Public Review for

A Distributed, Decoupled System for Losslessly Streaming Dynamic Light Probes to Thin Clients

Michael Stengel, Zander Majercik, Benjamin Boudaoud, Morgan McGuire

This paper is well-written and logically structured. It proposed a distributed graphics pipeline that computes diffuse global illumination on a remote server and direct illumination on a client to render high-quality graphics in real time. The state-of-the-art distributed graphics systems remotely render complete frames and then deliver the encoded frames to (low-end) clients. The limitation of this approach is that for the multi-user scenario, the graphics workload scales linearly with the number of users, making the real-world deployment challenging. Moreover, the constraint of a small form factor and the low energy consumption that is required for facilitating user comfort when rendering diverse graphics content for virtual reality (VR) and augmented reality (AR) necessitate an intelligent split of the pipeline between the remote server and the client.

In order to address the above issues, this paper leverages a key observation that view-independent effects such as diffuse global illumination are shared by multiple users and thus should be rendered on a remote server to amortize computation overhead among viewers. Hence, the authors extend the previous work on dynamic diffuse global illumination light probes (a new lighting representation) and improve the efficiency of the pipeline with a low-latency compression scheme for those light probes. By splitting the pipeline in this way, the proposed work enables the rendering of high-fidelity ray-traced global illumination remotely with low latency and makes the latest hardware features available to users with only a thin client. Furthermore, instead of streaming remotely-rendered full frames, this paper proposes to selectively update light probes, which avoids sending those not containing information relevant to client-side shading.

The authors built a prototype of the proposed solution and evaluated its performance on a real testbed with three scenes that cover different combinations of static/dynamic geometry, lighting conditions, and size. Benefiting from GPU-accelerated compression and other optimizations, the experimental results demonstrate that the proposed distributed system can indeed achieve both high quality lighting and high frame rate that are needed for a good quality of user experience at scale for many thin clients. Given the increasing popularity of emerging applications such as mobile cloud gaming, VR, AR, and mixed reality (MR), this paper sheds light on further improving the performance and quality of experience for those immersive applications that involve high-quality and real-time graphics rendering.

Public review written by
Bo Han
George Mason University, USA

A Distributed, Decoupled System for Losslessly Streaming Dynamic Light Probes to Thin Clients

Michael Stengel
NVIDIA
mstengel@nvidia.com

Zander Majercik
NVIDIA
amajercik@nvidia.com

Benjamin Boudaoud
NVIDIA
bboudaoud@nvidia.com

Morgan McGuire
NVIDIA
mcguire@nvidia.com

ABSTRACT

We present a networked, high-performance graphics system that combines dynamic, high-quality, ray traced global illumination computed on a server with direct illumination and primary visibility computed on a client. This approach provides many of the image quality benefits of real-time ray tracing on low-power and legacy hardware, while maintaining a low latency response and mobile form factor.

As opposed to streaming full frames from rendering servers to end clients, our system distributes the graphics pipeline over a network by computing diffuse global illumination on a remote machine. Diffuse global illumination is computed using a recent irradiance volume representation combined with a new lossless, HEVC-based, hardware-accelerated encoding, and a perceptually-motivated update scheme.

Our experimental implementation streams thousands of irradiance probes per second and requires less than 50 Mbps of throughput, reducing the consumed bandwidth by 99.4% when streaming at 60 Hz compared to traditional lossless texture compression.

The bandwidth reduction achieved with our approach allows higher quality and lower latency graphics than state-of-the-art remote rendering via video streaming. In addition, our split-rendering solution decouples remote computation from local rendering and so does not limit local display update rate or display resolution.

CCS CONCEPTS

• **Computer systems organization → Client-server architectures**; • **Computing methodologies → Ray tracing**.

KEYWORDS

Distributed Graphics, Cloud Rendering, Light Probes, Global Illumination, Split Rendering

ACM Reference Format:
Michael Stengel, Zander Majercik, Benjamin Boudaoud, and Morgan McGuire. 2021. A Distributed, Decoupled System for Losslessly Streaming Dynamic Light Probes to Thin Clients. In *ACM Multimedia Systems Conference (MMSys '21), September 28–October 1, 2021, Istanbul, Turkey.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458305.3463379

MMSys '21, September 28–October 1, 2021

ACM ISBN 978-1-4503-8434-6/21/09…$15.00
https://doi.org/10.1145/3458305.3463379

1 INTRODUCTION

Motivation. Today's high performance graphics systems approach cinematic rendering quality in real time using ray traced global illumination, with interaction latency measured in milliseconds. For applications like virtual and augmented reality (VR/AR), the need for low latency and high-fidelity rendering is augmented by a need for a small form factor and low thermal requirements to facilitate user comfort.

A small form factor and good thermals, however, directly constrain high-fidelity rendering at low latency. This constraint becomes more severe as rendering requirements increase and desired form factor and thermal budget decrease, making it impossible to render high quality graphics directly on AR/VR/mobile devices.

Distributed Graphics. The state of the art for accelerating high-quality rendering on heterogeneous, low-end clients is interactive video streaming. Several cloud gaming services distribute the graphics workload by running games on remote servers. These systems stream user inputs to the server and rendered video back to local clients. At standard dynamic range and 60 Hz frame rate, these solutions already consume moderate bandwidth despite leveraging high efficiency video encoding (HEVC). They can also experience significant latency, as the display framerate is limited by the network round trip time. In addition, rendering each user view as a full frame does not amortize computation over multiple users, causing the graphics workload to scale linearly with the number of users.

Decoupled Global Illumination. The computational cost of streaming full frames scales linearly because of view-dependent rendered effects: each client has a different viewpoint and so a completely different image must be rendered.

One example of a view dependent effect is a mirror reflection: when the viewer sees a mirror, what they see in it is highly dependent on their viewing angle. High-fidelity rendering, however, also comprises view-independent effects.

Diffuse global illumination is one example: light reflected from a rough surface, like stone or brick. Under a Lambertian reflection model, sufficiently rough surfaces reflect the same light in all directions no matter the direction of the outgoing eye ray [39]. This suggests separating one or more view-independent effects out of the full graphics pipeline and rendering them on a remote machine in order to amortize computation for multiple viewers. This is our approach. Specifically, we render the diffuse global illumination on a remote machine and then stream that data to a thin client where it can be efficiently queried during local rendering.

Shifting the diffuse global illumination (also referred to in our paper as global lighting) off-device has several unique advantages: a) As with full frame streaming, hardware limitations of thin clients

are avoided by rendering state-of-the-art ray-traced global illumination remotely, b) new hardware features can be integrated in a server long before they are propagated to thin clients, c) data for global illumination can be amortized over multiple users in the same virtual environment because it is not view dependent; decoupling it from users, frames, and pixels allows increased scaling and lowers the total cost of rendering, d) all but the crudest approximations of global illumination are too expensive to compute on low-end graphics devices, so computing global illumination remotely increases quality by enabling higher fidelity approximations, and e) significant lag in diffuse global illumination updates is perceptually tolerable (up to nearly half a second) [8].

Contributions. To the best of our knowledge our paper presents the first distributed system that enables streaming low-latency, dynamic, high-quality, ray-traced global illumination with visibility information from a remote machine to one or more thin clients.

Our contributions are:

(1) A system design for multi-client, distributed, high-quality update and streaming of global irradiance volumes with visibility data
(2) A low-latency, GPU-accelerated, lossless texture encoding and decoding scheme supporting 3x10-bit color and 2x16-bit visibility information
(3) Per-client prioritization schemes for GI probe updates reducing network traffic and encoding/decoding latency, and amortizing rendering across clients
(4) Specific best practices for avoiding CPU, GPU, or network stalls on servers and clients during lighting and state synchronization
(5) Evaluation on low-end, mobile gaming and AR/VR devices

2 RELATED WORK

The stages of an application's user interaction loop are: user input → simulation → CPU draw call generation → GPU graphics pipeline (indirect lighting → shadows → geometric culling → primary visibility → direct lighting) → framebuffer → display. Many of the system design challenges in distributed graphics stem from choosing the most efficient location(s) in this loop for a network connection and creating efficient representations for synchronizing data across it. The goals are to achieve low latency and low bandwidth consumption at high image quality, low system complexity, and relatively low client requirements.

The two extreme cases, fully local and remote rendering, are widely deployed due to their practical simplicity, while networking between CPU and GPU has been used in limited contexts (e.g., 2D, data visualization). *Split*, also called *hybrid* or *collaborative, rendering*, which uses both client and server-side 3D rendering, is an active research area that includes our work. Split renderers divide the graphics pipeline into two or more pieces connected by a network. The cost is increased application development overhead, but the opportunity is improvement in nearly all runtime metrics.

The disadvantages are that image quality is gated by the client device, and it is hard to support heterogeneous clients as new rendering features must be implemented for each one separately, sometimes using different algorithms.

2.1 Streaming Framebuffer

Remote rendering with a thin client distributes work by sending user input from a client to a server, which renders complete frames and streams them back to the client as video. This is comparable to replacing the connection between user input device and simulation engine, as well as framebuffer and display with a network [21, 37]. This method is used both by cloud gaming services and productivity-based remote desktop applications [4, 10]. Typically, for cloud gaming the application runs in a virtual machine (VM) on the cloud or edge server allowing the user input and network-attached framebuffer to appear local to the game engine [41].

The advantages of streaming full or partial frames include that it is a lowest common denominator approach suitable for heterogeneous, low-powered clients and leverages the mature technology ecosystem around efficient video compression. The disadvantages are that end-to-end video streaming can introduce latencies of above 100 ms [6, 10, 46], which can be hidden for video applications by buffering, but can be unacceptable for interactive applications and competitive first-person video games [1, 5, 30, 41, 42]. While video compression scales sublinearly in terms of bandwidth [44], the throughput requirements still grow with resolution and frame rate, and image quality can be suboptimal due to lossy encoding. Including additional information unique to 3D rendered content, such as depth and motion vectors, can provide better robustness on unreliable networks [35]. Gaze tracking information captured by the client in real time can be used to control block-wise video compression on the server, reducing the consumed network bandwidth up to 50% for high resolution video [19, 20].

More aggressive stream compression research renders a local, low-quality estimate of each frame, so only differences between the local render and the remote high-quality frame must be streamed [5, 22]. This is a more efficient compression mechanism than naïve video, but it does not address the latency or scaling problems of producing per-frame, per-pixel results on a server [5].

2.2 Streaming Shaded Textures

In this approach, texture space shading is used to reorder the graphics pipeline with direct lighting before visibility, and the system is split at this point, decoupling shading from local display refresh rate and resolution [16].

The systems challenges include computing and streaming dynamic atlases of texture-space lighting data while not exceeding available bandwidth and avoiding overshading (rendering imperceptible texture detail or unseen areas of the texture).

The algorithmic challenge is that most texture-space solutions are restricted to Lambertian surfaces and cannot produce effects such as as glossy highlights. All previous systems implementing these solutions [17, 31] have either no direct shadows or high latency in the shadows, which is why we focus on streaming indirect lighting and keep direct visibility shading local to the client.

Mueller et al. [31] achieve end-to-end latency of 86 ms for lossy results at 40 Mbps. Hladky et al. [17] use fewer samples due to better atlas packing but require a bandwidth of 45 Mbps. With comparable network performance, our system solves a complementary problem by providing lossless indirect lighting with both diffuse and rough glossy reflections.

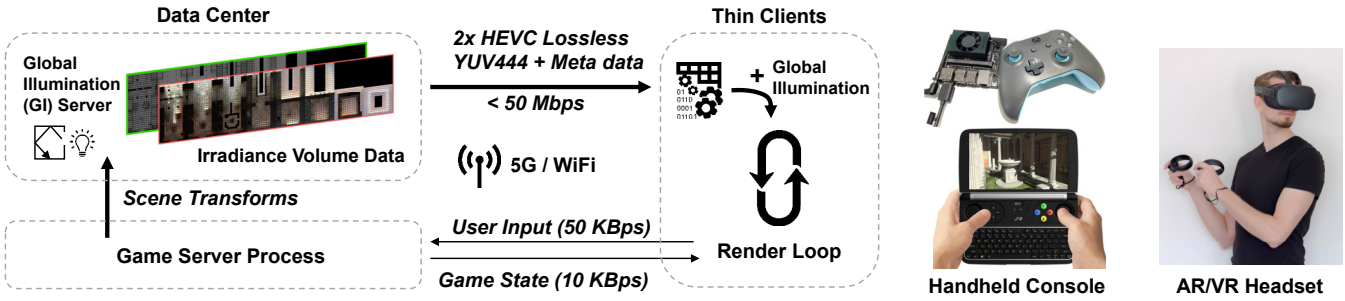

Figure 1: Light probe streaming architecture. Our system streams irradiance volume data from a cloud server to thin clients for mobile gaming and AR/VR. The streamed data enables dynamic high quality ray traced indirect lighting on the clients at low computational cost. We use efficient encoding and decoding hardware features to achieve high compression ratios with low latency. Thus a single server can update thousands of light probes per seconds across multiple connected thin clients, amortizing rendering costs.

2.3 Streaming Indirect Lighting

Computing view-independent indirect lighting on the remote server and primary visibility on the local client is another interesting approach. This has the advantage of providing minimal latency for interaction and camera movement, while offloading the most expensive and hardware feature-driven aspect of the rendering pipeline to the cloud. Using this approach, indirect lighting updates can be decoupled from frame rate and resolution. The added latency in lighting effects has been shown acceptable for diffuse indirect light, but can be problematic when sharp reflections, shadows, or direct illumination are delayed [8].

Previous work describes good solutions for asynchronously updating and efficiently streaming irradiance maps, photon maps, voxel lights [8], virtual point lights [3, 27], light propagation volumes [24, 25], and irradiance maps combined with probes [50]. These methods use a variety of lossy video and general lossless encoding techniques.

CloudLight provided inspiration for our work, but uses outdated algorithms and lacks visibility information [8].

In the ReGGI paper Magro et al. stream virtual point lights from server to clients and use synchronized voxel grids on both ends to compute global illumination [27].

We extend this body of split lighting work with a system design that targets the same point in the pipeline with novel lossless video encoding and a newer lighting representation: Dynamic Diffuse Global Illumination (DDGI) light probes [28]. However, our system is equally applicable to other GI techniques using irradiance volumes [9, 14, 38] or voxel grids represented as textures [8]. Generally, the techniques we develop are applicable to any GI solution amenable to amortization over multiple clients and representation as streamed texture data. The open problem that remains is combining or extending the described solutions to solve the problem of low latency, remote, direct glossy reflections.

2.4 General Texture Compression

Fixed-ratio, block texture compression schemes such as DXTn, BCn, or ASTC are often GPU-accelerated for fast random-access decompression [34, 47]. However, these schemes are optimized for efficient storage of single textures in content creation pipelines such as light maps and therefore are not ideal for real-time encoding and continuous streaming as they do not exploit information of previously streamed texture data [18]. State-of-the-art video compression algorithms such as H.265 (HEVC) allow significantly better compression ratios through temporal reuse and bidirectional block prediction, and much higher encoding performance provided by dedicated hardware units on the GPU [44].

With the advent of mobile AR/VR and low-cost volumetric capture devices, video-based point cloud compression and light field streaming techniques have recently received more attention [40, 48]. Broxton et al. achieve streaming of preprocessed lightfield videos encoded as multi-sphere image representations that allow specular effects and transparencies through view-dependent alpha blending of multiple scene layers [2]. Despite efficient, lossy mesh compression and color atlas compression the approach requires high data rates of 124 Mb/s to 322 Mb/s for streaming. Our implementation is a *sparse* light field scheme resulting in lower consumed bandwidth, as it encodes and streams dynamic, prefiltered color and visibility data for light probes.

Schwarz et al. propose a projection-based compression scheme for 3D scene data. A sequence of point clouds is replaced by a sequence of textures and geometry images after the 3D object is projected to 2D surfaces [40]. These 2D videos are then encoded using H.264 video compression. Additional metadata is streamed to reconstruct the point cloud on the receiver side. The metadata includes projection geometry information, depth offsets and per-frame depth quantization. Our approach differs as we only stream indirect lighting data and rely on primary visibility being rendered on the client.

Wilson et al. propose the Run length encoding and Variable Length encoding (RVL) lossless compression scheme to reduce the bandwidth required to transfer 16-bit, single channel depth images [49]. The algorithm provides low compression and decompression times and compression ratios of about 3:1. However, since the algorithm compresses individual images the achievable compression ratio is limited compared to using video encoding strategies. Liu et al. propose a hybrid approach for improving compression ratio with a trade-off to increased complexity. The approach encodes significant bits using lossless compression and the remaining bits with lossy compression [26].

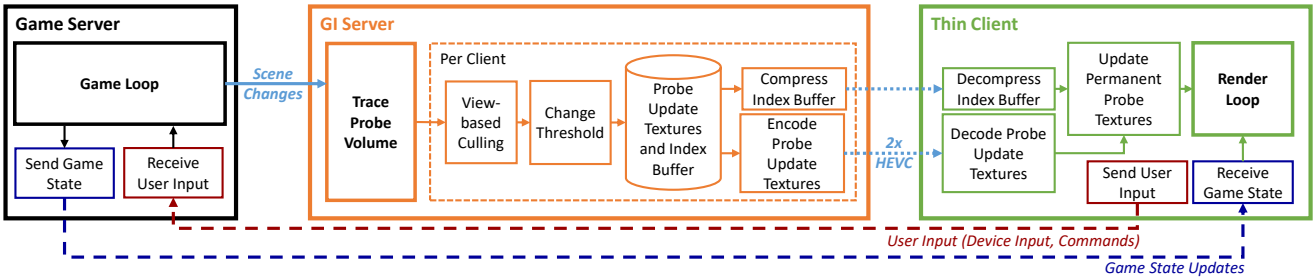

Figure 2: Data flow in light probe streaming architecture. From left to right for a single client: Game simulation loop on game server node (*black box*). Server-side irradiance volume rendering as well as per-client probe selection and encoding on GI server node (*orange box*). Network transfer of data to and from the client (*dashed lines*). Decoding of light probe data and shading on client (*green box*). Note that the game and GI server nodes are only conceptually separated, depending on the number of users and game both can run on a single machine in the data center.

3 SYSTEM DESIGN

In this section we present our system to stream GI data from a cloud rendering server to thin clients. The streamed data enables dynamic, high quality, ray traced diffuse GI on thin clients with no ray tracing capabilities, at low computational cost. We use efficient encoding and decoding hardware features to achieve high compression ratios with low latency. Further, we present heuristic light probe update schemes motivated by previous perceptual findings to reduce the amount of transferred GI data. As a result a single server can update (tens of) thousands of light probes per second, across multiple connected thin clients, amortizing the rendering costs.

3.1 Architecture Overview

Our light probe streaming pipeline can be summarized in four steps:

(1) Server-side ray tracing of the light probe volume
(2) Per-client optimized light probe data encoding
(3) Network transfer of encoded data to the client
(4) Client-side light probe data decoding and rendering

Figure 1 demonstrates the high-level design principle, whereas the data flow is described in Figure 2. Our client-server architecture distributes the lighting pipeline across a server and (multiple) clients instead of streaming full video frames. The server, assumed to be in a data center, conceptually consists of two nodes interconnected with a high bandwidth link. The first node acts as the game server to reliably receive user input from clients and update the game state accordingly. This is analogous to a "typical" multiplayer game server. The second node, the GI server, performs ray tracing of light probe data (Fig. 3) and encodes it for client transport. Any scene change is assumed to be reported to the GI server by the game server with negligible latency.

Using the current scene state, the GI server computes diffuse light probe texture data that is view-independent. As a result, this GI data can be reused by multiple clients in multiplayer game scenarios, amortizing render costs.

We exploit hardware-accelerated video encoding on the server to compress the light probe data. The encoded data is transferred to the client(s) over a combination of WiFi, 5G, and/or wired connections using a reliable low latency network protocol. Specifically, our prototype implements reliable UDP provided by the ENet library [11], although our architecture does not rely on any particular low-latency protocol or network implementation.

On the client side we use hardware video decoding (when available) for decompressing the light probe data with low latency. The client render loop then uses the uncompressed light probe textures to add dynamic and high quality diffuse GI at low computational cost. The GI texture decoding process is decoupled from the render loop and can happen at a completely different rate. This allows the client to run at full frame rate resulting in minimal perceived latency on user input or movement. Input latency, as it occurs in traditional cloud-rendering solutions, is therefore avoided.

Rendering GI data remotely offers specific benefits such as: (1) diffuse GI is view-independent and used across multiple users and potentially multiple frames; (2) rendering GI data is computationally too demanding for thin client hardware; and (3) users are less sensitive to delayed diffuse lighting [8] than view-dependent effects.

In the remainder of this section we explain our probe data encoding and decoding in detail. First, as a baseline in Section 3.2 we present streaming of complete and uncompressed irradiance volumes once per GI update. Second, in Section 3.3 we show how low-latency texture compression can significantly reduce the required network bandwidth per update. Furthermore, in Section 3.4 our scene-dependent and user-dependent selection strategies reduce the number of transferred light probes. In combination, our optimizations produce network bandwidth reductions of **10x to 100x** (depending the amount of changing light probe color and visibility) in comparison to uncompressed data.

3.2 Uncompressed Light Probe Streaming

Streaming Probe Color. Our decoupled rendering system adopts the irradiance volume approach by Majercik et al. [28].

Each probe uses an 8x8 texel array for the core light probe color extended by a 1-pixel guard band in each direction which is required for correct texture filtering (10x10 texels in total), with 32 bits of color per texel, to encode color data for each direction in form of an octahedral mapping (Fig.3, top). Thus, the required network throughput T_{color} for updating any given N_{probes} color probes, at a rate R_{color} (in Hz), is given by Eq. 1.

$$T_{color} = R_{color} \times N_{probes} \times 3.125 \text{ kb} \quad (1)$$

A 16x8x16 (2048) probe volume has an uncompressed DDGI color texture size of 6.25 Mb (0.78 MB). Updating 2048 uncompressed color probes at 10 Hz requires 62.5 Mbps of bandwidth.

Streaming Probe Visibility. DDGI stores mean distance and mean squared-distance for each probe. During shading, these values are used to determine visibility weights between probes and shaded points using a Chebyshev statistical test [28]. We refer to the mean distance/squared-distance data as "visibility data" and to the texture itself as the "visibility texture" (Fig.3, bottom).

The visibility texture contains 18x18 texels per probe encoded as a pair of half-precision (16-bit) floating-point values (32 bits/texel). Thus the required throughput ($T_{visibility}$) for a visibility update rate of $R_{visibility}$ is given by Eq. 2.

$$T_{visibility} = R_{visibility} \times N_{probes} \times 10.125 \text{ kb} \quad (2)$$

For the same probe volume as analyzed above for color (2048 probes), the size of the visibility texture is 3.24 times higher than that of the color texture, or 20.3 Mb (2.5 MB). Streaming the raw visibility texture at 10 Hz requires a throughput of 202.5 Mbps.

The minimum required throughput for uncompressed color and visibility textures, updated at 10 Hz for 2048 probes, is 265 Mbps. This is above practical bandwidth limitations, especially when considering multiple thin-client devices in common wireless networks. Though this naïve, uncompressed approach is lossless and maintains perfect probe "coherence", it requires unreasonably high network throughput as it sends all probe data regardless of whether the data is unchanged or likely to be used on the client. In the following sections we describe how this throughput can be reduced.

3.3 Low-latency Light Probe Compression

We target GPU-accelerated high dynamic range (HDR) video compression using the High Efficient Video Coding (HEVC) standard [44]. In comparison to earlier video compression schemes, such as H.264, HEVC allows for higher compression rates mostly due to better prediction features.

Specifically, our system exploits hardware-accelerated HDR10 encoding and decoding. The HDR10 profile allows for 10-bit color depth in videos whereas earlier low-dynamic range profiles have been limited to 8 bits. In the near future, the HDR10+ profile will provide an option for up to 16 bit color depth through dynamic metadata, though this is not (yet) an established standard and therefore not accelerated in hardware [15].

Hardware-accelerated HEVC decoders supporting HDR video are an important feature being widely adopted in the ecosystem of mobile devices, TVs, and gaming consoles as more HDR content becomes available [12]. AR/VR platform manufacturers have also announced HEVC decoding support for their next generation of display devices [36].

However, *encoding* HEVC videos is computationally expensive due to the complexity of integrated prediction models. GPU manufacturers have only recently integrated efficient HEVC encoders in hardware, allowing for a dramatic improvement in lossy and lossless encoding performance at rates faster than real time [32, 43].

Encoding Color. Since our goal is maintaining the original quality of ray traced irradiance volumes we implement *lossless* encoding and compress color data as follows.

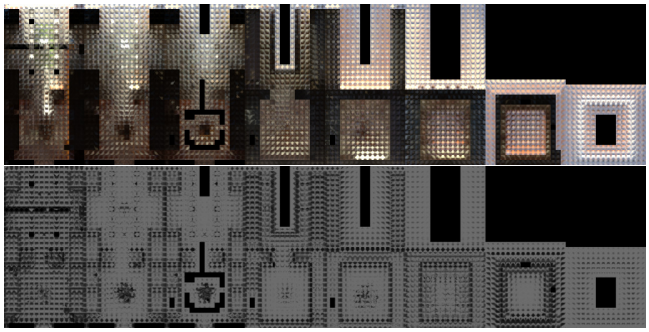

Figure 3: DDGI light probe volume textures. Light probe textures for the Greek Villa scene. The color texture (A2RGB10, top) contains 2048 individual light probes represented by 10x10 texel blocks. Black pixels represent inactive probes. The visibility texture (RG16F, bottom) includes depth values in 18x18 blocks per probe (only red color channel shown, values converted to a normalized grayscale image).

Normalized color values are saved in a A2RGB10F texture (10 bits unsigned small floats for each color, 2 unused bits for alpha). As the unsigned small floats are normalized we can quantize these values, without loss, into 10-bit unsigned integer values in the range [0,1023]. Our input format for the hardware encoder is 16 bit unsigned integers organized in 3 Y,U,V planes (linear Y-luma, U,V-chrominance). We use the YUV444 surface format to avoid chroma subsampling for lossless compression. The hardware encoder uses only 10 out of 16 bits for color encoding. The remaining bits are reserved for future versions of the codec which support 12-bit and 16-bit color encoding [15]. Therefore we set these bits to zero (Figure 4). The YUV tuples are then reordered into Y,U,V planes over which hardware encoding is applied. For decoding the process is performed in reverse. The same scheme can be also used for *perceptually lossless* encoding achieving higher compression rates.

Encoding Visibility. Our solution for visibility leverages *lossless* and *low bit depth* video encoding. The idea is to distribute a single 16-bit floating-point value across two adjacent 8-bit integer values (Figure 4, bottom). As the visibility texture for a light probe holds two channels (RG16F) we distribute the bits across four 8-bit integer values. The utilized hardware encoder does not support 4-channel images. As a workaround, we pack a sequence of three RG16F values into a sequence of four YUV values effectively distributing the bits into YUVY, UVYU, VYUV sequences. We widen the YUV texture to allocate enough memory for the visibility texture. For an original light probe visibility texture width x we increase the YUV texture width to $x' = \lceil 4/3 \cdot x \rceil$.

Threaded Client-side Decoding. In order to maximize throughput and avoid CPU blocking behavior between rendering updates and decoding incoming encoded GI data, texture decoding happens in a separate thread on the client. This unblocks the game loop during CPU-based texture decoding or GPU decoding dispatch.

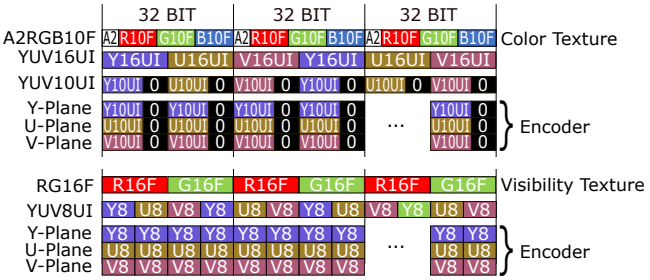

Figure 4: Memory layout for lossless texture encoding and decoding. Top: The normalized values in the A2RGB10 color texture are quantized to 10-bit unsigned integer color values and bit-shifted into 16-bit unsigned integer YUV values. The original 2-bit alpha values are not used. For hardware encoding the YUV tuples are reordered into YUV planes. Bottom: The visibility texels representing half-precision floating-point depth values are directly copied into 8-bit YUV planes to avoid quantization loss. For decoding the steps are performed in reverse order.

3.4 Selective Light Probe Updates

As a next step in optimizing network throughput per GI update we attempt to avoid sending those probes which do not contain information relevant to client-side shading. A relevant probe satisfies three conditions:

- It is being updated and used for shading in the scene
- Its affiliated texels have changed significantly since its last transmitted update to a client
- It is shading points potentially visible to a user

On the server, we gather all probes for which the preceding three conditions hold and transmit them to the client. At each transmission, we record transmitted probe data on the server and compare it to rendered probe data to determine which probes should be sent in the next transmission. Details of probe gathering and changed texel computation are given in the supplemental material.

Light Probe Repacking. Inactive probes can be removed from the original texture by moving to an index-based probe update system (selective, iterative) as opposed to a full texture transfer (global probe volume update). An additional reduction is possible for each core texel array in both color and visibility texture. The original DDGI textures include 1-pixel guard bands (per probe) for texture filtering that contain redundant information from the core texel array. This data can be removed in the packing step on the server and reconstructed without loss on the client during unpacking. The strategy reduces the size of transferred textures resulting in a benefit that is independent of the compression scheme (or uncompressed transfer). However, for temporal reuse, probe locations in the textures should be maintained over multiple frames if possible.

The index-based update scheme allows us to adjust the total packet size in flexible ways, as the number of iteratively updated probes can be specified. This effectively introduces a 'knob' to adjust throughput on a per-client basis.

In any given (well-placed) probe volume about 75% of probes are active [28]. Thus, by simply excluding probes located within/outside of static geometry a 25% reduction in network throughput can be achieved. Removing the guard bands decreases the color data by 36% and the visibility data by 21% per light probe considering a single texture set. In combination, we expect a reduction of 52% for color and 40.7% for visibility resulting in a reduction of 47.2% overall. This repacking scheme reduces the light probe update bandwidth by almost half without any loss of information compared to streaming complete irradiance volume textures.

Light Probe Atlas Texel Arrangement. As each probe texel represents directional information, it is understandable that adjacent probes contain angular redundancy. We experimented with different checkerboard patterns of probes with the goal of placing texels of neighboring probes with similar directions (used for probe tracing) adjacently in the probe atlas. However, we observed that the lossless HEVC video encoder is able to exploit redundancy best in the original DDGI layout where individual probes are arranged as self-contained blocks.

As video encoding is performed on blocks in a (large) atlas we reason that the encoder has a better chance in exploiting redundancies across *all* probes in the irradiance volume than from just angular redundancy of *neighboring* probes. Please find details of this analysis in the supplemental material.

Light Probe Update Textures. In the following we describe our strategy for selective probe updates on the server and client(s). The data flow for the described processes are visualized in Figure 2. Algorithmic pseudo code is provided in Algorithm 1 and 2.

The server stores permanent *client probe textures* as well as temporary selective *probe update textures* for each client. The permanent textures always represent the state of probes as they exist on each client. The temporary probe update textures are created per update step and store the probe information that is being renewed. Along with this texture, the client needs to be informed of which probes in the volume are being updated. This information is provided by the *probe index buffer* which maps probe update texture coordinates into the client's probe textures. In comparison to the probe update textures, the size of the probe index buffer (encoded as a single 2-bytes unsigned integer per probe) is very small and is further reduced by delta encoding. We observed average index buffer sizes of <1 kB even for large probe volumes.

For the client this process happens in reverse order (Algorithm 2). Only those light probes that have been sent to the client are updated in the permanent probe textures. The mapping from the update textures to the permanent light probe textures uses the received and decompressed probe index buffer. The remaining texels are assumed valid from the previous state. As the texture decompression and update step happen asynchronously in a parallel thread, the render loop is not blocked and can continuously run at full frame rate.

Note that until this point we have not covered how probes to be updated in the client view are selected and how we evaluate when a probe has changed. Conservatively, without loss in quality, we can select all probes that are *active* and therefore potentially contribute to shading of any part of the scene on the client side. However, to reduce the number of probes further we present optimizations on the server-side probe selection, specifically *Light Probe Change Thresholding* and *Server-side Client View Light Probe Culling*.

Light Probe Change Thresholding. To conservatively estimate all of the probes that could affect the primary client view, we consider all probes that have changed in value since the last time they were transmitted. We maintain two additional textures per connected client on the server, one for color and one for visibility, that store the state of each probe at the last time it was transmitted. At transmission time, we compare each probe to its last transmitted state and only consider it for transfer if its values changed beyond a specified threshold. Although higher thresholds might be perceptually tolerable and potentially reduce bandwidth, we conservatively mark a probe as changed if it is not texel-for-texel identical to its last transmitted state. To make this a useful heuristic (and to improve compression rates) we assume fully-converged color on the server making scene changes the only source of variance in the probe result. Before each probe is transmitted, we write its data to the respective last-transmitted texture.

Server-side Client View Probe Culling. Given a viewpoint within a scene, the subset of probe texels that might contribute to a rendered frame are only those which are close to points of primary visibility. This represents a, in some cases substantial, reduction of the overall number of active/updating probes from the full volume.

By limiting updated probes to those which contribute to shading the primary visibility frustum, the number of probes required to be updated can be reduced. Ideally a game-dependent client view prediction scheme could adjust the number of updated probes. However, such a prediction is strongly coupled to network latency and game content. Thus, we decide for a strategy that is rather application-independent. On the server we estimate the potential visible set (PVS) of probes contributing to client shading by rendering a spherical view from the position of the client. Gathering probes from this PVS estimation ensures correctness of the client view under camera rotation. The supplemental material describes this step in greater detail.

Algorithm 1: Selective probe update on server

```
ray trace probe textures;
for each client do
    bind permanent client probe textures;
    clear and bind probe update textures;
    clear and bind probe index buffer;
    // 1st pass on GPU
    calculate probe indices in client view;
    for each probe in client view do
        if probe info has changed and active then
            update probe info in client probe textures;
            append probe info to probe update textures;
            append probe to probe index buffer;
        end
    end
    // 2nd pass on GPU
    compress probe update textures;
    // cpu
    download and send probe index buffer and probe update
      textures;
end
```

Algorithm 2: Probe update on client

```
// on CPU
receive probe indices;
receive compressed probe update textures;
upload data to GPU;
// on GPU
decompress probe update textures;
update probe textures from probe update textures and
  indices;
```

4 RESULTS

We evaluate our system over three test scenes: Tomb Buddha, Greek Villa, and Night Street (Figure 5). Each scene provides different combinations of static/dynamic geometry, lighting conditions, and size. The probe volume is set to span the full scene bounds in each scene.

We evaluate our approach with regard to the network bandwidth required by a client to achieve GI update rates in the range of 10 to 30 Hz. The update rates are chosen to reach perceptually high to very high quality for indirect lighting. Detailed time plots for each scene can be found in the supplemental material.

Based on previous work by Majercik et al. [28] we consider DDGI probe grid sizes of up to 16x(8-32)x16 probes (2-8k total probes). We target full volume updates at 10-30 Hz producing probe throughput in the 20k-240k probes/s range. These update rates should allow to always stay within perceptually acceptable latency limits (< 500ms) for diffuse global illumination [8].

We use the NVIDIA Video SDK as the interface to the video encoder on the GPU that allows to allocate an unrestricted amount of parallel encoding sessions [32]. If a client does not support hardware HEVC decoding, we fall back to software decoding with the FFmpeg library at lower performance and higher CPU load. Prior to encoding we perform bit-wise conversion in a compute shader and copy the result into CUDA memory. Finally, we make sure to only copy compressed bit streams between CPU and GPU, to minimize bandwidth consumption and memory transfer costs within both the client and the server.

We optimize for latency by setting the encoder frame delay to zero. This trades off delay for compression performance, but allows our system to minimize overall latency in encoding and decoding. We also tune compression using the group of pictures (GOP) parameter for controlling the interval of reference frames. Low values are used for unreliable connections, where lost data requires a new I-frame to recover. As we use a reliable communication protocol, we use a high GOP length of 30 frames.

Testbed Systems. For our server-side tests we use a remote workstation (Intel i7-4930K CPU, 28 GB host memory, NVIDIA Quadro RTX 8000 GPU), hosting our minimal game logic and the GI updates in parallel. This remote server connects to the client via internet (3 hops), and an additional Wifi router (ASUS RT-AC5300) on the client side representing a common wireless home network configuration for online gaming and streaming.

		Tomb Buddha	Greek Villa	Night Street
Color	Uncompressed size (MB)	0.78	1.56	3.13
	Mean Comp. Ratio r_{avg}	3.4	14.1	92.4
	Min Comp. Ratio r_{min}	1.3	1.4	2.1
	Max Comp. Ratio r_{max}	4.8	16.8	164.6
Visibility	Uncompressed size (MB)	2.53	5.06	10.13
	Mean Compr. Ratio r_{avg}	4.5	208.8	1963.4
	Min Comp. Ratio r_{min}	1.1	1.1	1.5
	Max Comp. Ratio r_{max}	7.4	235.6	2807.9

Table 1: Compression ratios for full-frame lossless encoding. We provide results generated with our lossless compression scheme using HEVC encoding (group of frames = 30, no B-frames). The average compression ratio largely depends on the temporal reuse of previous results. Areas of low change in scene geometry and lighting (Night Street) allow for high compression ratios whereas highly dynamic scenes result in lower compression ratios (Tomb Buddha). Minimal compression ratios are given for I-frames. Highest ratios are achieved for P-frames.

For the client we tested 3 different hardware configurations: (1) a current generation gaming GPU (NVIDIA RTX 2080 Ti containing a single hardware video decoder) connected to a VR headset (Oculus Quest), (2) a mobile GPU platform (NVIDIA Xavier NX) representing a next-generation portable gaming console with two hardware decoding units, and (3) a handheld gaming platform (GPD Win 2) with only CPU-based decoding support.

NVIDIA Xavier NX and the GPD Win 2 (Fig. 1) both have greatly reduced system specifications compared to a gaming GPU [13, 33]. As a result, in order to allow for local rendering of direct visibility the Greek Villa scene was modified to reduce vertex count, resulting in a total triangle count of 786,030 and a total drawn triangle count of 1,572,735. The light probe volume size was also reduced to 16x8x8 (from 16x8x32).

Compression Performance. Table 1 presents lossless compression ratios for full volume encoding (including inactive probes and probe guard bands). We show these numbers as upper bounds as they include all probes that can change from frame to frame. In practice the size of the encoded probe textures are smaller as we provide several additional probe reduction strategies. The mean compression ratio (r_{avg}) is estimated by averaging encoding performance over 1000 frames. The worst compression performance is typically affiliated with independent reference frames (I-Frames). Higher compression is achieved for predicted frames (P-Frames).

We expect the amount of dynamics in lighting and geometry to have a strong influence on the compression performance. Our selected scenes represent scenarios of different dynamics in terms of color and visibility.

The Tomb Buddha scene is the most dynamic scene as the volume is inside a closed room with a collapsing roof. Over time more and more sunlight enters the area. This scenario not only changes the color received by all light probes but also the observed visibility. Permanently changing all light probe data represents the worst case for our encoding. As a result the video encoder has limited chances to reuse data from blocks from previous frames due to the required lossless encoding (mean compression ratio $r_{avg} = 3.4$ for color texture and $r_{avg} = 4.5$ for visibility texture).

		Pipeline Stage	Server	Stage Duration (ms) A	B	C
Server	1	Ray trace probe textures	5.63			
	2	Texture encoding	6.91			
	3	Network transport			L_{Tex}	
Client	4	Texture decoding				
		lossless		4.95	11.34	28.15
		(lossy)		(1.63)	(2.78)	21.57
	5	Render frame		4.48	10.52	13.21
	6	Client input transfer			L_{Input}	
Total time		lossless L_{Tex} + L_{Input} +		21.97	34.40	53.90
		(lossy)		18.65	25.84	47.32

Table 2: Latency analysis for full probe volume update. The given pipeline includes a full probe volume update. For each pipeline step we use the average result across all three test scenes (see results in Table 3). Client A measured on workstation GPU (NVIDIA RTX 2080 Ti, decoding in sequence using the single decoder unit), Client B measured on mobile GPU (NVIDIA Xavier NX, decoding using both decoders in parallel) and Client C with decoding on mobile CPU (GPD Win 2). Time series plots and statistics for network traffic and transport latency are provided for each tested scene in the supplemental material.

The Greek Villa, however, achieves a high mean compression ratio for visibility ($r_{avg} = 208.8$) as only parts of the scene geometry such as the orrery are changing.

The simulated day-night cycle results in permanent but slower change in indirect lighting than for the Tomb Buddha scene resulting in significantly higher color compression performance ($r_{avg} = 14.1$). The Night Street scene is fairly static overall and as a result we measure very good compression ratios for both textures ($r_{avg} = 92.4$ for color and $r_{avg} = 1963.4$ for visibility). The high amount of temporal reuse is close to a best-case scenario for the prediction algorithms in the video encoder.

These results show that for DDGI data, average video compression not only outperforms lossless compression routines (2:1) [7] but often even lossy fixed-ratio texture compression techniques (4:1) [34] that do not offer to exploit temporal redundancy. In terms of compression performance, our method is comparable to (lossy) supercompression techniques such as Crunch DXT or Basis [45] but our technique is orders of magnitude faster. In addition, exploiting dedicated encoding and decoding hardware units unblocks parallel processes running on the GPU. We show in Figure 6 how our solution scales with size of the GI irradiance volume.

For a single frame, our proposed strategy is not ideal in terms of compression ratio ($r_{max} = 1.3$ for color and $r_{max} = 1.1$ for visibility in the worst cases) when compared to complex compression schemes that achieve compression rates of 1.5x - 4x in average for floating-point data [23]. However, the video hardware decoder is optimized to exploit *temporal coherence* in data which occurs frequently in GI data [8, 28]. Therefore, when encoding multiple frames video compression rates quickly outperform single-frame fixed-rate compression schemes. This even holds for our most dynamic scene Tomb Buddha which has permanently changing color and visibility over a high number of probes.

	Tomb Buddha	Greek Villa	Night Street
Scene # triangles	1,195,188	4,538,959	672,206
Drawn # triangles (+shadow maps)	2,391,036	9,078,626	3,141,330
Total GI volume probes	2048 (16x8x16)	4096 (16x8x32)	8192 (32x4x64)
Total active probes	1580 (77.1%)	3349 (81.7%)	4160 (50.7%)

Figure 5: Scene statistics. We selected three scenes representative of different games scenarios. In Tomb Buddha (left) the roof caves in, resulting in highly dynamic geometry and lighting changing from very dark lighting to strong incoming sunlight. The Greek Villa (center) provides always changing lighting conditions due to a fast day/night cycle simulation. The geometry in this scene is mostly static with the exception of an oversized orrery influencing the incoming sunlight in the main room. Finally Night Street (right) contains lighting from a police car and the head lamps of a moving vehicle. *Buddha model and street scene from Morgan McGuire Computer Graphics Archive [29]. Greek Villa published under Royalty-free license from TurboSquid. Inc.*

In spite of our efficient compression scheme, 16-bit floating-point data (used for visibility representation) consumes more bandwidth and can be a challenge for large scenes with lots of moving geometry and high probe count. However, we argue that visibility changes are less likely than color changes from dynamic lights, resulting in less frequent visibility updates.

Transcoding Latency. We break down encoding and decoding delays for each scene (Table 3) and analyze end-to-end latency (Table 2). Note that the encoding time naturally increases with texture size. Most importantly however, encoding time depends on the resulting compression performance.

Starting with the smallest scene (Tomb Buddha with 2048 probes) we see an average encoding time increase of 53.9% for Greek Villa where the number of probes is doubled. When again doubling the number of probes for Night Street the average encoding time is only increased by 9.7% over Greek Villa. For visibility we see very different growths, specifically 27.9% and 60.79%. The encoding time increases sub-linearly with texture size as larger probe textures are expected to contain more redundancy. To understand these results we have to consider Table 1. The Night Street scene has much higher compression performance due to its rather static nature. Our measured encoding time also includes the required copy from GPU to host memory so that the encoded data can be sent via network.

The copy step is faster if the compression is higher. In addition the time the encoder spends on processing a frame not only depends on the frame resolution and bit depth but also on the chance for temporal reuse. Higher temporal reuse allows for faster processing.

Looking at the longest encoding delay instead of the mean is an alternative way for evaluation. This happens when an I-frame is encoded and temporal reuse is therefore impossible. All scenes have comparable worst-case compression ratios (r_{min}). When comparing encoding times for these cases we get a more objective analysis of how the encoding time behaves for various number of probes. For the color texture we observe growths of 42.3% and 34.8% when the number of light probes doubles. We measure a comparable increase for visibility data (53.5% and 39.1%).

			Tomb Buddha	Greek Villa	Night Street
Encoding	Irr.	Mean [σ]	**1.67** [0.43]	**2.57** [0.92]	**2.82** [0.85]
		Min. / Max.	1.36 / 4.14	1.60 / 5.89	2.26 / 7.94
	Vis.	Mean [σ]	**3.15** [1.79]	**4.03** [2.93]	**6.48** [3.96]
		Min. / Max.	2.38 / 13.00	3.09 / 19.96	5.32 / 27.77
Decoding	Irr.	Mean [σ]	**1.40** [0.55]	**2.62** [0.82]	**1.82** [1.11]
		Min. / Max.	0.88 / 4.19	1.37 / 6.40	1.38 / 7.94
	Vis.	Mean [σ]	**2.35** [2.01]	**2.62** [3.63]	**4.04** [5.01]
		Min. / Max.	1.25 / 12.93	1.72 / 22.71	2.89 / 30.99

Table 3: Encoding/Decoding performance. Times for lossless irradiance (Irr.) and visibility (Vis.) encoding and decoding given in milliseconds include data copy between GPU and host memory. Measured on desktop GPU (NVIDIA 2080 Ti). Standard deviations given in brackets.

In decoding times, we see comparable values (Table 3 and 2, Client A). In particular, the worst-case decoding timings are very close to their encoding counterparts. For highly compressed frames decoding is very fast. As for encoding times, latency reaches its highest values in cases of low compression ratios (I-frames) primarily due to the significantly higher amount of time spent on *uploading* the data to the GPU. The results also show that decoding visibility data is more expensive than color as the data is 3.46x larger (more texels per light probe, higher bit depth) and floating-point numbers are expected to have lower compression performance.

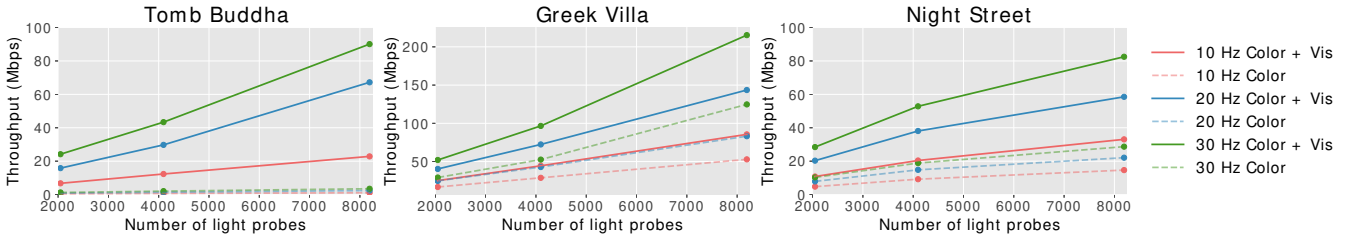

Figure 6: Consumed network bandwidth for full volume encoding. We show results for encoding the full light field volume as described in Section 3.3. For each scene we present the average network throughput for 3 different update rates (10, 20, and 30 Hz). Overall color textures (*Color*) achieve higher compression ratios and contribute less to the overall consumed bandwidth than visibility data (*Vis*). For scenes with static geometry the visibility data would not contribute to the long-term bandwidth consumption.

Full Pipeline Latency. We show results for the complete GI volume compression pipeline in Table 2. The given pipeline includes full probe volume updates and rendering, spanning the time spent on the server, network transport, and client. For each pipeline stage we present the average across all three test scenes (from Table 3).

For clarity, the texture encoding step on the server includes both textures (color and visibility) in sequence. This strategy ensures coherence of the texture information but represents the worst case in terms of latency. In practice, these textures are updated at different rates and encoded and transferred in parallel for lower latency.

Note that on the server we are using more samples for tracing each probe than proposed in the original DDGI paper by Majercik et al. [28] (256 instead of 64) as we assume converged probe textures for high image quality and better video compression performance.

On the tested hardware, our scheme is very fast on average considering the fact that all textures are encoded losslessly. The approach is especially beneficial for smaller textures. To be thorough, we also tested lossy compression for color textures resulting as expected in lower decoding times Timings (Table 2, values given in brackets). However, lossy compression requires careful perceptual quality analysis which we leave for future work. Our analysis shows that steps of the pipeline that we have control over have relatively low latency, with an average total of 22 ms on a powerful client and 34 ms on a thin client using lossless compression (19 and 26 ms when color is lossy compressed).

Depending on the hardware, the copy between GPU and host memory can become the latency bottleneck. However, for the tested probe volume sizes the proposed approach stays well within reasonable bounds for diffuse GI data [8].

On mobile hardware the copy is avoided due to unified memory (shared between CPU and GPU) leading to fast overall decoding times despite slower decoding hardware (Table 2, Client B). Without a dedicated GPU the latency increases significantly (Client C, 54 and 47 ms).

For expressing the full end-to-end latency we introduce two variables L_{Tex} and L_{Input} describing the network latency for the texture transfer from server to client and the user input data from client to game server. These delays are variable as they depend on the underlying network connection. We can safely assume 20 ms as a realistic user input delay (L_{Input}) and 100 ms as texture transfer delay (L_{Tex}). Under realistic internet streaming conditions (with 3 hops) we measure the following average texture transfer latency:

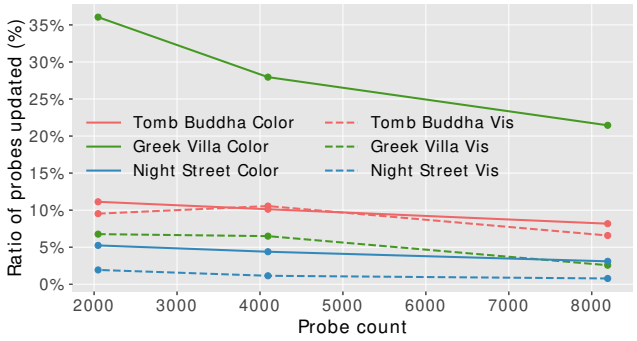

Figure 7: Selective probe reduction performance. Average percentage of probes updated (over time) with our potentially visible set method and the unchanged probes removed. Results are shown across various scenes and probe counts. We achieve reduction in transmitted probe count of between 65% and 99.5% of active probes. See the supplemental material for further comparison.

25.2 ms (σ = 18.23 ms) for Tomb Buddha, 35.2 ms (σ = 13.81 ms) for Night Street and 43.1 ms (σ = 24.45 ms) for Greek Villa. Time series plots for latency and encoded texture size for each scene are provided in the supplemental material.

Given these assumptions, the end-to-end latency of 150 ms stays well below the recommended bounds of 500 ms for diffuse GI [8]. This leaves a significant headroom of 350 ms for latency 'hitches' that can occur due to network congestion.

Selective Probe Transmission. On the server, all probes are necessary to encode the view-independent irradiance field with visibility. Only a fraction of these probes are necessary to correctly shade the client view (see Section 3.4). Culling unnecessary probes allows us to send fewer probes during transmission, thus reducing bandwidth.

In each transmission, we consider only probes that changed since they were last sent. We further cull from this set probes that do not shade any surface potentially visible from the client's camera position. The ratio of probes transmitted using this technique relative to sending all active probes is given in Figure 7. Further details of the probe selection process are given in the supplemental material.

Amortized Rendering and Scalability. Modern games range in player count from a single player to hundreds of players (MMOs) sharing a single world that could be as small as a single room, or as large as a modern city. While higher player-count games (i.e. Battle Royales) tend to come with larger scenes, the amortization benefits of our GI approach depends more on player *density* than on the player count or scene size alone. The more players that share a common set of probes, the more a server can amortize the computation affiliated with updating those probes.

Irradiance volume data is view independent and globally rendered by the server for all clients once per update. As a result any cost affiliated with this step can be effectively amortized across clients. However, our client-view specific optimizations must be completed on a per-client basis.

For many clients the server primarily needs to keep up with the total number of encoded textures as the ray tracing hardware is fast enough to trace the DDGI probes (used for all clients) as well as only additional tens of thousands of rays for probe selection for all clients. The GPU used in server setup is able to encode 1000+ frames per second using a single video encoder unit. A single GPU with one video encoder can therefore serve 16 clients at a GI update rate of 30 Hz with activated client-specific probe selection resulting in significant amortization compared to a single server instance per client. A dedicated server GPU with many video encoding chips could furthermore improve amortization by increasing overall GPU utilization until ray tracing capabilities are saturated.

5 DISCUSSION

GI Quality and Performance. In comparison to previous work our system provides better GI quality, lower latency and better compression.

ReGGI [27] can provide good quality but only at a high voxel resolution grid that needs to be reconstructed per frame for dynamic scenes. High-resolution voxel data is too expensive to transfer and the involved interpolation scheme requires significant computation on the client side. Maintaining a voxel grid in CPU memory is generally costly for sufficiently complex scenes.

We use DDGI which is the state-of-the-art algorithm for high quality realtime GI as it provides converged noise-free indirect lighting and it directly computes visibility information to avoid light leaks. DDGI does not require construction of a voxel representation of the scene but saves probe tracing results directly in textures.

ReGGI has to work with a low-resolution voxel grid and reduced GI update rates (9 to 10 Hz) to enable reasonable bandwidth requirements and keep the computation on the client sufficiently low whereas our system allows to stream DDGI data losslessly at 30 Hz due to very efficient compression with low latency and low computation overhead for server and client.

We don't perform expensive computation or interpolation on the client and decompress the GI data on the GPU using low latency video decoding. Our GPU-accelerated encoding/decoding scheme achieves compression rates of 3:1 to 92:1 (irradiance, lossless) and 4:1 to 1963:1 (visibility, lossless) and much higher rates if we accept loss whereas ReGGI uses general CPU-based compression with compression rates of less than 2:1 resulting in less GI information being available on the client side.

Color Compression. Our work focuses on lossless probe color compression for maximum image quality. As our compression analysis in Section 3.3 shows, lossy compression can leverage probe texel redundancy more successfully to further reduce the required bandwidth. Careful analysis of the perceptual impact of compression artifacts in GI data is needed to fully exploit lossy compression. We leave a perceptual investigation to future work.

Visibility Compression. We encode visibility information in irradiance probes losslessly to ensure preventing of light and shadow leaks. Our use of hardware encoding and delta compression is effective but does not take full advantage of spatial entropy analysis as in lossy supercompression methods. When using lossy compression, splitting 16-bit floating-point values across YUV color channels is a poor strategy, as lost bits can create severe artifacts in the reconstructed GI. An alternative is quantization to a lower range of integer values so that the values can be expressed in a single color channel. Although not without artifacts this approach limits the severity of potentially visible artifacts in rendering.

Certain software implementations of H.264 and HEVC support higher bit depth encoding (12 to 16 bit) but hardware encoding and decoding of 16-bit values is not yet available on consumer GPUs.

As an alternative, research by Liu et al. introduces a combination of lossless compression of the most significant bits and lossy compression otherwise [26]. This strategy can improve the compression ratio but increases transcoding complexity for servers and clients. This is left to future work.

Variable GI Update Rates. Depending on the nature of scene dynamics and probe volume organization, it may be possible to update GI for different parts of a scene at rates tuned to their local dynamics. The simplest version of this approach would involve the use of multiple GI probe volumes, with each volume updated at a rate specific to its coarseness or level of change. As an example, a small volume, that is camera-locked to the client might be updated at a high rate, while a more coarse volume, capturing global lighting could be updated more slowly.

A more sophisticated approach would entail interleaving probes from each of the volumes described above and combining both such volumes into a single transport based on how much they have changed and how long it has been since their last update for the client. This approach allows for better utilization of the bandwidth between the client and server to reduce latency for sudden, dramatic scene changes near a viewpoint (such as explosions), while deprioritizing large-scale global changes (e.g. sun motion) which are perceptually less latency sensitive.

Latency of Encoding. Video codec B-frames are relative to both past and future P-frames. They can improve HEVC compression performance by 25% [44]. We prevented the encoder from using B-frames because they inherently add multiple P-frames of delay, in order to look into the future. However, if the available network throughput is low and transporting a frame with less compression takes longer than multiple smaller frames, then B-frames may be beneficial. Furthermore, in scenarios where future game states are predictable, such as architectural walk-throughs or games with slow-moving or deterministic objects, the GI could be made available without delay, or even with look-ahead.

High Frequency Glossy Surfaces. Our system handles GI on rough glossy and diffuse surfaces by sending frequency-limited incident lighting data, and does so with correct occlusion. This is an improvement over previous systems limited to only diffuse or direct illumination that did not model dynamic occlusion. However, our approach does not simulate high-frequency glossy effects for smooth glossy surfaces and mirror reflections as the angular resolution of the probes is too low. Increasing the probe resolution directly would compromise their efficiency.

6 CONCLUSION

Distributed graphics is on the rise. The questions of how to distribute the work for interactive systems and where latency is a key consideration are core to its realization. Decoupling clients, pixels, and frames allows better scaling, but introduces system complexity. Early adopters in the distributed interactive graphics market have wisely chosen to use brute force remote rendering and video streaming to minimize complexity for first-generation systems. For the coming generations, more sophisticated approaches with elements such as the ones we have explored may reduce latency and improve end-to-end performance. Shifting the costs of power and computation between client and server and reducing the costs on both ends while also improving quality by amortizing work will be the long-term key to creating economically viable and sustainable distributed graphics systems.

REFERENCES

[1] David Bierton. Bierton. Face-Off: Gaikai vs. OnLive. https://www.eurogamer.net/articles/digitalfoundry-face-off-gaikai-vs-onlive

[2] 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 Transactions on Graphics (TOG)* 39, 4 (2020), 86–1.

[3] Keith Bugeja, Kurt Debattista, and Sandro Spina. 2019. An asynchronous method for cloud-based rendering. *The Visual Computer* 35, 12 (2019).

[4] Wei Cai, Min Chen, and Victor CM Leung. 2014. Toward gaming as a service. *IEEE Internet Computing* 18, 3 (2014), 12–18.

[5] De-Yu Chen and Magda El-Zarki. 2019. A Framework for Adaptive Residual Streaming for Single-Player Cloud Gaming. *ACM Trans. Multimedia Comput. Commun. Appl.* 15, 2s, Article 66 (July 2019), 23 pages. https://doi.org/10.1145/3336498

[6] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. 2012. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*. IEEE, 1–6.

[7] Yann Collet et al. 2013. Lz4: Extremely fast compression algorithm. *code. google. com* (2013).

[8] 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. *JCGT* 4, 4 (2015). http://jcgt.org/published/0004/04/01/

[9] Paul Debevec and Erik Reinhard. 2006. Session Details: High-Dynamic-Range Imaging: Theory and Applications. In *SIGGRAPH Courses* (Boston, Massachusetts). New York, NY, USA, 1. https://doi.org/10.1145/3245638

[10] Andrea Di Domenico, Gianluca Perna, Martino Trevisan, Luca Vassio, and Danilo Giordano. 2020. A network analysis on cloud gaming: Stadia, GeForce Now and PSNow. *arXiv preprint arXiv:2012.06774* (2020).

[11] ENet. 2020 (accessed April 22, 2020). *ENet Reliable UDP networking library.* http://enet.bespin.org/

[12] flatpanelshd.com. 2019 (accessed April 22, 2020). *The HDR Ecosystem Tracker.* https://www.flatpanelshd.com/focus.php?subaction=showfull&id=1559638820

[13] GPD. 2020 (accessed April 22, 2020). *GPD WIN2.* https://www.gpd.hk/gdpwin2

[14] Gene Greger, Peter Shirley, P.M. Hubbard, and D.P. Greenberg. 1998. The Irradiance Volume. *Computer Graphics and Applications, IEEE* 18 (04 1998), 32–43. https://doi.org/10.1109/38.656788

[15] LLC HDR10+ Technologies. 2019 (accessed April 22, 2020). *HDR10+ System Whitepaper.* https://hdr10plus.org/wp-content/uploads/2019/08/HDR10_WhitePaper.pdf

[16] Karl E Hillesland and JC Yang. 2016. Texel shading. In *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics: Short Papers*. 73–76.

[17] Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. 2019. Tessellated Shading Streaming. In *Computer Graphics Forum*, Vol. 38. Wiley Online Library, 171–182.

[18] Petr Holub, Martin Šrom, Martin Pulec, Jiří Matela, and Martin Jirman. 2013. GPU-accelerated DXT and JPEG compression schemes for low-latency network transmissions of HD, 2K, and 4K video. *Future Generation Computer Systems* 29, 8 (2013), 1991–2006.

[19] Gazi Karam Illahi, Thomas Van Gemert, Matti Siekkinen, Enrico Masala, Antti Oulasvirta, and Antti Ylä-Jääski. 2020. Cloud Gaming with Foveated Video Encoding. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 16, 1 (2020), 1–24.

[20] Teemu Kämäräinen, Matti Siekkinen, Jukka Eerikäinen, and Antti Ylä-Jääski. 2018. CloudVR: Cloud accelerated interactive mobile virtual reality. In *Proceedings of the 26th ACM international conference on Multimedia*. 1181–1189.

[21] Peter D Kirchner, James T Klosowski, Peter Hochschild, and Richard Swetz. 2003. Scalable visualization using a network-attached video framebuffer. *Computers & Graphics* 27, 5 (2003), 669–680.

[22] Marc Levoy. 1995. Polygon-Assisted JPEG and MPEG Compression of Synthetic Images. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '95)*. Association for Computing Machinery, New York, NY, USA, 21–28. https://doi.org/10.1145/218380.218392

[23] Peter Lindstrom. 2014. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2674–2683.

[24] Chang Liu, Jinyuan Jia, Qian Zhang, and Lei Zhao. 2017. Lightweight websim rendering framework based on cloud-baking. In *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. 221–229.

[25] Chang Liu, Wei Tsang Ooi, Jinyuan Jia, and Lei Zhao. 2018. Cloud Baking: Collaborative Scene Illumination for Dynamic Web3D Scenes. *ACM Trans. Multimedia Comput. Commun. Appl.* 14, 3s, Article 59 (June 2018), 20 pages. https://doi.org/10.1145/3206431

[26] Yunpeng Liu, Stephan Beck, Renfang Wang, Jin Li, Huixia Xu, Shijie Yao, Xiaopeng Tong, and Bernd Froehlich. 2015. Hybrid lossless-lossy compression for real-time depth-sensor streams in 3D telepresence applications. In *Pacific Rim Conference on Multimedia*. Springer, 442–452.

[27] Mark Magro, Keith Bugeja, Sandro Spina, and Kurt Debattista. 2019. Interactive Cloud-based Global Illumination for Shared Virtual Environments. In *2019 11th International Conference on Virtual Worlds and Games for Serious Applications (VS-Games)*. IEEE, 1–8.

[28] 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/

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

[30] Thomas Morgan. 2019. Doom Eternal on Stadia looks great - but the lag is just too high. https://www.eurogamer.net/articles/digitalfoundry-2020-doom-eternal-stadia-looks-the-part-but-lag-is-too-high

[31] 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

[32] NVIDIA. 2019 (accessed April 22, 2020). *Video Encode and Decode GPU Support Matrix.* https://developer.nvidia.com/video-encode-decode-gpu-support-matrix

[33] NVIDIA. 2020 (accessed June 26, 2020). *NVIDIA Xavier NX.* https://developer.nvidia.com/embedded/jetson-xavier-nx

[34] Jorn Nystad, Anders Lassen, Andy Pomianowski, Sean Ellis, and Tom Olson. 2012. Adaptive scalable texture compression. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Eurographics Association, 105–114.

[35] Dawid Pajak, Robert Herzog, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel. 2011. Scalable remote rendering with depth and motion-flow augmented streaming. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 415–424.

[36] Qualcomm. 2019 (accessed April 22, 2020). *Snapdragon 865 5G Mobile Platform.* https://www.qualcomm.com/products/snapdragon-865-5g-mobile-platform

[37] Tristan Richardson and John Levine. 2011. The remote framebuffer protocol. *IETF RFC 6143* (2011).

[38] Daniel Scherzer, Chuong H. Nguyen, Tobias Ritschel, and Hans-Peter Seidel. 2012. Pre-Convolved Radiance Caching. *Comput. Graph. Forum* 31, 4 (June 2012), 1391–1397. https://doi.org/10.1111/j.1467-8659.2012.03134.x

[39] Christophe Schlick. 1994. An inexpensive BRDF model for physically-based rendering. In *Computer graphics forum*, Vol. 13. Wiley Online Library, 233–246.

[40] Sebastian Schwarz, Marius Preda, Vittorio Baroncini, Madhukar Budagavi, Pablo Cesar, Philip A Chou, Robert A Cohen, Maja Krivokuća, Sebastien Lasserre, Zhu Li, et al. 2018. Emerging MPEG standards for point cloud compression. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 1 (2018), 133–148.

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

[42] Ivan Slivar, Mirko Suznjevic, Lea Skorin-Kapov, and Maja Matijasevic. 2014. Empirical QoE study of in-home streaming of online games. In *2014 13th Annual Workshop on Network and Systems Support for Games*. IEEE, 1–6.

[43] stereolabs.com. 2019 (accessed April 22, 2020). *H.264/H.265 Video Encoding Support Matrix for Nvidia Jetson*. https://www.stereolabs.com/blog/h-264-h-265-video-encoding-support-matrix-for-nvidia-jetson/

[44] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. 2012. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on circuits and systems for video technology* 22, 12 (2012), 1649–1668.

[45] Alexander Suvorov. 2020 (accessed April 22, 2020). *Crunch compression of ETC textures*. https://blogs.unity3d.com/2017/12/15/crunch-compression-of-etc-textures/

[46] Niraj Tolia, David G Andersen, and Mahadev Satyanarayanan. 2006. Quantifying interactive user experience on thin clients. *Computer* 39, 3 (2006), 46–52.

[47] JMP Van Waveren and Ignacio Castaño. 2007. Real-time YCoCg-DXT compression. *Tech. Rep., id Software, Inc. and NVIDIA Corp.* 2 (2007), 3.

[48] Maarten Wijnants, Hendrik Lievens, Nick Michiels, Jeroen Put, Peter Quax, and Wim Lamotte. 2018. Standards-compliant HTTP adaptive streaming of static light fields. In *Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology*. 1–12.

[49] Andrew D Wilson. 2017. Fast lossless depth image compression. In *Proceedings of the 2017 ACM International Conference on Interactive Surfaces and Spaces*. 100–105.

[50] Nathan Andrew Zabriskie. 2018. NetLight: Cloud Baked Indirect Illumination.