QuadStream: A Quad-Based Scene Streaming Architecture for Novel Viewpoint Reconstruction

JOZEF HLADKY, Max Planck Institute for Informatics, Germany and NVIDIA, Germany
MICHAEL STENGEL, NVIDIA, USA
NICHOLAS VINING, University of British Columbia, Canada and NVIDIA, Canada
BERNHARD KERBL, TU Wien, Austria
HANS-PETER SEIDEL, Max Planck Institute for Informatics, Germany
MARKUS STEINBERGER, Graz University of Technology, Austria

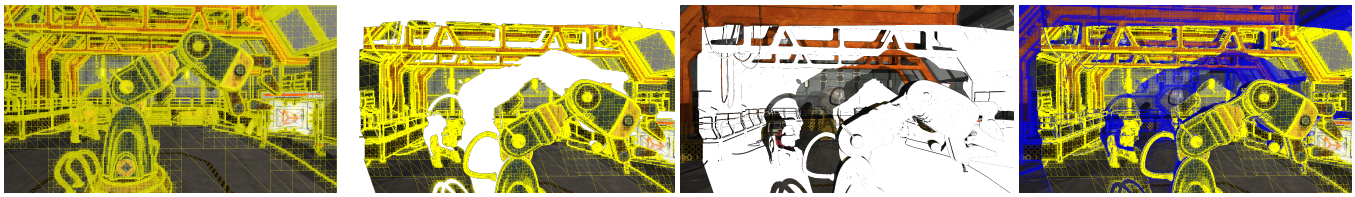


Fig. 1. The QuadStream architecture enables a server to rasterize and stream a scene to a thin client; the client can then rasterize the scene from a novel, unknown viewpoint. Unlike previous methods, our approach robustly handles disocclusions, view-dependent lighting, and transparent surfaces. From left to right: we decompose a rendered G-Buffer into *quad proxies*, consisting of a proxy plane and a series of projected quad surface elements (surfels); however, these may not provide enough information for the client to render the scene in the presence of disocclusions at novel views, causing visual artifacts (see the missing area behind the robot arm in white). We therefore augment the QuadStream with quad proxies selectively added from other views within the same view cell, allowing the client to fully reconstruct the scene under motion (additional quad proxies shown in blue).

Streaming rendered 3D content over a network to a thin client device, such as a phone or a VR/AR headset, brings high-fidelity graphics to platforms where it would not normally possible due to thermal, power, or cost constraints. Streamed 3D content must be transmitted with a representation that is both robust to latency and potential network dropouts. Transmitting a video stream and reprojecting to correct for changing viewpoints fails in the presence of disocclusion events; streaming scene geometry and performing high-quality rendering on the client is not possible on limited-power mobile GPUs. To balance the competing goals of disocclusion robustness and minimal client workload, we introduce *QuadStream*, a new streaming content representation that reduces motion-to-photon latency by allowing clients to efficiently render novel views without artifacts caused by disocclusion events. Motivated by traditional macroblock approaches to video codec design, we decompose the scene seen from positions in a *view cell* into a series of *quad proxies*, or view-aligned quads from multiple views. By operating on a rasterized G-Buffer, our approach is independent of the representation used for the scene itself; the resulting *QuadStream* is an approximate geometric representation of the scene that can be reconstructed by a thin client to render both the current view and nearby adjacent views. Our technical contributions are an efficient parallel quad generation, merging, and packing strategy for proxy views covering potential client movement in a scene; a packing and encoding strategy that allows masked quads with depth information to be transmitted as a frame-coherent stream; and an efficient rendering approach for rendering our QuadStream representation into entirely novel views on thin clients. We show that our approach achieves superior quality compared both to video data streaming methods, and to geometry-based streaming.

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

Additional Key Words and Phrases: texture-space shading, object space shading, shading atlas, streaming, temporal coherence, virtual reality

ACM Reference Format:
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 (December 2022), 13 pages. https://doi.org/10.1145/3550454.3555524

1 INTRODUCTION

Streaming rendering has been used to offer 3D graphics on mobile phones or other devices with limited GPU capability; to stream graphics to VR/AR head-mounted devices where a high-end GPU is not available due to thermal and power constraints; and to offer cloud gaming services such as NVIDIA's GeForce Now [NVIDIA 2021] or Google's Stadia [Google 2019] that target users without high-end hardware. At the same time, streaming rendering introduces new challenges. First, 3D content must be transmitted over a network, and hence must account for transmission latency and jitter, as well as potential connection dropouts. Second, the rendering workload must be separated between two or more disjoint

Authors' addresses: Jozef Hladky, Max Planck Institute for Informatics, Germany and NVIDIA, Germany, jhladky@mpi-inf.mpg.de; Michael Stengel, NVIDIA, USA, mstengel@nvidia.com; Nicholas Vining, University of British Columbia, Canada and NVIDIA, Canada, nvining@cs.ubc.ca; Bernhard Kerbl, TU Wien, Austria, kerbl@cg.tuwien.ac.at; Hans-Peter Seidel, Max Planck Institute for Informatics, Germany, hpseidel@mpi-sb.mpg.de; Markus Steinberger, Graz University of Technology, Austria, steinberger@icg.tugraz.at.

0730-0301/2022/12-ART233
https://doi.org/10.1145/3550454.3555524

physical locations; while servers in a streaming rendering environment are often very powerful, thin clients are commonly limited to simple processing to reduce battery consumption and thermal emission. Nonetheless, mobile devices now support a rich feature set for real-time rendering, making arbitrary splits in the rendering pipeline between a server and a client possible. We introduce *QuadStream*, a novel geometry and shading streaming technique which allows a mobile thin client to reconstruct novel viewpoints on the fly, including disocclusions (Fig. 1), and which provides high-fidelity streaming rendering while reducing the perceived latency.

The naive approach to streaming rendering is to run a traditional application instance in the cloud or on a server; the application instance receives input from the client, renders the next frame, encodes it using a hardware encoder [Ranganathan et al. 2021], and delivers the output as a video stream to the client. While this approach requires only a video decoder on the client, it implies a stable connection and introduces high *motion-to-photon latency*, defined as the time between the user sending an input and the updated image ultimately being displayed on the client screen. When the user receives updated renderings from the server, it is often already out-of-date, since the user has generated new input and moved the viewpoint in the meantime. A key disadvantage of video streaming is that it fails in the presence of *disocclusion events*—when previously occluded pieces of geometry become visible due to a change in view. While warping techniques [OculusVR 2018] can correct for rotational offsets, they cannot recover from disocclusion events as the occluded geometry is not present in the previous view. Attempts to inpaint missing data increase computation costs and often generate visible artifacts. On the other extreme, servers may only provide scene information, with all rendering tasks being performed on the client. While this approach is resilient to connection outages and can lead to low perceived latency, it limits visual quality to the capabilities of the client, voiding the benefits of cloud rendering.

Our *QuadStream* technique allows thin clients to reconstruct novel viewpoints on the fly, including viewpoints with disocclusions and dynamic geometry, and reduces motion-to-photon latency while providing high-fidelity rendering. We achieve this by transmitting an approximate geometric scene representation to the client that is much simpler than the full scene geometry, yet captures enough geometric fidelity and shading information to allow a thin client to reconstruct nearby novel viewpoints ad hoc. Classical work on geometry simplification [Garland and Heckbert 1997; Hoppe 1996; Sander et al. 2001] has focused on the problem of offline geometry simplification; however, these methods are too slow for real-time and too inflexible for dynamic scene geometry. We therefore take inspiration from traditional macroblock approaches to video codec design, and decompose a scene as seen from a *view cell* into a selection of fitted quads, or *quad proxies*, constructed on the fly from a G-Buffer. While we assume the scene is rendered from a given viewpoint, we augment the collection of quad proxies with additional quad proxies taken from other locations in the *view cell*—the space of plausible locations around the viewpoint where the client view may ultimately be. We then hierarchically merge quad proxies for efficient packing and streaming using a series of compute shaders, resulting in a flat buffer of proxy geometric data and a packed texture atlas of shading information that can be compressed with off-the-shelf video compression techniques. The resulting QuadStream, once received by the thin client, can be decompressed and rendered from a novel viewpoint within the same view cell via simple rasterization.

Our approach is independent of the geometric representation used for rendering on the server, unlike previous PVS-based approaches; this enables streaming applications regardless of geometric scene complexity. Our main performance factor is the resolution of the G-Buffer. QuadProxies can be efficiently streamed as a video stream enhanced with depth information, require little additional shading data, and are efficient to render on thin clients. For simple scenes, we compete with prior work in terms of achieved standard industry metrics (DSSIM [Wang and Li 2010], and FLIP [Andersson et al. 2020]), outperforming them when taking into account the number of samples that must be computed and streamed in each approach. For complex scenes, we significantly outperform prior work, as our approach is mostly independent of the scene complexity.

2 RELATED WORK

A recent overview of methods for splitting rendering between a server and a client is provided by Stengel et al. [2021]. We focus on approaches that have been applied to different streaming scenarios, organized in order of increasing client workload.

Image-Based Streaming. Current cloud-gaming services [Google 2019; NVIDIA 2021] rely on a simple video stream from the cloud to the client. Image-based rendering approaches can be used to adjust the video stream before display on the client to adjust for the most recent camera/head movement. When no depth information is available, homography can be used to warp the entire video stream on a plane [OculusVR 2018] which requires the server to render with increased field-of-view to avoid out-of-view artifacts. Adding a depth buffer to the stream increases possibilities: pixels can be individually warped [Chen and Williams 1993], a simple geometric proxy can be created [Mark et al. 1997], a grid can be used for warping [Didyk et al. 2010a,b], and a search in the received frame can be employed to increase quality [Bowles et al. 2012; Yang et al. 2011]. These approaches fail to handle disocclusions which are not covered by a single video stream; while multiple depth-augmented streams can be delivered to the client [Shi et al. 2012], this significantly increases bandwidth due to duplicated content being streamed.

Static Proxies. Many previous works have simplified the problem by restricting streaming to static geometry, then run extensive preprocessing to generate suitable image-based proxies: this includes view-dependent texture maps [Cohen-Or et al. 1999], imposters [Boos et al. 2016; Teler and Lischinski 2001], simplied geometric proxies[Reinert et al. 2016], surfel representations [Mann and Cohen-Or 1997; Pfister et al. 2000], and billboard clouds [Décoret et al. 2003; Lall et al. 2018]. Our approach is similar to the technique proposed by Lall et al. [2018], which computes a novel scene representation consisting of shaded quads that pack nicely into a video stream; however, our approach generates a quad-based representation in real-time for dynamically changing geometry and shading.

Geometry-Based Streaming. Recent approaches stream scene geometry alongside dynamically baked lighting to thin clients [Hladky

et al. 2019b, 2021; Mueller et al. 2018], moving more work to the client. This added information allows the client to render geometrically correct novel views; to keep the shading workload on the server, the shading is baked into textures and streamed in a video stream to the client. As long as shading is computed and transmitted for disoccluded regions, correct novel views can be generated. The key problem with these approaches is that they are inherently dependent upon the scene geometry; these approaches only stream geometry and shading for a potentially visible set (PVS) of geometry, which is difficult to compute analytically [Hladky et al. 2019a]. As the industry trends towards smaller geometry [Epic Games 2020], packing shaded triangles also becomes more difficult [Hladky et al. 2021; Mueller et al. 2018]. Our approach remedies these issues by dynamically constructing a simplified representation of the scene that is independent of the scene geometry. This leads to predictable, easy-to-reconstruct geometry and our quad-based approach can efficiently be packed into a video stream.

Advanced Streaming Techniques. Finally, with even more client compute power, more complex streaming rendering approaches can be built. Surveys over the last two decades clearly show that sending more abstract data to clients allows for more involved splits in the rendering architecture [Chang and Ger 2002; Noimark and Cohen-Or 2003; Shi and Hsu 2015]; for example, spatiotemporal upsampling is possible given multiple frames with depth information [Pajak et al. 2011]. Complete frames can be rendered speculatively [Lee et al. 2015] and potentially augmented with residual images [Bao and Gourlay 2004; Yoon and Neumann 2000]. Compute-intensive rendering tasks such as global illumination can be streamed [Majercik et al. 2019] to improve client shading. Finally, and somewhat extremely, the client can be equipped with complete scene rendering capabilities to render any parts that are missing from the server's prediction [Cuervo et al. 2015]. Our focus is not on the *thick* client devices [Stengel et al. 2021] that these systems target, but on light-weight devices where we aim to reduce rendering costs on the client. We additionally note that non-intrusive approaches like ours are easier to adopt in practice, as they require minimal changes to existing streaming-based rendering applications.

2.1 Alternative Representations

Layered representations. In a layered representation [Shade et al. 1998], each pixel captures multiple opaque surfaces; this allows for warping and handling of disocclusion events. Such layered representations can be used to create immersive videos supporting free head rotation and within bounds head translation [Buehler et al. 2001; Collet et al. 2015; Dou et al. 2016; Pozo et al. 2019; Zitnick et al. 2004]. Taking these approaches further actually leads to dense 4D lightfields [Gortler et al. 1996; Levoy and Hanrahan 1996]. Recent approaches focused on using layered representations to describe lightfields include Soft3D [Penner and Zhang 2017] and multi-plane images [Flynn et al. 2019; Zhou et al. 2018]. While these approaches were not designed for streaming, recently Broxton et al. [2020] has shown that a spherical layered representation can be sparsified and packed into a video stream atlas to support streaming rendering of free viewpoint videos. While their approach requires significant bandwidth and more than a day to preprocess a single frame, it shows the potential of streaming alternative representations.

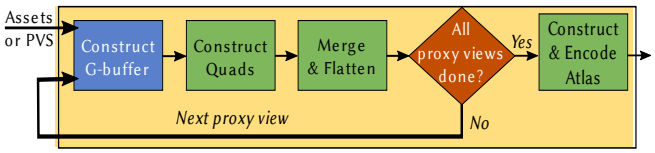
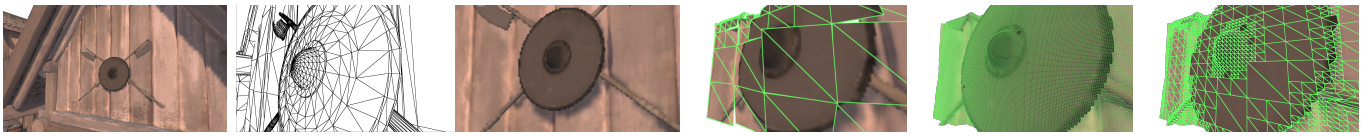


Fig. 2. QuadStream server pipeline: We iterate over a set of sampled views within a view cell; for each view, we render a G-Buffer, construct quad proxies following the rectangular grid of the view projection plane, merge suitable neighboring quad proxies, and potentially flatten them. After obtaining a set of quad proxies for all views in the view cell, we construct an atlas containing both the geometry and shading data, and stream it to the client device. The client uses this package to render novel views and can thus perform frame rate upsampling locally.

Neural Radiance Fields. The advances made for free-viewpoint video ware largely supported by learning-based methods to generate the layers representation; however, it is also possible to directly learn a representation for visual information in a scene [Sitzmann et al. 2019; Thies et al. 2019; Wizadwongsa et al. 2021]. Recent work on on Neural Radiance Fields (NeRFs) [Mildenhall et al. 2020] has sparked increased researched into learned representations. NeRFs can even be used to learn free-viewpoint video [Du et al. 2020; Li et al. 2020; Park et al. 2020; Pumarola et al. 2020; Xian et al. 2020]. While their compactness renders NeRFs especially interesting for streaming, training and inference times remain major hurdles for their application for our use case. Image generation for a current headset would require the device to offer 37 PetaFLOPS in compute power [Neff et al. 2021]. While training time is only just being tackled [Müller et al. 2022; Sara Fridovich-Keil and Alex Yu et al. 2022], there are approaches to reduce inference time, including learning partial integrals along the ray [Lindell et al. 2020], sparsifying the scene [Liu et al. 2020], or predicting scene ray intersections [Neff et al. 2021]. Another avenue to increase rendering speed is baking and caching various components of NeRFs [Garbin et al. 2021; Hedman et al. 2021; Reiser et al. 2021; Yu et al. 2021]. While those may reach real-time performance, they sacrifice compactness increasing memory requirements by 100x and more and thus trade one challenge for another from the perspective of streaming.

3 ALGORITHM OVERVIEW

Our goal is for a QuadStream client with no *a priori* knowledge of the scene to be able to fully reconstruct the scene appearance and geometry from a novel, updated view (within reasonable limits). The basic setup of our pipeline is outlined in Fig. 2. While the server necessarily does not know what this view will be, we assume that it can roughly predict the camera/head position by extrapolating from previous input and the estimated round-trip latency. As this estimate may not be accurate, the server must supply the client with not only the information required to render the predicted view, but information for rendering surrounding views, in particular disocclusions. We therefore consider a cuboid *view cell* of potential camera positions and rotations surrounding the predicted *center*



(a) Original scene (b) Scene geometry (c) Low-res client view (d) Proxy planes, initial (e) Tessellated (f) Merged and flattened

Fig. 3. **(a,b)** Original scene appearance and geometry captured by our QuadStream datastructure on the server for a client view **(c)** at low resolution (160×90) for demonstration purposes. **(d)** Proxy planes capture the geometry very coarsely. **(e)** Tessellating the proxy planes at each surfel and applying vertex offsets relative to the proxy planes enables us to reconstruct fine geometric details and close gaps. **(f)** We adaptively merge and flatten neighboring quads to avoid unnecessarily dense tessellation for regions with simple geometry. Our proxy data structures project to axis-aligned quads in **(c)**; note that the views shown in **(d,e,f)** are off-center relative to **(c)** for demonstration purposes only.

view. Our goal is for the client to be able to render the scene from any position and rotation within this view cell.

One of the main goals of QuadStream is to support easy integration into existing rendering systems. To this end, we operate on G-Buffers/visibility buffers that are typical for deferred shading approaches. To capture the content visible from within the view cell, we render multiple views from within the view cell when building the QuadStream—each view adding previously uncaptured data.

We start by rasterizing the center view into a G-Buffer and decompose it into a series of quadrilateral proxies, or *quad proxies*. Each quad proxy consists of two elements: a *proxy plane*, and a set of associated *quad surfels* (*cf.* Pfister et al. [2000]). The proxy plane roughly captures the geometry, while the quad surfels capture final shading, fine geometric details, and transparency when projected onto the proxy plane (Fig. 3). We construct quad proxies for a single view using a bottom-up approach: we start with small quads covering the G-Buffer that can be efficiently optimized locally, before repeatedly merging them to construct larger geometric elements and to facilitate data compression. After repeated merging, we have decoupled the rough geometric structure of the scene from fine surface variations, both in shading and local displacements.

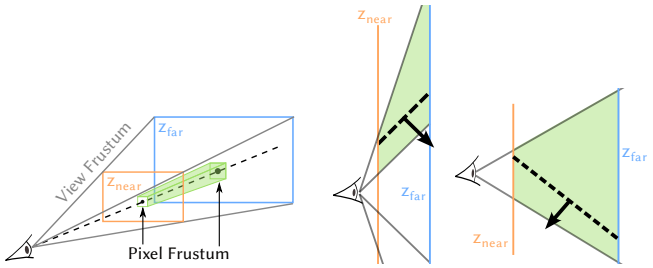
Relying on a single view/G-Buffer will ultimately result in disocclusion artifacts if the scene is reconstructed for any other camera location, as it captures the exact same content as a simple video stream. Our key insight is that we can easily augment the set of quad proxies by rendering new G-Buffers from additional viewpoints, i.e., by sampling the view cell. Naively rendering a new G-Buffer for every additional view would duplicate data already captured in previous views and would be prohibitively expensive. Thus, we use the already constructed quad proxies to fill the depth buffer for early and efficient depth rejection prior to rasterization and shading, avoiding quad proxy generation for previously seen content altogether.

We enable fast construction, streaming, and reconstruction of quad proxies by combining two key ideas. First, we observe that the supporting planes of quad proxies can be described implicitly by their rendered view location and footprint in the G-Buffer. Additional information, such as quad rotation, can be heavily quantized; fine-grained depth information can always be stored relative to coarser proxy planes. This approach leads to an exceptionally compact representation and allows for efficient encoding for streaming. Second, we observe that for finer surface details (stored as surfels), encoding them in the same domain in which they were sampled is the most effective, as resampling can be avoided. To enable efficient storage and reconstruction of this data, we need to be able to construct proxy planes with their rotation from the view location, and to ensure that surfels can be robustly projected onto them without artifacts. To this end, we show how to construct the *Quad Frustum Space (QFS)* for any rectangular region in a view. In this space, we can ensure that the coarse proxy planes are oriented so that their projection fully covers their assigned screen regions, and surfel projection (fine details) will not introduce major artifacts. This space also enables efficient quad reconstruction on the client.

Quad surfels from the center view lend themselves to straight forward packing in a video stream as they fully fill the image; however, quad proxies from additional views are typically sparse and scattered in the view. However, due to their design, quad proxies are easy to pack into a texture atlas: we limit their construction to power-of-two footprints and thus can pack them into blocks that are well-suited for standard video encoding. Once received, the client can render the scene from a novel view by simply rendering all streamed quad proxies from the novel view location. To reconstruct the quad proxies in world space, we rely on the QFS in combination with the inverse view projection matrix of the generating view, and apply displacement mapping and alpha testing using the surfel data.

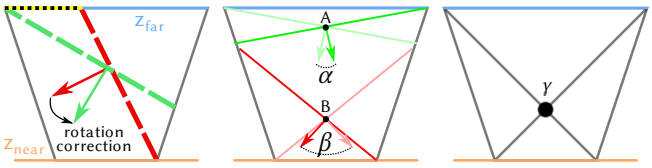
4 QUAD PROXY CONSTRUCTION

We process the scene G-Buffer to obtain a set of *quad proxies* which capture the scene appearance and geometry and enable rendering of novel viewpoints within a given view cell. Each quad proxy consists of two constructs; a oriented 3D *proxy plane* which coarsely approximates the underlying geometry of the scene, and a set of *surfels* that capture the scene shading. These surfels optionally store a depth offset relative to the proxy plane which allows for reconstruction of fine geometric details. The proxy plane is oriented according to the normal of the underlying surface it captures. Defining the proxy plane in a skewed transformation of camera view space (*Quad Frustum Space*; Sec. 4.1) enables us to identify an analytical range of possible plane orientations and to avoid back-facing normals. Furthermore, the orientation of the proxy plane needs to be corrected to avoid penetration at near or far planes of the view frustum (Sec. 4.2). The efficiency of our representation is improved by merging (and by optionally flattening) similar neighboring quad proxies, while quad proxies that capture large depth discontinuities (*e.g.* at object silhouettes) are split (Sec. 4.3). While this approach correctly captures the scene appearance and geometry for the original G-Buffer camera view, translating the camera reveals disocclusion artifacts.



(a) Projected pixel center and corners (b) Skewing (c) QFS Transform

Fig. 4. Computations in view space may result in planes facing away from the camera (b), especially near the edge of the view frustum. Computing the quad plane in QFS space fixes this issue (c).



(a) Required correction (b) Rotation limits (c) Deciding depth γ

Fig. 5. (a) Slanted planes may penetrate near/far bounds. To identify rotation limits at given points (b), we compute deciding depth γ (c).

To capture disoccluded content, additional G-Buffers rendered from proxy views are processed to augment the existing quad proxy set (Sec. 4.4). The resulting *QuadStream* datastructure supports transparent objects (Sec. 4.5), lends itself to memory efficient encoding (Sec. 4.6) suitable for streaming and enables scene appearance and geometry reconstruction from novel viewpoints (Sec. 4.7).

The most trivial quad proxy is constructed from a single G-Buffer sample. Assume a G-Buffer is obtained by rendering the scene with a projection matrix $\mathbf{P}$ and view matrix $\mathbf{V}$. Using the depth d of the G-Buffer location and the pixel coordinates, we use the inverse of the projection matrix to recover the clip space coordinates $\mathbf{C} = \begin{bmatrix} c_x & c_y & c_z & c_w \end{bmatrix}$ of the surfel, which we un-project into view space by $\mathbf{S}_v = \mathbf{C} \times \mathbf{V}^{-1}$. We therefore use the 3D point $\mathbf{S}_v$ and the G-Buffer normal $\mathbf{o}$ to construct the proxy plane in view space.

4.1 Quad Frustum Space

To bound the previously defined proxy plane to the pixel's extents, we define the *quad frustum* as follows. We first recall that under perspective projection, the *view frustum* is defined as the pyramid of planes passing through both the view origin and the near and far clip planes. For a given pixel in the G-Buffer, we can extend this construction to speak about the view frustum of a single pixel—the frustum generated for a small, off-center projection (Fig. 4(a)). Under projection, the center point of the quad frustum projects exactly to the center of the pixel, and the corner points project onto the corners of the pixel. By analogy, we can extend this to larger quads beyond individual pixels.

Due to perspective projection, the quad frustum in view space becomes more skewed as it gets closer to the edge of the camera view frustum (Fig. 4(b)). For any practical perspective parameters, this may cause some of the surface normals to be back-facing in view space (see the black surface and its corresponding normal). In order to restrict the space of possible orientations to only front-facing directions, as well as to analytically identify a range of possible orientations, we invert the skew of the view space by operating in *Quad Frustum Space (QFS)*—a transformation of the camera view space which centers the quad frustum around the $(0, 0, -1)$ axis. The transformation matrix $\mathbf{B}$ from camera view space to QFS, as illustrated in Fig. 4(c) is a shear matrix:

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \frac{c_x}{n_z} & \frac{c_y}{n_z} & 1 \end{bmatrix} \tag{1}$$

where n_z is the near plane distance from the camera (as used when computing the projection matrix $\mathbf{P}$). For the final transformation of a proxy plane from viewspace to QFS, we query the viewspace plane to obtain three points on the plane p_0, p_1, p_2 (one of which can be S_v); transform each point to QFS by multiplying $p'_i = \mathbf{B} \times p_i$; and finally construct the plane from these three points.

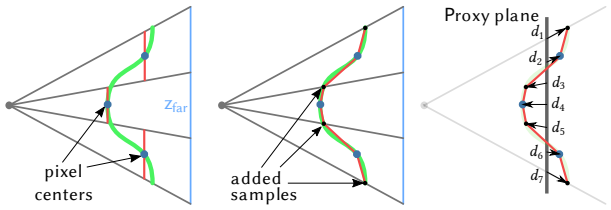
4.2 Orientation Correction

Using the surfel normal from the G-Buffer to orient proxy planes may result in an extreme orientation, especially when the normal vector is near-perpendicular to the camera forward vector. Such slanted planes may penetrate the near and far projection planes within QFS, and thus may fail to cover the whole projected area. As we aim for a closed representation for reprojection, this is undesirable as it would generate holes during rasterization (Fig. 5(a)). Furthermore, it would also complicate the process of merging quad proxies and projecting surfels onto the proxy planes. The maximum rotation ranges that are allowed for a proxy plane to result in a fully visible projection depend on the position of the plane in QFS (Fig. 5(b)). In the following, we describe how we obtain these ranges.

We start by intersecting a ray from the QFS origin in the direction $(0, 0, -1)$ with the proxy plane, and obtain a point $\mathbf{m}$ that projects exactly to the center of the quad in projection $\mathbf{P}$. As shown in Fig. 5(b), the rotation limits may be due to the near plane ($\mathbf{m} = \mathbf{B}$) or the far plane ($\mathbf{m} = \mathbf{A}$) in QFS, depending on the depth of $\mathbf{m}$. To distinguish between cases, we can compute the deciding depth γ by intersecting the diagonals (Fig. 5(c)). In each case, we can explicitly construct the two planes in QFS from point $\mathbf{m}$ that limit the rotation. Since the QFS is centered around $(0, 0, -1)$, our rotation range can only support front-facing orientations and can therefore be transformed into a 2D $\theta - \phi$ subspace of the spherical coordinate system. For more details please refer to the supplemental material.

4.3 Quad Merging, Flattening & Splitting

Using a simple quadrilateral reconstruction for every texel will lead to holes appearing between neighboring quads with the smallest change of view. To close the connection between neighboring quads, we split each quad into four and consider the depth of neighboring samples. By linearly interpolating between the quad center sample



(a) 3-surfel approximation (b) Multisample + vertex offset (7 surfels) (c) Final reconstructed Proxy Plane

Fig. 6. Proxy plane construction: a) Approximating a surface just from the pixel; and (b) from multiple samples (surfels) and neighboring pixels. (c) Example proxy plane constructed for the 7 surfels obtained by multisampling in (b). Their exact position is obtained by offsetting their projection by the distances $d_1 - d_7$ towards the projection center.

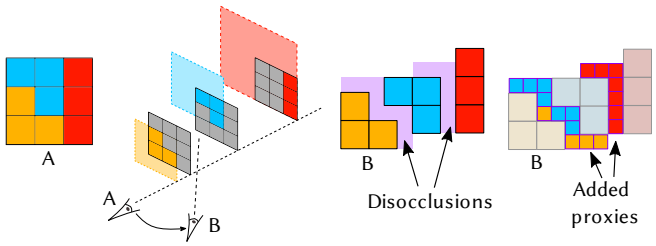


Fig. 7. Capturing three disjoint surfaces (yellow, blue, red) as seen from view **A** by splitting the corresponding quad proxy into three proxy planes. Note that their joined projection (left) covers the whole area of the proxy. Moving from the initial view **A** to an offset camera position **B**, holes caused by disocclusions appear. By rendering a G-Buffer with early rejection from viewpoint **B**, we augment the set of quad proxies to cover the disocclusions.

and its neighbors, we adjust the quad corners so that both quad proxies reach halfway towards the next, and completely close the holes (see Fig. 6). Splitting a single quad (4 vertices) into four quads (9 vertices) effectively samples the scene at twice the resolution of the G-Buffer; by inspecting the neighboring pixels of one quad, we can offset the corner vertices to close gaps with neighboring quads.

However, neighboring quads should not necessarily meet in all locations. If a connected region of the G-Buffer depicts a closed surface, we ideally want to reconstruct that surface with a sequence of closed quads. If there is a depth discontinuity, we wish to capture the discontinuity and fill in the potential disocclusion from other views. To describe where surfels should be joined, we define a threshold depth offset, $\delta_{\max}$, measured in the standard inverse depth space of the projection. Since we operate on a fixed threshold, we may still create false positive and false negative seams. While false positive seams will most often be filled by quad proxies from additional views, false negative seams may lead to missed disocclusions. We note that as we are operating in inverse depth space, when distant disjointed neighboring surfaces are wrongly merged, their parallax from the novel views is relatively small. We found in our experiments that merging such surfaces into one surface due to false negative seam detection hardly affects the reconstruction quality, as we are constructing quad proxies at approximately double the resolution of the G-Buffer. Storing and rendering a unique quad proxy for every texel is infeasible. To improve performance and reduce memory requirements, we traverse the quad proxies in a bottom-up fashion in parallel with the goal of merging similar neighbors, operating on groups of 4 neighboring quad proxies. We first transform the proxy plane of each proxy from QFS back to view space by inverting its transformation. If the four planes in view space are similar, as measured by the difference between their plane equations, we construct a new quad proxy spanning the joined region of the four planes. For the proxy plane of the new quad proxy, we average the normals of the merged planes; we then transform it by $\mathbf{B}_{new}$ of the new larger quad proxy into QFS, and readjust the depth offsets for each underlying surfel. Furthermore, if all depth surfels are below a fixed threshold in view space (δ_{flatten}), we mark the quad proxy as flattened, discarding any depth offsets from all surfels. Such quad proxy will be rendered as a flat quad. Note that even a flattened quad proxy retains the original shading resolution.

Quad Splitting. During quad proxy construction, if we detect that a quad proxy contains a seam due to a depth discontinuity, the underlying surfaces must be described by more than a single proxy plane, and the underlying surfels have to be assigned to the corresponding best fitting plane. To this end, we construct a pool of *candidate proxy planes* from the surfel pool of the quad proxy. Each surfel then votes on each plane's "fitness" to approximate it by computing its projected distance to the quad plane using the inverse depth $\delta_{\max} - |d|$. After each round of voting is complete, we iteratively accept the planes with the most votes until all surfels have been assigned to a proxy plane. We describe our implementation of GPU quad construction and splitting in the supplemental material.

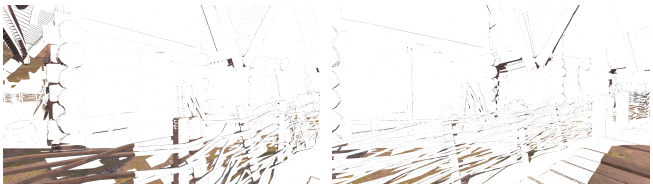
The joined projection of all accepted proxy planes fully fills the projected area of the quad proxy that spawned them (Fig. 7). Surfels of the proxy plane that do not contribute to the approximated surface (gray in Fig. 7) are masked out using a binary alpha mask. This allows us to capture arbitrary shapes with overlapping quad proxies, while relying on the efficiency of working on quads only.

4.4 Additional Proxy Views

When generating novel views (not coinciding with the center view) from our quad proxies, any two joined surfaces are geometrically fully approximated. However, when one quad proxy is split into multiple proxy planes, viewing them under an offset introduces disocclusion artifacts due to the parallax effect. We address this by sampling additional views from within the view cell, and augment the already constructed quad proxies by constructing only the yet-uncaptured geometry (disocclusions). To avoid capturing surfels again, we rasterize the already obtained quad proxies for each proxy view to pre-fill a depth mask which is then used for quad-based occlusion culling via warping in the G-Buffer construction (Fig. 7).

Even for scenes with high disocclusion complexity (e.g. groups of thin disjoint geometric structures), offsetting views within a view cell produces sparsely occupied G-Buffers (Fig. 8.) Using previous quad proxies as warped occluders for culling not only reduces the number of generated quad proxies, but also reduces the additional shading load on the server. To generate consistent shading, we shade

(a) Center view



(b) Left-shifted proxy view (c) Right-shifted proxy view

Fig. 8. (a) The thin fence structures in the Viking Village scene lead to many disocclusions. (b, c) The additional view samples from within the view cell supply the disoccluded areas. Note that the added data is sparse, yet vital to avoid disocclusion artifacts during novel view generation.

view-dependent effects exclusively from the center view. While view-dependent shading (e.g., specular lighting) will be slightly off for non-center view locations, it will be consistent among all quad proxies. We note that the human visual system is not sensitive to small mismatches in view-dependent shading [Mueller et al. 2021].

After constructing quad proxies for all proxy views within a view cell, we optionally construct one more G-Buffer—a low-resolution wide FOV projection of the whole view cell range, providing low resolution information for large and unexpected camera rotations.

4.5 Transparency

While our previous description only focuses on opaque surfaces, our approach extends to scenes with transparency provided that the G-Buffer construction is deep and contains multiple texels per pixel. By capturing a surfel's alpha, semi-transparent surfels captured in each texel in the G-Buffer can contribute to the quad proxy construction in the same manner as full-opaque surfels do.

4.6 Storage and Encoding

Our data structures are designed with streaming in mind; our key design criteria is that they must both have a low memory footprint and allow for efficient encoding. The streamed data packet for a frame consists of three main parts: a set of view descriptors to store the matrices for each used view; the surfels; and the quad proxies.

We pack the surfel data in an atlas texture on the server, divided into three data streams for color, depth, and alpha (if used). The color is transmitted with an h.264 YUV420 encoding. The depth offset of each surfel is stored relative to the proxy plane and thus compresses well. For depth encoding, we use the technique of Koniaris et al. [2018]; as they suggest working on differently sized quadratic blocks, we can apply them directly to our quad proxies. In our experiments, we either reached or surpassed their compression rate of 15×, which is likely due to the fact that our quad proxies are designed to capture parts of closed surfaces and are not limited to cells of a quad tree.

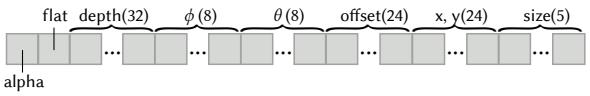


Fig. 9. Layout of packed quad proxy meta data (with bits per component).

The quad proxies constructed from opaque and alpha-clipped materials have the alpha expressed as a 1-bit mask, stored using run-length encoding and following a Hilbert curve in every quad. Blend materials store a 8-bit alpha. To exploit temporal coherence in the alpha stream, we compare direct run-length encoding to delta-encoding (where 1 indicates a bit flip) of the corresponding block in the previous frame; choosing the smaller of the two allows us to raise compression ratios from 15× to about 25×. If there is no matching previous frame block (*i.e.* because quad merging has changed), we use the direct run-length encoded data.

We pack quad proxy metadata as follows. Since we perform hierarchical merging, only a limited number of quad sizes are possible, all of which are a power of 2 (up to 1024, or 5 bits). We pack the spatial 2D coordinate of a quad proxy in the G-Buffer into 24 bits (assuming a pixel space of 4096×4096, *i.e.* 2×12 bits per axis). The surfel data requires offsets into the aforementioned data streams; for each such stream, 24 bits are sufficient. Each proxy plane is described by its orientation, which is stored as a spherical angle in θ, ϕ space; we quantize this using the diamond shape approximation with 8-bit precision per axis. We store the proxy plane depth with 32 bit precision, as the plane depth offset is the most crucial part for correct geometry reproduction. We reserve 1 bit for recognizing flattened quads (without any geometry offset for any surfels), and 1 bit for knowing whether any of their surfels have alpha < 1 (Fig. 9).

4.7 View Reconstruction

To use the streamed data package on the client for novel view extrapolation, we render the received quad proxies as standard forward rendered primitives. For each primitive we load the matrices describing the proxy view that spawned it, and reconstruct the quad planes with per-surfel offsets unprojected in world space. The geometric detail of non-flattened quads can be reconstructed using tessellation or mesh shaders, where offsets for each vertex can be computed. Alternatively, the proxy plane can be treated as planar billboard and ray marching can be used to compute displacement mapping in a fragment shader. The final shading color is then loaded from the atlas via texture lookup. We discuss details of our reconstruction including handling of transparency in the supplemental material.

5 EVALUATION

We evaluate QuadStream via a thorough comparison against prior art on consumer-grade PC hardware, and further assess its performance on a mobile thin client device (an Oculus Quest 2 with SnapDragon XR chip). We compare our QuadStream approach against two recent methods for streaming rendering: Shading Atlas Streaming (SAS) [Mueller et al. 2018] and SnakeBinning (SB) [Hladky et al. 2021]; we also compare against MeshWarping (MW) [Mark et al. 1997], since this approach also approximates the geometry and appearance from a G-Buffer, is easily adapted for streaming rendering

(a) Robot Lab, 472kΔ (b) Viking Village, 4.6MΔ (c) San Miguel, 9.96MΔ

Fig. 10. Still frames and triangle count (Δ) for the tested scenes, which we render with deferred shading, cascaded PCF shadow maps, environment-map lighting and tone mapping.

and can be tuned for the highest quality rendering by using a very fine mesh. For comparison purposes, we use the same scenes as SAS and SnakeBinning: Robot Lab (Fig. 10a) and Viking Village (Fig. 10b) and extend this set of scenes with the more challenging San Miguel atrium (Fig. 10c). Robot Lab is a small indoor scene with relatively coarse geometric structures, simple geometry, and view-independent shading. Challenging areas include thin structures, such as the railing and stairs. Viking Village is a larger outdoor scene with simple shading, but frequent disocclusions and varying geometric detail: the fences and pole structures are thin and lead to multiple visibility layers, the roofs consist of many slanted triangles, and skulls are modeled with an excessive amount of detail. San Miguel is a large and complex model of a courtyard with semi-transparent glassware, thin structures and ample vegetation, where each leaf is individually modeled. The water, glass, and metal parts include heavily view-dependent shading.

The streaming rendering pipeline was emulated on a device with a Intel Xeon-E5 2643CPU with 32 GB RAM and an NVIDIA RTX 3090; methods were evaluated on a pre-recorded camera path. To simulate real use cases, the client framerate is decoupled from the server framerate in all our measurements. In the walkthroughs, all methods render frames using the last received server data. Server prediction uses the camera's linear and angular velocity and acceleration; the client receives the server data structures at a delay, due both to the various methods' computation time + a simulated network latency of 10.00 ms ± 4.00 ms. This forces the client to always perform a "novel view extrapolation" from the server data structure, as the client viewpoint almost never coincides with the center of the server frame (except when the movement prediction is very accurate, *e.g.* when the client is still). We encode the shading atlas RGB channels using a *nvenc* h.264 stream. To evaluate our complete system performance, we gather 570 frames along a simulated camera path. We use three different view cell sizes, with 30, 60, 90cm translation and a rotation range of 100, 120 and 140°. For our approach, we use the center view on the camera path and one proxy view from each corner of the view cell. To simulate non-ideal prediction, we randomly sample 25 views in the cell as client view and average resulting quality.

5.1 Algorithm Parameters and Setup

We use the following parameters for our approach. We set $\delta_{\max} = 10^{-4}$ in inverse depth space; 10^{-2} as plane similarity threshold, the early Z rejection offset to 10^{-4} and the flattening threshold to $\delta_{\text{flatten}} = 10^{-1}$. All renderings are performed at 1920×1080 resolution. Our view sampling starts with the center view, followed by the front views of the view cell in clockwise order, followed by the back views, again in clockwise order. Finally, we gather an additional proxy at the center view with increased field of view (170°), but at a lower resolution of 1280×720. While the final view does not increase quality in our testing, as only views in the view cell are sampled, we include it to capture real system load.

While our approach captures the data for disocclusions through quad proxies from the proxy views, SAS and SB require an explicit potentially visible set (PVS) computation, which provides a list of scene triangles to be shaded and packed into their atlas streams. To obtain the PVS for these methods, we rasterize the same views as in our approach at 1920×1080, and store the triangle indices in the frame buffer. We then run a gathering pass over the entire screen, mark the triangles and compute their largest footprint among all views to determine the target size in the atlas.

All methods operate on a 4096×4096 texture atlas. SAS is able to dynamically adjust the shading rates to fit within these boundaries. Their atlas texel occupancy threshold was set to 75%, as higher values caused an atlas overflow on rapid view offsets. Atlas reset occurred every 5 frames and the shading factor grow/shrink was set to 110/90%, respectively. SB, which was originally evaluated with 8192×8192 atlas by the authors, needed parameter adjustments to avoid overflow: α was set to 15–90°, β 5–60° and height 1–256, using 10% extra allocation, while setting the height scaling factor to 0.55. MW captures the scene from a single 3840×2160 view at the center of view cell, with FOV set to encompass supported camera offset ranges. Thus, the center view stays true to client resolution.

5.2 Performance

Timing results are shown in Tab. 1. For QuadStream, time spent on the server is approximately equally split between G-Buffer gathering for the nine views, and overall construction time. On average, each G-Buffer pass takes about 3-4 ms; the first pass is the most expensive (about 10 ms in Robot Lab and Viking Village, about 35 ms in San Miguel), as the majority of shading is evaluated from the center view. Rendering the wide field-of-view proxy takes about half of the first G-Buffer pass time. The remaining time is split evenly between the corner views, which only add small disocclusions and thus are more efficient to generate and shade. Quad proxy construction time exhibits similar behaviour: the first view consumes the most time, as it generates the majority of the quad proxies—roughly 10 ms; the wide field of view requires about 5 ms; and every additional corner view takes 1–2 ms. Merging and flattening of the quad proxies (M & F) is very efficient, with little difference between run times for Robot Lab and Viking Village. San Miguel exhibits longer G-Buffer generation times due to the sheer amount of geometry, and also requires more time for other stages due to large amounts of disocclusion and masked rendering of the foliage. The increase in samples is modest considering the complexity of the foliage and added transparent objects.

The main factor impacting our speed is the resolution of the G-Buffers. Our server achieves speeds of 70–90 ms per frame for the simpler scenes, and approximately double the cost for San Miguel. We gather approximately 340k quad proxies per view cell for Robot Lab, 460k for the Viking Village and 770k for San Miguel, of which typically 90–95% are flattened (*i.e.* no depth map is actually necessary

Table 1. The measurements of our method, averaged over 570 frames in a walkthrough. The main cost of our approach is G-Buffer generation and quad proxy construction. Merging and Flattening (M & F) hardly contributes to the overall runtime. SB and SAS require similar runtimes when PVS generation is considered. As our method performs displacement mapping on the client, it requires more client processing power. We note that our approach is independent of the scene geometry and thus sample count hardly changes for RL and VV, even when the view cell count (VC1-3) is increased. SB and SAS need to capture significantly more samples. For San Miguel (SM), both SAS and SnakeBinning failed to render the scene correctly due to its geometric complexity. We omit comparison results as these methods fail to render large portions of the frame.

Scene &	**Our method**					**Sampled**	**SnakeBinning**			**Shading Atlas Streaming**			**Mesh Warping**		
Viewcell	G-Buffers	Construct	M & F	Samples	Client	**PVS**	Gather	Samples	Client	Gather	Samples	Client	Gather	Samples	Client
RL-VC1	29.49 ms	34.98 ms	1.73 ms	6.60 M	0.73 ms	41.31 ms	29.70 ms	5.32 M	0.10 ms	46.82 ms	10.51 M	0.49 ms	15.97 ms	8.29 M	1.50 ms
RL-VC2	31.83 ms	35.10 ms	2.11 ms	7.43 M	0.80 ms	44.38 ms	37.38 ms	5.91 M	0.10 ms	49.65 ms	10.64 M	0.53 ms	15.34 ms	8.29 M	1.57 ms
RL-VC3	33.01 ms	35.80 ms	1.97 ms	8.87 M	0.87 ms	50.01 ms	41.11 ms	6.58 M	0.11 ms	52.21 ms	10.97 M	0.58 ms	15.06 ms	8.29 M	1.52 ms
VV-VC1	38.10 ms	39.58 ms	2.19 ms	6.15 M	0.80 ms	63.02 ms	17.83 ms	7.45 M	0.12 ms	35.01 ms	10.31 M	0.38 ms	10.00 ms	8.29 M	1.59 ms
VV-VC2	40.21 ms	41.79 ms	2.43 ms	6.80 M	0.84 ms	65.37 ms	20.05 ms	8.02 M	0.14 ms	38.73 ms	10.50 M	0.43 ms	11.12 ms	8.29 M	1.58 ms
VV-VC3	43.35 ms	45.00 ms	2.88 ms	7.55 M	0.94 ms	70.33 ms	24.71 ms	8.94 M	0.15 ms	43.27 ms	10.97 M	0.53 ms	11.14 ms	8.29 M	1.59 ms
SM-VC1	77.12 ms	88.03 ms	5.29 ms	9.34 M	1.11 ms	143.10 ms	-	-	-	-	-	-	51.10 ms	8.29 M	1.59 ms
SM-VC2	83.45 ms	93.73 ms	5.97 ms	11.94 M	1.54 ms	149.32 ms	-	-	-	-	-	-	51.92 ms	8.29 M	1.59 ms
SM-VC3	91.33 ms	95.41 ms	6.73 ms	14.29 M	1.75 ms	154.47 ms	-	-	-	-	-	-	50.91 ms	8.29 M	1.59 ms

for fine detail reconstruction). Interestingly, the ratio of flattened proxies is about the same for all scenes; we note that in San Miguel the quads tend to be smaller and we also more aggressively flatten them to keep geometric complexity bounded.

Performance Comparison vs. Prior Art. Compared to SB and SAS, server runtimes are roughly similar when the costs of sampling a PVS are included. PVS sampling times are actually slightly higher than our G-Buffer gathering times in the Robot Lab and Viking Village, as they require a full screen harvesting pass and compaction of the data for forwarding to the next stage. In San Miguel, the difference is even bigger, as our GBuffer uses the quad proxies for early-z masking, while sampled PVS requires full geometry processing per each view. The gathering stages for SB and SAS depend on the amount of geometry and clearly increase with view cell size, showing their clear dependence on the geometric resolution of the scene. The downside of geometry-based approaches such as SB and SAS becomes clearly apparent for San Miguel: due to the sheer amount of fine granular geometry and the large amount of disocclusion events in the scene, both methods want to capture more geometry than fits into the atlas. We were unable to tune SB or SAS to shade an entire frame; the additional samples required around individual triangles and borders led to out of memory errors, rendering a comparison pointless as large portions of the frame are not shaded. Our approach, on the other hand, is independent from geometric resolution; our performance only depends on the size and distribution of disocclusions. MW performance is stable, as it simply renders a single RGBD map of the scene. On the client side, MW performs a warp of the captured RGBD image at full resolution. Spawning and warping 3840×2160 quads is a significantly higher workload than other client stages, and requires the most client runtime. We also experimented with reducing the MW vertex grid from 3840×2160 down to 160×68 (*i.e.* 32×32 pixel quads). This lowered the client times from 1.5 ms to 0.4 ms, but caused severe visual artifacts (see supplemental video). Thus, we performed the MW walkthroughs with a 3840×2160 vertex grid for highest quality.

Sample Rates vs. Prior Art. The most important criterion for evaluation is the number of samples generated by all methods. Our sample counts are mostly independent of view cell size and, especially for the Viking Village, are much smaller than both SB and SAS. Even for San Miguel, our increase in samples stays within reasonable bounds, although the the foliage results in a severely high number of disocclusions. We also note that we add about 30% of samples from our increased field of view proxy, which is not part of the other method's potentially visible set: when comparing against SB and SAS, this fact underlines our efficient use of samples. We attribute this fact to the shading cost of SB and SAS being based on scene geometry; even if only small parts of a triangle become visible, the entire triangle (SB) or entire patch of triangles (SAS) must be packed into the atlas and shaded. Furthermore, the borders of these primitives require additional treatment.

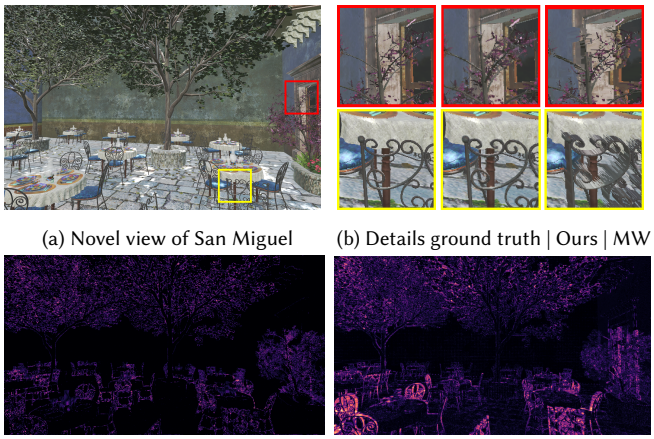
As we do not have a mobile implementation for the competing methods, we also run the client implementation on our desktop machine. Our client performance is slower than simple texture mapping of large geometry, as performed by SB and SAS; this is due to performing displacement mapping on a very fine geometric level for non-flattened quad proxies, where we use mesh shaders to generate two triangles per pixel and displace them geometrically. We also run an order-independent transparency algorithm to smoothly blend masked quad proxy surfels (see supplemental material for details.) Nevertheless, we can render novel views at speeds of up to 1000 fps; we note that this is a prototype and that a commercial QuadStream implementation could likely be further optimized.

5.3 Thin Client Rendering

We implemented and recorded performance measurements of QuadStream on a commercially available thin client, on the Oculus Quest 2 VR headset. The Quest 2 is capable of achieving its 72 FPS target framerate at the suggested eye resolution (1440×1584) while rendering as many as 700k–1M triangles each frame with our method. Please refer to the supplemental material and video for details.

Table 2. Image quality measurements for walkthroughs at 1920×1080 resolution. For both DSSIM and FLIP lower is better. Please compare results with Tab. 1: achieved quality in relation to needed samples favors our approach.

Scene	DSSIM				FLIP			
and PVS	Ours	SB	SAS	MW	Ours	SB	SAS	MW
RL-VC1	0.01	0.04	0.03	0.04	0.024	0.067	0.049	0.052
RL-VC2	0.01	0.04	0.03	0.07	0.025	0.067	0.054	0.075
RL-VC3	0.01	0.04	0.04	0.09	0.027	0.067	0.063	0.103
VV-VC1	0.02	0.03	0.02	0.04	0.027	0.048	0.037	0.052
VV-VC2	0.02	0.03	0.02	0.07	0.029	0.048	0.041	0.079
VV-VC3	0.02	0.03	0.03	0.09	0.03	0.049	0.047	0.097
SM-VC1	0.06	-	-	0.11	0.072	-	-	0.098
SM-VC2	0.07	-	-	0.14	0.078	-	-	0.159
SM-VC3	0.08	-	-	0.2	0.082	-	-	0.231



(a) Novel view of San Miguel (b) Details ground truth | Ours | MW

(c) FLIP Ours (d) FLIP MeshWarping

Fig. 11. Example frames from San Miguel walkthrough. The MeshWarping (MW) disocclusion artifacts are significant (*e.g.* on the doorway frame and chair seats) despite running at full geometric resolution (3840 × 2160).

Exploring the Robot Lab (about 360k quads) on the Quest 2 with our approach, the framerate never dropped below 72 FPS. For views completely filled by scene geometry, we achieved an average frame rate of 78–80 FPS. The stream of San Miguel (680k quads) achieved frame rates fluctuating between 42 and 51 FPS due to the high geometric load. However, in this case the Quest 2 can still produce seamless visuals by doubling the framebuffer update interval, *i.e.*, using asynchronous timewarping for every second frame.

5.4 Quality

Novel view generation quality results are summarized in Tab. 2, and an example view is shown in Fig. 11 and 12 . Please zoom in to the figures to see details; additional high-resolution images are provided in the supplemental material. As can be seen, our approach produces the best quality for all scenes by a significant margin. Considering the actual number of shading samples gathered and transmitted to the client, the advantage of our method is further emphasized. While ours uses less than 9 MPix in the Robot Lab and less than 8 MPix in the Viking Village, SAS uses 10 MPix in both Robot Lab and Viking Village. SAS adaptively adjusts its shading rate until it either hits the atlas occupancy threshold or achieves full shading rate. For all view cell sizes it reaches approximately the same atlas occupancy, hinting that in order to achieve full shading rate a larger shading atlas would be needed. As SAS operates on patches of up to three triangles, it tends to spread its shading samples inefficiently, especially for slanted geometric structures. Compared to our method, SB gathers fewer samples in RobotLab and more samples in the Viking Village. Tab. 2 shows that SB achieves marginally worse quality than SAS, while gathering significantly fewer shading samples. We selected the binning configuration that achieved the best shading quality without atlas overflow; yet on average it uses only up to 60% of the available atlas space. This is partly due to the fixed block layout of the bin "snakes", and partly due lock-in of all parameters for a single setup. Our results indicate that a more dynamic binning approach could make SB more effective, however it is unclear how this would interfere with its temporal coherence. For San Miguel, QuadStream can still operate with a 16 MPix atlas, whereas both SB and SAS run out of space. While SAS tries to scale down all patches, it still runs out of memory due to its minimum patch size. SB does not have a mechanism to dynamically adjust its memory requirements. Running QuadStream on San Miguel using the initial quad construction parameters resulted in a very fine grid even for the smallest viewcell (VC1), counting 1.3M quad proxies, 14 M shading samples, with the client rendering at 2 ms and 0.06 DSSIM. After tuning the parameters to encourage stronger flattening and merging (increasing δ_{flatten} to 1 and plane similarity threshold to 10^{-1}), we achieved 770 k quad proxies with 9.34 M shading samples with client rendering at 1.11ms and 0.07 DSSIM. MW does not handle disocclusions in any form and therefore achieves the worst image quality. Given the same shading atlas space, all compared methods achieve worse quality than ours.

Fig. 12 highlights the visual quality of the respective approaches. SAS distorts samples due to the way they map scene geometry into rectangular blocks in their atlas. Slanted triangles, long thin geometry, and even very large triangles (due to limitations in atlas block size) lead to very different sample distributions in the atlas than what is required during rendering; thus artifacts are distributed over the entire view. SB follows the triangle size of the reference view to capture shading; however, as triangles are binned, slanted triangles (like on the railing) still lead to distortions. We further note that borders are required around each individual triangle for bilinear interpolation, leading to significant waste of samples for scenes with high geometric complexity (like the Viking Village and San Miguel). In contrast, our quad proxies are constructed from the individual G-Buffers, and thus are completely independent of the scene geometry. Nevertheless, novel views for thin geometric features such as the railing and the wires in the RobotLab and the chairs and tables of San Miguel also lead to small artifacts. This is not surprising as we resample the scene from the G-Buffers; thus our reconstruction is limited by the sampling frequency.

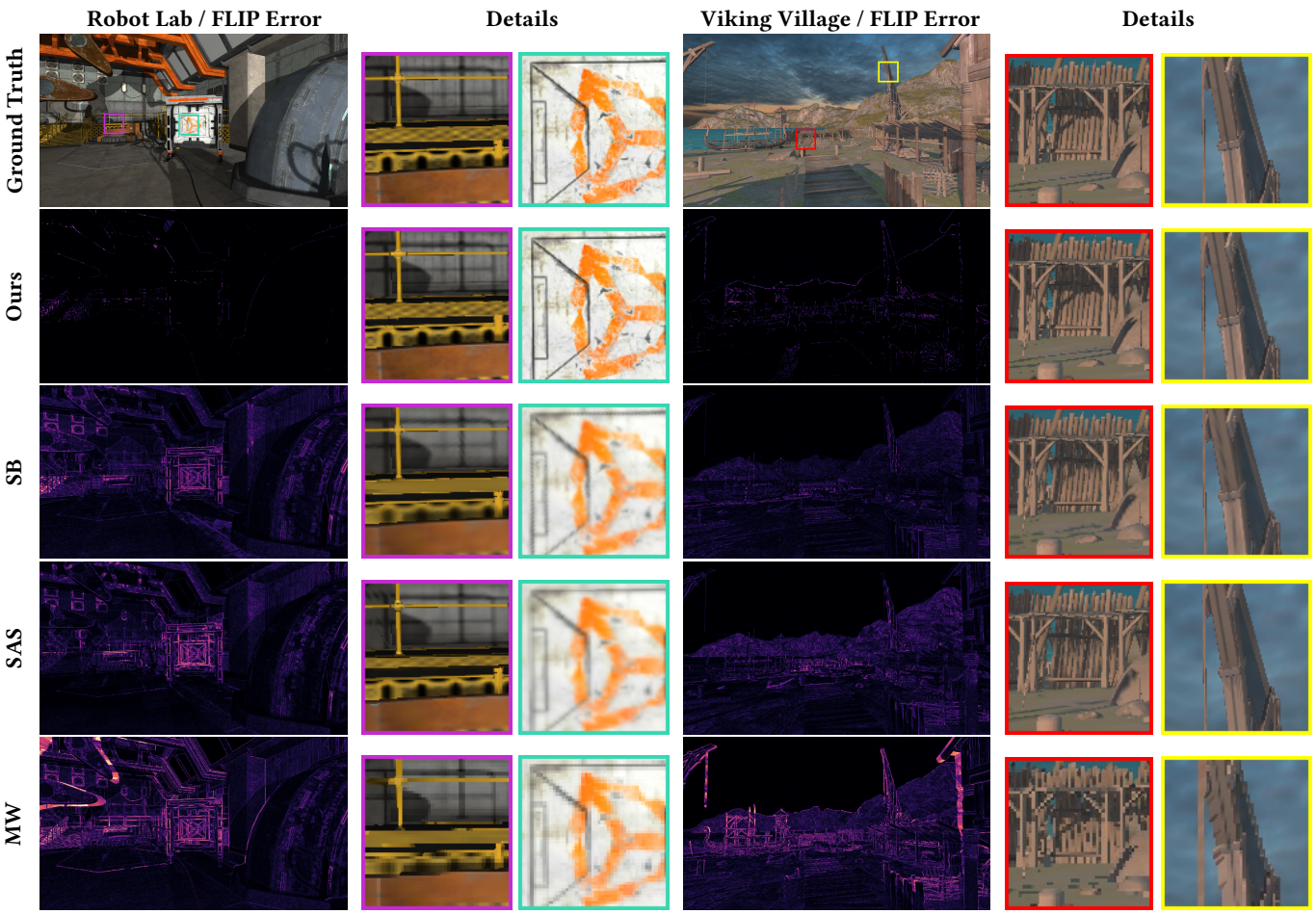


Fig. 12. A comparison of the novel view renderings for all tested methods: QuadStreaming (ours), Snake Binning (SB), Shading Atals Streaming (SAS) and MeshWarping (MW), with detail closeups. FLIP [Andersson et al. 2020] perceptual error was used, measured against ground truth for the novel view. Note that we both require significantly fewer samples than previous methods, while also improving on FLIP error vs ground truth for these novel views.

Transparency. We note that, unlike prior methods, QuadStream correctly handles the transparent glasses, the masked foliage, and the transparent water surface in San Miguel. This is possible as we support masked and transparent quad proxies by default. The primary complication is that the server must generate a deep G-Buffer in order to take the layers of transparent objects into account. Our approach encounters the most issues when multiple thin transparent layers coincide and wrap around, as with the transparent glasses (Fig. 13); in this case, it is nontrivial to reconstruct closed proxies that look correct under novel view points. Nevertheless, we are the only approach that can stream transparent geometry and handle alpha-masked foliage correctly. Our prototype implementation resolves the transparency in order-independent manner and bakes the resulting transparent color on top of the frontmost semi-transparent surface. Constructing the quad proxies through fragment list traversal is an interesting alternative to the presented construction based on G-Buffers, which we leave for future work.

5.5 Memory footprint & Integration

The PVS size for a viewcell mainly depends on the scene geometric complexity, while the QuadStream size mainly depends on the resolution of the G-Buffers. As can be seen in Fig. 14, scenes that contain only a few objects modeled with near pixel-sized geometry (*e.g.* Robot Lab and Viking Village) the Quadstream geometry representation achieved higher memory footprint than the sampled PVS. However, the dense concentrations of high geometric complexity regions in San Miguel (foliage) capture a very high amount of triangles in the sampled PVS. For such cases the QuadStream representation achieves lower memory footprint compared to the smapled PVS. The Quadstream method can be integrated with any renderer that uses a visibility (or G-Buffer) pass and shades on top of it. This fits *e.g.* Nanite-style [Karis et al. 2021] rendering, which would hardly be possible with geometry-based approaches.

(a) Transparent / Opaque G-Buffer (b) Client View (FLIP error inset)

Fig. 13. We process opaque and transparent geometry in separate GBuffers. The resulting transparent color is mixed in an order-independent manner and baked on top of the frontmost semi-transparent surface.

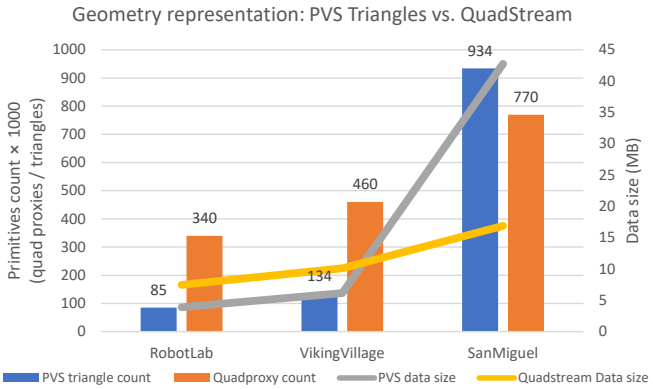


Fig. 14. Average PVS size, together with average quad proxy count for the tested configurations. QuadStream offers memory savings over a triangular PVS for viewcells containing more than 220K triangles. This fits the current trend towards pixel-sized geometry (e.g Unreal Nanite [Karis et al. 2021]).

6 CONCLUSION

We present *QuadStream*, a novel method for streaming geometry and shading information from a server to a thin client that allows for rendering from novel viewpoints and is robust against disocclusions. Our key idea is to decompose the scene into *quad proxies*, or view-aligned quads sampled from multiple views. In our experiments, QuadStream achieves superior quality compared both to streaming methods that rely on video data, and to geometry-based streaming.

Limitations. QuadStream does not attempt to address lighting effects that are view dependent such as specular highlighting, where a change to a novel view in the view cell may cause a change in highlight position. In practice, we find that small discrepancies between correct specular highlights from a novel view and our reconstruction under a novel view are not noticeable; however, this remains an important area for future work in cloud rendering. While we could show that QuadStream also supports transparency, large numbers of transparent fragments can lead to arbitrary increases in the data load. Especially, transparent particle systems may generate many disconnected small fragments, which will result in disjoint small quads. In theory we could merge particles into larger quad proxies. However, this requires further research into adaptive merging strategies which still result in acceptable novel view renderings under camera offsets.

QuadStream introduces visual artifacts around object boundaries and silhouettes with a similar appearance to aliasing. These artifacts are a side effect of our quad splitting strategy (Fig. 7); splitting a quad proxy introduces alpha-masked surfels at silhouettes even for fully opaque surfaces, and when the projection of multiple split quad proxies (constructed from multiple proxy views) coincides in the client view, transparency mixing may result in incorrect visuals. While these artifacts can be masked by temporal or morphological anti-aliasing on the client, improving reconstruction at quad splits remains an important area of future research.

Future Work. QuadStream opens up a number of areas for interesting future work. An important first extension to our method is the use of temporally stable atlases, in which we maintain running updates to the view cell and only stream those portions of the atlas that the server believes have been invalidated since the last client update. It would also be interesting to combine our quad-based technique with other forms of approximate geometry representation, such as imposters. Finally, while we are robust to disocclusions due to our rendering of novel viewpoints, we do not guarantee that every disocclusion possible is rendered with a fixed degree of accuracy. In the future, we may seek to refine our sampling of disocclusions, or to optionally reconstruct disocclusions on the thin client with a combination of novel viewpoint quad proxies and a machine learning-based infilling approach.

REFERENCES

Pontus Andersson, Jim Nilsson, Tomas Akenine-Möller, Magnus Oskarsson, Kalle Åström, and Mark D. Fairchild. 2020. FLIP: A Difference Evaluator for Alternating Images. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 2 (2020), 15:1–15:23.

Paul Bao and Douglas Gourlay. 2004. Remote walkthrough over mobile networks using 3-D image warping and streaming. *IEE Proceedings - Vision, Image and Signal Processing* 151, 4 (Aug 2004), 329–336.

Kevin Boos, David Chu, and Eduardo Cuervo. 2016. FlashBack: Immersive Virtual Reality on Mobile Devices via Rendering Memoization. In *MobiSys*. 291–304.

Huw Bowles, Kenny Mitchell, Robert W. Sumner, Jeremy Moore, and Markus Gross. 2012. Iterative Image Warping. *Computer Graphics Forum* 31, 2pt1 (2012), 237–246.

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. 39, 4 (2020), 86:1–86:15.

Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. 2001. Unstructured Lumigraph Rendering. In *Proc. SIGGRAPH*. 425–432.

Chun-Fa Chang and Shyh-Haur Ger. 2002. *Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1105–1111.

Shenchang Eric Chen and Lance Williams. 1993. View interpolation for image synthesis. In *SIGGRAPH*. 279–288.

Daniel Cohen-Or, Yair Mann, and Shachar Fleishman. 1999. Deep Compression for Streaming Texture Intensive Animations. In *SIGGRAPH*. 261–267.

Alvaro Collet, Ming Chuang, Pat Sweeney, Don Gillett, Dennis Evseev, David Calabrese, Hugues Hoppe, Adam Kirk, and Steve Sullivan. 2015. High-quality Streamable Free-viewpoint Video. *ACM Trans. on Graph.* 34, 4 (July 2015).

Eduardo Cuervo, Alec Wolmany, Landon P. Coxz, Kiron Lebeck, Ali Razeenz, Stefan Saroiuy, and Madanlal Musuvathi. 2015. Kahawai: High-Quality Mobile Gaming Using GPU Offload. In *MobiSys*. 121–135.

Xavier Décoret, Frédo Durand, François X. Sillion, and Julie Dorsey. 2003. Billboard Clouds for Extreme Model Simplification. *ACM Trans. Graph.* 22, 3 (jul 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. *Computer Graphics Forum* 29, 2 (2010), 713–722.

Piotr Didyk, Tobias Ritschel, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel. 2010b. Adaptive Image-space Stereo View Synthesis. In *15th International Workshop on Vision, Modeling and Visualization Workshop*. Siegen, Germany, 299–306.

Mingsong Dou, Sameh Khamis, Yury Degtyarev, Philip Davidson, Sean Ryan Fanello, Adarsh Kowdle, Sergio Orts Escolano, Christoph Rhemann, David Kim, Jonathan Taylor, Pushmeet Kohli, Vladimir Tankovich, and Shahram Izadi. 2016. Fusion4D: Real-Time Performance Capture of Challenging Scenes. *ACM Trans. Graph.* 35, 4, Article 114 (jul 2016), 13 pages. https://doi.org/10.1145/2897824.2925969

Yilun Du, Yinan Zhang, Hong-Xing Yu, Joshua B. Tenenbaum, and Jiajun Wu. 2020. Neural Radiance Flow for 4D View Synthesis and Video Processing. *arXiv preprint arXiv:2012.09790* (2020).

Inc Epic Games. 2020. *Unreal Engine 5.* https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5

John Flynn, Michael Broxton, Paul Debevec, Matthew DuVall, Graham Fyffe, Ryan Overbeck, Noah Snavely, and Richard Tucker. 2019. Deepview: View synthesis with learned gradient descent. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2367–2376.

Stephan J. Garbin, Marek Kowalski, Matthew Johnson, Jamie Shotton, and Julien Valentin. 2021. FastNeRF: High-Fidelity Neural Rendering at 200FPS. (2021). arXiv:2103.10380 [cs.CV]

Michael Garland and Paul S Heckbert. 1997. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. 209–216.

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

Steven J Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F Cohen. 1996. The lumigraph. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 43–54.

Peter Hedman, Pratul P. Srinivasan, Ben Mildenhall, Jonathan T. Barron, and Paul Debevec. 2021. Baking Neural Radiance Fields for Real-Time View Synthesis. *arXiv* (2021).

Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. 2019a. The Camera Offset Space: Real-time Potentially Visible Set Computations for Streaming Rendering. *ACM Trans. Graph.* 38, 6, Article 231 (nov 2019), 14 pages. https://doi.org/10.1145/3355089.3356530

Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. 2019b. Tessellated Shading Streaming. *Computer Graphics Forum* (2019). https://doi.org/10.1111/cgf.13780

Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. 2021. SnakeBinning: Efficient Temporally Coherent Triangle Packing for Shading Streaming. *Computer Graphics Forum* (2021). https://doi.org/10.1111/cgf.142648

Hugues Hoppe. 1996. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 99–108.

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

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 (jul 2018), 15 pages. https://doi.org/10.1145/3203193

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 *Proc. Mobile Systems, Applications, and Services*.

Marc Levoy and Pat Hanrahan. 1996. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. 31–42.

Zhengqi Li, Simon Niklaus, Noah Snavely, and Oliver Wang. 2020. Neural Scene Flow Fields for Space-Time View Synthesis of Dynamic Scenes. *arXiv:2011.13084* (2020).

David B Lindell, Julien NP Martel, and Gordon Wetzstein. 2020. AutoInt: Automatic Integration for Fast Neural Volume Rendering. *arXiv:2012.01714* (2020).

Lingjie Liu, Jiatao Gu, Kyaw Zaw Lin, Tat-Seng Chua, and Christian Theobalt. 2020. Neural sparse voxel fields. *arXiv preprint arXiv:2007.11571* (2020).

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/

Yair Mann and Daniel Cohen-Or. 1997. Selective Pixel Transmission for Navigating in Remote Virtual Environments. *Computer Graphics Forum* 16 (1997), C201–C206.

William R. Mark, Leonard McMillan, and Gary Bishop. 1997. Post-rendering 3D warping. In *Proc I3D*.

Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. 2020. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European Conference on Computer Vision*. Springer, 405–421.

Joerg H. Mueller, Thomas Neff, Philip Voglreiter, Markus Steinberger, and Dieter Schmalstieg. 2021. Temporally Adaptive Shading Reuse for Real-Time Rendering and Virtual Reality. *ACM Trans. Graph.* 40, 2, Article 11 (apr 2021), 14 pages.

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 (2018), 16 pages.

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.

Thomas Neff, Pascal Stadlbauer, Mathias Parger, Andreas Kurz, Joerg H Mueller, Chakravarty R Alla Chaitanya, Anton Kaplanyan, and Markus Steinberger. 2021. DONeRF: Towards real-time rendering of compact neural radiance fields using depth oracle networks. *arXiv preprint arXiv:2103.03231* (2021).

Yuval Noimark and Daniel Cohen-Or. 2003. Streaming Scenes to MPEG-4 Video-Enabled Devices. *IEEE Computer Graphics and Applications* 23 (01 2003), 58–64.

NVIDIA. 2021. *GeForce Now.* https://www.nvidia.com/en-us/geforce-now/

OculusVR. 2018. Rendering to the Oculus Rift. Visited on March 30, 2018..

Dawid Pajak, Robert Herzog, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel. 2011. Scalable Remote Rendering with Depth and Motion-flow Augmented Streaming. *Computer Graphics Forum* 30, 2 (2011), 415–424.

Keunhong Park, Utkarsh Sinha, Jonathan T. Barron, Sofien Bouaziz, Dan B Goldman, Steven M. Seitz, and Ricardo Martin-Brualla. 2020. Deformable Neural Radiance Fields. *arXiv preprint arXiv:2011.12948* (2020).

Eric Penner and Li Zhang. 2017. Soft 3D reconstruction for view synthesis. *ACM Transactions on Graphics (TOG)* 36, 6 (2017), 1–11.

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*. 335–342.

Albert Parra Pozo, Michael Toksvig, Terry Filiba Schrager, Joyce Hsu, Uday Mathur, Alexander Sorkine-Hornung, Rick Szeliski, and Brian Cabral. 2019. An Integrated 6DoF Video Camera and System Design. *ACM Trans. Graph.* 38, 6 (2019).

Albert Pumarola, Enric Corona, Gerard Pons-Moll, and Francesc Moreno-Noguer. 2020. D-NeRF: Neural Radiance Fields for Dynamic Scenes. *arXiv:2011.13961* (2020).

Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, et al. 2021. *Warehouse-scale video acceleration: co-design and deployment in the wild.* 600–615.

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

Christian Reiser, Songyou Peng, Yiyi Liao, and Andreas Geiger. 2021. KiloNeRF: Speeding up Neural Radiance Fields with Thousands of Tiny MLPs. (2021). arXiv:2103.13744 [cs.CV]

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*. 409–416.

Sara Fridovich-Keil and Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. 2022. Plenoxels: Radiance Fields without Neural Networks. In *CVPR*.

Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. 1998. Layered Depth Images. In *Proc. SIGGRAPH*. 231–242.

Shu Shi and Cheng-Hsin Hsu. 2015. A Survey of Interactive Remote Rendering Systems. *ACM Comput. Surv.* 47, 4, Article 57 (may 2015).

Shu Shi, Klara Nahrstedt, and Roy Campbell. 2012. A Real-time Remote Rendering System for Interactive Mobile Graphics. *ACM Trans. Multimedia Comput. Commun. Appl.* 8, 3s, Article 46 (oct 2012), 20 pages.

Vincent Sitzmann, Justus Thies, Felix Heide, Matthias Nießner, Gordon Wetzstein, and Michael Zollhofer. 2019. Deepvoxels: Learning persistent 3d feature embeddings. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

Michael Stengel, Zander Majercik, Benjamin Boudaoud, and Morgan McGuire. 2021. A Distributed, Decoupled System for Losslessly Streaming Dynamic Light Probes to Thin Clients. arXiv:2103.05875 [cs.DC]

Eyal Teler and Dani Lischinski. 2001. Streaming of Complex 3D Scenes for Remote Walkthroughs. *Computer Graphics Forum* 20, 3 (2001), 17–25.

Justus Thies, Michael Zollhöfer, and Matthias Nießner. 2019. Deferred neural rendering: Image synthesis using neural textures. *ACM Transactions on Graphics* (2019).

Zhou Wang and Qiang Li. 2010. Information content weighting for perceptual image quality assessment. IEEE Image Proc. 20(5), 1185-1198. *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society* 20 (11 2010), 1185–98.

Suttisak Wizadwongsa, Pakkapon Phongthawee, Jiraphon Yenphraphai, and Supasorn Suwajanakorn. 2021. NeX: Real-time View Synthesis with Neural Basis Expansion. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Wenqi Xian, Jia-Bin Huang, Johannes Kopf, and Changil Kim. 2020. Space-time Neural Irradiance Fields for Free-Viewpoint Video. *arXiv preprint arXiv:2011.12950* (2020).

Lei Yang, Yu-Chiu Tse, Pedro V. Sander, Jason Lawrence, Diego Nehab, Hugues Hoppe, and Clara L. Wilkins. 2011. Image-based Bidirectional Scene Reprojection. *ACM Trans. Graph.* 30, 6, Article 150 (dec 2011), 10 pages.

Ilmi Yoon and Ulrich Neumann. 2000. Web-Based Remote Rendering with IBRAC. *Computer Graphics Forum* 19, 3 (2000), 321–330.

Alex Yu, Ruilong Li, Matthew Tancik, Hao Li, Ren Ng, and Angjoo Kanazawa. 2021. PlenOctrees for Real-time Rendering of Neural Radiance Fields. In *arXiv*.

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 (jul 2018), 12 pages. https://doi.org/10.1145/3197517.3201323

C. Lawrence Zitnick, Sing Bing Kang, Matthew Uyttendaele, Simon Winder, and Richard Szeliski. 2004. High-Quality Video View Interpolation Using a Layered Representation. *ACM Trans. Graph.* 23, 3 (aug 2004), 600–608.