RenderFusion: Balancing Local and Remote Rendering for Interactive 3D Scenes

Edward Lu[*]
Carnegie Mellon University

Sagar Bharadwaj[*]
Carnegie Mellon University

Mallesham Dasari[*]
Carnegie Mellon University

Connor Smith[†]
NVIDIA
Previously: Magic Leap, Inc.

Srinivasan Seshan[*]
Carnegie Mellon University

Anthony Rowe[*]
Carnegie Mellon University
Bosch Research

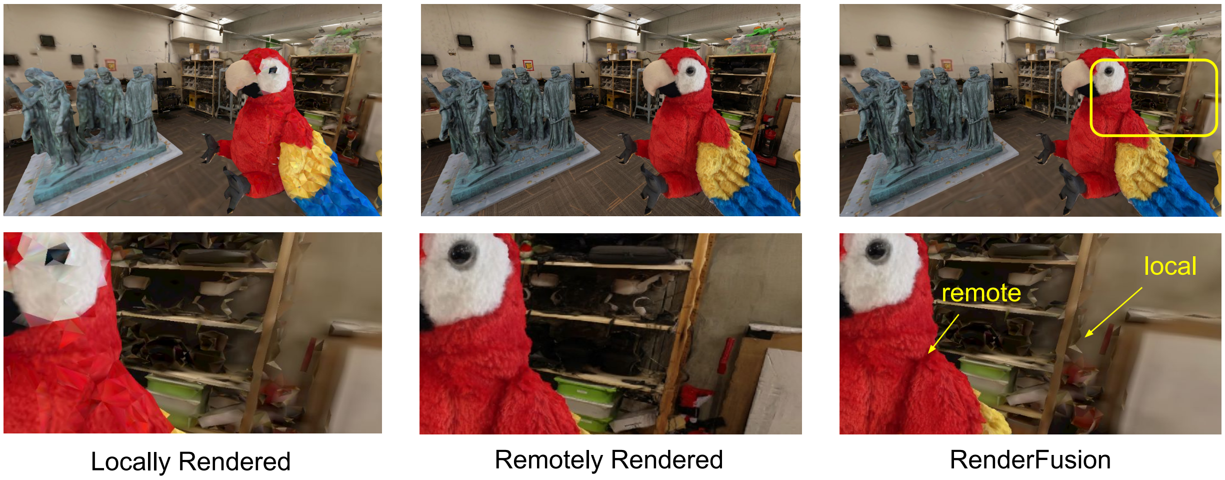


Figure 1: Rendering quality of local and remote rendering compared to RenderFusion. In this particular scene and viewport, RenderFusion selects the high-complexity background object to be locally rendered at a lower resolution and the statue and parrot models to be rendered remotely at a high resolution.

Abstract

Many modern-day XR devices (e.g. mobile headsets, phones, etc.) lack the computing resources required to render complex 3D scenes in real-time. Typically, to render a high-resolution scene on a lightweight XR device, 3D designers arduously decimate and fine-tune the objects. As an alternative, remote rendering systems can utilize powerful nearby servers to stream rendering results to a client. While this is a promising solution, it can introduce a variety of latency and reliability issues, especially under variable network conditions. In this paper, we present a distributed rendering system that combines both remote rendering and on-device, "local" rendering to add robustness to network fluctuations and device workloads. To maximize user QoE, our approach dynamically swaps an object's rendering medium, adjusting for client workload, low frame rates, and several perceptual characteristics. To model these characteristics, we perform a study under simulated conditions to measure how users perceive latency and complexity differences between objects in a scene. Using the results of the study, we then provide an algorithm for choosing the optimal object rendering medium, based on rendering complexity as well as network and latency models, ensuring that a target frame rate will be met. Finally, we evaluate this algorithm on a prototype implementation that can provide cross-platform split rendering using web technologies.

Index Terms: Computer systems organization—Architectures—Distributed architectures; Computing methodologies—Computer graphics—Graphics systems and interfaces—Mixed / augmented reality

1 Introduction

Real-time rendering systems aim to maximize user experience by balancing image quality, frame rate, and latency. On compute and memory-constrained devices, such as mobile phones and mixed reality headsets, rendering an entire 3D scene "locally" on the device itself provides low latency interactions at the cost of image quality. In contrast, remote rendering systems can leverage high-performance nearby servers to produce high-quality images but suffer the latency of transferring these images to display devices. Especially in immersive applications, motion-to-photon latency is critical for maintaining a good Quality-of-Experience (QoE). Techniques like post-rendering warping can help mask latencies, but it is often difficult to guarantee performance in remote rendering systems given the variability of wireless networks. Rather than rendering a 3D scene entirely remotely or locally, we believe that an optimal system combines *both*.

While there has been previous work exploring mixing local and remote rendering (sometimes called "split" rendering) [18, 20, 23, 26, 37], many of these solutions lack robustness to network fluctuations and/or large changes in device workloads when rendering complex objects locally. In some cases, rendering even a single high-resolution object on a device with limited memory and processing power can slow down the responsiveness of the entire system, leading to delayed user interactions. This is especially true for applications such as viewing complex medical scans, architectural diagrams, manufacturing parts, and volumetric videos, to name a few. Specifically, for these use cases, there is a need for an XR

[*]e-mail: {elu2, skalasib, malleshd, srini, agr}@andrew.cmu.edu
[†]e-mail: cosmith@nvidia.com

rendering system that can provide graceful degradation, ensuring that the application can continue to run and feel responsive despite intense rendering demands. To keep XR devices lightweight and efficient, the rendering of this kind of content *must* be offloaded to a powerful computer in a network-adaptive manner.

In this paper, we propose *RenderFusion*, a system that dynamically selects objects within a 3D scene to be rendered either on the local client or on a remote server based on network conditions and client compute capacity. Our technique renders certain objects on the headset at various resolutions, while remotely rendering other objects with higher fidelity, boosting QoE at a fixed frame rate.

Since many choices that impact QoE are subjective, we conduct a user study to generalize how people respond to post-rendering warping of remote-rendered frames (with reprojection artifacts as a side-effect), as well as the impact of interaction latency in split-rendered environments. To model client capabilities, we benchmark a few modern display devices to determine estimates for local frame rates based on scene complexity. Finally, we estimate the impact of network capacity on image quality (using SSIM) due to compression artifacts. Using these initial models, we formulate the decision of which objects to render locally or remotely as an optimization problem. The optimization models each object's importance (or "benefit") to QoE using a function that combines network link quality, geometric complexity, object interactivity, and various other perceptual characteristics. The model considers constraints such as computational budgets and device frame rates. We call this type of intelligent object swapping *dynamic scene partitioning*.

Finally, we also demonstrate a prototype open-source implementation of RenderFusion that specifically targets WebXR [44] clients for broader compatibility. We use Unity [39] as a remote rendering engine that streams to a web browser using WebRTC [46]. Our WebXR-based scene viewer is cross-platform and can operate on off-the-shelf desktop browsers, mobile phones, VR headsets (in stereo), and optical passthrough AR headsets. A Networked Scene Manager maintains a synchronized local and remote copy of an object-level scene graph with a decision-making algorithm that decides when objects should be rendered locally or remotely. The remote server transmits both color video frames as well as a depth map that the local client uses for reprojection to mask short-timescale inconsistencies. While there is room to further optimize our decision-making algorithm, we show that users prefer RenderFusion over entirely local and entirely remote rendering in a small user study exploring a live, interactable scene on both AR and VR headsets.

In summary, we make the following contributions:

- We describe RenderFusion for networked XR systems, as well as create a QoE optimization framework for partitioning rendering based on user study measurements.
- We design and conduct perceptual user studies to determine acceptable latency thresholds for remote rendering and to show user preferences towards RenderFusion when compared to other rendering systems.
- We create a reference open-source implementation able to run on stock web browsers on modern-day XR devices. We show that this dramatically simplifies creating high-quality WebXR applications with responsive interactions.

2 Related Work

Local Rendering: Most of today's 3D web applications adopt a fully local rendering approach [5, 14, 27, 34] that requires Level Of Detail (LOD) tuning to reduce the polycount of 3D models and downscale texture resolutions. This has obvious limitations in quality.

Remote Rendering: To provide high-fidelity content on compute-constrained devices, several systems [1, 2, 21, 24, 25, 29] (and [15, 40] for streaming to web browsers) have proposed using remote rendering methods that offload the entire rendering task to high-end remote servers. While remote rendering strategies have the potential for high visual quality, interactivity is often limited by high latencies.

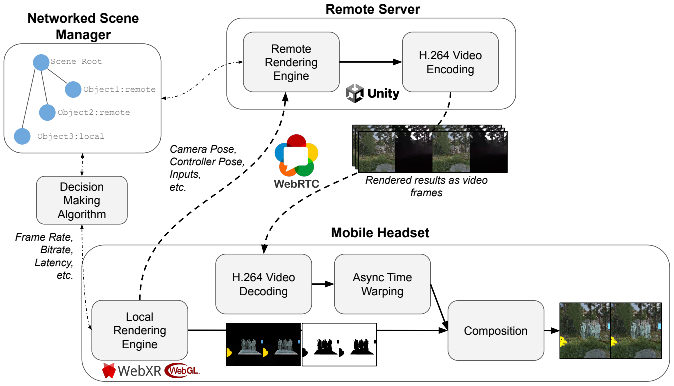


Figure 2: RenderFusion system architecture.

There has been extensive literature on reducing latency with methods that render and stream environment maps [8], panoramas [10,35], or 3D tiles [30]. These techniques often fail to accurately capture depth cues, especially for close-up objects. More recent methods use image warping techniques that leverage depth maps to generate novel view points [6, 7, 9, 36]. However, these methods struggle with dynamic scenes. When objects move, the remote server must regenerate environment or depth maps, increasing latency. Furthermore, these remote rendering strategies face constraints from network bandwidth. Insufficient bandwidth results in heavy video compression, leading to poor QoE for end users.

Split Rendering: Previous split rendering systems fall into two categories, which we call "per-object local and remote rendering," and "collaborative mixed rendering." Per-object local and remote rendering designs include Furion [20] and Coterie [26], which split a scene into either local or remote rendered objects using a static distance threshold that makes foreground objects local and background objects remote. However, this approach falters when close-up, high-polycount objects are in the foreground, causing potential memory issues or even system halts, especially in browser-based applications. Alternative solutions [18, 23, 37] use multiple versions of objects with different qualities to reduce bandwidth, but they suffer from high interaction latency, or they support only low-fidelity models to minimize latency. None of these approaches dynamically and adaptively optimize both object complexity *and* video quality.

Another broad class of techniques can be found in collaborative mixed rendering systems [13, 22, 33], that often use locally rendered objects as a reference to reproject a high-resolution remotely rendered frame to mask network latency. These solutions are unable to support highly dynamic scenes with interactable objects and animated backgrounds. Some also require sending multiple streams (more than two views) to fully cover disocclusions.

Dynamic Scene Partitioning: Funkhouser *et al.* [16] lays the foundation that we use in our system for dynamic scene partitioning, using an adaptive optimization algorithm for selecting the best LOD for each object in a scene. This was early work on balancing quality and render speed of a scene on a client device. We extend this framework to include split rendering with an objective function that now captures interaction latency and how network quality impacts streaming video. Teler *et al.* [38] also uses ideas from [16] to constrain bandwidth usage in a remote walkthrough scene.

3 System Overview

While the techniques presented in this paper can be applied to any pair of local and remote rendering engines, we specifically target stock web browsers (especially those with WebXR support) for broader accessibility of RenderFusion to a wide variety of existing

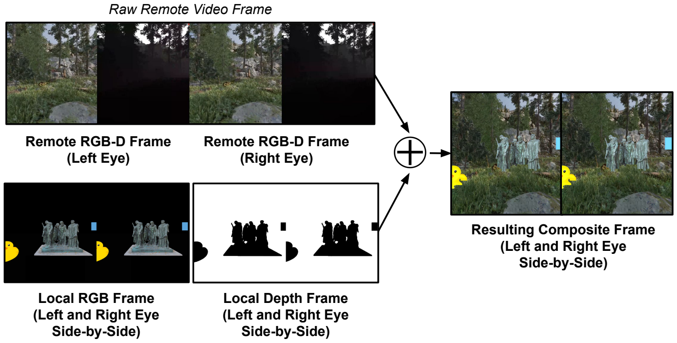

Raw Remote Video Frame
Remote RGB-D Frame (Left Eye)
Remote RGB-D Frame (Right Eye)
Local RGB Frame (Left and Right Eye Side-by-Side)
Local Depth Frame (Left and Right Eye Side-by-Side)
Resulting Composite Frame (Left and Right Eye Side-by-Side)

Figure 3: Example depth composition for stereoscopic devices.

mobile devices, ranging from smartphones to tablets, desktops, and headsets. Figure 2 shows the system architecture and workflow. At its core, RenderFusion requires at least two machines:
(1) An XR device that acts as the remote rendering *client*, receiving video streams from a server. It runs a web browser using WebGL [19] for 3D graphics, which can have limited memory and heavy overhead when calling GPU functions.
(2) A remote rendering *server* that renders high-resolution 3D content. Our remote rendering server was built using Unity and we leverage Unity's WebRTC package [41] to stream rendered results as video frames directly to web browsers. Note that while WebRTC is a popular streaming technology, it is not ideal for applications like VR that require low latency.

Both local and remote machines are synchronized with a *networked scene*, where each renderer has a copy of a meta-representation of every scene object. Individual object properties can be updated in a distributed manner, allowing the decision-making algorithm to signal both machines which objects to render/not render. In our implementation, we use ARENA [32], which uses a PubSub messaging broker to send object property updates across clients.

3.1 Operational Overview

RenderFusion cycles through three main phases of operation:

Dynamic Scene Partitioning: This phase consists of an algorithm that determines where each object should be rendered to maximize user QoE. Ideally, while a user is traversing a 3D scene, objects automatically change between local and remote rendering to optimize the user experience. See Section 4 for our QoE model. We combine this model with metrics that can be extracted from the browser, such as frame rate, WebRTC bitrate, latency, etc., to determine local and remote object assignments.

Rendering: In this phase, the local client and remote server both produce images. The remote rendering server streams video to the client using H.264 video compression. In scenes constructed for our user studies, the lighting in both the WebGL scene and the Unity scene were roughly matched. It is quite simple for Unity to render WebGL content in a manner that can be easily composited. However, it is difficult for WebGL to render using many of the techniques and shaders available on high-end servers running Unity. We discuss this in Section 7, but care needs to be taken when mixing content from different rendering engines to make the result look cohesive.

Composition: This phase merges the results of local and remote rendering into a coherent frame to be displayed, reconciling for local-remote object occlusions and temporal discrepancies between local and remote frames. In our implementation, the remote rendering server sends both color and depth images to handle object occlusions (see Figure 3). To synchronize local and remote camera viewports, we apply post-rendering warping to reproject both incoming remote images. While post-rendering warping is still an active research area, we simplify our implementation using Asynchronous Time Warping (ATW) [43]. While ATW can mask the effect of network latency, it introduces visual discontinuities after reprojection as shown in Figure 4b. To mask these artifacts, we employ several techniques: (1) In VR, we apply a closest-color in-fill (see Figure 4c). We ran a perceptual user study to model the noticeability of this specific type of artifact across a variety of simulated network delays (see Section 5). (2) For optical AR devices (which currently have small fields of view), we simply show real-world passthrough at any gaps caused by ATW (see Figure 4f). Note that ATW artifacts are more pronounced when viewed on 2D displays compared to stereoscopic devices, where they primarily occur in the user's periphery and may even be partially blocked by the device's eyepiece, as shown in Figures 4e and 4f. (3) On 2D displays, we can fill in discontinuities with ultra-low LOD (or vertex-colored) locally rendered background models like in Figure 4d.

While it is possible to mask discontinuities by overscanning the remote frame, this is not necessarily practical. We found that we would need to overscan by *at least* 25% in all directions to effectively mask artifacts on 2D screens (albeit with slow user rotations). This would require either sacrificing resolution by cropping remote frames, or sending more than 200% of our current video size, which can increase bandwidth usage, decrease video quality, and increase composition time, as the client needs to read and operate on a much larger texture. Instead, we bias our system for higher frame rates. Nonetheless, the exploration of partial, adaptive, and/or predictive overscan techniques is left to future work.

3.2 Implementation

The RenderFusion local client delivers 3D content using the A-Frame [14] virtual reality framework and the three.js [34] library. Our implementation adds a remote rendering layer on top of the ARENA web client. We developed custom WebGL shaders for depth composition, ATW and making sure the displayed aspect ratio of the remote frame aligns with that of the local frame.

While Unity already has an open-source remote rendering library [40], we develop our own package using Unity's WebRTC library for finer-grained control over the streaming stack[1]. This enables us to dynamically synchronize WebGL and Unity camera poses, match projection matrices, embed frame identifiers into each video frame, and seamlessly integrate with ARENA libraries [4].

In order to send depth information to the client for more accurate last-stage reprojection (i.e. ATW), we linearize Unity's depth texture at each pixel, converting it into an 8-bit value that spans the camera's near and far clipping planes. This value is duplicated into three channels of an RGB video frame. Currently, we only support 8-bit depth to avoid H.264 video encoding artifacts. However, there exist other approaches that can provide better depth resolutions [17, 31].

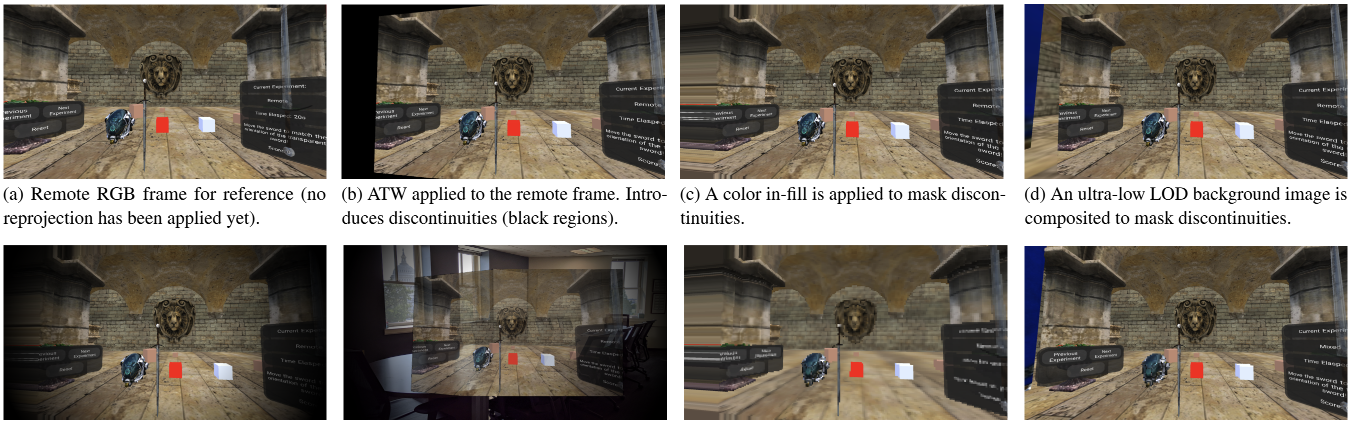
Additionally, to synchronize color and depth frames, we employ a custom post-processing shader that combines both frames into a single stitched RGB-D image to be streamed as video. For devices with a single display (e.g. desktops and mobile phones), we render RGB and depth side-by-side. For devices with stereoscopic displays (e.g. headsets), we render the stitched RGB-D frame of the left eye in the left half of the frame and the stitched RGB-D frame of the right eye on the right half (see Figure 3). Encoding stitched RGB-D using H.264 does not seem to introduce any noticeable artifacts.

4 A Model for Dynamic Scene Partitioning

In this section, we describe a decision model that allocates different parts of the scene to local and remote rendering engines such that the user *benefit* is maximized while meeting local resource usage constraints and maintaining high frame rates.

When an object is rendered locally, we can choose to render it with its full polycount or resort to a decimated low polycount variant.

[1] https://github.com/arenaxr/arena-renderfusion



(a) Remote RGB frame for reference (no reprojection has been applied yet).

(b) ATW applied to the remote frame. Introduces discontinuities (black regions).

(c) A color in-fill is applied to mask discontinuities.

(d) An ultra-low LOD background image is composited to mask discontinuities.

(e) Reprojected remote frame through eyepiece of VR headset (simulated).

(f) Reprojected remote frame through eyepiece of optical AR headset (simulated).

(g) Reprojected remote frame when network bitrate is low.

(h) Reprojected remote frame composited with local frame.

Figure 4: Results of applying ATW when remote and local cameras are not at the same pose. In all figures, all objects are rendered remotely, except for (h), which shows an outcome of RenderFusion. In (h), the background is local to reduce artifacts; helmet and blue cube are remote.

Representation	Local Resource Usage	Visual Quality	Response Latency
HL	High	High	Low
LL	Low	Low	Low
R	Very Low	High	High

Table 1: Summary of representations.

An object in a scene has three potential representations: *highpoly local* (*HL*)—the object rendered locally at its highest polygon count, *lowpoly local* (*LL*)—a geometrically decimated version of the object with a low polygon count rendered locally, and *remote* (*R*)—the object rendered on the remote server at its highest polygon count and then streamed as video frames to the device.

Each representation has its own costs and benefits. A *HL* representation of an object produces accurate renders of the object while also responding to user movements and actions with low latency. However, it consumes significant local resources; using the *HL* representation for many objects in the scene without careful consideration might result in reduced frame rates and poor user experience. On the other hand, *LL* representations afford low response latencies and consume fewer local resources. However, *LL* representations provide a lower level of detail and can affect users' visual perception of the scene. *R* provides high-quality renders and consumes minimal local resources on a per-object basis, but it comes at the cost of increased response times. Table 1 summarizes the three representations.

The task of our dynamic scene partitioning model is to choose one of the three representations for each object in the scene. We devise a decision model that maximizes the user *benefit* while constraining local resource usage and maintaining high frame rates. The dynamic scene partitioning problem is formalized as follows:

$$\max \quad \sum_{o \in O} A(o)B(o,r) \tag{1}$$

$$\text{s.t.} \quad \sum_{o \in O_L} \text{Polycount}(o) \leq \text{MaxLocalPolycount} \tag{2}$$

Let O be the set of all objects in the scene. A tuple (o,r), where $o \in O$ and $r \in \{HL, LL, R\}$ denotes an object o represented using the representation r. $B(o,r)$ is the *benefit* in user experience by representing an object o using representation r in the scene. Let O_L be the set of all objects that are rendered locally. This includes all objects whose representation is either *HL* or *LL*.

$A(o)$ is the solid angle subtended by the unoccluded parts of the object at the user's 3D coordinates. Factoring in $A(o)$ ensures that we give more importance to objects that appear larger to the user.

$Polycount(o)$ is the number of polygons in object o. *MaxLocalPolycount* is the maximum number of polygons that can be rendered locally without suffering a hit on the framerate while also considering local memory constraints. In Section 4.1, we describe how we estimate *MaxLocalPolycount*. The function $B(o,r)$ depends on several factors such as human perception of quality, latency to the remote server, bandwidth availability, etc. Section 4.2 describes the function B in detail.

We continually monitor device and network performance metrics and run the dynamic scene partitioning model every 3 seconds. We recognize that the factors that influence the decision model such as network bandwidth and latency are dynamic and can change within the interval of 3 seconds. However, we find that this cadence is sufficient for all practical purposes.

4.1 Maximum Local Polygon Count

When allocating objects to be rendered locally, we need to consider both memory and compute limitations at the local device. On lightweight devices, local memory is not large enough to store all scene objects at their highest polygon count. Similarly, local compute resources cannot render all objects at their highest polygon count at high enough frame rates. Note that both of these limitations relate to the total polygon count of the locally rendered objects in the scene. Therefore, Constraint (2) constrains the total polygon count of all locally rendered objects.

MaxLocalPolycount is chosen such that all polygons fit in memory and the local device can render *MaxLocalPolycount* number of polygons at a framerate of at least 60 FPS. Note that if the individual polycount of every object is greater than *MaxLocalPolyCount*, the optimization switches all objects to be remotely rendered. If the total polycount of all objects is less than *MaxLocalPolyCount*, the optimization will render everything locally at high resolution.

To view the effects of object polycount on device frame rate, we benchmark four devices: a Value Index tethered to an Intel NUC11BTMi7 with an RTX 3070, an M1 Macbook Pro, a Magic Leap 2 (ML2), and a Meta Quest Pro (MQP). Since WebGL uses rasterization, object size can also affect performance. To understand the relationship between object polycount, size, and frame rate, we synthetically generated objects of varying polycounts by randomizing vertex positions within a 2×2×2 unit cube. Note that this produces more of a lower bound on *MaxLocalPolyCount*, since triangle distribution here is much denser than typical 3D models, leading to more overdraw (three.js does not implement occlusion culling out the box). We placed these objects one by one in front of the camera at a distance of 50 units, moving them towards the cam-

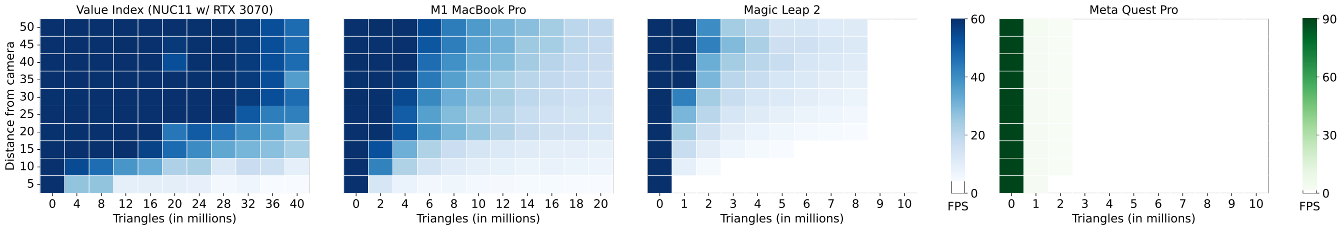


Figure 5: Effect of object polycount and distance on device frame rate. A blank/white slot denotes that the browser halted. Note that the x-axis is different across subfigures. The Quest browser implements WebXR's `updateTargetFrameRate`, so the max refresh rate was set to 90 Hz.

	$r = HL$	$r = LL$	$r = R$
Interactivity B_I	1	1	1 if latency $< l_t$, 0 otherwise. l_t is calculated using a user study.
Accuracy B$_A$	1	B_A^{LL} shown in Figure 8a.	B_A^R shown in Figure 8b.
Latency Distortion B$_L$	1	1	$\frac{1}{v}$ if latency $< l_t'$, 0 otherwise. v = length of intersection of the object with the screen edge. l_t' is calculated using a user study.

Table 2: Summary of sub-functions B_f

era by 5 units every 10s. Figure 5 shows the results. Note that the MQP's framerate falls off faster than other devices, most likely due to a need to maintain 90 FPS. Because of this, we do not perform our VR experiments on an MQP, but instead approximate the experience of viewing VR content on a mobile client using the Index.

While several other scene properties such as lighting, materials, shading, etc. can affect local device resource usage, we simplify our implementation by assuming total polygon count to be the limiting factor. However, our model can be fine-tuned to accommodate any other factor that affects rendering performance on the local device.

4.2 User Benefit Function

The benefit to user QoE by including an object o with representation r in the scene, $B(o,r)$, depends on a large number of factors and is highly dependent on user perception and the contents of the scene itself. Here, we describe some of the factors that we consider in our model along with some of our simplifying assumptions. We discover the impact of some of the factors on user experience through user studies, which are described in detail in Section 5.

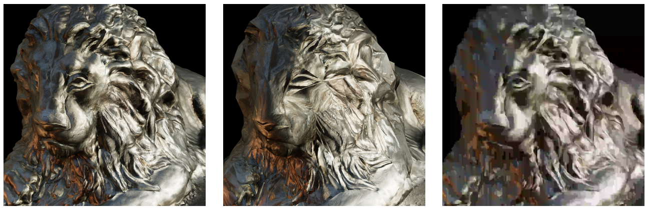
We decompose the function $B(o,r)$ into a number of sub-functions, B_i, corresponding to various influencing factors, i. We normalize each of these sub-functions between 0 and 1 and then calculate $B(o,r)$ as a product of all the sub-functions. That is, $B(o,r) = \prod_i B_i(o,r)$. In the next few subsections, we describe each influencing factor and how we estimate the effect of each influencing factor on the corresponding user benefit sub-function B_i. Table 2 summarizes the calculation of B_i's for different representations.

4.2.1 Interactivity B_I

All objects that the user directly interacts with need to have very low response latencies. Objects that are locally rendered have minimal response latency. Therefore, we set $B_I(o,HL) = 1$ and $B_I(o,LL) = 1$. The response latency of remotely rendered objects depends on the latency between the remote server and the local device. We set $B_I(o,R) = 1$ when latency is less than a threshold, l_t. Otherwise, we set $B_I(o,R) = 0$ to ensure all interactive objects are rendered locally when the remote latency is high, either using the HL or LL representation. We estimate the value of l_t from data collected from user studies (described in Section 5).

4.2.2 Accuracy of Representation B_A

B_A captures the perceptual similarity between the rendered representation of the object and the original object. Figure 6 shows the comparison between the three different representations of the object. The reduction in visual quality for the LL representation comes from geometric simplification in the LOD model. On the other hand, the R representation can sometimes suffer a loss of visual quality because of video compression artifacts.



(a) r = HL (b) r = LL (c) r = R (under low bitrate)

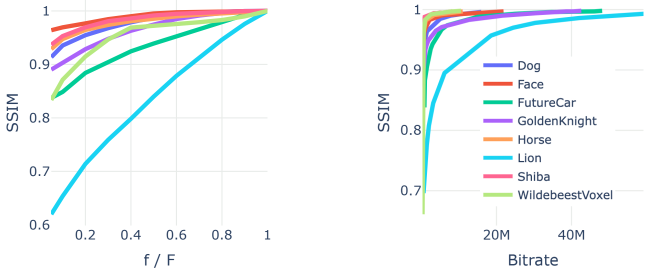
Figure 6: Visual accuracy at different representations.

The value of B_A for each representation r is calculated as follows:

$r = HL$: This is the most accurate representation of the object possible, so we set $B_A(o,HL) = 1\,\forall o \in O$.

$r = LL$: The accuracy of representation depends on the *level of detail* in the low polygon representation of the object. If f is the number of polygons in the LL representation of the object and F is the number of polygons in the HL representation, then the accuracy of representation reduces as the ratio $\frac{f}{F}$ decreases.

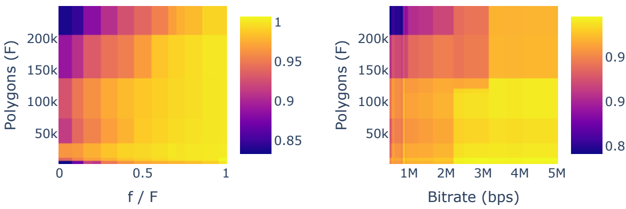
Figure 7a shows the relationship between SSIM [45], a perceptual similarity metric, and the ratio $\frac{f}{F}$. Each line in the figure represents a different object. To generate this data, we rendered multiple objects in the 3D rendering software, Blender [11]. We used the Decimate Modifier [12] in Blender to reduce the polygon count of an object from F to f. Each object was decimated with different ratios $\frac{f}{F}$. The rendered images were then compared with the full polygon count render to calculate SSIM. While SSIM increases as $\frac{f}{F}$ increases for each object, the rate of increase varies across different types of objects.



(a) Polygon count ratio vs SSIM (b) Video bitrate vs SSIM.

Figure 7: Effect of reduction in polygon count and video bitrate on perceptual similarity. Each colored line corresponds to an object.

We simplify this relationship by modeling SSIM as a bivariate function, $B_A^{LL}(f,F)$. The function predicts SSIM from the ratio $\frac{f}{F}$ and the full polygon count F by running inference on a Decision



(a) B_A^{LL}

(b) B_A^R.

Figure 8: Functions estimating B_A for low-polygon count (LL) and remote (R) representations of objects.

Tree Model [28] trained on the data collected using Blender images. The Decision Tree model has a mean absolute error of 0.013 SSIM. Figure 8a shows the value of B_A^{LL} at different values of $\frac{f}{F}$ and polygon count F.

$r = R$: When an object is remotely rendered and the resultant video is streamed, its visual quality depends on the available bandwidth between the server and client. A high bandwidth allows streaming the video at a high enough bitrate where the difference between the remote rendered and the local rendered representation of the full object is imperceptible. However, streaming the remotely rendered video at a low bitrate introduces video compression artifacts, thereby reducing the accuracy of representation.

Figure 7b shows the relationship between bitrate and SSIM for a few objects. To generate this data, we used Blender to render a 3 second video for each object. The video includes a preview of the object from all angles. The rendered video was then compressed to smaller bitrates using `ffmpeg` [3]. Each compressed video was compared against the original render to calculate SSIM. We see that SSIM increases with bitrate for each object, but the rate of increase is different across objects.

We simplify this relationship by modeling it as a bivariate function $B_A^R(b,F)$, where b is the bandwidth measured between the server and client and F is the number of polygons in the original object. We then train another Decision Tree to model the relationship between bitrate, polygon count F, and SSIM. The value of B_A^R for different bitrates and polygon counts are shown in Figure 8b.

4.2.3 Latency Distortion B_L

As described in Section 3.1, we use ATW to mask the effect of latency between the remote server and the local device. However, ATW causes smearing effects for objects on the edge of the user's view as shown in Figure 4c. The smearing effect gets worse as latency increases. Therefore, B_L depends on latency l. Furthermore, the amount of smearing that the user sees also depends on the length of the intersection of the object with the edge of the user's view, v. We define $B_L(o,R,l,v) = 1/v$ if latency $l < l_t'$. That is, at low enough latencies, one can use a remotely rendered version of the object even when it is at the edge of the screen as the smearing effect caused by ATW is acceptable to users. We proportionally reduce the benefit, B_L, by the size of the intersection of the object with the screen's edge as larger objects cause more smearing. We again estimate the value of l_t' from the study described in Section 5.

Network latency does not affect locally rendered objects, so $B_L(o,LL) = 1$ and $B_L(o,HL) = 1$.

5 Latency Threshold User Study

To get a rough idea of how additional latency introduced by remote rendering can impact user experience and to find reasonable latency thresholds for our optimization, we designed a user study to uncover user perceptual preferences at a variety of artificial latencies.

Environment: To finely control network latency, we implemented a web-based remote rendering VR simulation environment that closely resembles our runtime implementation and allows us to artificially apply any delay and quality to remote and local frames[2]. Users performed experiments on a Value Index VR headset tethered to an Intel NUC11BTMi7 running a WebXR-enabled Chrome browser on SteamVR [42]. We chose to use a tethered headset because it allows for rendering higher complexity objects, simulating high-quality remote renders. The baseline motion-to-photon latency of the Index (and the ML2) is roughly 16.6ms, which we estimate as the time it takes for a displayed frame to reflect the outcome of a controller click, assuming 60 FPS and negligible click transmission time. Due to the inherent limitations of local rendering latency, when we mention that our system has Xms latency, we are referring to Xms of *additional* latency introduced by the remote rendering.

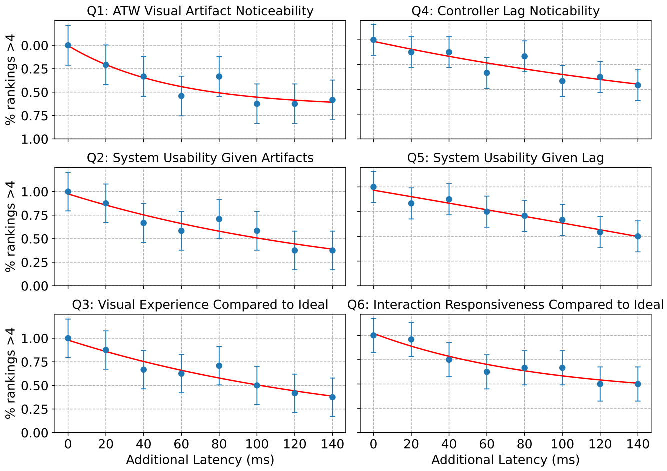


Figure 9: Psychometric functions for the percentage of positive ratings for all questions, obtained from latency threshold user study.

Participants: We recruited 24 participants (2 female, 22 male), aged between 18 and 51. Their experience ranged from mixed reality researchers to VR gamers to complete beginners. Those who had never worn a headset before were given a short primer on how to wear the headset and operate the controllers.

Experiment: Each user experienced induced latencies from 0 to 140ms at intervals of 20ms, in an order determined by a balanced Latin square. This range aligns closely with the latencies we observed in the actual system across most devices (see Figure 11). Before beginning, participants were shown the system with 0ms of additional latency and told that this trial shows the ideal system. They were allowed to perform the task any number of times as practice before beginning the trials. Including the trial at 0ms, a total of eight trials were conducted. Before each trial, the participants were asked to pay attention to "visual smearing" and controller delay. At the end of a trial, participants were asked a questionnaire related to the notability of ATW artifacts and latency, with their responses being a ranking from 1-7. These questions are listed below:

1. How noticeable was the visual artifacts on the side of the screen? (7 being very noticeable)
2. How likely would you use a system with such visual artifacts? (7 being you would definitely use again)
3. How would you rate the visual experience compared to the ideal system? (7 being nearly identical)
4. How noticeable was the lag when using the controllers to pick up and move the sword? (7 being very noticeable)
5. How likely would you use a system with such a lag? (7 being you would definitely use again)
6. How would you rate the interaction responsiveness of the system compared to the ideal system? (7 being nearly identical)

[2] `https://github.com/EdwardLu2018/renderfusion-playground`

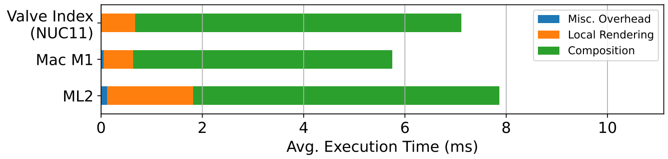


Figure 10: Execution time of RenderFusion on various platforms.

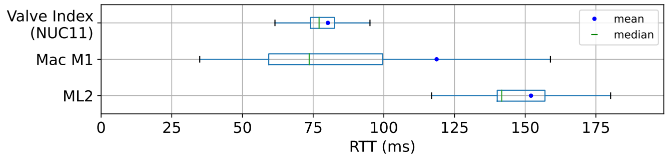


Figure 11: RTT of remote rendering on various platforms.

Task: Participants were asked to stay within our 5m×5m lab space. The VR environment is the same as that in Figure 4, with a high polycount castle as the background with a variety of low polycount objects with menu bars and instructions. Virtual 3D models of the Index controllers were placed in VR in the environment, alongside white lasers projecting from them to aim and pick up objects. The controller inputs (trigger clicks and pose changes) were artificially delayed according to the induced latency of the trial. Our simulation environment is open-sourced and we encourage future work to further explore the effects of ATW and controller delay on QoE.

During each trial, participants were asked to match the orientation of two swords: one transparent sword at a fixed target location and one draggable non-transparent sword, which would respawn randomly around the user when a match was achieved, sometimes causing participants to turn around and experience ATW smears. Our decision to use swords was deliberate; swords are thin and can be difficult to pick up with point-and-click VR controllers. Dragging the sword requires carefully aiming the laser at the object, holding the controller trigger, and moving the controller, which can become increasingly difficult to do at high speeds when latency increases and visual smearing caused by ATW becomes worse. Users were asked to repeat the task as many times as they could within 60s.

Results: Since the task is fairly repetitive, users naturally became better at the task regardless of latency, so we ignore the total number of sword placements in the discussion of the results.

Figure 9 shows the results. Note that for Q1 and Q4, lower rankings indicate more positive results, whereas for the other questions, higher rankings correspond to more positive results. Since the rankings are a function of human perception, our data was modeled as a continuous psychometric function of latency. From the plots, we see that rankings become less positive as latency increases. ATW artifacts seem to have more importance to a user's overall experience than controller delay, given the steeper decline of the fitted functions.

To assess the perceptible latency relative to 20ms, we conducted a binomial test on the relative rankings for each question. To handle ties, we include half the number of ties for both positive and negative observations. We did not compare with 0ms, since we don't expect real-world network latencies to ever be 0ms. For each question, we report the latency closest to 20ms where there was a significant difference in user rankings ($p < 0.05$). For Q1, that latency was 60ms, Q2: 100ms, Q3: 100ms, Q4: 120ms, Q5: 120ms, Q6: 120ms. We calculate the average of these latencies to determine a valid threshold for l_t' (the minimum latency until ATW smearing artifacts are noticeable) and l_t (the minimum latency until object interaction delay is noticeable). Q1, Q2, and Q3 correspond to l_t', so the cutoff is 86ms. Q4, Q5, and Q6 correspond to l_t, so the cutoff is 120ms. Again, note that these values represent *additional* latency introduced by remote rendering, and are added to the baseline local latency to form the total motion-to-photon latency for remote rendering.

6 Evaluation

The environment we used for our evaluations in this section is shown in Figure 1 and our demo video. We asked users to stay within our 5m×5m lab space. In VR, users see a virtual photogrammetry scan of the physical lab with several 3D objects of varying complexities (ranging from 12 to 20M triangles, some having multiple 4K textures). In AR, the background model is hidden, showing the real-world lab instead. When all objects are locally rendered at their *highest* resolution, this scene is unable to load on a web browser without halting on any mobile headset we tried (ML2, Quest Pro, etc.). When all objects are locally rendered at *reduced* resolution, then the scene loads and runs at 60 FPS on all devices we tried, at the cost of quality. The environment contains three objects that are draggable by clicking and aiming with a controller and one object that changes color when clicked on. The remote server was run on a Linux workstation with an RTX 4090 running an NVENC H.264 encoder for remote frames. WebRTC video size was set to 3840×1080 pixels (two stitched 1080p images for RGB and depth). The local and remote machines were connected to the same LAN. We ran our dynamic scene partitioning algorithm on the remote server, ensuring minimal overhead on the client. However, our algorithm is lightweight and can be executed on the client if necessary.

6.1 System Overhead and Round Trip Time

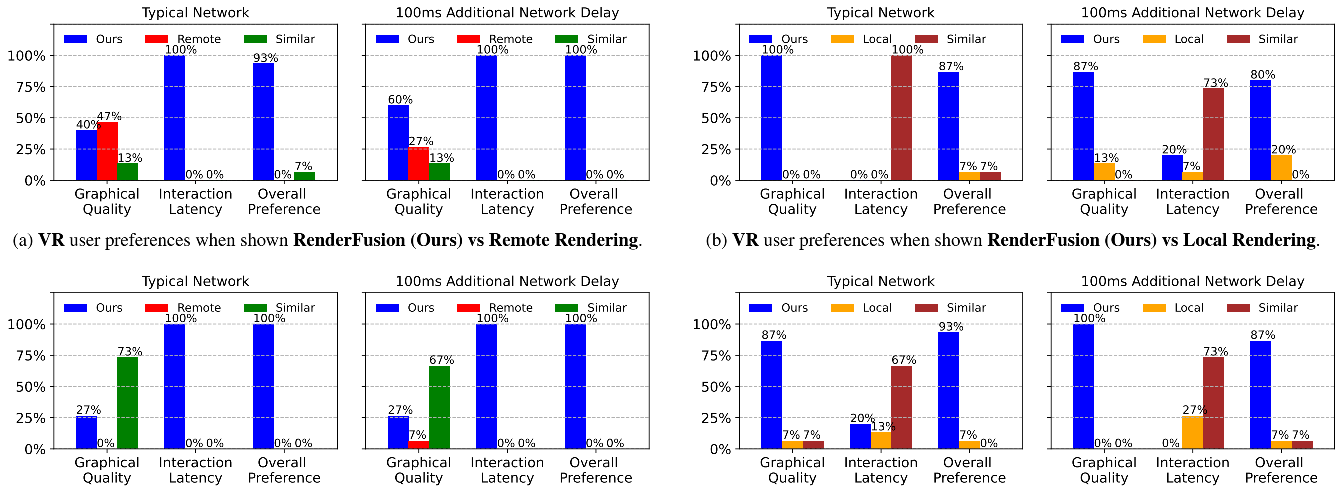
In Figure 10, we show benchmarks of rendering times on a variety of platforms used in our user studies. In RenderFusion, the client does quick calculations of $A(o)$ (the solid angle subtended by an object, used in the decision-maker), device frame rate, and network statistics, and streams this data to the remote server. In the figure, *Misc. Overhead* refers to the total execution time of these calculations. *Local Rendering* refers to the time it takes to render a local frame. *Composition* is the total time it takes to read the incoming remote frame, perform ATW, and composite it with the local frame.

We see that for this particular scene, *Composition* incurs a larger overhead than rendering the local objects, but the total execution time for all devices remains under 8.3ms. This means that despite the additional processing, RenderFusion can achieve *at most* 120 FPS on all devices we tested, though most browsers cap it at 60 FPS.

We also show the round trip time (RTT) of various devices when communicating with the remote server in Figure 11. RTT is defined as the total time it takes for a client to send the server its pose and receive the associated rendered frame. It includes network latency, server video encoding time, and client decoding time. Note we handle remote frames asynchronously as they arrive, so high RTTs do not necessarily impact frame rate.

6.2 Perceptual Evaluation

We assessed the effectiveness of RenderFusion by performing a perceptual user study with 15 participants (4 female, 11 male) aged 18 to 45 who had varying experiences with XR. We evaluated RenderFusion in both VR (using the Valve Index) and AR (using the ML2). We conducted preference tests where users compared the rendering quality and latency of our system with two alternatives: one where the rendering was done locally at a lower resolution and another where it was done remotely at a higher resolution. Users were also offered the choice to indicate that the two systems being compared were "similar" if they could not discern any differences between them. Since some state-of-the-art systems like Furion or remote IBR techniques lack open-source implementations for web browsers, our comparison was limited to pure local and remote rendering systems. To assess RenderFusion's performance in challenging network conditions, we used the Linux `tc` tool to artificially add a 100ms delay to the network. Each trial allowed participants to interact with objects, evaluating overall graphical quality and interaction latency with RenderFusion enabled and disabled. We maintained anonymity for the current rendering methods and network condition



(a) **VR** user preferences when shown **RenderFusion (Ours) vs Remote Rendering**.

(b) **VR** user preferences when shown **RenderFusion (Ours) vs Local Rendering**.

(c) **AR** user preferences when shown **RenderFusion (Ours) vs Remote Rendering**.

(d) **AR** user preferences when shown **RenderFusion (Ours) vs Local Rendering**.

Figure 12: Results of perceptual user study. Height of bars indicates the percentage of users who preferred the associated schema, or could not tell the difference between the two schemas.

being tested and exposed each user to all operating points, in an order determined by a balanced Latin square.

The VR results can be seen in Figures 12a and 12b. Notably, under typical networks, participants did not notice a difference in graphical quality in VR between remote rendering and RenderFusion. However, with 100ms delays, the impact of ATW artifacts in VR became more prominent, possibly causing users to prefer the visuals of RenderFusion's low polycount background with no smearing over a high-quality remote background with smearing. Furthermore, users perceived RenderFusion's interaction latency to be superior to that of remote rendering under both network conditions. When comparing RenderFusion to local rendering, users favored RenderFusion's rendering quality and could not tell a difference in latencies between the two rendering schemes. Lastly, the figures illustrate that the majority of VR users preferred RenderFusion over remote and local rendering overall.

The AR results are shown in Figures 12c and 12d. Participants in AR did not seem to notice any graphical differences between remote and RenderFusion, since the low-quality, virtual background is no longer visible in AR, but replaced by the real-world lab through passthrough. Similar to in VR, users preferred RenderFusion's lower latency interactions when compared to remote. Users felt that RenderFusion's quality surpassed that of local rendering, likely due to RenderFusion's higher resolution, remotely rendered foreground objects. They did not notice any latency difference between RenderFusion and local. Like in VR, users in AR exhibited an overall preference for RenderFusion compared to both remote and local.

7 Discussion and Limitations

One limitation of RenderFusion, due to being a WebXR client, is the lack of advanced lighting effects. Ideally, object swapping should appear seamless, but moving an object from a non-physically based local renderer to a physically based remote renderer, and vice versa, for instance, can be jarring. Re-lighting a local object can become complicated, depending on the polycount and number of lights in the remote scene. To avoid this, we ensured that lighting on both WebGL and Unity was approximately matched and that there were no reflective materials or shadows. As a result, this limits the rendering capabilities of the remote rendering engine. An avenue for future work could be to explore local-remote composition techniques that take into account lighting, reflections, shadows, etc. Nevertheless, it is possible to carefully design scenes where complexly-lit remote content would seamlessly blend with simply-lit local content. For instance, if all a user needs is a local UI element that controls advanced remote content, that is definitely possible with RenderFusion.

Our system also does not have a solution for highly interactive, high-resolution objects (currently, they are shifted to remote). We assume interactable objects are mostly low polycount or have a low polycount variant, but certain applications—e.g., grabbing (as opposed to just viewing) high-resolution medical imaging data—have models that cannot be low polycount forever. For these, we recommend swapping these objects to be locally rendered at a low resolution when being manipulated and then swapping them back to being remotely rendered when they are finished being manipulated.

Targeting web browsers limits us to only monitoring device metrics obtainable through JavaScript APIs. Looking ahead, we envision a native RenderFusion implementation that can monitor CPU and GPU usage, optimizing for power consumption as well.

Currently, RenderFusion can only support a single user per Unity instance since objects need to be made invisible in the scene when swapped from remote to local. With a static configuration of objects, RenderFusion is only limited in terms of clients by the rendering capacity of the server and the network bandwidth. It may be possible to run Unity in a headless mode where tens of instances can be executed per server depending on GPU requirements.

8 Conclusion

In this paper, we present RenderFusion, a system that dynamically interchanges a 3D object's rendering medium to optimize for user QoE and device frame rate. We describe models for per-object benefits towards good QoE and show that we can optimize for these benefits to intelligently determine which object should be rendered where at runtime. Additionally, we perform user studies to model the effect of ATW reprojection artifacts and controller delay on remote rendering QoE. Using our models, we implement an open-source web implementation, able to run on stock web browsers and WebXR-supported devices. We show that, when compared to traditional, locally rendered systems and fully remotely rendered systems, users prefer the mixed rendering option RenderFusion provides.

Acknowledgments

This work was supported in part by the NSF under Grant No. CNS-1956095, the NSF Graduate Research Fellowship under Grant No. DGE-2140739, and Bosch Research. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

[1] Gaminganywhere. https://gaminganywhere.org/. Accessed: June 10, 2023.

[2] Geforce now. https://www.nvidia.com/en-us/geforce-now/. Accessed: June 10, 2023.

[3] ffmpeg, 2023. Online. Accessed: April 2023.

[4] ARENA. Arena unity library. https://github.com/arenaxr/arena-unity/. Online. Accessed: March 2023.

[5] BabylonJS. Babylon.js. https://www.babylonjs.com/. Online. Accessed: June 2023.

[6] P. Bao and D. Gourlay. A framework for remote rendering of 3-d scenes on limited mobile devices. *IEEE Transactions on Multimedia*, 8(2):382–389, 2006.

[7] P. Bao, D. Gourlay, and Y. Li. Deep compression of remotely rendered views. *IEEE transactions on multimedia*, 8(3):444–456, 2006.

[8] A. Boukerche and R. W. N. Pazzi. Remote rendering and streaming of progressive panoramas for mobile devices. In *Proceedings of the 14th ACM international conference on Multimedia*, pp. 691–694, 2006.

[9] C.-F. Chang and S.-H. Ger. Enhancing 3d graphics on mobile devices by image-based rendering. In *IEEE Pacific Rim Conference on Multimedia*, pp. 1105–1111, 2002.

[10] S. E. Chen. Quicktime vr: An image-based approach to virtual environment navigation. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 29–38, 1995.

[11] B. O. Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[12] B. O. Community. Decimate modifier, 2023. Online. Accessed: April 2023.

[13] E. Cuervo, A. Wolman, L. P. Cox, K. Lebeck, A. Razeen, S. Saroiu, and M. Musuvathi. Kahawai: High-quality mobile gaming using gpu offload. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, p. 121–135. Association for Computing Machinery, New York, NY, USA, 2015. doi: 10.1145/2742647.2742657

[14] K. N. Diego Marcos, Don McCurdy. A-frame framework (v1.4.2), April 2023.

[15] Epic Games. Pixel streaming in unreal engine. https://docs.unrealengine.com/5.2/en-US/pixel-streaming-in-unreal-engine/, May 2023.

[16] T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pp. 247–254, 1993.

[17] S. N. Gunkel, R. Hindriks, K. M. E. Assal, H. M. Stokking, S. Dijkstra-Soudarissanane, F. t. Haar, and O. Niamut. Vrcomm: an end-to-end web system for real-time photorealistic social vr communication. In *Proceedings of the 12th ACM Multimedia Systems Conference*, pp. 65–79, 2021.

[18] H. Kato, T. Kobayashi, M. Sugano, and S. Naito. Split rendering of the transparent channel for cloud ar. In *2021 IEEE 23rd International Workshop on Multimedia Signal Processing (MMSP)*, pp. 1–6. IEEE, 2021.

[19] Khronos Group. Webgl 2.0 specification. https://www.khronos.org/registry/webgl/specs/latest/2.0/. Online. Accessed: April 2023.

[20] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, and N. Dai. Furion: Engineering high-quality immersive virtual reality on today's mobile devices. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, MobiCom '17, p. 409–421. Association for Computing Machinery, New York, NY, USA, 2017. doi: 10.1145/3117811.3117815

[21] F. Lamberti and A. Sanna. A streaming-based solution for remote visualization of 3d graphics on mobile devices. *IEEE transactions on visualization and computer graphics*, 13(2):247–260, 2007.

[22] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 151–165, 2015.

[23] M. Levoy. Polygon-assisted jpeg and mpeg compression of synthetic images. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pp. 21–28, 1995.

[24] Y. Lu, S. Li, and H. Shen. Virtualized screen: A third element for cloud–mobile convergence. *Ieee Multimedia*, 18(2):4–11, 2011.

[25] Magic Leap. Magic leap remote rendering. https://developer-docs.magicleap.cloud/docs/guides/remote-rendering, May 2023.

[26] J. Meng, S. Paul, and Y. C. Hu. Coterie: Exploiting frame similarity to enable high-quality multiplayer vr on commodity mobile devices. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, p. 923–937. Association for Computing Machinery, New York, NY, USA, 2020. doi: 10.1145/3373376.3378516

[27] Mozilla. Mozilla hubs. https://hubs.mozilla.com/. Online. Accessed: June 2023.

[28] A. J. Myles, R. N. Feudale, Y. Liu, N. A. Woody, and S. D. Brown. An introduction to decision tree modeling. *Journal of Chemometrics: A Journal of the Chemometrics Society*, 18(6):275–285, 2004.

[29] Y. Noimark and D. Cohen-Or. Streaming scenes to mpeg-4 video-enabled devices. *IEEE Computer Graphics and Applications*, 23(1):58–64, 2003.

[30] J. Park, P. A. Chou, and J.-N. Hwang. Volumetric media streaming for augmented reality. In *2018 IEEE Global Communications Conference (GLOBECOM)*, p. 1–6. IEEE Press, 2018. doi: 10.1109/GLOCOM.2018.8647537

[31] F. Pece, J. Kautz, and T. Weyrich. Adapting standard video codecs for depth streaming. In *EGVE/EuroVR*, pp. 59–66, 2011.

[32] N. Pereira, A. Rowe, M. W. Farb, I. Liang, E. Lu, and E. Riebling. Arena: The augmented reality edge networking architecture. In *2021 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pp. 479–488, 2021. doi: 10.1109/ISMAR52148.2021.00065

[33] B. Reinert, J. Kopf, T. Ritschel, E. Cuervo, D. Chu, and H.-P. Seidel. Proxy-guided Image-based Rendering for Mobile Devices. *Computer Graphics Forum*, 2016.

[34] Three.js Developers. Three.js Library. https://threejs.org/. Online. Accessed: May 2021.

[35] S. Shi, V. Gupta, M. Hwang, and R. Jana. Mobile vr on edge cloud: A latency-driven design. In *Proceedings of the 10th ACM Multimedia Systems Conference*, MMSys '19, p. 222–231. Association for Computing Machinery, New York, NY, USA, 2019. doi: 10.1145/3304109.3306217

[36] S. Shi, K. Nahrstedt, and R. Campbell. A real-time remote rendering system for interactive mobile graphics. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 8(3s):1–20, 2012.

[37] L. Sun, H. A. Osman, and J. Lang. A hybrid remote rendering method for mobile applications. *Multimedia Tools and Applications*, 79:3333 – 3358, 2019.

[38] E. Teler and D. Lischinski. Streaming of complex 3d scenes for remote walkthroughs. In *Computer Graphics Forum*, vol. 20, pp. 17–25. Wiley Online Library, 2001.

[39] Unity Technologies. Unity, 2005. Online. Accessed: April 2023.

[40] Unity Technologies. Unity render streaming. https://docs.unity3d.com/Packages/com.unity.renderstreaming@3.1/manual/index.html, May 2023.

[41] Unity Technologies. Webrtc for unity framework. https://docs.unity3d.com/Packages/com.unity.webrtc@3.0/manual/index.html, May 2023.

[42] Valve Corporation. Steamvr. https://store.steampowered.com/app/250820/SteamVR, 2014.

[43] J. M. P. van Waveren. The asynchronous time warp for virtual reality on consumer hardware. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, p. 37–46. Association for Computing Machinery, New York, NY, USA, 2016. doi: 10.1145/2993369.2993375

[44] W3C. Webxr device api. Online. Accessed: April 2023.

[45] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[46] WebRTC Working Group. Webrtc, 2011. Online. Accessed: April 2023.