Multimedia Tools and Applications (2026) 85:581
https://doi.org/10.1007/s11042-026-21733-5

Context-aware point cloud streaming for high-fidelity holographic telepresence

Vasileios Ektor Kotsis-Panakakis[1] · Iordanis Evangelou[1] · Fotios Bistas[1] · Andreas A. Vasilakis[1] · Georgios Papaioannou[1] · Anastasios Gkaravelis[2] · Nikolaos Vitsas[2]

Received: 22 February 2026 / Revised: 3 June 2026 / Accepted: 10 June 2026

Abstract
Real-time holographic communication enables immersive remote interaction by transmitting dynamic three-dimensional representations of users and environments, providing a stronger sense of presence than conventional video conferencing. However, the high data throughput requirements of volumetric streaming introduce challenges related to bandwidth variability and heterogeneous device capabilities. This paper presents an end-to-end system for scalable, real-time holographic telepresence, designed to operate within a fixed volumetric data transmission budget. The proposed approach combines importance-driven and semantic-aware filtering to preserve perceptually and interaction-critical details, while degrading less relevant data. We also leverage gesture recognition to automatically identify and emphasize points of interest, enhancing interactivity and realism. Experimental results demonstrate high visual fidelity and responsive performance with low bandwidth requirements, enabling efficient holographic communication for diverse use cases across different platforms.

Keywords Adaptive sampling · 3D compression · Holographic communication · Telepresence · Point cloud streaming · Volumetric video

1 Introduction

The rapid evolution of Extended Reality (XR) applications and the emerging Metaverse paradigm are fundamentally transforming how remote interaction is facilitated between users [1]. Although traditional video conferencing has successfully bridged geographical distances, it remains confined to two-dimensional planar displays. These 2D media often fail to generate the critical geometric cues essential for genuine human connection and understanding [2]. To address these limitations, real-time holographic communication or volumetric video explicitly captures and streams dynamic, three-dimensional representa-

Vasileios Ektor Kotsis-Panakakis and Iordanis Evangelou contributed equally to this work.

Extended author information available on the last page of the article

Published online: 29 June 2026

tions of content as unordered point clouds [3]. By enabling remote participants to view subjects from any angle with Six Degrees of Freedom (6DoF), this technology enforces a sense of "co-presence" that 2D video cannot replicate.

However, the transition from planar video to volumetric streaming is non-trivial and introduces significant technical constraints. The most prominent challenge lies in the dynamic point clouds generated by commodity depth sensors or multi-camera rigs. These datasets are inherently data-intensive. For instance, an uncompressed stream with just 200K points per frame (15 bytes/point) at 30 FPS requires approximately 720 Mbps connection, assuming no other bilateral overheads. Scaling this to high-fidelity human representations creates a bottleneck that standard consumer networks cannot handle without aggressive optimization. Consequently, transmitting raw or uniformly compressed volumetric data over fluctuating networks often results in high latencies, frame drops, or geometric artifacts, all of which severely degrade the Quality of Experience [4].

Existing solutions largely treat volumetric data in a holistic, content-agnostic manner. These methods typically rely on general-purpose point cloud compression schemes, such as MPEG-V-PCC and G-PCC [5] or CWI-PC [6], public point cloud libraries [7–9], or network-protocol-specific designs [10, 11] (detailed in Section 3). While effective at reducing overall data size, these approaches are insufficient for the interactive tasks presented in this work, including real-time content monitoring, 6DoF manipulation of virtual objects, and human holographic telepresence. Being content-agnostic implies assigning equal importance to all generated content, inevitably wasting resources on points irrelevant to the context. By ignoring the visual attention that the content producer assigns, and uniformly degrading quality across the entire volume, these methods fail to preserve semantically important regions.

To address these challenges, this work presents a Holographic Optimized Processing Engine: an end-to-end system for scalable, real-time holographic telepresence designed for constrained networks and hardware-limited devices (see Fig. 1). Existing approaches suffer from two major limitations. First, they apply static compression parameters uniformly across the entire input. Second, they fail to exploit the massive parallelism offered by modern GPUs on either the server or client side. In this work, we integrate context-awareness directly into the streaming pipeline. On the server side, we fully leverage the GPU parallel execution model using CUDA to enable real-time point cloud preparation (see Section 4.1). On the client side, to achieve zero-copy rendering, we transmit the content in an optimized memory layout (see Section 4.2) that allows direct consumption by the parallel rasterization pipeline. Furthermore, leveraging real-time semantic segmentation provided by state-of-the-art foundation neural models and gesture recognition, the server dynamically identifies

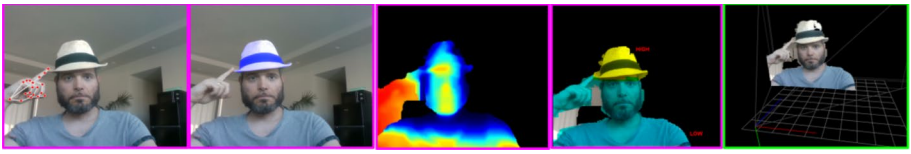

Fig. 1 Producer–consumer pipeline for robust, realtime context-aware holographic streaming. RGB-D data from a single depth camera is processed through gesture-guided region prioritization, segmentation, background removal, and adaptive compression (left, columns 1-4), before low-latency GPU-accelerated transmission and real-time XR rendering (right). Server-side operations are highlighted with pink color while client-side with green color

Regions of Interest (ROI, see Section 4.1.3) and compresses the output stream based on perceptual importance (see Section 4.1.4). This adaptive architecture prioritizes critical regions while aggressively compressing or filtering less relevant data (see Section 4.1.4), ensuring that the communication intent remains consistent even under bandwidth limitations (see Section 5).

This user-driven approach is adopted for three key reasons. First, it guarantees that critical visual details, such as task-relevant object features, are consistently preserved. Second, it naturally adapts to dynamic conversational and collaborative contexts, in which points of interest may shift rapidly over time. Finally, by relying on automated inference, the method mitigates the risk of incorrect semantic assumptions, thereby ensuring that the content presented to the consumer remains relevant and accurate. The specific contributions of this work can be summarized as follows:

- We propose an adaptive streaming architecture that prioritizes perceptually critical regions. By applying aggressive filtering or/and compression to non-essential data while preserving high fidelity in the RoI, we can optimize the requested point cloud budget for constrained networks.
- We introduce a mechanism that leverages state-of-the-art foundation neural models and gesture recognition to dynamically identify semantic RoI within the volumetric scene, distinguishing between critical features and irrelevant background noise.
- We demonstrate how to take full advantage of available hardware for highly parallel execution of point cloud compression on the server side and decompression on the client side.
- The system is designed with broad interoperability in mind, allowing the rendered 3D stream to be visualized on various hardware-limited devices via standard WebXR interfaces, removing the need for specialized, high-end viewing hardware.
- We validate our approach across multiple distinct interaction scenarios. Furthermore, we provide the datasets and results publicly to foster future research in context-aware volumetric streaming.

Note that this paper extends our previous poster work presented at EuroXR 2025 [12], building upon the initial concepts of importance-aware volumetric transmission to address similar holographic tasks. In this work we transform both the server and the client side architecture in order to take full advantage of the underlying GPU capabilities and at the same time improving the overall visual quality.

2 Background work

To contextualize the proposed system, it is first necessary to provide a brief overview to the taxonomy related to volumetric data streaming, the current standards for compression, and the protocols used for transmission. For each subcategory we also briefly justify our design choices. It is worth noting that there is considerable work in the literature that studies this class of problems from diverse perspective always adapted to the intended task that is targeted for. In this section we provide a high level overview but the interested reader may refer to several surveys in the field, that discuss these subjects in detail [3, 4, 13].

2.1 3D video streaming

Interactive telepresence applications cover a wide range of real-time scenarios, from immersive one-to-one communication to multi-party sessions distributed across multiple locations [4]. Among these, *one-to-many* interactions, in which a single producer streams real-time volumetric content to multiple consumers, support a range of use cases that are not easily addressed by traditional one-to-one systems. Examples include live lectures or presentations, remote collaborative design reviews, broadcasting immersive performances, and multi-participant training or education sessions. These scenarios require systems that can efficiently handle scalable content distribution, low-latency rendering, and consistent visual quality across multiple endpoints, while still allowing interactivity and responsiveness for the audience. Many systems adopt a server–client architecture, where the server is responsible for capturing, processing, compressing, and distributing holographic or 3D content, while clients handle reception, decoding, and rendering in real time. On the server side, modern implementations leverage parallelism to accelerate tasks such as point cloud processing, mesh reconstruction, and compression. Clients are typically designed to support low-latency rendering pipelines.

In this work, we focus specifically on one-to-many interactions (see Section 4), targeting similar scenarios such as live teleconference, remote object manipulation review, and online participant monitoring, which provide representative use cases for evaluating the performance and scalability of our volumentric streaming system (see Section 5).

2.2 Volumetric data representations

2.2.1 Explicit and Implicit Learned Representations

Learned representations utilize pre-captured imagery to synthesize novel views of an environment. Most common frameworks are based on implicit representations such as Neural Radiance fields [14] or, explicit ones such as Gaussian Splatting [15]. While in theory any camera can be used, high-fidelity setups typically employ arrays of cameras to overcome the limited viewing angles and restrictive positional freedom associated with single-lens solutions. Current research largely focuses on capturing static images or video with stationary camera setups, as the complexity increases exponentially with motion and efficient transmission and visualization in the context of teleconference is impractical.

2.2.2 Volumetric Video

In contrast to light fields, represents the scene using three-dimensional geometric objects, most commonly as unordered Point Clouds. A point cloud is a set of data points defined in a 3D coordinate system. Beyond spatial position, each point can carry attributes such as color (three-channel), material properties, transparency, or surface normals. Point clouds share similarities with 3D meshes but offer distinct advantages for telepresence. First, topology independence which unlike meshes, that rely on triangles, point clouds are unstructured. This provides greater flexibility for culling and tiling, as the data is not restricted by connectivity. Secondly, texture mapping is simplified to a one-to-one mapping between a point's position and its color attribute, avoiding the complex UV unwrapping required for meshes

[16]. While meshes allow for specific graphics pipeline optimizations, point clouds are often the native output of capture devices, reducing the need for computationally expensive conversion steps. This approach significantly reduces the computational overhead to generate since existing hardware can exploit depth maps. Moreover, transmission bandwidth required for the client-side communication compared to light fields is considerable lower (i.e. less attributes are required for transmission and visualization) and can largely exploit existing network protocols.

The generation of point clouds is generally achieved through two primary paradigms: (i) active sensing, which involves emitting signals to measure range, and (ii) passive reconstruction, which derives 3D structure from optical images using computational algorithms or statistical methods.

One prominent active techniques is Light Detection and Ranging (LiDAR). LiDAR sensors emit pulsed laser light toward a target and measure the reflected pulses. By calculating the Time of Flight (ToF); the precise duration required for the light to travel to the object and return, the sensor determines the distance to the surface. Because LiDAR supplies its own illumination source, it remains effective in low-light conditions, making it the industry standard for autonomous driving and large-scale topographic mapping, though it can be cost-prohibitive for smaller applications.

Alternatively, for close-range interactions such as augmented reality, RGB-D depth sensors act as a cost-effective active sensing solution. These devices typically employ Structured Light or continuous-wave ToF technologies. In structured light approaches, an infrared (IR) projector casts a known pattern onto the scene. An IR camera captures the distortion of this pattern on 3D surfaces, and the system triangulates the depth for every pixel. This results in a depth map that can be back-projected into 3D space to form a point cloud. While highly accurate indoors, these sensors often struggle in direct sunlight due to infrared interference.

Moving beyond hardware-centric approaches, passive reconstruction leverages standard 2D imagery processed through advanced algorithms. Historically, this has been achieved through Structure from Motion (SfM) [17] and Multi-View Stereo (MVS) [18] algorithms, where features are matched across multiple overlapping photographs taken from different angles to triangulate 3D positions. Recently, neural-based approaches, enabled the generation of point clouds from even a single image [19, 20]. Deep foundation models can learn contextual cues, such as perspective, occlusion, and object scale, from vast datasets to predict pixel-wise depth.

In this work, we exploit active sensing using an Intel RealSense camera [21] to generate the depth maps which are then unprojected to world coordinates to form the initial point cloud. Each point is associated with its own color attribute associated with the pixel value captured by the color sensor and baked on the color buffer. We choose the RGB-D camera setup over some neural depth estimation method since volumetric video must operate strictly within real-time budgets that is, 33-16ms (30-60 frames per second) per frame for the whole pipeline (see Section 4).

2.3 Compression standards

The prohibitive bandwidth requirements of raw point clouds necessitate aggressive compression strategies to make real-time volumetric streaming feasible. The fundamental mechanism employed across most standards is quantization, which limits the precision of

position variables and attributes (such as color) to significantly reduce the bit depth required for each point. Furthermore, tree-based data structures are commonly utilized to allow for lossy compression by limiting tree depth, effectively merging points located deeper in the hierarchy into single voxels at the center of the final level. Current standardization efforts and industry solutions generally fall into geometry-based and video-based approaches, each with distinct algorithmic trade-offs.

Geometry-based Point Cloud Compression (G-PCC) [5] encodes the point cloud directly in 3D space without intermediate projection. The process begins by converting floating-point positions into an integer representation via scaling and translation, enabling the data to be voxelized and structured into an occupancy octree. To optimize efficiency, G-PCC employs entropy encoding for dense areas to exploit spatial redundancy, while isolated points utilize a Direct Coding Mode (DCM) to avoid the overhead of tree structures. For attribute compression, such as color, methods like the Region-Adaptive Hierarchical Transform (RAHT) predict values from lower tree levels, or Lifting schemes assign weights to points based on their level of detail.

In contrast, Video-based Point Cloud Compression (V-PCC) [5] seeks to leverage mature 2D video coding standards. This approach projects 3D point cloud data into 2D patches, which are then organized onto a 2D plane using a patch packing algorithm that rotates and fits patches to maximize packing efficiency. These generated 2D maps—representing geometry, occupancy, and texture are then encoded using standard video codecs like HEVC. By exploiting the temporal correlation between frames inherent in video codecs, V-PCC achieves superior compression efficiency compared to G-PCC. However, the computational complexity involved in the projection and patch packing processes introduces significant latency. Therefore, both V-PCC and G-PCC compression standards are often unsuitable for real-time interactive applications like the cases we study in this work. This bottleneck is also confirmed by other similar works [11, 22, 23].

For scenarios prioritizing speed over maximum compression, Google Draco [8] has emerged as a widely adopted standard. Although originally designed for static meshes, Draco utilizes k-d trees rather than octrees and supports point cloud compression through quantization. It allows for fine-grained control over output quality via specific quantization parameters, though this quantization affects precision rather than the total point count. While Draco lacks temporal compression and processes video as a sequence of independent frames, its simplicity results in significantly faster encoding and decoding times. However, at its current state, the library provides only a CPU-based implementation resulting in a highly inefficient tool for time budgets that are strictly bounded by real-time constraints. It is worth noting that, parallelizing this process in the encoding/decoding phase is still insufficient compared to our proposed method as we will demonstrate in Section 5. This limitation, in terms of computational overhead, in the encoding and the decoding side can also be verified by other researchers [11, 24, 25].

In this work, we develop a fast GPU implementation of the quantization scheme used by Draco for both positional and color information using CUDA. Our custom implementation enables the transmission of the encoded stream in a layout optimized for loading into the client-side GPU rasterizer with zero-copy overhead (see Section 5). By performing decoding within the highly parallel execution model of the GPU, we eliminate the need for complex, device-specific multi-threading schedules (e.g., for mobile or XR devices). This guarantees that the decoding is fully accelerated by the hardware's rasterizer during the

rendering phase. As a result, this makes our approach faster by design on the client side and helps bridge the gap towards real-time teleconferencing.

3 Related work

Volumetric data streaming has been approached from different perspectives over the years. Different protocols has been proposed to mitigate for the intense data generation and transmission over the network layer. Balancing the trade-off between bandwidth economy and latency can be a tedious task and often custom solutions, based on the intended task, are proposed.

For mobile environments where resources are strictly limited, Han et al. [26] proposed ViVo, a visibility-aware streaming system. ViVo introduces three key optimizations: Viewport Visibility (VV), which discards data outside the user's field of view; Occlusion Visibility (OV), which reduces density for occluded points; and Distance Visibility (DV), which adjusts quality based on distance from the camera. The system achieves significant data savings while maintaining high structural similarity, making it highly effective for 5G mobile networks.

Complementing this, Lee et al. [24] proposed GROOT, a real-time streaming system specifically designed for high-fidelity volumetric videos on mobile devices. GROOT addresses the computational bottleneck of decoding by introducing a PD-Tree (Parallel Decoding Tree) structure, which allows for parallelized decoding on mobile GPUs. Furthermore, it employs a region-based adaptive streaming strategy that partitions the volume into independent blocks, allowing the client to selectively fetch and decode only the visible regions. Experimental results demonstrate that GROOT can sustain high frame rate streaming with considerably lower latency than standard single-threaded decoding approaches, effectively enabling high-fidelity renderings on commodity smartphones.

To manage bandwidth variability, a number of works have adapted the Dynamic Adaptive Streaming over HTTP (DASH) [27] standard for volumetric content. Heidarirad and Wang [28] introduced VV-DASH, an end-to-end framework that includes a codec-agnostic DASH Volumetric Video segment format. This format consolidates compressed data into DASH-ready segments, enabling standard bitrate adaptation algorithms to function effectively. Similarly, Hosseini and Timmerer [10] proposed DASH-PC, a system that spatially sub-samples point clouds into multiple quality representations. By defining a specialized Media Presentation Description for 3D data, DASH-PC allows clients to request different densities based on available bandwidth and user distance, significantly reducing bandwidth usage while aligning data density with human visual acuity.

Efficient volumetric data transmissions are also studied in the network protocol level. To address the high bandwidth and low-latency requirements of immersive communication, De Fré et al. [11], proposed a volumetric video delivery pipeline leveraging Low Latency, Low Loss, and Scalable Throughput (L4S) standards. Their architecture utilizes Accurate Explicit Congestion Notification (AccECN) and packet pacing to proactively manage network congestion before packet loss occurs. When compared to a traditional WebRTC implementation using Google Congestion Control (GCC), the L4S-based approach demonstrated a faster bandwidth convergence time and maintained consistent low latency even in the presence of competing TCP flows. Furthermore, the L4S pipeline eliminated packet loss

entirely, whereas the WebRTC approach resulted in of frames being undecodable due to congestion induced loss.

Finally, Thomas and Xylomenos [25] investigated the feasibility of ultra-low latency volumetric streaming over 5G Stand Alone (SA) networks. They developed a custom UDP-based streaming tool utilizing the Draco codec to analyze the trade-offs between compression efficiency, processing latency, and network throughput. By combining multithreaded encoding with a reduction in point cloud resolution, the authors achieved consistent generation close to interactive frame rates.

In contrast to the methods described above, this work focuses on scenarios where the server must broadcast a specific RoI to connected clients. The RoI is determined by the server or equivalently the client. This is orthogonal to other existing works that exploit other kinds of spatial cues (e.g., distance of the client viewer [29]). This requirement is common in remote applications where practitioners need to inspect specific items in detail or in teleconference or monitoring where the user body or head, is most of importance. Once the presenter indicates the RoI, the system automatically defines the context. Subsequently, the associated unprojected point cloud is broadcast to clients at the maximum allowed quality, while the complementary set is compressed into a more approximate representation. This allows us to apply the proposed encoding schemes to the resulting subsets effectively.

4 Methodology

We present a pipeline optimized for context-aware teleconferencing that operates on a server-client (one-to-many) architecture (see Section 2.1). A host server transmits content to connected users (clients) for visualization. Clients may connect via any device capable of standard rasterization, including mobile devices, VR headsets, or web browsers. The server dynamically manages the visual quality of specific regions based on transmission requirements. Conversely, clients operate passively, they listen to the transmitted stream and propagate it to the rendering engine. This design eliminates the need for bi-directional communication during streaming, thereby minimizing latencies induced by network packet exchange.

At a high level, the pipeline comprises two primary components. The server-side component, detailed in Section 4.1, acquires point clouds and associated color attributes using a depth sensing camera. This data is selectively clustered to perform simplification and compression derived from the requested context. The processed content is then broadcast to all connected clients, which are solely responsible for decoding and visualizing the input stream, as discussed in Section 4.2. A complete overview of the proposed architecture is depicted in Fig. 2.

4.1 Server-side: 3D data generation & processing

On the server side, the "Producer" (see Fig. 2) operates as a multiprocessing pipeline architected to maximize hardware resource utilization. The main process is exclusively dedicated to handling capture metadata from the camera (see Section 4.1.1). Concurrently, separate processes execute computational tasks across both CPU and GPU resources. These include: i) a process managing hand and gesture tracking (see Section 4.1.2), ii) a process handling

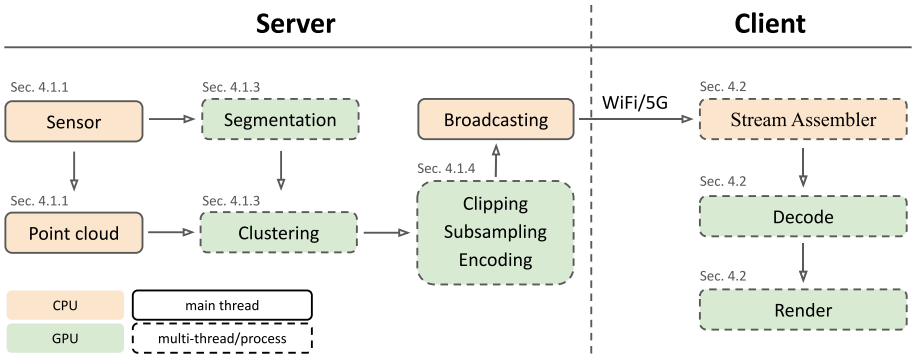

Fig. 2 The proposed pipeline consists of server-side modules (left) and client-side modules (right). On the server side, point clouds generated by the sensor are processed via collaborative multi-processing within the GPU (see Section 4.1). Once all post-processing events are applied, the points are transmitted to all connected clients as a byte stream. The clients leverage the highly optimized WebXR framework to load the stream with zero-copy overhead, decoding them within the parallel architecture of the rasterizer for real-time rendering (see Section 4.2)

the prediction and extraction of the salient region of interest (see Section 4.1.3), and iii) a dedicated process responsible for encoding (see Section 4.1.4). Finally, the main thread handles the broadcasting of each point cloud cluster to the connected clients.

A strict synchronization constraint is enforced during each frame cycle: all clusters must be broadcast before the pipeline advances to the next frame. Conversely, non-critical operations are executed asynchronously, their results are integrated into the pipeline on the fly as they become available, allowing for updates that may span across iteration boundaries. This workload distribution is designed to maximize parallelism and hide latency, ensuring near-synchronization of all processes. In our experimental setup (see Section 5), the main loop maintains real-time performance with a 33ms work cycle.

4.1.1 Point cloud generation

The server captures the scene using an Intel RealSense D435i sensor [21], configured for `rgb8` color and `z16` depth stream formats. Specific camera configurations for each case study are detailed in Section 5. To maintain plausible visuals, all evaluations are conducted at 30 frames per second. This yields approximately 407K points (i.e., default resolution of 848×480) per frame for a full scene reconstruction. Since each point stores three color values (1 byte each) and three spatial coordinates (4 bytes each), the resulting data payload is approximately 6 MB per frame. Even with a network bandwidth of 200 Mbps (symmetric upload/download), transmitting a single uncompressed frame would incur a latency of approximately 244 ms (~ 4.1 fps) far exceeding the threshold for real-time interaction. To achieve a smooth client-side frame rate of 30 fps, the transmission and processing window must be restricted to approximately at most 33 ms. Meeting this constraint requires a frame size of no more than 825 KB, equivalent to approximately 55K points with attributes. Consequently, direct transmission is infeasible, necessitating a fixed point budget and a robust optimization mechanism involving *clipping*, *subsampling*, and *encoding* (see Section 4.1.4).

To address this, the point cloud extracted from each frame undergoes pre-processing before transmission. To maximize client-side visual fidelity, we identify three regions of interest in image space. These regions are adaptively selected to simplify the geometry, followed by encoding to minimize the transmission payload. The regions are defined as *high-*, *medium-*, and *low-quality*, respectively, ensuring a smooth visual transition from the highest quality region to the lowest. We detail the region determination process in Section 4.1.3 and the quality characteristics of each cluster in Section 4.1.4.

Simplification and encoding are the only tasks treated as blocking overhead within the server's main loop. Other processes, such as gesture tracking and segmentation, execute asynchronously. These auxiliary processes contribute no computational overhead beyond negligible shared-memory transactions via the CUDA *interprocess communication system* [30, refer to "Section 4.15. Interprocess Communication"]. Thus, simplification and encoding constitute the only operations that must complete prior to invoking each broadcasting event. Once all three broadcasting events are complete, the system advances to the next frame generation event.

4.1.2 Context extraction

To enable user-driven selection of RoI (either server or client side) for high-quality transmission, we employ a hand-tracking algorithm provided by mediaPipe library [31]. This module detects the pose of either the left or right hand and extracts a set of skeletal markers corresponding to joint locations. By applying geometric constraints based on the distances and angles between these markers, we identify pointing gestures and map the indicated region in image space to the generated point cloud.

Technically, the gesture recognition relies on pixel-space coordinates of specific joint landmarks (see Fig. 3). To determine if the index finger is extended in a pointing configuration, the algorithm verifies the collinearity of three consecutive markers: the metacarpophalangeal (MCP), distal interphalangeal, and the fingertip (see [31] for further details about the marker labels). Simultaneously, to confirm that the remaining fingers form a fist, we verify that the Euclidean distance from each non-index fingertip to the wrist marker is less than the distance from the index finger's MCP to the wrist. This tracking module executes asynchronously, introducing negligible overhead to the main processing loop. If the indicated region in images is temporarily stable for a sufficient amount of consecutive frames ($t_{ges} = 15$ in our case) we prompt that region to the segmenter module, discussed in the following section. The gesture module runs on a separate thread, other than that of the main loop of the server and the segmenter. As soon as the prompt is available, it is forwarded to the seg-

Fig. 3 Demonstration of the gesture tracking module, which accurately identifies index finger pointing across various directions and hand orientations. If a gesture remains spatially stable over consecutive frames, the module extracts pixel-space coordinates to generate a spatial prompt (see Section 4.1.2), which is then asynchronously forwarded to the segmentation module to define the Region of Interest (see Section 4.1.3)

mentation thread. Running the gesture module is an extremely lightweight operation that requires roughly 3 ms per frame and is anyway executed in parallel to the other server tasks.

4.1.3 Frame segmentation

We implement two distinct methods to extract the user-selected RoI for high and medium quality cluster encoding. Both methods leverage deep neural networks. Specifically, a variant of SAM2 [32] adapted for live video feed and YOLO11e [33] are used. Each network accepts the current RGB frame and a spatial prompt derived from the hand-tracking algorithm as input, producing a binary segmentation mask of the target object in image space. This mask indicates the cluster which must be transmitted in high quality. To derive the image space region of the medium quality cluster we mark the region along the silhouette of the predicted mask. We demonstrate an example case in Fig. 6. Efficiently calculating the region of the medium quality cluster can be done in GPU-level using fast tensor operations. Specifically, we apply on the model's predicted mask a max-pooling operation with fixed window size equal to 32 and stride 1. Then, we upsample the resulting image to the original resolution using the nearest filter and mask the regions that are non-zero on the newly created mask and zero in the initial. We choose the pooling window to be sufficiently large in order to retain spatial details around the region of interest. All remaining points are categorized as background and processed using the lower-fidelity configuration.

The two neural predictors offer distinct trade-offs regarding flexibility and computational cost:

YOLO11e This method operates on a predefined set of detectable objects or classes (see Fig. 4, column 1). It can be configured to recognize specific items (e.g., a "hat" or a "person") based on textual descriptions or visual prompts (e.g., a bounding box). This approach is highly efficient, with inference times ranging from approximately 20-30 ms per frame, depending on the model size. This makes YOLO11e ideal for applications requiring high-speed detection of known object categories but with less accurate mask generation.

SAM2 In contrast, this method offers zero-shot generalization. It can detect and track any coherent object indicated by the prompt (e.g., a specific pixel coordinate), without being constrained to a fixed vocabulary of object categories (see Fig. 4 (columns 2-4)). However, this flexibility incurs a higher computational cost. The inference phase requires approximately 40-70 ms per frame. Ultimately, the choice of predictor depends on available hardware resources and application requirements.

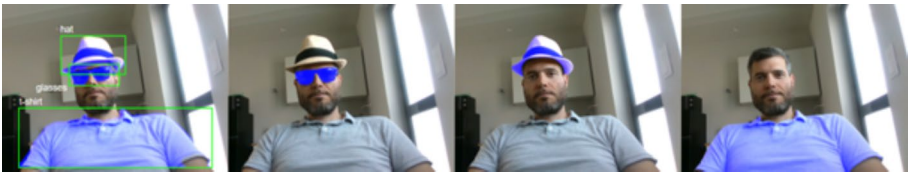

Fig. 4 Comparison of segmentation predictors used to isolate the high-quality ROI (see Section 4.1.3). YOLO11e (column 1) relies on predefined text labels for near real-time inference (20-30 ms/frame), making it ideal for known object categories. In contrast, SAM2 (columns 2-4) utilizes the tracking mechanism detailed in Fig. 3 to compute a zero-shot, fine-grained ROI mask for arbitrary objects, trading higher computational cost (40-70 ms/frame) for greater flexibility and precision

YOLO11e is preferable for scenarios involving predefined targets where low latency is critical, while SAM2 is superior for tasks requiring granular detail and the ability to select arbitrary objects "in the wild".

4.1.4 Point cloud processing

The server partitions the point cloud into three distinct subsets: the **high quality cluster**, containing points within the user-selected RoI; the **medium quality cluster**, containing points in the transition zone spatially adjacent to the high quality cluster; and the **low quality cluster**, comprising the remaining points. We apply a three-way partition to balance visual fidelity and bandwidth. The primary region is transmitted at the highest quality, while the background is kept at a lower quality to save bandwidth. The third intermediate zone is only present to soften abrupt transitions between the two primary clusters, often caused by camera flickering or spatial offset of the segmentation masks from differing frame rates (see Section 4.1.3).

Subsampling To adhere to a fixed transmission budget, we first determine the final number of points allocated to each cluster. We follow a cascading process, prioritizing the most important clusters when distributing the available budget. If a cluster's population exceeds its allocation, we reduce it uniformly. To minimize runtime overhead, we do not apply true uniform random sampling [34]. Instead, we deterministically define a linear span over the flattened indices and select points using a fixed stride. It is worth noting that in this work we assume no specific ordering of these indices, targeting arbitrary point clouds, with no specific device-dependent ordering. As discussed in Section 5, we typically have a budget sufficient to transmit $80 - 100$K points; therefore, subsampling is usually only applied to the low-quality cluster and rarely to the medium-quality one. Points in each cluster are then encoded using specific configurations before transmission, as detailed below.

High quality encoding In this configuration, we quantize 3D coordinates to fit within a single 32-bit integer. We allocate 11 bits for the X and Y axes and 10 bits for Z, effectively discretizing space to a 2^{11} (or 2^{10}) resolution. Since the sensor covers an effective range of 4 meters, our quantization yields a precision of $4/2^{11} \approx 0.2$ cm (X, Y) and ≈ 0.4 cm (Z). We do not compress color channels, forwarding the full 1 byte per channel. Consequently, if this cluster contains N points, its size is reduced from $15N$ to $7N$ bytes, a $\sim 53\%$ reduction with negligible error.

Medium quality encoding In this configuration, we apply the same quantization rules for position and color attributes as used in the high-quality setting. However, unlike the high-quality case, we also reduce the point population if the budget is insufficient. To preserve fidelity, we ensure this reduction never drops below 70% of the initial population. This cluster acts as a transition region between high- and low-quality configurations. Its primary goal is to retain maximum detail to compensate for the inherent, temporary instability of real-time camera depth estimations.

Low-quality encoding In this configuration, we quantize all 3D coordinates to 8-bit accuracy, discretizing the space to a 2^8 resolution. This results in a precision of approximately

1.5 cm ($4/2^8$) along each axis. Color channels remain unaffected to maintain visual fidelity. To handle discretization that is lower than the native camera resolution (e.g., 720 pixel resolution compressed into 256 discrete locations), we apply deduplication to the encoded points, reducing the payload size along the X and Y major axes. Furthermore, to preserve depth quality near the camera, we encode inverse depth values ($z = 1/d$). The client-side decoder then performs the reciprocal transformation ($d = 1/z$) to accurately reconstruct the original depth.

We intentionally selected these encoding schemes to enable seamless propagation of incoming buffers with zero-copy overhead in the rasterizer (see Section 4.2). We also demonstrate the layout of our network packages in Fig. 5. Note that the precision cited above represents an upper bound to the error. In reality, errors are significantly lower because clusters rarely span the full spatial extent (e.g., 4 meters).

For completeness, Fig. 6 demonstrates representative cases where we aggressively simplified non-RoI points to highlight the visual results.

Finally, while texture encoding schemes exist for color (e.g., H.264), transmission of color buffer/structure stream is not recommended, as WebXR API currently lacks support for certain features considered standard in desktop graphics APIs for random access and data augmentation. The same argument applies to processing arbitrary streams of unordered point clouds, for which we must encode the color channel with methods other than the quantization scheme, as in [24]. The limiting factor that this approach is not feasible in a WebXR environment is that unordered indexing in the vertex shader or, dispatching of a compute shader, is currently not supported. Nevertheless, our architecture remains compatible and adaptable to future iterations.

Clipping Initial depth clipping can be optionally applied to the generated point cloud to reduce bandwidth requirements in specific scenarios, such as live teleconferences. In our setup, we leverage the built-in depth clipping functionality of the Intel RealSense camera and treat the near-far clipping range $[Z_n, Z_f]$ as a tunable hyperparameter. This operation can reduce the point cloud up to $\sim 50\%$ based on the given scenario, while introducing only about 2ms of additional processing time per frame (see Fig. 7). For example, in our telecon-

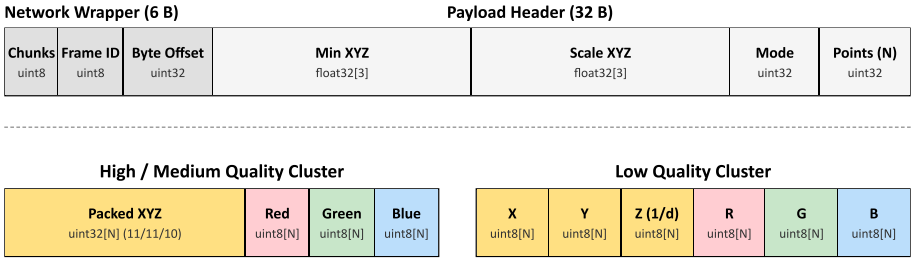

Fig. 5 Layout of the transmitted network packets. Each packet consists of a fixed header (top) and a dynamic data payload (bottom) that varies by quality tier. The header contains a network wrapper for transport including the total chunk count for frame reassembly, a sequential frame ID for synchronization, and a byte offset for direct client-side buffer insertion. It also includes a payload header providing the bounding coordinates and scaling factors required for spatial dequantization, an enumerator specifying the decoding mode, and the total point count (N). The bottom part of the figure illustrates the data payloads, which are packed and padded prior to transmission

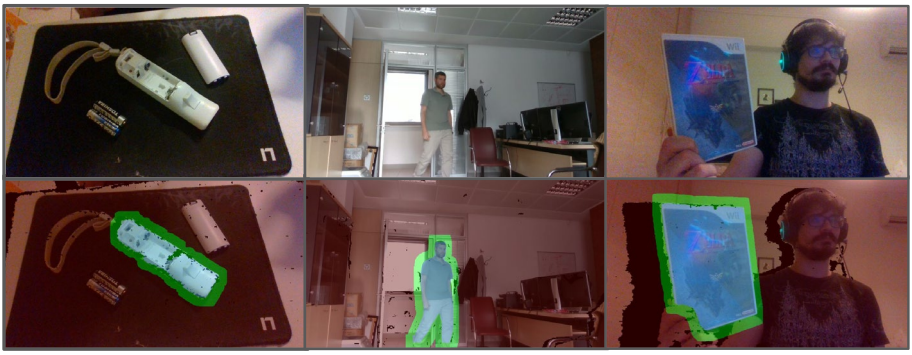

Fig. 6 Indicative clusters generated by our pipeline for the test scenarios described in Section 5. The top row displays the natively generated images, while the bottom row shows the clusters selected by the SAM2/YOLO11 segmentation framework (see Section 4.1.3). In the bottom row, a transparent overlay highlights the generated clusters (see Section 4.1.4): blue indicates high-quality clusters, green indicates medium-quality clusters, and red represents the background, the low quality cluster

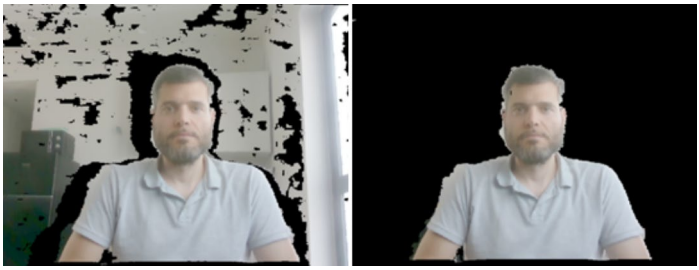

Fig. 7 Impact of the optional Near-Far depth clipping filter (see Section 4.1.4). By discarding geometry outside a defined depth threshold (left), the system can drastically reduce the point cloud volume requiring processing and transmission (right) with minimal computational overhead ($\sim$ 2ms/frame), effectively optimizing bandwidth for certain teleconferencing scenarios

ferencing experiment (see Section 5.1), we discard points closer than $Z_n = 0.01$m or farther than $Z_f = 1$m from the camera.

4.2 Client-side architecture

On the client side, the rendering application listens for broadcast data. Upon establishing a connection, the client performs a handshake to determine the expected number of clusters per frame and the maximum point count for visualization. Our application is compatible with any WebXR capable platform including Meta Quest headsets, mobile devices, and desktop allowing users to access the visualization via a single URL.

To ensure smooth content presentation, we employ a double-buffered architecture. A "frontend" thread collects incoming data chunks into a shared memory buffer, while a "backend" thread simultaneously issues draw calls for the buffer populated in the previous iteration. To maintain consistency, we track the current frame index and cluster type, both

transmitted by the server. The draw call is issued only once all required clusters for the correct frame index are successfully received.

Rasterization and decoding In the rasterization phase, we leverage the predefined memory layout of each cluster (see Section 4.1.3). We map byte buffers directly to typed buffer attributes in the vertex shader (currently supporting 8, 16, and 32-bit integers). We then fetch and decode these values for both position and color directly within the vertex shader.

This approach offers two distinct advantages. First, it ensures zero-copy overhead on the host side, allowing the compiler and hardware to optimize the transaction from CPU to GPU. Second, and most importantly, the decoding logic is executed in parallel within the vertex shader. This is of outmost importance for WebXR environments and stands in sharp contrast to existing approaches (such as the Draco library) that do not explicitly exploit hardware acceleration for decoding. Note that we obtain both optimizations in a platform agnostic manner and at the same time in an optimal manner. We demonstrate the superiority of this method in Section 5.

Finally, to improve visual quality, we implement adaptive point sizing. Specifically, we linearly interpolate the point size between a maximum of 10 pixels and a minimum of 1 pixel based on the viewer's distance. This ensures that distant clusters remain visible while proximal clusters reveal higher detail without occlusion (see Fig. 8). On mid- and high-end clients, we also perform iterative in-painting as a post-processing filter to fill holes.

4.3 Complexity model

4.3.1 Computational cost

To model the end-to-end latency, we must sum the critical path segment timings from server to client. Let N be the total number of points in the input raw point cloud. This is directly dependent on the capture system, which we have no control over. Let also N_{tr} be the subset of points transmitted after sub-sampling, and p be the number of parallel GPU threads.

Fig. 8 Visual comparison of static versus adaptive point sizing on the client-side resterization-based renderer, on the surveillance scenario (see Section 5.3). Rendering point clouds with a static, default setting (point size 1) results in sparse, noisy visuals (left). Applying the proposed linear, depth-based point size interpolation (right) dynamically scales points based on viewer distance, significantly improving visual coherence without occluding proximal details

Server-side latency Because segmentation operates asynchronously on a separate thread (as detailed in Section 4.1.3), it does not block the main frame loop. Therefore, the server's critical path latency is defined as:

$$t_{\text{server}} = t_{\text{camera}}(N) + t_{\text{clusters}}(N) + t_{\text{tr}}(N_{\text{tr}}), \tag{1}$$

where $t_{\text{camera}}(N)$ is the time to fetch the RGB-D buffer using the camera API and has a fixed cost proportional to N ($\mathcal{O}(N)$). The cost $t_{\text{clusters}}(N)$ includes clustering, subsampling, and encoding. Because this is executed in parallel on the GPU at full occupancy, the time to complete this step is proportional to N/p. Finally, $t_{\text{tr}}(N_{\text{tr}})$ is the CPU time to package and dispatch the data to the client. This scales linearly, yielding $\mathcal{O}(N_{\text{tr}})$. Note also that $N >> N_{\text{tr}}$. Therefore, with increasing point cloud density N, processing time also scales linearly in the cumulative computational cost. But due to the GPU-accelerated pipeline, the wall-clock time scales by $\mathcal{O}(N/p)$. As long as N does not exceed the parallel capacity of the GPU amortized over the duration of a single RGB-D frame capture, the impact on t_{server} remains marginal.

Client-side latency Thanks to the zero-copy architecture, decoding and rendering occur entirely in the parallel rasterization pipeline. This enables us to derive the following latency:

$$t_{\text{client}} = t_{\text{prep}}(N_{\text{tr}}) + t_{\text{render}}(N_{\text{tr}}), \tag{2}$$

where, $t_{\text{prep}}(N_{\text{tr}})$ is the time to prepare the rendering buffers with a linear complexity on the input size $\mathcal{O}(N_{\text{tr}})$ and is mostly dominated by the bus capacity to copy the buffers from the host-side to the GPU. $t_{\text{render}}(N_{\text{tr}})$ is the time to dispatch the point rendering commands through the rasterization pipeline, which is generally $\mathcal{O}(N_{tr})$ for primitive dominated workloads, such as unlit shading of input point clouds.

4.3.2 Memory cost

We can also formally define the memory footprint across the pipeline to demonstrate the system's lightweight nature.

Server memory The server must hold the raw point cloud array of size N, the captured image buffers of the primary (or only) with a resolution of $W \times H$ and the subsampled segmentation color image buffer of resolution $\tilde{W} \times \tilde{H}$. For the encoded point cloud, we additionally require the final encoded payload buffer of N_{tr} point records. However, N generally depends on the camera resolution WH, $\tilde{W}\tilde{H}$ is constant, as it constitutes the fixed input to the segmentation neural network and $N >> N_{tr}$, where N_{tr} is adjusted to fit the transmission time budget per frame. Therefore,

$$\mathcal{M}_{\text{server}} = \mathcal{O}(N). \tag{3}$$

Client memory The client only needs to pre-allocate an upper bound of WebXR buffer memory for the received payload, therefore:

$$\mathcal{M}_{\text{client}} = \mathcal{O}(N_{\text{tr}}), \tag{4}$$

which implies that memory cost on the client side scales linearly with the capacity of the transmitted content that is agreed during connection with the server or adjusted during transmission. Note that this upper bound includes buffers that include both positions and colors.

5 Evaluation

We conducted a performance evaluation of the developed holographic telepresence system in an in-house testbed under various scenarios. The goal was to assess the system's ability to process, compress, stream and decode 3D holographic video data in real time using different environments. For this purpose we designed 3 use cases detailed in the following sections.

In order to conduct reproducible experiments, we captured and stored offline all data generated by the RealSense sensor, i.e. the original, uncompressed point cloud and the depth/color buffers. We then load each frame to the server application, apply all operations described in Section 4 and transmit it to the connected clients for visualization. For all our scenarios the sensor is configured to run at 848×480 resolution for both color and depth buffers and 30 fps. The source code for the experiments, the captured and processed datasets are open (see Declarations section).

For the server hardware we use a desktop equipped with an Intel i9-14900K CPU and an NVIDIA RTX 5090 GPU. Our server-side infrastructure is implemented in python and fully exploits CUDA API for all processing steps done to the clusters prior to broadcasting to the clients. For our client devices we select 3 different platforms, a laptop, a mobile and a VR headset. Each device connects to the server via URL to visualize the transmitted content. Our client-side infrastructure is developed in javascript that natively supports WebXR.

For each experiment, we provide both qualitative and a performance analysis of the resulting point cloud. For the first, we have captured the video stream of the rendered point cloud of three point cloud versions:

- **Raw sensor point cloud**. This is the full-detail, uncompressed point cloud without undersampling. Although the bandwidth requirements cannot be met for this dataset, we consider it the ground truth in terms of quality and compare the other two versions against it when measuring perceptual quality.
- **Importance-based optimized point cloud**. This is the version that is produced by our pipeline, including the importance-based segmentation, variable subsampling and application of different quantization levels, as discussed in Section 4. Parameters are tuned to meet the transmission bandwidth requirements and requested frame rate.
- **Uniformly degraded point cloud**. This version is a point cloud matching the same bandwidth requirements as the importance-based version, but here the same quantization and sub-sampling rate has been used for the entire frame. We do not re-use the same subsampling and quantization parameters with any of the clusters of the importance-based case; they are independently computed to maintain comparable frame size with the importance-based version.

The visual quality is evaluated by comparing rendered images of the point cloud, using a relatively simple dedicated visualization methodology: we perform point-based rendering by displaying each sample as a particle with a fixed world-space radius. The written foreground pixels of the frame buffer are then iteratively dilated in a fragment shader to close small gaps using a pair of "ping-pong" temporary swap buffers. The final rendered frames are then stored and passed to the established perceptual difference evaluation method ꟻLIP [35]. This metric is the most appropriate metric for perceptual (w.r.t human eye) measurements since, the task is targeting the teleconference domain. On the contrary, PSNR and SSIM metrics are statistical formulas often evaluated when optimizing subject to them.

For the performance analysis, we provide frame-by-frame measurements for both server- and client-side statistics, alongside the average performance and bandwidth occupancy. In Section 5.4 we demonstrate the performance of our method compared to Draco library use by [25] and [34]. Additionally, we omit network latency measurements, which we have no control of, as our pipeline is primarily optimized for transmission of fully dynamic point cloud streams, with no caching opportunities near the edges. For completeness and simplicity, all our network operations are done using the python (server-side) and JavaScript (client-side) *Web-socket* API in order to guarantee frame ordered delivery.

For the Teleconferencing and Object Manipulation scenarios (Sections 5.1 and 5.2), we employ the SAM2 segmenter, which requires approximately 70 ms per frame. In contrast, the Surveillance case (Section 5.3) utilizes YOLO11e for person detection, taking roughly 30 ms per frame. Both segmentation tasks operate asynchronously on the server. Consequently, a newly generated mask is applied only once it is ready otherwise, the system reuses the mask from the previous frame. Because the input image dimensions remain constant across all scenarios, the processing times are consistent, and we report them as such.

With this in mind, the server-side overhead is effectively reduced to two primary functions. First, the system generates the point cloud from the sensor data and optionally clips the depth extents. This process takes approximately 15 ms given our specific camera configuration. Second, multiprocessing is applied to each cluster to handle deduplication, subsampling, encoding, and broadcasting the content stream. Regarding the computational load of deduplication, subsampling and encoding, 90% of the total reported time is attributed to point quantization. Parallel broadcasting events for all three clusters requires ~ 1 ms. As we will discuss in the following sections, since all clusters are processed in parallel, we maintain a consistent processing cycle in the server-side of less than 10 ms across all test cases.

5.1 Teleconferencing use case

In this experiment, the RGB-D camera's field of view is targeted at a person. The ROI may be the person's face or some other tracked object of interest. The tracked object must be delivered with the best possible quality, whereas other parts of the point cloud are considered less significant for the particular sequence of captured frames. Obviously, tracked objects of generally captured geometry features may change over time, as indicated by the participating user's gestures.

The test sequence consists of 200 frames of point cloud data showing a person holding and interacting with the case of a computer game, which is the tracked object of interest. The reference object was selected by the gesture recognition algorithm and segmented using the SAM2 neural model. For this test case, we consider a visually plausible point cloud

size per frame to be at most 80K so that it can be transmitted over a 11Mbps connection and presented on a laptop device over WiFi at 30fps. The laptop device that visualizes the content is equipped with a CPU Intel Ultra 9, having 32GB of RAM and an NVIDIA RTX 5070 8GB VRAM GPU.

The numerical and visual results of the perceptual evaluation are presented in Fig. 9. The perceptual difference reported by ꟻLIP for the comparison of the importance-based and uniformly degraded versions compared to the original point cloud stream indicates that the error of the uniformly subsampled and quantized output is consistently twice that of the point cloud stream produced by our approach. In this particular example, it is evident that the more careful quantization and sub-sampling of the importance-driven methodology, not only preserves more detail in the region of interest, but also allows for a larger budget of samples to be dedicated to the non-important point clusters (e.g. the background wall here), improving the overall quality of the output.

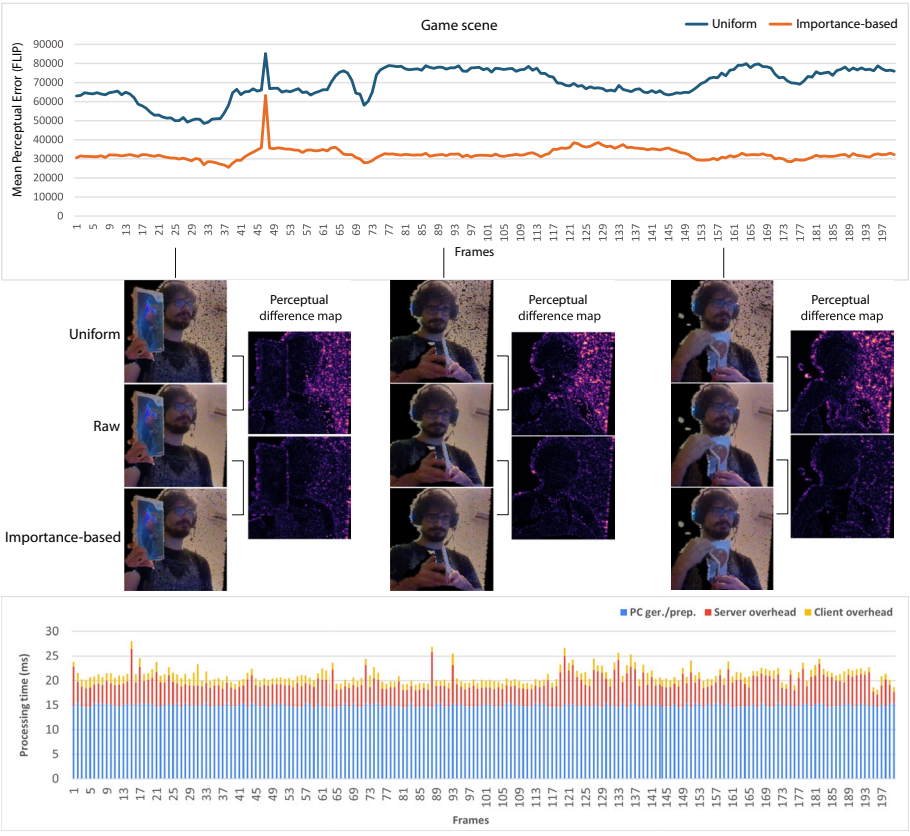

Fig. 9 Teleconferencing scenario: In first and second figures, ꟻLIP error comparison shows that our importance-based method achieves roughly half the error of uniform subsampling while preserving visual detail. Depth clipping is omitted for clarity (see text). Third figure depicts per-frame time consumption to a) generate the image buffer and extract the point cloud from the RealSense sensor (blue stack), apply in parallel all cluster operations discussed in Section 4.1.4 (red stack) and render the final result in the client as discussed in Section 4.2 (yellow stack)

Server-side processing In this scenario, our server processes an average of 3.7M sensor generated points per frame. Since only the reference entity along the indicated object is informative, we also apply depth clipping and keep valid points within $Z_n = 0.1$m, $Z_f = 1.3$m range. This effectively reduces the population to be processed to 156K on average. The high, medium, and low clusters process and encode approximately $C_{\text{high}} = 32$ K, $C_{\text{med}} = 9$K, and $C_{\text{low}} = 115$K points, respectively. This translates to roughly 2.23MB of data to be transmitted per frame. After the encoding phase, the clusters are reduced to $C_{\text{high}} = 29$K, $C_{\text{med}} = 8$K, and $C_{\text{low}} = 17$K points, respectively. This yields a total of 54K points ($\sim 2.8\times$ fewer), amounting to just 0.35MB ($\sim 6.3\times$ less) of data. The entire encoding phase runs in parallel across all clusters and takes roughly 4ms in total. Figure 9 depicts the cumulative per-frame work cycle of the server in order to apply subsampling, deduplication and encoding of all clusters.

Client-side processing Upon delivery, the point clouds transmitted by the server are immediately dispatched for rendering. Point cloud preparation consists solely of assigning the incoming byte streams to the associated WebXR vertex attribute buffers. This step requires roughly 0.2ms per delivered frame. Rendering the entire frame via the rasterization pipeline is also relatively fast, amounting to ~ 1.1ms per frame. Figure 9 depicts the cumulative per-frame work cycle of the client in order to apply buffer preparation and point cloud rendering rendering.

5.2 Object manipulation use case

For the object manipulation experiment (see Fig. 10), the RGB-D camera tracks a specific target object, focusing on its manipulation, which introduces occlusions and disocclusions of its geometry and requires a higher definition of the constituent parts in order for the client to understand the functional aspects of the manipulation. A possible application for such a test case is instructional 3D video streaming and task assessment.

The test sequence consists of 230 frames of point cloud data showing a remote controller, which becomes the ROI, as the user manipulates the device and explains how to change batteries. Here, it is assumed that the representation of the controller's geometry is important to clearly understand the spatial relationship of the participating objects and be able to discern functional details. Similar to the previous case, the reference object is detected and segmented using the SAM2 model. For this test case, we consider a viable size per frame to be ~ 50K so that the point cloud can be transmitted over 9.5Mbps and presented on a mobile device Samsung Galaxy A55 5G, powered by a Samsung Exynos 1480 processor and 8GB of RAM.

The perceptual difference against the ground truth reported by ꟻLIP for the importance-based is smaller than the one reported for the uniformly degraded version, but the visual quality gain is less pronounced compared to the teleconference case. However, sharp depth transitions were improved in the case of the importance-driven optimization (see ꟻLIP map insets), rendering the point cloud with better clarity, which is important in instructive videos.

Server-side processing Similar to the previous case, the server processes an average of 370K points per frame. However, due to the nature of this case, we omit the depth clipping filter. Throughout the frame sequence, the clusters are associated with approximately

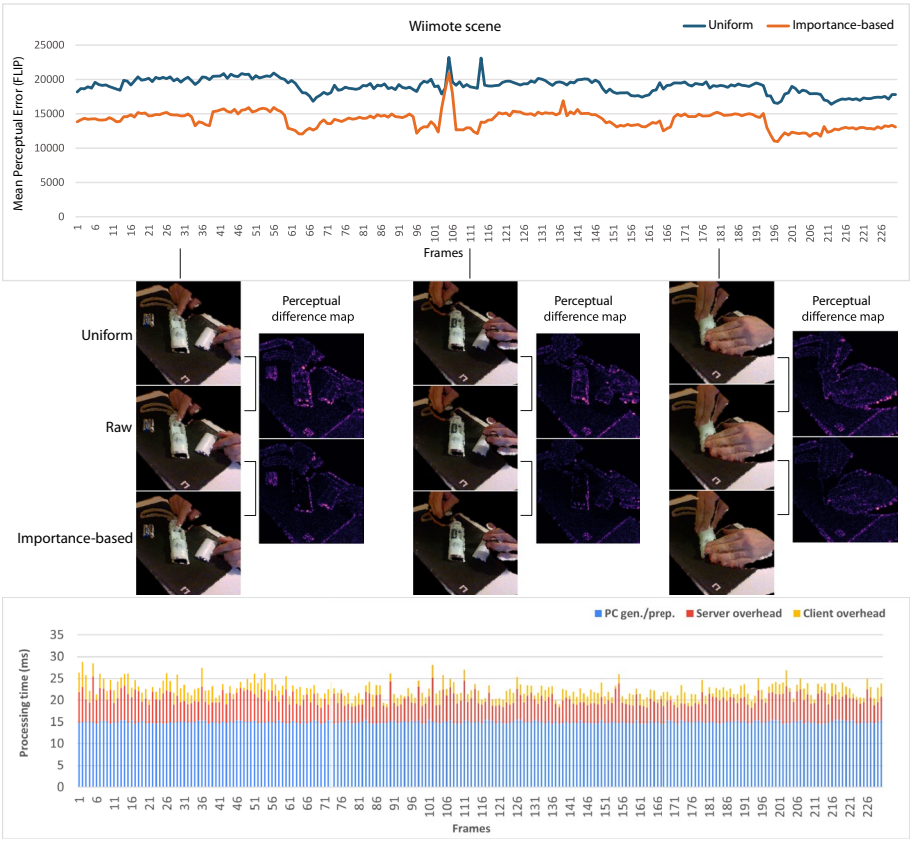

Fig. 10 Object manipulation scenario. In first and second figures, ꟻLIP comparison shows that our importance-based method yields lower error than uniform degradation, with improved depth transitions and visual task clarity. The third figure illustrates the per-frame time consumption broken down into three main stages: (**a**) generating the image buffer and extracting the point cloud from the RealSense sensor (blue stack), (**b**) applying the parallel cluster operations detailed in Section 4.1.4 (red stack), and (**c**) rendering the final output on the client side, as described in Section 4.2

$C_{\text{high}} = 17\text{K}$, $C_{\text{med}} = 16\text{K}$, and $C_{\text{low}} = 337\text{K}$ points, amounting to roughly 5.35MB of data. After the encoding phase, the resulting clusters contain an average of $C_{\text{high}} = 17$ K, $C_{\text{med}} = 14\text{K}$, and $C_{\text{low}} = 16\text{K}$ points, respectively. This yields a total of 47K points ($\sim 7.8\times$ fewer) and reduces the data size to about 0.3MB, requiring $\sim 17\times$ less bandwidth. Encoding this number of points requires approximately 5ms per frame. Finally, Fig. 10 depicts the cumulative per-frame work cycle of the server in order to apply subsampling, deduplication encoding of all clusters.

Client-side processing On this mobile device, the point clouds are processed and rendered as soon as all clusters are successfully delivered. Given the aforementioned transmission bandwidth, buffer preparation requires ~ 0.5ms, while the rendering phase takes about ~ 1 ms. As in the previous case, we fully exploit the rasterization pipeline and the WebXR API

to optimize our routines. Figure 10 depicts the cumulative per-frame work cycle of the client in order to apply buffer preparation and point cloud rendering.

5.3 Surveillance use case

One challenging use case for RGB-D cameras is the monitoring of a larger space, tracking user movement and actions within it. RGB-D cameras have very poor accuracy in the encoding of distances near the maximum detection distance and the use of uniform quantization and/or subsampling may introduce significant errors to already sparse point clouds with significant errors. Furthermore, quantization in non-important regions can be quite aggressive in the case of importance-based optimization, without sacrificing any quality, due to the noise already being present in the unmodified input point cloud.

The particular example chosen shows 400 frames of a user entering a room and interacting with two pieces of furniture (Fig. 11). The motion of the tracked human spans all but the closest part of the sensor range. For this test scenario we detect the region of interest (i.e., the human) via the YOLO11e approach. This approach is superior to the SAM2 alternative because our objective is not to detect a specific, granular object within the scene, but rather to segment a complete entity. We visualize the content through a Meta Quest 3 device powered by the Qualcomm Snapdragon XR2 Gen 2 platform, featuring 8GB of RAM. In this device we transmit roughly 95K per frame at 16Mbps network bandwidth over WiFi.

The perceptual difference against the ground truth reported by ꟻLIP for the importance-based is smaller than the one reported for the uniformly degraded version. However, the most important finding is the significant quality improvement of the tracked human. This can be easily observed by examining the indicative perceptual difference maps, where the person is clearly delineated as a dark patch, signifying low difference.

Server-side processing In this test case, the server processes an average of 400K points per frame. We also omit depth clipping filter. The clusters are associated with approximately $C_{\text{high}} = 16\text{K}$, $C_{\text{med}} = 14\text{K}$, and $C_{\text{low}} = 370\text{K}$ points, amounting to roughly 5.72MB of data. After the encoding phase, the resulting clusters contain an average of $C_{\text{high}} = 15$ K, $C_{\text{med}} = 10\text{K}$, and $C_{\text{low}} = 70\text{K}$ points, respectively. This yields a total of 95K points ($\sim 4.2\times$ fewer) and reduces the data size to about 0.55MB, requiring $\sim 10\times$ less bandwidth. Encoding this number of points requires approximately 5ms per frame. Figure 11 depicts the cumulative per-frame work cycle of the server.

Client-side processing On VR device the point cloud processing time behaves similar to the prior scenarios. Buffer preparation requires $\sim$ 1ms, while the rendering phase is real-time at $\sim$ 1ms. Figure 11 depicts the cumulative per-frame work cycle of the client.

5.4 Comparison with Draco

In this final section, we evaluate our method against the Draco compression library [8]. All our results are summarized in Table 1. For all use cases presented in the previous section, we apply the exact same configuration to encode and decode the stream. On the server side, we use the Python bindings for the Draco library. We also apply a multi-threaded scheme for each cluster to simulate the same behavior as the CUDA implementation. Moreover, we

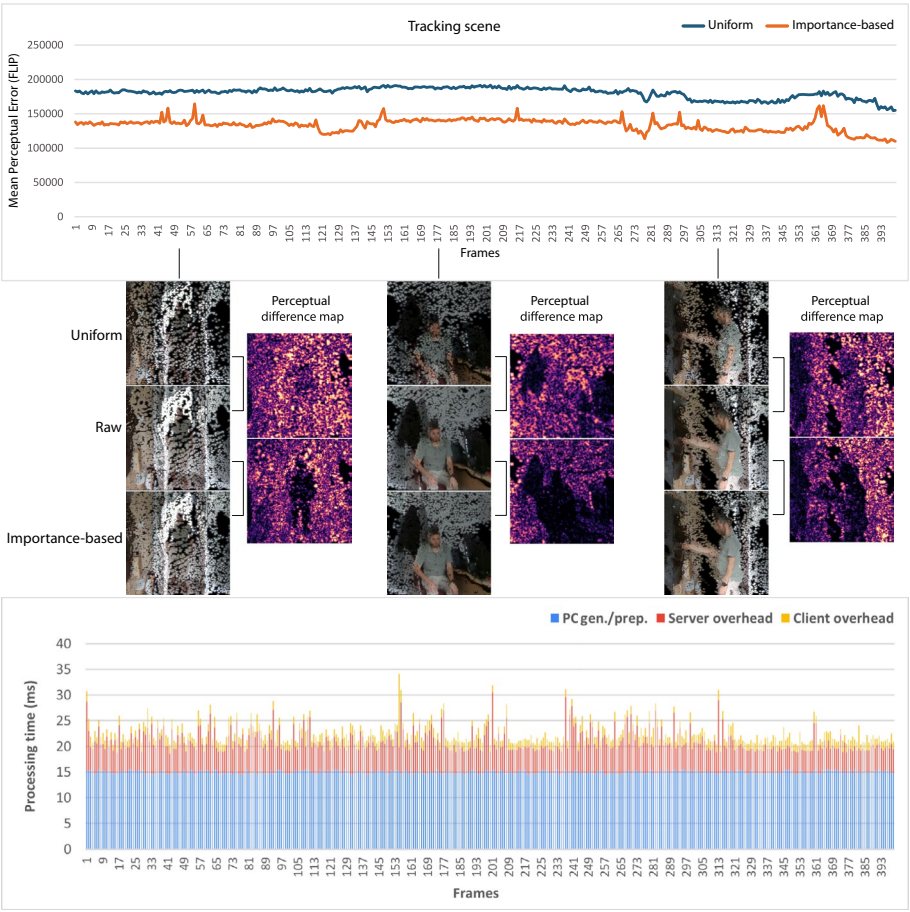

Fig. 11 Surveillance scenario: In first and second figures, ꟻLIP comparison shows that the importance-based method reduces error versus uniform degradation, with a marked quality improvement on the tracked human. The third figure presents a stacked breakdown of the per-frame time consumption. The blue stack represents the time to generate the image buffer and extract the RealSense point cloud. The red stack corresponds to the parallel execution of all cluster operations (Section 4.1.4), while the yellow stack denotes the time taken to render the final result on the client (Section 4.2)

Table 1 For each use case in Sections 5.1, 5.2, and 5.3 we depict average performance over all frames for Draco library versus our approach. We also highlight standard deviation for each measurement in the parenthesis. The teleconference client operates on a laptop device. The object manipulation case operates on a mobile device. Finally, the surveillance case operates on Oculus Quest device

	Encoding (ms)		Decoding (ms)		Rendering (ms)		Package (KB)	
Use case	Draco	Ours	Draco	Ours	Draco	Ours	Draco	Our
Teleconference	21.0 (±8.2)	4.0 (±1.4)	7.4 (±3.1)	0.2 (±0.08)	1.1 (±0.41)	1.3 (±0.52)	161 (±52)	358 (±136)
Object manipulation	20.0 (±4.3)	4.8 (±1.7)	18.5 (±5.3)	0.5 (±0.38)	0.3 (±0.09)	0.5 (±0.52)	130 (±21)	311 (±46)
Surveillance	36.9 (±6.7)	5.6 (±1.8)	39.4 (±8.7)	0.8 (±0.39)	0.6 (±0.51)	0.8 (±0.33)	262 (±17)	572 (±29)

use the fastest compression option for Draco which is, set at *compression level* equals 0. Likewise, for the client-side decoder, we use JavaScript WASM extensions to decode the Draco stream, which is then copied to the GPU buffer for visualization. We provide measurements for performance-critical procedures that highlight the significance of our contribution. However, we omit quality metrics, as the point cloud quantization algorithm is essentially the same for both methods, producing identical results.

In the *Encoding* column (top left block), we report the server-side encoding overhead for each use case. Note that all experiments use the same machine for the encoding phase but employ different algorithms (i.e., CPU vs GPU encoding). Our implementation consistently outperforms the Draco encoder, with performance gain ranging from $4\times$ to $6\times$. Since encoding is a mandatory step for each frame on the server side, our proposed implementation allows for real-time frame generation and broadcasting, unlike the Draco alternative.

In the *Decoding* column (top right block), we present the client-side decoding overhead for each use case. Note that decoding time is measured on the respective device, which differs depending on the use case, as detailed in previous sections. Our custom implementation for the decoding phase using rasterization demonstrates superior performance compared to decoding the stream with the Draco library and subsequently uploading the data to the GPU for visualization. This performance gain ranges from $37\times$ to $49\times$. Furthermore, as shown in the *Visualization* column (bottom left block), our vertex shader decoding introduces near-zero additional overhead (less than half a millisecond) to the total rendering time. This implies that our technique can be as well scaled to even larger point clouds without impacting rendering performance.

Finally, in the *Bandwidth* column (bottom right block), we measure the average bandwidth required to transmit packets per use case. Because our method relies on buffers with fixed padding so they can be directly uploaded to the client device, our transmission bandwidth is inherently larger, roughly $2.2\times$. Specifically, our positional encoder for high and medium quality clusters is transmitting at 32-bit alignment (see Section 4.1.4). While the low quality cluster is transmitted at $3\times$ 8-bit alignment buffers. Similarly, our color buffer is packed and transmitted pre-aligned at $3\times$ 8-bit buffers (see Section 4.1.4), as we do not apply any compression at any quality level. This is not the case for Draco, that applies several algorithms to further reduce size, but pays the price in terms of encoding/decoding/memory mapping performance.

6 Conclusions and future work

In this work, we proposed a real-time, context-aware point cloud streaming engine. We demonstrated how to fully exploit GPU features and algorithms, both on the server and the client, to achieve fast point cloud encoding and visualization for remotely connected clients with heterogeneous and potentially limited hardware. On the server side, we demonstrated how to use CUDA to parallelize the encoding of multiple clusters, alongside efficient neural network segmentation methods to identify critical regions. On the client side, we leveraged the rasterization pipeline and its attributes to bind incoming buffer streams with zero-copy overhead. Consequently, the primary bottleneck is rendering the final points, which can be performed in real-time by any device supporting rasterization-based point rendering. This

effectively allows us to scale our method to a few hundreds of thousands of points without severe overheads.

Currently, the allocation of the point cloud budget across the three distinct quality tiers, is driven by empirical heuristics designed to guarantee real-time performance. While highly effective in our testbed, formulating this importance-based allocation as a rigorous optimization problem represents a critical direction for future research. Specifically, we aim to model cluster allocation as a constrained bitrate allocation problem, where the objective is to maximize the perceptual utility of the client, such as minimizing the perceptual error, subject to real-time network bandwidth limits and latency thresholds. By moving away from static empirical presets and dynamically solving for optimal subsampling and quantization parameters on a per-frame basis, the system could theoretically guarantee the optimal balance between visual fidelity and transmission overhead under any fluctuating network state.

Our frame generator (server) and frame consumer (client) are sufficiently fast to accommodate transmissions over networks with relatively high latency. Furthermore, our approach serves as a stepping stone for incorporating multiple server-side cameras. This will enable point cloud fusion and adaptive spatial encoding while maintaining real-time frame rates. Moving forward, we hope that extending this architecture to address the challenges of large-scale deployment is a necessary step. While our current system effectively serves targeted clients, integrating more sophisticated network graphs such as, edge-computing nodes, could ensure low-latency distribution across dispersed user base. This will inevitably require to address severe network instabilities. Future iterations will explore dynamic, adaptive streaming protocols capable of real-time adjustment to fluctuating bandwidth and packet loss, ensuring uninterrupted holographic telepresence even under suboptimal conditions. However, in the special case of a single camera setup, it would also be an interesting avenue to study cases where the server is also an edge device. This would imply that different neural segmentation models should be considered in order to also run on these devices. Thanks to its zero-copy overhead technique, our client-side implementation is highly optimized to process point clouds even larger than those presented in this study. Yet, future research must also address hardware-specific constraints, such as thermal throttling and battery drain, on untethered mobile and XR devices.

Even more importantly, future iterations of WebXR will unlock capabilities currently restricted to desktop hardware, further enhancing parallel operations within the rasterization pipeline. This will allow our system to employ more sophisticated methods for decoding the input stream.

Acknowledgements This work was funded by the European Union under the SPIRIT project (Grant Agreement no. 101070672).

Author Contributions All authors contributed to the conception, design, and implementation of the work. Data collection and analysis were carried out by V. Kotsis-Panakakis and I. Evangelou. The first draft of the manuscript was written by I. Evangelou and A. A. Vasilakis, and all authors provided critical feedback on previous versions. All authors have read and approved the final manuscript.

Funding Open access funding provided by HEAL-Link Greece. This research received funding from the European Union under the SPIRIT project (Grant Agreement No. 101070672). Specifically, it was supported through a Financial Support to Third Parties (FSTP, cascading funding) project, HOPE, under Subgrant Agreement No. 37573. More information on the HOPE project is available at this link.

Data Availability The raw video streams and point cloud datasets generated and analyzed by this research work are publicly available on our GitHub repository.

Code Availability The complete implementation, developed in Python with multiprocessed architecture for real-time performance (Server) and in JavaScript using Three.js and WebXR (Client), including source code, configuration files, and deployment instructions, is available on our: GitHub repository. All algorithms are modular, allowing substitution of alternative algorithms or streaming components as needed.

Declarations

Ethics approval All participants in this study were the authors themselves. All participants provided informed consent prior to data collection. The consent process included clear explanations of the data being recorded (including camera recordings), the purpose of the data collection, and how the data would be used. Participants were informed about the possibility of long-term data storage and potential sharing with third parties or public access. The consent process also ensured that participants understood their right to withdraw consent at any time, as well as the anonymization procedures applied to protect their personal data. All procedures were conducted in full compliance with ethical guidelines for research involving human participants.

Financial interests The authors have no relevant financial or non-financial interests to disclose.

References

1. Wang P, Yang H, Billinghurst M, Zhao S, Wang Y, Liu Z, Zhang Y (2025) A survey on XR remote collaboration in industry. J Manuf Syst 81:49–74. https://doi.org/10.1016/j.jmsy.2025.05.016
2. Anton D, Kurillo G, Bajcsy R (2018) User experience and interaction performance in 2D/3D telecollaboration. Futur Gener Comput Syst 82:77–88. https://doi.org/10.1016/j.future.2017.12.055
3. Enenche P, Kim DH, You D (2025) On the road to the metaverse: point cloud video streaming: perspectives and enablers. ICT Express 11(1):93–104. https://doi.org/10.1016/j.icte.2024.11.001
4. Viola I, Cesar P (2023) Chapter 15 - volumetric video streaming: current approaches and implementations. In: Valenzise G, Alain M, Zerman E, Ozcinar C (eds) Immersive Video Technologies. Academic Press, pp 425–443. https://doi.org/10.1016/B978-0-32-391755-1.00021-3
5. Graziosi D, Nakagami O, Kuma S, Zaghetto A, Suzuki T, Tabatabai A (2020) An overview of ongoing point cloud compression standardization activities: video-based (V-PCC) and geometry-based (G-PCC). APSIPA Transactions on Signal and Information Processing 9. https://doi.org/10.1017/ATSIP.2020.12
6. Mekuria R, Blom K, Cesar P (2017) Design, implementation, and evaluation of a point cloud codec for tele-immersive video. IEEE Trans Circuits Syst Video Technol 27(4):828–842. https://doi.org/10.1109/TCSVT.2016.2543039
7. Rusu RB, Cousins S (2011) 3D is here: point cloud library (PCL). In: 2011 IEEE international conference on robotics and automation, pp 1–4. https://doi.org/10.1109/ICRA.2011.5980567
8. Galligan F, Hemmer M, Stava O, Zhang F, Brettle J (2018) Google/draco: a library for compressing and decompressing 3D geometric meshes and point clouds. https://github.com/google/draco
9. Esri (2016) LEPCC: limited error point cloud compression. https://github.com/Esri/lepcc
10. Hosseini M, Timmerer C (2018) Dynamic adaptive point cloud streaming. In: Proceedings of the 23rd packet video workshop. PV '18. Association for Computing Machinery, New York, NY, USA, pp 25–30. https://doi.org/10.1145/3210424.3210429

11. De Fré M, Hooft J, Chang C-Y, De Schepper K, Alface PR, De Vleeschauwer D, Wauters T, Steenkiste P, De Turck F (2025) Low-latency volumetric video conferencing in congested networks through L4S. In: Proceedings of the 16th ACM multimedia systems conference. MMSys '25. Association for Computing Machinery, New York, NY, USA, pp 113–123. https://doi.org/10.1145/3712676.3714443
12. Kotsis-Panakakis V-E, Evangelou I, Bistas F, Gkaravelis A, Vitsas N, Vasilakis A-A, Papaioannou, G (2025) HOPE: holographic optimized processing engine. In: EuroXR 2025: proceedings of the application, poster, and demo tracks of the 22nd EuroXR international conference. VTT Technical Research Centre of Finland, Finland, pp 149–151. https://doi.org/10.32040/2242-122X.2025.T440
13. Alkhalili Y, Meuser T, Steinmetz R (2020) A survey of volumetric content streaming approaches. In: 2020 IEEE Sixth International Conference on Multimedia Big Data (BigMM). IEEE Computer Society, Los Alamitos, CA, USA, pp 191–199. https://doi.org/10.1109/BigMM50055.2020.00035
14. Mildenhall B, Srinivasan PP, Tancik M, Barron JT, Ramamoorthi R, Ng R (2021) NeRF: representing scenes as neural radiance fields for view synthesis. Association for Computing Machinery, New York, NY, USA, vol 65, pp 99–106. https://doi.org/10.1145/3503250
15. Kerbl B, Kopanas G, Leimkuehler T, Drettakis G (2023) 3D Gaussian splatting for real-time radiance field rendering. ACM Trans Graph 42(4). https://doi.org/10.1145/3592433
16. Hormann K, Polthier K, Sheffer A (2008) Mesh parameterization: theory and practice. In: ACM SIGGRAPH ASIA 2008 courses. SIGGRAPH Asia '08. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1508044.1508091
17. Özyeşil O, Voroninski V, Basri R, Singer A (2017) A survey of structure from motion. Acta Numer 26:305–364. https://doi.org/10.1017/S096249291700006X
18. Wang F, Zhu Q, Chang D, Gao Q, Han J, Zhang T, Hartley R, Pollefeys M (2026) Learning-based multi-view stereo: a survey. IEEE Transactions on Pattern Analysis and Machine Intelligence 1–20. https://doi.org/10.1109/TPAMI.2026.3654665
19. Yang L, Kang B, Huang Z, Zhao Z, Xu X, Feng J, Zhao H (2024) Depth anything V2. In: Proceedings of the 38th international conference on neural information processing systems. NIPS '24. Curran Associates Inc., Red Hook, NY, USA
20. Bochkovskii A, Delaunoy A, Germain H, Santos M, Zhou Y, Richter SR, Koltun V (2025) Depth pro: sharp monocular metric depth in less than a second. In: International conference on learning representations. arxiv:2410.02073
21. Intel Corporation (2017) Intel realsense SDK 2.0. GitHub. https://github.com/IntelRealSense/librealsense
22. Yang M, Luo Z, Hu M, Chen M, Wu D (2023) A comparative measurement study of point cloud-based volumetric video codecs. IEEE Trans Broadcast 69(3):715–726. https://doi.org/10.1109/TBC.2023.3243407
23. Santos C, Rehbein G, Costa EAC, Corrêa G, Porto MS (2025) Efficiency and complexity analysis of video-based and geometry-based point cloud encoders. J Real Time Image Process 22(3):114. https://doi.org/10.1007/s11554-025-01689-9
24. Lee K, Yi J, Lee Y, Choi S, Kim YM (2020) GROOT: a real-time streaming system of high-fidelity volumetric videos. In: Proceedings of the 26th annual international conference on mobile computing and networking. MobiCom '20. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3372224.3419214
25. Thomas Y, Xylomenos G (2026) Ultra-low latency point cloud streaming in 5G. In: Michael-Grigoriou D, Zachmann G, Kopper R, Yoon SH, Zollmann S, Bourdot P (eds) Virtual reality and mixed reality, pp 22–39. Springer, Cham. https://doi.org/10.1007/978-3-032-03805-0_2
26. Han B, Liu Y, Qian F (2020) ViVo: visibility-aware mobile volumetric video streaming. In: Proceedings of the 26th annual international conference on mobile computing and networking. MobiCom '20. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3372224.3380888
27. ISO/IEC 23009-1 (2014) 2014: Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats, Geneva, CH
28. Heidarirad H, Wang M (2025) VV-DASH: a framework for volumetric video dash streaming. In: Proceedings of the 16th ACM Multimedia Systems Conference. MMSys '25. Association for Computing Machinery, New York, NY, USA, pp 256–262. https://doi.org/10.1145/3712676.3718339
29. De Fré M, Hooft J, Wauters T, De Turck F (2025) Scalable MDC-based WebRTC streaming for one-to-many volumetric video conferencing. ACM Transactions on Multimedia Computing, Communications, and Applications. https://doi.org/10.1145/3768314
30. NVIDIA Corporation (2025) CUDA C++ Programming Guide. NVIDIA. NVIDIA. https://docs.nvidia.com/cuda/cuda-programming-guide/

31. Lugaresi C, Tang J, Nash H, McClanahan C, Uboweja E, Hays M, Zhang F, Chang C-L, Yong M, Lee J, Chang W-T, Hua W, Georg M, Grundmann M (2019) MediaPipe: a framework for perceiving and processing reality. In: Third workshop on computer vision for AR/VR at IEEE computer vision and pattern recognition (CVPR) 2019
32. Ravi N, Gabeur V, Hu Y-T, Hu R, Ryali C, Ma T, Khedr H, Rädle R, Rolland C, Gustafson L, Mintun E, Pan J, Alwala KV, Carion N, Wu C-Y, Girshick R, Dollár P, Feichtenhofer C (2024) SAM 2: segment anything in images and videos. arXiv:2408.00714
33. Wang A, Liu L, Chen H, Lin Z, Han J, Ding G (2025) YOLOE: real-time seeing anything. arxiv:2503.07465
34. De Fré M et al (2023) Low-latency volumetric-video delivery for real-time conferencing. Master's thesis, Master's Thesis Dissertation. Master's thesis. http://lib.ugent.be/catalog/rug01:003150149
35. Andersson P, Nilsson J, Akenine-Möller T, Oskarsson M, Åström K, Fairchild MD (2020) FLIP: A difference evaluator for alternating images. Proc ACM Comput Graph Interact Tech 3(2). https://doi.org/10.1145/3406183

Authors and Affiliations

Vasileios Ektor Kotsis-Panakakis[1] · Iordanis Evangelou[1] · Fotios Bistas[1] · Andreas A. Vasilakis[1] · Georgios Papaioannou[1] · Anastasios Gkaravelis[2] · Nikolaos Vitsas[2]

✉ Iordanis Evangelou
iordanise@aueb.gr

Vasileios Ektor Kotsis-Panakakis
vas.kotsispanakakis@aueb.gr

Fotios Bistas
fot.bistas@aueb.gr

Andreas A. Vasilakis
abasilak@aueb.gr

Georgios Papaioannou
gepap@aueb.gr

Anastasios Gkaravelis
anastasios@phasmatic.com

Nikolaos Vitsas
nick@phasmatic.com

[1] Department of Informatics, Athens University of Economics and Business, Patision 76, Athens 10434, Attiki, Greece

[2] Phasmatic Private Company, Prokopiou 11, Peristeri 12131, Attiki, Greece