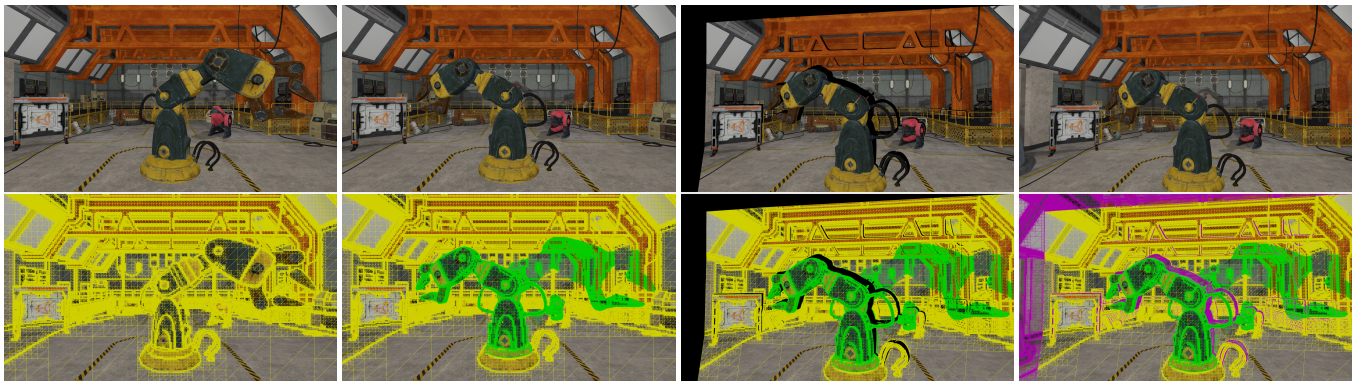
QUASAR: Quad-based Adaptive Streaming And Rendering

EDWARD LU, Carnegie Mellon University, USA
ANTHONY ROWE, Carnegie Mellon University, USA and Bosch Research, USA



(a) (b) (c) (d)

Fig. 1. We present a quad-based streaming system for dynamic scenes that dramatically reduces data transmission rates. Instead of sending entirely new sets of quads for each frame, our method selectively sends only the quads affected by scene changes and disocclusions. We first fit quads to a scene using a series of G-Buffers (**1a**, **yellow**), which are sent to a client to reconstruct and render. To reduce bandwidth usage, we selectively only transmit quads that capture scene geometry changes from a reference viewpoint (**1b**, **green**) and newly revealed regions due to potential disocclusions (**black** regions in **1c**). The client integrates these residual quads (**1d**, **green** and **magenta**) with the cached quads from earlier frames. Designed for latency masking in remote rendering, our approach captures motion parallax, disocclusion events, and transparency effects. Furthermore, it is more compact and efficient than previous methods, while being well suited for video codecs and adaptable to varying network bandwidth conditions.

As AR/VR systems evolve to demand increasingly powerful GPUs, physically separating compute from display hardware emerges as a natural approach to enable a lightweight, comfortable form factor. Unfortunately, splitting the system into a client-server architecture leads to challenges in transporting graphical data. Simply streaming rendered images over a network suffers in terms of latency and reliability, especially given variable bandwidth. Although image-based reprojection techniques can help, they often do not support full motion parallax or disocclusion events. Instead, scene geometry can be streamed to the client, allowing local rendering of novel views. Traditionally, this has required a prohibitively large amount of interconnect bandwidth, excluding the use of practical networks.

This paper presents a new quad-based geometry streaming approach that is designed with compression and the ability to adjust Quality-of-Experience (QoE) in response to target network bandwidths. Our approach advances previous work by introducing a more compact data structure and a temporal compression technique that reduces data transfer overhead by up to 15×, reducing bandwidth usage to as low as 100 Mbps. We optimized our design for hardware video codec compatibility and support an adaptive data streaming strategy that prioritizes transmitting only the most relevant geometry updates. Our approach achieves image quality comparable to, and in many cases exceeds, state-of-the-art techniques while requiring only a fraction of the bandwidth, enabling real-time geometry streaming on commodity headsets over WiFi.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Virtual reality**; Image-based rendering.

Additional Key Words and Phrases: Streaming, Compression, Texture-space shading, Object-space shading, Temporal coherence, Virtual reality

ACM Reference Format:
Edward Lu and Anthony Rowe. 2025. QUASAR: Quad-based Adaptive Streaming And Rendering. *ACM Trans. Graph.* 44, 4 (August 2025), 19 pages. https://doi.org/10.1145/3731213

1 INTRODUCTION

Remote rendering [Shi and Hsu 2015] aims to deliver high-quality 3D graphics to devices with limited resources by offloading rendering calculations and streaming the results to an end client. The main challenge of many untethered remote rendering systems is dealing with the latency and unreliability of typical networks. This leads to high motion-to-photon latency, negatively affecting Quality-of-Experience (QoE). For remote rendering, it is crucial to ensure that perceived latency remains low to avoid motion sickness.

Current remote rendering systems often utilize the speed and efficiency of built-in video codec hardware available on most devices. In typical designs, a high-end server receives a camera pose from a client, renders a high-quality image corresponding to that pose, and

Authors' addresses: Edward Lu, elu2@andrew.cmu.edu, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA, 15213, USA; Anthony Rowe, agr@andrew.cmu.edu, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA, 15213, USA and Bosch Research, 2555 Smallman St, Unit 301, Pittsburgh, PA, 15222, USA.

ACM 0730-0301/2025/8-ART
https://doi.org/10.1145/3731213

transmits the image as part of a video stream. When transmitted over a network, this can lead to the display of *late* frames on the client side. To mitigate this, many systems include an additional *reprojection* step, where the received frame is warped to the current viewpoint before being displayed. The simplest form of reprojection is Asynchronous Time Warping (ATW) [van Waveren 2016], where a homography is applied to the received frame. Commercial systems [Google 2019; Magic Leap 2024; Meta Platforms 2021, 2024; Microsoft 2020; NVIDIA 2020, 2023; Valve 2015] use a combination of close-by edge servers, server-side client pose prediction, and ATW to achieve low perceived latency. Although pose prediction systems have been shown to work under low network latencies (< 40 ms RTT) [Lee et al. 2015], more complex reprojection is required when latencies are higher. Although effective in compensating for rotational mismatches, ATW fails to capture the full *motion parallax* of a 3D environment, which becomes more apparent if the user moves quickly or if latencies are higher. More advanced image-based rendering techniques like Asynchronous Space Warping (ASW) [Oculus 2016] can account for motion parallax but fail to capture *disocclusions*, where occluded portions of a scene (e.g., geometry hidden behind other geometry) become visible under camera movements.

The problem can be formulated as follows: if the client pose is mispredicted—such that the server believes the client will be at pose P_s, but the client is actually at P_c when it receives the frame—the server must transmit enough information to fully compensate for rendering mismatches due to the pose error $|P_s - P_c|$. This can be thought of as preparation for an imaginary client-side *viewing volume* centered at P_s, where the client can be anywhere within the volume, and all potential disocclusions within the range should be fully supported. As a result, these systems often arrive at a bandwidth vs. reprojection quality trade-off where they transmit additional scene information to capture disocclusions and mask latency.

To address this, recent works [Hladky et al. 2021, 2019; Mueller et al. 2018; Voglreiter et al. 2023] have proposed sending both *visible* and *potentially visible* scene geometry along with intermediate shading results, usually stored in a packed texture atlas. These systems employ a "split" in the rendering pipeline, where the server performs expensive lighting calculations, while the client primarily handles vertex processing and texture mapping of incoming geometry, generating novel views with a forward rendering pass until the next server frame arrives. However, if a client is to have no *a priori* knowledge of the scene, then geometry must be sent in real time. As industry moves towards higher polycount scenes [Karis et al. 2021], it remains challenging to compress large amounts of scene data (e.g., triangles) to fit within standard bandwidth requirements. As a result, other approaches transmit an *approximated* form of the scene geometry, combining multiple potential viewpoints within the viewing volume. Notably, [Hladky et al. 2022; Lall et al. 2018] use a series of fitted and merged tiles (or *quads*) generated from a G-Buffer and projected into 3D. These approaches support client movement from within a head box and use overlapping quads to handle disocclusions. Although the transmitted data (typically quads defined by a normal and depth value) can be heavily quantized, a single frame can still consume a significant amount of bandwidth. In our experiments, these techniques can result in data rates on the order of 5 Gbps, which dramatically exceed most networks and are only applicable for custom board-level interconnects.

In this paper, we present several enhancements to quad/tile-based scene approximation and transmission that make it possible to stream geometry information to mobile AR/VR headsets. We support scenarios where network latencies and bandwidths are greater than 40 ms with hundreds of Mbps of throughput. This could be a local rendering machine with a wireless link or even a nearby cloud-hosted option.

We built on a quad-based approximation [Hladky et al. 2022; Lall et al. 2018] because it provides high visual quality and allows for distortion-free texture mapping from a video stream. To support commodity network links, we introduce a streaming scheme that leverages temporal coherence by transmitting only updated and newly revealed geometry caused by scene and viewpoint changes, leveraging the fact that most content remains static across frames. We also improve upon the real-time quad generation algorithm of prior work, achieving more compact data sizes and faster performance. Instead of merging multiple camera viewpoints, which requires significant quad overlap, we handle potentially disoccluded regions by uncovering hidden fragments using a depth peeling method from [Kim and Lee 2023]. We show that our representation is more amenable to network rate adaptation and QoE optimization, which are essential in any streaming system. Specifically, we describe a quad packing and texture atlas creation scheme that makes it easy to downsample hidden layers and reduce data rate to adapt to bandwidth availability. Lastly, our technique delivers a higher quality than several baseline methods and achieves a quality comparable to [Hladky et al. 2022], while requiring less memory and operating at a lower data rate.

In summary, we make the following contributions:

- An end-to-end quad-based geometry approximation and streaming system for remote rendering under typical network bandwidths. We improve state-of-the-art geometry streaming techniques with a depth peeling technique which leads to a more compact data structure and introduce a lightweight temporal compression scheme.
- A discussion of how our technique can be network rate-adaptive and better for QoE optimization.
- A characterization and evaluation of our system under varying network latencies and bandwidths.
- An open-source implementation of our approach and all of our baseline approaches (many of which do not have open implementations) to support the research community. Our implementations can operate in real time over an actual network. The code for this paper can be accessed at: https://github.com/EdwardLu2018/QUASAR.

2 RELATED WORK

In this section, we discuss several remote rendering approaches, emphasizing systems that aim to minimize perceived latency and bandwidth usage. We describe image-based rendering (IBR), scene approximation, and other split rendering techniques and how they relate to our work.

2.0.1 IBR and reprojection. IBR techniques such as ATW [van Waveren 2016] and ASW [Oculus 2016] warp received image frames to novel viewpoints. However, because of the absence of additional geometric information (e.g., depth), ATW fails to capture motion parallax under high network latencies. Although ASW accounts for motion parallax, it does not handle disocclusion effects caused by motion, often resulting in visible artifacts, particularly for objects near the camera. Outatime [Lee et al. 2015] incorporates the prediction of client poses along with a coarse depth map to improve reprojection. Similar methods [Didyk et al. 2010a,b; Mark et al. 1997] stream depth maps to generate meshes or splats to approximate the geometry of the scene. Nevertheless, a single depth map is insufficient to fully capture disocclusions and out-of-frame content, as new geometry may be revealed under viewpoint shifts.

2.0.2 Multi-view warping. ProxyIBR [Reinert et al. 2016] transmits multiple rendered views with varying fields of view to address disocclusions and out-of-view regions. Similarly to our approach, it employs depth peeling to transmit hidden portions of the scene to the client. In addition, ProxyIBR sends decimated scene geometry as a proxy to improve the accuracy of reprojection. Similarly, MPEG's immersive video standard (MIV) [MPEG 2023] adopts a multiview encoding strategy, packing color and depth images into a single HEVC video frame. The client reconstructs these frames and renders the scene using multi-layered spheres.

2.0.3 Depth streaming. To support these techniques, an efficient mechanism is required to stream depth images. High-accuracy scene reconstruction typically demands high bit-width depth representations (commonly 16 bits or more), which are challenging to compress. Previous work has explored the compression of depth into standard video codecs [Pece et al. 2011], although the resulting reconstruction may be inaccurate or unsuitable for some applications. Other approaches leverage GPU-accelerated decoding to transmit multiple depth views for light field streaming [Koniaris et al. 2018], or use masks and bounding boxes to temporally compress depth updates [Koniaris et al. 2017]. Inspired by these methods, our method adopts a similar strategy, emphasizing the transmission of only the updated scene information to the client.

2.0.4 Streaming lighting. Lighting calculations are often a significant component of the rendering pipeline. Previous work [Crassin et al. 2015; Majercik et al. 2019] has shown that global illumination (GI) can be precomputed and streamed to a client to enable advanced shading effects. Light probes [Stengel et al. 2021] can be packed into a video stream and used to apply precomputed irradiance. On the client, a full rendering pass with direct lighting is performed, with the streamed irradiance applied to approximate global illumination (GI). Although this approach offers high visual quality with minimal reprojection artifacts, it requires substantial client-side computation, as the client must render entire portions of the scene. As scenes grow increasingly complex [Karis et al. 2021], client memory remains limited, making it unlikely that full 3D scenes will fit on portable devices with constrained resources.

2.0.5 Collaborative rendering. Rather than requiring the client to perform a complete render pass of the entire scene, some techniques partition the scene into two sets of objects: those rendered remotely for high quality and those rendered locally for low latency [Lai et al. 2017; Lu et al. 2023]. However, this approach demands careful matching between locally and remotely rendered content, as local objects should be rendered with full lighting effects. Taking this a step further, the server can render and stream *residual images* [Meng et al. 2020], representing the difference between low- and high-quality render passes from a given viewpoint. The client performs a low-resolution pass and applies the residual for refinement. Furthermore, high-quality received images can be cached and reused for reprojection [Boos et al. 2016; Cuervo et al. 2015]. View-dependent texture maps [Cohen-Or et al. 1999] can also be transmitted, allowing the client to warp and adjust rendered images during viewpoint changes to further optimize performance and quality. However, these approaches assume powerful clients with substantial rendering capabilities. In contrast, our work targets *thin clients* with limited memory and thermal capacity, which are unlikely to support full-scene rendering.

2.0.6 Geometry streaming. Instead of sending depth maps that can limit the resolution of the reconstructed geometry, triangle primitives can be transmitted to capture the full geometry of a view. This is similar to out-of-core rendering solutions [Karis et al. 2021; Mlakar et al. 2024], which attempt to compress and simplify complex scene geometry stored on disk (in the remote rendering case, stored on a resource-plentiful server) into smaller payloads to stream to the GPU. The technique described in [Debevec et al. 1998] can be adapted to stream *visible* triangles and fill holes by averaging the colors of close-by polygons. Extending this idea, [Mueller et al. 2018] captures both *visible* and *potentially visible* triangles and packs shading into a texture atlas sent as a video stream. [Hladky et al. 2021, 2019] improves upon the sampling and temporal stability of these atlases. Similarly to our technique, [Voglreiter et al. 2023] uses geometric erosion and depth peeling to uncover potentially visible triangles. Although these methods can fully capture disocclusions under high network latencies, they often rely on complex texture-packing algorithms, with temporal stability across consecutive atlases largely managed manually. Additionally, the triangles themselves can be difficult to compress while maintaining high quality. Although geometry simplification methods exist [Garland and Heckbert 2023; Sander et al. 2001], they are not suitable for real-time rendering.

2.0.7 Geometry proxy streaming. Instead of transmitting full mesh information for each view, geometry can be approximated using techniques such as billboarding [Andujar et al. 2004; Décoret et al. 2003] or surfel-based representations [Pfister et al. 2000; Shade et al. 1998; Zwicker et al. 2001]. Quadtree-based approaches [Didyk et al. 2010c] fit and stream hierarchical quads to a scene viewpoint for warping. Seurat [Lall et al. 2018] employs an offline process to generate fitted quads that approximate up to 64 potential viewpoints within a viewing volume. QuadStream [Hladky et al. 2022] extends this by enabling real-time quad construction, generating quads from a G-Buffer and packing their color into a texture atlas, while incorporating multiple viewpoints to handle disocclusions. Although QuadStream achieves high reconstruction quality, it incurs substantial data rates, typically around 1–5 Gbps based on our experiments.

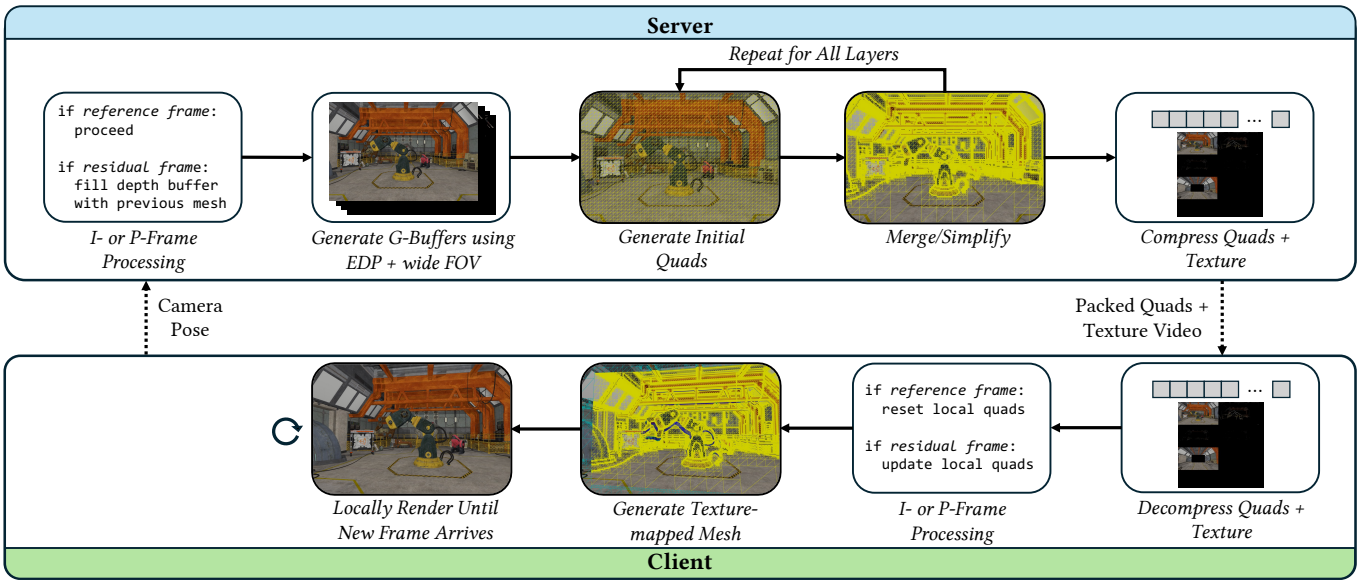


Fig. 2. High-level system pipeline. Using the predicted camera pose, we render a set of G-Buffers using Effective Depth Peeling, which lets only potentially visible fragments pass. For each layer, we generate a corresponding set of quads, which are then merged and simplified to be packed and compressed. We additionally perform temporal compression, sending updates of non-static geometry. A texture atlas is created by appending the color of each layer into a video frame. Upon receiving the compressed quads and video stream, the client constructs a layered textured mesh to render novel views.

Our method improves upon QuadStream by applying temporal compression and quad caching to significantly reduce data transmission.

2.0.8 Layered representations. Layered representations such as multiplane images (MPIs) [Mildenhall et al. 2019; Penner and Zhang 2017; Zhou et al. 2018], layered meshes [Jeschke and Wimmer 2002], and layered imposters [Gernot 1998] approximate scene geometry using stacks of alpha-blended surfaces to capture disocclusions and view-dependent effects. Multi-sphere images (MSIs) [Attal et al. 2020] extend this concept to spherical panoramas for immersive VR. However, depth discretization in these representations makes it challenging to accurately model slanted or oblique surfaces without introducing a large number of layers, which increases storage and computational costs. Additionally, generating these representations typically requires many input images and significant processing time. [Broxton et al. 2020] improves upon MSIs by generating a layered mesh structure to reduce the bandwidth for 3D video streaming. We adopt a similar layered mesh representation, which we find critical for real-time rendering on thin clients and robustly handling disocclusions, while also enabling a more compact and accurate approximation of scene geometry.

2.0.9 Learned representations. Neural and inverse rendering methods can generate scene representations with superior visual quality compared to traditional geometry-based approaches. These techniques directly optimize for photometric accuracy and view-dependent appearance, capturing complex lighting effects and fine scene details. Learning-based methods such as Neural Radiance Fields (NeRFs) [Mildenhall et al. 2021] use *implicit* representations to reconstruct continuous volumetric radiance fields, while Gaussian Splats [Kerbl et al. 2023] generate *explicit* 3D Gaussian primitives for novel view synthesis. Thin-client rendering for these approaches has advanced significantly: NeRFs have achieved real-time performance on mobile hardware through hash grid encoding [Müller et al. 2022] and mesh-based texture baking [Chen et al. 2023], while Gaussian Splats can be sparsified to run efficiently on resource-constrained devices [Mallick et al. 2024]. Streaming techniques are also improving; NeRFPlayer [Song et al. 2023] exploits the temporal coherence of 4D NeRF features for dynamic scene streaming, and real-time Gaussian splat video playback has been demonstrated [Wu et al. 2024], with additional work leveraging hardware-accelerated image and video codecs to further reduce bandwidth [Morgenstern et al. 2025; Wang et al. 2024]. Although both methods can produce near-photorealistic results, they are often constrained by slow training times, typically requiring minutes for moderate quality and several hours for high fidelity, as well as a need for large collections of input images. While ongoing research is accelerating training efficiency [Wang et al. 2023], achieving real-time training from sparse inputs remains an open challenge.

3 SYSTEM OVERVIEW

We make several assumptions common to remote rendering systems. (1) Higher latency and jitter increase pose errors, causing visual artifacts such as misalignment, blurring, and/or warping. (2) Pose prediction can be unreliable under poor network conditions. The best we can do to compensate is define a client-side "viewcell" (e.g., a cube or sphere), sized by the worst-case prediction error, and transmit enough data to cover all views within it. (3) Client motion between server frames is relatively small (i.e., latency and/or client speed is not so high that the client's displacement is significantly large), ensuring some overlap in geometry across frames. In our

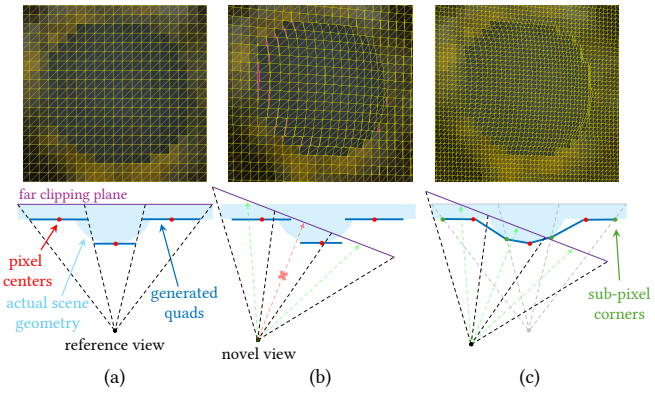


Fig. 3. **(a,b)**: Using one quad per pixel can cause small gaps (in **magenta**) under view offsets. **(c)**: Splitting into four sub-quads and adjusting corner positions with depth offsets can help cover these gaps.

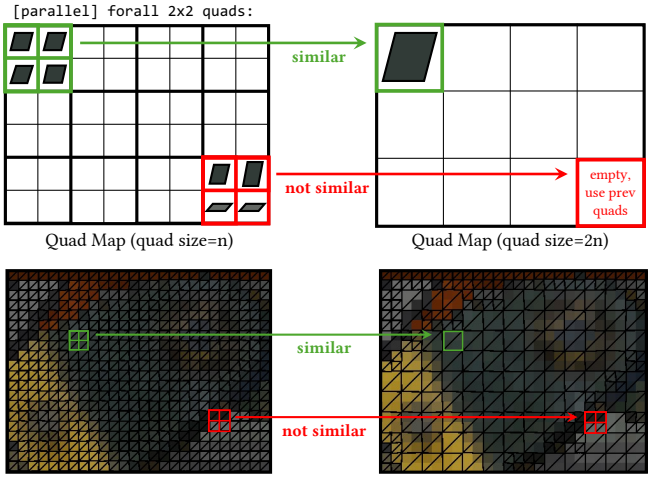


Fig. 4. **Top:** Quad merging and storage with consecutive Quad Maps. Quads are merged based on plane similarity. (quad `size=n`) means the quad stored in the associated Quad Map spans n×n pixels in image space. Once the quads are merged, their positions in the previous Quad Map are cleared. **Bottom:** Visualization of merged quads during reconstruction.

experiments, we target typical human moving speeds of around 1–4 m/s.

3.1 High-level System Pipeline

Our system consists of a server and a client, where the server has significantly more resources—such as greater CPU/GPU performance, memory, and storage—while the client has modest but sufficient computing capabilities. Periodically, the client sends its current pose and camera parameters to the server, the rate of which determines the server's framerate. In our experiments, we set this rate to be 30 FPS, while the client runs at the framerate of its display (e.g., 60 or 72 Hz). The server receives the client pose at a delay depending on network latency and jitter.

Using this pose (along with previous poses), the server predicts where the client will be n ms into the future, where n is determined by a rough estimate of the current network latency (this can be done by pinging the client and measuring the time difference). Using the predicted client pose, the server renders k layers using depth peeling (see Sec. 3.3), which reveals *visible* and *potentially visible* fragments. This generates k G-Buffers, with $k-1$ hidden layers. For each layer, the server generates *quad surfels* using a technique similar to QuadStream [Hladky et al. 2022], with modifications to support further compression (see Sec. 3.2). Lastly, we generate a final set of quads at a wider field of view to capture out-of-frame regions.

For color, we create a texture atlas by stitching the final color of all layers into a single H.264 video. In Sec. 3.4.1, we show that straightforward sequential stitching makes better use of video encoding hardware compared to a random arrangement of image patches.

Quads are compressed using *zstd* [Meta Platforms 2016] compression, which we chose for its balance of compression quality and decompression speed. The texture atlas is compressed using the *nvenc* encoder and sent over UDP, while quad metadata is transmitted via TCP. Although we assume a lossless network, real-world deployment would benefit from connectionless protocols (like UDP) with typical multimedia error correction techniques. Dropped packets containing quads could be recovered using neighboring data, but we leave loss handling for future work.

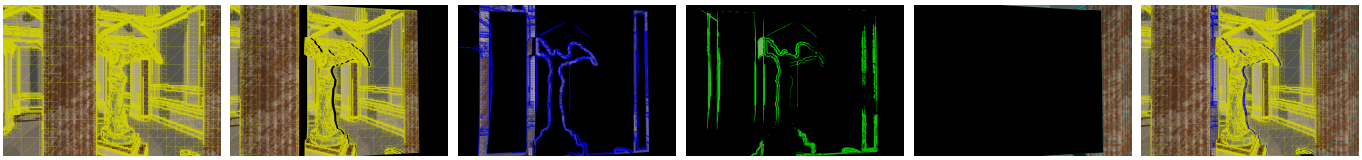
Once received, payloads are decompressed by the client, which runs a series of compute shaders to generate a layered mesh, texture-mapped with a video frame. Using a simple forward rendering pass, the client renders the mesh until a new frame from the server arrives. See Fig. 2 for a high-level visualization.

3.2 Quad Generation

In this section, we present our quad construction algorithm, which is based on the method introduced in QuadStream. Given a G-Buffer, our algorithm fits a set of *quad proxies*—3D primitives created by intersecting planes with *quad frustums*, which represent the local view frustums of individual pixels. Each quad proxy is aligned with a pixel's surface normal and positioned at the corresponding depth. Optionally, each proxy may incorporate a set of *depth offsets* that locally adjust its geometry to better align with the underlying scene geometry (see Fig. 3). By incorporating these offsets, quad proxies can accurately approximate complex surfaces.

Given a predicted pose, the server renders k G-Buffers of size (width, height). To support quad creation, we pre-allocate $m = \lfloor \log_2(\min(\text{width}, \text{height})) \rfloor + 1$ GPU-resident *Quad Maps*—a hierarchy of 2D buffers at progressively halved resolutions, from full size down to $(\text{width}, \text{height})/2^m$. These act as lookup tables for retrieving quad proxies of various sizes (e.g., an 8×8 quad maps to (width, height)/8). We also allocate a 16-bit RGBA *Depth Offset Texture* at twice the G-Buffer resolution. Quad generation and merging are run entirely on the server due to the high memory requirements.

3.2.1 Creating Initial Quad Proxies. For each G-Buffer, a compute shader is dispatched to generate an initial dense quad set. The objective is to fit a quad to each pixel, and subdivide each quad into four sub-quads to create a set of depth offsets. This is achieved by sampling the depth and normals of a target pixel and its 8 neighbors to construct 9 planes. The depth at each sub-quad corner is determined



(a) 1st set of quads from *visible layer 0.* (b) Novel view from inside viewing sphere. (c) 2nd set of quads from *hidden layer 1.* (d) 3rd set of quads from *hidden layer 2.* (e) Final set of quads from *wide FOV.* (f) Mesh is reconstructed on a client from quads.

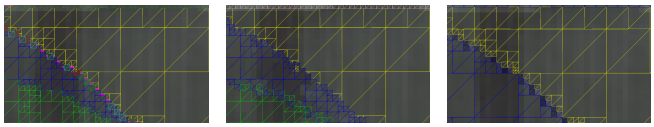
Fig. 5. **5a**: A series of quads are fitted to a scene on the server using the received camera view. The quads are merged to simplify the data size and geometry. **5b**: When the client receives the data payload containing the quad data structures, it is at a different view, revealing holes (black regions). **5c,5d**: To account for disocclusions, we use an Effective Depth Peeling technique to peel away fragments, uncovering potentially visible hidden fragments and create an additional set of quads to cover holes. **5e**: We also render a wide FOV, using the already created quads as a depth mask to create a final set of quads to cover out-of-frame regions. **5f**: Using these quads, the client reconstructs the scene as a multi-layered mesh.

by projecting the corner onto the target pixel's plane and the two or three nearest neighboring planes. If the distance between the target and nearest projections are below a depth threshold δ_{max}, the depth of the corner is the average of the neighboring projections; otherwise, it is the depth of the target pixel projection. This approach captures depth discontinuities while also filling small gaps caused by quad-based geometry approximation. After the corner depths are computed, we find the distance of each corner to the target pixel's plane, creating 16 depth offsets, 4 per sub-quad. Although some of these values may be duplicated, having 16 depth offsets instead of 9 simplifies lookup operations. If all depth offsets are above a quad flattening threshold δ_{flatten}, the values are stored in four RGBA slots in the Depth Offset Texture. The target pixel's plane is stored into the Quad Map of size (width, height). For reference, the code for this algorithm is provided in our open-source release.

3.2.2 Merging Quad Proxies. Using the initial set of quads directly results in complex geometry that is impractical for streaming. To address this, $m-1$ consecutive compute shaders are dispatched to progressively simplify the quads. Each shader operates on consecutive Quad Maps, loading groups of four adjacent proxies from the higher-resolution map. If the differences between their plane equations fall below a plane similarity threshold δ_{sim}, the four planes are averaged, depth offsets are recalculated, and a larger size quad is stored in the lower-resolution map. The corresponding entries in the previous map are then cleared. See Fig. 4 for a visualization.

To reduce the number of quads, we can apply a coarser merging heuristic during the initial passes. Planar similarity constraints can be relaxed and merging is performed if the four quads are similar *or* the quads do not fall within a depth discontinuity (i.e., the depth differences between the corresponding G-Buffer pixels are below δ_{max}). This *coarse merging* strategy essentially performs an edge-aware downsampling of the input G-Buffer which dramatically reduces data size while preserving depth discontinuities. The number of coarse merging passes is a tunable parameter in our system.

3.2.3 Gathering Final Quad Proxies. Finally, a series of compute shaders are executed to scan all m Quad Maps, collecting non-empty entries to generate the final quad proxy set to stream. The data structure used to store these quads is detailed in Sec. 3.4.



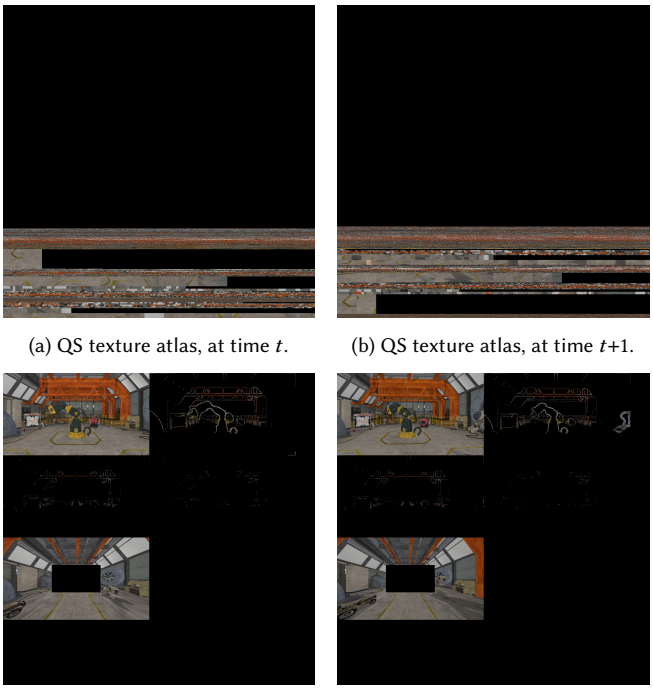
(a) QuadStream w/o boundary expansion. (b) QuadStream w/ boundary expansion. (c) Ours w/o boundary expansion.

Fig. 6. Impact of boundary expansion on reconstruction coverage. We show a zoomed-in portion of a novel view rendering. The center view quads are **yellow**, while additional views/layers are **blue**, **green**, **red**, **cyan**, and **pink**. **6a:** QuadStream can have visible gaps (in **magenta**) due to non-pixel-aligned G-Buffers from view sampling. **6b:** Expanding quad boundaries by one pixel can help mitigate this, at the cost of increasing the number of quads. **6c:** In contrast, our method generates pixel-aligned G-Buffers, avoiding these artifacts entirely and enables more quad merging.

3.3 Adding Potentially Visible Quads

Approximating only a single view with a series of quad proxies leads to disocclusion artifacts, such as visible holes, when rendering novel views from within a viewcell. As the viewpoint shifts, previously occluded regions of the scene become visible and should be captured and transmitted to the client. To address these artifacts, it is necessary to identify *potentially visible* geometry and generate additional quad proxies for coverage. QuadStream [Hladky et al. 2022] adds eight additional quad sets generated with viewpoints at the corners of a box-shaped viewcell, using previously generated quads as masks to reduce redundancy. While this approach offers partial coverage, undersampling may miss geometry that is occluded across all the sampled views, resulting in holes in novel viewpoints. For reference, Seurat [Lall et al. 2018], an offline method, samples 16–64 viewpoints within a view box, which has been shown to provide sufficient coverage.

In contrast, our method captures potentially visible geometry by using a depth peeling technique [Everitt 2001; Mammen 1989], which can capture hidden fragments behind a visible view. However, naively applying depth peeling will introduce significant redundancy, as many fragments remain occluded or are irrelevant in all views within a viewcell. To address this, we adopt the *Effective Depth Peeling (EDP)* method proposed in [Kim and Lee 2023], which introduces the concept of *Potentially Visible Hidden Volumes (PVHVs)* and incorporates an early visibility test to selectively identify and



(a) QS texture atlas, at time t. (b) QS texture atlas, at time t+1.

(c) Our texture atlas, at time t. (d) Our texture atlas, at time t+1.

Fig. 7. Example 4096×4096 texture atlas video frames at two consecutive timestamps for QuadStream and our method. **7a,7b:** QuadStream employs a binning strategy using the generated quad proxies and attempts to reuse patches across frames. **7c,7d:** Our atlas is constructed by appending all the layers generated by EDP. Due to an improved temporal consistency, our method makes better use of video codecs, achieving smaller video sizes.

process only the fragments likely to contribute to a novel view. Using the method described in the paper, we generate $k-1$ additional G-Buffers, each corresponding to a hidden layer (see Fig. 5).

Since EDP supports novel views from within a defined search radius, our technique supports viewing *spheres*, in contrast to cube-shaped viewcells used in QuadStream or Seurat. We define the bounding radius for EDP E as half the diameter of our viewing sphere. We discard any fragments that fall inside a PVHV as well as any non-opaque fragments to handle transparency. In our experiments, we found that the false positive rate (a fragment is captured but will never be visible) is small, and erroneously created quad proxies are not visible to the user. False negative fragments can create holes, but the rate is also small and is not noticeable in practice. For a detailed evaluation, refer to [Kim and Lee 2023].

EDP offers several advantages over prior methods. As shown in Fig. 6, QuadStream's sparse sampling can lead to visible gaps after reconstruction. These gaps are not always fully covered by additional views, so QuadStream introduces one-pixel boundary quads to fill them. Our QuadStream implementation checks each empty pixel during initial quad generation and inserts an extra quad if it neighbors a non-empty pixel, assigning it the neighbor's color. Although these could later be merged, they can also increase the overall size of the data. Using EDP avoids this issue entirely, as all the generated G-Buffer layers are pixel-aligned. Moreover, QuadStream relies on multiple viewpoints to cover occluded regions, leading to overlapping quads that are difficult to merge. In contrast, EDP can capture complete occluded geometry within a single layer, enabling more aggressive quad merging. As shown in Fig. 6, our method covers the same region with just two quad sets compared to four for QuadStream, achieving similar visual quality with fewer quads.

3.4 Storage and Data Payload

In this section, we describe how we construct our data payload, which consists of a video texture and metadata for quad proxy structures.

3.4.1 Texture Atlas. After we create the quad proxies for all layers and wide FOV, we composite the final video frame by appending the color buffers generated from each G-Buffer. Our texture atlas is constructed using a straightforward approach: individual color textures are sequentially appended to form a 4096×4096 video frame, which is then passed to a video encoder. This simple linear packing strategy surprisingly outperforms the bin-packing method used in QuadStream by around 1.5–2× in terms of video data rates in our experiments. This can be attributed to the inherent design of video encoders, which are specifically optimized to take advantage of spatial and temporal locality typically observed in natural video footage. See Fig. 7 for a reference video frame for both methods. Linear packing of full image frames also gives our approach greater flexibility in allocating resolution, enabling reducing resolution for color data derived from hidden layers. Given that texture data are relatively small compared to geometry data, we leave these optimizations for future work.

This approach has two main limitations. First, to fit into a 4096×4096 video frame, we can support only up to three hidden layers along with a wide FOV, which we found to be sufficient in our experiments. Second, blurring can occur at the edges of the image due to the nature of block-based video codecs. As a result, color information from background quads can bleed into neighboring foreground quads at object boundaries, leading to perceptual edge artifacts. Although these artifacts are typically subtle, they can become noticeable in high-contrast regions or under close inspection. Future work could explore a video macroblock-aware quad generation algorithm to more tightly couple our technique with existing video codecs.

3.4.2 Quad Proxy Packing. A quad proxy is defined as a plane in view space, determined by casting rays from the corners of an n×n group of pixels. It is characterized by its size (n), a normal vector, and a reference point on the plane. At full precision, it can be represented using a 32-bit unsigned integer and six 32-bit floating-point numbers, though many of these values can be quantized to reduce storage.

Following the approach in QuadStream, we constrain the quad proxy size to be powers of 2, up to 2048, allowing the size to be encoded in 6 bits. The normal vector always faces the camera, so it can be represented by two spherical coordinates and quantized into two 8-bit values. The plane point is decomposed into x and y image space coordinates along with a depth value. x and y are stored as 12-bit values, supporting resolutions up to 4096×4096, though we practically use 11 bits for full HD 1920×1080. Depth is preserved

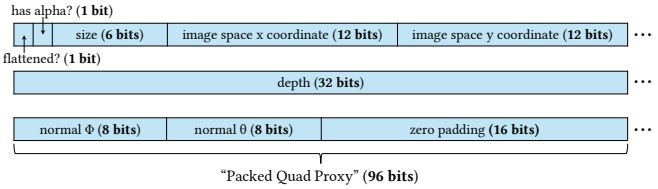


Fig. 8. Data layout of a single packed quad proxy data structure. Each field (e.g., depth, normal, etc.) is grouped across all structs, combined into a single data block, and compressed using *zstd*.

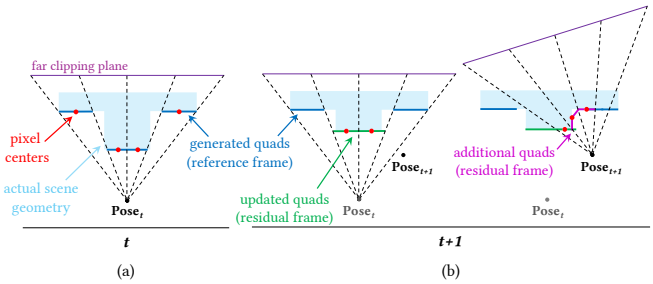


Fig. 9. **(a):** Given a G-Buffer, we generate a set of quads to approximate the scene surfaces and send it to the client to reconstruct. We call this the *reference frame*. **(b):** To account for scene updates, we use the reference quads to fill the depth buffer as a mask and rerender the scene at the reference camera pose, generating new quads where geometry has changed (bottom left). To account for newly revealed geometry, we render the scene at the new camera pose using a depth mask filled with the old mesh updated with changed regions (bottom right). We send the client both the newly generated quads from the reference camera view and the revealed quads from the new camera view. We call this the *residual frame*.

at full 32-bit floating-point precision to ensure high-quality reconstruction. Additionally, a single bit is reserved to indicate whether the quad is flattened (whether or not to apply depth offsets). This flattened bit, along with the size and xy values, are packed into a single 32-bit integer, to ensure memory alignments. We also reserve a bit to determine if the pixel has an alpha value or not.

This encoding enables a quad proxy to be stored using only three 32-bit values, cutting the memory footprint by more than half (see Fig. 8). To ensure 32-bit alignment, 16 bits of zero padding are added, which are effectively removed during final *zstd* compression. Unlike QuadStream, which relies on a 24-bit offset per quad proxy to store texture coordinates, our sequential texture stitching approach allows us to derive the texture coordinates directly from a quad's x and y. We add a fixed header in our data payload that encodes the number of quads per layer, enabling dynamic computation of texture atlas offsets. As a result, our method requires only 80 bits per quad, compared to QuadStream's 103 bits.

We perform this quantization for each quad in every layer and store the resulting structure as an array of values, meaning the normal, depth, and flattened/size/xy fields are not interleaved. We found that this produces higher compression ratios when we run *zstd* and allows for optimized memory access on the client.

3.5 Temporal Compression

Sending a completely new set of quads for each frame is extremely bandwidth-intensive. Instead, we can exploit the fact that much of the scene geometry remains static, even when animations are run or when new geometry is revealed. Mainly, the appearance across frames changes mostly due to lighting and/or viewpoint changes. By caching previously received quad proxies, the client can reuse and render them alongside newly sent ones. Lighting changes can be reflected by updating textures.

Caching many quad proxy sets over time is impractical due to memory constraints. Instead, we cache only a single previous frame, representing one prior view. Storing all layers from that frame can also have a large memory overhead, so we cache only the visible (first) layer, which accounts for approximately 60% of the total data payload. We will call this set of quads a *keyframe*, or *reference frame*. It serves as a complete set of visible quads from a past view and is used as a reference for subsequent *residual frames*, which contain updated and/or revealed geometry. This lets the client reuse cached geometry and reduces the size of future data payloads.

3.5.1 *Static Scenes.*

In static scenes, the server reconstructs the cached reference frame and uses it as a mask to cull regions of the G-Buffer where geometry has already been sent. By rendering the reference frame's mesh and pre-filling a depth mask, the server generates new quads only for regions revealed by disocclusions. Since this residual set of quads is sampled from an arbitrary new viewpoint, its quads are not pixel-aligned with those of the reference; therefore, we apply the boundary expansion method described in Sec. 3.3 to the residual quad set only. Despite the added quads, the residual frame remains significantly smaller than a full reference frame, typically less than half its size. Since the client caches only a single reference frame, significant viewpoint shifts may lead to noticeable deviations. To address this, the server can periodically resend new reference frames (e.g., every 5 frames or based on distance and angle thresholds).

3.5.2 *Dynamic Scenes.*

In dynamic scenes, static assumptions break down because the content of a reference frame can change over time. To address this, the server updates the cached reference frame by sending additional quads that reflect the changes. Specifically, the server creates an updated reference frame from the previous viewpoint and compares it to the original reference frame. Using the depth and stencil buffer, the server identifies regions of the scene where the geometry has changed and generates new quads only for those regions. This ensures that the residual frame size remains small while accounting for dynamic content. See Fig. 1 and Fig. 9 for visualizations.

Thus, a full residual frame comprises three sets of quads: one set corresponding to the updated geometry from the reference frame, another set representing new geometry uncovered due to camera movements, and the final set capturing hidden layers and wide FOV. This allows the client to update its cached reference frame and maintain an accurate representation of the scene within the viewcell without needing the transmit a full frame from the server. In our experiments, we find that the use of residual frames can result in

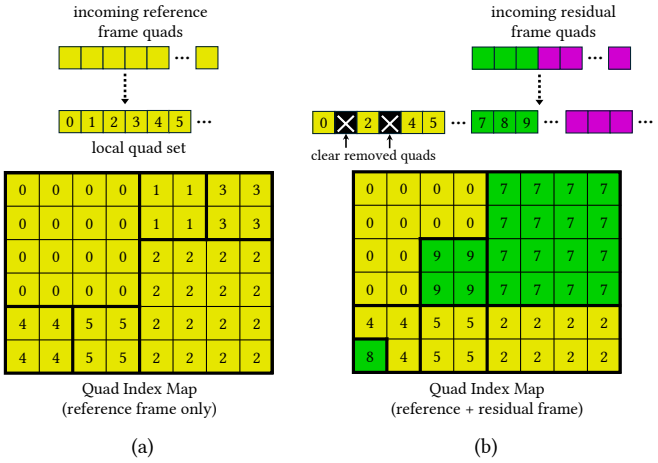


Fig. 10. Client quad management. **(a):** Upon receiving a reference frame (**yellow**), the client stores its quads in a local buffer and rasterizes their indices into a 2D map. **(b):** On receiving a residual frame, new quads are appended, and updated indices (**green**) are rasterized. Fully covered quads are cleared and marked for replacement. Quads revealed from disocclusions (**magenta**) are simply reconstructed and replaced in subsequent frames. The client uses the Quad Index Map to select quads for visible layer reconstruction. Fig. 1 shows an example visualization.

data rates 2–3× smaller than using only reference frames. To update the color of the reference frame, we fit an additional image into our texture atlas that corresponds to shading of revealed geometry.

Since residual frames encode only frame-to-frame differences, our temporal compression technique is inherently agnostic to the type of scene changes, enabling it to accommodate a wide range of geometric updates, including rigged animations and deformable meshes. However, in fully dynamic scenarios—for instance, ocean simulations where a continuously moving water mesh occupies the entire view—this approach offers limited benefit over reference frame-only streaming. Addressing such cases may require a finer-grained residual representation, potentially by computing and quantizing temporal differences in quad data structures rather than transmitting entirely new quads for scene updates. Furthermore, an interesting avenue for future research would involve the use of quad-based motion compensation, similar to those used in 2D video codecs, to enhance inter-frame compression.

3.6 Client Reconstruction

The client pre-allocates a buffer on the GPU that holds the local set of quad proxies, a 16-bit RGBA texture that holds the local depth offsets, and several vertex and index buffers for the layered mesh. It also allocated a 2D buffer of 32-bit integers of size (width, height), called the *Quad Index Map*, which tracks updates to the reference frame by storing the index of the quad covering each pixel.

When a *reference frame* arrives, the client dispatches a compute shader to overwrite the local quad buffer with incoming data. Using the prior camera pose, quad indices are rasterized into the Quad Index Map.

When a *residual frame* is received, its quads are appended to the quad buffer rather than replacing it. It will rasterize the quads corresponding to reference frame changes to update the Quad Index Map, allowing for the integration of residual quads without reprocessing the full reference frame. If any residual quads entirely cover a reference quad, the corresponding entry in the local buffer is marked as empty and will be replaced in a subsequent frame. This enables memory reuse and prevents unbounded growth of the local quad buffer. Additionally, quads corresponding to newly revealed geometry due to viewpoint changes are reconstructed and marked for replacement in subsequent frames. See Fig. 10 for a visualization.

A final compute shader reads the Quad Index Map to retrieve the quads required to reconstruct the multi-layer mesh.

3.7 Transparency

Since depth peeling was originally designed to generate multiple layers for order-independent transparency, incorporating transparency effects into our system is straightforward. An optional 4096×4096 8-bit texture atlas can be introduced to store alpha values, initialized under the assumption that all surfaces are fully opaque. The alpha atlas shares the same layout as the color atlas, ensuring easy-to-compute texture coordinates. During the construction of the G-Buffer layers, whenever a transparent pixel is encountered, its corresponding alpha value is recorded in the alpha atlas, and an alpha flag is set in the associated quad proxy data structure. If a fragment in a previous layer is transparent, we retain the subsequent hidden fragment rather than discarding it, ensuring its inclusion during quad generation.

During the merging phase, if any of the four quads being processed are marked as containing alpha, the resulting merged quad is similarly flagged. During mesh reconstruction, if the client identifies a quad as containing transparency, it retrieves the corresponding alpha values for the quad's covered pixels from the alpha atlas. This mechanism supports quads with spatially varying alpha, enabling mixed transparency within a single quad.

To support this functionality, the alpha atlas must be streamed to the client, which can be accomplished by streaming an additional H.264 video stream (which would only contain a luma channel) or by compression using *zstd* combined with delta encoding, similar to the scheme used by QuadStream. In practice, the additional bandwidth overhead is minimal as most of our tested scenes contain only a small number of transparent objects.

4 EVALUATION

We evaluate our system based on end-to-end performance, visual quality, and data efficiency. Our method surpasses basic Asynchronous Time Warping (ATW) and depth-based MeshWarping (MW) in visual quality while achieving a quality comparable to QuadStream (QS). At the same time, it achieves a significant reduction in data footprint by using EDP and temporal compression.

4.1 Experimental Setup

Our system is written using OpenGL 4.6 and tested on a desktop with an AMD Ryzen 9 7950X 16-Core Processor and an NVIDIA GeForce RTX 4090. We report performance for a server and a simulated client

(a) Robot Lab (RL)

(b) Sun Temple (ST)

(c) Viking Village (VV)

(d) San Miguel (SM)

Fig. 11. Scenes used in evaluations. Rendered with deferred shading.

running on the desktop, as well as for a client on a Meta Quest 3 VR headset. The Quest 3 prototype is implemented using OpenGL ES 3.2 and OpenXR. The headset has a Snapdragon XR2 Gen 2 chipset and renders using the `OVR_multiview` extension at a resolution of 1680×1760 per eye. The desktop client renders full HD resolution (1920×1080) images. Most of the code is shared across platforms.

To evaluate real-world game scenes, we tested on the same environments as QuadStream and added an additional scene (Sun Temple) to show robustness. We tested in Unity's "Robot Lab" [Unity Technologies 2018], UE4's "Sun Temple" [Epic Games 2017], Unity's "Viking Village" [Unity Technologies 2022], and the "San Miguel" atrium [McGuire 2017]. Each scene contains image-based lighting, along with one directional light and four point lights, all of which cast Percentage Closer Filtering (PCF) shadows. We implement frustum culling for performance, but do not perform instancing and all objects are rendered at their highest Level-of-Detail (LOD). To show that we can support dynamic scenes, we added animations to all scenes. The Robot Lab (0.47M triangles, 635 objects, moderate depth complexity) is a medium-sized indoor scene with four dynamic objects, challenging thin structures such as railings, stairs, and perforated walls. The Sun Temple (0.6M triangles, 1057 objects, high depth complexity), another medium-sized scene, is a long narrow hallway with open views, stairs, sharp-cornered columns, and rotating and flying angel statues. The Viking Village (4.3M triangles, 1266 objects, high depth complexity), a large outdoor scene, has animated dragon masts on buildings, rotating skull-adorned sticks, flying swords, and rotating rocks. Its challenges include detailed roofs and overlapping buildings. San Miguel (9.9M triangles, 1600 objects, high depth complexity), a large partly indoor scene, contains very detailed individually modeled plants, dense foliage, and many thin moving chairs. See Fig. 11 for a still frame of each scene.

4.1.1 Baselines. We compare our approach to three previous remote rendering reprojection schemes. (1) ATW [van Waveren 2016] warps a received frame along a plane. It is simple to implement and has the least computational overhead. (2) MeshWarp [Mark et al. 1997] generates a mesh from a G-Buffer. It is also simple to implement and can be network-adaptive by tuning the depth map resolution. (3) QuadStream [Hladky et al. 2022] uses multiple G-Buffers at different viewpoints to generate quads that are quantized and streamed to a client, similar to our approach. Due to the lack of open-source implementations available, we approximate QuadStream by adapting our existing code to generate quads, rather than replicating the original method exactly. Our QuadStream leverages multiple camera views and pre-fills a depth mask using quads generated from other views, following a strategy similar to that described in their paper [Hladky et al. 2022].

Table 1. Average absolute error of client pose prediction under simulated network latencies of 20 ms (with ±10 ms jitter) and 50 ms (with ±20 ms jitter). Errors are reported as **mean** ± std / **max**.

Scene	Errors w/ Pred		Errors w/o Pred	
	Position	Rotation	Position	Rotation
20 ± 10 ms Latency				
RL	9 ± 6 / 46 cm	0.9 ± 1.3 / 14.7°	32 ± 12 / 76 cm	2.3 ± 3.0 / 22.7°
ST	7 ± 4 / 23 cm	0.6 ± 1.3 / 14.4°	25 ± 7 / 50 cm	1.5 ± 2.7 / 21.1°
VV	7 ± 5 / 23 cm	0.4 ± 0.8 / 8.6°	27 ± 9 / 45 cm	1.0 ± 1.6 / 13.3°
SM	2 ± 2 / 14 cm	0.6 ± 0.8 / 8.5°	9 ± 4 / 20 cm	1.8 ± 1.8 / 14.4°
50 ± 20 ms Latency				
RL	53 ± 20 / 122 cm	2.4 ± 3.2 / 26.1°	64 ± 24 / 142 cm	4.7 ± 5.5 / 37.6°
ST	42 ± 12 / 87 cm	1.5 ± 2.7 / 21.2°	51 ± 15 / 96 cm	3.1 ± 5.0 / 29.4°
VV	45 ± 15 / 84 cm	1.2 ± 1.8 / 12.8°	56 ± 18 / 88 cm	2.1 ± 2.9 / 18.0°
SM	15 ± 6 / 39 cm	1.7 ± 1.8 / 14.6°	19 ± 8 / 38 cm	3.6 ± 3.1 / 19.2°

Pure triangle streaming with texture-space shading systems—such as those in [Hladky et al. 2021, 2019; Mueller et al. 2018]—also lack open-source implementations, making direct comparisons difficult. Instead, we provide a conceptual discussion. Triangle streaming transmits raw triangle data to the client, offering high geometric fidelity, latency masking via potentially visible set calculations, and caching for static geometry. However, it scales poorly with scene complexity, leading to higher data rates and reduced client performance. Although decimation can help, real-time mesh simplification remains limited. Shading quality can also suffer due to difficulties in packing numerous colored triangle-shaped patches into a single video frame. As shown in [Hladky et al. 2022], such systems even fail to operate in scenes with high model complexity. In contrast, quad-based methods leverage geometric *proxies*, reducing both data and shading rates while maintaining visual quality. Compared to prior texture-space systems, they better exploit screen-space coherence and achieve smaller payloads through more effective quantization. As scene complexity grows with virtual geometry systems like Nanite [Karis et al. 2021], we believe quad-based representations offer a more scalable solution, although triangle streaming still remains viable for simpler scenes.

4.1.2 Configuration of parameters. For ATW, in out-of-frame regions, we apply a closest-pixel in-fill instead of displaying a black border. This approach slightly enhances visual quality and helps hide sharp edges at frame boundaries, but produces a smearing-like artifact. ATW transmits a full 1920×1080 resolution video stream.

For MeshWarp, we report performance using 32-bit depth maps with vertical FOVs of 60° (matching the client) and 120°. Doubling the FOV better captures out-of-frame regions but lowers central resolution, so we double the resolution to 3840×2160 for 120°. Depth

Table 2. **Server** computation times. Measured on a high-end desktop. For ATW and MW, *Render* is the time for the server to render the scene and store the results for streaming. For QS and Ours, *G-Buffers* is the total amount of time to render G-Buffers for any step of the method. *Quads* is the total amount of time to fit the initial set of dense quads proxies from the G-Buffers and *Merge* is the amount of time it takes to merge and simplify the quads. *Encode* is the total amount of time to perform *zstd* compression. While we report total compression times, the process itself can be pipelined and run in a background thread.

Scene &	**ATW**	**MW (120°)**		**QuadStream**				**Ours (Reference Frames)**				**Ours (Residual Frames)**			
VC Size	Render	Render	Encode	G-Buffers	Quads	Merge	Encode	G-Buffers	Quads	Merge	Encode	G-Buffers	Quads	Merge	Encode
RL (25cm)	3.05 ms	4.32 ms	4.09 ms	23.51 ms	5.78 ms	5.36 ms	80.58 ms	13.24 ms	3.73 ms	1.12 ms	35.65 ms	20.15 ms	5.14 ms	9.38 ms	39.82 ms
RL (50cm)	3.05 ms	4.32 ms	4.09 ms	23.43 ms	6.01 ms	5.62 ms	80.23 ms	12.98 ms	3.76 ms	1.20 ms	35.44 ms	19.73 ms	5.20 ms	9.45 ms	39.39 ms
RL (100cm)	3.07 ms	4.31 ms	4.23 ms	23.87 ms	6.42 ms	6.43 ms	85.69 ms	1.15 ms	3.84 ms	1.33 ms	35.28 ms	20.07 ms	5.34 ms	7.77 ms	39.26 ms
ST (25cm)	6.17 ms	7.43 ms	4.18 ms	40.21 ms	5.69 ms	3.09 ms	71.50 ms	23.97 ms	3.71 ms	1.40 ms	33.70 ms	37.60 ms	5.19 ms	4.20 ms	38.11 ms
ST (50cm)	6.17 ms	7.43 ms	4.18 ms	40.43 ms	5.94 ms	3.25 ms	74.69 ms	23.66 ms	3.83 ms	1.47 ms	35.24 ms	37.25 ms	5.30 ms	4.32 ms	39.88 ms
ST (100cm)	6.19 ms	7.46 ms	4.05 ms	38.95 ms	6.40 ms	3.29 ms	108.12 ms	24.02 ms	4.09 ms	2.30 ms	36.05 ms	37.56 ms	5.50 ms	3.80 ms	40.57 ms
VV (25cm)	2.59 ms	3.98 ms	4.06 ms	19.63 ms	5.48 ms	2.48 ms	125.28 ms	10.57 ms	3.35 ms	2.46 ms	41.30 ms	16.42 ms	4.59 ms	8.76 ms	43.64 ms
VV (50cm)	2.59 ms	3.98 ms	4.06 ms	19.94 ms	5.75 ms	2.67 ms	129.20 ms	10.52 ms	3.53 ms	3.11 ms	42.36 ms	16.43 ms	4.74 ms	9.31 ms	44.78 ms
VV (100cm)	2.55 ms	4.00 ms	4.13 ms	20.41 ms	6.20 ms	2.86 ms	135.51 ms	10.77 ms	3.81 ms	2.49 ms	43.73 ms	16.73 ms	5.01 ms	10.39 ms	45.94 ms
SM (25cm)	9.73 ms	10.06 ms	6.74 ms	67.13 ms	6.74 ms	2.35 ms	110.27 ms	38.59 ms	4.28 ms	1.11 ms	50.38 ms	59.24 ms	5.70 ms	18.87 ms	53.07 ms
SM (50cm)	9.73 ms	10.05 ms	6.74 ms	67.57 ms	7.14 ms	2.57 ms	115.71 ms	38.55 ms	4.43 ms	1.73 ms	49.21 ms	59.40 ms	5.86 ms	19.71 ms	51.69 ms
SM (100cm)	9.84 ms	10.06 ms	6.95 ms	69.12 ms	7.74 ms	2.92 ms	122.78 ms	38.65 ms	4.61 ms	2.20 ms	49.33 ms	59.26 ms	6.00 ms	22.26 ms	51.69 ms

maps are compressed with a BC4-like 8×8 block scheme [Koniaris et al. 2018], reducing the size by 8×, followed by *zstd* compression.

For QuadStream, we use the same parameter values as in the original paper, except for the proxy flattening threshold (δ_{flatten}), which we set to 0.05 to reduce the amount of depth offsets. Specifically, we used a depth threshold (δ_{max}) of 10^{-4} and a plane similarity threshold (δ_{sim}) of 0.1. Around 90–95% of the depth offsets are empty. We follow the original viewcell sampling order—center view, then front and back corners clockwise—and generate a wide FOV proxy set at 120°, rendered at 1280×720. We also apply our "coarse merging" heuristic in the first merge pass, which is conceptually equivalent to the "quad splitting" strategy described in the original paper. Binary quad metadata is compressed using *zstd*.

For our method, to balance the data rate and visual quality, we adopt more liberal parameter settings. We set δ_{flatten} to 0.2 and δ_{sim} to 0.5 to further reduce the number of depth offsets. We apply coarse quad merging during the first 3 merging passes, essentially downsampling the G-Buffer by 8× in non-edge regions. We also include a 120° wide FOV proxy at half the visible view resolution (960×520). For EDP, we use a connectivity depth threshold of 0.001. For each EDP hidden layer and for the wide FOV, we relax δ_{flatten} and δ_{sim} by 10× and apply course merging 4 times, as these views occupy a smaller portion of the client's view and can tolerate more aggressive simplification. For temporal compression, we insert a reference frame every 5 frames; otherwise, we send residual frames.

4.1.3 Experiments. For each scene, we evaluate the performance along a camera trace consisting of 1500 poses. The server operates at 30 FPS, and we simulate two network conditions: 20 ms ("local") and 50 ms ("cloud") one-way delays. For 20 ms, we add ± 10 ms of uniformly random network jitter, while for 50 ms we add ± 20 ms. These delays represent one-way latency (half the round-trip time), meaning the server receives client poses, and the client receives server frames with corresponding delays. The desktop client runs at 60 FPS and the headset client at 72 FPS; the client frame rate is decoupled from the server's. Each method is evaluated by running all four traces at the tested latency, collecting 1500 frames per trace, which are then compared against the corresponding 1500 ground truth frames using perceptual similarity metrics. To maintain synchronization, we ensure that the server and client rates are matched so that the animations remain aligned. If the server cannot maintain 30 FPS, we virtually slow the time proportionally across all reprojection methods to preserve frame correspondence. However, all run-time performance results are reported in actual (wall-clock) time. For each network delay setting, we test with viewcell sizes of 25, 50, and 100 cm.

4.1.4 Pose Prediction. We perform kinematic pose prediction on the server using the client's translational and angular velocity and acceleration. While more advanced prediction methods exist, we opt for a simpler approach to reduce system complexity. Tab. 1 reports the average absolute pose errors from our prediction. The table shows that at lower latency and jitter, basic kinematic pose prediction performs well. However, these results fall short of the practical limits—approximately 5 cm and 5°—reported in prior work [Shotton et al. 2013], demonstrating the need for reprojection to maintain visual consistency. Under higher latency and jitter, prediction accuracy degrades significantly, introducing instability. To mitigate this, we apply pose smoothing using a Savitzky-Golay filter for 50±20 ms latency only, which reduces error. The errors for 50±20 ms reported in the table are *after* filtering is applied.

These increases in pose error due to latency and jitter will directly affect visual quality, leading to method-specific artifacts. For ATW, high pose error increases translational mismatches and out-of-frame artifacts. In MW, higher pose error causes more disocclusion events, amplifying "rubber sheet" artifacts. For QS and our method, increased pose error raises the risk of the client moving outside the viewcell or undergoing extreme rotations, leading to visible reconstruction gaps and reduced resolution in out-of-frame regions. Larger viewcells or higher resolution wide FOV G-Buffers can help mitigate these artifacts at the cost of increasing data rates.

4.2 End-to-End Performance

The server performance results are shown in Tab. 2, and the client performance is shown in Tab. 3. The timing measurements reflect the wall-clock execution time of each major processing step for all methods. Server results were obtained on a high-end PC, while client performance was estimated by replaying saved camera frames

Table 3. **Client** computation times. Several frames were saved from our traces and loaded on a Quest 3 VR headset. ***Native*** shows the performance of the scene rendering locally on the headset, without remote rendering. *Render* is the amount of time it takes the client to render a frame. *Decode* is the amount of time it takes *zstd* to decompress the data payload received. *Mesh* is the amount of time required to fill the vertex and index buffers using the received information. *Quads* is the amount of time it takes to update local quad buffers (either filling buffers for reference frames or updating/appending quads for residual frames).

Scene &	Native	ATW	MW (120°)			QuadStream			Ours (Reference Frames)				Ours (Residual Frames)			
VC Size	Render	Render	Decode	Mesh	Render	Decode	Mesh	Render	Decode	Quads	Mesh	Render	Decode	Quads	Mesh	Render
RL (25cm)	123.8 ms	0.01 ms	5.03 ms	0.09 ms	95.30 ms	13.45 ms	0.29 ms	18.94 ms	11.73 ms	0.40 ms	0.16 ms	12.68 ms	10.04 ms	0.45 ms	0.20 ms	15.10 ms
RL (50cm)	122.9 ms	0.01 ms	5.37 ms	0.07 ms	99.42 ms	13.78 ms	0.29 ms	18.95 ms	11.56 ms	0.40 ms	0.16 ms	13.60 ms	10.91 ms	0.45 ms	0.20 ms	15.23 ms
RL (100cm)	123.4 ms	0.01 ms	5.50 ms	0.09 ms	97.20 ms	13.89 ms	0.31 ms	21.41 ms	12.39 ms	0.40 ms	0.16 ms	14.06 ms	10.75 ms	0.45 ms	0.21 ms	15.78 ms
ST (25cm)	216.7 ms	0.01 ms	5.21 ms	0.08 ms	97.34 ms	12.14 ms	0.28 ms	21.89 ms	10.98 ms	0.47 ms	0.15 ms	13.01 ms	10.44 ms	0.49 ms	0.20 ms	16.01 ms
ST (50cm)	214.2 ms	0.01 ms	5.18 ms	0.08 ms	96.24 ms	12.81 ms	0.28 ms	23.72 ms	10.05 ms	0.47 ms	0.16 ms	15.21 ms	10.44 ms	0.49 ms	0.18 ms	16.17 ms
ST (100cm)	220.8 ms	0.01 ms	5.61 ms	0.10 ms	99.10 ms	12.95 ms	0.28 ms	26.84 ms	10.59 ms	0.47 ms	0.16 ms	15.98 ms	10.40 ms	0.49 ms	0.20 ms	16.66 ms
VV (25cm)	127.8 ms	0.01 ms	5.93 ms	0.09 ms	95.82 ms	15.78 ms	0.29 ms	21.07 ms	13.44 ms	0.40 ms	0.18 ms	13.12 ms	10.07 ms	0.45 ms	0.20 ms	15.77 ms
VV (50cm)	125.7 ms	0.01 ms	5.99 ms	0.10 ms	96.52 ms	15.81 ms	0.30 ms	22.78 ms	13.44 ms	0.40 ms	0.19 ms	14.54 ms	10.40 ms	0.45 ms	0.19 ms	15.94 ms
VV (100cm)	128.6 ms	0.01 ms	5.91 ms	0.13 ms	97.98 ms	15.86 ms	0.32 ms	24.08 ms	13.35 ms	0.40 ms	0.20 ms	14.90 ms	10.14 ms	0.45 ms	0.20 ms	16.42 ms
SM (25cm)	382.0 ms	0.01 ms	5.89 ms	0.08 ms	94.39 ms	14.67 ms	0.38 ms	27.44 ms	12.34 ms	0.45 ms	0.19 ms	18.95 ms	10.02 ms	0.47 ms	0.19 ms	18.53 ms
SM (50cm)	387.8 ms	0.01 ms	6.05 ms	0.08 ms	92.90 ms	14.85 ms	0.38 ms	28.18 ms	12.23 ms	0.45 ms	0.19 ms	19.88 ms	10.09 ms	0.47 ms	0.19 ms	18.69 ms
SM (100cm)	383.3 ms	0.01 ms	5.92 ms	0.09 ms	92.64 ms	14.90 ms	0.39 ms	28.51 ms	12.19 ms	0.45 ms	0.19 ms	19.13 ms	10.19 ms	0.47 ms	0.19 ms	18.16 ms

on a Quest 3 VR headset. Note that CPU-bound *zstd* compression and decompression times are reported, but these operations can be pipelined and offloaded to a background thread. We report the timing results for MeshWarp at 120° FOV only, as it consistently produced higher visual quality compared to 60° FOV in our evaluations.

4.2.1 Server performance. As shown in Table 2, ATW achieves the shortest runtime, mainly constrained by rendering time. MeshWarp follows, incurring additional overhead from rendering a higher-resolution image and exporting the depth map for streaming.

QuadStream has the highest overall runtime among the methods. Our implementation achieves performance comparable to that of the original paper, with G-Buffer creation accounting for the majority of processing time. Since QuadStream requires rendering 10 views (center, eight corner views, and a wide-angle view), it incurs significant overhead. The center view is the most expensive, contributing approximately 30% of the G-Buffer creation time due to a high number of non-empty pixels. The wide FOV view follows at around 20%, while each corner view takes progressively less time as most fragments are early-discarded by depth testing.

We achieve significantly faster G-Buffer creation and quad generation times compared to QuadStream, enabled by two key optimizations. First, we perform depth peeling within a single render pass, avoiding the overhead of multiple passes per layer. Second, by capturing entire hidden layers and employing more aggressive quad merging, we reduce the overall number of quads that need to be processed, thereby accelerating quad generation. As shown in Fig. 2, we report compute times for both reference and residual frames. Residual frames incur higher G-Buffer creation times due to an additional render pass used to re-render the previous reference frame and capture scene updates. Furthermore, residual frames require the creation of two additional sets of quads, which increases the time required for quad generation and merging. However, the resulting reduction in data size leads to balanced overall performance.

4.2.2 Client performance. Although our traces are emulated on a high-end server, we report client performance metrics collected directly from a Meta Quest 3 VR headset to better reflect real-world execution times. That said, all methods achieve over 1000 FPS on the simulated client when run on the server.

Tab. 3 reports measured client-side performance. As expected, rendering scenes directly on the client yields the lowest framerate, highlighting the benefits of offloading rendering to the server.

Among the tested methods, ATW achieves the best performance due to its very minimal computational overhead, requiring only a single post-processing pass. In contrast, MeshWarp performs suboptimally due to the high-resolution mesh generated from a 4K depth map, which is costly to rasterize on the client. While not included in the reported results, we found that downsampling the mesh to 960×540 enables the client to reach 72 FPS, but with a significant reduction in visual quality. Similar to compression, *zstd* decompression can be offloaded to a background thread on the CPU.

QuadStream's client-side performance is efficient across both mesh generation and rendering. Mesh creation consistently takes approximately 0.3–0.4 ms across all tested scenes, while rendering the simplified quad geometry completes in 19–29 ms, corresponding to 34–53 FPS. This efficiency stems from the reduced geometric complexity achieved through quad simplification, making it significantly lighter than the high-resolution meshes used in MeshWarp. However, during evaluation, we observed instances of local reprojection applied by the headset, particularly in the San Miguel scene to ensure a stable framerate.

Our technique improves client-side performance by promoting increased quad merging, which reduces the number of quads rendered per frame and, in turn, lowers decompression and rendering overhead. However, we incur some minor overhead updating local data structures to handle residual frames. Our frame times range from roughly 13 to 20 ms, corresponding to 50–77 FPS. For residual frames, additional reconstruction passes are required to process both modified reference quads and newly revealed quads, leading to slightly increased mesh generation times and additional overhead in updating local data structures compared to reference frames.

4.3 Visual Quality vs. Data Rate

We evaluated image quality using PSNR, SSIM, and FLIPmetrics [Andersson et al. 2020], with quantitative results summarized in Tab. 4.

Table 4. Image quality comparisons of our system compared to baseline methods at simulated network latencies of 20 ms (with 10 ms jitter) and 50 ms (with 20 ms jitter) for 1920×1080 resolution client renderings. Results are reported as the average quality of all the images rendered using each method compared to a ground truth image (with "0 ms latency") for a 1500 image trace. The results for QuadStream and our technique are shown for three different viewcell sizes to highlight the effect of viewcell size on visual quality. This figure should be cross-referenced with Fig. 12 to show the trade-off between quality and data rate.

Method & VC Size	Robot Lab			Sun Temple			Viking Village			San Miguel		
	PSNR↑	SSIM↑	FLIP↓	PSNR↑	SSIM↑	FLIP↓	PSNR↑	SSIM↑	FLIP↓	PSNR↑	SSIM↑	FLIP↓
20 ± 10 ms Latency												
ATW	24.11	0.555	0.140	24.94	0.648	0.135	25.42	0.698	0.116	22.89	0.563	0.144
MW (60°)	28.09	0.863	0.046	31.89	0.943	0.039	29.20	0.925	0.041	26.30	0.874	0.063
MW (120°)	30.06	0.915	0.044	30.85	0.911	0.040	28.44	0.890	0.047	25.51	0.812	0.068
QS (25cm)	32.66	0.926	0.034	34.29	0.949	0.031	29.71	0.931	0.039	27.77	0.874	0.053
QS (50cm)	32.67	0.925	0.034	34.27	0.948	0.031	29.57	0.922	0.040	27.57	0.871	0.054
QS (100cm)	31.86	0.919	0.035	34.24	0.946	0.032	29.34	0.916	0.042	27.36	0.868	0.056
Ours (25cm)	32.17	0.920	0.034	34.22	0.947	0.032	30.92	0.932	0.036	28.71	0.882	0.050
Ours (50cm)	32.10	0.919	0.034	34.04	0.947	0.032	30.90	0.931	0.036	28.66	0.882	0.050
Ours (100cm)	32.63	0.923	0.034	33.80	0.946	0.032	30.82	0.930	0.036	28.65	0.882	0.050
50 ± 20 ms Latency												
ATW	18.82	0.380	0.264	18.44	0.476	0.280	20.39	0.513	0.225	18.70	0.353	0.261
MW (60°)	24.98	0.820	0.070	27.82	0.883	0.058	25.74	0.899	0.060	21.96	0.744	0.105
MW (120°)	26.72	0.872	0.062	28.49	0.914	0.056	25.52	0.861	0.063	23.13	0.792	0.100
QS (25cm)	28.69	0.901	0.045	30.22	0.931	0.042	28.23	0.906	0.046	26.15	0.850	0.062
QS (50cm)	30.02	0.906	0.040	32.00	0.937	0.037	28.74	0.908	0.044	25.97	0.846	0.064
QS (100cm)	30.47	0.904	0.040	32.33	0.938	0.036	28.78	0.905	0.045	25.37	0.840	0.067
Ours (25cm)	28.49	0.900	0.046	29.25	0.925	0.047	28.88	0.914	0.044	27.09	0.858	0.059
Ours (50cm)	29.94	0.902	0.042	30.94	0.930	0.041	29.65	0.917	0.041	27.38	0.860	0.057
Ours (100cm)	31.11	0.907	0.039	32.16	0.933	0.037	30.00	0.917	0.040	27.36	0.861	0.057

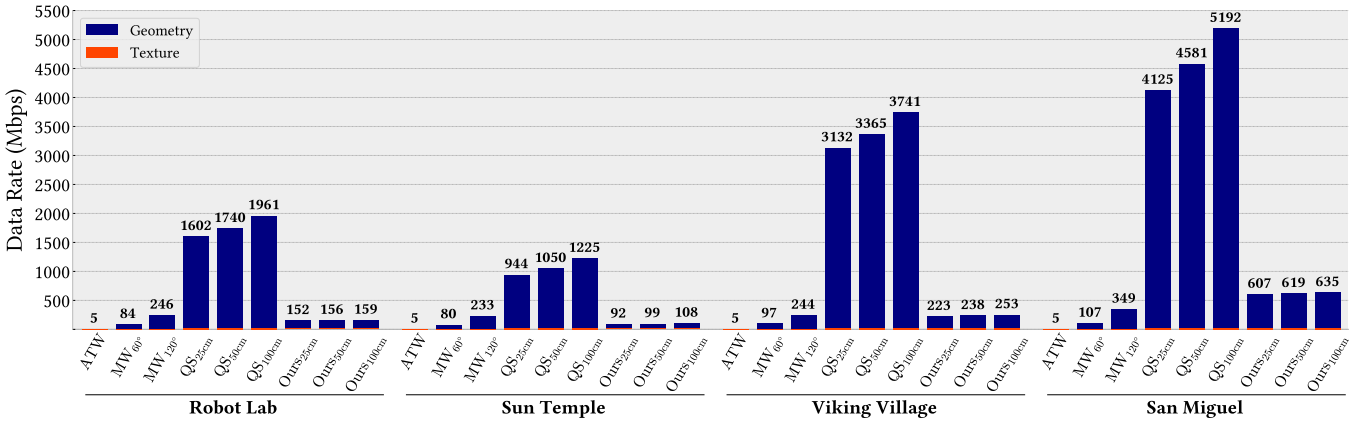


Fig. 12. Average data rates of all methods for each trace in our evaluations. Results for QuadStream and our technique are shown for three different viewcell sizes to highlight the effect of viewcell size on data rate. Reported values are the average of data rates across both tested latencies. This figure should be cross-referenced with Tab. 4.

Furthermore, Fig. 12 presents the data rates for each method across all evaluated traces, and Fig. 13 provides visual comparisons. Our supplemental video includes trace recordings along with error maps to provide a qualitative comparison.

As shown in Tab. 4, ATW has the lowest image quality among the evaluated methods, primarily due to translation mismatches and out-of-frame artifacts. However, it achieves the smallest data rate, as it avoids streaming any geometry or additional views. This efficiency comes at the cost of significantly degraded visual fidelity—ATW performs an order of magnitude worse in FLIP and reports the lowest PSNR and SSIM for both latencies tested.

MeshWarp with a 120° FOV provides good image quality, due to a higher shading rate in peripheral regions compared to the 60° FOV configuration. However, doubling both FOV and resolution substantially increases data rates by around 3–4×. MeshWarp's visual quality is limited by "rubber sheet" artifacts and the absence of

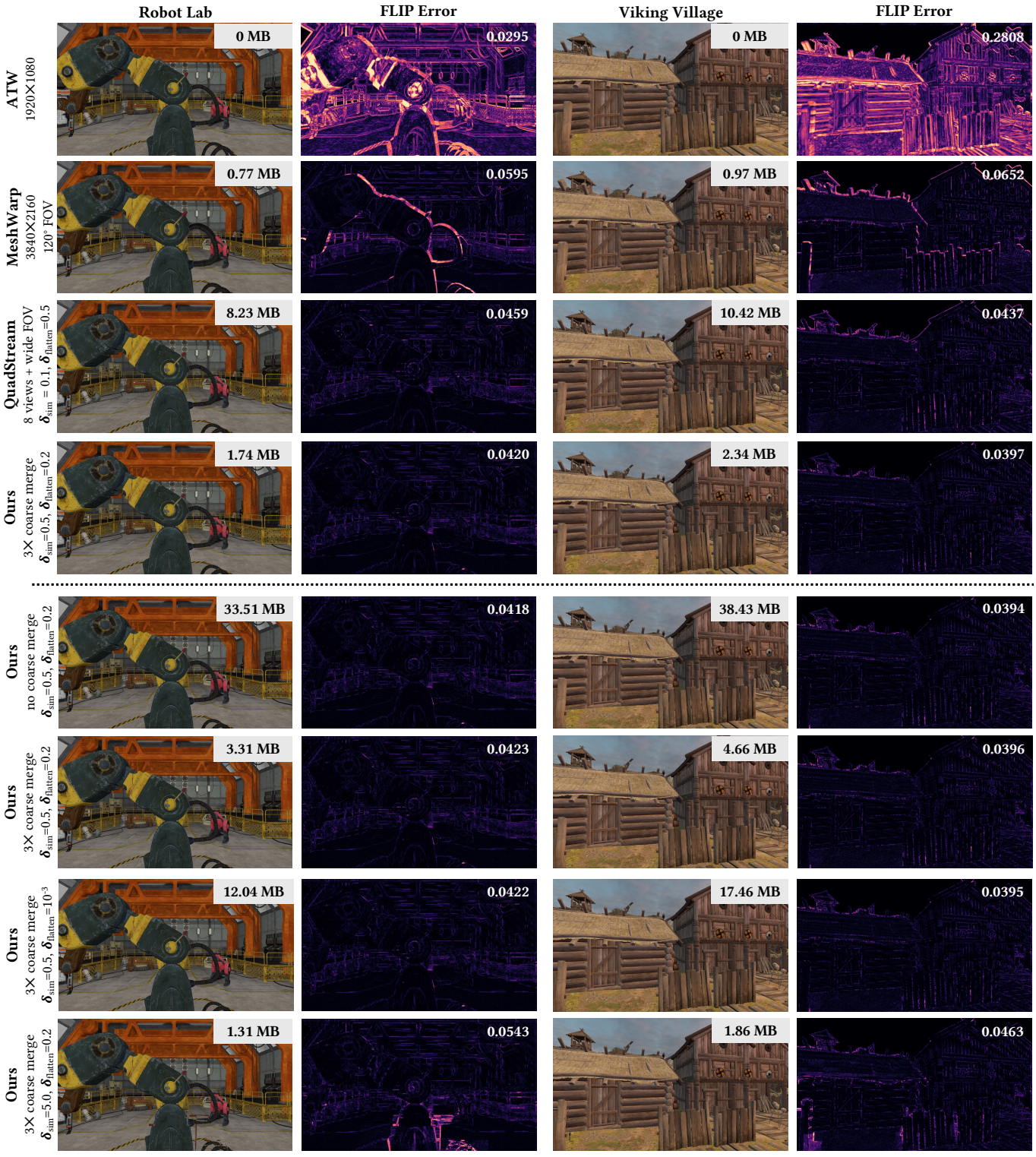


Fig. 13. **Top half:** Visual comparisons of tested methods from a novel view. MB denotes the per-frame *geometry* size (note that ATW does not send geometry). Our data size is shown as the size of a *reference frame*. **Bottom half:** Algorithm parameters and their impact on quality and data size. We can adjust the level of quad merging by adjusting the plane similarity threshold (δ_{sim}), the flattening threshold (δ_{flatten}), and the amount of simplify passes we apply coarse merging (which essentially downsamples the G-Buffer in non-edge regions).

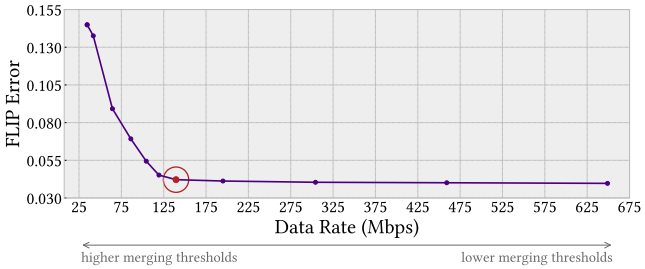


Fig. 14. Effect of quad merging on visual quality and data rate for the Robot Lab scene (50 cm viewcell). Trials vary plane similarity and flattening thresholds, impacting geometric resolution and compression efficiency. Parameter values between adjacent points differ by a factor of 2. The chosen operating point (δ_{sim}=0.5 and δ_{flatten}=0.2) is highlighted.

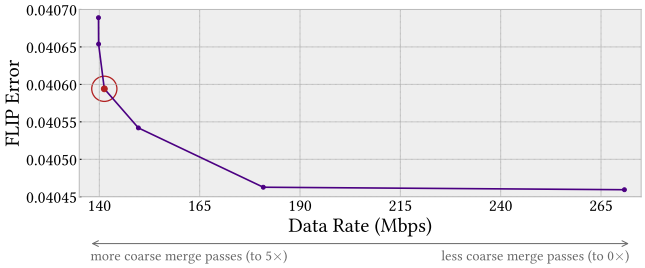


Fig. 15. Effect of applying our coarse merging heuristic on visual quality and data rate for the Robot Lab scene (50 cm viewcell). The chosen operating point (3× coarse merging) is highlighted. As we increase the number of coarse merge passes, note that FLIP error exhibits minimal improvement (~0.0002 from 0 to 5×), while data rate undergoes a significant reduction.

disocclusion events, resulting in lower perceptual quality compared to both QuadStream and our method. Furthermore, its quality significantly degrades with higher latency as disocclusions become more frequent and pronounced.

QuadStream offers high visual quality, outperforming ATW and MeshWarp, particularly under higher latency conditions. Its robustness stems from its use of high-resolution geometry and explicit handling of disocclusions, which preserve detail and visual consistency across frames. While MeshWarp performs well at low latency, the lack of disocclusions leads to noticeable quality drops as latency increases. At low latency, QuadStream viewcell size has limited influence on visual quality, as smaller viewcells sufficiently cover potential viewpoints. In contrast, at higher latency, increasing the viewcell size improves visual quality by covering a wider range of potential viewpoints, mitigating the effects of frame delay. Unfortunately, QuadStream demands significantly higher data rates, often an order of magnitude greater than other methods, exceeding 1 Gbps for nearly all traces.

Our method delivers visual quality comparable to, and often exceeding, QuadStream, while significantly reducing bandwidth usage. This is due to improved reconstruction quality and more effective capture of potentially visible content. Despite employing aggressive downsampling to improve efficiency, we observe a minimal impact on perceived visual quality, as texture detail tends to mask geometric approximations. Our approach also demonstrates robustness across varying network latencies, with only minor degradations in quality at higher latencies. Similarly to QuadStream, larger view cells at low latency contribute to marginal quality improvements, since small view cells capture all potential client viewpoints. However, in our method, increasing the viewcell size tends to expose additional quads on hidden surfaces, which in turn facilitates greater merging opportunities; as a result, the associated bandwidth increase is small. Our bandwidth usage is modest for some scenes, but can be higher for more complex ones, ranging from 92 to 635 Mbps, depending on scene complexity. In general, our method maintains a balance between visual fidelity, latency masking, and data efficiency, outperforming all other approaches in overall performance.

4.3.1 Adapting to Fit the Best QoE. Geometry decimation plays a key role in balancing visual quality and data efficiency. The lower

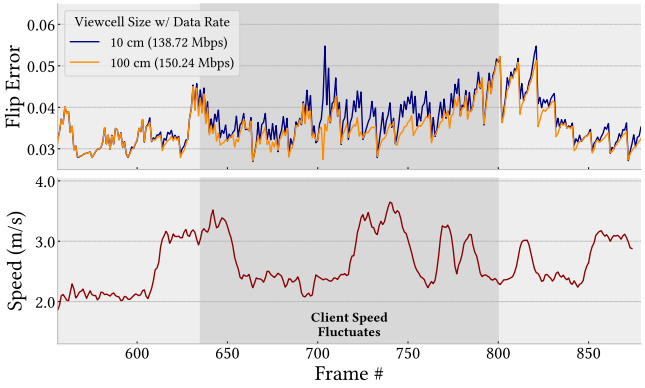


Fig. 16. Effect of viewcell size on latency masking for our method. For this trace (Robot Lab with 50 ± 20 ms delay), rapid viewpoint changes between frames 635–800 are better handled with a larger viewcell, reducing FLIP error but increasing data transmission.

half of Fig. 13 visualizes how the varying quad merging thresholds impact image quality. Higher thresholds yield better compression but can introduce artifacts, while lower thresholds preserve detail at the cost of increased bandwidth. Fig. 14 shows the quality-data rate trade-off of our method. Note that we jointly vary plane similarity and flattening thresholds, as they are interdependent. The figure shows that the impact of quad merging on visual quality follows a roughly exponential decay, and we select our operating point near the crossover where quality begins to degrade sharply. Fig. 15 highlights the effect of our course merging strategy. From this, we choose to apply coarser merging in the first three simplify passes, around where improvements in data rate start to show diminishing returns. Adapting these decimation parameters dynamically to adapt to network conditions is a promising direction for future work.

As shown in Fig. 12, increasing the viewcell size in our system results in only a modest increase in data rate, ranging from 4 to 30 Mbps, even with a 4× enlargement of the view sphere. In contrast, QuadStream exhibits a significantly steeper increase—ranging from 150 to 1000 Mbps—as the view box size grows. This highlights viewcell size as a key factor in balancing latency masking, visual quality,

and bandwidth. Fig. 16 demonstrates that dynamically adjusting the viewcell size—rather than using a static configuration—can help mitigate latency during rapid user motion, albeit with increased data usage. This underscores a fundamental trade-off between latency robustness and bandwidth efficiency, suggesting adaptive viewcell sizing as a promising strategy for optimizing performance based on application needs (see Sec. 5).

5 DISCUSSION

Geometry-based streaming approaches offer a number of benefits for remote rendering. First, they are scalable, as future increases in VR/AR display resolutions will drive up demand for pixel shading, necessitating the need for remote rendering; meanwhile, vertex processing can remain tied to the transmitted geometry, which can be quantized or decimated with less impact on visual quality. In addition, they support future varifocal VR displays [Ebner et al. 2022; Qin et al. 2023], enabling clients to render multiple focus planes locally at low latency. Compared to transmitting quantized triangle primitives, proxy-based geometry streaming can significantly reduce data rates, making real-time streaming more feasible under real-world network conditions. Our work is a step toward making these techniques practical with realistic interconnects.

QoE controls are an important factor to consider when designing new representations for streaming. We show that we can decimate geometry without much loss in quality and can more efficiently compress existing quad-based streaming representations. This is critical when considering that geometry data is often masked by texture quality, and yet, due to its complex nature, it often consumes considerable bandwidth. Our approach enables an application designer to define trade-offs based on the characteristics of their application that could downsample textures, geometry, and/or viewing volume size to best fit their application. For example, a high-speed driving game might prefer to increase headbox size to better mask latency, while a less dynamic game might bias the system towards texture and geometry quality. In the worst case, our system can simply degrade to ATW or even MeshWarp if bandwidth is not available.

5.0.1 Limitations and future work. Our quad-based representation, similar to QuadStream, approximates curved edges using square-shaped quads, leading to aliasing-like effects near object boundaries. While client-side anti-aliasing can help reduce these artifacts, they come at a performance cost. Accurately representing smooth edges remains a key challenge for quad-based methods.

Entirely gap-free reconstruction can be challenging, especially when the client deviates significantly from the center view. In these cases, small gaps may appear, occasionally causing flickering artifacts when applying residual frames, particularly in scenes with complex occluders. These gaps can lead to quads from the wide FOV set appearing in the main view, resulting in occasional visual artifacts. To address this, we introduce a boundary expansion step during residual quad generation to help fill gaps and suppress such issues, though its effectiveness may be limited in certain cases.

Our method struggles with view-dependent effects like reflections, refractions, and dynamic lighting, which require real-time updates based on the client's viewpoint. Network latency delays these updates, making it difficult to render such effects accurately in a server-client architecture. Additionally, volumetric effects such as smoke or clouds can be difficult to represent accurately with a limited number of quads. Future work could explore hybrid approaches that combine local rendering of view-dependent effects with server-side geometry streaming. Additionally, our system could be extended to support alpha-blended multi-layered meshes, simulating view-dependent effects, similar to [Broxton et al. 2020].

A key challenge is improving geometry visibility determination to minimize false positives and negatives from a single view. Although EDP works well, it can sometimes miss disoccluded regions. Adjusting thresholds can mitigate this issue, but increases data size due to false positives. A more adaptive and precise approach is needed to balance accuracy and efficiency.

A promising direction for future research would be the development of a fully rate-adaptive scheme that takes advantage of the tunable parameters presented in this work. Such a system could dynamically balance viewcell size, quad merging thresholds, and temporal compression settings based on current bandwidth conditions and rendering requirements. This could enable real-time optimization of data rates without compromising visual fidelity, particularly in constrained network environments.

Split rendering techniques have potential applications beyond XR headsets, such as cloud-based 3D gaming and volumetric video streaming. Although our current pipeline supports high-speed interconnects (100–500 Mbps), broader deployment over typical consumer internet connections, especially in bandwidth-constrained or variable environments, requires aggressive bitrate reduction, targeting sub-100 Mbps to ensure accessibility at scale.

6 CONCLUSION

We present a quad-based geometry streaming system for splitting complex scene rendering between a server and a lightweight client, such as AR/VR headsets. Our approach uses Effective Depth Peeling to capture visible and potentially visible geometry from within a viewing sphere, enabling robust occlusion handling in dynamic scenes. Temporal compression exploits frame-to-frame redundancy, while a video-friendly texture atlas improves shading compression and reduces video size.

Evaluated on real-world game scenes, our method handles varying geometric and lighting complexity, outperforming prior quad-based techniques in data size while maintaining visual quality. It achieves data rates comparable to depth- or mesh-based approaches, while effectively capturing disocclusion events.

ACKNOWLEDGMENTS

This work was supported in part by the NSF under Grant No. CNS1956095, the NSF Graduate Research Fellowship under Grant No. DGE2140739, and Bosch Research. Any opinion, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D. Fairchild. 2020. FLIP: A Difference Evaluator for Alternating Images. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3, 2 (2020), 15:1–15:23. https://doi.org/10.1145/3406183

C. Andujar, P. Brunet, A. Chica, I. Navazo, J. Rossignac, and A. Vinacua. 2004. Computing Maximal Tiles and Application to Impostor-Based Simplification. *Computer Graphics Forum* 23, 3 (2004), 401–410. https://doi.org/10.1111/j.1467-8659.2004.00771.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2004.00771.x

Benjamin Attal, Selena Ling, Aaron Gokaslan, Christian Richardt, and James Tompkin. 2020. MatryODShka: Real-time 6DoF Video View Synthesis Using Multi-sphere Images. In *Computer Vision – ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part I* (Glasgow, United Kingdom). Springer-Verlag, Berlin, Heidelberg, 441–459. https://doi.org/10.1007/978-3-030-58452-8_26

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* (Singapore, Singapore) *(MobiSys '16)*. Association for Computing Machinery, New York, NY, USA, 291–304. https://doi.org/10.1145/2906388.2906418

Michael Broxton, John Flynn, Ryan Overbeck, Daniel Erickson, Peter Hedman, Matthew Duvall, Jason Dourgarian, Jay Busch, Matt Whalen, and Paul Debevec. 2020. Immersive light field video with a layered mesh representation. *ACM Trans. Graph.* 39, 4, Article 86 (Aug. 2020), 15 pages. https://doi.org/10.1145/3386569.3392485

Zhiqin Chen, Thomas Funkhouser, Peter Hedman, and Andrea Tagliasacchi. 2023. MobileNeRF: Exploiting the Polygon Rasterization Pipeline for Efficient Neural Field Rendering on Mobile Architectures. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 16569–16578. https://doi.org/10.1109/CVPR52729.2023.01590

Daniel Cohen-Or, Yair Mann, and Shachar Fleishman. 1999. Deep compression for streaming texture intensive animations. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. ACM Press/Addison-Wesley Publishing Co., USA, 261–267. https://doi.org/10.1145/311535.311564

Cyril Crassin, David Luebke, Michael Mara, Morgan McGuire, Brent Oster, Peter Shirley, Peter-Pike Sloan, and Chris Wyman. 2015. CloudLight: A System for Amortizing Indirect Lighting in Real-Time Rendering. *Journal of Computer Graphics Techniques (JCGT)* 4, 4 (15 October 2015), 1–27. http://jcgt.org/published/0004/04/01/

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* (Florence, Italy) *(MobiSys '15)*. Association for Computing Machinery, New York, NY, USA, 121–135. https://doi.org/10.1145/2742647.2742657

Paul Debevec, Yizhou Yu, and George Borshukov. 1998. Efficient View-Dependent Image-Based Rendering with Projective Texture-Mapping. In *Rendering Techniques '98*, George Drettakis and Nelson Max (Eds.). Springer Vienna, Vienna, 105–116.

Xavier Décoret, Frédo Durand, François X. Sillion, and Julie Dorsey. 2003. Billboard clouds for extreme model simplification. *ACM Trans. Graph.* 22, 3 (July 2003), 689–696. https://doi.org/10.1145/882262.882326

Piotr Didyk, Elmar Eisemann, Tobias Ritschel, Karol Myszkowski, and Hans-Peter Seidel. 2010a. Perceptually-motivated Real-time Temporal Upsampling of 3D Content for High-refresh-rate Displays. *Comput. Graph. Forum* 29 (05 2010), 713–722. https://doi.org/10.1111/j.1467-8659.2009.01641.x

Piotr Didyk, Tobias Ritschel, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel. 2010b. Adaptive Image-space Stereo View Synthesis. In *Vision, Modeling, and Visualization (2010)*, Reinhard Koch, Andreas Kolb, and Christof Rezk-Salama (Eds.). The Eurographics Association. https://doi.org/10.2312/PE/VMV/VMV10/299-306

Piotr Didyk, Tobias Ritschel, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel. 2010c. Adaptive Image-space Stereo View Synthesis. In *Vision, Modeling, and Visualization (2010)*, Reinhard Koch, Andreas Kolb, and Christof Rezk-Salama (Eds.). The Eurographics Association. https://doi.org/10.2312/PE/VMV/VMV10/299-306

Christoph Ebner, Shohei Mori, Peter Mohr, Yifan Peng, Dieter Schmalstieg, Gordon Wetzstein, and Denis Kalkofen. 2022. Video See-Through Mixed Reality with Focus Cues. *IEEE Transactions on Visualization and Computer Graphics* 28, 5 (2022), 2256–2266.

Epic Games. 2017. Unreal Engine Sun Temple, Open Research Content Archive (ORCA). http://developer.nvidia.com/orca/epic-games-sun-temple `http://developer.nvidia.com/orca/epic-games-sun-temple`.

Cass Everitt. 2001. *Interactive Order-Independent Transparency*. White Paper 6. NVIDIA. 7 pages. https://www.nvidia.com/en-us/drivers/Interactive-Order-Transparency/

Michael Garland and Paul S. Heckbert. 2023. *Surface Simplification Using Quadric Error Metrics* (1 ed.). Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3596711.3596727

Schaufler Gernot. 1998. Image-based object representation by layered impostors. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (Taipei, Taiwan) *(VRST '98)*. Association for Computing Machinery, New York, NY, USA, 99–104. https://doi.org/10.1145/293701.293714

Google. 2019. Google Stadia. https://stadia.google.com

J. Hladky, H.P. Seidel, and M. Steinberger. 2021. SnakeBinning: Efficient Temporally Coherent Triangle Packing for Shading Streaming. *Computer Graphics Forum* 40, 2 (2021), 475–488. https://doi.org/10.1111/cgf.142648 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.142648

J. Hladky, H. P. Seidel, and M. Steinberger. 2019. Tessellated Shading Streaming. *Computer Graphics Forum* 38, 4 (2019), 171–182. https://doi.org/10.1111/cgf.13780 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13780

Jozef Hladky, Michael Stengel, Nicholas Vining, Bernhard Kerbl, Hans-Peter Seidel, and Markus Steinberger. 2022. QuadStream: A Quad-Based Scene Streaming Architecture for Novel Viewpoint Reconstruction. *ACM Trans. Graph.* 41, 6, Article 233 (Nov. 2022), 13 pages. https://doi.org/10.1145/3550454.3555524

Stefan Jeschke and Michael Wimmer. 2002. *Textured Depth Meshes for Real-Time Rendering of Arbitrary Scenes*. Technical Report TR-186-2-02-13. Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria. https://www.cg.tuwien.ac.at/research/publications/2002/Jeschke-2002-TDM/ human contact: technical-report@cg.tuwien.ac.at.

Brian Karis, Rune Stubbe, and Graham Wihlidal. 2021. A Deep Dive into Nanite Virtualized Geometry. In *ACM SIGGRAPH*.

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkuehler, and George Drettakis. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Trans. Graph.* 42, 4, Article 139 (July 2023), 14 pages. https://doi.org/10.1145/3592433

Janghun Kim and Sungkil Lee. 2023. Potentially Visible Hidden-Volume Rendering for Multi-View Warping. *ACM Trans. Graph.* 42, 4, Article 86 (July 2023), 11 pages. https://doi.org/10.1145/3592108

Babis Koniaris, Maggie Kosek, David Sinclair, and Kenny Mitchell. 2017. Real-time Rendering with Compressed Animated Light Fields. In *Proceedings of the 43rd Graphics Interface Conference* (Edmonton, Alberta, Canada) *(GI '17)*. Canadian Human-Computer Communications Society, Waterloo, CAN, 33–40.

Babis Koniaris, Maggie Kosek, David Sinclair, and Kenny Mitchell. 2018. GPU-accelerated depth codec for real-time, high-quality light field reconstruction. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 1, Article 3 (July 2018), 15 pages. https://doi.org/10.1145/3203193

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 *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (Snowbird, Utah, USA) *(MobiCom '17)*. Association for Computing Machinery, New York, NY, USA, 409–421. https://doi.org/10.1145/3117811.3117815

Puneet Lall, Silviu Borac, Dave Richardson, Matt Pharr, and Manfred Ernst. 2018. View-Region Optimized Image-Based Scene Simplification. *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2, Article 26 (Aug. 2018), 22 pages. https://doi.org/10.1145/3233311

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* (Florence, Italy) *(MobiSys '15)*. Association for Computing Machinery, New York, NY, USA, 151–165. https://doi.org/10.1145/2742647.2742656

Edward Lu, Sagar Bharadwaj, Mallesham Dasari, Connor Smith, Srinivasan Seshan, and Anthony Rowe. 2023. RenderFusion: Balancing Local and Remote Rendering for Interactive 3D Scenes . In *2023 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. IEEE Computer Society, Los Alamitos, CA, USA, 312–321. https://doi.org/10.1109/ISMAR59233.2023.00046

Magic Leap. 2024. Magic Leap Remote Rendering. https://developer-docs.magicleap.cloud/docs/guides/remote-rendering

Zander Majercik, Jean-Philippe Guertin, Derek Nowrouzezahrai, and Morgan McGuire. 2019. Dynamic Diffuse Global Illumination with Ray-Traced Irradiance Fields. *Journal of Computer Graphics Techniques (JCGT)* 8, 2 (5 June 2019), 1–30. http://jcgt.org/published/0008/02/01/

Saswat Subhajyoti Mallick, Rahul Goel, Bernhard Kerbl, Markus Steinberger, Francisco Vicente Carrasco, and Fernando De La Torre. 2024. Taming 3DGS: High-Quality Radiance Fields with Limited Resources. In *SIGGRAPH Asia 2024 Conference Papers (SA '24)*. Association for Computing Machinery, New York, NY, USA, Article 2, 11 pages. https://doi.org/10.1145/3680528.3687694

A. Mammen. 1989. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications* 9, 4 (1989), 43–55. https://doi.org/10.1109/38.31463

William R. Mark, Leonard McMillan, and Gary Bishop. 1997. Post-rendering 3D warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics* (Providence, Rhode Island, USA) *(I3D '97)*. Association for Computing Machinery, New York, NY, USA, 7–ff. https://doi.org/10.1145/253284.253292

Morgan McGuire. 2017. Computer Graphics Archive. https://casual-effects.com/data `https://casual-effects.com/data`.

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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 923–937. https://doi.org/10.1145/3373376.3378516

Inc. Meta Platforms. 2016. Zstandard. GitHub repository. https://github.com/facebook/zstd

Inc. Meta Platforms. 2021. Air Link. https://www.meta.com/blog/quest/introducing-oculus-air-link-a-wireless-way-to-play-pc-vr-games-on-oculus-quest-2-plus-infinite-office-updates-support-for-120-hz-on-quest-2-and-more/

Inc. Meta Platforms. 2024. Meta Horizon Hyperscape Demo. https://www.meta.com/experiences/meta-horizon-hyperscape-demo/7972066712871980/

Microsoft. 2020. Xbox Cloud Gaming. https://www.xbox.com/en-us/play

Ben Mildenhall, Pratul P. Srinivasan, Rodrigo Ortiz-Cayon, Nima Khademi Kalantari, Ravi Ramamoorthi, Ren Ng, and Abhishek Kar. 2019. Local Light Field Fusion: Practical View Synthesis with Prescriptive Sampling Guidelines. *ACM Transactions on Graphics (TOG)* (2019).

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2021. NeRF: representing scenes as neural radiance fields for view synthesis. *Commun. ACM* 65, 1, 99–106. https://doi.org/10.1145/3503250

D. Mlakar, M. Steinberger, and D. Schmalstieg. 2024. End-to-End Compressed Meshlet Rendering. *Computer Graphics Forum* 43, 1 (2024), e15002. https://doi.org/10.1111/cgf.15002 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.15002

Wieland Morgenstern, Florian Barthel, Anna Hilsmann, and Peter Eisert. 2025. Compact 3D Scene Representation via Self-Organizing Gaussian Grids. In *Computer Vision – ECCV 2024*. Springer Nature Switzerland, Cham, 18–34. https://doi.org/10.1007/978-3-031-73013-9_2

MPEG. 2023. MPEG Immersive Video. https://mpeg-miv.org/

Joerg H. Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. 2018. Shading atlas streaming. *ACM Trans. Graph.* 37, 6, Article 199 (Dec. 2018), 16 pages. https://doi.org/10.1145/3272127.3275087

Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.* 41, 4, Article 102 (July 2022), 15 pages. https://doi.org/10.1145/3528223.3530127

NVIDIA. 2020. GeForce NOW. https://www.nvidia.com/en-us/geforce-now/

NVIDIA. 2023. NVIDIA CloudXR. https://www.nvidia.com/en-us/design-visualization/solutions/cloud-xr/

Oculus. 2016. Asynchronous Space Warp. https://developers.meta.com/horizon/blog/asynchronous-spacewarp/

Fabrizio Pece, Jan Kautz, and Tim Weyrich. 2011. Adapting standard video codecs for depth streaming. In *Proceedings of the 17th Eurographics Conference on Virtual Environments & Third Joint Virtual Reality* (Nottingham, UK) *(EGVE - JVRC'11)*. Eurographics Association, Goslar, DEU, 59–66.

Eric Penner and Li Zhang. 2017. Soft 3D reconstruction for view synthesis. *ACM Trans. Graph.* 36, 6, Article 235 (Nov. 2017), 11 pages. https://doi.org/10.1145/3130800.3130855

Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. 2000. Surfels: surface elements as rendering primitives. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '00)*. ACM Press/Addison-Wesley Publishing Co., USA, 335–342. https://doi.org/10.1145/344779.344936

Yingsi Qin, Wei-Yu Chen, Matthew O'Toole, and Aswin C. Sankaranarayananan. 2023. Split-Lohmann Multifocal Displays. *ACM Transactions on Graphics / SIGGRAPH* 31 (Aug 2023). https://doi.org/10.1145/3592110

Bernhard Reinert, Johannes Kopf, Tobias Ritschel, Eduardo Cuervo, David Chu, and Hans-Peter Seidel. 2016. Proxy-guided Image-based Rendering for Mobile Devices. *Comput. Graph. Forum* 35, 7 (Oct. 2016), 353–362.

Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. 2001. Texture mapping progressive meshes. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 409–416. https://doi.org/10.1145/383259.383307

Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. 1998. Layered depth images. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98)*. Association for Computing Machinery, New York, NY, USA, 231–242. https://doi.org/10.1145/280814.280882

Shu Shi and Cheng-Hsin Hsu. 2015. A Survey of Interactive Remote Rendering Systems. *ACM Comput. Surv.* 47, 4, Article 57 (may 2015), 29 pages. https://doi.org/10.1145/2719921

Jamie Shotton, Ben Glocker, Christopher Zach, Shahram Izadi, Antonio Criminisi, and Andrew Fitzgibbon. 2013. Scene Coordinate Regression Forests for Camera Relocalization in RGB-D Images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2930–2937. https://doi.org/10.1109/CVPR.2013.377

Liangchen Song, Anpei Chen, Zhong Li, Zhang Chen, Lele Chen, Junsong Yuan, Yi Xu, and Andreas Geiger. 2023. NeRFPlayer: A Streamable Dynamic Scene Representation with Decomposed Neural Radiance Fields. *IEEE Transactions on Visualization and Computer Graphics* 29, 5 (2023), 2732–2742. https://doi.org/10.1109/TVCG.2023.3247082

Michael Stengel, Zander Majercik, Benjamin Boudaoud, and Morgan McGuire. 2021. A distributed, decoupled system for losslessly streaming dynamic light probes to thin clients. (2021), 159–172. https://doi.org/10.1145/3458305.3463379

Unity Technologies. 2018. Robot Lab. https://assetstore.unity.com/packages/essentials/tutorial-projects/robot-lab-unity-4x-7006

Unity Technologies. 2022. Viking Village URP. https://assetstore.unity.com/packages/essentials/tutorial-projects/viking-village-urp-29140

Valve. 2015. Steam Link. https://store.steampowered.com/remoteplay

J. M. P. van Waveren. 2016. The Asynchronous Time Warp for Virtual Reality on Consumer Hardware. In *Proceedings of the 22nd ACM Conference on*

Virtual Reality Software and Technology (Munich, Germany) *(VRST '16)*. Association for Computing Machinery, New York, NY, USA, 37–46. https://doi.org/10.1145/2993369.2993375

Philip Voglreiter, Bernhard Kerbl, Alexander Weinrauch, Joerg Hermann Mueller, Thomas Neff, Markus Steinberger, and Dieter Schmalstieg. 2023. Trim Regions for Online Computation of From-Region Potentially Visible Sets. *ACM Trans. Graph.* 42, 4, Article 85 (July 2023), 15 pages. https://doi.org/10.1145/3592434

Peng Wang, Yuan Liu, Zhaoxi Chen, Lingjie Liu, Ziwei Liu, Taku Komura, Christian Theobalt, and Wenping Wang. 2023. F2-NeRF: Fast Neural Radiance Field Training with Free Camera Trajectories . (June 2023), 4150–4159. https://doi.org/10.1109/CVPR52729.2023.00404

Penghao Wang, Zhirui Zhang, Liao Wang, Kaixin Yao, Siyuan Xie, Jingyi Yu, Minye Wu, and Lan Xu. 2024. V^3: Viewing Volumetric Videos on Mobiles via Streamable 2D Dynamic Gaussians. *ACM Transactions on Graphics (TOG)* 43, 6 (2024), 1–13.

Guanjun Wu, Taoran Yi, Jiemin Fang, Lingxi Xie, Xiaopeng Zhang, Wei Wei, Wenyu Liu, Qi Tian, and Xinggang Wang. 2024. 4D Gaussian Splatting for Real-Time Dynamic Scene Rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 20310–20320.

Tinghui Zhou, Richard Tucker, John Flynn, Graham Fyffe, and Noah Snavely. 2018. Stereo magnification: learning view synthesis using multiplane images. *ACM Trans. Graph.* 37, 4, Article 65 (July 2018), 12 pages. https://doi.org/10.1145/3197517.3201323

Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. 2001. Surface splatting. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. Association for Computing Machinery, New York, NY, USA, 371–378. https://doi.org/10.1145/383259.383300