MeshReduce: Scalable and Bandwidth Efficient 3D Scene Capture

Tao Jin* Carnegie Mellon University
Mallesham Dasari† Northeastern University
Connor Smith‡ NVIDIA
Kittipat Apicharttrisorn§ Nokia Bell Labs
Srinivasan Seshan¶ Carnegie Mellon University
Anthony Rowe‖ Carnegie Mellon University, Bosch Research

ABSTRACT

3D video enables a remote viewer to observe a 3D scene from any angle or location. However, current 3D capture solutions incur high latency, consume significant bandwidth, and scale poorly with the number of depth sensors and size of scenes. These problems are largely caused by the current monolithic approach to 3D capture and the use of inefficient data representations for streaming. This paper introduces `MeshReduce`, a distributed scene capture, stream, and render system that advocates for the use of textured mesh data representation early in the 3D video capture and transmission process. Textured meshes are compact and can provide lower bitrates for the same quality compared to other 3D data representations. However, streaming textured meshes creates compute and memory challenges to achieve bandwidth efficiency. `MeshReduce` addresses these issues by using a pipeline that creates independent mesh reconstructions and incrementally merges them, rather than creating a single mesh directly from all sensor streams. While this enables a more efficient implementation, this approach requires optimal exchange of textured meshes across the network. `MeshReduce` also incorporates a novel approach for network rate control that divides bandwidth between texture and mesh for efficient, adaptive 3D video streaming. We demonstrate a real-time integrated embedded compute implementation of `MeshReduce` that can operate with commercial Azure Kinect depth cameras as well as a custom sensor front-end that uses LiDAR and 360° camera inputs to dramatically increase coverage.

Index Terms: Computing methodologies—Computer graphics—Graphics systems and interfaces—Mixed / augmented reality; Information systems—Information systems applications—Multimedia information systems—Multimedia content creation

1 INTRODUCTION

Recent advances in high-performance hardware, better depth sensing technology, and advanced graphics algorithms have brought us closer to practical real-time 3D video streaming systems. A core component of existing systems [30,41,47] is the ability to acquire and digitize 3D scenes in real-time and stream this data over the Internet at practical bitrates. Unlike traditional 2D videos, acquiring 3D scenes requires multiple cameras capturing both color and depth information from different viewpoints. These color and depth streams are merged into a single 3D scene representation to enable remote users to view the captured scene with 6 Degrees-of-Freedom.

In this paper, we present `MeshReduce`, a live 3D scene capture streaming system designed with the following requirements:

1. **Low Latency:** For interactive live streaming, we expect latencies on the order of 2D video conferencing systems (<100ms).
2. **Scalable:** 3D video quality is often a function of number of cameras and scene size. An ideal capture solution should support dozens of sensors with commodity hardware.
3. **Adaptive Streaming:** The system must operate given practical bitrates for Internet streaming, and the quality of the system should adapt to bandwidth availability.

Existing 3D scene capture systems encompass a diverse range of technologies, ranging from systems that directly generate point clouds [23,27,32,50], to those producing textured meshes [11,16,41], and extending to systems that stream compressed implicit volumetric representations [51–53]. In comparing point clouds and textured meshes, a key distinction lies in representation efficiency: textured meshes typically require less data for storage and transmission to achieve comparable quality. This efficiency is attributed to two main factors: **(1)** the natural ability of mesh geometry to capture and compactly represent planar features prevalent in real-world scenes, and **(2)** the separation of geometry from texture data in textured meshes, which facilitates the creation of efficient 3D bitstreams for adaptive streaming. Conversely, while streaming implicit volumetric representation has demonstrated advantages in streaming data rate and compression latency, it introduces additional complexity on the client side, particularly in the extraction of explicit surfaces from implicit representations. Considering these aspects, `MeshReduce` advocates the use of textured mesh representation for 3D streaming, recognizing their balanced blend of efficiency and practicality.

However, streaming textured meshes is not without challenges. This approach places the computational burden of converting a sensor's raw data (e.g., RGB-D or point cloud) into a mesh representation on the capture side (or sender side) of the streaming pipeline. Existing systems [17,41] rely on monolithic scene capture pipelines and, as a result, incur high data rates (e.g., Holoportation requires 2 Gbps bandwidth for each scene [41]). These systems also scale poorly with scene size and sensor count due to the high compute and GPU memory demands, which in turn limits their capture quality (i.e., support only a few sensors while most systems need more than a dozen sensors for high-quality capture [11]). More importantly, there hasn't been much exploration on generating optimal bitrates for network adaptive textured mesh transmission.

`MeshReduce`, as depicted at the top of Figure 1, implements a distributed approach for scene capture (§3.1), effectively addressing the constraints commonly associated with monolithic systems. At the core of this design are edge devices at each sensor, which are responsible for performing individual 3D scene reconstructions. These are then cohesively integrated by a hierarchical merging server to form a unified textured mesh (§3.2). This approach allows us to handle the compute and memory demands more efficiently by operating on smaller, per-sensor scenes.

Key considerations in this architecture involve the strategic distribution of computational tasks between the sensor-side compute nodes and the hierarchical merging servers to optimize latency and resource utilization. Furthermore, `MeshReduce` produces a

*e-mail: taojin@andrew.cmu.edu
†e-mail: m.dasari@@northeastern.edu, work done while at CMU
‡e-mail: cosmith@nvidia.com, work done while at Magic Leap
§e-mail: kittipat.apicharttrisorn@nokia-bell-labs.com, work done while at CMU
¶e-mail: srini@cs.cmu.edu
‖e-mail: agr@andrew.cmu.edu

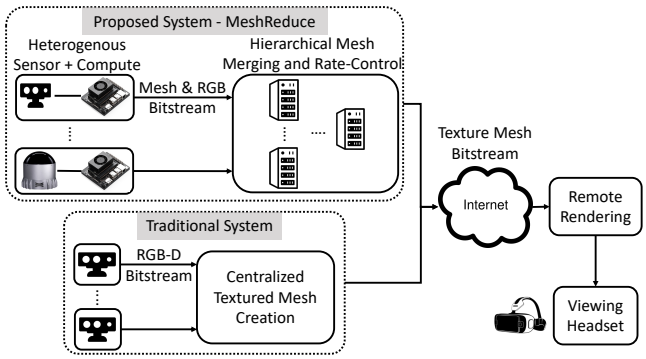


Figure 1: Traditional vs proposed system. Unlike the existing centralized pipeline, MeshReduce adopts a distributed approach where each sensor (RGB-D or LiDAR) is equipped with an embedded edge device processing per-sensor scene, with hierarchical merging servers merging all the partial scenes and preparing optimal 3D textured mesh bitstreams for adaptive network transmission.

bandwidth-efficient texture atlas (§3.3) to maintain texture spatial-temporal coherency and eliminate redundant camera views prior to compression and streaming. An additional aspect of MeshReduce involves rate control for streaming textured meshes (§4). This encompasses adapting to varying Internet bandwidth conditions, which involves carefully considering how bandwidth is allocated between texture and mesh components and selecting coding parameters that strike an optimal balance between quality and bitrate.

We prototype MeshReduce using off-the-shelf depth cameras and evaluate our own custom bitrate control and streaming schemes that can all execute in real-time on Nvidia Jetson class embedded processors. Current commercial depth sensors have a limited range and field of view, making them impractical for capturing scenes beyond capture domes [11]. We experiment with a custom LiDAR + 360° camera solution and show its promise in enabling wide-area coverage. Compared to existing capture pipelines, MeshReduce significantly reduces bitrate requirements (10-16×) for the same visual quality, offers lower latency, and scales well with a large number of sensors as well as larger area scenes. In summary, our key technical contributions are the following:

- The design and implementation of MeshReduce, an end-to-end 3D scene capture system that provides Internet-friendly flexible bitrates with low latency.
- A series of systematic measurement studies to study key bottlenecks of the scene capture pipeline and decompose the 3D scene reconstruction pipeline (§3.1) and mesh merging (§3.2).
- A bandwidth efficient texture mapping pipeline (§3.3).
- Scene-independent, efficient textured mesh bitstream generation using a predictive model of rate-distortion curves (§4).
- An open source implementation of MeshReduce and a first-of-its-kind dataset with 3D scenes captured using multiple commodity and custom LiDAR + 360° depth sensors.

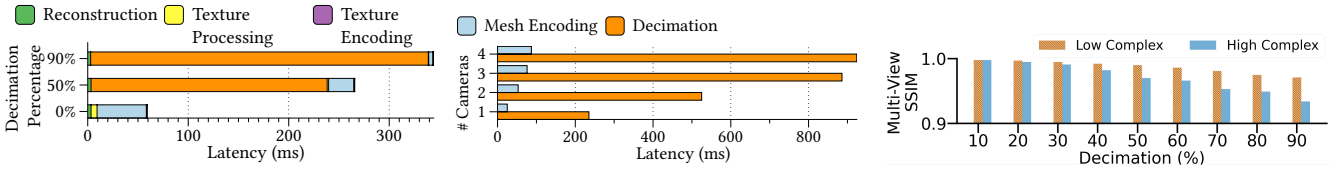
2 Related Work, Background and Motivation

Many of the opportunities we explore in this paper stem from a gap that exists between the graphics and networked systems communities. The graphics community has focused primarily on the visual quality of scene capture through textured meshes [38,41,51], which is often best explored in a centralized fashion with lossless data sources. In contrast, the networking and systems community has focused on directly streaming low-level sensor data (e.g., point cloud) to end users [22,23,32,61]. In this work, we focus on streaming textured mesh sequences as a 3D video.

3D Scene Capture: There has been extensive work on 3D scene capture pipeline. Much of the prior work focuses on efficient algorithms to generate high-quality scenes [11,24,33–35,39] and often ignores the computational complexity of capture pipeline. Recent solutions tackle this with memory-efficient data structures for faster reconstruction or consume less compute resources in limited environments without sacrificing the quality of the scenes [15,29,31,40,60]. Built on such extensive work, recent solutions introduce end-to-end 3D streaming pipelines using textured mesh representation such as FVV [11], Holoportation [41], Fusion4D [16] and Montage4D [17]. However, these systems mostly focus on high bandwidth (e.g., ≈1-2Gbps) locally networked scenarios and do not discuss the practicality of live streaming on wide area Internet where only a few tens of Mbps bandwidth is typically available. Another branch of work leverages compressed implicit volumetric representations [52,53] for capture and streaming. While enabling Internet-friendly data rate and real-time operation, these methods place compute and memory intensive surface extraction workloads on client devices. [48] uses spatial-temporally offset cameras to increase coverage, quality, and capture frame rate. While using more cameras to capture a constrained area with a single server to perform textured mesh reconstruction, as also demonstrated in [41,52], it is not scalable in terms of area coverage due to implicit surface memory requirement increases with respect to scene size rather than number of sensors.

3D Streaming Over Wide-Area Internet: Many of the prior work on 3D streaming uses point clouds or RGB-D [21–23,32,61]. These systems suffer from high bandwidth requirements and impose a significant computation burden of 3D scene reconstruction on the client side, which is not desirable. Also, these systems focus on on-demand streaming and suffer from latency in live scenarios. VR-Comm [22] proposes a real-time system with low latency but uses RGB-D representation and suffers from bandwidth overheads as well. Various techniques for maximizing streamed textured mesh perceptual quality under network bandwidth constraints have been explored. [54] proposes a bit-allocation algorithm that maximizes progressive textured mesh quality. [10] explores network adaptive streaming of Level-of-Detail textured meshes. [13] adapts the 3D mesh capture pipeline to generate and stream varying quality data based on network conditions. Output-sensitive approaches for avatar streaming have also been explored recently. [28] leverages feedback from the viewing client to guide the 3D capture and reconstruction to produce merged avatars for better visual quality while consuming less memory and bandwidth. Although this approach shows promise in delivering a decreased bandwidth for the server output stream, it does not scale in terms of area coverage when many users are present and looking at an expansive area. Unlike the prior work, MeshReduce adopts DCT (Discrete Cosine Transform) energy to establish a rate control tailored for capturing and streaming 3D textured meshes, delivering high-quality textured meshes at reduced bitrates and latency. The aforementioned previous work on adaptive streaming focuses on making optimal decisions on which bitrate/quality to select for a given network condition. On the contrary, our work is more focused on generating those bitrates (by analyzing the optimal Pareto curves) to enable adaptive streaming. More importantly, MeshReduce focuses on capturing and streaming unbounded and full scenes, whereas much of the existing work is on isolated objects (e.g., faces or human bodies). Our approach also optimizes the capture-side output by identifying the optimal mesh reconstruction parameters and generates a spatial-temporal consistent texture map that modern video codecs can leverage.

Mesh Reconstruction Pipeline: Creating textured meshes involves three tasks: (1) Reconstruction, (2) Decimation, (3) Texture Mapping. The reconstruction task takes overlapping RGB-D frames (from multiple sensors) as input and incrementally combines them into an implicit surface representation (e.g., Truncated Signed Distance Function (TSDF) [12]). Once the implicit surface is created, a



(a) Single camera scene (b) Multi-camera scene (4 cameras) (c) Decimation as knob to adapt quality

Figure 2: Breakdown of scene capture latency: (a) Decimation has the highest latency among all tasks, (b) Decimation latency increases as we add more sensors to capture scenes for better quality, (c) Decimation can be used to get high quality at low bitrates.

polygon extraction algorithm, such as Marching Cubes [33], is used to create a mesh. Decimation reduces polygon count from the initial mesh while preserving the overall shape, volume, and boundaries as much as possible. We can leverage decimation as an important knob to remove a certain percentage of polygons in a given mesh and reduce data rate. Texture mapping task maps each polygon to a corresponding texture region. This mapping information is stored inside the mesh data structure and used to apply texture onto polygons during rendering.

3 Distributed Mesh Reconstruction

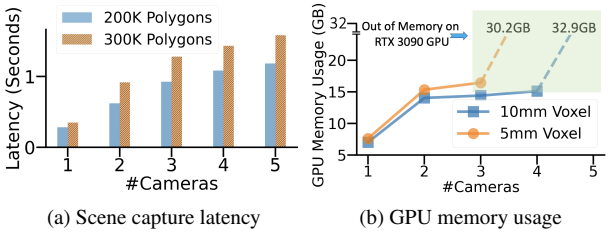
In this work, our vision centers around adopting textured mesh representation for efficient 3D streaming. `MeshReduce` is designed to generate textured mesh data with low latency, addressing memory and scalability issues, and achieving practical bitrates for Internet streaming. A block-level overview is depicted in Figure 5.

3.1 Decomposing Mesh Reconstruction

Compute Latency Analysis: In order to optimize the mesh capture pipeline effectively, a deep understanding of the compute latency of different tasks is crucial. As illustrated in Figure 2 and Figure 3a, we analyze the computation latency across various tasks in the textured mesh capture pipeline. In this experiment and our system implementation, we use TSDF to integrate sensor depth readings, followed by Marching Cubes for surface extraction. TSDF integration is implemented with voxel hash data structure [40]. Mesh decimation is based on Quadric Error Metric (QEM) [19]. Texture processing uses the projective texture process, and its encoding is handled by H.264 hardware encoder [44]. Mesh encoding is done with Edgebreaker algorithm [14, 49].

This experiment involved capturing a 5x5m area using multiple RGB-D cameras, with the reconstruction resolution set to 1cm. Our findings indicate that mesh `decimation` stands out as the most time-consuming task. This is primarily due to our single-core implementation of the QEM algorithm, which involves calculating the Quadric Error for each vertex in a sequential manner, selecting the optimal edges for a collapse based on these metrics, and then iteratively updating the mesh and recalculating error metrics after each collapse. While simpler and more straightforward, this method contributes to the significant time consumption observed in the mesh decimation task. As evidenced in Figure 2a, even with a single camera scene at 50% decimation, the latency exceeds 200ms, and this latency further escalates with higher levels of decimation. While our actual system implementation includes a more advanced, parallel version of the QEM algorithm similar to [63] for enhanced efficiency, *it remains the most significant contributor to overall latency*, shown in (§6.2), Figure 2 serves as a crucial reference point, illustrating the computational demands and latency issues inherent in the single-core approach to QEM-based mesh decimation.

This latency becomes more acute (>1s, see Figure 2b) when we decimate more complex meshes, i.e., more complete mesh created from more sensors. On the other hand, decimation significantly reduces polygon count while maintaining quality. A qualitative example of decimation value in reducing polygon count vs. texture quality can be seen in Figure 4. We also objectively evaluate the decimation impact in reducing bitrate while maintaining quality (Figure 2c). For a low complexity scene, Multi-View SSIM of the rendered quality is reduced by only 0.02 relative to the original when 90% of the triangles are removed. Even a high-complexity scene can be decimated by 50% without having a significant perceptual quality difference. This shows that decimation can be used as an important knob to trade mesh and texture bitrates when streaming over a network since decimating mesh significantly reduces bandwidth (more details on texture vs. geometry adaptation in §4).



(a) Scene capture latency (b) GPU memory usage

Figure 3: Challenges of streaming textured meshes in a centralized scene capture architecture: (a) High scene capture latency, (b) High GPU memory usage with the increase in the number of cameras.

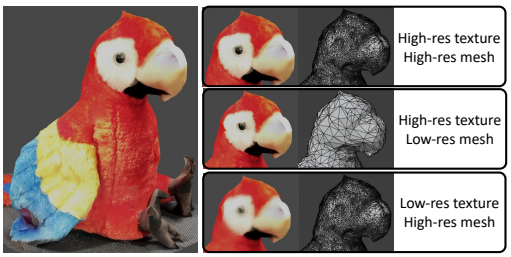


Figure 4: Decimation example showing significantly reduced polygon count with little impact on quality (middle compared to the top). Reducing texture resolution, on the other hand, degrades the quality significantly (bottom). The left side shows the original model.

GPU Memory Limit: Another key finding is that the reconstruction task leads to GPU memory bottleneck. While previous mesh reconstruction approaches were CPU-constrained, recent advancements (e.g., voxel hash data structures, data-parallel polygon extraction) [33, 40, 51] have significantly accelerated this process. This acceleration is evident as shown in Figure 2a, which shows a latency of less than 1 millisecond in the reconstruction process.

However, this increase in speed comes with a substantial trade-off in GPU memory usage. The mesh reconstruction process now requires considerable GPU memory, especially for implicit surface storage and polygon extraction. This demand significantly constrains the system's capacity, limiting it to handling only 3-5 cameras at a time, even with advanced GPUs like the NVIDIA RTX 3090 (shown in Figure 3b). This limitation becomes especially pronounced when

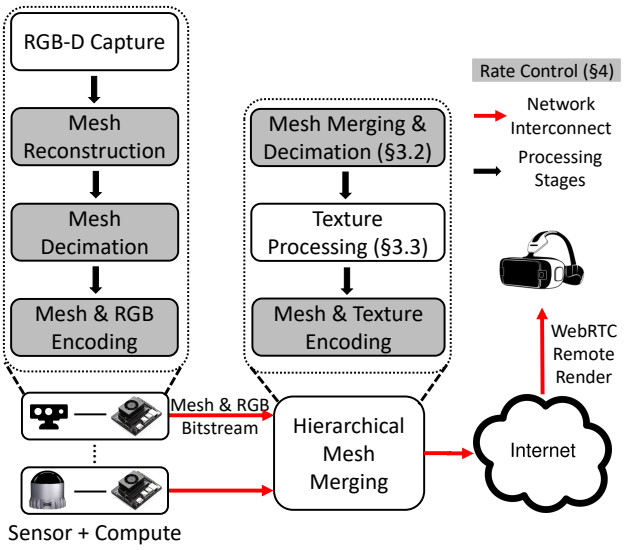


Figure 5: `MeshReduce`'s distributed scene capture and rate control pipeline for textured mesh streaming.

considering the potentially cubic increase in GPU memory usage with every doubling of the capture resolution in three dimensions.

Having an understanding of the bottlenecks of the existing mesh reconstruction pipeline, we now discuss our principles of leveraging edge (sensor side) embedded compute to enable a telepresence system that operates at interactive latency while providing wide area coverage. This is achieved through (1) decomposing mesh reconstruction to sensor side compute nodes and (2) composite meshes reconstructed from heterogeneous sensors into a unified 3D scene.

Splitting the Capture Pipeline: In order to alleviate the GPU memory and latency bottlenecks shown above, we split the traditional monolithic capture pipeline across edge nodes where each node performs a smaller single-camera mesh reconstruction and decimation. The system then merges these per-camera reconstructions into a single mesh on hierarchical merging server. Each edge node has sufficient resources to reconstruct a single sensor scene and decimate with minimal latency due to limited scene coverage. Another important result of this design is that the merging process at the server is fast even on the CPU and does not necessitate GPU compute, making it DRAM-bounded (which is abundant in practice).

The above distributed pipeline can be deployed in two ways: (1) each camera transmits its RGB-D streams to a central server farm that has a cluster of compute nodes, where each node performs per-camera scene reconstruction, and another node merges all of them into one, or (2) the per-camera compute node is co-located with the camera and transmits the reconstructed mesh and RGB bitstreams to a central location for merging. `MeshReduce` takes (and advocates) the latter approach for the following reasons: (i) RGB-D consumes more bits for the same quality than mesh streams, and converting to mesh representation as early as possible in the pipeline is more efficient, and (ii) we envision the future 3D sensors to perform the sensor-side reconstruction pipeline in the embedded hardware, much like current generation cameras performing video compression.

3.2 Mesh Aggregation and Merging

Once the per-sensor reconstructions are available at the merging server, the next step is to merge all the partial reconstructions into a single model. In multi-view sensor setups, mesh reconstructions often contain overlapping regions, similar to previous work that explored merging multiple meshes to form complete 3D scans [55]. Ideally, merging these meshes seems like a straightforward process of concatenating mesh data structures and removing duplicate polygons. This approach, however, presumes a perfect simulation environment where sensors are idealized, consistently yielding identical triangle configurations (i.e., same vertex positions for the same surface) as produced by surface extraction algorithms such as Marching Cubes. In contrast, real-world conditions introduce significant complexities. Factors such as sensor noise and imperfect calibration lead to variations and misalignments in the generated mesh, making the simple mesh merging process infeasible. In addition, it is challenging to determine mesh intersection and union, given the partial meshes do not have clear boundaries and are not watertight.

Realistically, when overlaying meshes are produced by different sensors with a partial overlapping field of view, it is common to observe non-perfect mesh overlaps. Figure 6(a) shows an example of overlaying two meshes with partial overlap; regions that demonstrate interleaving red and blue colors consist of two layers of mesh with inaccurate reconstruction due to sensor noise and calibration error, along with Z-fighting. To merge these two independent meshes into one, one approach can be grouping the two sets of mesh vertices into clusters based on their proximity and then replacing each cluster with a representative vertex. This is similar to the idea of vertex clustering in mesh decimation. However, vertex clustering based on vertex proximity leads to worse geometry accuracy as it does not consider the error introduced in this process.

To mitigate this, we opt for a two-step procedure. First, we use raycasting, implemented with Bounding Volume Hierarchy for efficiency, to accurately identify and remove overlapping regions, obtaining meshes with no overlap. Then, we use a nearest-neighbor search, implemented with the KD-Tree data structure, to merge the boundaries of the meshes, forming a seamless model.

Overlap Removal: The raycasting step is crucial for determining which parts of a mesh surface are inside or obscured by another mesh. This approach is particularly effective when dealing with the complexity of intersecting non-watertight surfaces. For this step, we gather all cameras' intrinsic and extrinsic parameters to construct rays based on the pinhole camera model. These rays are then cast across the partial meshes to detect intersections.

The outcomes of the raycasting process are twofold: Firstly, if a ray results in no hits or just a single hit, it indicates the absence of overlap, signifying that the scanned volume is captured exclusively by one camera. Secondly, more than one hit suggests a potential overlap. However, this does not sufficiently confirm an actual overlap, as the ray might intersect multiple triangles, either representing different objects or the same surface as seen from various camera perspectives.

To discern whether a potential overlap is indeed an actual overlap, we analyze whether the rays intersect with triangles within a predefined range threshold. An overlap is confirmed if the intersecting hits fall within this range. Conversely, overlaps are disregarded if the hits lie beyond this range. The value is carefully chosen based on the accuracy and precision of the sensors involved, ensuring a balance between accurate overlap detection and the avoidance of false positives. We empirically set this threshold as double the amount of sensor precision. Once we have determined the case where there are multiple triangles within this set threshold, we remove the ones that have a further distance with respect to the camera. At the end of this process, we also perform island removal, based on connected triangle areas, to remove any floating surfaces as a result of the overlap removal process. The surface area threshold is determined as double the mesh reconstruction resolution from the TSDF block.

Hierarchical Mesh Merging: Once we successfully eliminated overlapping regions in the meshes, the remaining partial meshes each distinctly represent separate volumes of the space, as shown in Figure 6b. Our goal now is to create a seamless model by integrating the partial mesh boundaries. We use nearest-neighbor search (with KD-Tree data structure based on radius) to identify and collapse close vertices. We locate and operate only on mesh boundaries

(a) Before redundancy removal (b) After redundancy removal (c) Zooming in merged mesh

Figure 6: (a) Overlapping mesh from multiple camera angles. The surface has an interleaving topology due to camera calibration error and sensor noise. Redundant triangles incur higher bandwidth requirements and negatively affect perceptual quality. (b) cleaner and more efficient geometry representation after using raycasting to remove redundant triangles. Blank region due to camera occlusion. (c) zoomed in qualitative demonstration of mesh merging. Two independent mesh volumes are connected on edges, forming a complete and smooth 3D scene.

(by checking boundary conditions, i.e., the number of connected neighbors of a triangle edge) instead of the whole mesh model for efficiency. This ensures the traversal of only necessary mesh vertices for merging. In our implementation, the radius search threshold is determined as the square root of the reconstruction resolution. However, it's important to note that this parameter is highly dependent on the specifics of the scene and typically requires fine-tuning through empirical experimentation. The end result of this process is depicted in Figure 6c.

To address the scalability of the mesh merging design, we introduce hierarchical mesh merging that uses multiple merging servers at different layers to ensure each server is capable of handling the merging workload with enough throughput. The process begins with each server conducting parallel localized merges of partial meshes. This initial step effectively reduces the overall data volume, setting the stage for more manageable subsequent processing. This hierarchical approach mitigates latency limitations in mesh merging by distributing the workload across multiple servers and allowing for concurrent processing. As a result, it enables a scalable and distributed pipeline that can efficiently handle large-scale mesh integration tasks. The choice of the number of servers and layers is highly scene-dependent and needs to be tested and determined based on sensor arrangement and server capability.

3.3 Bandwidth-efficient Texturing and Atlas Generation

Spatial-Temporal Coherent Texture Generation: After merging partial meshes from each sensor view, we focus on generating a bandwidth-efficient texture map and atlas using the camera RGB frames. This process serves two purposes. Firstly, it creates a 3D to 2D texture map that enables a rendering client to project textures onto the geometry during rendering. Secondly, it forms a bandwidth-efficient and spatial-temporal consistent texture atlas by excluding redundant camera views.

We employ projective texturing to establish a mapping between 3D triangle vertices and 2D camera image coordinates. Traditional methods often utilize an angle test to determine this mapping, assigning triangle vertices to image coordinates by selecting cameras that minimize the angle between the camera's view vector and the triangle vertex normal, thereby ensuring minimal angle distortion [7]. This approach, however, can lead to spatial incoherence in the texture atlas, as neighboring triangle vertices might be mapped from different camera views. Such a scenario becomes particularly problematic when overlapping views are removed to enhance bandwidth efficiency, resulting in a loss of spatial coherence in the texture atlas. Such a scenario results in suboptimal compression efficiency for block-based 2D video coding methods like H.264 or VP9 ([4, 59]).

To address this issue, our method prioritizes spatial coherency by selecting the same camera to texture map adjacent triangle vertices whenever possible, especially when overlapping camera views are within a predetermined angle threshold. This approach ensures a more cohesive texture atlas, resulting in better compression. However, we revert to the angle test for better visual quality in situations where cameras are significantly divergent in their angles. Additionally, to further enhance video compression ratios, we replace pixels that don't correspond to any triangles with a uniform color (black in our case). We efficiently identify these regions by reusing the raycasting results from the mesh merging step, as detailed in §3.2, to pinpoint areas that do not register a hit. This strategy leverages video codecs' spatial and temporal redundancy capabilities, as such uniform areas are compressed more efficiently. Figure 7b illustrates an example of our spatially coherent texture atlas.

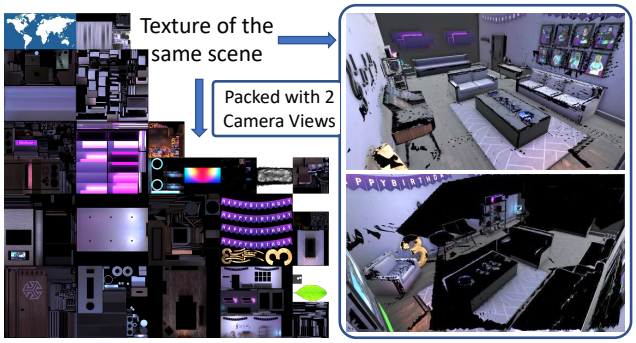
Texture Processing Placement: In deciding the optimal placement for texture atlas generation within our scene capture pipeline, we initially considered streaming RGB videos directly from the sensor side to the Internet, aligning with our system's distributed design approach. This method could reduce the computational load on the hierarchical merging server. However, we have decided not to choose this design based on two key factors:

Firstly, streaming textures directly from the cameras incurs a higher bandwidth overhead. In contrast, processing textures at the hierarchical merging server allows for the creation of a stitched and spatial-temporally coherent panoramic video, which is more bandwidth-efficient. The traditional 2D video codecs, such as H.264 or VP9, utilized in this process are more effective at compressing a unified panoramic view rather than separate views due to their ability to exploit spatial and temporal redundancy across views.

Secondly, the computational tasks at the hierarchical merging server, like mesh merging and compression, are predominantly CPU-intensive. At the same time, texture processing, especially using hardware codecs like NVENC [44], is GPU-intensive. By handling texture processing at the hierarchical merging server, we can parallelize these CPU and GPU tasks effectively, optimizing the overall efficiency and resource utilization.

3.4 Wide Area Coverage

Today's RGB-D (e.g., Azure Kinect [1]) sensors are not suitable to capture scenes at scale due to their limited field of view and range. In practical scenarios, it is important that we can capture wide-area environments by simply deploying a few sensors. To address this issue, we custom build a new wide area sensing module using LiDAR and 360° camera (see Figure 10). While providing sparse point clouds compared to RGB-D sensors, LiDAR excels in its long-range and wide field-of-view depth sensing, making it an ideal candidate for wide area coverage. However, LiDAR sensors lack color information, which is critical for telepresence applications. To address this limitation and provide color information, we co-locate a 360° color camera for each LiDAR sensor. By fusing data from a combination of these RGB-D and LiDAR + 360° sensors, we obtain textured mesh for wide area room-scale environments.



(a) Traditional Texture Atlas (b) MeshReduce Texture Atlas

Figure 7: Texture atlases of existing approach [20] vs. MeshReduce. (a) Existing approach ignores spatial coherency, (b) MeshReduce preserves spatio-temporal camera view structure (with blacked-out overlapping regions) to allow efficient video compression.

4 Rate Control for Textured Mesh Streaming

Given our proposed distributed sensor network for 3D capture, one of the core challenges is being able to adapt how we transmit data either between the internal wireless links or across the Internet in the presence of variable network conditions. This requires efficient bitstream generation for transmission with the goal of approaching optimal bitrates for rate adaptation algorithms. In Figure 5, we show this capability where all the red edges have textured mesh transmission over links with variable bandwidth.

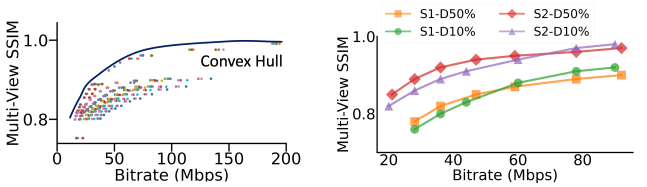
4.1 Rate Control Problem Formulation

Traditionally (e.g., in 2D video), the streaming quality is optimized for a given target bitrate using a rate-distortion function: $\mathscr{L} = \mathscr{D} + \lambda\mathscr{R}$, where $\mathscr{R}$ is the bitrate, $\mathscr{D}$ is the distortion caused after compression and λ is a tuning parameter to trade video quality with bitrate. Naturally, this function can be extended to 3D rate control as: $\mathscr{L} = \mathscr{D} + \lambda_t\mathscr{R}_t + \lambda_g\mathscr{R}_g$, where $\mathscr{R}_t$ and $\mathscr{R}_g$ represent the bitrates for texture and geometry respectively. Unlike point clouds, where colors are baked into the same data structure, texture and mesh are two independent data structures. Thus, they need to go through separate compression pipelines. Note that our goal is to maximize the final rendered quality, given the combination of texture and geometry bitrates. This requires adjusting λ_t and λ_g to minimize the overall distortion. Therefore, the rate control problem for 3D traffic requires us to answer two key questions: *(1) how do we divide the target bitrate (i.e., available bandwidth) between texture and geometry, and (2) what are the coding parameters for both texture and geometry to produce optimal quality for a given bitrate.*

4.2 Rate Control Model

A natural solution to find the optimal rate-distortion curve (i.e., bitrate vs. quality) is to explore all possible coding parameters that produce the best quality for a given bitrate offline and use those parameters to generate suitable bitrates online based on the available bandwidth. The relevant coding parameters include resolutions, encoding levels (e.g., quantization parameter) for texture and mesh, and an additional decimation parameter for mesh. Figure 8a shows an example of a rate-distortion curve for one scene that is decimated and encoded at different levels and resolutions. Each point in the plot denotes a combination of <resolution, decimation, compression> level, and different combinations are the best fit for different bitrates. The points corresponding to these best parameters at different bitrates can form a convex curve that represents an optimal Pareto frontier of coding efficiency.

Having a 'one-size-fits-all' Pareto frontier curve for any scene is ideal when preparing the bitstream for fine-grained rate adaptation over a wireless network. However, the rate-distortion curves are a



(a) Convex curve representing optimal bitrate vs. quality pairs. (b) Bitrate vs. quality for two scenes with different complexity.

Figure 8: Impact of coding parameters on bitrate vs. quality for textured mesh: (a) exploration space with optimal rate distortion curve, (b) Rate distortion with two scenes (S1, S2) with two decimation levels (10%, 50%) encoded at different bitrates. For the same bitrates, S1 and S2 produce different qualities, and hence they need different coding parameters to produce same bitrates at same optimal quality.

function of the underlying scene complexity, and the optimal coding parameters for one scene can be suboptimal for others (Figure 8b). A naive approach to deal with this is computing the Pareto frontier by exploring the entire state space through exhaustive search. However, exhaustive search is an extremely compute-intensive task because it involves exploring thousands of coding parameters to find the best, creating a major challenge for finding optimal rates in real time.

To address the above challenge, we propose a low-complexity feature-based predictive model for rate control. We formulate the problem as a multi-objective classification problem, where we train an offline model that can predict optimal coding parameters for texture and mesh online (i.e., during live streaming). The coding parameters are multiple classes, and each parameter value needs to be classified to produce the suitable bitrate. Given the wide variety of prediction mechanisms, particularly in the machine learning space, a key design decision we need to make is which algorithm to use— we can use either a simple and lightweight but slightly less accurate model (e.g., SVM classifier), or use a more accurate but more compute expensive (e.g., neural network style) model for prediction. Given the real-time constraints of our live application scenario, we adopt the former approach of simple ML models in order to be lightweight. However, unlike neural networks, which automatically extract features from raw frames, training these models requires manual feature extraction that again brings computation-related challenges that we describe below.

Low Complexity Feature Extraction: In order for the end-to-end feature extraction and prediction to be achieved in under a few milliseconds, we need features to be computationally low complexity. To this end, we adopt a simple frequency distribution based features as a proxy to represent a scene signature. Specifically, we use the below Discrete Cosine Transform (DCT) energy function from [26] to compute spatial *information* in each frame (both for depth and RGB texture frames).

$$\mathscr{E}_{dct} = \sum_{i=w}^{j=h} e^{[(\frac{ij}{wh})^2-1]} |DCT(i-1, j-1)|$$

where, where w and h are the width and height of each block, and $DCT(i,j)$ is the $(i,j)^{th}$ DCT component when $i+j>2$, and 0 otherwise [26]. Figure 9 shows an example of different scenes with different complexity and the corresponding DCT energy of the scene in each frame. We find that DCT energy tends to be high for complex scenes and vice versa, and hence, it can be used as a rich feature in classifying the coding parameters. We also compute DCT energy temporally (to accommodate for temporal redundancy), same as the approach outlined in [36], and other features such as variance and gradient of temporal energy (to capture motion), and along with the target bitrate as features to train the classifier.

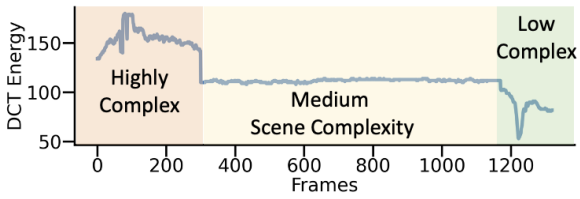


Figure 9: DCT energy as a feature to measure scene complexity. A low DCT energy indicates a low complexity scene and vice versa.

Joint Rate Allocation Model: Given the raw RGB-D frames and the target bitrate, the goal is to select a set of coding parameters such that the bitrates generated by texture and mesh using those coding parameters sum up to the target bitrate. Here, we use coding parameters resolution and compression levels for texture and an additional decimation level for mesh. The inputs for the model are the above features extracted from the depth and RGB frames, along with the target bitrate, and the output is predicted coding parameters for texture and mesh. A key advantage of jointly training the classifier to predict the parameters for both texture and mesh is that it eliminates the need to manually tune λ_t and λ_g (as described in §4.1) to split the bandwidth between the two. The classifier model outputs parameters for both.

Our current implementation shows a simple proof-of-concept that rate control is a non-trivial multi-dimensional problem. We provide one such predictive solution, but there is tremendous room for improvement (in terms of scene features and more fine-grained parameter control) for the textured mesh streaming systems that we leave for future work.

5 Implementation and System Setup

`MeshReduce`'s end-to-end system (Figure 5) consists of capturing, mesh merging, and streaming to prepare a 3D video bitstream. It also involves a rendering server on the client side to perform viewport-based remote rendering. The procedure of the pipeline is (1) sensor-side compute nodes reconstruct, decimate, and encode the mesh and an RGB video to the hierarchical merging server, (2) the hierarchical merging server processes all the sensor node partial meshes and forms a texture atlas, (3) the merging server uses rate control model to generate adaptive bitrates for texture and geometry streams for the Internet delivery, (4) client-side rendering server receives a unified 3D scene and remotely renders view based on client viewport.

We built `MeshReduce` in C++ on top of Open3D [45], Intel Embree [56], Google Draco [2], WebRTC [9]. We use four Azure Kinect cameras [1] for high-resolution capture, recording color and depth frames at 3840x2160 and 640x576 at 30FPS. For wide area capture, we deploy the OS-Dome LiDAR [42] (2048x128 resolution) combined with a 360° camera (Ricoh Theta X [46]), as shown in Figure 10. Kinect sensors are calibrated with stereo calibration. LiDAR and camera are calibrated with line and plane correspondences [62].

For each sensor side compute node, we use a Jetson Orin Nano embedded computer to perform per-camera mesh operations. We use Google Draco [2] for mesh compression, and `h264_nvenc` [44] for color/texture video compression. Each node streams color and mesh bitstream to the merging server over a TCP connection.

The merging servers are Linux machines with an AMD 5950x CPU, 64GB RAM, and NVIDIA RTX 3090 GPU that receive color and mesh bitstreams from separate sensors. Once the bitstreams are received, the mesh is decoded using Draco [2] for mesh merging, and the color frames are decoded using `h264_nvdec` [44] to prepare a texture atlas. Once the merged scene is ready with both mesh and texture atlas, the mesh is again decimated and compressed with Draco [2], and the texture atlas is compressed with `h264_nvenc` [44]. The resulting product is an internet-friendly texture mesh bitstream.

Additional System Optimization: We implement local parallel decimation and compression on each sensor compute node to reduce the latency. We do this using our split-merge design idea without modifying the decimation or compression algorithms, where we locally decompose the depth maps into M depth patches according to the machine core count and reconstruct a mesh out of each patch on a separate core. We then parallel decimate, compress, and stream each reconstruction to the hierarchical merging server. We evaluate the quality vs latency impact of such local decomposition in §6.2.

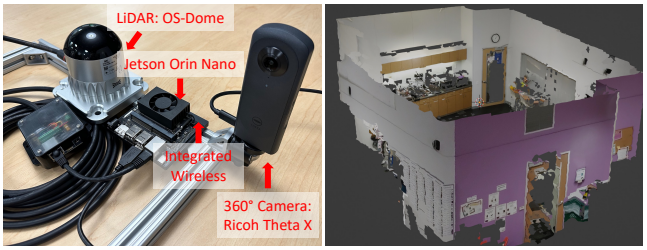


Figure 10: Left shows `MeshReduce`'s custom LiDAR and Camera setup to enable wide-area capture. Right side shows a large area (8 x 5 meter) 3D capture with this setup mounted on the ceiling.

Figure 11: A qualitative result of two high-resolution people reconstructed in 3D from our testbed (using 4 RGB-D cameras), composited with a photogrammetry background, and rendered from three arbitrary viewpoints.

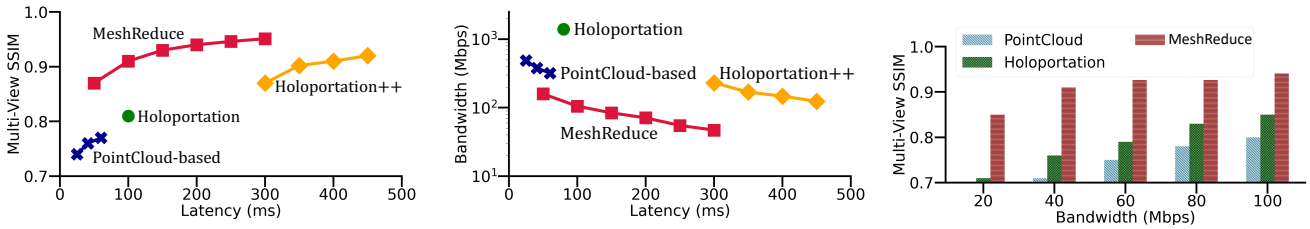
6 Evaluation

6.1 Evaluation Methodology

Dataset: We collect several samples of 3D video under different environments (e.g., conference room, office space, corridor, and hallways) by capturing RGB-D frames from multiple sensors. We construct point cloud and textured mesh frames from the synchronized RGB-D frames. To evaluate the quality after introducing coding artifacts, we create reference frames with a 1cm voxel resolution to extract mesh from the depth frames. Figure 11 shows an example of a rendered scene, composited from two people captured from our four camera testbed in a lab area, with a pre-scanned photogrammetry model as background, using our mesh merging algorithm. We do not apply color correction or perform any smoothing between the mesh segments, but there are well known graphics techniques that can be used to improve the final image quality.

Evaluation Metrics: Our key evaluation metrics are the system's final rendered quality, bitrate, and latency. We evaluate quality using a Multi-View SSIM metric that computes quality from multiple viewpoints to cover the 3D scene. Latency is the time needed to generate mesh bitstreams starting from the RGB-D frame acquisition.

3D Quality Evaluation: Unfortunately, there are no well-defined metrics for 3D visual quality assessment. We propose adapting SSIM (structural similarity index metric) [58], a popular 2D video quality metric that measures the perceptual quality difference between two videos. We compute the SSIM of the 2D rendering of the 3D scene from a predefined set of views. We refer to our metric as `Multi-View SSIM`, which reports the average SSIM across the predefined viewpoints. The views are selected based on Voronoi path planner [8] to avoid locations and angles that do not accurately capture useful locations for viewers (e.g., under a table, looking into



(a) Latency vs. Quality (b) Latency vs. Bandwidth (c) Bandwidth vs. Quality

Figure 12: Latency, Quality, and Bandwidth trade-offs of `MeshReduce` and its benefits compared to baseline streaming systems. `MeshReduce` achieves significantly higher quality while maintaining low latency and bandwidth. Points on the graphs are the chosen experiment latencies.

a corner, etc.). Our collected datasets come with a predefined set of viewpoints (on average ≈30).

Baselines. We implemented and compared against the following variants of existing work[1].

- **Holoportation [41],** a state-of-the-art textured mesh streaming system for real-time 3D telepresence. It uses lightweight compression (LZ4 [5]) for texture and geometry streaming.
- **PointCloud-based,** a live point cloud based 3D reconstruction system. The closest similar system is LiveScan3D [27]; however, it does not discuss much about compression and streaming. We apply MPEG VPCC [6] compression when comparing the rate-distortion results. The majority of the volumetric video streaming work [23, 32, 61] also falls under this category.
- **Holoportation++,** a custom-designed and optimized version of the original Holoportation system. The Holoportation system does not use efficient compression mechanisms for both texture and geometry. We introduce Edgebreaker (Draco) mesh compression along with parallel decimation and `h264_nvenc` for texture compression to Holoportation and call it Holoportation++. This is distinct from `MeshReduce` as it does not distribute mesh creation followed by the merging step.

6.2 End-to-End Results

Figure 12 shows the end-to-end scene capture latency against the quality and bandwidth requirement of `MeshReduce` compared with its alternatives. We fix the bandwidth at 100 Mbps in Figure 12a by choosing different coding parameters for each method that result in different latency and quality because of their respective reconstruction pipeline and data representation. Under this setting, `MeshReduce` improves the quality by 18% and 27% when compared to state-of-the-art Holoportation and Pointcloud-based 3D streaming systems. This translates to 0.1 and 0.15 SSIM, which is a significant improvement given that even a 0.05 SSIM value can show a considerably noticeable difference in terms of perceived visual quality [58]. On the other hand, the optimized version of Holoportation performs close to `MeshReduce` but at the expense of 4× higher latency.

Figure 12b shows the bandwidth consumption for each method. Here, we fix the quality for each method at ≈0.92 average Multi-View SSIM, which results in different latency and bandwidth requirements. `MeshReduce` requires 40× and 3× less bandwidth compared to Holoportation and point cloud based systems. Similar to earlier, the bandwidth requirement of Holoportation++ is close to `MeshReduce` but suffers significantly from latency because of its monolithic mesh reconstruction as well as the decimation on the entire mesh model at once. Figure 12c shows the quality achieved for each system at different bandwidths for a latency of 100ms. `MeshReduce` can have perceivable quality (i.e., above 0.8 SSIM value) even at as low bitrates as 20 Mpps, while providing higher quality as we increase the bandwidth. On the other hand, both Holoportation and point cloud systems are below the perceivable SSIM threshold (i.e., less than 0.8) under 60 Mbps.

At a given latency, a significant part of the improvements of `MeshReduce` is from mesh decimation, effectively reducing the required bandwidth while not affecting the quality when compared with Holoportation. While the Holoportation++ system does have decimation, it does not use the sensor side compute node and suffers from high latency. On the other hand, the point cloud based system suffers mainly because of its compression inefficiency.

Latency Breakdown on Different Edge Devices: Table 1 shows `MeshReduce`'s sensor side performance with different platforms. We demonstrate `MeshReduce` can effectively distribute capture workloads and run on Jetson embedded platforms under 100ms latency. It is important to note that pipelining the compute stages can further improve system throughput.

Table 1: `MeshReduce` latency on different platforms (ms)

Compute Stages	Jetson Orin Nano	Jetson AGX Orin	Desktop i9 RTX 3070
Mesh Reconstruction	15	10	2
Texture Processing	7	4	3
Texture Encoding	27	12	8
Mesh Encoding	10	8	3
Mesh Decimation	32	20	11

Ablation Study: Figure 13 shows the impact of `MeshReduce`'s components at fixed 100 Mbps bitrate and 100ms latency: (1) `MeshReduce` without texture atlas optimization (TAO) from §3.3, and (2) `MeshReduce` without the local depth decomposition at the sensor for parallel decimation. The figure shows that both components are critical to the performance of `MeshReduce`. `MeshReduce` without TAO performs poorly compared to full `MeshReduce` because of the compression inefficiency of spatially incohesive texture atlas. On the other hand, `MeshReduce` without local decomposition has to trade with lightweight decimation to achieve the same latency and bitrate, resulting in poor quality.

Impact of Local Decomposition: The multi-core parallelization of decimation by locally decomposing the mesh reconstruction decreases latency significantly but also has an impact on quality. Figure 14 shows the latency and quality loss due to the decomposition and merging when compared with the single reconstruction. As we increase the decomposition size, the latency decreases significantly. However, we observe a noticeable quality of around 0.02 SSIM only after a decomposition of more than 5×5 decomposition size. This shows an interesting trade-off between quality and latency to decimate the geometry for providing adaptive bitrates.

[1] Other related work, such as viewport adaptive 3D streaming [23], focuses on rate adaptation algorithms and are slightly orthogonal to our work since we mainly focus on capturing and preparing bitstreams efficiently. `MeshReduce` can be used in these solutions for the capture side.

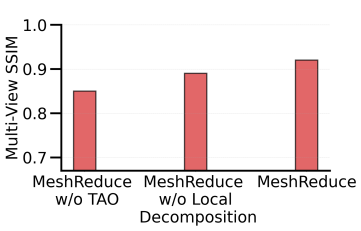


Figure 13: MeshReduce's individual components performance breakdown.

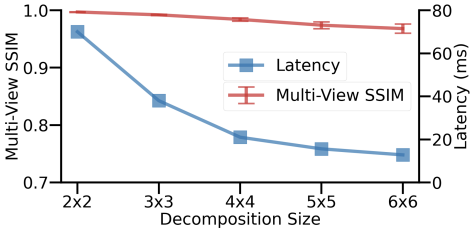


Figure 14: Impact of multi-core local decomposition on quality and latency.

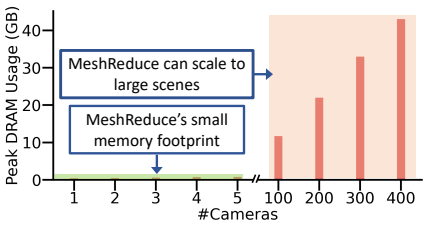


Figure 15: MeshReduce is DRAM bounded and can scale to a large number of cameras.

6.3 Scalability with Large Scenes

A key outcome of MeshReduce is that it eliminates the GPU memory bottleneck by effectively splitting the mesh reconstruction task. Each sensor-side compute node has enough GPU memory to reconstruct its own view (since our maximum per-sensor scene GPU memory requirement is 3.5 GB, see §2). While the merging server still processes all the per-sensor reconstructions, the merging step can be bound only by CPU and DRAM (though GPU-based raycasting approaches can further reduce latency). Since the commodity devices are often shipped with large amounts of DRAM, MeshReduce's reconstruction pipeline can scale well to many challenging scene capture scenarios — large-scale scenes, many sensors, etc. Figure 15 shows an experimental result on the scaling limits of MeshReduce with synthetic scenes (each of 6m×9m) captured with several cameras at 1cm voxel resolution. For small-scale scenes (i.e., up to 5 cameras), the DRAM usage is as small as under 2 GB. However, even at 400 cameras, the DRAM usage is around only 43.2 GB, which is only a fraction of the DRAM capacity even on today's server devices. Supporting such large scenes demonstrates that MeshReduce is efficient for room-scale 3D telepresence applications and enables campus-wide building-scale remote exploration of spaces.

While MeshReduce supports a large number of sensor scenes, one caveat is the computational complexity of merging tasks. In our experiments, we observe a merging latency of 1.2 seconds (mainly from the raycasting task) when we scale to 400 cameras. However, this would be mitigated with our hierarchical merging design, and MeshReduce's real-world evaluation focus is streaming small-scale 3D scenes with low bitrates and low latency, and hence we leave optimizing of latency for large scenes/sensors for future work. Existing high complex scene rendering solutions such as R2E2 [18] can be applied to render such large scenes with low latency.

6.4 Rate Control Sensitivity Analysis

We compare the rate-distortion performance of MeshReduce with an offline generated Pareto-optimal rate-distortion curve over ten scenes from our dataset. We first generate the optimal curves by selecting the best coding parameters as described in §4. Then, we use MeshReduce's rate control model to predict the coding parameters and generate rate-distortion curves with the predicted parameters. For each scene, we compute the SSIM difference between the quality of the optimal curve and our predicted curve for different bitrates (ranging from 10-100 Mbps), and plot the distribution of the difference in SSIM. We also show the SSIM difference for an alternative lookup-table solution; the coding parameters can be profiled for different scenes with different DCT energy and stored in the table for online lookup. Figure 16 shows a cumulative distribution of the difference in SSIM for each model compared against the optimal rate-distortion curve. MeshReduce can achieve within 0.15 SSIM of the Pareto-optimal curve with an average of 0.08 SSIM difference. This is noticeably better than the lookup table, which has an average SSIM difference of 0.11 and a maximum difference of 0.2 SSIM. The lookup-table suffers mainly because it is difficult to store the coding parameters for all possible scene dynamics accurately.

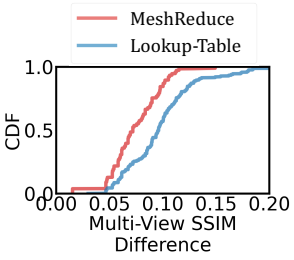


Figure 16: MeshReduce's rate control model performance.

6.5 Hardware and Cost Analysis

This section discusses the hardware cost analysis of centralized capture systems and our proposed MeshReduce. We analyze small and large conference rooms that need 4 and 8 capturing sensors. In the centralized systems, the central server needs server-class GPUs that support up to 40GB of memory for 4 sensors (see Figure 3b) and 80GB of memory for 8 sensors. Therefore, they incur a high cost with GPUs such as (e.g., NVIDIA A100). On the other hand, MeshReduce leverages distributed compute located at the capturing sensors so that the final computation requirements at the hierarchical merging server are significantly lowered and need only consumer-grade servers and GPUs (e.g., NVIDIA RTX 3090). It is worth noting that the system might have dozens or even hundreds of cameras in VFX production workflows. In these cases, a centralized approach is totally infeasible and requires lengthy offline post-processing. MeshReduce could still support these types of applications in real-time with marginal additional cost, given that these systems are often on the order of hundreds of thousands of dollars.

7 Conclusion and Future Work

In summary, this paper introduces MeshReduce, an open-source 3D scene capture system that fuses RGB-D streams from network-connected sensors in real-time. Our key insight is that independently created mesh reconstructions can be merged incrementally in a distributed manner instead of at one centralized source without a significant loss in quality. Our approach leads to a more compact intermediate representation of the 3D data. MeshReduce's texture mesh excels in capturing planar surfaces and managing geometry and texture data separately, enhancing adaptability to network changes.

Future directions include developing rate adaptation algorithms for 3D streaming, using progressive meshes [25] and texture mipmaps [38], akin to layered video streaming [3]. This would allow gradual improvement in geometry detail based on compute/network capacity. Additionally, we can adopt viewport prediction for streaming user viewport [23], and explore 3D video adaptive streaming to accommodation-supporting interactive 3D displays [43].

Finally, there is a large body of work on light-field 3D reconstruction [57] and a rapidly growing community exploring neural scene reconstruction [37] techniques. We believe our distributed architecture could be applied to both by adapting our merge function.

Acknowledgments

This work was supported by the National Science Foundation under the award CNS-1956095, Bosch Corporate Research, and CMU Manufacturing Futures Institute.

REFERENCES

[1] Azure Kinect DK. https://azure.microsoft.com/en-us/services/kinect-dk/. Online. Accessed: May 2022.

[2] Google Draco. https://github.com/google/draco. Online. Accessed: Sep 2022.

[3] H.264/AVC scalability extension. https://avc.hhi.fraunhofer.de/svc.

[4] libvpx-vp9. https://trac.ffmpeg.org/wiki/Encode/VP9.

[5] LZ4. https://github.com/lz4/lz4. Online. Accessed: Sep 2022.

[6] MPEG Point Cloud Compression. https://mpeg-pcc.org/. Online. Accessed: Sep 2022.

[7] Y. Alj, G. Boisson, P. Bordes, M. Pressigout, and L. Morin. Multi-texturing 3d models: how to choose the best texture? In *2012 International Conference on 3D Imaging (IC3D)*, pp. 1–8. IEEE, 2012.

[8] P. Bhattacharya and M. L. Gavrilova. Roadmap-based path planning - using the voronoi diagram for a clearance-based shortest path. *IEEE Robotics & Automation Magazine*, 15(2):58–66, 2008. doi: 10.1109/MRA.2008.921540

[9] N. Blum, S. Lachapelle, and H. Alvestrand. Webrtc: Real-time communication for the open web platform. *Communications of the ACM*, 64(8):50–54, 2021.

[10] I. Cheng and P. Boulanger. Adaptive online transmission of 3-d texmesh using scale-space and visual perception analysis. *IEEE transactions on multimedia*, 8(3):550–563, 2006.

[11] A. Collet, M. Chuang, P. Sweeney, D. Gillett, D. Evseev, D. Calabrese, H. Hoppe, A. Kirk, and S. Sullivan. High-quality streamable free-viewpoint video. *ACM Transactions on Graphics (ToG)*, 34(4):1–13, 2015.

[12] B. Curless and M. Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 303–312, 1996.

[13] K. Desai, K. Bahirat, S. Raghuraman, and B. Prabhakaran. Network adaptive textured mesh generation for collaborative 3d tele-immersion. In *2015 IEEE International Symposium on Multimedia (ISM)*, pp. 107–112. IEEE, 2015.

[14] O. Devillers and P.-M. Gandoin. Geometric compression for interactive transmission. In *Proceedings Visualization 2000. VIS 2000 (Cat. No. 00CH37145)*, pp. 319–326. IEEE, 2000.

[15] W. Dong, J. Shi, W. Tang, X. Wang, and H. Zha. An efficient volumetric mesh representation for real-time scene reconstruction using spatial hashing. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6323–6330. IEEE, 2018.

[16] M. Dou, S. Khamis, Y. Degtyarev, P. Davidson, S. R. Fanello, A. Kowdle, S. O. Escolano, C. Rhemann, D. Kim, J. Taylor, P. Kohli, V. Tankovich, and S. Izadi. Fusion4d: Real-time performance capture of challenging scenes. *ACM Trans. Graph.*, 2016.

[17] R. Du, M. Chuang, W. Chang, H. Hoppe, and A. Varshney. Montage4d: Interactive seamless fusion of multiview video textures. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. Association for Computing Machinery, New York, NY, USA, 2018. doi: 10.1145/3190834.3190843

[18] S. Fouladi, B. Shacklett, F. Poms, A. Arora, A. Ozdemir, D. Raghavan, P. Hanrahan, K. Fatahalian, and K. Winstein. R2e2: low-latency path tracing of terabyte-scale scenes using thousands of cloud cpus. *ACM Transactions on Graphics (TOG)*, 41(4):1–12, 2022.

[19] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pp. 209–216, 1997.

[20] D. Graziosi, O. Nakagami, S. Kuma, A. Zaghetto, T. Suzuki, and A. Tabatabai. 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:e13, 2020.

[21] Y. Guan, X. Hou, N. Wu, B. Han, and T. Han. Metastream: Live volumetric content capture, creation, delivery, and rendering in real time. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pp. 1–15, 2023.

[22] S. N. Gunkel, R. Hindriks, K. M. E. Assal, H. M. Stokking, S. Dijkstra-Soudarissanane, F. t. Haar, and O. Niamut. Vrcomm: an end-to-end web system for real-time photorealistic social vr communication. In *Proceedings of the 12th ACM Multimedia Systems Conference*, pp. 65–79, 2021.

[23] B. Han, Y. Liu, and F. Qian. Vivo: Visibility-aware mobile volumetric video streaming. In *Proceedings of the 26th annual international conference on mobile computing and networking*, pp. 1–13, 2020.

[24] A. Hilton, A. J. Stoddart, J. Illingworth, and T. Windeatt. Reliable surface reconstruction from multiple range images. In *Computer Vision—ECCV'96: 4th European Conference on Computer Vision Cambridge, UK, April 15–18, 1996 Proceedings, Volume I 4*, pp. 117–126. Springer, 1996.

[25] H. Hoppe. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 99–108, 1996.

[26] M. King, Z. Tauber, and Z.-N. Li. A new energy function for segmentation and compression. In *2007 IEEE International Conference on Multimedia and Expo*, pp. 1647–1650. IEEE, 2007.

[27] M. Kowalski, J. Naruniec, and M. Daniluk. Livescan3d: A fast and inexpensive 3d data acquisition system for multiple kinect v2 sensors. In *2015 international conference on 3D vision*, pp. 318–325. IEEE, 2015.

[28] A. Kreskowski, S. Beck, and B. Froehlich. Output-sensitive avatar representations for immersive telepresence. *IEEE Transactions on Visualization and Computer Graphics*, 28(7):2697–2709, 2020.

[29] O. Kähler, V. Prisacariu, J. Valentin, and D. Murray. Hierarchical voxel block hashing for efficient integration of depth images. *IEEE Robotics and Automation Letters*, 1(1):192–197, 2016. doi: 10.1109/LRA.2015.2512958

[30] J. Lawrence, D. B. Goldman, S. Achar, G. M. Blascovich, J. G. Desloge, T. Fortes, E. M. Gomez, S. Häberling, H. Hoppe, A. Huibers, C. Knaus, B. Kuschak, R. Martin-Brualla, H. Nover, A. I. Russell, S. M. Seitz, and K. Tong. Project starline: A high-fidelity telepresence system. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, 40(6), 2021.

[31] K. Lee, J. Yi, and Y. Lee. Farfetchfusion: Towards fully mobile live 3d telepresence platform. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pp. 1–15, 2023.

[32] K. Lee, J. Yi, Y. Lee, S. Choi, and Y. M. Kim. Groot: a real-time streaming system of high-fidelity volumetric videos. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pp. 1–14, 2020.

[33] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21(4):163–169, 1987.

[34] A. Maimone and H. Fuchs. Encumbrance-free telepresence system with real-time 3d capture and display using commodity depth cameras. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pp. 137–146. IEEE, 2011.

[35] A. Maimone and H. Fuchs. Real-time volumetric 3d capture of room-sized scenes for telepresence. In *2012 3DTV-Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON)*, pp. 1–4, 2012. doi: 10.1109/3DTV.2012.6365430

[36] V. V. Menon, C. Feldmann, H. Amirpour, M. Ghanbari, and C. Timmerer. Vca: Video complexity analyzer. In *Proceedings of the 13th ACM multimedia systems conference*, pp. 259–264, 2022.

[37] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng. Nerf: Representing scenes as neural radiance fields for view synthesis, 2020. doi: 10.48550/ARXIV.2003.08934

[38] J. H. Mueller, P. Voglreiter, M. Dokter, T. Neff, M. Makar, M. Steinberger, and D. Schmalstieg. Shading atlas streaming. *ACM Transactions on Graphics (TOG)*, 37(6):1–16, 2018.

[39] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE international symposium on mixed and augmented reality*, pp. 127–136. Ieee, 2011.

[40] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3d reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (ToG)*, 32(6):1–11, 2013.

[41] S. Orts-Escolano, C. Rhemann, S. Fanello, W. Chang, A. Kowdle, Y. Degtyarev, D. Kim, P. L. Davidson, S. Khamis, M. Dou, et al.

Holoportation: Virtual 3d teleportation in real-time. In *Proceedings of the 29th annual symposium on user interface software and technology*, pp. 741–754, 2016.

[42] Ouster. Ouster Dome LiDAR Website. `https://ouster.com/products/hardware/osdome-lidar-sensor`. Online. Accessed: Sep 2023.

[43] Y. Qin, W.-Y. Chen, M. O'Toole, and A. C. Sankaranarayanan. Split-lohmann multifocal displays. *ACM Transactions on Graphics (TOG)*, 42(4):1–18, 2023.

[44] NVIDIA. NVIDIA Video Codec SDK Website. `https://developer.nvidia.com/nvidia-video-codec-sdk`. Online. Accessed: Sep 2022.

[45] Open3D. Open3D Website. `http://www.open3d.org/`. Online. Accessed: Sep 2022.

[46] Ricoh. Ricoh Theta X Camera Website. `https://us.ricoh-imaging.com/product/theta-x/`. Online. Accessed: Sep 2023.

[47] Volograms. Volograms. `https://www.volograms.com/`. Online. Accessed: Sep 2022.

[48] G. Rendle, A. Kreskowski, and B. Froehlich. Volumetric avatar reconstruction with spatio-temporally offset rgbd cameras. In *2023 IEEE Conference Virtual Reality and 3D User Interfaces (VR)*, pp. 72–82. IEEE, 2023.

[49] J. Rossignac. 3d compression made simple: Edgebreaker with zipand-wrap on a corner-table. In *Proceedings International Conference on Shape Modeling and Applications*, pp. 278–283. IEEE, 2001.

[50] C. Schröder, M. Sharma, J. Teuber, R. Weller, and G. Zachmann. Dyncam: A reactive multithreaded pipeline library for 3d telepresence in vr. In *Proceedings of the Virtual Reality International Conference-Laval Virtual*, pp. 1–8, 2018.

[51] P. Stotko, S. Krumpen, M. B. Hullin, M. Weinmann, and R. Klein. Slamcast: Large-scale, real-time 3d reconstruction and streaming for immersive multi-client live telepresence. *IEEE transactions on visualization and computer graphics*, 25(5):2102–2112, 2019.

[52] D. Tang, M. Dou, P. Lincoln, P. Davidson, K. Guo, J. Taylor, S. Fanello, C. Keskin, A. Kowdle, S. Bouaziz, et al. Real-time compression and streaming of 4d performances. *ACM Transactions on Graphics (TOG)*, 37(6):1–11, 2018.

[53] D. Tang, S. Singh, P. A. Chou, C. Hane, M. Dou, S. Fanello, J. Taylor, P. Davidson, O. G. Guleryuz, Y. Zhang, et al. Deep implicit volume compression. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 1293–1303, 2020.

[54] D. Tian and G. AlRegib. Progressive streaming of textured 3d models over bandwidth-limited channels. In *Proceedings.(ICASSP'05). IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, vol. 2, pp. ii–1089. IEEE, 2005.

[55] G. Turk and M. Levoy. Zippered polygon meshes from range images. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pp. 311–318, 1994.

[56] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree: a kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (TOG)*, 33(4):1–8, 2014.

[57] T.-C. Wang, J.-Y. Zhu, N. K. Kalantari, A. A. Efros, and R. Ramamoorthi. Light field video capture using a learning-based hybrid imaging system. *ACM Trans. Graph.*, 36(4), jul 2017. doi: 10.1145/3072959.3073614

[58] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[59] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

[60] L. Xu, Z. Su, L. Han, T. Yu, Y. Liu, and L. Fang. Unstructuredfusion: Realtime 4d geometry and texture reconstruction using commercial rgbd cameras. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 42(10):2508–2522, 2020. doi: 10.1109/TPAMI.2019.2915229

[61] A. Zhang, C. Wang, B. Han, and F. Qian. {YuZu}:{Neural-Enhanced} volumetric video streaming. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 137–154, 2022.

[62] L. Zhou, Z. Li, and M. Kaess. Automatic extrinsic calibration of a camera and a 3d lidar using line and plane correspondences. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 5562–5569. IEEE, 2018.

[63] W. Zielonka. Parallelqslim. `https://github.com/Zielon/ParallelQSlim`, 2021.