Xingyu Chen University of California San Diego xic063@ucsd.edu

Xinyu Zhang University of California San Diego xyzhang@ucsd.edu Xinmin Fang University of Colorado Denver xinmin.fang@ucdenver.edu

Liang He University of Nebraska Lincoln liang.he@ucdenver.edu Shuting Zhang Guangdong University of Technology ShutingZhang1114@163.com

Zhengxiong Li University of Colorado Denver zhengxiong.li@ucdenver.edu

ABSTRACT

Mobile Virtual Reality (VR) is essential for achieving convenient and immersive human-computer interaction and realizing emerging applications such as Metaverse and spatial computing. However, existing VR technologies require two separate renderings of binocular images, thereby causing a significant bottleneck for mobile devices with limited computing and battery capacity. This paper proposes a new approach to optimizing mobile VR rendering called YORO. By utilizing the per-pixel attribute, YORO can generate binocular VR images from the monocular image through genuinely one rendering, saving half the computation over conventional approaches. Our experimental evaluation and detailed user study indicate that, YORO can save 27% power consumption on average and increase frame rate by 115.2%, while maintaining similar binocular image quality compared with state-of-the-art mobile VR rendering solutions. YORO is production-ready and has already been tested in real VR applications. The source code, demo video, prototype android app, video game engine plugins, and more are released anonymously at YORO-VR.github.io.

CCS CONCEPTS

• Human-centered computing → Virtual reality; • Computer systems organization → Real-time systems; • Computing methodologies → *Rendering*.

KEYWORDS

Virtual Reality, Rendering Optimization, Energy Efficiency

ACM Reference Format:

Xingyu Chen, Xinmin Fang, Shuting Zhang, Xinyu Zhang, Liang He, and Zhengxiong Li. 2025. You Only Render Once: Enhancing Energy and Computation Efficiency of Mobile Virtual Reality. In The 23rd Annual International Conference on Mobile Systems, Applications and Services (MobiSys '25), June 23–27, 2025, Anaheim, CA, USA. ACM, New York, NY, USA, 14 pages. https: //doi.org/10.1145/3711875.3729133

Xingyu Chen and Xinmin Fang contributed equally to this work.

MobiSys '25, June 23–27, 2025, Anaheim, CA, USA

© 2025 Association for Computing Machinery.

ACM ISBN 979-8-4007-1453-5/25/06...\$15.00 https://doi.org/10.1145/3711875.3729133

1 INTRODUCTION

Mobile Virtual Reality (VR), encompassing smartphone VR and standalone or all-in-one VR systems (e.g., Meta Oculus Quest), utilizes mobile devices to create a binocular, image-based virtual space. It is capable of simulating vision and a sense of depth, thereby immersing the user in meticulously crafted scenes [28]. Consequently, mobile VR users have skyrocketed in recent years, now surpassing desktop/PC VR users by nearly threefold, and the global market for mobile VR is anticipated to grow to \$20.9 billion by 2025 [13]. Furthermore, VR technology is increasingly seen as the next significant mobile computing platform. It is deemed an essential content type for the forthcoming iterations of the Internet and various emerging applications, including Digital Twins, spatial computing, and the Metaverse [56].



Figure 1: YORO proposes an approach of mobile VR optimization for energy-saving and efficiency with a negligible loss in image quality.

However, certain challenges, especially those related to rendering, significantly impede the widespread adoption of mobile VR. Rendering is particularly resource-intensive, consuming almost twice the energy required for rendering conventional planar videos. This can easily cause the mobile device to heat up and exceed its Thermal Design Point [32, 41]. Such high power consumption and thermal radiation hinders high frame rates (90-120 Frames-Per-Second) and high-quality graphics, which in turn compromises immersion and can cause motion sickness for users. As a result, current mainstream mobile devices, typically equipped with 3500 to 5000 mAh batteries, can only sustain mobile VR applications for approximately two to three hours, which is insufficient for many practical usage scenarios [27]. Moreover, mobile VR usually can only support VR apps with a computational complexity ranging from 350k to 1,000k triangles [16] in the mesh-represented VR scene.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

In contrast, profitable and complex VR scenes often contain more than 2,000k triangles, limiting the functionality and applicability of mobile VR. Additionally, the current VR frame rate on mobile devices still falls short of the ideal 90-120 frames-per-second (FPS) range required for immersive VR experience [8]. Consequently, optimizing VR rendering to minimize computational requirements can fundamentally decrease power consumption and enhance the usability of mobile VR.

In this paper, we propose YORO, a rendering optimization framework for mobile VR. As shown in Figure 1, unlike the conventional binocular VR rendering method, YORO only needs rendering once for the binocular image. Unlike existing works that only save computation on geometrical information in the rendering process, YORO saves an entire rendering pass calculation, including geometrical information and shading information such as textures, lighting, and shadowing (detailed in Section 3.1). YORO is designed to achieve the following salient properties: (i) Energy efficiency: It requires less energy to provide an equivalent user experience. It prevents heat and processor degradation while improving battery life on mobile VR applications; (ii) Computational efficiency: It makes mobile VR more efficient and reliable on less computing recourse in practice. Compared to conventional algorithms, it enables higher and more stable frame rates for the same VR scenes and can handle more complex scenes with the same computational resources; (iii) Practicality: It is a general framework-level approach for VR applications, does not need specialized hardware, and is compatible with most current mobile platforms.

To realize YORO, we need to address two technical challenges: (1) A new computational optimization paradigm for mobile VR. We propose to optimize the mobile VR through VR binocular image generation by rendering once. We develop a new reprojection matrix to quickly reproject frames from one eye to the other, followed by a new filter-based patching method to fill in the missing information. Our algorithm has about half of the computational complexity compared to conventional rendering algorithms (Section 10.3). This in turn improves the energy efficiency of the entire VR system. (2) Implement the YORO as an efficient software framework underlying practical VR applications. To achieve the goal of computation efficiency and energy saving, we design and implement the YORO rendering algorithms in a lightweight and highly parallel way. Our implementation resides at the framework layer, which contains Reprojector and Patcher modules to generate binocular VR images with a single-round rendering (Section 4). Moreover, we build this framework from scratch with low third-party and system functions dependencies, while allowing it to function properly for customized VR apps and commercial VR app products across multiple mobile platforms, including Android phones and standalone VR headsets. Our evaluation indicates that, on average, YORO can save 27% power consumption and increase the frame rate by 115.2% compared to the baseline binocular rendering and state-of-art VR optimization methods. Meanwhile, our quantitative analysis along with subjective user study indicate that YORO maintains similar binocular image quality and achieves a smoother VR experience owing to a much higher frame rate.

Our contributions can be summarized in four-fold:

- We investigate the redundancies in the VR rendering process and propose a novel paradigm of mobile VR optimization. To our knowledge, this is the first work to attempt "rendering once" for efficient mobile VR.
- We design a lightweight post-processing rendering mechanism to achieve the YORO principle, and implement our design as a general framework layer to support various mobile VR applications. This implementation includes the codbase, along with an open-source dataset with 29,128 groups of images from eight representative scenes with mainstream rendering style.
- We extensively evaluate YORO in eight representative VR 3D scenes and two open-source VR app products with different shading styles and scene complexities. Further, we test the robustness and practicality of YORO on eight different mobile VR devices across various mobile platforms.

2 RELATED WORK

Visual Optimization for VR: There are three mainstream solutions for optimizing the visual/graphics aspects of VR.

(1) Content Optimization: Content optimization, such as Level-Of-Detail (LOD) [21, 45] and Occlusion Culling (OC) [24]), can increase efficiency by reducing the amount of content to be rendered. However, such methods strongly rely on the visual content, and require extra analysis and modifications of each 3D model. (2) Rendering Optimization: Rendering optimization usually reduces the amount of computation and increases efficiency by adjusting the image generation pipeline. The most straightforward approach is to hugely decrease the image quality (e.g., resolution) or frame regions to reduce workload [27, 29, 61, 65]. However, this approach can significantly degrade the image quality/user experience or require specialized hardware modification with eye-tracking functionality. An alternative technique, Single Pass Instancing (SPI) [60] (or Single Pass Stereo [59] on PC and MultiView [10] on Oculus), can improve efficiency by reusing computational information across eye views, reducing GPU draw calls and geometric processing overhead. However, SPI only reduces GPU draw calls on the CPU side [5] and requires significant additional shader code modifications. The existing SPI implementation only supports forward rendering mode, wherein each light source requires separate computational passes, making it challenging to efficiently process scenes with dynamic or multiple light sources (including reflections). As a result, SPI-based VR applications are restricted to environments with minimal lighting complexity. Our experiments show that the rendering efficiency of SPI falls short in practical VR scenes (detailed in Section 3.1&10.3&10.6). Notably, aside from application programming interfaces (APIs), no open-source SPI implementation is currently available. Another emerging solution attempts to separate the frame regions for rendering [28]. However, this approach cannot reduce the computation significantly because it still needs to render the foreground twice. (3) Reprojection: The reprojection-based solutions are also developing swiftly in recent years. A few designs have been proposed [52, 66] to achieve reprojection and mitigate the associated artifacts (details in Sections 6 & 7). However, existing works only target boosting high-end PC VRs' frame rates and are not applicable to mobile VR optimization, due to fundamental differences in hardware capabilities, power constraints, and

computational resources. PC VR reprojection techniques assume high-end graphics processing and consistent power supply, whereas mobile VR requires ultra-lightweight methods that minimize energy consumption and computational overhead. *In contrast to these existing three categories of solutions, YORO is a lightweight rendering optimization framework that is ready to deploy on mobile VR*.

Computational Optimization for VR: Table 1 summarizes various methods for improving the computational efficiency of VR applications. One representative approach leverages specialized hardware components (e.g., the display panel), or offloads the computation to remote edge/cloud servers [25, 31, 36, 40]. However, these methods often entail additional hardware cost, and often struggle to reduce the overall energy consumption due to sophisticated computation-communication trade-offs. Other approaches include supersampling [47, 67], image precomputing and caching [23], motion prediction [51, 54], parallel computing [35], and communication protocol [43, 44, 54]. However, these techniques, while effective for high-end PC VR systems, are often impractical for mobile VR applications. They tend to fundamentally prioritize performance and image quality rather than addressing constraints on energy, processing power, and thermal management. Notably, YORO's monocular-to-binocular rendering method is complementary and can be combined with these techniques once mobile computing power evolves to today's PC level.

3 BACKGROUND AND PRELIMINARY



Figure 2: We can reuse the spatial pixel information of a single-eye image to construct the information of the other eye in a speedy manner via one single matrix transformation, thus saving one entire rendering pass and about 50% computation amount. Note: The IPD between the left eye and right eye (i.e., the pixel difference among eyes) here is enlarged for illustration.

3.1 VR Rendering Process

The rendering process in VR generates 2D images or frames as a field of view (FoV) originating from the 3D scene. The locations and shapes of the objects in an image/frame are determined by their geometry, the characteristics of the environment, and the placement of the camera in that environment. The appearance of the objects is affected by material properties, light sources, textures, and shading models [17]. Specifically, geometry is described by a large collection of triangles grouped into 3D meshes together to approximate the contour of 3D objects in the scene. Therefore, the number of triangles is the measure of scene complexity, with a higher number of triangles usually resulting in more detailed and realistic imagery. In mobile VR, rendering relies on rasterization, a computationally efficient technique that transforms 3D scenes into 2D pixels. This approach balances visual quality and performance, and is expected to remain the primary rendering method for VR in the future [7].

The conventional rasterized rendering pipeline comprises two steps: (1) Projection: The renderer utilizes view matrices that depend on the position and rotation of the camera to transform the input geometry from model coordinate space to view space. Then the geometry will be converted into clipping space using the projection matrix, which depends on the parameters of the camera. Here the redundant geometry is clipped out, and finally, the geometry is mapped to the screen space. (2) Shading: The geometry is then rasterized to the screen pixels and colored by the fragment shader. The color of a pixel depends on many factors, such as texture, reflection, refraction, direct and indirect light, and air medium. Therefore, the shading process is more computationally expensive than the projection process (about 3:1). As VR games or other applications evolve towards higher 3D fidelity, more factors and computational load will be added. Generally, in VR rendering, the conventional process is to render the left eye and the right eye images separately, which requires rendering twice, 2× a frame rendering consumption, including two projection processes and two shading processes [5, 20]. Existing rendering optimization, such as Single Pass Instancing (SPI) [60], can only eliminate one projection process. In contrast, YORO can reduce more energy by eliminating one projection and one shading process as shown in Figure 2.

3.2 G-Buffers Exploration for VR Optimization

The renderer mentioned in Section 3.1 usually outputs the following property images: (1) RGB: The Red-Green-Blue three-channel image represents the color of the rendering. RGB is the color model used in mainstream electronic devices and picture formats [34], as it is based on the principle of monitor display and human perception of color. Almost all mainstream renderer solutions output RGB images. (2) Depth: The depth image is a grayscale image (single-channel) in which each pixel's brightness represents the distance of the object in logarithmic space. The brighter the pixel, the closer to the camera. These above-mentioned images are parts of G-buffers [3], a screen space representation of geometry and material information of the rendering process. It is worth noting that getting the G-buffers does not add extra computation since it is already given by the regular rendering pipeline. After obtaining the screen space information, we can usually simulate visual effects on images, such as post-processing effects (occlusion, reflection, shadow, mobile blur, etc.). Thus, this mechanism can also aid us in optimizing mobile VR by utilizing the G-buffers which have already been generated as part of the regular rendering pipeline, without extra computational costs.

Moreover, conventional binocular rendering methods require separate rendering for both eyes (parallax) to allow users to have a sense of depth (stereopsis). The depth information is a consistent information in the rendering process. Therefore, in principle, it is possible to do a single rendering of the image with the depth information to build binocular images without rendering twice.

Works	Optimization Method	Frame Rate Improvement	Power Consumption	Data Transmission Amount	Image Quality	Mobile Generalization
Huang et al. [35]	Parallel the rendering process on CPU end	High (+37%)	High (+5% \sim 10%)	No Reduction	Lossless	Low
Pohl et al. [51]	Reduce rendering area	High (+90%)	High (Raytraicng)	No Reduction	High (-5%)	Low
Xiao et al. [67]	Super sampling	High (+40%)	High (+10% ~ 20%)	Not Discussed	Medium (-10%)	Low
LookinGood [47]	Neural re-rendering	Low (-27%)	High (+50% ~ 150%)	Low (-50% ~ 75%)	Medium (-3% ~ 10%)	Low
Friston et al. [29]	Accelerate rasterization using perceptual information	Low ($-10\% \sim 15\%$)	Low (-20% ~ 35%)	Low (-25%)	High (-1%)	Low
Single Pass Instancing (SPI) [60]	Use a warped camera to render two view at once	High (+15% \sim 20%)	Medium (-5% ~ +5%)	No Reduction	Lossless	Low
Fink <i>et al.</i> [28]	Render far field using one camera and near field using two	Low (-5%)	Medium (+2%)	No Reduction	High (-3%)	Low
CollabVR [38]	Render one eye on the cloud, and reproject the other eye on the edge	Not Discussed	Low (-71%) on edge, extra on cloud	Low (-42%)	High ($-3\% \sim 10\%$)	High
YORO (Ours)	Reprojection + Patcher	High (+115.2%)	Low (-27%)	Low (-39.6%)	High (-3%)	High

Table 1: A Comparison of Mobile VR Optimization approaches.



Figure 3: We can reuse the spatial pixel information of a single eye image to construct the information of the other eye image in a speedy manner via one single matrix transformation, thus saving one entire rendering pass and saving about 50% computation amount. Note: The IPD between right and left eyes (i.e., the pixel difference among eyes) here is enlarged for illustration.

4 SYSTEM OVERVIEW

An overview of YORO is shown in Figure 4. In contrast to the conventional VR rendering, which requires one render for each eye image, YORO only renders once for the dominant eye of the user. The dominant eye is decided by personal habits and unchanged across VR applications. The renderer outputs the intermediate result that contains the RGB color image and the depth image. The intermediate results are then fed into the Reprojector. The Reprojector quickly creates a new cropped geometry based on the RGB and depth pixel information. This cropped geometry is then reprojected to the new eye matrix. The final output of the Reprojector is resolution-independent Intermediate Buffers (ImBuffer) (detailed in Section 6). The ImBuffer are then fed into the Patcher, which leverages information from the ImBufferto sample and fill in the disocclusion, i.e., scene regions that become newly visible to one eye but were not visible in the original rendering for the dominant eye. The rendered and patched frames are combined as the binocular image and then displayed on the VR headset.

5 OBSERVATIONS AND RATIONALE FOR YORO

To ensure YORO can run effectively on mobile GPUs, we also **consider** the following computation architecture aspects: **(C1)**

Read/Write Operations. On the mobile GPU, textures, and buffers are stored in texture memory, which takes much longer time to access than the GPU's local memory and registers. Therefore, it is imperative to avoid frequent reads and writes to texture memory on the mobile GPU to improve efficiency. **(C2) Thread safety.** Multiple threads reading and writing to the same memory location at the same time can cause thread conflicts, not only by creating flickering artifacts but also by significantly increasing the time consumption due to the racing condition. It is thus critical for us to ensure thread safety.

In addition, we exploit two practical **Observations** to curtail unnecessary computation: (O1) When reprojecting from one eye to the other, the pixels will only be displaced in the opposite direction. For example, when we reproject from the right eye to the left eye, all pixels will only displace along the positive X-axis (i.e., to the right) for a certain distance (range from 0 to texture width), as shown in Figure 3 (3-Pixel Shift). Therefore, we can save computing time by completely disregarding the calculation of the Y-axis and the negative X-axis. (O2) The depth of the disocclusion is always further than the nearest colored pixel in the opposite direction (when the right eye is the dominant eye), as shown in Figure 3 (2-Left Eve Depth (Ref.))&(4-Reprojected). In other words, the disocclusion should always be patched with backgrounds, not foregrounds. This observation helps optimize the rendering process by reducing unnecessary calculations and focusing only on the background when filling in the disocclusion.

6 REPROJECTOR

This module is mainly designed to solve the monocular-to-binocular generation problem in mobile VR, i.e., generating a one-eye frame from another one with depth information to form a binocular image. To this end, we explore the reprojection technique, which reconstructs a new frame with a different perspective through existing color and depth information-information that can be naturally obtained from the renderer. There exist a few works on using reprojection to achieve VR optimization. Asynchronous Reprojection [6] or Asynchronous Time Wrap [46] (ATW) uses reprojection to generate a frame from previous frames for a new head rotation for VR when the GPU can not keep up with the headset's target frame rate. However, It requires motion vectors to achieve reprojection which already requires substantial computation [12], unsuitable for mobile devices. Besides, Wißmann et al. [66] attempts to use hardware tessellation to achieve projection, and Schollmeyer et al. [52] achieves reprojection via a warping grid and ray-casting. However, both have heavy computation overhead, lack system design and



Figure 4: The system overview for YORO achieves the high-efficiency VR binocular image generation. Compared to conventional VR rendering, YORO only uses one rendering. Therefore it reduces the amount of computation and thus increases efficiency and saves energy.

implementation details, or rely on information that is not practical on mobile GPUs, such as the accumulation buffer (A-buffer) [2]. Therefore, *existing reprojection methods do not properly take into account the constraints of mobile VR devices*.

Before introducing our **Reprojector**, we brief on the principle of the renderer. As mentioned in Section 3.1, the current mainstream real-time renderer is dominated by the rasterization renderer. Its core idea is to traverse each triangle of each 3D model in the scene and project it from the world space to the screen space using the view and projection matrix. The matrices are denoted as:

$$\mathbf{R} = \begin{bmatrix} 1 - 2r_{Z}^{2} - 2r_{W}^{2} & 2r_{Y}r_{Z} - 2r_{X}r_{W} & 2r_{Y}r_{W} + 2r_{X}r_{Z} & 0\\ 2r_{Y}r_{Z} + 2r_{X}r_{W} & 1 - 2r_{Y}^{2} - 2r_{W}^{2} & 2r_{Z}r_{W} - 2r_{X}r_{Y} & 0\\ 2r_{Y}r_{W} - 2r_{X}r_{Z} & 2r_{Z}r_{W} + 2r_{X}r_{Y} & 1 - 2r_{Y}^{2} - 2r_{Z}^{2} & 0\\ 0 & 0 & 0 & 1 \end{bmatrix},$$
(1)
$$\mathbf{V} = \begin{bmatrix} 1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & -1 & 0\\ 0 & 0 & 0 & 0 \end{bmatrix} \cdot \mathbf{R} \cdot \begin{bmatrix} 1 & 0 & 0 & t_{X} \\ 0 & 1 & 0 & t_{Y} \\ 0 & 0 & 1 & t_{Z} \\ 0 & 0 & 0 & 1 \end{bmatrix},$$
(2)
$$\begin{bmatrix} \frac{1}{Aspect \times size} & 0 & 0 & 0\\ 0 & \frac{1}{2} & 0 & 0 & 0 \end{bmatrix}$$

$$P = \begin{bmatrix} 0 & \frac{1}{size} & 0 & 0\\ 0 & 0 & -\frac{2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix},$$
(3)

where R is the rotation matrix. V is the view matrix. P is the projection matrix. (t_x, t_y, t_z) is a 3D vector represents the camera world position. (r_x, r_y, r_z, r_w) is a unit quaternion that represents the camera rotation. *Aspect* is the screen aspect ratio, *size* is half height of the view frustum. *far* is the distance of camera's far plane, far = 1000 be default [58]. *near* is the distance of camera's near plane, *near* = 0.3 be default [58]. The 3D camera can only render objects with distances between the far plane and the near plane.

The projection of rasterization can be formulated as:

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{bmatrix} \cdot \mathbf{VP} = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \\ \mathbf{d} \\ 1 \end{bmatrix}, \quad (4)$$

where (x, y, z) is the world position of mesh model's vertex. (u, v) is the pixel position on the screen, and *d* is the depth of the corresponding pixel.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}_{\text{left}} \cdot \boldsymbol{M} = \begin{bmatrix} u \\ d \\ 1 \\ 1 \end{bmatrix}_{\text{right}}, \quad \boldsymbol{M} = (\boldsymbol{V}\boldsymbol{P}_{\text{left}})^{-1} \cdot \boldsymbol{V}\boldsymbol{P}_{\text{right}}, \quad (5)$$

With Equation 5, we can perform reprojection, which essentially calculates the other camera's screen coordinates of each pixel from the depth map of the current camera. This reprojection process is simply a single matrix transformation and can be computed in parallel on the GPU. To further enhance the performance, our Reprojector design and employs four key strategies. We specifically focus on minimizing computational bottlenecks, such as GPU-CPU communication, and reducing the overall computational load through both architectural (a-c) and algorithmic (d) optimizations.

(a) Thread-safe Hybrid Shader Architecture: Reprojection operation in VR commonly employs Compute Shader (CS), i.e., specialized programs designed for parallel GPU processing. However, mobile devices provide limited support for CS, resulting in an insufficient performance boost and often causing additional computation burden. Flickering artifacts also appear due to the conflict of multiple threads writing to the same pixel location, which will cause flicker and shake on certain areas of the images. To overcome these challenges, we propose a thread-safe hybrid shader architecture that leverages the strengths of both Compute Shaders and Image Effect Shaders (IES). Specifically, we apply the principle of separation of concerns, under the guidance of Observation O1 in Section 5. The IES is used to efficiently handle matrix transformation computations, which are typically uniform and do not require random access to memory. On the other hand, the Compute Shader is specifically tasked with buffer random writing, but instead of allowing threads to operate freely across the entire image, the workload is parallelized per row of pixels. By restricting each thread to operate within a specific row, the chances of multiple threads writing to the same pixel location are eliminated.

(b) Disocclusion Tracking: To optimize the use of information shared between modules during computation, we introduce a novel Disocclusion Tracking method. In this approach, the CS is designed to operate in a per-row parallelized manner, enabling it to calculate and store both the location and width of disocclusions caused by the reprojection process in a single pass. By efficiently capturing this disocclusion data during the same operation, it can be seamlessly utilized by the subsequent module (i.e., the Patcher, as detailed in Section 7) to accelerate its processing. This design minimizes additional computational overhead while significantly improving the overall efficiency of the pipeline.

(c) Resolution-Independent Intermediate Buffers: As the displays of mobile VR devices evolve, their resolution will gradually increase to 4K or even 8K [15]. However, the reprojection is independent of the scene complexity but is related to the screen resolution (Section 10.3), which may significantly increase the computation load. To proactively address this issue, we propose the resolution-independent Intermediate Buffers (ImBuffer). The

MobiSys '25, June 23-27, 2025, Anaheim, CA, USA

Xingyu Chen, Xinmin Fang, Shuting Zhang, Xinyu Zhang, Liang He, and Zhengxiong Li

resolution of ImBuffer can be set to a constant or downsampled 1/2 to 1/16 of per-eye resolution before applying YORO shaders. The ImBuffer records the distance the pixel shifts along the horizontal contour. The final full-resolution image is sampled based on linear interpolation of the distance shifted. This will avoid the extra computation burden when YORO is applied to higher resolution.

(d) Linear Interpolation: When down-sampling the ImBuffer from floating-point UV coordinates to integer XY pixel coordinates, errors can occur if the fractional positions are not correctly handled. We address these issues by applying linear interpolation at the horizontal axis to improve image quality. While this approach doubles the shader operations, making it optional serves as a strategic design choice that enables dynamic adaptation to diverse mobile hardware capabilities - high-end devices can enable it for maximum visual quality, while devices with limited processing power can disable it to maintain performance, thereby effectively implementing the hardware adaptability principle outlined in **Consideration C1** of Section 5.

Algorithm 1 Reprojector Stage1 (Image Effect Shader, Parallel per pixel)

Input: D: Depth buffer of rendered eye.
M: Precomputed transformation matrix in Eq.4.
uv: The resolution-independent pixel coordinate ranges (0,1). Provided by the shader
Output: Intermediate buffer pixel at coordinate <i>uv</i> .
1: $c \leftarrow (uv *2-1, D[uv], 1)$
2: $\hat{c} \leftarrow M^* c$
3: return (\hat{c} .x + 1 * 0.5, \hat{c} .z)

Algorithm 2 Reprojector Stage 2 (Compute Shader, Parallel per row of pixel)

Input: S: Intermediate buffer.	
W: Texture width. y: The row id. Provided	l by the shader.
Output: Ŝ: Transformed Intermediate buffer.	
1: $x_0 = 0$	
2: for $x = 0$: W do	▹ For each pixel along the row
3: $c = S[x,y]$	
4: $\hat{x} \leftarrow c.x$	Get the x location to write to
5: $d \leftarrow c.y$	▶ Get the depth
6: if $\hat{S}[\hat{x}, y].y < d$ then	
7: $w \leftarrow \hat{x} - x_0$	 Calculate the width of disocclusion
8: for $i = x_0 + 1 : \hat{x}$ do	▷ And iterate through it
9: $\hat{S}[i,y] \leftarrow (x_0, d, w)$	▹ Stores disocclusion's
10: end for	▷ (left edge, depth of right edge, width)
11: $\hat{S}[\hat{x}, y] \leftarrow (x, d, 0)$	▹ Stores (pixel location that
12: end if	 comes from, reprojected depth)
13: $x_0 \leftarrow \hat{x}$	 Update disocclusion tracking pixel
14: end for	

As shown in Algorithm 1, YORO first takes the full resolution depth map and downsamples ImBuffer as input, and computes the *"location will be written to"* value and *"reprojected depth"* value via a per-pixel-parallel image effect shader. However, Algorithm 1 calculates the matrix transformation but does not perform buffer random read/write operations (writing to texture location that doesn't belong to the current thread). Therefore, the ImBuffer is further fed into a per-row-parallel compute shader Algorithm 2 and transforms the *"location will be written to"* value to *"locations that come from"* value via a scan along the X-axis. When multiple pixel values are written to the same location, Algorithm 2 keeps the value with the lowest depth. Besides, it will also detect if the *"location will be written to"* value has a change of more than one pixel (i.e., a disocclusion) and add its start location and width to the ImBuffer. Examples of reprojection are shown in Figure 3 and 5 (using the right eye as the dominant eye). The reprojected image has a new perspective but inevitably contains some disocclusion. Therefore, we use the **Patcher** to fill in the disocclusion (detailed next).

7 PATCHER

As shown in Figure 5, the reprojected image inevitably contains some disocclusions, missing some information/details. The problem of filling in the missing information of an image is called image patching or image inpainting. Existing solutions for image patching can be divided into the following five main categories: (1) ReRendering renders the disocclusion region via traditional renderer and scene information [66]. This approach still needs to compute the geometric information twice, and necessitates ray tracing which causes heavy computation overhead, thus making it ineffective for mobile VR. (2) Iterative approach applies optimization algorithms on the image and fill in the information by continuously iterating and optimizing the pixels of the image [49]. Therefore, it can not process real-time tasks and, naturally, is not suitable for this work. (3) Sequential method needs to iterate through all the unpatched pixels in one thread [22, 62]. Besides, it cannot be parallelized due to the read/write conflicts and thread safety (see Consideration C1&C2, Section 5), not suitable for improving efficiency. (4) Data-driven neural network models [30] are trained with large amounts of data and then empirically fill in the missing information in the images. They require a large memory space, computational power, and massive training data, and their generalization capability remains a concern. (5) Filter-based approach can be parallelized and requires a relatively small amount of memory compared to other image patching categories. However, this approach still entails onerous computation, unsuitable for mobile VR [50, 68]. It is important to note that ReRendering and Iterative approaches can not work well on mobile VR and are thus excluded in the evaluation.



Figure 5: (a) is the unpatched input image and the depth map. (b) is the patched image. (c) is the rasterized left ground truth image.

To enhance the image quality, we design YORO's **Patcher**, a new filter-based approach, to fill the image details. We note that the existing filter-based methods overlap the kernel range with the disocclusion region when sampling the target pixel, repeatedly reading pixels with zero information that does not contribute to the final result. The traditional kernel designs also read and calculate foreground pixels, leading to incorrect results. Therefore, we store the disocclusion information in advance in the previous **Reprojector** stage (see Section 6 **(b) Disocclusion Tracking**). The information contains the location of the nearest non-disocclusion pixel and the width of the disocclusion. This allows us to quickly

determine the kernel starting position and reduce the waste of texture reading operations.



Figure 6: The illustration of YORO's Patcher's kernel. Since the pixels are shifting right, we only sample toward left. Using ImBuffer we can quickly determine the starting location of the kernel without wasting samples on pixels that do not contribute to final results. The foreground pixels can also be skipped since the disocclusion comes from the background.

As shown in Algorithm 3, YORO's **Pathcer** is lightweight (~ 20 texture memory access per pixel compared to ~ 2500 access from Xiao et al. [68]) and parallel per pixel. For each pixel, it first checks if the pixel is disocclusion (line 3). If not, then return the color sampled from the full-resolution renderer image of the rendered view, using the location provided by the ImBuffer. Since the ImBuffer is downsampled, we use UV-Coordinate samplers where linear interpolation is automatically applied. If the current pixel is disocclusion (line 5), we accumulate the values of all pixels within the kernel and apply the weights. We skip the foreground according to **Observation O2** in Section 5 by checking the depth of pixel candidates (line 8). The kernel has the same width as the disocclusion at the current row and height of h (h = 3 by default). The weights W are calculated by:

Algorithm 3 Patcher (Image Effect Shader, Parallel per pixel)

 Input: I: Downsampled intermediate buffer. C: RGB color buffer of renderer eye. uv: The resolution-independent pixel coordinate ranges (0,1). Provided by the shader.
 Output: Final full-resolution patched image at coordinate uv

1:	$r \leftarrow I[uv]$	⊳ Intermediate Info
2:	$u, v \leftarrow uv$	
3:	if $r.z == 0$ then	If isn't disocclusion
4:	return $C[(r.x, v)]$	▹ return sampled color
5:	else	▹ If is disocclusion
6:	$c_{acc}, w_{rem} \leftarrow 0, 1$	
7:	for each pixel j in kernel from $(r.x - $	-r.z:r.x) do
8:	if $I[j]. y > r. y$ then	
9:	Continue	Skip foreground pixels.
10:	end if	
11:	$w \leftarrow W(u, r, w_{rem})$	
12:	$c_{acc} \leftarrow c_{acc} + C[j] \cdot w$	
13:	$w_{rem} \leftarrow w_{rem} - w$	
14:	end for	
15:	return $c_{acc}/(1 - w_{rem})$	
16:	end if	

$$W(\boldsymbol{u},\boldsymbol{r},\boldsymbol{w}) = \frac{\boldsymbol{w}}{2} + 0.3\,\boldsymbol{w} \times (\frac{\boldsymbol{u} - \boldsymbol{r}_x}{\boldsymbol{r}_z}),\tag{6}$$

where u is the coordinate provided by the shader, r is the intermediate info, w is the remaining weight. The kernel generation is visualized in Figure 6.

8 IMPLEMENTATION

YORO Implementation. YORO was implemented as a modular and highly parallelizable framework tailored to operate seamlessly across various mobile VR platforms. The development process involved addressing unique challenges, such as the efficient implementation of compute shaders and image shaders, which differ significantly from CPU-based implementations. Additional challenges included adapting to diverse hardware architectures and platforms, optimizing computational efficiency.

Given the varying levels of support for compute shaders and VR frameworks across platforms, we made platform-specific adjustments to YORO. On PC, we utilized the Mock HMD framework in conjunction with the Vulkan API. On mobile platforms, we employed Google Cardboard and the OpenGL API, while on Oculus Quest, we employed the Quest framework with the OpenGL API.

We have adapted and tested our implementation on seven representative different mobile VR devices (six smartphone VR and one standalone VR) across different mainstream mobile platforms (see Table 2). Considering the high integrity of mobile devices, we employ the Android Battery Historian [19] to measure power consumption in a non-invasive way.

Implementation of Baselines: We have implemented two group of VR optimization systems on the respective mobile platforms as the baselines.

(1) End-to-End Systems: (i) We consider the conventional binocular rasterizer as the baseline (Ground Truth, GT) for the entire VR system evaluation. The renderer parameters are the default parameters of the Unity engine, where the rendering path is set to Deferred Shading [33]. (ii) Besides, the state-of-the-art solution from Fink et al. [28] and SPI [60] are also employed. Fink et al. did not publicize the source code, and some technical details in the paper are unclear. Thus, we reproduce its work following the descriptions in [28]. SPI is implemented using Unity XR's built-in APIs.

(2) Patcher Systems: We employ three types of patcher algorithms for comparison, including four handcrafted algorithms and one deep-learning-based algorithm. (i) Filter-based: Median Filter [50] and Xiao et al. [68]; (ii) Sequential: Navier-Stokes [22] and Telea [62] with the default parameters and are ported to mobile devices with OpenCV for Unity [18]; (iii) Data-driven: The Generative Adversarial Network (GAN) style (pix2pix-based) [37]. The model is pre-trained under default settings for each scene on the dataset and is deployed on mobile devices via Unity Barracuda [63]. It is worth mentioning that Xiao et al. (2022) [68] initially implemented CUDA, targeting PC VR and running on the PC GPU. We have thus translated its open-sourced CUDA code to mobile-capable shader code.

9 EVALUATION SETUP

VR Scene Preparation: To verify the effectiveness of our YORO system, we selected eight representative 3D VR scenes with varying rendering styles and complexity. VR application rendering styles can be categorized into realistic shading, which aims for photo-realistic images using high-frequency textures, complex lighting, and detailed models, and stylized shading (e.g. cartoon, anime), which uses simpler textures and models. We chose comprehensive scenes covering both rendering styles (Figure 9), including those developed by us and from public archives [9, 48], depicting typical

MobiSys '25, June 23-27, 2025, Anaheim, CA, USA

Xingyu Chen, Xinmin Fang, Shuting Zhang, Xinyu Zhang, Liang He, and Zhengxiong Li

environments and objects like humans, plants, vehicles and buildings. The triangle counts of the scenes ranged from 22k to 2,833k, representing different levels of complexity. This diversity of scenes allows thorough testing of YORO and baselines.

Evaluation Metrics: We evaluate the YORO's optimization performance on frame rate improvement, energy cost, and CPU overhead using percentages where the conventional rendering is ground truth. In addition, we evaluate the stability by analyzing the standard deviation of frame rate and memory usage. Besides, we evaluate the image quality of the binocular image generated by YORO using Structural Similarity (SSIM) and Peak Signal Noise Ratio (PSNR) as the evaluation metrics. SSIM has gualified to reflect the subjective quality perception of the human eye. PSNR is a widely used image quality evaluation metric, as a higher value means less distortion. The image quality is widely recognized as excellent and acceptable if the image's SSIM reaches over 0.95 among [0,1] and PSNR reaches over 20.0 [42, 64]. SSIM is denoted $\frac{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)},$ where x and y show the as: SSIM(x, y) =the baseline image (GT here) and the processed image respectively, μ_x is the average of x, μ_y is the average of y, σ_x^2 is the variance of x, σ_y^2 is the variance of y. $c_1 = (k_1 L)^2$, $c_2 = (k_2 L)^2$ are the constants used to maintain stability. L is the dynamic range of the pixel values. $k_1 = 0.01$, $k_2 = 0.03$. PSNR is denoted as: $PSNR = 10 \log_{10}(\frac{(2^n - 1)^2}{MSE})$, where MSE is the mean square error between the baseline image (GT here) and the processed image.

10 SYSTEM PERFORMANCE EVALUATION

We evaluate YORO with the **End-to-End System Evaluation** baselines and metrics in Section 9 on all six representative smartphone VRs (to keep the similar VR type) in all eight VR scenes mentioned in Section 9, unless specified otherwise. Particularly, SPI is also tested in Section 10.6 due to its limitation.

10.1 Power Consumption

To measure the power consumption on these mobile devices, we test each device lasts for 10 minutes for each VR scene. We include a Meta Oculus Quest 2 in this evaluation. As shown in Figure 7(a), YORO's power consumption keeps lower than the conventional and Fink's ones in all VR scenes. The lowest power consumption is achieved among scene *LostEmpire*, which is 27% lower than the conventional one. The average power consumption is only 73% of the conventional one (a reduction of 27%), reaching 340 mW. For comparison, the power consumption of watching a video is 335 mW, and browsing a website is 380 mW, which is measured in the same condition. The power consumption on mobile VR is slightly higher than these common workloads, which fits the user experience and further illustrates the necessity to save energy in mobile VR. These results prove YORO has the ability to provide extended battery life and even support sustainability in computing.

10.2 Frame Rate

In this section, we evaluate YORO's performance in frame rate, as shown in Figure 7(b). The Meta Quest 2 is included in this test. YORO's frame rate is higher than the conventional and Fink et al. in all eight scenes. The performance of the conventional one and Fink et al. are equivalents. It is worth mentioning that in the



Figure 7: (a) The power consumption compared with the conventional and Fink et al. in different VR scenes. (b) The frame rate compared with the conventional and Fink et al. in different scenes.

low-complexity scenes (e.g., UnityChan), the frame rate of YORO is around 115-125% higher than the conventional process. While, in the high-complexity scenes, like *Bistro* and *SciFiCity*, the frame rates are 97.4% and 107.5% higher, respectively. The highest frame rate we get happens in the scene *Conference* (53.2 FPS), which is 119.4% higher than the conventional one. The average frame rate is 115.2% higher, proving YORO the ability to improve the frame rate under various conditions. Besides, YORO can achieve frame rate improvements beyond the theoretical computational complexity analysis in Section 10.3 (100%) on specific mobile devices. This is related to buffering and scheduling in the mobile GPU. Especially for binocular image rendering, extra bit block transfer (BitBilt) operations and synchronization between the two eyes are required.

10.3 Computational Complexity

Since raster rendering requires traversing all triangles in the scene and projecting them into screen space, this step is one of the main bottlenecks in rendering (see Section 3.1). With a consistent shading style, the rendering time complexity can be formulated as O(n) = N, where N is the number of scene triangles. Since we only need to render once, compared to the complexity T(n) = 2Nfor conventional binocular rendering, the complexity of YORO is T(n) = N + C, C << N where C represents the time required for projection and patching. It is a constant time for a given resolution, regardless of the scene complexity. In general, the overall computation amount for YORO is about half (50%) of the conventional method. While SPI can only reduce about 25% of computation on the CPU side and about 5% on the GPU side [5]. Thus, YORO can perform less computation and achieve better efficiency and energy saving.

Besides the theoretical analysis, to further illustrate this reduction, we compare the computational time consumption with the conventional one, Fink et al., and SPI solutions under a controlled experiment with linearly increased scene complexity. The results



Figure 8: The evaluation of the computational time consumption regarding the scene complexity between YORO and other existing approaches.

are shown in Figure 8. As the scene complexity grows, the time consumption of the conventional one grows more significantly than YORO. The results are consistent with our derivation. The computational complexity of YORO is almost half (51.40%) of the conventional one on average, while Fink et al. and SPI are 81.97% and 75.96% of the conventional one, respectively. Therefore, the results match the theoretical analysis, showing our method has excellent optimization regarding the VR scene complexity.

10.4 Image Quality

To evaluate the image quality produced by YORO, we generate images of the left eye from the right eye utilizing YORO. As shown in Table 3 and Figure 9, the average image quality reached 0.97 for SSIM and 34.09 for PSNR (over 0.95 and 20.0 thresholds, respectively, see Section 9), where the standard deviation of SSIM is less than 0.02, and the PSNR is less than 4.12. The image quality reaches the best in scene Sponza (SSIM: 0.98, PSNR: 41.90). Besides, the generation of each binocular image by YORO is 53.1% faster than conventional on average. The generated image of the left eye is close to the ground truth in most scenes. Additionally, to better illustrate the performance under extreme conditions, the zoom-in images in Bistro, Prototype and UnityChan are shown in Figure 9. Generally, only a few pixels are incorrect because the objects are too close to the user's eyes (< 50 cm), and the view image is highly complex (more discussion in Section 14). However, these extreme situations rarely happen in practical VR applications [53, 55]. These results prove that YORO can generate binocular images with high image quality, providing an excellent immersive experience.

10.5 System Overhead

We evaluate the system overhead of YORO in controlled conditions by measuring the impacts on mobile CPU and temperature. The frame rate and ambient temperature factors are controlled to the



Figure 9: System performance evaluation of Binocular image generation methods by YORO. GT is Ground Truth (conventionally rendering twice).

same before each test. The resulting mobile CPU overhead and temperature performance are shown in Figure 10. YORO's average CPU overhead and temperature are lower than the conventional and Fink et al. in all VR scenes. The lowest CPU overhead is achieved in scene *LostEmpire*, which is 24.05% lower than the conventional process. Besides, YORO keeps the temperature lower than 40°C in all scenes and reaches the lowest in scene *Chat*, proving YORO the ability to reduce the computation amount and the VR application



Figure 10: (a) The CPU overhead compared with the conventional and Fink et al. in different scenes. (b) The temperature of the mobile device compared with the conventional and Fink et al. in different scenes.

10.6 Comparison with Single Pass Instancing

In this section, we compare YORO with SPI, a representative and production-ready VR optimization method [60]. As mentioned in Section 1, SPI has limitations in finishing some regular tasks and even crashes under those conditions, due to unsupported postprocessing methods and the number of light sources. Although this has already proven the limited adaptability of SPI, we still try to compare the optimization performance of YORO and SPI. Therefore, we set up a group of additional experiments where VR scenes have a smaller number of light sources (<5) and just basic post-processing to meet the limitations of SPI (Note: this setting for SPI is not practical for VR applications that may require advanced/custom visual effects or a certain amount of light sources, such as for design, tourism, training, education, and healthcare). Additionally, these experiments are performed on the same devices shown in Table 2.

As shown in Figure 11(a), SPI can provide a frame rate boost compared to the conventional one, but the average amount of frame rate boost is inferior to YORO. The average frame rate of SPI is 57.76 FPS, while the YORO is 68.21 FPS. Besides, we observe that SPI shows strong instability in optimization performance across different devices and rendering styles. Especially, SPI performs worse than conventional on some devices (e.g., Samsung S10 and Samsung S21), probably due to compatibility issues on the Samsung platform (14.8% lower). While YORO performs reliably and consistently in this regard. Moreover, the results also illustrate that SPI causes 35.8% more power consumption, 63.4% more CPU usage (Figure 11(b)), and 2.9°C higher average temperatures than YORO, further highlighting the superiority of YORO.

11 ROBUSTNESS EVALUATION

11.1 Performance on Different Devices

To illustrate if YORO can work pervasively across the different mobile platforms, we conduct the following experiments. The evaluation results are shown in Table 2. The frame rate increases and

Table 2: YORO Performance on different mobile devices across different mobile platforms. YORO can generally reduce the amount of computation, increase efficiency, and save energy among representative mobile VR devices over the conventional approach.

Mobile VR Device Type	Honor 8	Honor 30	Honor 30s	Samsung Galaxy S10	Samsung Galaxy S21 5G	Xiaomi MI 6	Meta Quest 2	Meta Quest 3
Device RAM	3 GB	8 GB	8 GB	8 GB	8 GB	6 GB	6 GB	8 GB
Battery Capacity	3000 mAh	4000 mAh	4000 mAh	3400 mAh	4000 mAh	3350 mAh	3636 mAh	5050 mAh
Screen Size	5.2 inches	6.53 inches	6.5 inches	6.1 inches	6.2 inches	5.15 inches	N/A	N/A
Power reduction (mW)	-35%	-24%	-25%	-31%	-15%	-49%	-6%	-4%
Frame rate increased (FPS)	+149.64%	+142.65%	+151.51%	+169.77%	+133.57%	+139.31%	+49.03%	+22.63%
Execution time reduced (ms)	-59.94%	-58.79%	-60.24%	-62.93%	-57.19%	-58.21%	-32.87%	-17.49%
Memory reduced (MB)	-2.13%	-2.33%	-2.26%	-4.64%	-4.57%	-4.55%	-1.26%	-2.90%

Table 3: A comparison of binocular VR image generation between YORO and the convention method (GT). The PSNR for GT is N/A since the MSE for itself is 0.

	Scene Name	Time	Memory	PNSR	SSIM
	Bistro	28.6ms	52.21 MB	31.99	0.9505
	SciFiCity	25.4ms	61.73 MB	29.58	0.9516
	Conference	18.8ms	60.34 MB	34.68	0.9805
YORO	Sponza	19.8ms	51.30 MB	41.90	0.9802
(Ours)	LostEmpire	20.8ms	50.50 MB	39.25	0.9747
	Prototype	19.6ms	56.10 MB	32.98	0.9751
	Chat	19.4ms	70.20 MB	29.79	0.9697
	UnityChan	20.2ms	61.78 MB	28.74	0.9643
	Average	21.6ms	58.02 MB	34.09	0.9679
GT	Average	46.1ms	60.51 MB	N/A	1



Figure 11: (a) The frame rate compared with the conventional, Fink et al., and SPI in different scenes. (b) The CPU overhead compared with the conventional, Fink et al., and SPI in different scenes.

the power consumption drops on all mobile devices compared to the conventional one. Besides, YORO shows superior performance on devices with early-released mobile devices (e.g., Samsung Galaxy S10), where the frame rate increases 169.8% over the conventional one. Additionally, towards the flagship mobile VR devices (e.g., Meta Oculus Quest 2), YORO improves FPS by about 49.03%. Moreover, the power consumption for all devices decreases significantly. These results show YORO can smoothly work on a variety of mobile devices without any extra hardware or software modification.

11.2 Adaptive Shading Style

Different VR applications usually have various rendering styles. To ensure YORO can be used as a pervasive optimization algorithm, we evaluate YORO's image quality under different shading styles. As shown in Figure 12(a), the five most representative and typical shading styles are recruited. The average SSIM image quality is 0.9679, and the average standard deviation is lower than 0.02, proving YORO is capable of adapting various shading styles in most VR applications.



Figure 12: (a) The SSIM and PSNR of different shading styles. (b) The SSIM and PSNR with different patching methods.

11.3 Adjustable Interpupillary Distance

Adjustable Interpupillary Distance (IPD) is an essential part of commercial VR products since users have different interpupillary distances. The IPD is the distance between the two lenses of VR HMD. We evaluate YORO's image quality at different IPD from 5.0 to 7.5 cm, the eye distance range of normal adult humans [26]. As shown in Figure 13(a), the average SSIM is 0.963, and the average standard deviation is lower than 0.02. The average PSNR is 31.31, with an STD of 3.87. The results show that YORO has the ability to adjustable IPD and broad applicability.



Figure 13: (a) The SSIM and PSNR of different IPD. (b) The SSIM and PSNR of different object distance.

11.4 Impact of Object Distance

As discussed in Section 10.4, some close objects are not shown correctly. Since the view field of the human eye is a perspective frustum (i.e., the closer the object is, the less pixel information can be reused), subsequently, YORO inevitably has a limitation in rendering near objects. The minimum object distance for YORO is: $D = \frac{I}{2} \times \cot(\frac{\theta}{2})$, where *D* is the minimum distance, *I* is the IPD distance, and θ is the field of view. *D* by default (I = 0.075, $\theta = 60$) is calculated at 0.129 meter. However, regular users will not encounter objects in VR so closely, and it rarely happens in real practice.

UnityChan SciFiCity Bistro LostEmpire VR Scene (Reference)

Figure 14: The error magnitude at each pixel (1-SSIM [28]) from YORO and Patcher Evaluation baselines are evaluated for image quality.

Besides, the minimum practical distance in most VR applications (e.g., supported by Unity Engine) is 0.3 meters. We conduct a micro bench on the impact of object rendering distance. We chose an irregular object (i.e., the bunny) as the subject and placed it at a distance from 0.5 to 3 meters. The object's scale is automatically adjusted to maintain a consistent size across screens. The results are shown in Figure 13(b). YORO keeps satisfactory image quality of 0.95 SSIM and 38.73 PSNR, showing only a slight drop within a distance of one meter. Moreover, to solve this extreme case, we propose using a hybrid rendering approach by sacrificing energy saving ability, using the conventional approach to render the near objects (< 0.129 meters), and YORO for the rest of the objects.

11.5 Patcher Evaluation

To evaluate if our Patcher can repair the disocclusion of the image, we recruit five algorithms as baselines (details see Section 8). As shown in Figure 12(b) and Table 4, YORO has the highest efficiency, with the patching time of each frame roughly 0.3ms, which is faster than other algorithms. In addition, YORO's SSIM keeps above the threshold of 0.95 and PNSR over 20.0, providing excellent image quality and immersive user experience. Moreover, it is worth mentioning that although Median Filter [50] and Xiao et al. [68] (Filter-Based type) keeps patching time at around 1ms and achieve high FPS (around 100 FPS), their image quality is somewhat limited, may not sufficient in mobile VR applications. Besides, although the patching execution time of Navier-Stokes [22] and Telea [62] (Sequential type) are at 10-20ms, they have limitations in working parallel that restrict the whole rendering process. They can only support around 3 FPS when applied to VR products, which is far less than 30 FPS, and it is not practical on mobile VR.

Besides, the image quality comparison via error magnitude (1-SSIM) [28] is shown in Figure 14. Bright pixels in the image represent the difference compared to the ground truth. Generally, fewer bright pixels in the image and better patching performance achieve. The results show YORO achieves equal or better patching performance than other baselines.

12 REAL-WORLD EVALUATION

To reflect the actual performance, we evaluate YORO in Meta Oculus Quest 2, a commercial Mobile VR device. Meta Oculus Quest 2 is



Figure 15: (a, b) are the FPS performance comparisons with YORO, the conventional, Fink et al. and SPI in *Unity VR Sample* [4] and *The Escape Room* [11], respectively. Bellows are the typical scene view of (1) *Unity VR Sample* and (2) *The Escape Room*.

currently the most potent and popular dedicated standalone mobile VR device with high resolution (1832x1920 pixels per eye) and high frame rate (90 FPS in default mode, 120 FPS in experimental mode). This evaluation includes both local and streaming VR modes in real practice.

Local VR application: To prove their feasibility, we apply YORO to actual VR products. We employ two well-recognized official public-accessible VR applications, as shown in Figure 15: (1) Unity VR Sample [4] and (2) The Escape Room [11]. We recompile the selected applications with YORO, install them into Oculus Quest 3, and make them work in the experimental mode. The results of the graphic profiler are shown in Figure 15. The results show that YORO can boost the frame rate of The Escape Room by 32.1% than the conventional (YORO: Average 82.00 FPS, STD 23.39; the Conventional one: Average 62.09 FPS, STD 24.49; Fink et al.: Average 73.09 FPS, STD 26.82; SPI: Average 62.68 FPS, STD 30.98), and boost the frame rate of Unity VR Sample by 23% (YORO: Average 116.96 FPS, STD 8.60; the Conventional one: Average 95.36 FPS, STD 22.81; Fink et al.: Average 82.21 FPS, STD 18.82; SPI: Average 113.54 FPS, STD 15.95). The results show YORO can support VR APP products working at higher and more stable frame rates compared with other baselines, even reaching 110-120 FPS (Note: 120 FPS is the current highest refresh rate for mainstream mobile screen products).

13 USER STUDY ON IMMERSIVE EXPERIENCE

To evaluate the overall user experience and image quality, we recruited 14 participants, aged between 20 and 32 years, following the Institutional Review Board (IRB) protocol. Each participant viewed eight VR scenes using both YORO and the conventional method (GT), following a random order. Participants are instructed to move freely around the scene, interacting with objects as they would in a typical game experience. They are also encouraged to move and rotate rapidly to test the frame rate and latency. Participants rated

 Table 4: Performance comparison of representative image inpainting/patching algorithms on mobile devices.

Approach	Patching Time	FPS	Memory	PNSR	SSIM
Xiao et al.	1.02ms	90.1	76.81MB	29.21	0.9170
Median	0.45ms	113.0	76.74MB	31.01	0.9516
NS [22]	28.0ms	11.1	79.88MB	31.96	0.9539
Telea [62]	48.8ms	6.5	79.89MB	32.25	0.9542
GAN [37]	37,142ms	0.03	165.39MB	28.07	0.9413
YORO	0.35ms	115.9	73.30MB	34.09	0.9679

the image quality and user experience using a continuous slider scale from 1 to 5 points.

We conducted a **paired t-test** [57] to compare the image quality and user experience ratings between the two methods. The null hypothesis assumed no difference between the methods. For **Image Quality**, the t-statistic was 0.6196, with a p-value of 0.5462, an effect size (Cohen's d_z) of 0.1656, and a statistical power of 0.0887. For **User Experience**, the t-statistic was 0.2946, with a p-value of 0.7730, an effect size (Cohen's d_z) of 0.0787, and a statistical power of 0.0586. Given these results, the paired t-test does not provide sufficient evidence to reject the null hypothesis, *implying no significant difference between the two methods in terms of image quality and user experience.*

However, the low values of effect size and statistical power are noteworthy. A small effect size indicates the difference observed between the samples is practically insignificant. The low statistical power value suggests a risk of a Type II error, meaning the test might fail to reject a false null hypothesis. Thus, the paired t-test may not be sensitive enough to detect a difference if it exists. To further evaluate the rating scores, we conducted an equivalence test, specifically Two One-Sided Tests (TOST) [39], to determine if the performances of YORO and the conventional method are similar.

For **Image Quality**, the lower p-value was 1.1343×10^{-9} and the upper p-value was 3.4048×10^{-9} , with a confidence interval (CI) of (-0.0559, 0.1008). For **User Experience**, the lower p-value was 1.1266×10^{-8} and the upper p-value was 2.0626×10^{-8} , with a CI of (-0.0802, 0.1055). All p-values were clearly within the prespecified significance level ($\alpha = 0.05$), and both CI results fell entirely within the equivalence bounds ([-0.5, 0.5]). These results indicate that the two methods can be considered equivalent, and YORO and the conventional method achive similar performance in terms of image quality and user experience.

Moreover, participants' feedback supported these findings, with some noting no difference in image quality and others preferring aspects of YORO, such as smoother frame rates and sharper textures. The user study confirms that YORO offers a comparable immersive experience to the conventional method in real-world scenarios.

14 DISCUSSION

Post-processing: Post-processing techniques, including bloom, screen space ambient occlusion (SSAO), anti-aliasing, etc. are applied in the evaluation by default. These representative techniques can provide more necessary functionalities and a better user experience, which have been widely used in medical, construction, gaming, and other VR applications. Besides, through these settings, the evaluations can reflect the real performance of YORO and other related work comprehensively. In addition, post-processing techniques are applied before YORO module; therefore, no alphablending is involved during the projection stage. It is also worth noting that the transparent geometry is not discussed in YORO. Mobile VR barely supports transparent geometry practically since the transparent geometry requires an extra forward pass to render, and this extra effort will severely affect immersive performance [14]. But for future work in other working scenarios, YORO can

still work with transparency objects by inserting the YORO module before the transparent pass with further modification of the renderer pipeline.

OpenGL Dependencies: To achieve the algorithms working parallel, we use the Compute Shader to implement them on mobile GPU. The Compute Shader runs on devices supporting at least OpenGL Version 3.1, which was introduced in 2009 and is now supported by most mobile VR devices [1].

Image Quality: During the evaluation, we observed that only a small portion, approximately 1%, of the frame region, shows observable differences from the ground truth. One substantial difference (around 22% among this 1%) occurs when the object distance is less than 0.5 meters. However, this distance happens rarely in mainstream VR applications [53, 55]. Even if it occurs, the VR device can revert back to the conventional binocular renderer. In addition, other slight differences (around 78% among this 1%) are attributed to minor shading variations between the two viewing angles, which would not impact the user's viewing experience in a noticeable way. Immersive Content Streaming: Our approach can be viewed as a form of semantic compression, where the extracted semantic information is primarily depth data. thereby provides significant advantages for VR streaming applications. While traditional binocular VR streaming requires six-channel buffers (two RGB images), our method needs only a four-channel buffer (one RGB image plus depth), reducing data volume by 33% theoretically and achieving 39.6% reduction in practical experiments with H.264 encoding. These results demonstrate our method's effectiveness in data transmission reduction, with practical savings exceeding theoretical predictions.

Future Applications: YORO can work on the driver layer, allowing it to integrate seamlessly with the operating system and support the wide use on VR applications.

15 CONCLUSION

In this paper, we designed and implemented a novel mobile VR optimization method, YORO, a post-processing-based technique to generate binocular VR images via rendering once. Compared to conventional VR rendering methods, YORO saves 27% in power consumption and improves frame rates by 115.2%. In addition, YORO can also be used as a rendering method for streaming VR videos, capable of saving 39.6% of data transmission. We extensively evaluated YORO on comprehensive and representative multiple VR scenes with different scene complexity and shading styles with six smartphone VRs and one standalone VR. The results reflect the feasibility of this practical mobile VR optimization approach across different platforms and the superior performance in high efficiency and energy saving, which pave the road to the next mobile VR development.

ACKNOWLEDGMENTS

We appreciate the insightful comments and feedback from the anonymous reviewers and shepherd. This work is partially supported by the NSF under Grants CNS-2426470, CNS-2312715, CNS-2403124, CNS-2128588 and Center for Wireless Communications at UC San Diego.

REFERENCES

- [1] OpenGL. https://www.opengl.org, 1997.
- [2] Accumulation buffer support? https://forum.unity.com/threads/accumulationbuffer-support.7203/, 2007.
- [3] Deferred shading. https://learnopengl.com/Advanced-Lighting/Deferred-Shading, 2015.
- [4] Get started with vr! sample pack learning articles. https://blog.unity.com/games/get-started-with-vr-sample-pack-learningarticles, 2015.
- How to maximize ar and vr performance with advanced stereo rendering. https://blog.unity.com/technology/how-to-maximize-ar-and-vr-performancewith-advanced-stereo-rendering, 2017.
- [6] Asynchronous reprojection | google vr | google developers. https://developers.google.com/vr/discover/async-reprojection, 2018.
- [7] Opengl 4.6. https://www.khronos.org/registry/OpenGL/specs/gl /glspec46.core.pdf, 2018.
- [8] The importance of frame rates irisvr. https://help.irisvr.com/hc/enus/articles/215884547-The-Importance-of-Frame-Rates, 2019.
- [9] Synty studios asset store. https://assetstore.unity.com/publishers/5217, 2019.
- Multi-view: Oculus developers. https://developer.oculus.com/ documentation/native/android/mobile-multiview/, 2020.
- [11] VR beginner: The escape room: Tutorials. https://assetstore.unity.com/packages/templates/tutorials/vr-beginner-theescape-room-163264, 2021.
- [12] Application spacewarp developer guide: Native: Oculus developers, 2022.
- [13] Augmented reality and virtual reality market. https://www.marketsandmarkets.com/Market-Reports/augmented-realityvirtual-reality-market-1185.html, 2022.
- [14] Material and shader best practices for artists. https://developer.arm.com/documentation/102576/0100/Transparency-bestpractise: :text=Rendering%20an%20object%20with %20transparency.of%20one%20another%20many%20times.. 2022.
- [15] Pimax: The world's first 8k vr headset, Nov 2022.
- [16] Testing and performance analysis: Oculus developers. https://developer.oculus.com/documentation/unity/unity-perf/, 2022.
- [17] Vr performance for mobile devices. https://ltu.divaportal.org/smash/get/diva2:1112571/FULLTEXT01.pdf, 2022.
- [18] OpenCV for unity: Integration. https://assetstore.unity.com/packages/tools/ integration/opencv-for-unity-21088, 2023.
- [19] Profile battery usage with batterystats and battery historian. https://developer.android.com/topic/performance/power/setup-batteryhistorian, 2023.
- [20] Rendering with opengl psychxr latest documentation. 2023.
- [21] Kanchan Bahirat, Chengyuan Lai, Ryan P McMahan, and Balakrishnan Prabhakaran. A boundary and texture preserving mesh simplification algorithm for virtual reality. In Proceedings of the 8th ACM on Multimedia Systems <u>Conference</u>, pages 50–61, 2017.
- [22] Marcelo Bertalmio, Andrea L Bertozzi, and Guillermo Sapiro. Navier-stokes, fluid dynamics, and image and video inpainting. In Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. <u>CVPR 2001</u>, volume 1, pages I–I. IEEE, 2001.
- [23] Kevin Boos, David Chu, and Eduardo Cuervo. Flashback: Immersive virtual reality on mobile devices via rendering memoization. In <u>Proceedings of the 14th Annual</u> International Conference on Mobile Systems, Applications, and Services, pages 291–304, 2016.
- [24] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In <u>Proceedings of the 1997 symposium on Interactive 3D graphics</u>, pages 83-ff, 1997.
- [25] Vittorio Cozzolino, Leonardo Tonetto, Nitinder Mohan, Aaron Yi Ding, and Jorg Ott. Nimbus: Towards latency-energy efficient task offloading for ar services. IEEE Transactions on Cloud Computing, 2022.
- [26] Neil A Dodgson. Variation and extrema of human interpupillary distance. In <u>Stereoscopic displays and virtual reality systems XI</u>, volume 5291, pages 36–46. International Society for Optics and Photonics, 2004.
- [27] Budmonde Duinkharjav, Kenneth Chen, Abhishek Tyagi, Jiayi He, Yuhao Zhu, and Qi Sun. Color-perception-guided display power reduction for virtual reality. ACM Transactions on Graphics (TOG), 41(6):1–16, 2022.
- [28] Laura Fink, Nora Hensel, Daniela Markov-Vetter, Christoph Weber, Oliver G Staadt, and Marc Starnrninger. Hybrid mono-stereo rendering in virtual reality. In <u>VR</u>, pages 88–96, 2019.
- [29] Sebastian Friston, Tobias Ritschel, and Anthony Steed. Perceptual rasterization for head-mounted display image synthesis. <u>ACM Trans. Graph.</u>, 38(4):97–1, 2019.
- [30] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. Advances in neural information processing systems, 27, 2014.

- [31] Jawad Haj-Yahya, Jisung Park, Rahul Bera, Juan Gómez Luna, Efraim Rotem, Taha Shahroodi, Jeremie Kim, and Onur Mutlu. Burstlink: Techniques for energyefficient video display for conventional and virtual reality systems. In <u>MICRO-54:</u> 54th Annual IEEE/ACM International Symposium on Microarchitecture, pages 155–169, 2021.
- [32] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In <u>2016 IEEE International Symposium on High</u> <u>Performance Computer Architecture (HPCA)</u>, pages 64–76. IEEE, 2016.
- [33] Shawn Hargreaves and Mark Harris. Deferred shading. In <u>Game Developers</u> Conference, volume 2, page 31, 2004.
- [34] Robert Hirsch. Exploring colour photography: a complete guide. Laurence King Publishing, 2004.
- [35] Wumeng Huang, Jing Chen, and Mengyun Zhou. A binocular parallel rendering method for vr globes. <u>International Journal of Digital Earth</u>, 13(9):998–1016, 2020.
- [36] Muhammad Huzaifa, Rishi Desai, Samuel Grayson, Xutao Jiang, Ying Jing, Jae Lee, Fang Lu, Yihan Pang, Joseph Ravichandran, Finn Sinclair, et al. Illixr: Enabling end-to-end extended reality research. In <u>2021 IEEE International Symposium on</u> Workload Characterization (IISWC), pages 24–38. IEEE, 2021.
- [37] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In <u>Proceedings of the IEEE</u> conference on computer vision and pattern recognition, pages 1125–1134, 2017.
- [38] Zhihui Ke, Xiaobo Zhou, Dadong Jiang, Hao Yan, and Tie Qiu. Collabvr: Reprojection-based edge-client collaborative rendering for real-time high-quality mobile virtual reality. In <u>2023 IEEE Real-Time Systems Symposium (RTSS)</u>, pages 304–316. IEEE, 2023.
- [39] Daniël Lakens, Anne M Scheel, and Peder M Isager. Equivalence testing for psychological research: A tutorial. <u>Advances in Methods and Practices in</u> <u>Psychological Science</u>, 1(2):259–269, 2018.
- [40] Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In <u>Proceedings</u> of the 13th Annual International Conference on Mobile Systems, <u>Applications</u>, and Services, pages 151–165, 2015.
- [41] Yue Leng, Chi-Chun Chen, Qiuyue Sun, Jian Huang, and Yuhao Zhu. Energyefficient video processing for virtual reality. In <u>Proceedings of the 46th</u> <u>International Symposium on Computer Architecture</u>, pages 91–103, 2019.
- [42] Xiangjun Li and Jianfei Cai. Robust transmission of jpeg2000 encoded images over packet loss channels. In <u>2007 IEEE International Conference on Multimedia</u> <u>and Expo</u>, pages 947–950. IEEE, 2007.
- [43] Yong Li and Wei Gao. Deltavr: Achieving high-performance mobile vr dynamics through pixel reuse. In Proceedings of the 18th International Conference on Information Processing in Sensor Networks, pages 13–24, 2019.
- [44] Luyang Liu, Ruiguang Zhong, Wuyang Zhang, Yunxin Liu, Jiansong Zhang, Lintao Zhang, and Marco Gruteser. Cutting the cord: Designing a high-quality untethered vr system with low latency remote rendering. In <u>Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services, pages 68–80, 2018.</u>
- [45] David Luebke, Martin Reddy, Jonathan D Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. Level of detail for 3D graphics. Morgan Kaufmann, 2003.
- [46] William R Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3d warping. In Proceedings of the 1997 symposium on Interactive 3D graphics, pages 7–ff, 1997.
- [47] Ricardo Martin-Brualla, Rohit Pandey, Shuoran Yang, Pavel Pidlypenskyi, Jonathan Taylor, Julien Valentin, Sameh Khamis, Philip Davidson, Anastasia Tkach, Peter Lincoln, et al. Lookingood: enhancing performance capture with real-time neural re-rendering. <u>ACM Transactions on Graphics (TOG)</u>, 37(6):1–14, 2018.
- [48] Morgan McGuire. Computer graphics archive, July 2017. https://casual-effects.com/data.
- [49] Simon Niklaus and Feng Liu. Softmax splatting for video frame interpolation. In <u>Proceedings of the IEEE/CVF Conference on Computer</u> Vision and Pattern Recognition, pages 5437–5446, 2020.
- [50] H Noori and Saeid Saryazdi. Image inpainting using directional median filters. In <u>2010 International Conference on Computational</u> <u>Intelligence and Communication Networks</u>, pages 45–49. IEEE, 2010.
- [51] Daniel Pohl, Nural Choudhury, and Markus Achtelik. Concept for rendering optimizations for full human field of view hmds. In 2018 <u>IEEE Conference on Virtual Reality and 3D User Interfaces (VR)</u>, pages 663–664. IEEE, 2018.

MobiSys '25, June 23-27, 2025, Anaheim, CA, USA

Xingyu Chen, Xinmin Fang, Shuting Zhang, Xinyu Zhang, Liang He, and Zhengxiong Li

- [52] Andre Schollmeyer, Simon Schneegans, Stephan Beck, Anthony Steed, and Bernd Froehlich. Efficient hybrid image warping for high framerate stereoscopic rendering. <u>IEEE transactions on visualization and computer graphics</u>, 23(4):1332–1341, 2017.
- [53] Raunaq Shah. New to vr? here is what you must know to get started, Jan 2021.
- [54] Shu Shi, Varun Gupta, and Rittwik Jana. Freedom: Fast recovery enhanced vr delivery over mobile networks. In <u>Proceedings of the 17th</u> <u>Annual International Conference on Mobile Systems, Applications,</u> <u>and Services, pages 130–141, 2019.</u>
- [55] Takashi Shibata, Joohwan Kim, David M Hoffman, and Martin S Banks. The zone of comfort: Predicting visual discomfort with stereo displays. Journal of vision, 11(8):11–11, 2011.
- [56] Marco Speicher, Sebastian Cucerca, and Antonio Krüger. Vrshop: a mobile interactive virtual reality shopping environment combining the benefits of on-and offline shopping. <u>Proceedings of the ACM on</u> <u>Interactive, Mobile, Wearable and Ubiquitous Technologies</u>, 1(3):1–31, 2017.
- [57] Student. The probable error of a mean. Biometrika, 6(1):1-25, 1908.
- [58] Unity Technologies. Unity camera component. https://docs.unity3d.com/Manual/class-Camera.html, 2014.
- [59] Unity Technologies. Single pass stereo rendering. https://docs.unity3d.com/Manual/SinglePassStereoRendering.html, 2017.
- [60] Unity Technologies. Single pass instanced rendering. https://docs.unity3d.com/Manual/SinglePassInstancing.html, 2018.

- [61] Unity Technologies. Dynamic resolution. https://docs.unity3d.com/2020.3/Documentation/Manual/ DynamicResolution.html, 2022.
- [62] Alexandru Telea. An image inpainting technique based on the fast marching method. Journal of graphics tools, 9(1):23–34, 2004.
- [63] Unity-Technologies. Unity-technologies/barracuda-release. https://github.com/Unity-Technologies/barracuda-release, 2019.
- [64] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. <u>IEEE transactions on image processing</u>, 13(4):600–612, 2004.
- [65] Martin Weier, Michael Stengel, Thorsten Roth, Piotr Didyk, Elmar Eisemann, Martin Eisemann, Steve Grogorick, André Hinkenjann, Ernst Kruijff, Marcus Magnor, et al. Perception-driven accelerated rendering. In <u>Computer Graphics Forum</u>, volume 36, pages 611–643. Wiley Online Library, 2017.
- [66] Niko Wißmann, Martin Mišiak, Arnulph Fuhrmann, and Marc Erich Latoschik. Accelerated stereo rendering with hybrid reprojection-based rasterization and adaptive ray-tracing. In <u>2020 IEEE Conference on</u> Virtual Reality and 3D User Interfaces (VR), pages 828–835. IEEE, 2020.
- [67] Lei Xiao, Salah Nouri, Matt Chapman, Alexander Fix, Douglas Lanman, and Anton Kaplanyan. Neural supersampling for real-time rendering. <u>ACM Transactions on Graphics (TOG)</u>, 39(4):142–1, 2020.
- [68] Lei Xiao, Salah Nouri, Joel Hegland, Alberto Garcia Garcia, and Douglas Lanman. Neuralpassthrough: Learned real-time view synthesis for vr. In <u>ACM SIGGRAPH 2022 Conference Proceedings</u>, pages 1–9, 2022.