Coterie: Exploiting Frame Similarity to Enable High-Quality Multiplayer VR on Commodity Mobile Devices

Jiayi Meng[*]
Purdue University
meng72@purdue.edu

Sibendu Paul[*]
Purdue University
paul90@purdue.edu

Y. Charlie Hu
Purdue University
ychu@purdue.edu

Abstract

In this paper, we study how to support high-quality immersive multiplayer VR on commodity mobile devices. First, we perform a scaling experiment that shows simply replicating the prior-art 2-layer distributed VR rendering architecture to multiple players cannot support more than one player due to the linear increase in network bandwidth requirement. Second, we propose to exploit the similarity of background environment (BE) frames to reduce the bandwidth needed for prefetching BE frames from the server, by caching and reusing similar frames. We find that there is often little similarly between the BE frames of even adjacent locations in the virtual world due to a "near-object" effect. We propose a novel technique that splits the rendering of BE frames between the mobile device and the server that drastically enhances the similarity of the BE frames and reduces the network load from frame caching. Evaluation of our implementation on top of Unity and Google Daydream shows our new VR framework, Coterie, reduces per-player network requirement by 10.6X-25.7X and easily supports 4 players for high-resolution VR apps on Pixel 2 over 802.11ac, with 60 FPS and under 16ms responsiveness.

CCS Concepts. • **Human-centered computing → Ubiquitous and mobile computing systems and tools**; **Ubiquitous and mobile computing design and evaluation methods**.

Keywords. Virtual Reality, multiplayer, mobile devices, frame similarity

[*]Both authors contributed equally to this research.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7102-5/20/03...$15.00
https://doi.org/10.1145/3373376.3378516

ACM Reference Format:
Jiayi Meng, Sibendu Paul, and Y. Charlie Hu. 2020. 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), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378516

1 Introduction

Virtual Reality (VR) promises applications in diverse fields such as entertainment, health care, education, and many others. According to Market Research Engine [13], the global virtual reality market is expected to exceed $33 billion by 2022 and will grow at a CAGR of more than 55% in the forecast period.

Despite its high promise, the industry has struggled with a "chicken-and-egg" dilemma, lack of content and slow market penetration of custom-made VR units. One promising direction to break the dilemma for wide adoption of VR is to enable high-resolution VR games on commodity mobile devices (*i.e.,* smartphones) over WiFi. Such an approach, however, is extremely challenging due to the limited hardware (CPU/GPU) capabilities of modern smartphones, which in turn are dictated by their battery constraint.

Furion [27] is the state-of-the-art VR system that supports high-resolution VR apps on commodity mobile devices. It splits rendering of VR contents between the mobile device and the server to meet the stringent QoE requirement, *e.g.,* 60 FPS. Furion exploits a key observation that for most VR apps, the VR content rendered can be divided into *foreground interactions* (FI) which are hard to predict but cheap enough to render on the mobile device, and *background virtual environment* (BE) which are expensive to render but fairly predictable following the player's movement and hence can be pre-rendered and prefetched from the server during game play just in time to satisfy the stringent per-frame rendering latency requirement. However, Furion was designed and demonstrated to support 4K-resolution VR games for a single player on commodity mobile devices.

Supporting multiplayer VR is posing a significant new opportunity and challenge to the VR industry [3, 8, 14]. As such, we set out to tackle the next research challenge – *Is* *it feasible to enable high resolution multiplayer VR apps on commodity mobile devices over WiFi?*

We start with a scaling experiment to gain insight into the performance bottleneck if we simply replicate the Furion architecture N-fold, one for each player, and add synchronization of foreground interactions of multiple players. Such an approach would incur little extra load on each mobile device. Our measurement of three representative multiplayer VR games on Pixel 2 phones confirms that such an approach is bottlenecked by the increased load on the wireless network, which in turn leads to almost proportional increase in the latency for prefetching each BE frame from the server and consequently the per-frame rendering latency. Our experiment suggests that the key to support multiplayer VR apps on commodity devices and WiFi is to cut down the network bandwidth requirement.

Since the frames are already compressed with the best video compression technique [27], we ask the question – can we cut down the frame transfer frequency by exploiting the similarity of BE frames in nearby locations in the virtual game world? For example, if the BE frames for the consecutive locations that a player travels through in the virtual world are sufficiently similar, we can reuse the prefetched BE frame for one location for the next adjacent locations. We denote this form of locality of BE frames as *intra-player frame similarity*. In addition, frame similarity may exist between the BE frames needed by two players as they travel through nearby locations of the virtual world, for example, in a car-racing VR game. We denote this second form of locality as *inter-player frame similarity*.

However, our measurement results from 9 representative 4K VR apps (6 outdoor and 3 indoor) show that even the BE frames of adjacent locations in the virtual game world are rarely similar enough, due to a "near-object" effect where a small displacement of near objects changes many more pixels in the rendered frame than that of faraway objects.

We propose a novel technique that overcomes the "near-object" effect by separating the part of the BE that is near the player's location from those that are far away, denoted as *near BE* and *far BE*, respectively. We show such decoupling when applied to the same 9 VR games above significantly improves both intra-player and inter-player frame similarity. Interestingly, we found that exploring inter-player similar frames on top of intra-player similar frames offers little additional benefits, even for multiplayer games with high multiplayer movement proximity, because multiple players rarely travel along exactly the same path in the virtual world.

To demonstrate the effectiveness of exploiting frame similarity in supporting high-resolution multiplayer VR apps, we design and implement Coterie, a new VR framework that renders FI and near BE on the mobile, prefetches prerendered far BE from the server, and uses a frame cache to exploit reuse of intra-player similar frames.

A key challenge in the design of Coterie is to determine the cutoff radius that separates near BE from far BE. Ideally, we should use the largest possible cutoff radius that does not cause the FI and near BE rendering time (RT) on the mobile device to exceed the latency constraint, *i.e.,* 16.7ms, to maximize far BE similarity, However, since the FI and near BE rendering time depends on the device and location (*i.e.,* the object density) in the virtual world, determining the optimal cutoff requires searching the cutoff for every location in the virtual world which is computational prohibitive (a VR virtual world can have hundreds of millions of locations).

To address the above challenge, we develop an adaptive cutoff scheme that recursively partitions the game virtual world until the cutoff radiuses within each subregion become roughly uniform. For the largest VR game we experimented with (CTS), our adaptive scheme reduces the number of cutoff calculations needed from 268 million locations in the virtual world to 235 subregions, which makes offline cutoff calculation feasible (at most a few hours).

Finally, our testbed evaluation of the Coterie implementation using three open-source high-quality VR apps from Unity ported to Google Daydream, Viking Village [12], CTS Procedural World [4] and Racing Mountain [10], shows that Coterie reduces per-player network requirement by 10.6X—25.7X and comfortably supports the QoE (60 FPS and 16ms latency) for 4K-resolution versions of the VR apps for 4 players on Pixel 2 phones over 802.11ac, while incurring acceptable resource usage (under 65% GPU usage and 40% CPU utilization), which allows the system to sustain long running of VR apps without being restricted by temperature control.

In summary, this work makes the following contributions:

- It experimentally demonstrates that the prior-art VR architecture for commodity mobile devices cannot support 4K-resolution multiplayer VR apps due to the linear increase in network load;
- It proposes to explore the similarity of BE frames and caching to reduce the network load in prefetching BE frames from the server to drastically cut down the network demand;
- It presents a novel technique for significantly enhancing the similarity of BE frame of nearby locations in the virtual world by decoupling the rendering of near BE frames and far BE frames;
- It presents the design and implementation of the Coterie VR system that demonstrates exploiting BE frame similarity allows supporting 4K-resolution multiplayer VR apps on Pixel 2 over 802.11ac.

Finally, the 10.6X-25.7X per-player network load reduction achieved by exploiting intra-player frame similarity not only enables high-resolution multiplayer VR but also significantly optimizes single-player VR on commodity mobile devices.

2 Background

In this section, we briefly review the QoE requirement of VR apps and how the prior art, Furion [27], supports 4K-resolution VR apps on mobile devices for a single player.

2.1 QoE Requirements of VR apps

Supporting VR is more challenging than supporting 360 videos (*e.g.,* [30, 31]). In addition to a headset and a renderer, a VR system also consists of a controller that receives user interactions via physical buttons and sensors which enable user interactivity with the virtual world.

While the QoE requirements for acceptable user experience in VR systems appear the same as in 360 videos, namely, (1) **Low input-to-display latency**, *i.e.,* under 10-25ms [2] motion-to-photon latency, (2) **High-quality visual effects**, *i.e.,* supporting 4K or higher resolution frames, and (3) **Mobility**, *i.e.,* the headset or the VR system should be untethered so as not to constrain user interactions [19, 20], supporting user interactions in VR via the controller makes meeting these QoE requirements more challenging. This is because unlike in 360-degree video streaming where head movement can be predicted (*e.g.,* [30]) to prefetch video frames, in VR user interactions via controllers are hardly predictable.

2.2 Supporting Single-player VR on Mobile

Supporting high-quality VR games on commodity mobile devices is challenging even for a single player due to the constrained mobile hardware and wireless technologies. We briefly review two straight-forward approaches followed by the prior art, Furion.

Local rendering (Mobile). The simplest approach to supporting VR on untethered mobile devices is to perform rendering entirely on the mobile device, which has been made available on commercial VR systems such as Google Daydream [6] and Samsung Gear VR [5]. Due to the constrained CPU/GPU capabilities of mobile devices, such systems can only support low-resolution VR apps.

Remote rendering (Thin-client). The obvious alternative approach is to avoid overloading the mobile device CPU/GPU altogether and instead offload the compute-intensive rendering load to a powerful server and wirelessly stream the rendered frames to the headset or mobile device used as the display device. However, supporting high-resolution VR apps this way would require multi-Gbps bandwidth to satisfy the per-frame latency constraint [19]. Such high data rate would exhaust the CPU of the mobile device from packet processing, *e.g.,* Furion [27] estimates processing 4Gbps would require 16 equivalent cores running at 70% utilization on a Pixel phone.

Furion. Furion [27] presents the first VR system design that meets all three QoE requirements on commodity mobile devices and WiFi, for single-player 4K-resolution VR apps.

To overcome the resource exhaustion of local rendering and remote rendering, Furion employs a split architecture that splits the rendering task between the mobile device and the server, based on the key observations that (1) for most VR apps, the VR content rendered can be divided into *foreground interactions* (FI) and *background virtual environment* (BE); (2) FI are triggered by players operating the controller or signals from other players, and hence are random and hard to predict, while BE is updated according to the user movement and thus changes continuously and is predictable; and (3) FI are much more lightweight to render compared to BE.

To enable pre-rendering and prefetching of frames, VR systems such as [23, 27] discretize the virtual world of a VR game into a finite number of grid points, so that the server just needs to pre-render frames when viewed from those grid points. With virtual world discretization, Furion achieves 4K-resolution mobile VR by

1. dividing the VR rendering workload when a player moves from grid point i to the next grid point i+1 into FI and BE,
2. leveraging the mobile GPU to render FI at grid point i,
3. leveraging the remote rendering engine on the server to pre-render and prefetch the background environment for grid point i+1,
4. decoding previously prefetched BE for grid point i on the phone, and
5. combining FI and BE for grid point i on the phone to generate the final frame.

Additionally, because after arriving at the next grid point, the player may change her head orientation which is hard to predict, Furion prefetches 4K panoramic frames (3840×2160 pixels) of BE from the server which can be cropped to render any Field-of-View (FoV) frame for background environment at almost no cost or delay.

3 Challenges in Supporting Multiplayer VR

The straightforward way of extending a single-player VR solution such as Furion to support multiplayer VR apps is to replicate the client-server architecture N-fold, one for each player. To understand how scalable such an approach is, we experimentally evaluate a prototype.

Multi-Furion. Our prototype, Multi-Furion, extends the single-player Furion to support multiple players by adding support for players (mobile devices) to exchange FI so that every player can independently render the FI of all players locally, which will then be integrated with the prefetched BE as before.

We implement exchanging FI among the mobile devices in Multi-Furion through Photon Unity Networking (PUN) [9]. PUN enables Unity object synchronization with low latency.

Using PUN, an FI object can sync the position, rotation and animation with its remote duplicates for every frame.[1]

Experimental setup. To gain insight into how well Multi-Furion as well as Mobile-only and Thin-client approaches scale in supporting multiple-player VR apps, we ported three popular high-resolution VR apps from the top three genres of Unity Store, Viking Village [12], CTS Procedural World [4] and Racing Mountain [10] (originally ported from virtual-world Unity apps), using the Google Daydream SDK.

For all three versions of each app, we embedded PUN downloaded from Unity with foreground interactions to support the multiplayer scenarios. In both Thin-client and Multi-Furion versions, the server uses H.264 (found in [27] to outperform MJPG and V9 for encoding VR frames) to encode rendered frames before transmitting them to the clients.

Our testbed (used throughout the paper) includes 4 Pixel 2 phones and a desktop server with an Intel 6-core Xeon E5-1650 CPU, NVIDIA GeForce GTX 1080 Ti graphic card, and 16GB RAM that runs Windows 10 and Unity 5.6. Using iperf, we measured the TCP download throughput from the server over 802.11ac to be around 500Mbps.

Findings. Each game is played in the same way 5 times for each version, each lasting 10 minutes. Table 1 shows the average performance of the three VR designs in supporting the three high-resolution VR apps for 1 player and for 2 players. We make the following observations. (1) The performance of the Mobile version stays about the same when running with 1 player and with 2 players in terms of FPS and frame latency and in terms of CPU/GPU load. (2) As expected, the Thin-client version suffers close to 2X increase in network transfer latency of each frame, because the server has to transfer 2 frames (to the 2 players) in each rendering interval, which inflates the already high inter-frame latency from the 41–50ms to 52–64ms range, reducing the FPS from 20 or so down to 16 or so. (3) Like Thin-client, Multi-Furion also suffers close to 2X increase in network transfer latency in prefetching BEs from the server, due to the doubled load on the network, which inflates the inter-frame latency from just meeting the 16.7ms requirement for 60 FPS to almost 21ms, which reduces the FPS down to 42–48. (4) In all experiments, the server CPU load remains under 12% (not shown) and hence is not the cause of network delay. The server GPU is also not the bottleneck in Multi-Furion which fetches pre-rendered BE frames. (5) Under Multi-Furion, the phone GPU is lightly used, around 15%. This observation is exploited later in our key idea (§4.3).

4 Exploiting Frame Similarity

Our measurement study suggests that the key to supporting multiplayer VR is to cut down the network bandwidth requirement, which scales with the number of players, per-frame size, and how often a frame is transferred from the server to each client. Since the frames are already compressed with the best video compression technique available [27], we ask the question – can we cut down the frame transfer frequency?

[1]It takes 2-3ms for each client to sync its FI with the server which are combined and retrieved by all players for rendering in the next interval.

Table 1. Performance of Mobile, Thin-Client, and Multi-Furion on Pixel 2 phones over 802.11ac.

App (players)	FPS	Inter-Frame lat. (ms)	Phone CPU load (%)	Phone GPU load (%)	Per Frame size (KB)	Net. delay (ms)
Mobile						
Viking (1P)	26	38.2	17.3	88.0	-	-
CTS (1P)	24	42.0	9.5	99.0	-	-
Racing (1P)	27	38.2	10.3	92.0	-	-
Viking (2P)	24	42.5	19.6	88.0	-	-
CTS (2P)	21	48.3	9.6	99.0	-	-
Racing (2P)	25	40.3	13.2	93.0	-	-
Thin-client						
Viking (1P)	24	41.1	25.1	7.0	586	9.7
CTS (1P)	20	50.3	24.5	9.5	590	9.9
Racing (1P)	20	50.0	21.2	10.9	680	11.3
Viking (2P)	19	52.2	25.0	8.0	586	19.8
CTS (2P)	16	59.0	30.4	9.7	590	20.1
Racing (2P)	15	64.1	21.2	13.9	680	21.2
Multi-Furion						
Viking (1P)	60	16.0	23.2	13.2	550	9.2
CTS (1P)	60	16.6	29.7	12.9	440	7.5
Racing (1P)	60	16.5	29.4	14.0	564	9.3
Viking (2P)	45	22.2	31.0	16.4	550	18.3
CTS (2P)	48	20.8	32.9	15.4	440	16.2
Racing (2P)	42	23.8	32.2	14.8	564	18.5

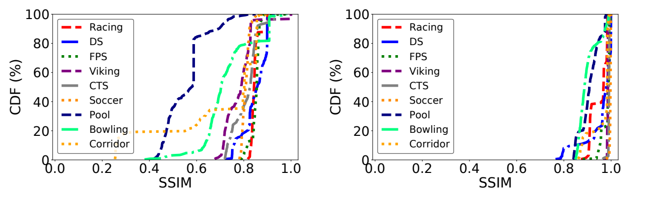
Intuitively, if the BE frames for nearby grid points in the game virtual world are similar enough, a client can reuse the prefetched BE frame for one grid point at the next adjacent grid point(s) to reduce the frequency for prefetching BE frames. The effectiveness of such an approach rests on answering two questions:

1. How similar are BE frames for nearby locations?
2. How to maximally expose the similarity among BE frames to reduce the network load in multiplayer VR apps?

4.1 How Similar are Nearby BE Frames?

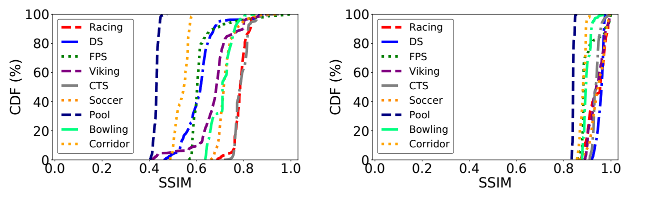
Intra-player frame similarity. We first measure the similarity between BE frames for adjacent grid points traveled by a player, for 6 outdoor and 3 indoor VR apps ported from 3D games from Unity Asset Store [11] (by adding foreground interactions), as listed in Table 2. The player plays each game (with 4K resolution) on a Pixel 2 phone for 10 minutes.

We record the player trajectory in the virtual world during game play under Multi-Furion. We then offline generate the panoramic BE frame for each grid point in the trajectory on



(a) Before decoupling (whole BE) **(b)** After decoupling (far BE)

Figure 1. Similarity of adjacent BE frames for 6 outdoor and 3 indoor VR apps before and after near BE-far BE separation.



(a) Before decoupling (whole BE) **(b)** After decoupling (far BE)

Figure 2. Best-case similarity of BE frames between 2 players for 9 VR apps before and after near BE-far BE separation.

Table 2. List of 6 outdoor and 3 indoor VR apps in our study.

Game	Genre	FI	Type
Racing	racing/chasing	racing car movement	outdoor
DS	racing/chasing	racing car movement	outdoor
Viking	competing shooting	roaming and killing enemies	outdoor
CTS	group adventure/mission	walking and jumping	outdoor
FPS	competing shooting	roaming and killing enemies	outdoor
Soccer	group adventure/mission	moving and hitting balls	outdoor
Pool	static sports	walking and hitting balls	indoor
Bowling	static sports	walking and throwing balls	indoor
Corridor	group adventure	roaming	indoor

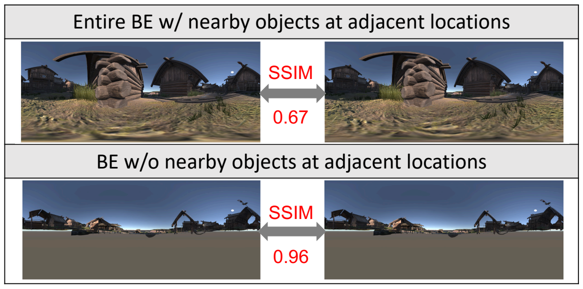

Entire BE w/ nearby objects at adjacent locations
SSIM
0.67
BE w/o nearby objects at adjacent locations
SSIM
0.96

Figure 3. "Near-object" effect degrades the frame similarity.

the server and compute the similarity between adjacent BE frames.

As in previous VR systems [23, 27], we choose Structural Similarity (SSIM) [39] as the metric to quantify the commonality between a pair of frames. SSIM is a *de facto* metric for measuring the similarity between two images by modeling the human eye's perception [23]. The human subject study in [23] concludes that an SSIM value higher than 0.90 indicates that the distorted frame well approximates the original high-quality frame and provides "good" visual quality.

Figure 1(a) plots the CDF of the similarities between each BE frame and its next adjacent frame in the trajectory. To our surprise, we see close to zero frame similarity, *i.e.,* the percentage of BE frames that exhibit an SSIM value larger than 0.90 when compared to its adjacent BE frame in the trajectory ranges between 0% and 20% for the 9 VR games.

Inter-player frame similarity. For a class of outdoor multiplayer VR apps, we observe that the multiple players typically interact closely and hence tend to be in close proximity with each other in the virtual world at any given moment through out the game play. For example, in a typical car racing game, multiple cars will chase each other closely in the same track, and in an adventure game, multiple avatars closely follow each other to survive and defeat their enemies. For such multiplayer games, a frame rendered for one player may potentially be sufficiently similar to that for a nearby player and hence reused by the other player.

Motivated by this observation, we next measure the similarity between the BE frames for two players in playing the same 9 VR apps, where two players play each game on Pixel 2 phones for 10 minutes, simultaneously.

We record the trajectories of both players during game play. We then offline generate the panoramic BE frame for each grid point in each player's trajectory on the server and compute the inter-player frame similarity as follows. For each panoramic BE frame in Player 1's trajectory, we search through all the panoramic frames rendered for Player 2 to find the one that is the most similar, based on the SSIM value, and record the similarity. We denote this similarity as *best-case similarity*, as it assumes we can always find the frame of the other player that is most similar.

Figure 2(a) plots the CDF of the best-case similarities between the two players. As with intra-player similarity, there is almost zero inter-player frame similarity, *i.e.,* the percentage of Player 1's BE frames that have best-case SSIM values larger than 0.90 is close to 0% for the 9 VR games.

4.2 The "Near-Object" Effect

To gain insight into why the BE frames for nearby locations rarely have high SSIM, we visually compared many pairs of adjacent BE frames in the single-player traces. We observed that *many BE frames contain objects (assets in Unity's terminology) near the player in the virtual world, because of which even a slight displacement of the player location can lead to visible change between the frames.*

Figure 3 shows an example where the BE frames from two nearby locations in the Viking Village game have a low SSIM value of 0.67 (top figures) but have a high SSIM value of 0.96 (bottom figures) after removing the objects near the players, *i.e.,* the viewing points.

The above "near-object" effect can be explained by the technique used by VR engines such as Unity in rendering

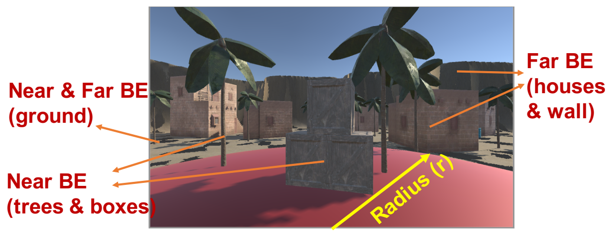


Figure 4. Separating near and far BE.

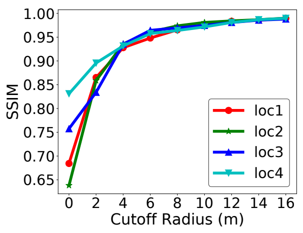


Figure 5. Adjacent far BE frame similarity vs. cutoff radius for 4 randomly sampled locations.

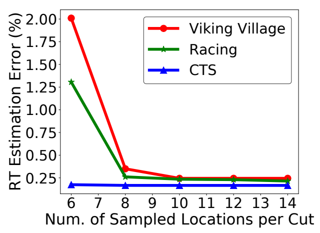


Figure 6. Percentage of randomly sampled locations whose near BE rendering time exceeds Constraint 1.

the frames, called *Perspective Projection* [38]. The technique emulates how human eyes view the world by mapping a 3D grid-point of the game world into a point in the 2D frame. Specifically, it converts faraway objects to be viewed smaller and the nearby objects to be viewed larger in order to provide realism to human eyes. As a result, a small displacement of near objects is more pronounced, *i.e.,* it changes many more pixels in the rendered frame, than that of faraway objects.

4.3 Key Idea: Exposing BE Frame Similarity

The above insight into how near-objects diminish the similarity of BE frames rendered from nearby locations in the virtual world, coupled with the observation that the GPU is underutilized in the split-rendering architecture (about 15% utilization as shown in Table 1), motivates us to design a new way of splitting the VR content to be rendered on the mobile device and the server.

Decoupling near and far BE. The "near-object" effect suggests that if we separate the part of the BE that is near the player's location from those far away, denoted *near BE* and *far BE*, respectively, the similarity of the far BE frames rendered from nearby locations in the virtual world should be significantly enhanced. Procedurally, we define *near BE* and *far BE* based on a *cutoff radius*, where the objects within and outside the radius belong to near BE and far BE, respectively, as shown in Figure 4.[2]

Where to render near and far BE? To incorporate such a separation into the split-rendering architecture design, we need to decide where to render near and far BE.

Since far BE tends to contain many more objects than near BE, especially for outdoor VR apps (see Table 2) which accounts for a majority of the VR apps in VR appstores [15–17], rendering far BE will remain computationally prohibitive for the mobile device and hence should be performed on the server (and far BE frames should be prefetched as before).

Near BE contains fewer objects than far BE and thus can potentially be rendered on either the mobile device or on the server. However, because near BE is near the viewing point, a near BE frame will occupy a significant portion of the final frame. As a result near BE frames will remain large; our results show that near BE and far BE frames are of comparable sizes, about half of the original BE frame. Thus rendering near BE on the server would not lower the network load much, and near BE should be rendered on the mobile device.

The remaining design challenge is how to determine the cutoff radius that separates near objects in the virtual world, from faraway ones. We first discuss the constraint on the cutoff radius and then present a practical scheme that avoids deciding a cutoff radius for every grid point.

Constraint on cutoff radiuses. Intuitively, the larger the cutoff radius, the further away the objects in the far BE, and the more similar the far BE when viewed from nearby locations will be. Figure 5 shows the SSIM between two adjacment far BE frames at each of four randomly chosen locations in Viking Village as we vary the near BE-far BE cutoff radius. We see that the SSIM value quickly and monotonically increases with the cutoff radius, from 0.68, 0.63, 0.75, and 0.83 at the cutoff radius of 0 (*i.e.,* the whole BE is the far BE) to above 0.9 at the cutoff radius of 4 meters. However, a larger radius will also result in higher rendering load on the mobile device since near BE will contain more objects. This tradeoff suggests that we should use the largest possible cutoff radius that does not cause the rendering time (RT) on the mobile device to exceed the latency constraint, *i.e.,* 16.7ms:

$$RT_{FI} + RT_{NearBE} \; < \; 16.7ms$$

The above constraint on the cutoff radius suggests that the right choice of cutoff is app and device dependent, as both the FI and near BE rendering time are app and device dependent. This in turn suggests that the cutoff needs to be determined for each app offline, *i.e.,* during app installation, based on measurement on the device.

Determining RT_{FI}. For each app, the rendering time for FI is determined by the nature of the game-allowed FI and generally does not depend on the location in the virtual world. Therefore, we can first experimentally determine the FI rendering time on a given device to find an upper bound.[3]

[2]It is ok for an object to be cut in the middle; its two halves will be rendered in the near BE and far BE, respectively.

[3]Typical user foreground interactions with the game, including for multi-players, can be prerecorded and replayed during installation time.

For example, for the three 4K-resolution VR apps, we measured RT_{FI} based on player game plays to be bounded well below 4ms on Pixel 2, which gives a conservative time constraint on rendering near BE on Pixel 2 in determining the cutoff radius:

$$RT_{NearBE} < 16.7ms - RT_{FI} = 16.7ms - 4ms = 12.7ms \quad (1)$$

Calculating cutoff radiuses. Since the rendering speed is correlated with the triangle count of the objects [1], we first measure the object density at a location in the game virtual world by the triangle count within a fixed cutoff radius. We observe that the object density across the virtual world of the VR games can vary significantly. This suggests that using a single cutoff radius that satisfies Constraint 1 for the entire virtual world will be inefficient as we will end up with a smaller radius than necessary for many locations. On the other hand, customizing one cutoff radius for every location in the virtual world is computationally infeasible as a VR game can easily contain up to hundreds of millions of grid points in the virtual world.

We make a key observation that though the object density can vary in different locations of the game world, it changes gradually and tends to be uniform within a small region. Building on the observation, we propose *an adaptive cutoff scheme* that drastically reduces the number of cutoff radiuses that need to be calculated. In particular, the scheme recursively partitions the game virtual world until the cutoff radiuses for different locations with each subregion become roughly uniform.

The adaptive scheme works as follows. Since the players move in 2D in the virtual world in typical VR games, the recursive partitioning of the virtual world will be 2D. The partitioning procedure is recursive, first invoked with the complete game world. At each invocation, it samples K random locations from the input game region, calculates the maximal cutoff radius for each location that satisfies Constraint 1, and checks if the radiuses for the K sampled locations are similar. If so, it records the minimal radius as the radius for the region and returns. If not, the object density in the region is likely uneven, and the procedure partitions the region into 4 equal-sized subregions and recurs on them. Effectively, the recursively partitioned subregions form a quadtree, and we denote the subregions that are not partitioned further the "leaf regions".

We experimentally determine the suitable choice of K. For each K value, we follow the above procedure to recursively partition the game world into a quadtree of leaf regions, and then experimentally measure the percentage of locations in the player trajectory traces from experiments in §4.1 that do not satisfy Constraint 1 using the derived cutoff radiuses for their leaf regions. Figure 6 shows that for the 3 VR apps, Viking Village, Racing, and CTS, if we set K to be 10, the percentage of locations in the trace whose rendering time RT_{nearBE} using the calculated cutoff violates Constraint 1 is less than 0.25%. Thus we use 10 as a suitable choice for K in our design.

In summary, our adaptive cutoff scheme adaptively generates cutoff radiuses according to the varying object density in the game world to minimize the total number of cutoff radiuses that need to be calculated for separating far/near BE frames at different locations in the game world while maximizing the cutoff radius for each leaf region and hence the locations within. The maximized cutoff radius in each leaf region in turn leads to high similarity between the far BE frames in nearby locations and hence high frame cache hit ratio (shown in §7).

Table 3. Game stats and the output and running time of the adaptive cutoff scheme.

App	Game Dimension (meter^2)	Grid Points (million)	Quadtree Depth (avg/max)	Leaf Reg.	Proc. Time (hrs)
Viking Village	187×130	24.90	5.87/6	2944	6.60
CTS	512×512	268.40	3.81/4	235	1.30
Racing Mt.	1090×1096	7.70	3.70/4	136	1.25
DS	1286×361	3.00	3.80/4	160	1.66
FPS	71×70	5.09	3.92/4	208	1.10
Soccer	104×140	14.90	3.88/4	136	1.18
Pool	10×13	0.13	2.68/3	19	0.14
Bowling	34×41	1.43	2.00/2	16	0.13
Corridor	50×30	1.54	2.80/3	40	0.29

4.4 Effectiveness of the Adaptive Cutoff Scheme

To measure the effectiveness of the adaptive cutoff scheme, we apply it to the same 6 outdoor and 3 indoor multiplayer VR games evaluated in §3 to partition the virtual world into leaf regions and find their corresponding near BE-far BE cutoff radiuses.

Table 3 summarizes the basic stats about the 9 games and the output of the adaptive cutoff scheme. The results show the effectiveness of the scheme: (1) In general, the larger the dimension of the virtual world, the deeper the quadtree of recursively partitioned leaf regions. For example, compared to the 6 outdoor games, the 3 indoor games have smaller dimensions and consequently shallower quadtrees and fewer leaf regions. (2) However, the quadtree depth of a specific game depends on the variation of object density in the game world. Viking Village has about 20× smaller game dimension than Racing and DS, but has a deeper quadtree and 10×-20× more leaf regions than them. This is because Viking Village has higher variation (non-uniformity) of object density (*i.e.,* triangle counts) in its virtual world.

Figure 7 shows the distribution of the cutoff radiuses for the leaf regions of the 9 games. We see that the cutoff radius stays in a small range for all except two outdoor apps, DS where half of the cutoff radiuses are spread between 10 and

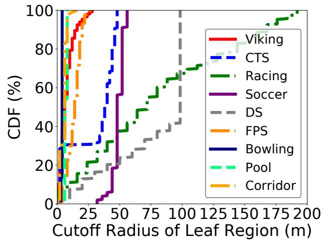


Figure 7. CDF of cutoff radiuses of the leaf regions of 9 VR games generated by the adaptive scheme.

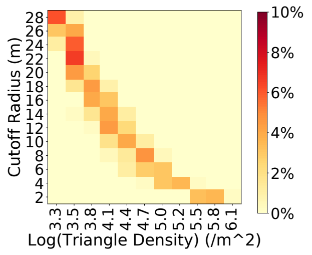


Figure 8. Cutoff radius vs. triangle density for 420 leaf regions (heatmap).

100 meters, and Racing Mountain where all cutoff radiuses are evenly spread between 10 and 180 meters. For DS, the widespread of cutoff radiuses is because the regions near start/end locations of racing are densely populated with game assets including people and big stadiums, while the rest of the vast virtual world is sparsely populated with game assets. For Racing Mountain, a few regions along the track are very close to a forest of trees while other regions are sparsely populated with few assets.

We zoom into Viking Village to see how the adaptive scheme adapts the cutoff radius with the object density. Figure 8 shows the object density (measured in triangle count per meter^2) and the corresponding cutoff radius generated for 420 leaf regions (30 per cutoff radius between 2 and 28 meters) in a heatmap. We see a clear correlation: the higher the object density, the smaller the generated cutoff radius.

In summary, our adaptive cutoff scheme is highly effective – for the largest VR game we experimented with, CTS, it reduces the number of cutoff calculations needed from 268 million locations in the virtual world to 235 subregions. The scheme makes offline cutoff radius calculation feasible, *i.e.*, at most a few hours each for the 9 games.

4.5 Frame Similarity Improvement

To measure the impact of separating near BE from far BE on frame similarity, we calculate the intra-frame and inter-frame similarities for far BE frames only for the same 9 multiplayer VR games using the cutoff radiuses generated by the adaptive cutoff scheme in §4.4.

Figure 1(b) shows that the similarity between adjacent far BE frames for a single player is drastically higher for the 6 outdoor games; the percentage of adjacent BE frame pairs that have an SSIM of over 0.9 ranges between 85% for DS-Racing to 100% for the other games. Similar improvement is observed for the 3 indoor games; the percentage ranges between 65% to 90%.

Similarly, Figure 2(b) shows that the best-case similarity between far BE frames for two players in the 6 outdoor VR apps also increases significantly: the percentage of best-case

Table 4. Five frame cache configurations.

Version	Reuse Intra-player Frames	Reuse Inter-player Frames
Version 1	✓(exact)	
Version 2		✓(exact)
Version 3	✓(similar)	
Version 4		✓(similar)
Version 5	✓(similar)	✓(similar)

Table 5. Cache hit ratio of Viking Village under 5 cache versions.

Version	1-player	2-player	3-player	4-player
Version 1	0.0%	0.0%	0.0%	0.0%
Version 2	0.0%	0.0%	0.0%	0.0%
Version 3	80.8%	80.8%	80.8%	80.8%
Version 4	0.0%	63.9%	67.2%	65.4%
Version 5	80.8%	80.4%	80.4%	87.7%

BE frame pairs that have an SSIM of over 0.9 ranges between 55% for FPS to 100% for the other games. The inter-player similarity is much lower for the 3 indoor games however; the percentage varies between 2% (Pool) to 33% (Bowling). This is expected as multiple players in these indoor games do not follow each other closely in the virtual world.

4.6 Added Benefit of Exploiting Inter-player Frame Similarity

To understand the added benefits of exploiting inter-player frame similarity on top of intra-player frame similarity, we perform the following caching experiment.

We collect the per-player movement trace for the 1, 2, 3, and 4-player testbed experiments in §7. We then replay the players' movement and emulate each player's far BE frame requests against an infinite-sized frame cache under different cache lookup schemes for each player, and assume the reply from the server is overheard and cached by all the players. Note there is no need to generate and manipulate the actual far BE frames as the cache lookup outcome is determined by the frame locations in the game (explained below).

Specifically, we compare 5 versions of cache lookup configurations as shown in Table 4. Version 1 caches frames prefetched by the local client and serves exactly matched frames. Version 2 caches frames sent to other players (*e.g.*, from "overhearing") and serves exactly matched frames. Version 3 adds on top of Version 1 serving similar frames according to cache lookup algorithm in §5.3. Version 4 adds on top of Version 2 serving similar frames as in Version 3. Finally, Version 5 combines Versions 3 and 4 to serve exact and similar frames, self-prefetched or prefetched by other players.

Table 5 shows the results for Viking Village which has the lowest inter-player movement locality and SSIM higher

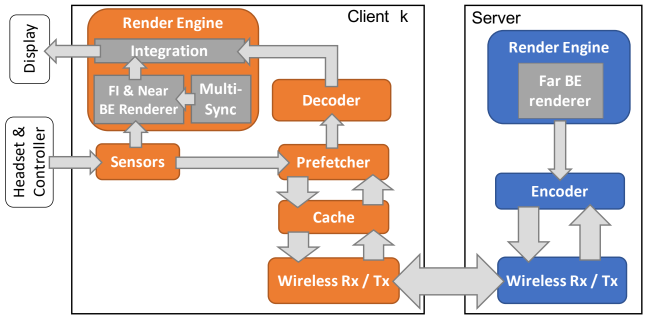


Figure 9. COTERIE architecture.

than 0.9 for 100% best-case BE frame pairs. We make the following observations. (1) Version 1 has no cache hit as the player does not traverse the exactly same path. Similarly, Version 2 has no cache hit, suggesting even for VR games with high player movement locality, the trajectories of different players rarely overlap exactly. (2) Version 3 which reuses cached frames prefetched by a client itself already achieves 80% cache hit ratio, which reduces the frame prefetching frequency by 5X. (3) Version 4 which reuses cached frames sent to other players also achieves a high cache hit ratio, 63.9%-67.2% for 2-4 players. (4) But finally, if a client already reuses self-prefetched similar frames, caching frames sent to other players does not give significant additional benefit, as shown in the almost identical hit ratios for Version 3 and Version 5, for 2-4 players. This is because multiple players usually do not follow the same identical path.

The above experiment suggests that Version 3, which reuses similar frames prefetched by a client itself, already reaps most of the benefits of frame caching. For this reason, and also because wireless overhearing (*i.e.*, the promiscuous mode) is not supported by current Android Qualcomm built-in NICs for security reasons, we do not exploit inter-player frame similarity in the final COTERIE design.

5 COTERIE Design

The goal of COTERIE design is to enable high-resolution multiplayer VR apps on commodity mobile phones and WiFi networks by reducing the network bandwidth requirement. COTERIE achieves this goal via a 3-layer rendering architecture that (1) decouples BE into near BE and far BE to expose the similarity of far BE frames, (2) renders FI and near BE on the mobile devices, and (3) caches and reuses similar far BE frames to significantly reduce BE frame prefetching from the server.

5.1 Architecture Overview

Figure 9 shows the COTERIE system architecture which consists of a server running on a desktop and multiple clients each running on a commodity phone.

COTERIE **server.** In offline processing, the server first runs the adaptive cutoff algorithm to recursively divide the virtual world into multiple leaf regions, and then pre-renders and pre-encodes (using x264 [18] with Constant Rate Factor of 25 and fastdecode tuning enabled) panoramic far BE frames for all the grid points the player can reach. During game play, the server replies to far BE frame requests from any client with encoded prerendered panoramic far BE frames in TCP.
COTERIE **client.** During each time window when a player moves from one grid point in the virtual world to the next, the client renders the frame for the next grid point by concurrently performing four tasks, followed by frame merging:

1. **FI and near BE rendering:** The hardware sensors collect the new pose from the headset and the movement and FI from the controller, which trigger the Render engine to start rendering FI (with the latest multiplayer synchronization) and near BE for the next grid point.
2. **Decoding:** The prefetched compressed far BE frame for the next grid point is read from the frame cache and sent to the upper layer, Android hardware decoder, for decoding.
3. **Prefetching and caching:** The prefetcher determines the far BE frames for the neighbors of the next grid point, and searches in the frame cache for the exact or similar frames. If all needed frames are found in the frame cache, the prefetching is skipped; otherwise, the prefetching request is sent to the server, and replied frames are cached.
4. **Synchronizing FI:** The FI of the players are synchronized via PUN (over UDP) on all mobile devices via the server; the synchronized FI will be used in the next frame rendering time slot.
5. **Merging:** The decoded far BE frame is merged with the locally rendered FI and near BE in the Render engine, which is then projected for each eye for display to the player.

The four time-critical tasks are performed in parallel. The end-to-end latency to render a new frame is:

$$T_{split\_render} = max(T_{phone\_render\_FI} + T_{phone\_render\_nearBE}, T_{phone\_decode\_farBE}, T_{phone\_prefetch\_next\_farBE}, T_{phone\_sync\_FI}) + T_{merge} \quad (2)$$

The design of tasks 1, 2, 4 and 5 are straight-forward and we discuss their implementation details in §6. Below, we discuss task 3 (prefetching and caching) in detail.

5.2 Prefetching Far BE frames

At each COTERIE client, each far BE frame prefetching request is first sent to the frame cache, and is only sent out to the server if the cache cannot find a similar frame. Because of the high similarity of adjacent far BE frames, a prefetched far BE frame for grid point *i* can often be reused for adjacent

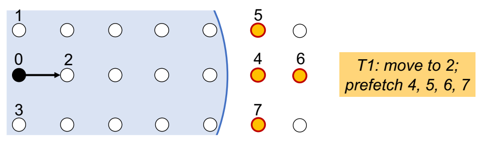


Figure 10. Prefetching far BE frames.

grid points within several hops. Figure 10 shows an example where the far BE frame at grid point 0 resides in the cache and can be reused at nearby grid points within the shaded region.

This frame reuse not only reduces the frequency of prefetching from the server, but also creates a larger time window for each prefetching. When the Coterie client is moving from point 0 towards point 2, it just needs to finish prefetching the far BE frame for point 4 and its neighbors 5, 6, 7 at any time before reaching point 4.

The large prefetching window simplifies scheduling the prefetching of multiple clients. Instead of coordinating the prefetching requests of multiple clients, *e.g.,* using TDMA, to avoid collision, we simply let each Coterie client start prefetching the next set of frames not in the cache right away after the first time reusing a cached frame, based on the current moving direction.

5.3 Frame Cache

The frame cache at each Coterie client caches far BE frames prefetched by the client. Intuitively, a cached far BE frame for a location can be reused for any sufficiently close location in the virtual world. However, since our adaptive cutoff scheme can lead to different cutoff radiuses for nearby locations, *e.g.,* those belonging to different leaf regions, the cache lookup algorithm also needs to check for additional conditions other than proximity.

Cache lookup algorithm. The cache stores relevant metadata for each cached far BE frame, such as its corresponding grid point and belonging leaf region. A cache lookup for grid point k returns a cached far BE frame as a hit if it satisfies three criteria: (1) Its grid point is within some distance threshold $dist\_thresh$ from grid point k; (2) It is in the same leaf region as grid point k, since different regions may have different cutoff radiuses (§4.3), which will cause a gap between the near BE and far BE of the two locations; (3) Its corresponding near BE contains the same set of objects as that for grid point k to ensure there are no missing parts after merging the rendered near BE with the cached similar far BE frame. The three conditions ensure a reused far BE frame will integrate with the near BE rendered smoothly. Out of all the cached frames that satisfy the above constraints, the one closest to the grid point being looked up is returned as the most similar frame.

The distance threshold $dist\_thresh$ is derived offline, one for each leaf region, by sampling K grid points in the region. For each sampled grid point l, we binary-search $dist\_thresh_l$ (*e.g.,* starting from 32 downwards) until using it, the far BE frame has high similarity (*i.e.,* SSIM > 0.9) with that of another random grid point within the radius of $dist\_thresh_l$. Finally, for each leaf region, we choose the minimum $dist\_thresh_l$ of the K sampled grid points as its distance threshold.

Cache replacement policy. For fast access, the frame cache needs to be in phone memory. Since smartphones typically come with limited memory, *e.g.,* 4GB on Pixel 2, we need an effective replacement policy to maximize the cache hit ratio. We explore two replacement policy designs. **(1) LRU**, which exploits temporal locality in access patterns; **(2) FLF** (*Furthest location first*), which exploits spatial locality by evicting the cached frame that is among the furthest from the player's current position in the virtual world.

6 Implementation

We implement Coterie in Unity and Google Daydream.

Offline preprocessing. We implement an offline preprocessing module to implement the adaptive cutoff scheme (§4.3) and determine the distance threshold used for cache lookup (§5.3). Considering the varying elevation and slope of the terrains where players stand in the virtual world, we apply ray tracing to find the foothold of the players and then adjust the height of the camera to gain the same views as the players. The offline preprocessing module is implemented in 1200 lines of C# code.

Renderer. To implement the renderer module, we create a Unity prefab called *SphereTexture*. At each rendering interval, the prefab sets the camera's far clip plane to be the preprocessed cutoff radius, it uses Unity APIs to load the far BE panoramic frame decoded from the prefetched video into the texture, and the Unity engine uses its built-in camera to crop the far BE from *SphereTexture* and render near BE and FI together with the cropped far BE to the display.

Decoder. Coterie leverages a third-party video player IJKPlayer [7] to exploit hardware-accelerated Android *MediaCodec* to decode the videos based on H.264. We modified about 1100 lines of C code in IJKPlayer to make it compatible with the renderer module.

Frame Cache and Prefetcher. We implement the frame cache module and the prefetcher module as a Unity plugin for the VR games in around 1100 lines of C code. The former checks for similar frames for a prefetching request. The latter sends the request to the server and then passes the server reply to the frame cache.

Sensing and Projection. We utilize Google Daydream SDK to (1) pass data collected from the sensors on the phone and the controller to the Unity engine; (2) project the frame generated by the renderer into two frames for the two eyes.

Ease of porting VR apps. We develop the Coterie framework to be app-independent which simplifies VR game development. To port a new VR game, the game developer just

Table 6. Average cache hit ratio across the players in the 3 games.

Game	Viking	Racing	CTS
Avg. cache hit ratio	80.8%	82.3%	88.4%

needs to (1) apply the offline preprocessing module to determine the cutoff radiuses and distance threshold; (2) generate panoramic frames of far BE based on the generated radiuses; (3) attach *SphereTexture* as a Unity prefab to the player camera to merge near and far BE properly; and (4) simply apply all other Coterie modules as plugins to the Unity project.

7 Testbed Evaluation

In this section, we evaluate Coterie's end-to-end performance and system resource usage in supporting VR games running on commodity phones. Due to page limit, we pick three multiplayer VR apps, Viking Village, CTS Procedural World, and Racing Mountain, one each from the 3 outdoor game categories (Table 2), that also have the largest dimension and the most grid points in the game world out of the 9 games (Table 3) and thus are the most challenging to implement on mobile devices. Our experiment setup is the same as described in §3. We compare Coterie with Thin-client and Multi-Furion as described in §3.

Offline preprocessing. Coterie first runs the adaptive cutoff algorithm to recursively divide the virtual world into multiple leaf regions. The results were discussed in §4.5.

Caching results. Both LRU and FLF work effectively as spatial locality and temporal locality coincide well in each player's movement. We omit the details due to page limit. Table 6 shows that exploiting intra-player frame similarity of far BEs achieves 80.8%, 82.3%, and 88.4% average frame cache hit ratios across 4 players for the 3 games, respectively, in the experiments below. These cache hit ratios tranlate into 5.2X, 5.6X, and 8.6X reduced far BE frame prefetching frequency per player from the server for the 3 games, respectively.

7.1 Visual Quality, Frame Rate, Responsiveness

We first compare the QoE of the three VR apps running on Thin-client, Multi-Furion, and Coterie with 2 players.

Image quality. In our experiment, we measure the SSIM between the frames rendered by each of the schemes with the frames directly generated on the client with 1920×1080 pixels, the resolution of Pixel 2 display. Table 7 shows the image quality results. Coterie obtains SSIM scores above 0.93, representing *good* visual quality of the generated frames. The reason that Coterie achieves higher SSIM than Multi-Furion and Thin-client is because it renders both FI and near BE locally without suffering encoding and decoding loss.

Frame rate. Table 7 shows that across all three high-quality VR games, Thin-client delivers the lowest FPS, between 15-19 FPS, Multi-Furion delivers 42-48 FPS, due to

Table 7. Visual quality, FPS and Responsiveness of each VR app under different implementations. (T: Thin-client, M: Multi-Furion, C: Coterie)

App	Visual Quality (Average SSIM)	Average FPS	Responsiveness (ms)
Viking (T)	0.912	19	41.0
CTS (T)	0.904	16	50.0
Racing (T)	0.949	15	42.2
Viking (M)	0.915	45	22.0
CTS (M)	0.907	48	20.1
Racing (M)	0.953	42	21.2
Viking (C)	0.937	60	15.8
CTS (C)	0.979	60	15.9
Racing (C)	0.975	60	15.6

extra network delay incurred by the second player, while Coterie comfortably delivers 60 FPS.

Responsiveness. Following [22], we define responsiveness as the motion-to-photon latency, *i.e.,* the elapsed time from when a user interaction for motion is detected by the device to when the corresponding frame is sent to the display. Table 7 shows that Coterie achieves far better responsiveness than the other systems. In particular, Coterie maintains the latency of 15.9ms, while Multi-Furion incurs the high latency of 22.0ms from the linearly increased network load.

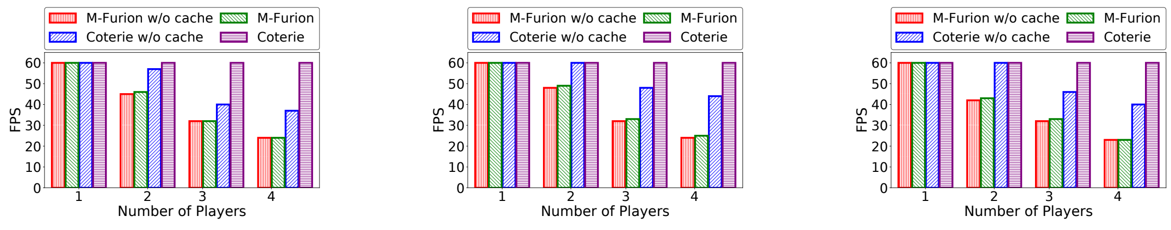
7.2 Scalability

Figure 11 shows how Coterie and Multi-Furion, with and without cache, scale with the number of players for the three apps. The original Multi-Furion has no frame cache. Multi-Furion with frame cache caches locally prefetched BE frames and performs exact matching. Coterie without cache differs from Multi-Furion in that it prefetches far BE frames as opposed to BE frames, which are about 2X-3X the size of far BE frames. We make the following observations. (1) All 4 versions achieve 60 FPS for 1 player, because the network is not bottlenecked. (2) The FPS of Multi-Furion with or without cache are almost identical, and gradually degrade with more players, reaching 24 for 4 players, due to increased network contention and lack of similarity between BE frames. (3) The FPS of Coterie without cache also degrades as the number of players increases, but slower than Multi-Furion, due to less network contention from prefetching smaller far BE frames. (4) Coterie with cache comfortably maintains 60 FPS for 4 players, from high reuse of similar frames in the cache.

7.3 Resource Usage

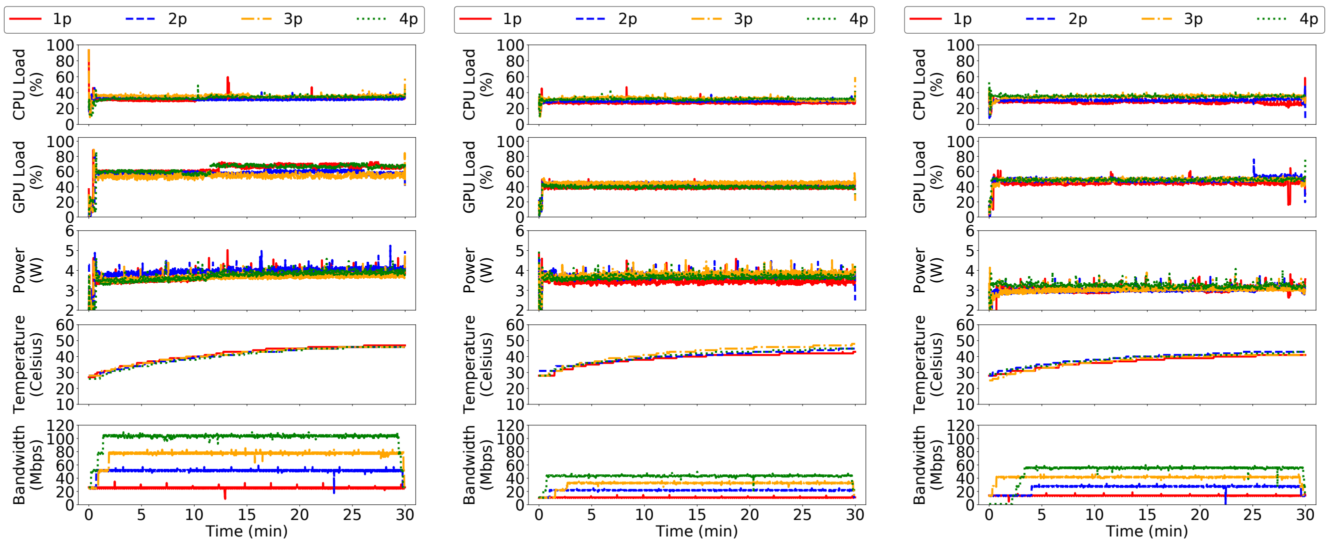
We next measure the resource usage of Coterie in supporting the multiplayer VR apps.

CPU/GPU utilization. We leverage procfs (/proc/stat) and sysfs (/sys/class/kgsl/kgsl-3d0) to obtain CPU and GPU load stats. Table 8 shows that Coterie incurs 32% CPU usage



(a) Viking Village **(b)** CTS **(c)** Racing

Figure 11. Scalability of Coterie compared with Multi-Furion.



(a) Viking Village **(b)** CTS **(c)** Racing

Figure 12. Scalability of resource usage under Coterie.

Table 8. Performance of Coterie on Pixel 2 over 802.11ac.

App (players)	FPS	Inter-Frame lat. (ms)	Phone CPU load (%)	Phone GPU load (%)	Per Frame size (kB)	Net. delay (ms)
Viking (1P)	60	16.0	31.76	55.51	280	7.0
CTS (1P)	60	16.6	27.76	44.81	150	6.0
Racing (1P)	60	16.0	26.99	39.18	194	6.5
Viking (2P)	60	16.5	31.89	57.24	280	8.9
CTS (2P)	60	16.6	28.13	46.89	150	6.3
Racing (2P)	60	16.2	28.98	43.25	194	7.5

and 58% GPU usage on average on the Pixel 2 phones in supporting the three high-quality multiplayer VR games with inter-frame latency below 16.7ms. Moreover, the GPU usage also does not increase with 2 players compared to with 1 player and the far BE frame transfer delay remains less than 9ms.

Network usage. Table 9 shows that for Coterie, as the number of players increases from 1 to 4, the average bandwidth usage of transmitting BE frames across the 3 games increases from 11-26 Mbps to 42-100 Mbps, while that of exchanging FI among the players increases from 1 Kbps to 275 Kbps, around 2-4 orders of magnitude lower than the traffic for BE. Since two or more players under Multi-Furion saturated the network bandwidth as shown in Table 1, we only show the network load for 1 player for Multi-Furion. Compared to Multi-Furion, Coterie reduces the network load of each player by 10.6X-25.7X. In other words, Coterie not only enables high-resolution multiplayer VR but also significantly optimizes single-player VR on commodity mobile devices.

CPU/GPU usage over time. Next, we measure the resource usage while playing each game for 30 minutes, with 1, 2, 3 and 4 players. Figure 12 shows that the CPU/GPU load under Coterie remains steady over the 30-minute window. We see Coterie incurs up to 40% CPU load and 65% GPU load for the three apps even for four players. Moreover, the CPU and GPU utilization do not increase with the number of players. This is because the rendering load on individual Coterie clients remains independent of the number of players.

Table 9. Network bandwidth usage of server transmitting far BE frames (in Mbps) and exchanging FI (in Kbps).

App	**Net. bandwidth usage** (BE/FI)				
	Multi-Furion	Coterie			
	1P	**1P**	**2P**	**3P**	**4P**
Viking	276/1	26/1	52/71	76/153	100/266
CTS	264/1	14/1	27/68	42/151	56/260
Racing	283/1	11/1	22/52	34/129	42/275

Temperature over time. Since VR apps are known to be power-intensive, we also measure the phone SoC temperature while running the three VR games. Figure 12 shows the SoC temperature increases gradually but stays under the thermal limit of Pixel 2, *i.e.,* 52 Celsius (from reading /vendor/etc/thermal-engine.conf).

Battery power draw over time. To measure the power draw during the 30-minute run, we log the current and voltage values from /sys/class/power_supply/battery and then compute the battery power offline. In all the experiments, the screen brightness is locked at 100% under the VR mode by the Android framework. Figure 12 shows that the power draw stays fairly steady at 4W on average for the three VR games. This can be explained by the steady CPU/GPU load and network load on the WiFi radio as we increase the number of players. Given Pixel 2's battery energy capacity of 2770 mAh, at the observed power draw rate, all three high-quality multiplayer VR apps can last for more than 2.5 hours.

7.4 User Study

Coterie ensures reused BE frames are sufficiently similar to the original frames. However, it may potentially increase the discontinuity of adjacent frames for each player. Since there are no standard metrics for measuring discontinuity, we conducted an IRB-approved user study with 12 participants to assess the impact of potential discontinuity on the user experience. To ensure the participants will watch the same game play, we first collected 6 single-player movement traces (*i.e.,* location in the virtual world), 2 traces for each of the three games and 20 seconds per trace, and then replayed the traces to the participants on Pixel 2 phones and Daydream headsets on top of Multi-Furion and Coterie, respectively. The participants were asked to grade for each game replay the difference between Multi-Furion and Coterie from 1 (very annoying) to 5 (imperceptible).

Table 10 shows the average score of 6 traces ranges from 4.5 to 4.75, meaning most of the participants think the difference is either invisible or visible but acceptable. Some volunteers observed slight stuttering at locations where the cutoff radius was small and a few objects were visually large in far BE, which made slight changes in far BE noticeable.

[3]User study score: 1: very annoying; 2: annoying; 3: slightly annoying; 4: perceptible but not annoying; 5: imperceptible.

Table 10. User scores in replaying the three VR games.[4]

Score	1	2	3	4	5
Percentage	0.0%	0.0%	5.5%	29.2%	65.3%

8 Related Work

We discuss closely related work in three areas.

Mobile virtual reality. Supporting untethered high-quality interactive VR, or "cutting the cord", has attracted strong interests from both industry and academia. Flashback [22] prerenders and caches on the mobile device all possible views, which can not support interactive VR imposed by user actions and requires overwhelming storage on the phone. Luyang *et al.* [30] exploit the adaptive Vsync hardware feature to adaptively change the rendering timings on the server in order to reduce the rate of missing frames, but assume a strong laptop as the mobile client to perform parallel decoding. Furion [27] as shown in §2 can only support a single player.

Mobile 360-degree video streaming. Many works study supporting high-quality 360-degree video streaming on head-mounted displays or commodity phones. Several works [21, 26] propose to pre-cache panoramic frames to provide the clients the freedom of changing orientation during playback. Other works [24, 25, 31, 35, 41, 42] exploit different projection and tile-based or viewport-based video encoding schemes to save network bandwidth. Compared to VR, 360-degree streaming does not need to render FI or predict player movement. EVR [29] co-optimizes cloud offloading with device hardware to reduce the energy of 360 video rendering.

Offloading for mobile gaming. There have been a large body of work on real-time rendering/cloud gaming for mobile devices [23, 28, 32–34, 36, 37, 40]. The latency requirement of VR systems (*e.g.,* under 25ms) is more stringent than games (*e.g.,* 100-200ms [34]) due to the near-eye setting.

9 Conclusion

To our best knowledge, Coterie presents the first framework that enables high-quality, immersive multiplayer VR on commodity mobile devices. Coterie tackles the challenge in scaling the prior-art VR system for multiple players, the proportional increase of network load, by exploiting similarity between background environment (BE) frames for nearby locations fetched from the server. Exploiting such frame similarity allows Coterie to reduce the prefetching frequency for each player by 5.2X-8.6X and the per-player network load by 10.6X-25.7X and comfortably support 4 players for 4K-resolution VR apps on Pixel 2 over 802.11ac.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This project is supported in part by NSF/Intel grant 1719369.

References

[1] 2013. Triangles Per Second: Performance Metric or Chocolate Teapot? https://community.arm.com/developer/tools-software/graphics/b/blog/posts/triangles-per-second-performance-metric-or-chocolate-teapot.

[2] 2014. What VR Could, Should, and almost certainly Will be within two years. http://media.steampowered.com/apps/abrashblog/Abrash%20Dev%20Days%202014.pdf.

[3] 2017. What Makes Multiplayer VR A Success And A Challenge? https://www.vrfocus.com/2017/09/what-makes-multiplayer-vr-a-success-and-a-challenge/.

[4] 2018. CTS Procedural World. https://assetstore.unity.com/packages/tools/terrain/cts-complete-terrain-shader-91938.

[5] 2018. Gear VR. http://www.samsung.com/global/galaxy/gear-vr/.

[6] 2018. Google Daydream. https://vr.google.com/daydream/.

[7] 2018. Ijkplayer. https://github.com/Bilibili/ijkplayer.

[8] 2018. Location-based entertainment is a multi-billion dollar market for virtual reality. https://greenlightinsights.com/location-based-entertainment-a-multi-billion-dollar-market-for-virtual-reality/.

[9] 2018. Photon Unity Networking. https://www.photonengine.com/en-US/PUN.

[10] 2018. Racing Mountain. https://assetstore.unity.com/packages/templates/systems/racing-game-template-41864.

[11] 2018. Unity 3D. https://unity3d.com.

[12] 2018. Viking Village. https://assetstore.unity.com/packages/essentials/tutorial-projects/viking-village-29140.

[13] 2018. Virtual Reality Market Size, Share And Forecast To 2022. https://www.marketwatch.com/press-release/virtual-reality-market-size-share-and-forecast-to-2022-2018-07-04.

[14] 2018. Why Multiplayer Is Key to Location-Based VR Success. https://immersed.io/multiplayer-location-based-vr/.

[15] 2019. Oculus Go VR Store. https://www.oculus.com/experiences/go/?locale=en_US.

[16] 2019. Oculus Quest VR Store. https://www.oculus.com/experiences/quest/?locale=en_US.

[17] 2019. PlayStation VR Store. https://www.playstation.com/en-gb/explore/playstation-vr/games/.

[18] 2019. x264 library. http://www.videolan.org/developers/x264.html.

[19] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. 2016. Cutting the Cord in Virtual Reality. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*. ACM, 162–168.

[20] Omid Abari, Dinesh Bharadia, Austin Duffield, and Dina Katabi. 2017. Enabling High-Quality Untethered Virtual Reality.. In *NSDI*. USENIX, 531–544.

[21] Matthias Berning, Takuro Yonezawa, Till Riedel, Jin Nakazawa, Michael Beigl, and Hide Tokuda. 2013. pARnorama: 360 Degree Interactive Video for Augmented Reality Prototyping. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication* (Zurich, Switzerland) *(UbiComp '13 Adjunct)*. ACM, New York, NY, USA, 1471–1474. https://doi.org/10.1145/2494091.2499570

[22] Kevin Boos, David Chu, and Eduardo Cuervo. 2016. FlashBack: Immersive Virtual Reality on Mobile Devices via Rendering Memoization. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 291–304.

[23] Eduardo Cuervo, Alec Wolman, Landon P Cox, Kiron Lebeck, Ali Razeen, Stefan 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*. ACM, 121–135.

[24] Jian He, Mubashir Adnan Qureshi, Lili Qiu, Jin Li, Feng Li, and Lei Han. 2018. Rubiks: Practical 360-Degree Streaming for Smartphones. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services* (Munich, Germany) *(MobiSys '18)*. ACM, New York, NY, USA, 482–494. https://doi.org/10.1145/3210240.3210323

[25] Mohammad Hosseini and Viswanathan Swaminathan. 2016. Adaptive 360 VR Video Streaming: Divide and Conquer! *CoRR* abs/1609.08729 (2016). arXiv:1609.08729 http://arxiv.org/abs/1609.08729

[26] Evgeny Kuzyakov and David Pio. 2016. Next-generation video encoding techniques for 360 video and vr.(2016). https://code.facebook.com/posts/1126354007399553/nextgeneration-video-encoding-techniques-for-360-video-and-vr.

[27] Zeqi Lai, Y. Charlie Hu, Yong Cui, Linhui Sun, and Ningwei Dai. 2017. Furion: Engineering High-Quality Immersive Virtual Reality on Today's Mobile Devices. In *Proc. of ACM MobiCom*.

[28] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. 2015. 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*. ACM, 151–165.

[29] Yue Leng, Chi-Chun Chen, Qiuyue Sun, Jian Huang, and Yuhao Zhu. 2019. Energy-efficient Video Processing for Virtual Reality. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. ACM, New York, NY, USA, 91–103. https://doi.org/10.1145/3307650.3322264

[30] Luyang Liu, Ruiguang Zhong, Wuyang Zhang, Yunxin Liu, Jiansong Zhang, Lintao Zhang, and Marco Gruteser. 2018. Cutting the Cord: Designing a High-quality Untethered VR System with Low Latency Remote Rendering. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services* (Munich, Germany) *(MobiSys '18)*. ACM, New York, NY, USA, 68–80. https://doi.org/10.1145/3210240.3210313

[31] Feng Qian, Bo Han, Qingyang Xiao, and Vijay Gopalakrishnan. 2018. Flare: Practical Viewport-Adaptive 360-Degree Video Streaming for Mobile Devices. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking* (New Delhi, India) *(MobiCom '18)*. ACM, New York, NY, USA, 99–114. https://doi.org/10.1145/3241539.3241565

[32] Ryan Shea, Di Fu, and Jiangchuan Liu. 2015. Rhizome: Utilizing the public cloud to provide 3D gaming infrastructure. In *Proceedings of the 6th ACM Multimedia Systems Conference*. ACM, 97–100.

[33] Ryan Shea and Jiangchuan Liu. 2013. On GPU pass-through performance for cloud gaming: Experiments and analysis. In *Proceedings of Annual Workshop on Network and Systems Support for Games*. IEEE Press, 1–6.

[34] Ryan Shea, Jiangchuan Liu, Edith C-H Ngai, and Yong Cui. 2013. Cloud gaming: architecture and performance. *IEEE Network* 27, 4 (2013), 16–21.

[35] Shu Shi, Varun Gupta, and Rittwik Jana. 2019. Freedom: Fast Recovery Enhanced VR Delivery Over Mobile Networks. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services* (Seoul, Republic of Korea) *(MobiSys '19)*. ACM, New York, NY, USA, 130–141. https://doi.org/10.1145/3307334.3326087

[36] Shu Shi and Cheng-Hsin Hsu. 2015. A survey of interactive remote rendering systems. *Comput. Surveys* 47, 4 (2015), 57.

[37] Shu Shi, Cheng-Hsin Hsu, Klara Nahrstedt, and Roy Campbell. 2011. Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming. In *Proceedings of the 19th ACM international conference on Multimedia*. ACM, 103–112.

[38] Robert Toth, Jim Nilsson, and Tomas Akenine-Möller. 2016. Comparison of Projection Methods for Rendering Virtual Reality. In *Proceedings of High Performance Graphics* (Dublin, Ireland) *(HPG '16)*. Eurographics Association, Goslar Germany, Germany, 163–171. https://doi.org/10.2312/hpg.20161202

[39] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

[40] Jiyan Wu, Chau Yuen, Ngai-Man Cheung, Junliang Chen, and Chang Wen Chen. 2015. Enabling adaptive high-frame-rate video streaming in mobile cloud gaming applications. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 12 (2015), 1988–2001.

[41] Xiufeng Xie and Xinyu Zhang. 2017. POI360: Panoramic Mobile Video Telephony over LTE Cellular Networks. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies* (Incheon, Republic of Korea) *(CoNEXT '17)*. ACM, New York, NY, USA, 336–349. https://doi.org/10.1145/3143361.3143381

[42] Chao Zhou, Zhenhua Li, and Yao Liu. 2017. A Measurement Study of Oculus 360 Degree Video Streaming. In *Proceedings of the 8th ACM on Multimedia Systems Conference* (Taipei, Taiwan) *(MMSys'17)*. ACM, New York, NY, USA, 27–37. https://doi.org/10.1145/3083187.3083190