MeshReduce: A Distributed Scene Capture Architecture for 3D Telepresence

Mallesham Dasari[1], Tao Jin[1], Connor Smith[2], Anthony Rowe[1], Srinivasan Seshan[1]
Carnegie Mellon University[1], Magic Leap[2]

ABSTRACT

Live volumetric video streaming enables a remote viewer to observe a 3D scene from any angle or location. However, current solutions incur high latencies, consume significant bandwidth, and scale poorly with the number of cameras and size of scenes. These problems are largely caused by the current monolithic approach to scene capture and the use of inefficient data structures for streaming (e.g. RGB-D and Point Cloud). This paper introduces MeshReduce, a distributed scene capture system that advocates for the use of textured mesh data representation for volumetric video streaming. Textured meshes are compact and can provide lower bitrates for the same quality compared to other volumetric data formats. However, using textured meshes creates compute and memory challenges. 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 camera streams. While this enables a more efficient implementation, this approach requires efficient exchange of textured meshes across the network. MeshReduce also incorporates a novel approach to dividing the bandwidth between texture and mesh for efficient, adaptive volumetric video streaming.

1 INTRODUCTION

Recent advances in high performance hardware, better depth sensing technology, and advanced graphics algorithms have brought us closer to practical real-time volumetric (3D) video streaming systems. A core component of existing systems [24, 34, 39] 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 multiple viewpoints. These multiple 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 (users can explore the scene not only by changing a view direction,as supported by a flat 360° video, but can also change a position to move around in the 3D space.)

In this paper, we present MeshReduce, a 3D scene capture system designed to support live volumetric video streaming applications. Key requirements for such a system include:

(1) **Low Latency.** For interactive live streaming, we expect latencies on the order of 2D video conferencing systems (below 100ms).
(2) **Scalable.** Volumetric video quality is often a function of number of cameras and scene size. An ideal capture solution should support dozens of cameras 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 fall into two categories: ones that output point cloud [19, 22, 25] representations and ones that output textured meshes [10, 13, 41]. Prior work often gravitates towards point cloud systems because the data is closes to the native output of the sensor and relatively straightforward to scale. This is often suitable for industrial applications that are more concerned with structural accuracy compared to visual quality. For volumetric streaming applications, clients are forced to perform the more complex task of creating visually convincing rendered scenes. In contrast, textured meshes[1] yield themselves more naturally to realistic rendering on the client device, but require additional processing steps during creation.

A significant trade-off between these approaches, that we quantify in §2.2, is that *mesh representation requires less data to store and transmit compared to point cloud formats for a given quality*. This is due to two key reasons: **(1)** mesh representations naturally capture and compactly approximate planar features that are commonly found in real-world scenes, and **(2)** textured mesh representation makes it easy to decouple geometry data from texture data for efficient 3D bitstream preparation. For these reasons, MeshReduce advocates for adopting textured meshes as an intermediate data structure for streaming (more details in §2).

[1]A textured mesh consists of a 1) geometric **mesh** which is a set of polygons that defines the shape of an object, 2) a **texture** which contains color (and optionally other attributes such as transparency, reflectivity, etc.)

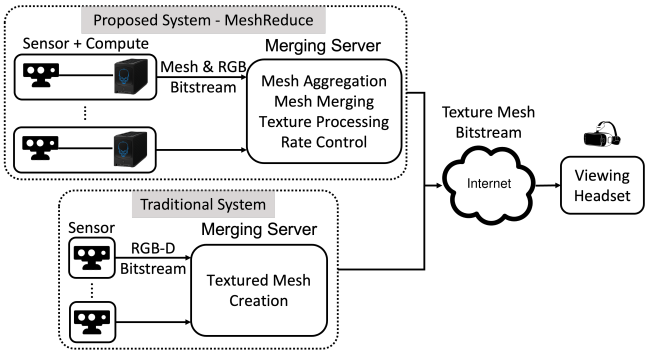


Figure 1: Existing vs proposed system. Unlike existing centralized 3D scene capture architecture, MeshReduce adopts a distributed approach where each camera is equipped with a compute node processing per-camera specific 3D scene, with a central node merging partial reconstructions, and prepares optimal 3D textured mesh bitstreams for Internet transmission.

However, streaming textured meshes is not without challenges (§2.3). For example, 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 pipeline. Existing systems [14, 34] rely on centralized monolithic scene capture pipelines and, as a result, incur extremely high data rates (e.g., Holoportation requires 1-2 Gbps bandwidth for each scene [34]) or take significant time (seconds or minutes) to create textured meshes (Figure 1). Existing designs scale poorly with scene size and camera count due to the high compute and GPU memory demands which in turn limits their capture quality (existing systems [34] support less than a dozen cameras while most systems need more than a few tens of cameras for high quality scenes [10]). More importantly, there are no existing flexible rate control strategies for preparing textured mesh bitstreams for Internet transmission.

Instead, MeshReduce (top of Figure 1) employs a distributed approach to scene capture and overcomes the above limitations. MeshReduce deploys compute nodes close to cameras to perform per-camera reconstructions. A merging server (deployed within the capture environment) aggregates all the per-camera reconstructions and merges them into a single mesh. This approach eliminates the compute and memory bottlenecks present in a centralized design since operations are on smaller (per-camera specific) scene. While distributing the scene capture pipeline to many compute nodes is an appealing strategy, realizing MeshReduce end-to-end requires us to answer address several nontrivial systems challenges: *(i) Decomposing scene reconstruction* (§3.2): Creating a textured mesh and generating an efficient bitstream involves several key steps. We must determine where to split this computation pipeline between the camera-side compute node and centralized merging server. A poor choice can lead to inefficient use of compute resources and high latency overheads. In addition, we need to seamlessly merge all the per-camera reconstructions to ensure good quality across the split regions and remove cross-camera overlapping regions; *(ii) Bandwidth-efficient texture processing* (§3.3): Streaming the texture component of a textured mesh is more straightforward than mesh geometry because of its similarity to traditional 2D video content. However, much like merging partial reconstructions into one scene, we need techniques to remove redundant regions from multiple cameras (i.e., creating a panorama of all the camera views) before compression and streaming. Inefficiently packing textures can lead to a spatially unstructured texture, and consequently inefficient compression.

The above challenges focus on the computation stages of the capture pipeline. An important additional challenge we address is: *(iii) Rate control for texture and geometry*[2] (§4): The streaming of textured meshes between nodes of our system and from our capture system to Internet clients must adapt to variable bandwidth. For this, MeshReduce answers two key questions— 1) given a target bitrate, how much of it should be allocated for texture and for mesh, and 2) how to select optimal coding parameters (e.g., resolution, compression level) for both texture and mesh that strikes a good balance between the quality and bitrate.

We created a prototype of MeshReduce in C++ using off-the-shelf hardware and software tools such as Azure Kinect sensors [1] to capture raw RGB-D frames, Open3D [38] for mesh reconstruction, Draco [2] for compression. Compared to existing capture pipelines, MeshReduce significantly reduces bitrate requirements (10-16×) for the same visual quality, offers lower latency, and scales well with large number of cameras 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 scene capture pipeline (§3.1) and distribute the scene reconstruction (§3.2).
- A bandwidth efficient texture processing pipeline (§3.3).
- Scene-independent, pareto-efficient bitstream generation for textured meshes using a predictive model of rate distortion curves (§4).
- An open source implementation of MeshReduce that includes RGB-D capture, mesh reconstruction, texture processing, and a first-of-its-kind dataset with 3D scenes captured using multiple commodity depth cameras.

[2]We interchangeably use terms mesh, geometry, polygons and triangles.

Ethical issues: This work does not raise any ethical issues.

2 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 [30, 34, 41], which is often best explored in a centralized fashion with lossless data sources. In contrast, the networking and systems community has focused on directly transferring low-level sensor data (e.g., point cloud) to end users [18, 19, 25, 46]. In this section, we characterize these data representations with an emphasis on demonstrating their implications on live 3D video creation and streaming.

2.1 Volumetric (3D) Video Representations

Most of the prior 3D video streaming work [18, 19, 25, 46] uses two 3D representations: Point Cloud and Textured Mesh.

Point Clouds consist of a set of 3D points, each assigned (X, Y, Z) position and (R, G, B) color value. Point clouds are typically captured from depth sensors (e.g., Lidar [4]) or can be created from RGB-D (color and depth) images (e.g., Kinect [1]). While streaming point clouds does not involve much computation on the capture-side, much of the computation burden is shifted to client-side (e.g., merging multiple camera views into one 3D scene). Such pipeline leads to several nontrivial challenges for client headsets with limited compute and memory resources [19, 22, 25].

Textured Mesh representation is an alternative to point clouds, which consists of polygonal geometry (e.g., a triangle) and a texture image that is mapped onto polygons' surface. Textured meshes can be extracted from raw sensor data such as RGB-D or point clouds. Unlike point clouds, textured mesh based 3D streaming places the 3D scene creation at the capture-side. The mesh and texture data is completely decoupled and the structure is directly compatible with modern graphics hardware on client-side, and hence incurs much less computation on the client compared to point clouds. More details on textured meshes[3] are in §2.3.

Key insight: An observation that is foundation to this work is that textured mesh requires much less data to store and transmit compared to point cloud for a given quality. The key reason behind this result is that meshes can approximate real-world surfaces well with polygons whereas point cloud has to fill the surface with 3D points requiring much more data. An intuitive example is shown in Figure 2. A flat surface such as Cube is a best case for mesh because it can represent its shape with as few as 12 triangles, while point cloud must fill the entire cube with points. Even in the worst case (i.e., a sphere which requires complex polygonal geometry), the mesh representation requires less data than point cloud for the same rendered quality. The preponderance of flat surfaces in indoor spaces suggests that mesh-based approaches are likely to perform especially well in such settings.

[3]A mesh can be standalone on its own, but we call it textured mesh in our work because our system need both texture and mesh for visual quality.

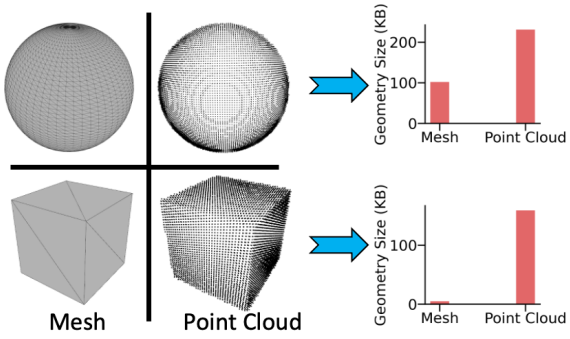


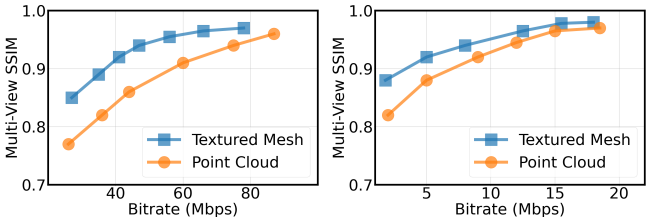
Figure 2: Qualitative example showing that mesh requires significantly less data than point cloud. Even in the worst case scenario (i.e., sphere), mesh has lower bitrate than point cloud for the same quality.

Next, our goal is to study these two representations systematically to ascertain the finding that textured mesh indeed performs better than point cloud and establish the need to stream textured mesh for volumetric video streaming.

2.2 Point Cloud Vs. Textured Mesh

In this section, we systematically compare point cloud and textured mesh representations using bitrate vs. quality tradeoffs. We compute optimal rate distortion curves (i.e., bitrate vs. quality curves) over two datasets— 1) from our own custom testbed, 2) a commonly used volumetric video dataset known as V-Sense [35]. The dataset details are described in §6.1. In addition to the dataset, comparing 3D content requires careful measurement methodology in terms of quality evaluation and optimal bitrate curves as described below.

3D quality evaluation: Unfortunately, there are no well-defined metrics for 3D visual quality assessment. We propose adapting SSIM (structural similarity index metric) [43], a popular 2D video quality metric that measures 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 [9] to avoid locations and angles that do not accurately capture useful locations for viewers (e.g. under a table, looking into a corner, etc). Each of our collected datasets comes with a predefined set of view points (on average ≈30).



(a) Our Dataset (b) V-Sense Dataset [35]

Figure 3: Quality vs. bitrate curves for comparing point clouds and textured meshes. Textured mesh requires less bitrate compared to point cloud for same quality.

Compression methods: To produce rate distortion curves, we use state-of-the-art compression methods for both representations. For point clouds, we use the MPEG vpcc [7] compression, and for textured meshes, we use Google Draco [2] and h264_nvenc [37] for mesh and texture compression respectively. Since this is an offline analysis, we were able to select optimal compression parameters for both cases while generating the rate distortion curves.

Results: Figure 3 compares rate-distortion for each representation. Textured mesh always provides the best quality for a given bitrate, and benefits with mesh are even greater at low bitrates. The fundamental problem with point clouds is that in order to have a high-quality video, the scene needs to be represented with high point density (e.g., at millimeter scale) and with each point being a floating point 3D coordinate introduces extremely high raw data rates. In addition, a practical problem with point clouds is the resolution gap between today's depth and camera sensors, where the majority of depth sensors have significantly lower resolution (e.g., 480P) than color sensors (e.g., 4K). Consequently, when transforming the RGB-D to a point cloud, a significant portion of texture resolution is lost because the color is assigned to only the coarse-grained 3D point.

From the rate distortion standpoint, it is clear that textured meshes are more efficient compared to point clouds, and it makes perfect sense to use textured mesh representation for volumetric video streaming. However, unlike point clouds, creating textured meshes involves several computationally expensive tasks that pose nontrivial challenges for live streaming. In the following section, we briefly explain key tasks involved in mesh creation and the challenges associated with the traditional mesh streaming pipeline.

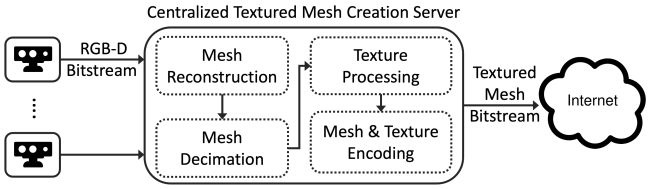


Figure 4: Textured mesh streaming pipeline in a traditional centralized scene capture architecture.

2.3 Challenges of Capturing Textured Mesh Bitstream in Real Time

Unlike point clouds, creating textured meshes involves three tasks that are essential for a compact 3D representation: 1) Reconstruction, 2) Decimation, 3) Texture processing. As background to understand MeshReduce design (§3), we briefly explain these tasks below.

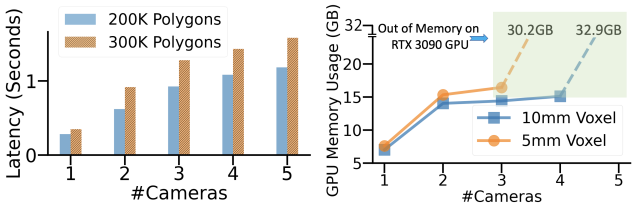
Reconstruction task starts by taking overlapping RGB-D frames (from multiple cameras) as input and incrementally combines them into a voxel grid [11]. Once the voxel grid is created from RGB-D, a polygon extraction algorithm is used to create a mesh [26].

Decimation reduces polygon count from the above reconstructed mesh while keeping the overall shape, volume and boundaries preserved 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. In our experiments, we find that decimation in fact plays a significant role in reducing bitrate while retaining high quality (more in §3.1).

Texture processing task maps each polygon to a corresponding texture region. This mapping information is stored inside mesh data structure and is used during rendering to apply texture onto polygons.

Existing solutions use this pipeline in a centralized architecture [34]. Figure 4 shows such a pipeline, where the RGB-D frames are passed to mesh reconstruction, decimation, and texture processing, followed by mesh and texture encoding to prepare the textured mesh bitstream. However, the problem with this approach is that these tasks are extremely computationally expensive and poses several nontrivial challenges for live 3D video streaming, described below.

1) *High latency for creating mesh bitstream:* Figure 5a shows the end-to-end 3D scene capture latency for two scenes that have different geometric complexity (with 200K and 300K polygons). We observe that the latency increases significantly with the number of cameras as well as scene complexity. Most interactive volumetric video applications target a latency below 100ms. In the next section (§3.1), we perform micro-benchmarks to identify which components are most significant and which can potentially be parallelized.



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

Figure 5: 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.

2) *Poor scalability due to high GPU memory usage:* Figure 5b shows GPU memory usage as we add more cameras to our system with different voxel resolutions. Even a high-end GPU like an Nvidia RTX 3090 which has 24GB of memory, supports less than 5 cameras for as coarse voxel resolution as 10mm. This is mainly because of the GPU memory hungry reconstruction task which extracts polygons in parallel on GPU. In general, more fine-grained voxel resolutions (e.g., 1mm) and more (a few 10s to hundreds) cameras are needed to capture a high quality 3D scenes and the problem becomes more acute as we capture larger scenes.

3) *Generating optimal bitrates for texture and geometry:* Finding optimal resolution, decimation, and compression levels to generate a high-quality texture and mesh for a given target bitrate, theoretically involves a computationally expensive exhaustive search. This is mainly because an offline analysis of parameters results in suboptimal quality in a live setting because of the scene dynamics both in terms of texture and mesh geometry (more in §4). More importantly, the target bitrate needs to be optimally split between texture and mesh geometry so that the final rendering quality is optimal.

Takeaway: *Textured mesh requires significantly lower data rates compared to other 3D data representations for the same quality. However, compact mesh reconstruction is extremely compute and memory expensive, presenting a major challenge for low latency, live 3D streaming systems.*

3 MESHREDUCE: DESIGN

Our overarching vision in this work is to adopt textured mesh representation for live 3D video streaming, given its bandwidth efficiency compared to other 3D formats. However, creating textured mesh using a traditional monolithic pipeline [34] has several bottlenecks in live streaming scenarios. To address these, we propose MeshReduce, a distributed approach for creating textured mesh streams. MeshReduce generates output data with low latency, eliminates memory and scalability issues, and achieves practical bitrates for

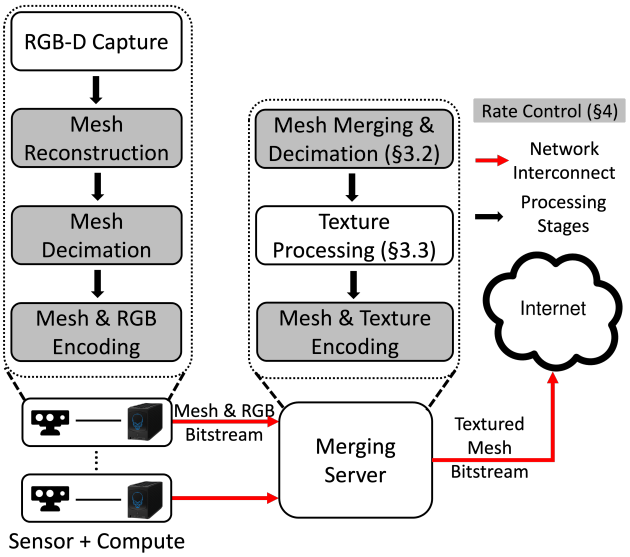


Figure 6: MeshReduce's distributed scene capture and rate control pipeline for textured mesh streaming.

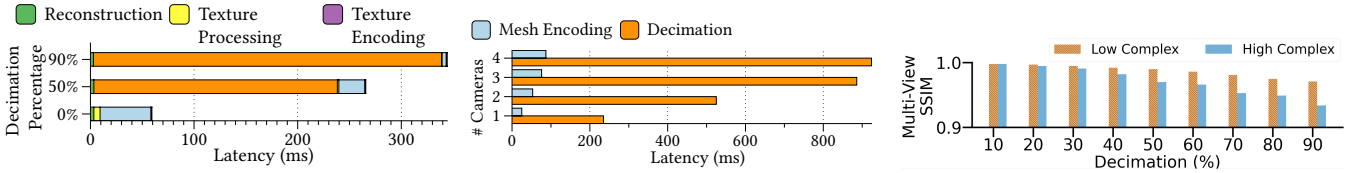
streaming textured mesh on the Internet. Figure 6 shows a block-level overview of MeshReduce.

Our first step in designing MeshReduce is to understand the root cause of the bottlenecks in the texture mesh creation pipeline. For this, we dissect the pipeline and study several key tasks described in §2.3, such as reconstruction, decimation, texture mapping, and encoding (§3.1).

Based on insights from this study, MeshReduce uses a distributed architecture where each camera feed is locally converted into a mesh geometry and RGB view that is streamed to a server (§3.2). This server merges these individual mesh streams (from multiple cameras) into one scene. Similarly, the server merges texture streams of multiple cameras efficiently for bandwisth (§3.3). This approach eliminates the processing and memory bottlenecks that existing centralized designs suffer from. Finally, we develop a new rate control problem for streaming textured meshes, that addresses the challenges of selecting best compression knobs as well as dividing the available bandwidth between mesh and texture streams (§4).

3.1 Root Cause Analysis

Figure 7 shows a breakdown of the computation latency for various tasks in the textured mesh scene capture pipeline. We find that mesh decimation which is used to reduce the polygon count (see §2.3), is the most time-consuming task. This is mainly because the mesh decimation algorithm runs iteratively to reduce polygon count and difficult to parallelize the task (note that we adopt a commonly used decimation algorithm in practice [16]). Figure 7a shows that even for a single camera scene with 50% decimation we observe more



(a) Single camera scene (b) Multi-camera scene (4 cameras) (c) Decimation has little impact on quality

Figure 7: Breakdown of scene capture latency: a) Decimation has highest latency, b) Decimation latency increases as we add more camera scenes for better quality, c) Decimation can be used to get high quality at low bitrates.

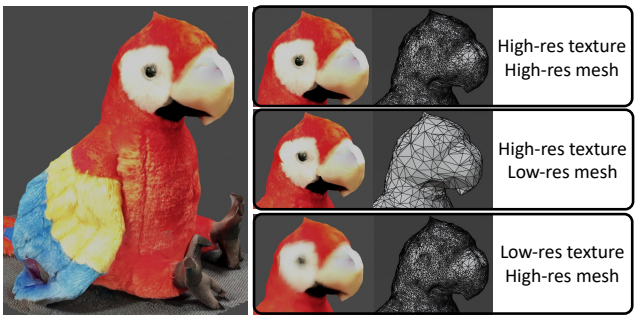


Figure 8: Decimation example showing significantly reduced polygon count with little impact on quality (middle figure compared to the top). Reducing texture resolution on the other hand degrades the quality significantly (bottom figure). Left side is a full parrot scene.

than 200ms latency and that as we increase the decimation level, its latency increases significantly.

Unlike other tasks such as mesh reconstruction which can run on GPUs and achieve low latency, the decimation algorithm is not data parallel, and hence, is not accelerated by GPUs in practice. This latency becomes more problematic (>1s, see Figure 7b) when we decimate bigger meshes i.e., mesh created from more cameras. On the other hand, decimation is needed to significantly reduce polygon count without affecting quality much. A qualitative example of decimation value in reducing polygon count vs. texture quality can be seen in Figure 8. We also show an objective evaluation of decimation impact in reducing bitrate while maintaining quality (see Figure 7c). 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 decimated by 50% without having a significant noticeable 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).*

Another key finding from our analysis is that reconstruction task is the root cause for GPU memory bottleneck. While previous mesh reconstruction approaches were CPU-limited, recent advances (e.g., voxel data structures and data parallel polygon extraction) [32, 41], have significantly accelerated the reconstruction process (Figure 7a shows <1ms latency) at the expense of GPU memory consumption. Mesh reconstruction now requires significant GPU memory for parallel polygon extraction and is limited to just 3-5 cameras even on an RTX 3090 GPU (as shown in Figure 5b).

3.2 Distributed Capture Pipeline

In order to alleviate these GPU memory and latency bottlenecks, we propose splitting the monolithic capture pipeline (shown in Figure 4) across many compute nodes where each node performs a smaller single-camera mesh reconstruction as well as decimation. The system then merges these per-camera reconstructions into a single mesh on a merging server. A compute node and its GPU has sufficient resources to reconstruct a single camera scene and decimate with minimal latency since one camera can only cover a small scene[4]. Another important result of this design is that the merging process at the server is fast even on CPU and does not require any 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 that has a cluster of compute nodes, where each node does 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 node 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 cameras to perform the camera-side reconstruction pipeline in hardware, much like current generation cameras performing video compression.

Mesh merging: Once the per-camera mesh reconstructions are available at the merging server, the next step is to merge

[4]Note, we parallelize decimation on each compute node by locally decomposing depth frame into several depth patches to reduce latency (see §5).

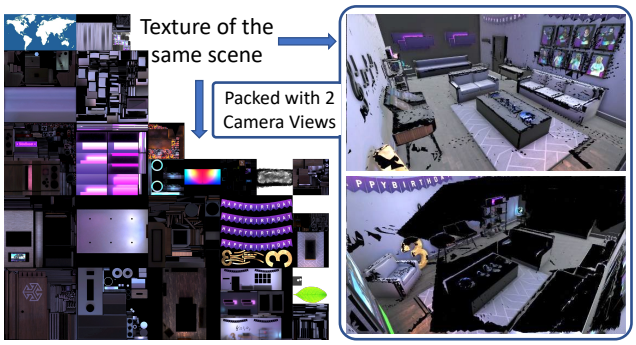
them into a single mesh to represent a complete volumetric scene. Since our objective is to capture 3D spaces from multiple view angles, overlapping mesh geometry from multiple camera views will unavoidably exist. Thus, it is necessary to remove the overlapping mesh regions for bandwidth efficiency. If camera feeds were perfect, it is straightforward to merge different mesh regions by simply combining them. However, in practice, the imprecise camera calibration, as well as noisy camera depth readings, necessitate the need for a new solution. In MeshReduce, we have developed a mesh merging algorithm that is able to accommodate noisy sensor data, eliminate duplicate mesh information and smoothly stitch together the different views. This process is described in detail in the Appendix B (and depicted in Figure 18).

3.3 Bandwidth-efficient Texture Processing

Once the individual mesh reconstructions from each camera are merged, and we have a complete mesh of the environment, our next step is to process texture (i.e., mapping texture to polygons, texture compression etc) from each camera. This raises an interesting question of bandwidth efficiency for Internet streaming: *where in the scene capture pipeline it is appropriate to process textures— on the camera-side compute node or at the merging server?* Naturally from our distributed design standpoint, it is intuitive to process the textures at the camera and stream directly to the Internet. This would reduce computation load on the merging server.

However, despite these arguments, we find that texture processing at the merging server presents more benefits than at the camera-side compute nodes. There are multiple reasons behind this counter intuitive result: First, directly streaming textures (to the Internet) from cameras results in significant bandwidth overhead because of the redundant regions across overlapping cameras. Second, the compression efficiency is much higher with a stitched panoramic video of all the camera views rather than encoding each camera view separately irrespective of whether there is cross-camera overlapping or not. This is mainly because of the intra- and inter-frame compression used by the traditional 2D codecs that take advantage of spatial and temporal redundancy within and across the frames. This compression opportunity is lost if each camera view is encoded separately. Finally, the mesh processing tasks at the merging server such as mesh merging, decimation, and mesh encoding are mainly CPU intensive, whereas texture (video) codecs e.g., nvenc [37], are mainly GPU-intensive, and hence both mesh and texture tasks can be effectively parallelized at the merging server.

Texture optimization at merging server: Similar to mesh merging task (in 3.2), we stitch the texture streams (from multiple cameras) into a single panoramic texture, we call texture atlas. We create this texture atlas by simply concatenating



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

Figure 9: Texture processing of existing approach [17] 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.

textures (i.e., increasing the texture resolution) and blacking out the regions that are overlapping from multiple cameras. This is unlike traditional approaches that try to pack all the textures into a smaller resolution texture [35][17]. See Figure 9. The key difference between the two approaches is that while existing approach to texture atlas has lower resolution, its texture packing lacks spatio-temporal consistency, and hence result in compression inefficiency. Instead, we find that having larger resolution of stitched textures while maintaining spatio-temporal structure results in much better compression efficiency even with larger resolutions. This is mainly becuase we use traditional 2D codecs for texture compression that exploits spatio-temporal redundancy well. The blacked-out of regions is discussed in detail in Appendix C.

4 RATE CONTROL FOR TEXTURED MESH STREAMING

Given our proposed distributed architecture, one of the core challenges is being able to adapt and how we transmit data either between our internal links or across the Internet in the presence of unpredictable network capacity. This requires efficient bitstream generation for transmission with the goal of approaching optimal bitrates for rate adaptation algorithms. In Figure 6, we show this capability where all the red edges have textured mesh transmission over links with variable bandwidth.

4.1 Rate Control Problem for 3D

Traditionally (e.g., in 2D video), the streaming quality is optimized for a given target bitrate using a rate distortion function: $\mathcal{L} = \mathcal{D} + \lambda\mathcal{R}$, where $\mathcal{R}$ is the bitrate, $\mathcal{D}$ is the distortion caused after compression and λ is a tuning parameter to trade video quality with bitrate. Naturally, this function

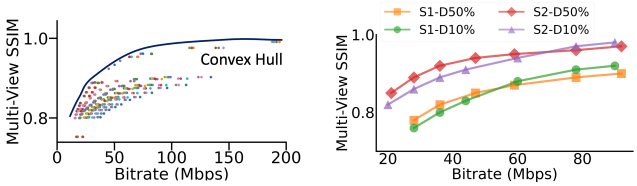
can be extended to 3D rate control as: $\mathcal{L} = \mathcal{D} + \lambda_t \mathcal{R}_t + \lambda_g \mathcal{R}_g$, where $\mathcal{R}_t$ and $\mathcal{R}_g$ represent the bitrates for texture and geometry respectively. Note that our goal is to maximize final rendered quality given the combination 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 Allocation for Texture and Mesh

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 10a shows an example of rate distortion points for one scene that is decimated and encoded at different levels with different resolutions. Each point in the plot denotes a combination of <resolution, decimation, compression> level and different combinations are a 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' such Pareto frontier curve for any scene is ideal when preparing the bitstream for fine-grained rate adaptation. However, the rate distortion curves are a function of the underlying scene complexity and the optimal coding parameters for one scene can be suboptimal for others (see Figure 10b). This creates a major challenge to prepare Pareto optimal curves in case of live streaming. A naive approach to deal with this is by computing the Pareto frontier online by exploring the entire state space through exhaustive search. However, the exhaustive search is extremely compute intensive task because it involves exploring thousands of coding parameters to find the best.

We propose a predictive modeling approach by formulating the problem as a multi-objective classification problem. 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 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



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

Figure 10: 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.

use either a simple and lightweight but slightly less accurate models (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 adapt to the fast paced scene and variable network conditions. However, unlike neural networks which automatically extract features from raw frames, training these models requires manual feature extraction that involves a set of features that correlate well with the outcome of the model.

Feature extraction: Our goal is to identify computationally low complex features so that end-to-end feature extraction and prediction can be achieved in under a few milliseconds. To this end, we adopt a simple frequency distribution based features as a proxy to represent scene signature. Specifically, we use below DCT energy function from [21] to compute spatial *information* in each frame (both for depth and RGB).

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

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 [21]. Figure 11 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 used as a rich feature in classifying the coding parameters. We also compute DCT energy temporally (to accommodate for temporal redundancy) 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.

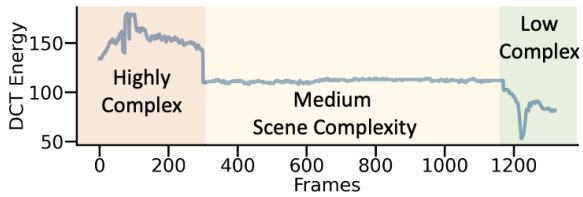


Figure 11: DCT energy as a feature to measure scene complexity. A low DCT energy indicates low complex scene and vice versa.

Joint rate allocation: 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 with 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) given texture mesh streaming systems that we leave for future work.

5 IMPLEMENTATION AND SYSTEM SETUP

MeshReduce's end-to-end system consists of scene capture, reconstruction, streaming and merging to prepare a textured mesh video bitstream. Figure 6 shows a breakdown of system components. The step-by-step procedure of the pipeline is 1) camera-side compute nodes reconstructs mesh, then decimate and encode it before streaming meshes and RGB images to a merging server, 2) the merging server processes all the camera node reconstructions as well as processes textures, 3) the merging server uses rate control model to generate target bitrates for texture and geometry streams for the Internet delivery.

We built MeshReduce in C++ on top of Open3D [38]. For our experiments, we use four Azure Kinect cameras capturing the scene using Kinect SDK [1]. The cameras are calibrated with an AprilTag [33] for initial pose alignment and then fine tuned automatically with Iterative Closest Point (ICP) refinement [40]. We capture the color and depth frames at a resolution of 3840x2160 and 640x576, at 30FPS. The raw color is read in the form of RGB (24 bits per pixel) while the depth is in 16-bit unsigned integer format.

We use a Linux machine with an Intel i9 16 core 12900 CPU, 64GB RAM, and Nvidia 3070 Ti GPU at each camera for our camera side mesh reconstruction. Note that this setup is over-provisioned in terms of compute on each camera, but this gave us headroom to explore higher resolution sources. Open3D provides various mesh reconstruction, decimation, and manipulation functions. We use Google Draco [2] for mesh compression and `h264_nvenc` [37] for color/texture video compression. Each node streams color and mesh bitstream to a central merging server over a TCP connection.

The merging server is a Linux machine with AMD 16 core 5950x CPU, 64GB RAM, and Nvidia 3090 GPU that listens with TCP socket on separate threads, receiving color and mesh bitstreams from separate camera. Once the bitstreams are received, the mesh is decoded using Draco [2] for mesh merging, and the color frames are decoded using `h264_nvdec` [37] 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], the texture atlas is compressed with `h264_nvenc` [37]. The resulting product is a internet friendly texture mesh bitstream.

Rate control model training: We use a multi-output classification using classifier chains [36] to exploit the correlation among the coding parameters rather than independently selecting each coding parameter. We fit three models—SVM, Logistic Regression, and a RandomForests classifier and perform a grid search on each of them to tune the best parameters. After the grid search, we select the best model (RandomForests in our case) and its optimal parameters from the search.

Additional system optimization: The off-the-shelf software for decimation and compression used in MeshReduce originally runs on a single CPU core, underutilizing the camera-side compute capacity. We introduce an additional optimization for parallel decimation and compression locally on each camera compute node and bring down the latency. We do this by without modifying the internal decimation or compression algorithms using our split-merge design idea, where we locally decompose the depth maps into $M \times M$ depth patches, and reconstruct a mesh out of each patch on separate core. We then decimation and compress each of these reconstructions in parallel and stream all the patches together to the merging server. The server the decodes such partial reconstructions from all cameras and performs the global merging step that we introduced in §3.2. We evaluate the impact of such local decomposition on latency as well as quality in §6.2.

Figure 12: A qualitative result of a 3D scene reconstructed from our testbed, rendered from three arbitrary viewpoints.

6 EVALUATION

Our evaluation goal is to answer the following questions:

- What are the bitrate requirements of MeshReduce and how does it compare to state-of-the-art systems?
- What is the impact of latency and bitrate trade-off on final rendered quality?
- How does MeshReduce scale across the 3D scene size and number of capture cameras?

6.1 Evaluation Methodology

In this section, we describe the methodology to evaluate MeshReduce's end-to-end capture performance.

Dataset: We collect several samples of 3D video scenes under different environments (e.g., conference room, office space, corridor, and hallways) by capturing RGB-D frames from multiple cameras. We construct point cloud and textured mesh frames from the synchronized RGB-D frames. Using RGB videos from four cameras and the corresponding depth frames covering multiple viewpoints, we construct synchronized point cloud and mesh frames. To evaluate the quality after introducing coding artifacts, we create reference frames with a 1mm scale high voxel resolution to extract a mesh from the depth frames. The dataset details are described in detail in Appendix §A. Figure 12 shows an example of a rendered scene captured from our four camera testbed in a lounge area. 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 final rendered quality, bitrate, and latency of the system. We evaluate quality using a multi-view SSIM metric, that computes quality multiple viewpoints to cover the entire 3D scene quality as defined in §2.2. Latency is the time needed to generate a mesh bitstreams starting from the RGB-D frame acquisition.

Baselines. We compare MeshReduce's benefits with the following variants of existing work[5]. We implement these baselines on our testbed for fair comparison.

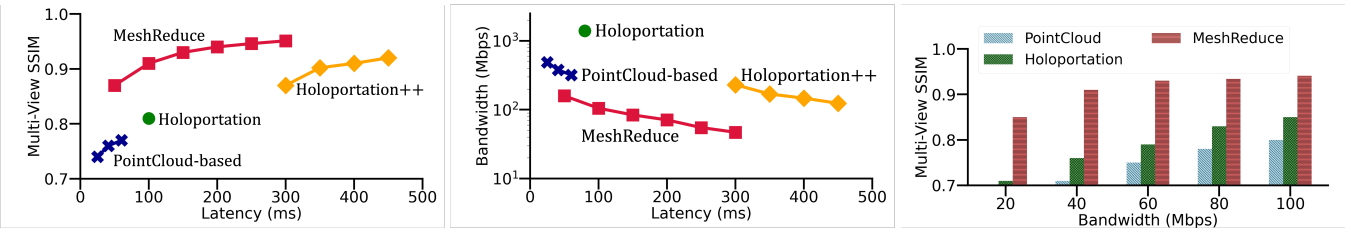
- **Holoportation [34],** a state-of-the-art textured mesh streaming system for real-time 3D telepresence. Holoportation uses a lightweight compression (LZ4 [6]) for both texture and geometry to provide real-time scene capture.
- **PointCloud-based,** a live point cloud based 3D reconstruction system. The closest similar system is LiveScan3D [22]; however, it does not discuss much about compression and streaming. For fair comparison, we apply MPEG vpcc [7] compression when comparing the rate distortion results. Majority of the volumetric video streaming work [19, 25, 46] also fall 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 as well as geometry. We introduce Draco compression for mesh along with decimation and h264_nvenc for texture compression for Holoportation, along with parallel decimation optimizations on a centralized server, and call it Holoportation++. This is distinct from MeshReducein that it does not distribute mesh creation followed by merging step.

6.2 End-to-End Results

Figure 13 shows the end-to-end scene capture latency against quality and bandwidth requirement of MeshReduce comparing with its alternatives. We fix the bandwidth at 100 Mbps in Figure 13a by choosing different coding parameters for each method that results 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 [43]. On the other hand, the optimized version of Holoportation performs close to MeshReduce but at the expense of 4× higher latency.

Figure 13b 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

[5]There are other related work such as viewport adaptive volumetric streaming [19], focusing on rate adaptation algorithms. These are slightly orthogonal to our work since we mainly focus on capturing and preparing bitstreams efficiently. Mesh representation is amenable to viewport adaptation and MeshReduce can be used in these solutions on the capture side.



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

Figure 13: Latency, Quality, and Bandwidth trade-offs of MeshReduce and its benefits compared to state-of-the-art 3D real-time streaming systems. MeshReduce can achieve significantly higher quality while maintaining low latency and low bandwidth.

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 13c shows the quality achieved for each system at different bandwidths for a latency of 100ms. MeshReduce can have perceivable (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 below perceivable SSIM threshold (i.e., < 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 is decimating the entire 3D scene at once and suffers from high latency. On the other hand, the point cloud based system suffers mainly because of its compression inefficiency as described in §2.2.

Ablation study: Figure 14 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 camera 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 with out 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 impact on quality. Figure 15 shows the latency and quality loss due to the decomposition and merging when comparing 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.

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 camera-side compute node has enough GPU memory to reconstruct its own single view (since a maximum per-camera scene GPU memory requirement in our scenes is 3.5 GB, see §2.3). While the merging server still processes all the per-camera reconstructions, the merging is only a CPU and DRAM bounded task. 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 cameras or many scenes, etc. Figure 16 shows an experimental result on the scaling limits of MeshReduce with synthetic scenes (each of 6m×9m) captured with several cameras. For small scale scenes (i.e., up to 5 cameras), the DRAM usage is as small as under 2 GB, and increases linearly with the number of cameras. However, even at 400 cameras, the DRAM usage is around only 43.2 GB, which is only fraction of hundreds of the DRAM capacity even on today's commodity devices. Supporting such large scenes demonstrates that MeshReduce is not only efficient for room-scale 3D telepresence applications, but also enables campus-wide building-scale remote exploration of spaces.

While MeshReduce supports large number of camera scenes, one caveat with such scenario is the computational complexity of merging task. In our experiments, we observe a merging latency of 1.2 seconds (mainly from the raycasting task) when we scale to 400 cameras. However MeshReduce's focus is streaming small-scale 3D scenes with low bitrates and low latency, and hence we leave optimizing of latency for large scenes/cameras for future work. Existing high complex scene rendering solutions such as R2E2 [15] can be applied to reconstruct such large scenes with low latency.

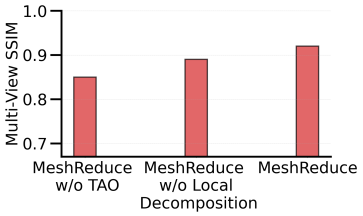


Figure 14: Breakdown of MeshReduce's performance with its individual components.

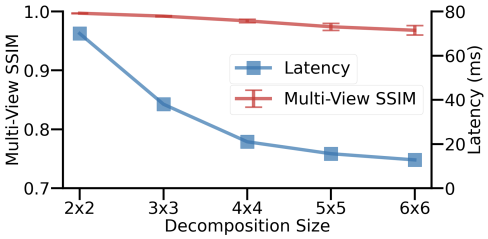


Figure 15: Impact of multi-core local decomposition on quality and latency.

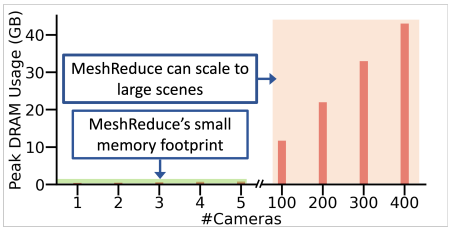


Figure 16: MeshReduce is CPU DRAM limited (as opposed to GPU memory) and hence can scale to large number of cameras.

6.4 Rate Control Sensitivity Analysis

We compare the rate distortion performance of MeshReduce with an offline generated pareto-optimal rate distortion curves 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 store in table for online lookup. Figure 17 shows a cumulative distribution of difference in SSIM for each models compared against the optimal rate distoriton curve. MeshReduce can achieve within 0.15 SSIM of pareto-optimal curve with an average of 0.08 SSIM difference. This is noticeably better than 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 accurately store the coding parameters for all possible scene dynamics.

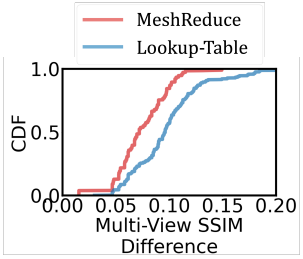


Figure 17: MeshReduce's rate control model performance.

7 RELATED WORK

3D (volumetric) video streaming: Majority of the prior work on 3D streaming uses point clouds [19, 25, 46]. While these systems not only suffer from high bandwidth requirements, they impose significant computation burden of 3D scene creation on the client-side which is not desirable. Also, these systems focus on on-demand streaming and suffer from latency in live scenarios. VRComm [18] proposes a real-time system with low latency but uses RGB-D representation and suffers from bandwidth overheads. Unlike the prior work, MeshReduce adopts textured meshes for streaming and introduces a practical system to capture 3D scenes with lower bitrates and lower latency, while maintaining the quality.

3D scene capture: There has been extensive work on 3D scene capture pipeline. Much of the prior work focus on efficient algorithms to generate high quality scenes [10, 26–28, 31] and often ignore the computational complexity of capture pipeline. Recent solutions tackle this with memory efficient data structures for faster reconstruction or relatively to consume less compute resources in limited environments while sacrificing quality of the scenes [12, 23, 32, 45]. Built on such extensive work, recent solutions introduce end-to-end 3D streaming pipelines using textured mesh representation such as FVV [10], Holoportation [34], Fusion4D [13] and Motange4D [14]. However, these systems mostly focus on high bandwidth locally networked scenarios and does not discuss the practicality of live streaming on the Internet.

8 CONCLUSION AND FUTURE WORK

In conclusion, this paper presented MeshReduce, an open-source volumetric scene capture system that can fuse RGB-D streams from multiple network-connected cameras in real-time. Our key insight is that independently created mesh reconstructions can be merged incrementally in a distribute manner instead of at one centralized source without a significant loss in quality. This split-merge approach leads to a more compact intermediate representation of the 3D data that yields equal render quality. Texture mesh representations of scenes are naturally better at capturing planar surfaces and decoupling geometry from texture data. This enables MeshReduce to more easily adapt to network fluctuations.

As future work, we envision rate adaptation algorithms for streaming 3D content using the concept of progressive meshes [20] and texture mipmaps [30]. These approaches are widely used in traditional graphics pipelines for adapting to the rendering capability of a platform. Progressive

meshes are analogues to a form of scalable (layered) video compression in 2D video streaming [3], where the quality of geometry reconstruction can be incrementally improved by adding more vertices and edges as the computation or network conditions improve. Furthermore, we can adopt viewport prediction based adaptive streaming methods to stream only the region the user views [19].

Finally, there is a large body of work on lightfield volumetric reconstruction [42] as well as a rapidly growing community exploring neural scene reconstruction [29] techniques. We believe our split-merge architecture could be applied to both with an adaptation of our merge function.

REFERENCES

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

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

[3] [n. d.]. H.264/AVC Scalability Extension. https://avc.hhi.fraunhofer.de/svc. ([n. d.]).

[4] [n. d.]. Intel RealSense LiDAR Camera L515. https://www.intelrealsense.com/lidar-camera-l515/. ([n. d.]). Online. Accessed: May 2022.

[5] [n. d.]. libvpx-vp9. https://trac.ffmpeg.org/wiki/Encode/VP9. ([n. d.]).

[6] [n. d.]. LZ4. https://github.com/lz4/lz4. ([n. d.]). Online. Accessed: Sep 2022.

[7] [n. d.]. MPEG Point Cloud Compression. https://mpeg-pcc.org/. ([n. d.]). Online. Accessed: Sep 2022.

[8] Youssef Alj, Guillaume Boisson, Philippe Bordes, Muriel Pressigout, and Luce Morin. 2012. Multi-texturing 3D models: how to choose the best texture?. In *2012 International Conference on 3D Imaging (IC3D)*. IEEE, 1–8.

[9] Priyadarshi Bhattacharya and Marina L. Gavrilova. 2008. Roadmap-Based Path Planning - Using the Voronoi Diagram for a Clearance-Based Shortest Path. *IEEE Robotics & Automation Magazine* 15, 2 (2008), 58–66. https://doi.org/10.1109/MRA.2008.921540

[10] Alvaro Collet, Ming Chuang, Pat Sweeney, Don Gillett, Dennis Evseev, David Calabrese, Hugues Hoppe, Adam Kirk, and Steve Sullivan. 2015. High-quality streamable free-viewpoint video. *ACM Transactions on Graphics (ToG)* 34, 4 (2015), 1–13.

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

[12] Wei Dong, Jieqi Shi, Weijie Tang, Xin Wang, and Hongbin Zha. 2018. An efficient volumetric mesh representation for real-time scene reconstruction using spatial hashing. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 6323–6330.

[13] Mingsong Dou, Sameh Khamis, Yury Degtyarev, Philip Davidson, Sean Ryan Fanello, Adarsh Kowdle, Sergio Orts Escolano, Christoph Rhemann, David Kim, Jonathan Taylor, Pushmeet Kohli, Vladimir Tankovich, and Shahram Izadi. 2016. Fusion4D: Real-Time Performance Capture of Challenging Scenes. *ACM Trans. Graph.* (2016).

[14] Ruofei Du, Ming Chuang, Wayne Chang, Hugues Hoppe, and Amitabh Varshney. 2018. 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. https://doi.org/10.1145/3190834.3190843

[15] Sadjad Fouladi, Brennan Shacklett, Fait Poms, Arjun Arora, Alex Ozdemir, Deepti Raghavan, Pat Hanrahan, Kayvon Fatahalian, and Keith Winstein. 2022. R2E2: low-latency path tracing of terabyte-scale scenes using thousands of cloud CPUs. *ACM Transactions on Graphics (TOG)* 41, 4 (2022), 1–12.

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

[17] D Graziosi, O Nakagami, S Kuma, A Zaghetto, T Suzuki, and A Tabatabai. 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 (2020), e13.

[18] Simon NB Gunkel, Rick Hindriks, Karim M El Assal, Hans M Stokking, Sylvie Dijkstra-Soudarissanane, Frank ter Haar, and Omar Niamut. 2021. VRComm: an end-to-end web system for real-time photorealistic social VR communication. In *Proceedings of the 12th ACM Multimedia Systems Conference*. 65–79.

[19] Bo Han, Yu Liu, and Feng Qian. 2020. ViVo: Visibility-aware mobile volumetric video streaming. In *Proceedings of the 26th annual international conference on mobile computing and networking*. 1–13.

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

[21] Michael King, Zinovi Tauber, and Ze-Nian Li. 2007. A new energy function for segmentation and compression. In *2007 IEEE International Conference on Multimedia and Expo*. IEEE, 1647–1650.

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

[23] Olaf Kähler, Victor Prisacariu, Julien Valentin, and David Murray. 2016. Hierarchical Voxel Block Hashing for Efficient Integration of Depth Images. *IEEE Robotics and Automation Letters* 1, 1 (2016), 192–197. https://doi.org/10.1109/LRA.2015.2512958

[24] Jason Lawrence, Dan B Goldman, Supreeth Achar, Gregory Major Blascovich, Joseph G. Desloge, Tommy Fortes, Eric M. Gomez, Sascha Häberling, Hugues Hoppe, Andy Huibers, Claude Knaus, Brian Kuschak, Ricardo Martin-Brualla, Harris Nover, Andrew Ian Russell, Steven M. Seitz, and Kevin Tong. 2021. Project Starline: A high-fidelity telepresence system. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)* 40(6) (2021).

[25] Kyungjin Lee, Juheon Yi, Youngki Lee, Sunghyun Choi, and Young Min Kim. 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*. 1–14.

[26] William E Lorensen and Harvey E Cline. 1987. Marching cubes: A high resolution 3D surface construction algorithm. *ACM siggraph computer graphics* 21, 4 (1987), 163–169.

[27] Andrew Maimone and Henry Fuchs. 2011. 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*. IEEE, 137–146.

[28] Andrew Maimone and Henry Fuchs. 2012. 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)*. 1–4. https://doi.org/10.1109/3DTV.2012.6365430

[29] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. (2020). https://doi.org/10.48550/ARXIV.2003.08934

[30] Joerg H Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. 2018. Shading atlas streaming. *ACM Transactions on Graphics (TOG)* 37, 6 (2018),

1–16.
[31] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. 2011. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE international symposium on mixed and augmented reality*. Ieee, 127–136.
[32] Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Marc Stamminger. 2013. Real-time 3D reconstruction at scale using voxel hashing. *ACM Transactions on Graphics (ToG)* 32, 6 (2013), 1–11.
[33] Edwin Olson. 2011. AprilTag: A robust and flexible visual fiducial system. In *2011 IEEE International Conference on Robotics and Automation*. 3400–3407. https://doi.org/10.1109/ICRA.2011.5979561
[34] Sergio Orts-Escolano, Christoph Rhemann, Sean Fanello, Wayne Chang, Adarsh Kowdle, Yury Degtyarev, David Kim, Philip L Davidson, Sameh Khamis, Mingsong Dou, et al. 2016. Holoportation: Virtual 3d teleportation in real-time. In *Proceedings of the 29th annual symposium on user interface software and technology*. 741–754.
[35] Rafael Pagés, Konstantinos Amplianitis, Jan Ondrej, Emin Zerman, and Aljosa Smolic. 2021. Volograms & V-SENSE Volumetric Video Dataset. *ISO/IEC JTC1/SC29/WG07 MPEG2021/m56767* (2021).
[36] Jesse Read, Luca Martino, Pablo M Olmos, and David Luengo. 2015. Scalable multi-output label prediction: From classifier chains to classifier trellises. *Pattern Recognition* 48, 6 (2015), 2096–2109.
[37] NVIDIA. [n. d.]. NVIDIA Video Codec SDK Website. https://developer.nvidia.com/nvidia-video-codec-sdk. ([n. d.]). Online. Accessed: Sep 2022.
[38] Open3D. [n. d.]. Open3D Website. http://www.open3d.org/. ([n. d.]). Online. Accessed: Sep 2022.
[39] Volograms. [n. d.]. Volograms. https://www.volograms.com/. ([n. d.]). Online. Accessed: Sep 2022.
[40] Szymon Rusinkiewicz and Marc Levoy. 2001. Efficient variants of the ICP algorithm. In *Proceedings third international conference on 3-D digital imaging and modeling*. IEEE, 145–152.
[41] Patrick Stotko, Stefan Krumpen, Matthias B Hullin, Michael Weinmann, and Reinhard Klein. 2019. SLAMCast: Large-scale, real-time 3D reconstruction and streaming for immersive multi-client live telepresence. *IEEE transactions on visualization and computer graphics* 25, 5 (2019), 2102–2112.
[42] Ting-Chun Wang, Jun-Yan Zhu, Nima Khademi Kalantari, Alexei A. Efros, and Ravi Ramamoorthi. 2017. Light Field Video Capture Using a Learning-Based Hybrid Imaging System. *ACM Trans. Graph.* 36, 4, Article 133 (jul 2017), 13 pages. https://doi.org/10.1145/3072959.3073614
[43] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.
[44] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. 2003. Overview of the H.264/AVC video coding standard. *IEEE Transactions on circuits and systems for video technology* 13, 7 (2003), 560–576.
[45] Lan Xu, Zhuo Su, Lei Han, Tao Yu, Yebin Liu, and Lu Fang. 2020. UnstructuredFusion: Realtime 4D Geometry and Texture Reconstruction Using Commercial RGBD Cameras. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 10 (2020), 2508–2522. https://doi.org/10.1109/TPAMI.2019.2915229
[46] Anlan Zhang, Chendong Wang, Bo Han, and Feng Qian. 2022. {YuZu}:{Neural-Enhanced} Volumetric Video Streaming. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 137–154.

A APPENDIX

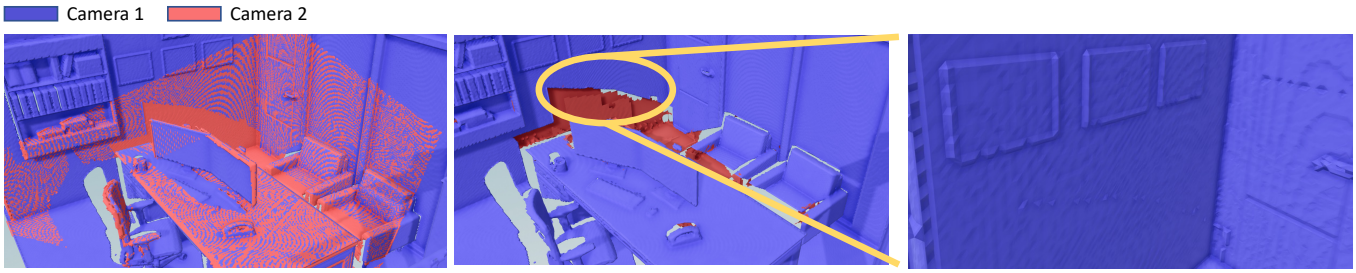
As part of our benchmark studies in §2, we create dataset consisting of RGB-D, point cloud and textured mesh and open it to the public further exploration of 3D content understanding. Creating such a database for a systematic comparison between the point cloud and mesh representation requires: 1) both point cloud and mesh frames should be perfectly synchronized i.e., they should represent the exact same scene; 2) reference frames that are used to compare the quality with after distorting (i.e., compressing or downscaling) the frames. We use Azure Kinect DK and its SDK [1] to capture synchronized RGB-D frames from four cameras and extract point clouds and textured meshes. Our dataset mainly consists of small-scale scenes such as conference rooms, office spaces, hallways, kitchen, corridors, etc. We have a total of 30 scenes with clips ranging from a few seconds to minutes. Overall, our to-be-released dataset contains the following:

- Reference textured meshes in the format of '.obj' (mesh) + '.yuv' (texture) + '.mtl' (color to mesh mapping) files.
- Reference colored point clouds in the format of '.ply' files.
- A collection of decimated and compressed textured mesh sequences in the format of '.drc' and '.mp4' for mesh and texture video sequences respectively.
- A collection of compressed colored point cloud sequences in the format of '.bin' having texture and geometry.
- Matterpak scans of the space (from a Leica BLK360) and/or iPhone Lidar scans for future use.
- A set of camera test poses for our MV-SSIM metric.

B MESH MERGING DETAILS

From a high level, `MeshReduce` accurately identifies and removes overlapping mesh regions in the presence of sensor noise and then connects edges of the non-overlapping, independent meshes. To find the overlapping regions accurately, we leverage a standard computer graphics technique called `raycasting`, emitting rays from the camera perspective and determining if the rays intersect with objects along its path. If there is more than one ray-object intersection along the ray path, we classify if the intersected object is to be removed based on distances between intersections. Essentially if two surfaces are very close together, we assume they are the same surface viewed by different cameras (with some noise-related error). The ray-object intersection distance threshold is determined based on sensor calibration error and noise characteristics. We repeat this process by casting $\mathcal{K}$ rays for each camera, and for all N cameras in the capture site, meaning a total of $\mathcal{K} \times N$ rays. The choice of $\mathcal{K}$ is determined by the camera texture atlas generation (discussed in §3.3).

Once we remove mesh overlapping regions, the next step is to merge the independent meshes to form a complete 3D scene. A straightforward approach is to concatenate different



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

Figure 18: (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 requirement and negatively affects 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 volumetric scene.

meshes together. However, we observe that this produces visual artifacts due to sensor noise and calibration errors. So instead, we use the nearest neighbor search to identify regions between different independent meshes, connecting and smoothing the nearest triangles into new ones. This approach produces smooth and visually complete scene representation. Again, based on sensor noise characteristics, we define a nearest neighbor search threshold χ to merge those triangles.

C TEXTURE PROCESSING DETAILS

The goal of this process is twofold: One, creating a 3D to 2D texture map, so that a rendering client can use the map to project the texture onto the geometry during rendering. Second, creating a texture atlas by removing the camera views that are either not in depth camera field of view or redundant views that also seen by other cameras. Leveraging the same raycasting technique presented in §3.2 and reusing the results, we prepare a texture atlas by removing parts of the texture from each camera view. For each camera, we perform a hit test by shooting a ray from each pixel onto the geometry. There are two outcomes from the raycast: i) if there is no hit, that means there is no triangle that uses this pixel and can be discarded, ii) if there is one or more hits, and if the first hit does not match the triangle, we can discard the pixel because the hit triangle is within the field of the camera but is blocked by some other object in the front the camera. However, suppose the first hit of the multiple hits matches the triangle coordinates. In that case, that means it is a potential candidate texture pixel for the triangle, but not necessarily so because the triangle can also be in the view of other cameras.

A naive way to determine this is by using traditional angle test [8], where a camera more aligned with the mesh surface is selected to minimize visual distortion. However, this results in a spatially incohesive texture atlas where neighboring pixels can be selected from different cameras, and results in inefficient texture compression because of the block based 2D video coding methods (e.g., H.264/VP9 [5, 44]). Instead, we prioritize the spatial coherency over angle distortion by selecting same camera as its neighbor pixel camera if all the overlapping cameras are within certain angle threshold. However, if the cameras far apart that introduces high texture map distortion, and therefore we prefer angle test over spatial coherency. Here, the number of rays hit i.e., $\mathcal{K}$ is equal to the number of total pixels of all cameras. Once we repeat this process for the all pixels in each camera, we remove the texture in deselected regions of each camera. We do this by simply replacing every color with a single pixel color for implementation flexibility while in theory can be completely removed. An example of our spatially texture atlas can be see in Figure 9.