. This may not be the most appropriate hardware given we are trying to target thin clients.

In particular, our rendering loop involves transferring the visibility buffer from host to device memory in the client and device to host in the server. Memory architectures is very differently in dedicated GPUs such as RTX series cards and GPUs in systems on a chip (SoC) that could be found in thin clients. Big dedicated GPUs are able to move huge amounts of data at high speeds, while SoCs have much smaller bandwidth for power consumption reasons. This may mean our solution may not work well in a real world setting with thin clients.

5.3 Future Work

The current rendering pipeline is still quite simple involving only one ray tracing dependent shadows computation step and hence the performance of our system is still good. It would be important to see how the hybrid setup performs even for more complex rendering.

Currently, the value x for number of frames ahead predicted is manually set. It would be interesting to be able to dynamically set the x value at run time based on some group of metrics . This would be similar to the automatic switching originally implemented in the old framework discussed in Section 3.5. This would allow us to best balance the three resources at runtime no matter what scene is being run or the network condition.

I think the multiple client system is a good idea that could be explored more. There could also be savings if clients are in the same scene, the simplest being that textures do not have to be loaded and reloaded on the server side. The server also only has the load the textures once for all the clients. This makes sense in metaverse applications, since multiple users could very likely be in the same "room" or scene (eg. Having a meeting or meetup), and so many of the resources are the same for each user.

Performing rasterization on the client while ray tracing on the server in parallel can give an overall improvement in performance as compared to pure remote rendering. It would be valuable to test the two approaches side by side to find if hybrid rendering is able to keep up with, or maybe even outperform full rendering.

References

[1] Amazon. 2021. Luna. https://www.amazon.com/luna/

[2] Ebrahim Babaei, Mahmoud Reza Hashemi, and Shervin Shirmohammadi. 2017. A State-Based Game Attention Model for Cloud Gaming. In Proceedings of the 15th Annual Workshop on Network and Systems Support for Games (Taipei, Taiwan) (NetGames '17). IEEE Press, 34–36.

[3] Colin Barré-Brisebois, Henrik Halén, Graham Wihlidal, Andrew Lauritzen, Jasper Bekkers, Tomasz Stachowiak, and Johan Andersson. 2019. Hybrid Rendering for Real-Time Ray Tracing. In Ray Tracing Gems, Eric Haines and Tomas Akenine-Möller (Eds.). Apress, Chapter 25. http://raytracinggems.com.

[4] Steve Beck, A. C. Bernstein, Daniel Danch, and Bernd Fröhlich. 1981. CPU-GPU Hybrid Real Time Ray Tracing Framework, Vol. 0. The Eurographics Association and Blackwell Publishing Ltd., 1–8. https://api.semanticscholar.org/CorpusID: 18758412

[5] João Pedro Guerreiro Cabeleira. 2010. Combining Rasterization and Ray Tracing Techniques to Approximate Global Illumination in Real-Time. Master's thesis. Portugal. http://voltaico.net/files/article.pdf

[6] Kar-Long Chan, K. Ichikawa, Yasuhiro Watashiba, P. Uthayopas, and Hajimu Iida. 2017. A Hybrid-Streaming Method for Cloud Gaming: To Improve the Graphics Quality delivered on Highly Accessible Game Contents. Int. J. Serious Games 4 (2017). https://doi.org/10.17083/ijsg.v4i2.163

[7] De-Yu Chen and Magda El-Zarki. 2019. A Framework for Adaptive Residual Streaming for Single-Player Cloud Gaming. ACM Trans. Multimedia Comput. Commun. Appl. 15, 2s, Article 66 (July 2019), 23 pages. https://doi.org/10.1145/3336498

[8] Seong-Ping Chuah and Ngai-Man Cheung. 2014. Layered Coding for Mobile Cloud Gaming. Association for Computing Machinery, New York, NY, USA, 1–6. https://doi-org.libproxy1.nus.edu.sg/10.1145/2577387.2577395

[9] Seong-Ping Chuah, Ngai-Man Cheung, and Chau Yuen. 2016. Low Bit-Rate Mobile Cloud Gaming. In Proceedings of the 3rd Workshop on Mobile Gaming (Singapore, Singapore) (MobiGames '16). Association for Computing Machinery, New York, NY, USA, 17–22. https://doi.org/10.1145/2934646.2934649

[10] Eduardo Cuervo, Alec Wolman, Landon P. Cox, Kiron Lebeck, Ali Razeen, Ste-fan Saroiu, and Madanlal Musuvathi. 2015. Kahawai: High-Quality Mobile Gaming Using GPU Offload. In Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (Florence, Italy) (MobiSys '15). Association for Computing Machinery, New York, NY, USA, 121–135. https://doi.org/10.1145/2742647.2742657

[11] LZ4. Extremely fast compression algorithm. https://github.com/lz4/lz4

[12] DHR: Distributed Hybrid Rendering using 5G Edge Computing for Thin-Client Games and Virtual Reality Experiences

[13] P. Eisert and P. Fechteler. 2008. Low delay streaming of computer graphics. In 2008 15th IEEE International Conference on Image Processing. 2704–2707. https://doi.org/10.1109/ICIP.2008.4712352

[14] Epic Games. 2017. Unreal Engine Sun Temple, Open Research Content Archive (ORCA). https://developer.nvidia.com/ue4-sun-temple

[15] Google. 2021. Stadia. https://stadia.google.com/

[16] Stefan Hertel, Kai Hormann, and Rüdiger Westermann. 2009. A Hybrid GPU Rendering Pipeline for Alias-Free Hard Shadows. In Eurographics 2009 Areas Papers, D. Ebert and J. Krüger (Eds.). München, Germany, 59–66.

[17] O. Holthe, O. Mogstad, and L. A. Ronningen. 2009. Geelix LiveGames: Remote Playing of Video Games. In 2009 6th IEEE Consumer Communications and Networking Conference. 1–2. https://doi.org/10.1109/CCNC.2009.4784713

[18] Gazi Karam Illahi, Thomas Van Gemert, Matti Siekkinen, Enrico Masala, Antti Oulasvirta, and Antti Ylä-Jääski. 2020. Cloud Gaming with Foveated Video Encoding. ACM Trans. Multimedia Comput. Commun. Appl.16, 1, Article 7 (Feb.2020), 24 pages. https://doi.org/10.1145/3369110

[19] Sanjeev J. Koppal. 2014. Lambertian Reflectance. Springer US, Boston, MA, 441–443. https://doi.org/10.1007/978-0-387-31439-6_534

[20] Christian Lauterbach and Dinesh Manocha. 2009. Fast Hard and Soft Shadow Generation on Complex Models Using Selective Ray Tracing. Technical Report TR09-004. UNC CS.

[21] Daniel Valente De Macedo, Ygor Rebouças Serpa, and Maria Andréia Formico Rodrigues. 2018. Fast and Realistic Reflections Using Screen Space and GPU Ray Tracing—A Case Study on Rigid and Deformable Body Simulations. Comput. Entertain. 16, 4, Article 5 (Nov. 2018), 18 pages. https://doi.org/10.1145/3276324

[22] Adam Marrs, Josef Spjut, Holger Gruen, Rahul Sathe, and Morgan McGuire. 2018. Adaptive Temporal Antialiasing. In Proceedings of the Conference on High-Performance Graphics (Vancouver, British Columbia, Canada) (HPG '18). ACM, New York, NY, USA, Article 1, 4 pages. https://doi.org/10.1145/3231578.3231579

[23] NVIDIA. 2021. GeForce NOW. https://www.nvidia.com/en-us/geforce-now/

[24] Wig42. 2014. The Modern Living Room. https://www.blendswap.com/blends/view/75692

[25] Chris Wyman. 2018. Introduction to DirectX Raytracing. In ACM SIGGRAPH 2018 Courses (Vancouver, Canada) (SIGGRAPH '18). http://cwyman.org/code/dxrTutors/dxr_tutors.md.html

[26] NVIDIA. Falcor 5.1. https://github.com/NVIDIAGameWorks/Falcor