TileSplat: Practical System for Real-Time Streaming and Browser Rendering of City-Scale Photorealistic 3D Scenes

Anonymous Author(s)

Abstract

There is a growing demand for high-fidelity, interactive 3D web content, spurred by its adoption in fields like digital twins, cultural heritage, and e-commerce. However, 3D Gaussian Splatting (3DGS), the state-of-the-art technique for generating such photorealistic scenes, typically produces monolithic, multi-gigabyte files that are fundamentally incompatible with the progressive loading and resource limits of standard web browsers. To bridge this gap, we present TileSplat, an end-to-end system to enable the seamless streaming and real-time rendering of city-scale scenes on the web.

Our contributions are threefold. First, TileSplat novelly splits massive 3DGS data into a spatial tile hierarchy with quality-preserving levels of detail (LOD), maintaining high visual fidelity across view distances. Second, we propose adaptive streaming protocol combining real-time trajectory prediction with adaptive quality selection under varying network conditions. Third, our WebGL-optimized rendering pipeline with hierarchical cache management achieves 30+ FPS rendering on city-scale scenes. Extensive evaluation demonstrates TileSplat uniquely renders scenes that cause competing systems to fail (out-of-memory), achieving superior visual quality with 2.2× bandwidth reduction versus commercial progressive streaming and 4.3× versus full-scene loading. Trajectory prediction improves cache hit rates from 68.3% to 91.2%, reducing time-to-good-quality by 42%. These results validate that tile-based streaming with predictive prefetching successfully deploys photorealistic 3D content within browser constraints. Demo can be found at: https://anonymous.4open.science/r/TileSplat-7982/.

CCS Concepts

• **Information systems** → **Data streams**; **Web applications**; **Multimedia streaming**; **Data compression**; • **Computing methodologies** → *Reconstruction*; *Rendering*.

Keywords

Stream Processing, Web Performance, Data visualization, Data compression, 3D Gaussian Splatting

1 Introduction

The demand for photorealistic, interactive 3D content on the web [1] is accelerating rapidly across diverse domains such as digital twins, cultural heritage [2], virtual tourism [3], and e-commerce experiences [4]. Unlike traditional polygon-based representations [5, 6], 3D Gaussian Splatting (3DGS) [7] has emerged as a powerful technique for generating photorealistic scene reconstructions from multi-view imagery, achieving real-time rendering quality that rivals offline path tracing [8], surpassing traditional polygons in visual fidelity (Fig. 1), and attracting extensive attention. However, a fundamental deployment barrier prevents these high-fidelity scenes from reaching web users: *3DGS produces monolithic, multi-gigabyte files that are fundamentally incompatible with the progressive loading paradigm and resource constraints of modern web browsers.* A typical city-scale 3DGS scene contains millions of Gaussian primitives and easily exceeds 10GB in size—far beyond the 1-2GB JavaScript heap limits imposed by browser engines, and impractical to transmit over networks where users expect sub-second time-to-first-render.

Progressive loading and level-of-detail (LOD) [9] techniques have enabled efficient web visualization of large-scale polygonal meshes and point clouds—often via standards such as glTF [10] and OGC 3D [11] Tiles. However, these paradigms do not apply directly to 3DGS. Unlike discrete primitives, 3DGS models scenes as overlapping, semi-transparent Gaussian distributions that require depth-sorting and alpha blending for correct rendering. This fundamental difference invalidates the spatial partitioning and simplification assumptions underlying existing web 3D streaming systems.

Web-Specific System Challenges. Deploying large-scale 3DGS in web browsers presents interconnected system challenges that are distinct from native application deployment, requiring web-native architecture innovations, not straightforward JavaScript ports:

Browser memory constraints: Modern browsers impose strict heap memory limits (~1-2GB) to prevent tab crashes and maintain system stability. A single high-quality 3DGS scene can require 8-12GB when fully loaded. Consequently, streaming must retain only visible data in memory. However, unlike mesh LOD where coarser representations naturally reduce memory footprint, 3DGS's overlapping, view-dependent primitives make selective loading technically challenging without introducing visual artifacts.

Network transmission latency: Transmitting multi-gigabyte scenes over HTTP introduces prohibitive startup delays. While progressive mesh transmission can display coarse geometry quickly and refine iteratively, 3DGS lacks this graceful degradation: uniform primitive subsampling creates severe artifacts, such as holes in coverage (Appendix A.2). Further, current powerful compression methods [12, 13] could reduce final size but are not optimized for progressive delivery and can still require minutes to fetch.

WebGL API and rendering pipeline constraints. WebGL 2.0 imposes strict buffer size limits that the common vertex attribute approach quickly exhausts for city-scale scenes with millions of Gaussians. Rendering high-quality 3D scenes in real-time is challenging in JavaScript's execution model despite Web Workers.

TileSplat addresses above challenges via following innovations:

Quality-preserving spatial tile hierarchy. We introduce a tile-based spatial partitioning and level-of-detail structure specifically designed for Gaussian primitives. Unlike geometric LOD approaches that simplify surface meshes, our method employs weighted clustering to aggregate Gaussians while preserving visual appearance across view distances. This hierarchy is organized by Mercator tile scheme, enabling HTTP/2 streaming and browser cache integration.

Adaptive streaming protocol with trajectory-based prefetching. We design a streaming protocol that proactively fetches tiles based on predicted camera movement. A lightweight Temporal

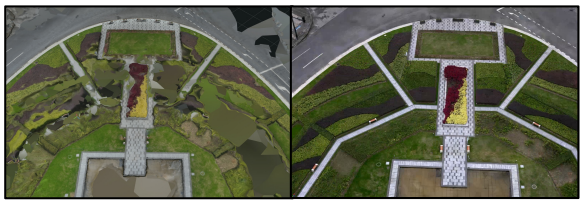

Figure 1: Visual Quality Comparison between Traditional Polygon and 3D Gaussian Splatting.

Convolutional Network executes in-browser via ONNX Runtime, predicting future trajectories to determine tile visibility times. These predictions drive a deadline-constrained optimization problem that selects tile quality levels maximizing perceptual utility while respecting bandwidth, browser memory limits, and fetch deadlines, achieving bandwidth-adaptive delivery without quality oscillations.

WebGL-optimized rendering pipeline. We propose a browser-side renderer that avoids large, implementation-dependent VBO allocations, which we observed to be unstable in practice around 256 MiB, and implements a spatially aware cache management strategy, achieving ≥ 30 FPS on city-scale scenes on consumer hardware.

In summary, we propose the first end-to-end system for streaming and rendering city-scale 3D Gaussian Splatting scenes in web browsers. Our extensive evaluations indicate that TileSplat can achieve high visual quality, bandwidth efficiency, and interactive performance simultaneously on scenes that exceed browser memory limits. We highlight key findings from evaluations as follows:

- TileSplat is the only system that successfully renders the extreme large-scale dataset on the web. It achieves higher visual quality of SSIM (0.946–0.966) across all evaluated scenes while maintaining competitive frame rates (28–36 FPS), outperforming commercial progressive streaming by ∼10% in SSIM.
- TileSplat could achieve high quality with only small fraction of scene data, achieving 2.2× bandwidth reduction versus Polycam and 4.3× reduction versus full-scene loading through FOV-aware tile prioritization.
- TileSplat's trajectory prediction achieves 91.2% cache hit rate versus reactive loading's 68.3%, reducing time-to-good-quality by 42% while incurring acceptable 12.5% wasted downloads.
- TileSplat's deadline-constrained quality selection achieves 2–9% higher utility with 3–16× lower miss rates compared to greedy baseline methods across varying network conditions.

2 Related Work

The web's rise as an interactive 3D platform has driven extensive research into a core problem: how to efficiently stream and render massive, photorealistic scenes within constraints of a web browser.

2.1 Web-Based 3D Streaming Systems

Progressive streaming of massive 3D content to web browsers has evolved through standardized formats and hierarchical spatial organization. glTF 2.0 [14] emerged as the widely-adopted format for 3D asset transmission, with Lemoine et al. [15] demonstrating progressive streaming at the bufferView granularity to enable client-side rendering during download. For geospatial applications, the OGC 3D Tiles standard [16] provides a specification for streaming heterogeneous 3D datasets with hierarchical LOD and implicit tiling, implemented in production systems like CesiumJS [17]. Schilling et al. [10] established early patterns for converting CityGML city models to glTF for WebGL streaming, while Plesch and McCann [18] recently explored interoperability between X3D and 3D Tiles standards. For point clouds, Subramanyam et al. [19] demonstrated that user-centered adaptive delivery with spatial tiling achieves up to 57% bitrate reduction through viewport-aware quality selection. These systems primarily target traditional mesh geometries or sparse point clouds. Despite initial attempts to support dense, photorealistic Gaussian primitives, they still struggle to deliver a seamless experience due to the massive data volume.

2.2 Adaptive Streaming and Quality Selection

Extending MPEG-DASH principles to volumetric content enables bandwidth-adaptive 3D delivery in browsers. Forgione et al. [20] pioneered DASH-based streaming for 3D networked virtual environments, organizing polygon meshes with view-dependent utility metrics compatible with standard CDN infrastructure. Van der Hoort et al. [21] introduced PCC-DASH for point cloud streaming with rate allocation heuristics considering user position and bandwidth. Recent work by Lim et al. [22] presents WIDE-VR, a complete web-based VR system combining DASH-like adaptive streaming with viewport prediction for 6DoF content, while Farrugia et al. [23] provide an open-source framework comparing multiple utility metrics and selection strategies for 3D mesh streaming. Heidarirad and Wang [24] demonstrate VV-DASH achieving 37.8% higher streaming bitrate for volumetric video through temporal segmentation. These works facilitate streaming large-scale 3D environments by dynamically adjusting quality based on network bandwidth.

2.3 Trajectory Prediction for Proactive Fetching

Predictive viewport and trajectory models enable proactive content prefetching, reducing visible pop-in during navigation. For 360° video, Chopra et al. [25] integrate object trajectories with user head movement patterns achieving 30% QoE improvement through tile-based prefetching, while Tian et al. [26] employ transformer-based cross-modal fusion of trajectory and saliency features maintaining 0.662 precision at 1-second horizons. For VR environments, Lemic et al. [27] apply GRU networks for movement prediction under redirected walking, demonstrating that incorporating virtual world context enhances accuracy. Hou et al. [28] predict both head and body motion for 6DoF VR using 840K+ samples, enabling predictive pre-rendering for mobile devices. These techniques translate directly to predicting camera trajectories in 3D web viewers, where multi-modal approaches combining movement history, scene structure, and gaze patterns inform which tiles require prefetching at which quality levels. However, existing models either require heavy compute resources unsuitable for in-browser execution or lack .

2.4 3D Gaussian Splatting on the Web

3D Gaussian Splatting[1] [7] achieves photorealistic real-time rendering but generates large scene files ($\geq 100MB$) requiring compression and streaming for web deployment. WebGL implementations by Kwok [29] pioneered browser-based rendering using asynchronous CPU sorting in web workers, while Tyszkiewicz [30] demonstrated rasterization-based approaches achieving interactive

[1] More details are illustrated in Appendix A

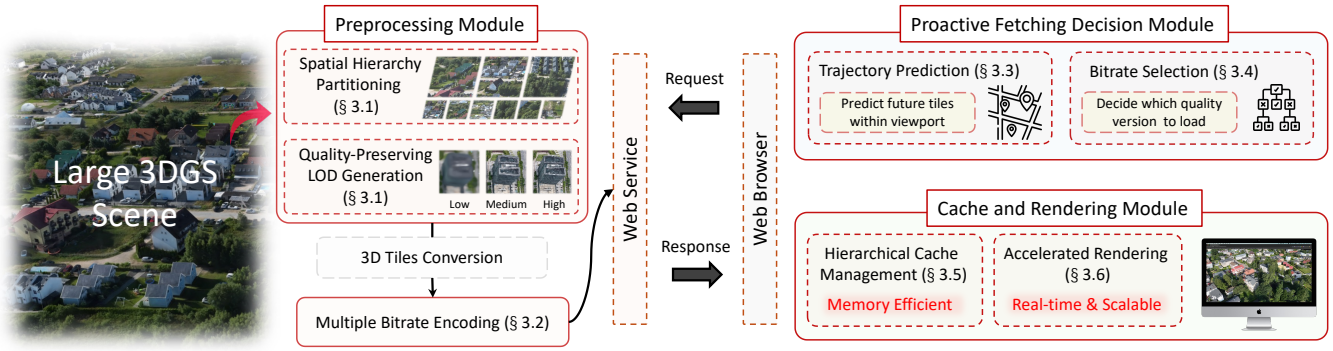

Figure 2: Overview of the proposed TileSplat.

frame rates. Compression techniques prove essential: Niedermayr et al. [31] achieve 31× reduction through sensitivity-aware quantization, while Chen et al. [13] demonstrate context-based compression exceeding 75× reduction using hash-grid mutual information. For streaming, Sun et al. [32] present LTS-DASH achieving 99.7% reduction in missing frames through multi-layer, tiled, segment-based adaptive delivery, while Shi et al. [33] propose layered progressive representations adapted to bandwidth constraints. Despite these advances, existing works fail to scale effectively to gigabyte-scale 3DGS scenes for smooth web-based streaming and rendering.

While prior work has focused on individual components, TileSplat is the first end-to-end system specifically designed to stream and render city-scale 3D Gaussian Splatting scenes in web browsers. Our key contribution is an integrated approach that combines a *progressive layered tiled* data structure with a *viewport-aware, quality-adaptive streaming* protocol. This allows our system to proactively fetch and render the necessary data in real-time, addressing the complete challenge from data preprocessing to final rendering.

3 TileSplat

We present TileSplat, the first streaming system for city-scale 3DGS scenes in web browsers. Fig. 2 provides an overview of our system.
Server-side preprocessing generates a spatial tile hierarchy across multiple zoom levels, with each tile encoded at four quality levels through Mercator-based spatial partitioning, quality-preserving LOD generation via weighted clustering (§ 3.1), and progressive quantization (§ 3.2). Output tiles conform to OGC 3D Tiles standards and are served as static HTTP assets.
Client-side streaming predicts future camera trajectories using a lightweight TCN model (§ 3.3), then solves a deadline-constrained optimization (§ 3.4) to determine which tiles to fetch at which quality levels. The optimization maximizes visual quality while respecting bandwidth constraints, browser memory limits, and fetch deadlines derived from predicted visibility times. Downloaded tiles are cached using hierarchical strategy (§ 3.5) and rendered via customized WebGL 2.0 pipeline (§ 3.6).

3.1 Spatial Tile Hierarchy and LOD Generation

3.1.1 Mercator-Based Spatial Partitioning. We adopt the Mercator tile scheme used by web mapping systems (OpenStreetMap [34], Google Maps), providing a natural geographic coordinate system for world-scale scenes. Each zoom level z corresponds to a $2^z \times 2^z$ grid of tiles, with spatial resolution doubling at each successive level. The resolution at $z = 20$ is set to 0.1 m; thus, at an arbitrary level $z = l$, the resolution equals $0.1 \times 2^{20-l}$ meters. In our experiments, we use zoom levels $z \in \{14, 15, \ldots, 20\}$. Level 14 offers a coarse overview, containing less than 1% of the total primitives and loading almost instantly, while level 20 can cover areas exceeding 100 km² at high detail. Given a 3DGS scene containing N Gaussian primitives, each primitive p_k with position $\mathbf{x}_k = (x_k, y_k, z_k)$ is assigned to tile (z, t_x, t_y) based on its Mercator coordinates:

$$t_x = \lfloor 2^z \cdot \frac{\lambda + \pi}{2\pi} \rfloor, \quad t_y = \lfloor 2^z \cdot \frac{\pi - \ln(\tan(\pi/4 + \phi/2))}{2\pi} \rfloor \quad (1)$$

where (λ, ϕ) are the longitude and latitude derived from world coordinates. This spatial hashing enables parallel tile generation and naturally clusters nearby primitives. Primitives are assigned to tiles based on their center position. Fig. 3 demonstrates how a 3DGS scene (left) is partitioned (right).

3.1.2 Quality-Preserving LOD Hierarchy. Directly subsampling Gaussian primitives produces *severe visual artifacts* due to loss of volumetric coverage (Appendix A.2). We instead generate parent tiles at lower zoom levels through weighted clustering that preserves visual appearance. For each parent tile at zoom level $z - 1$, we collect all child primitives from corresponding z-level tiles. Using a KD-tree spatial index, we cluster primitives within distance threshold $d = r \cdot 2^{20-z}$ meters, where $r = 0.1$m is the base resolution parameter. For each cluster $C = \{p_1, p_2, \ldots, p_k\}$, we compute a representative Gaussian p^* through alpha-weighted averaging.

Denote $\boldsymbol{\mu}_i \in \mathbb{R}^3$, $\mathbf{q}_i \in \mathbb{H}$, $\mathbf{s}_i \in \mathbb{R}^3$, $\mathbf{c}_i \in \mathbb{R}^3$, and $\alpha_i \in [0, 1]$ as the position, rotation quaternion, scale, RGB color, and opacity of primitive p_i, respectively. The primitive attributes are computed as:

$$\boldsymbol{\mu}^* = \frac{\sum_{i=1}^k \alpha_i \boldsymbol{\mu} i}{\sum i = 1^k \alpha_i}, \quad \mathbf{q}^* = \text{QuatAvg}(\mathbf{q}i, \alpha_i), \quad \mathbf{c}^* = \frac{\sum i = 1^k \alpha_i \mathbf{c} i}{\sum i = 1^k \alpha_i},$$
$$\mathbf{s}^* = \frac{1}{2}\left(\max_{i \in C}(\boldsymbol{\mu}_i + \mathbf{s}i) - \min i \in C(\boldsymbol{\mu} i - \mathbf{s}i)\right), \quad \alpha^* = \frac{\sum i = 1^k \alpha_i^2}{\sum i = 1^k \alpha_i} \quad (2)$$

where QuatAvg denotes weighted quaternion averaging using Markley's eigenvector method. The scale $\mathbf{s}^*$ is computed as half the bounding box diagonal encompassing all primitives in the cluster, ensuring complete spatial coverage without gaps. Alpha-weighted averaging ensures visually important (high-opacity) primitives dominate the representative.

Our clustering reduces the primitive count by 10–50% per level, generating six distinct LOD, spanning zoom levels 20 to 14. As shown in Fig. 4, higher LODs preserve finer geometric details. The root level contains approximately 0.08% of the original primitives while retaining a recognizable scene structure. This approach avoids

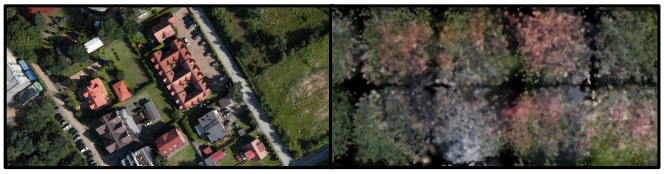

Figure 3: Mercator-Based Spatial Tile Hierarchy Partitioning

Figure 4: Comparison between different level of details.

the loss of volumetric coverage typically associated with conventional downsampling methods.

3.2 Multi-Quality Tile Encoding

Each tile is encoded at $M = 4$ quality levels (Best, High, Medium, Low) through progressive quantization of Gaussian attributes. This enables bandwidth-adaptive streaming where tile quality is selected based on network conditions and predicted viewport importance.

3.2.1 Progressive Quantization. We quantize each Gaussian attributes with progressively fewer bits across quality levels and the detailed quantization procedures are in Appendix B. Quantized attributes are stored column-wise and compressed with gzip, achieving 40-60% additional reduction.

3.2.2 Visual Quality Characterization. For quality selection, we quantify perceptual quality by rendering each compressed tile from 8 viewpoints and computing SSIM [35] against uncompressed one:

$$u_{ij} = \frac{1}{|\mathcal{V}_i|} \sum_{v \in \mathcal{V}_i} \text{SSIM}(I_{ij}^{(v)}, I_{\text{ref}}^{(v)}) \tag{3}$$

where $\mathcal{V}_i$ is tile i's viewpoint set. Position quantization limits SSIM up to ~14 bits; beyond that, rotation and color errors dominate. Medium quality yields SSIM > 0.95; Low quality degrades to SSIM $\approx$ 0.88-0.92. These utility scores u_{ij} drive the optimization (§ 3.4).

3.3 Trajectory Prediction for Proactive Fetching

Reactive tile fetching (fetching when tiles enter viewport) causes visible pop-in artifacts. We predict future camera trajectories using a learned model, enabling proactive tile fetching with sufficient lead time for network transmission.

3.3.1 Problem Formulation. Let $\mathbf{p}(t) = (x(t), y(t), z(t)) \in \mathbb{R}^3$ denote camera position and $\theta(t) \in [0, 2\pi)$ the yaw angle at time t. The camera state is $\mathbf{s}(t) = [\mathbf{p}(t), \theta(t)] \in \mathbb{R}^4$. Velocity is implicit through temporal position changes.

Given history window $H = 5$ seconds with sampling interval $\Delta t = 83$ms (12 Hz), the trajectory sequence is:

$$\mathcal{S}_t^H = \{\mathbf{s}(t - H), \mathbf{s}(t - H + \Delta t), \ldots, \mathbf{s}(t)\} \tag{4}$$

We predict future trajectory over horizon T seconds:

$$\hat{\mathcal{S}}_t^T = \{\hat{\mathbf{s}}(t + \Delta t), \hat{\mathbf{s}}(t + 2\Delta t), \ldots, \hat{\mathbf{s}}(t + T)\} \tag{5}$$

For each predicted state $\hat{\mathbf{s}}(t')$, we compute potentially visible tiles via frustum-OBB intersection:

$$\mathcal{T}_{\text{visible}}(t') = \{i \in \mathcal{I} : \text{OBB}_i \cap \mathcal{F}(\hat{\mathbf{p}}(t'), \hat{\theta}(t')) \neq \emptyset\} \tag{6}$$

where $\mathcal{F}(\mathbf{p}, \theta)$ is the viewing frustum. The predicted tile set is $\mathcal{T}_{\text{pred}} = \bigcup_{t'=t+\Delta t}^{t+T} \mathcal{T}_{\text{visible}}(t')$. For each tile $i \in \mathcal{T}_{\text{pred}}$, we record its first predicted visibility time feeds into the optimization framework (§ 3.4) to compute fetch deadlines.

3.3.2 Lightweight TCN Architecture for Browser Execution. The predictor must run in-browser with minimal latency to enable real-time navigation. We design a compact Temporal Convolutional Network (TCN) [36] that achieves 87% tile coverage accuracy while maintaining < 100ms inference latency on consumer CPUs, enabling execution in Web Workers without blocking the rendering thread.
Encoding. The architecture comprises an encoder processing $\mathcal{S}_t^H$ through $L = 4$ residual blocks with 128 channels and exponentially increasing dilation rates $d_l = 2^l$. Each residual block applies dilated causal convolutions:

$$\mathbf{h}^{(l)} = \text{ReLU}(\mathbf{W}_2^{(l)} * \text{ReLU}(\mathbf{W}_1^{(l)} * \mathbf{h}^{(l-1)})) + \mathbf{h}^{(l-1)} \tag{7}$$

where $*$ denotes dilated causal convolution and $\mathbf{h}^{(0)} = \mathcal{S}_t^H$. With kernel size $k = 3$ and exponentially increasing dilations, the receptive field spans:

$$\text{RF} = 1 + 2(k-1)\sum_{l=1}^{L} 2^l = 1 + 2(k-1)(2^L - 1) = 61 \text{ timesteps} \tag{8}$$

where the exponential dilation enables efficient long-range temporal modeling.
Decoding. Next, an LSTM decoder generates multi-step predictions autoregressively, outputting probabilistic predictions:

$$\hat{\mathbf{s}}(t + k\Delta t) \sim \mathcal{N}(\boldsymbol{\mu}_k, \text{diag}(\boldsymbol{\sigma}_k^2)) \tag{9}$$

Optimization. The model is trained using Gaussian Negative Log-Likelihood loss:

$$\mathcal{L} = \frac{1}{N_T} \sum_{k=1}^{N_T} \left[\frac{1}{2} \log(\boldsymbol{\sigma}_k^2) + \frac{(\mathbf{s}_{\text{true}}(t + k\Delta t) - \boldsymbol{\mu}_k)^2}{2\boldsymbol{\sigma}_k^2} \right] \tag{10}$$

where $N_T = \lfloor T/\Delta t \rfloor$ is the number of prediction steps. This loss encourages accurate mean predictions while learning appropriate uncertainty estimates.
Web deployment. The model is trained in PyTorch then exported to ONNX format for browser deployment via ONNX Runtime Web, which enables cross-platform execution with minimal overhead.

3.4 Deadline-Constrained Quality Selection

Given predicted tiles $\mathcal{T}_{\text{pred}}$ and their visibility times from §3.3, we determine which quality version of each tile to download, maximizing visual quality subject to bandwidth, memory, and deadline constraints. We formulate this as an optimization problem and solve it optimally via dynamic programming.

3.4.1 Constraint Derivation from Predictions. For each predicted tile $i \in \mathcal{T}_{\text{pred}}$, the trajectory predictor provides the first visibility time $t_{\text{first}}(i)$. We define the fetch deadline as:

$$t_i = t_{\text{first}}(i) - \frac{s_{ij}}{\hat{B}} \tag{11}$$

where s_{ij} is the size of tile i at quality j, and $\hat{B}$ is the estimated bandwidth (exponentially weighted moving average over recent downloads, updated every 500ms).

3.4.2 Optimization Formulation. Let $\mathcal{I} = \{1, \ldots, N\}$ denote predicted tiles sorted by deadline t_i in ascending order, and $\mathcal{J} = \{1, \ldots, M\}$ the quality levels ($M = 4$). Decision variables $x_{ij} \in \{0, 1\}$ indicate whether tile i is downloaded at quality j. Each version has utility $u_{ij} \in [0, 1]$ (SSIM score from §3.3) and size s_{ij} bytes.

We model time-varying bandwidth as piecewise constant $\hat{B}(t)$ based on recent throughput measurements. Download duration for tile i at quality j completing at time τ is:

$$d_{ij}(\tau) = \min \left\{ d \in \mathbb{Z}_+ : s_{ij} \le \sum_{t=\tau-d}^{\tau} \hat{B}(t)\Delta t \right\} \tag{12}$$

Let T_k denote cumulative time to download the first k tiles. The optimization problem is:

$$\max_{x_{ij}} \quad \sum_{i=1}^{N} \sum_{j=1}^{M} u_{ij} x_{ij} \tag{13}$$

$$\text{s.t.} \quad T_i \le t_i \quad \forall i \in \mathcal{I} \tag{14}$$

$$\sum_{j=1}^{M} x_{ij} = 1 \quad \forall i \in \mathcal{I} \tag{15}$$

$$\sum_{i=1}^{N} \sum_{j=1}^{M} s_{ij} x_{ij} \le M_{\text{cache}} \tag{16}$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \tag{17}$$

Constraint (14) requires each tile to complete before its deadline (derived from predicted visibility). Constraint (15) ensures exactly one quality per tile. Constraint (16) bounds total memory by cache capacity M_{cache}.

3.4.3 Dynamic Programming Solution. With fixed download order (tiles sorted by t_i), we define state $U(i, \tau)$ as the maximum utility achievable for the first i tiles if they complete by time τ. We discretize time with granularity $\delta = 100$ms, balancing deadline precision with computational efficiency. The recurrence is:

$$U(i, \tau) = \max \begin{cases} U(i-1, \tau) & \text{(skip tile } i\text{)} \\ \max_{j \in \mathcal{J}} \{U(i-1, \tau - d_{ij}(\tau)) + u_{ij}\} & \text{if } \tau \le t_i \end{cases} \tag{18}$$

with boundary $U(0, \tau) = 0$. The optimal solution is $U(N, t_N)$.

Complexity Analysis. With time discretization $\delta = 100$ms and prediction horizon $T_{\max} = 60$s, the algorithm evaluates $O(NM \cdot T_{\max}/\delta) = O(50 \cdot 4 \cdot 600) = 120,000$ states, completing in < 100ms on consumer CPUs.

Figure 5: Hierarchical cache management with prioritized parent tile preservation. Parent tiles receive reduced eviction priority to maintain visual continuity during network latency and rapid viewpoint changes.

3.5 Hierarchical Cache Management

Downloaded tiles are stored in an LRU cache with capacity $M_{\text{cache}} = 1$GB, ensuring compatibility across browsers while providing sufficient buffering for city-scale scenes. When capacity is exceeded, tiles are evicted based on spatial distance and frustum visibility:

$$\mathcal{E} = \{i \in C : \|\mathbf{p} - \mathbf{c}_i\| > 150\text{m} \vee i \notin \text{Frustum}(\mathbf{p}, \theta)\} \tag{19}$$

where C is the cache contents and $\mathbf{p}$ is the current camera position. This eviction strategy is augmented by a critical fault-tolerant mechanism: parent tiles at lower zoom levels receive systematically reduced eviction priority. As illustrated in Fig. 5, this hierarchical approach ensures that unvisited or distant regions retain coarse parent tiles, recently visited or nearby areas maintain intermediate LOD tiles, and the currently visible region is rendered at the highest available quality. This hierarchical preservation strategy maintains visual continuity with minimal overhead during rapid viewpoint movement, prevents rendering artifacts during network latency, and delivers a seamless navigation experience across diverse network conditions.

3.6 Rendering at Scale in WebGL

Downloaded tiles are rendered with a custom WebGL 2.0 pipeline that avoids large, implementation-dependent VBO allocations. We store Gaussian attributes in RGBA data textures and use chunked buffers instead of monolithic vertex-attribute buffers. In our tests across mainstream browsers and GPUs, single-buffer allocations became unstable around a few hundred MiB ($\approx$ 256MiB); we therefore adopt this texture-based, chunked design for robustness. This threshold is mandated by the WebGL specification. Additionally, correct alpha blending requires back-to-front rendering order. We employ a lazy sorting strategy that triggers depth-based sorting only when camera position changes exceed 2m or orientation changes exceed 15 degrees. Hierarchical frustum culling on the tile tree eliminates 70-85% of tiles for typical viewports.

This elaborate design enables rendering of millions of primitives per tile in real-time. Detailed rendering pipeline design and implementation is provided in Appendix A.3.

4 Experiment

4.1 Experimental Setup

Implementation Details. Full implementation details are provided in the appendices. We describe the quantization parameters

for multi-quality tile encoding in Appendix B, the training data and hyperparameter configurations for our trajectory predictor in Appendix C, and our WebGL rendering architecture in Appendix D. The trajectory prediction model was trained on a server equipped with two 10-core Intel Xeon Silver 4210 CPUs, 128GB of RAM, and an NVIDIA A6000 GPU with 48GB of VRAM. Client-side performance was tested on a MacBook Air (M4, 32GB RAM), a standard commercial laptop whose GPU performance is comparable to an NVIDIA RTX 3060.

Benchmark Dataset. We benchmarked our system on the four datasets of varying scales and densities detailed in Table 1. These datasets, ranging from hundreds of megabytes to tens of gigabytes, were preprocessed by removing their spherical harmonic coefficients to reduce data size with minimal impact on visual quality.

Table 1: Datasets across different scales and densities.

Scene	Raw (MB)	Processed (MB)	#GS	Area	Density
shore-large	20104	3358	102M	>10 KM^2	Low
shore-small	2024	587	10M	>2 KM^2	Low
Technopark	482	139	2.5M	≈ 1 KM^2	Medium
Park	691	152	3M	<1 KM^2	High

Baseline Systems. We compare TileSplat against baselines from both academia and industry, selected to represent three distinct architectural strategies for deploying large-scale 3DGS on the web:

- **GSplat [37]**: An open-source library for 3DGS that employs *server-side rendering*. To avoid large-scale data transmission to memory-constrained browsers and reduce client-side computational costs, GSplat renders views on powerful servers in real-time and transmits 2D renderings to clients. This architecture provides instant feedback with high visual quality but introduces network latency and requires continuous server-client communication, making it representative of compute-offloading approaches.
- **SuperSplat [38]**: An open-source tool for editing and publishing 3DGS that uses *full client-side rendering*. SuperSplat requires downloading the entire scene before rendering begins, with no progressive transmission mechanism. This approach eliminates server dependencies and network latency during interaction but faces critical limitations for large-scale scenes: prohibitive initial load times and browser memory constraints that prevent loading scenes exceeding heap limits. It represents traditional client-side rendering without streaming optimizations.
- **Polycam [39]**: A popular commercial application for 3D creation and viewing that implements *spatial-aware progressive downloading*. Polycam attempts to address both memory and network constraints through progressive transmission, sacrificing quality for improved rendering speed and reduced memory footprint. This represents the current state-of-practice in commercial progressive 3D streaming for Web browsers.

4.2 System-Level Performance Analysis

4.2.1 Rendering Quality and Efficiency: Table 2 demonstrates TileSplat's capability to render scenes that exceed browser memory limits. Only TileSplat successfully renders the large-scale Shore dataset. Both GSplat and SuperSplat fail due to out-of-memory errors, while

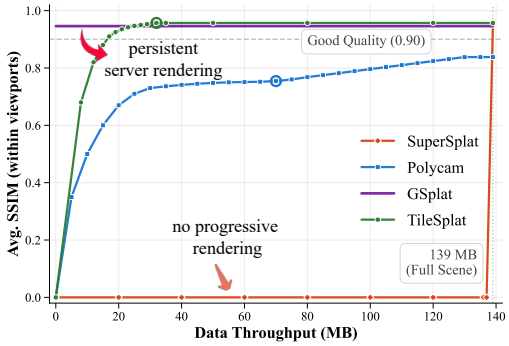

Figure 6: Streaming efficiency on Technopark scene.

Polycam does not permit uploads of this size. Across all scenes, TileSplat achieves superior visual quality with competitive frame rates. Polycam's higher frame rates come at substantial quality cost (~10%) lower, while GSplat's server-side rendering introduces encode-transmit-decode latency that limits performance and raises scalability concerns with concurrent users. These results validate that tile-based streaming with hierarchical cache management successfully addresses the Web deployment challenge: enabling city-scale 3DGS scenes within browser constraints while maintaining both visual fidelity and interactive performance.

4.2.2 Evaluating Priority and LOD Streaming Strategies. To validate our FOV-prioritized streaming, we measure visual quality (average SSIM across 100 sequential viewports) against cumulative data throughput, isolating algorithmic efficiency from network infrastructure variances.

We compare the streaming efficiency on the 139 MB Technopark scene in Fig. 6. TileSplat converges at 32 MB (23% of scene, SSIM 0.957), achieving 2.2× data reduction versus Polycam (70 MB, SSIM 0.838) and 4.3× versus the full scene. The discrete loading pattern reveals our strategy: large initial tile for immediate base quality, followed by progressive refinement through small FOV-aware tiles prioritizing visible high-impact content. Polycam's uniform loading reaches 90% quality at 70 MB before slowly converging to 139 MB. SuperSplat provides no progressive rendering—quality remains zero until the full 139 MB loads. GSplat maintains constant quality (SSIM 0.946) through server-side streaming but requires continuous GPU resources. TileSplat achieves higher quality with one-time 32 MB transfer, enabling offline viewing while reducing infrastructure costs. These results validate that FOV-aware tile prioritization addresses the core Web-scale deployment challenge: achieving acceptable quality within browser memory constraints and network latency budgets that preclude full-scene loading.

Fig. 7 showcases how TileSplat's viewport-aware, progressive strategy dramatically enhances rendering speed. At 100Mbps, TileSplat delivers a recognizable model in 1.6 seconds and a complete view in 3.4 seconds. By contrast, the baseline approach fails to produce a discernible shape even after 17 seconds. Prioritizing the viewport provides a fast, interactive experience, effectively eliminating the long wait times of traditional methods.

4.3 Spatial Partitioning and Progressive Quality.

The distribution of primitives across LODs follows the hierarchical Mercator-based tiling, where tile size and spatial resolution jointly influence per-tile primitive counts. Fig. 8 illustrates that coarser

Table 2: Comparison of Rendering Quality and Efficiency. The best and the second best results are highlighted.

Method	Shore-large				Shore-small				Technopark				Park			
	FPS↑	PSNR↑	SSIM↑	LPIPS↓	FPS↑	PSNR↑	SSIM↑	LPIPS↓	FPS↑	PSNR↑	SSIM↑	LPIPS↓	FPS↑	PSNR↑	SSIM↑	LPIPS↓
Polycam	-	-	-	-	38	20.7	0.836	0.269	45	24.3	0.838	0.292	42	21.6	0.869	0.381
GSplat	OOM	OOM	OOM	OOM	8	33.6	0.935	0.185	14	33.6	0.946	0.194	10	31.0	0.959	0.249
SuperSplat	OOM	OOM	OOM	OOM	14	33.9	0.933	0.177	29	34.2	0.951	0.139	19	31.8	0.962	0.228
TileSplat	28	36.9	0.947	0.120	31	34.2	0.946	0.115	36	34.5	0.957	0.113	29	32.5	0.966	0.174

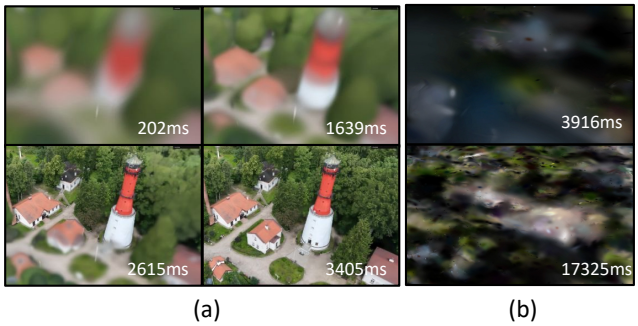

Figure 7: Qualitative results comparing the efficiency of various transmission strategies. (a) Viewport-aware, layered, and incremental progressive transmission. (b) Baseline transmission of the entire scene at once.

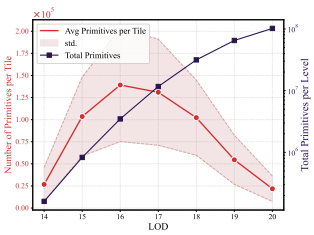

Figure 8: Primitives Distribution Across LODs

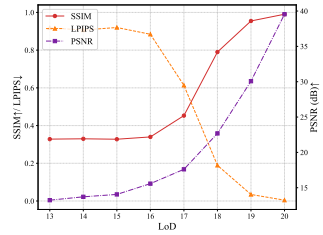

Figure 9: Averaged Visual Quality Across LODs

levels (e.g., 14) feature large tiles covering broad areas but with fewer primitives per tile, while mid-levels (e.g., 16) balance tile size and detail, yielding the highest primitives per tile. The finest level (20) uses many small tiles with fine resolution but fewer primitives per tile due to aggressive subdivision. For streaming, this means mid-level LODs carry dense, larger tiles requiring more bandwidth to deliver rich detail, whereas finer levels consist of numerous lightweight tiles allowing flexible on-demand loading. Coarser levels provide quick, low-data overviews. Overall, this distribution supports adaptive streaming that efficiently balances bandwidth, load times, and visual fidelity based on viewer distance and rendering needs. Fig. 9 demonstrates how visual quality varies, and quality does not noticeably improve until exceeding level 15. This indicates that there is redundancy and streaming efficiency can be improved through careful setting LOD range according to scenario in hand.

4.4 Camera Trajectory Prediction Validation

4.4.1 Prediction Accuracy. In this section, we provide detailed quantitative analysis of prediction performance, and visualization of collected trajectory samples (Fig. 13) and trajectory prediction (Fig. 14) please refer to Appendix C.

Table 3: Trajectory Prediction Accuracy Comparison

Method	ADE (m)↓	FDE (m)↓	Coverage↑	Latency↓
Constant Velocity	12.4	28.4	23%	<1
Linear Extrapolation	6.3	14.1	39%	<1
LSTM (2L, 128)	1.5	3.5	81%	95
Oracle (Perfect)	0.0	0.0	100%	-
TCN (Ours)	**1.2**	**2.8**	**87%**	**58**

Table 4: System Performance Impact of Trajectory Prediction

System Variant	*CHR*↑	*TGQ*↓	*WD*↓
TileSplat Reactive	68.3% ± 5.8	5.5s ± 1.8	3.2% ± 1.2
TileSplat Linear	82.5% ± 4.5	4.1s ± 1.3	18.7% ± 5.2
Oracle (Perfect)	96.8% ± 1.8	2.1s ± 0.5	8.3% ± 2.1
TileSplat (Full)	91.2% ± 3.2	3.2s ± 0.9	12.5% ± 3.5

Baseline Methods and Evaluation Metrics. We compare against Constant Velocity, Linear Extrapolation, LSTM, and Oracle (perfect future knowledge), spanning heuristics to optimal bounds. We evaluate using Average Displacement Error (ADE) [40], Final Displacement Error (FDE) [41], and Tile Coverage Accuracy—the percentage of correctly prefetched tiles that users view, quantifying prefetching effectiveness.

Table 3 evaluates our TCN predictor on held-out trajectories. At 30s horizon, TCN achieves 1.2m ADE, 2.8m FDE, and 87% tile coverage, outperforming Linear Extrapolation (6.3m/14.1m, 39%) and Constant Velocity (12.4m/28.4m, 23%). LSTM matches accuracy (1.5m ADE) but requires 95ms inference versus TCN's 58ms, risking frame drops. Fig. 10 shows TCN degrades gracefully (0.4m→1.2m ADE from 5s→30s), while Linear hits a performance cliff at 15s due to failed constant-velocity assumptions. TCN's dilated convolutions capture both short-term motion and long-range trends through its 61-timestep receptive field.

4.4.2 The Impact of the Trajectory Prediction Module on the System. System Variants and System Performance Metrics. We implement four TileSplat variants: (1) TileSplat-Full—our complete system with TCN prediction; (2) TileSplat-Reactive—loads tiles only when visible; (3) TileSplat-Linear—replaces TCN with Linear Extrapolation; and (4) Oracle—uses perfect future knowledge as upper bound. This ablation reveals whether prediction helps (Reactive vs. Full), whether TCN outperforms simpler methods (Linear vs. Full), and remaining headroom (Full vs. Oracle). We measure Cache Hit Rate (CHR)—percentage of viewed tiles already cached; Time to Good Quality (TGQ)—seconds to reach SSIM ≥ 0.90; and Wasted Downloads (WD)—percentage of fetched tiles never viewed.

Table 4 demonstrates substantial system-level benefits from trajectory prediction. TileSplat-Full achieves 91.2% cache hit rate versus TileSplat-Reactive's 68.3%—when 91% of viewed tiles are already

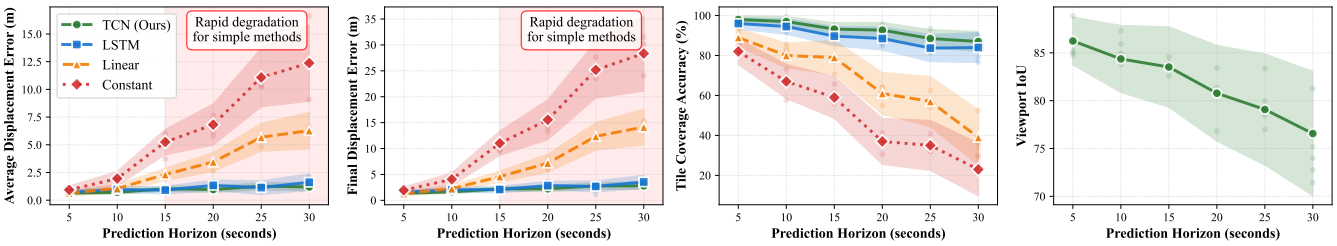

Figure 10: Trajectory Prediction Accuracy vs. Prediction Horizon

Table 5: Optimality Verification: DC-DP vs. Baseline Methods Across Network Conditions

Scenario	Method	Utility	Avg. Quality	Miss (%)	Util/MB	Time (ms)
100 Mbps	DC-DP	**29.34**	**0.978**	**0.0**	0.2445	56.6
	Greedy-Deadline	28.76	0.959	3.3	0.2398	0.12
	Greedy-Quality	29.28	0.976	16.7	0.2112	0.08
	Distance-Based	28.14	0.938	23.3	0.1845	0.02
50 Mbps	DC-DP	**28.62**	**0.954**	**1.7**	**0.2521**	35.7
	Greedy-Deadline	27.45	0.915	10.0	0.2284	0.15
	Greedy-Quality	27.92	0.931	30.0	0.2087	0.09
	Distance-Based	26.84	0.895	36.7	0.1756	0.02
10 Mbps	DC-DP	**27.18**	**0.906**	**3.3**	**0.2847**	59.2
	Greedy-Deadline	25.62	0.854	20.0	0.2134	0.18
	Greedy-Quality	25.98	0.866	46.7	0.1892	0.11
	Distance-Based	25.14	0.838	50.0	0.1645	0.02

cached, users experience near-instantaneous loading, whereas reactive loading causes visible pop-in artifacts. TCN prediction significantly improves the time to acceptable quality, reducing it by 42% from 5.5s to 3.2s and crossing the critical 3-second responsiveness threshold. This proactive prefetching ensures seamless navigation to predicted viewpoints during initial exploration. TileSplat-Linear achieves intermediate performance (82.7% CHR, 3.8s TGQ), confirming simple prediction outperforms on-demand loading but TCN provides measurable additional benefit. The gap between TileSplat-Full (91.2%) and Oracle (96.8%) reveals 6% remaining headroom, suggesting our model captures most predictable navigation patterns. Proactive fetching incurs 12.5% wasted downloads versus reactive's 3.2%, but this 17.4 MB overhead on Technopark is negligible compared to eliminating 5+ second loading delays. These results validate that TCN-based prediction effectively balances prefetching accuracy with bandwidth efficiency for Web-scale streaming.

4.5 Quality Selection under Constraints

We evaluate our DC-DP (Deadline-Constrained Dynamic Programming) quality selection algorithm against four baseline methods across three bandwidth conditions using 30 tiles with deadlines uniformly distributed between 5-30 seconds and four quality levels. The baselines represent different optimization strategies: *Greedy-Deadline* (per-tile optimization, industry standard), *Greedy-Quality* (quality-first), and *Distance-Based* (spatial LOD). Table 5 shows DC-DP achieves 2-9% higher utility with 3-16× lower miss rates (0-3.3% vs. 3.3-50.0%). Utility efficiency (Util/MB) improves under constraint (0.245 at 100 Mbps to 0.285 at 10 Mbps), confirming intelligent quality adaptation. Runtime remains stable at 30-60ms across bandwidths, validating complexity $O(N \cdot M \cdot T/\delta)$ with no bandwidth dependency, meeting the <100ms real-time requirement. As demonstrated in Figure 6, the combined efforts of our layered progressive data structure and efficient DP quality selection result in a huge improvement in overall utility efficiency.

4.6 Discussion: Balancing Cost and Accuracy

Rendering large-scale 3DGS scenes is already computationally expensive—baseline methods frequently encounter out-of-memory failures. We therefore intentionally avoid excessive algorithmic complexity, prioritizing lightweight methods that reduce streaming and rendering overhead to enable city-scale deployment within browser constraints.

Our trajectory prediction (TCN-based) and quality selection (dynamic programming) are deliberately lightweight—a systems engineering decision driven by Web deployment constraints rather than algorithmic limitations. More sophisticated alternatives like Transformer-based predictors [42] or graph neural networks [43] could potentially improve prediction accuracy but introduce prohibitive computational overhead in resource-constrained browsers. Analysis in § 4.2 validates this design choice: our lightweight approach achieves 91.2% cache hit rate, approaching Oracle's 96.8% theoretical upper bound, demonstrating that simple methods achieve near-optimal performance under browser resource budgets.

5 Conclusion

We presented TileSplat, the first end-to-end system for streaming city-scale 3D Gaussian Splatting scenes in web browsers. Unlike existing approaches that either fail on large scenes (OOM errors) or sacrifice substantial visual quality, TileSplat integrates four Web-native architectural innovations to address browser memory constraints, network latency, and WebGL API limitations: (1) Mercator-based tile hierarchy with quality-preserving LOD generation maintaining visual fidelity across scales, (2) lightweight TCN trajectory prediction (87% coverage, 58ms latency) enabling proactive prefetching, (3) deadline-constrained dynamic programming optimally allocating bandwidth across tile quality levels, and (4) texture-based WebGL rendering with hierarchical cache management.

Experimental validation demonstrates TileSplat uniquely renders multi-gigabyte scenes (Shore-large: 3.4GB) while achieving superior quality (SSIM 0.957) with 2.2× bandwidth reduction versus commercial systems and 4.3× versus full-scene loading. Trajectory prediction improves cache hit rates from 68.3% to 91.2%, reducing time-to-good-quality by 42% (5.5s to 3.2s). These results validate that our integrated approach successfully balances visual fidelity, responsiveness, and bandwidth efficiency within browser constraints.

TileSplat's HTTP-based architecture aligns with existing Web infrastructure and 3D standards (OGC 3D Tiles, glTF), enabling immediate deployment via CDNs while providing a foundation for standardizing photorealistic 3DGS delivery as adoption grows across various immersive Web applications.

References

[1] Kristian Sons, Felix Klein, Dmitri Rubinstein, Sergiy Byelozyorov, and Philipp Slusallek. Xml3d: interactive 3d graphics for the web. In *Proceedings of the International Conference on Web 3D Technology*, pages 175–184, 2010.

[2] Tomas Nilson and Kristina Thorell. Cultural heritage preservation: The past, the present and the future, 2018.

[3] Sanjeev Verma, Lekha Warrier, Brajesh Bolia, and Shraddha Mehta. Past, present, and future of virtual tourism-a literature review. *International Journal of Information Management Data Insights*, 2(2):100085, 2022.

[4] Mohammad Al Khaldy, Abdelraouf Ishtaiwi, Ahmad Al-Qerem, Amjad Aldweesh, Mohammad Alauthman, Ammar Almomani, and Varsha Arya. Redefining e-commerce experience: An exploration of augmented and virtual reality technologies. *International Journal on Semantic Web and Information Systems*, 19(1):1–24, 2023.

[5] Miklós Vincze, Marianna Dimitrova Kucarov, Bence Biricz, Abdallah Benhamida, Melvin Ogbolu, Miklós Kozlovszky, Viktor Jònàs, and Róbert Paulik. Comparison of polygon and voxel-based visualization of miroscopic serial sections. In *Proceedings of the International Conference on Computational Cybernetics and Cyber-Medical Systems*, pages 000029–000034. IEEE, 2022.

[6] Kevin Lin, Lijuan Wang, and Zicheng Liu. Mesh graphormer. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 12939–12948, 2021.

[7] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.*, 42(4):139–1, 2023.

[8] David Bauer, Qi Wu, Hamid Gadirov, and Kwan-Liu Ma. Gscache: Real-time radiance caching for volume path tracing using 3d gaussian splatting. *arXiv preprint arXiv:2507.19718*, 2025.

[9] Jianxiong Shen, Yue Qian, and Xiaohang Zhan. Lod-gs: Achieving levels of detail using scalable gaussian soup. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pages 671–680, 2025.

[10] Arne Schilling, Jannes Bolling, and Claus Nagel. Using gltf for streaming citygml 3d city models. In *Proceedings of the 21st International Conference on Web3D Technology*, pages 109–116, 2016.

[11] Ziyang Song and Bo Yang. Ogc: Unsupervised 3d object segmentation from rigid dynamics of point clouds. *Advances in Neural Information Processing Systems*, 35:30798–30812, 2022.

[12] Yihang Chen, Qianyi Wu, Mengyao Li, Weiyao Lin, Mehrtash Harandi, and Jianfei Cai. Fast feedforward 3d gaussian splatting compression. In *The Thirteenth International Conference on Learning Representations*, 2025.

[13] Yihang Chen, Qianyi Wu, Weiyao Lin, Mehrtash Harandi, and Jianfei Cai. Hac: Hash-grid assisted context for 3d gaussian splatting compression. In *Computer Vision – ECCV 2024*. Springer, 2024.

[14] Khronos Group. glTF 2.0 specification. https://registry.khronos.org/glTF/specs/2.0/glTF-2.0.html, 2017. Accessed: 2025-10-07.

[15] Wouter Lemoine and Maarten Wijnants. Progressive network streaming of textured meshes in the binary gltf 2.0 format. In *Proceedings of the 28th International ACM Conference on 3D Web Technology*, pages 1–11, 2023.

[16] Patrick Cozzi and Sean Lilley. 3d tiles specification. OGC Community Standard 22-025r4, Open Geospatial Consortium (OGC), 2023.

[17] Inc. Cesium GS. Cesiumjs. https://github.com/CesiumGS/cesium, 2011–. Open-source JavaScript library for 3D globes and maps. Accessed: 2025-10-07.

[18] Anita Havele, Andreas Plesch, and Mike McCann. Conceptualizing interoperable 3d geospatial data visualization with x3d and ogc 3d tiles. In *Proceedings of the 29th International ACM Conference on 3D Web Technology*, pages 1–6, 2024.

[19] Shishir Subramanyam, Irene Viola, Alan Hanjalic, and Pablo Cesar. User centered adaptive streaming of dynamic point clouds with low complexity tiling. In *Proceedings of the 28th ACM international conference on multimedia*, pages 3669–3677, 2020.

[20] Thomas Forgione, Axel Carlier, Géraldine Morin, Wei Tsang Ooi, Vincent Charvillat, and Praveen Kumar Yadav. Dash for 3d networked virtual environment. In *Proceedings of the 26th ACM international conference on Multimedia*, pages 1910–1918, 2018.

[21] Jeroen Van Der Hooft, Tim Wauters, Filip De Turck, Christian Timmerer, and Hermann Hellwagner. Towards 6dof http adaptive streaming through point cloud compression. In *Proceedings of the 27th ACM International Conference on Multimedia*, pages 2405–2413, 2019.

[22] May Lim, Abdelhak Bentaleb, and Roger Zimmermann. Wide-vr: An open-source prototype for web-based vr through adaptive streaming of 6dof content and viewport prediction. In *Proceedings of the 16th ACM Multimedia Systems Conference*, pages 221–227, 2025.

[23] Jean-Philippe Farrugia, Luc Billaud, and Guillaume Lavoué. Adaptive streaming of 3d content for web-based virtual reality: an open-source prototype including several metrics and strategies. In *Proceedings of the 14th Conference on ACM Multimedia Systems*, pages 430–436, 2023.

[24] Hadi Heidarirad and Mea Wang. Vv-dash: A framework for volumetric video dash streaming. In *Proceedings of the 16th ACM Multimedia Systems Conference*, pages 256–262, 2025.

[25] Lovish Chopra, Sarthak Chakraborty, Abhijit Mondal, and Sandip Chakraborty. Parima: Viewport adaptive 360-degree video streaming. In *Proceedings of the Web Conference 2021*, WWW '21, pages 2379–2391, New York, NY, USA, April 2021. Association for Computing Machinery.

[26] Yangsheng Tian, Yi Zhong, Yi Han, and Fangyuan Chen. Viewport prediction with cross modal multiscale transformer for 360° video streaming. *Scientific Reports*, 15(1):30346, 2025.

[27] Filip Lemic, Jakob Struye, and Jeroen Famaey. Short-term trajectory prediction for full-immersive multiuser virtual reality with redirected walking, 2022.

[28] Xueshi Hou, Jianzhong Zhang, Madhukar Budagavi, and Sujit Dey. Head and body motion prediction to enable mobile vr experiences with low latency. In *Proceedings of the 2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7, Waikoloa, HI, USA, December 2019.

[29] Kevin Kwok. splat: Webgl 3d gaussian splat viewer. https://github.com/antimatter15/splat, 2023. Demo: https://antimatter15.com/splat/ Accessed: 2025-10-07.

[30] Michał Tyszkiewicz. Web viewer for gaussian splatting nerfs (webgpu/webgl). https://github.com/cvlab-epfl/gaussian-splatting-web, 2023. Project page: https://jatentaki.github.io/portfolio/gaussian-splatting/ Accessed: 2025-10-07.

[31] Simon Niedermayr, Josef Stumpfegger, and Rüdiger Westermann. Compressed 3d gaussian splatting for accelerated novel view synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10349–10358, 2024.

[32] Yuan-Chun Sun, Yuang Shi, Cheng-Tse Lee, Mufeng Zhu, Wei Tsang Ooi, Yao Liu, Chun-Ying Huang, and Cheng-Hsin Hsu. Lts: A dash streaming system for dynamic multi-layer 3d gaussian splatting scenes. In *Proceedings of the 16th ACM Multimedia Systems Conference (MMSys '25)*, pages 136–147, Stellenbosch, South Africa, 2025. Association for Computing Machinery.

[33] Yuang Shi, Géraldine Morin, Simone Gasparini, and Wei Tsang Ooi. Lapisgs: Layered progressive 3d gaussian splatting for adaptive streaming. *arXiv preprint*, 2024. Project page: https://yuang-ian.github.io/lapisgs/.

[34] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org, 2017.

[35] Jim Nilsson and Tomas Akenine-Möller. Understanding ssim. *arXiv preprint arXiv:2006.13846*, 2020.

[36] Colin Lea, Rene Vidal, Austin Reiter, and Gregory D Hager. Temporal convolutional networks: A unified approach to action segmentation. In *European conference on computer vision*, pages 47–54. Springer, 2016.

[37] Vickie Ye, Ruilong Li, Justin Kerr, Matias Turkulainen, Brent Yi, Zhuoyang Pan, Otto Seiskari, Jianbo Ye, Jeffrey Hu, Matthew Tancik, and Angjoo Kanazawa. gsplat: An open-source library for gaussian splatting. *Journal of Machine Learning Research*, 26(34):1–17, 2025.

[38] PlayCanvas Ltd. Supersplat - 3d gaussian splat editor, 2024.

[39] Polycam Inc. Polycam: Cross-platform 3d scanning, floor plans & drone mapping. https://poly.cam/, 2025.

[40] Stefano Pellegrini, Andreas Ess, Konrad Schindler, and Luc Van Gool. You'll never walk alone: Modeling social behavior for multi-target tracking. In *Proceedings of the International Conference on Computer Vision*, pages 261–268. IEEE, 2009.

[41] Alexandre Alahi, Kratarth Goel, Vignesh Ramanathan, Alexandre Robicquet, Li Fei-Fei, and Silvio Savarese. Social lstm: Human trajectory prediction in crowded spaces. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 961–971, 2016.

[42] Tim Salzmann, Boris Ivanovic, Punarjay Chakravarty, and Marco Pavone. Trajectron++: Dynamically-feasible trajectory forecasting with heterogeneous data. In *European Conference on Computer Vision*, pages 683–700. Springer, 2020.

[43] Abduallah Mohamed, Kun Qian, Mohamed Elhoseiny, and Christian Claudel. Social-stgcnn: A social spatio-temporal graph convolutional neural network for human trajectory prediction. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 14424–14432, 2020.

A Preliminary: 3D Gaussian Splatting

A.1 Representation

Starting from a set of points, each point is designated as the position (mean) μ of a 3D Gaussian.

$$G(x) = e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)} \tag{20}$$

where $\mu \in \mathbb{R}^3$ is its center and $\Sigma \in \mathbb{R}^{3\times3}$ denotes covariance matrix, which can be decomposed into a scaling factor S and a rotation quaternion R.

$$\Sigma = RSS^TR^T \tag{21}$$

Each Gaussian is characterized by attributes including position $X \in \mathbb{R}^3$, color $C \in \mathbb{R}^{(k+1)^2 \times 3}$ (where k represents the degrees of freedom), opacity $\alpha \in \mathbb{R}$, rotation factor $R \in \mathbb{R}^4$, and scaling factor $S \in \mathbb{R}^3$. The final rendered color $C(x')$ at a pixel position x' is determined using a tile-based rasterizer that employs alpha-blending with sorted 2D Gaussians:

$$C(x') = \sum_{i \in N} c_i \sigma_i \left(\prod_{j=1}^{i-1} (1 - \sigma_j) \right) \tag{22}$$

where N denotes the number of sorted 2D Gaussians associated with the queried pixel, $\sigma_i = \alpha_i G'_i(x')$, and the terms are sorted in descending order of opacity.

A.2 Challenge in 3DGS Downsampling

The homogeneous nature of 2D images and 3D point clouds, characterized by uniform distributions of pixels and points respectively, allows them to maintain high visual quality even after significant downsampling (Fig. 11, left and middle). In contrast, 3DGS suffers a substantial decline in visual quality, as depicted in right panel of Fig. 11, primarily due to the nonuniform distribution of ellipsoids and its heterogeneous attributes.

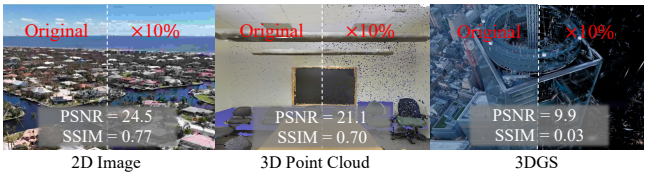

Figure 11: Side-by-side comparison of ground truth versus 10% downsampled representations

B Detailed Quantization Parameters

Table 6 shows the complete quantization bit allocation across the four quality levels. The specific bit depths were determined empirically by rendering thousands of test views and measuring SSIM versus file size tradeoffs.

Table 6: Quantization bit allocation across quality levels.

Attribute	Best	High	Medium	Low
Position (per coord)	24 bits	24 bits	24 bits	24 bits
Rotation (per comp)	12 bits	8 bits	6 bits	4 bits
Scale (per axis)	12 bits	8 bits	6 bits	4 bits
SH DC (per channel)	10 bits	8 bits	6 bits	4 bits
SH Higher (per coef)	8 bits	6 bits	5 bits	4 bits
Opacity	12 bits	8 bits	6 bits	4 bits
Bytes/primitive	**69.75**	**53.75**	**45.375**	**37**

Fig. 12 shows how quantization on attribute affects Gaussians reconstruction quality. Coordinate distortions significantly reduce visual quality, in contrast to other attributes' lesser impact when quantized at 5 bits or higher.

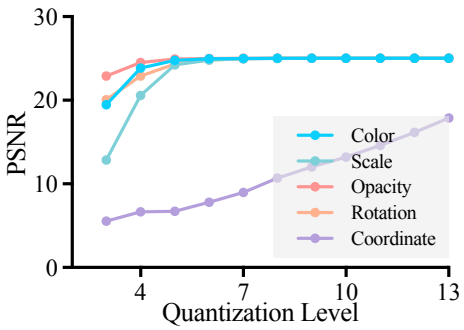

Figure 12: Attribute-Specific Quantization Impact.

C Camera Trajectory Prediction Model

C.1 Data Collection and Processing

Virtual navigation interface: Users view 3DGS scenes in the Web-based viewer. The frontend is implemented in JavaScript. The camera uses first person controls. WASD controls horizontal movement. Q and E control vertical movement. The mouse controls yaw and pitch.

Data recording: During navigation the client records state at 12 Hz. All data are collected in the browser. We collected trajectories from 20 users in three city scale scenes: shore, park, and industrial area. Each user explored 1 to 2 scenes. Each scene lasted 15 to 30 minutes. The dataset contains 1,847 segments. Segmentation occurs at natural pauses longer than 10 seconds. The total duration is 3.6 hours. The average segment length is 35 seconds. Fig. 13 shows a group of collected trajectory samples.

Data Processing: We apply a Kalman filter to smooth jitter from discrete key presses. We discard segments shorter than 10 seconds or longer than 5 minutes. The processed set retains 1,247 segments for training. We split data by scene rather than by user. For each scene the training set is 70 percent. The validation set is 15 percent. The test set is 15 percent. For evaluation we replay each test trajectory and compute visible tiles from the ground truth camera state. This yields the set T_{actual} for coverage metrics. A tile is visible if its oriented bounding box intersects the viewing frustum and its projection area in screen space exceeds 4 pixels. Fig. 14

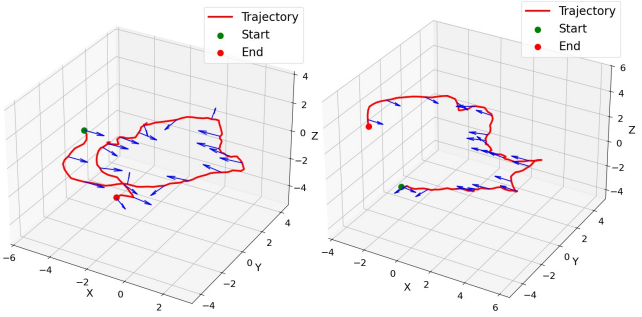

Figure 13: Collected Trajectory Samples: The blue arrow refers to camera yaw direction.

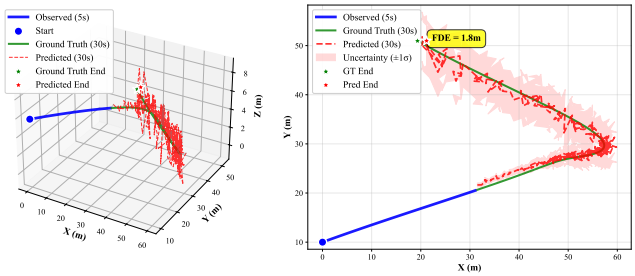

Figure 14: Qualitative Trajectory Prediction: The leftmost figure is a 3D view of a trajectory prediction example, and the rightmost one is a top-down view with uncertainty.

shows the predicted trajectory. The left panel presents a 3D view of one example. The right panel presents a top down view with uncertainty. Shaded regions indicate predictive variance. We use the mean prediction for tile mapping, and we compute all metrics with the ground truth T_{actual}.

C.2 Network Architecture Details

The complete TCN architecture consists of four main components: input normalization, temporal encoder, LSTM decoder, and probabilistic output heads.

Input normalization processes camera position and yaw. Each dimension is normalized using running statistics from the training set. For positions we use μ_{pos} and σ_{pos} to denote the mean and standard deviation of positions. For yaw we use μ_θ and σ_θ. Because angles are circular, we compute statistics on the unit circle with $(cos\theta, sin\theta)$ and then reconstruct the normalized angle.

The temporal encoder has four residual blocks. Each block has 128 channels. Each block contains two dilated causal convolutions. The kernel size is 3. The dilation rates for the four blocks are $\{1, 2, 4, 8\}$. The activation is ReLU. All convolutional layers use weight normalization. We apply dropout of 0.15 after each convolution.

The dilated causal convolution ensures that the prediction at time t only depends on past observations $\{\mathbf{s}(t'), t' \leq t\}$, maintaining temporal causality. The causal padding is achieved by padding the left side of the sequence by $(k-1) \times d$ where k is kernel size and d is dilation, then removing $(k-1) \times d$ elements from the right side after convolution (implemented via the Chomp1d operation).

The LSTM decoder takes the representation from the last TCN layer at time t. It has 2 LSTM layers. Each layer has 128 hidden units. The dropout is 0.1. The LSTM is unrolled for N_T steps to generate the future trajectory. At each step k, the LSTM hidden state is passed through two separate linear layers to produce mean $\boldsymbol{\mu}_k$ and log-variance $\log \boldsymbol{\sigma}_k^2$ predictions for 4D state (x, y, z, yaw).

The probabilistic head outputs Gaussian parameters at each time step. The standard deviation is parameterized as $\boldsymbol{\sigma}_k = \exp(\text{FC}_\sigma(\mathbf{h}_k))$ where FC_σ is a linear layer, ensuring $\boldsymbol{\sigma}_k > 0$. During inference, we use the mean prediction deterministically: $\hat{\mathbf{s}}(t + k\Delta t) = \boldsymbol{\mu}_k$.

C.3 Training Procedure

Regularization: We add an L2 regularization term with weight $\lambda = 10^{-5}$ in Eq. 10 to prevent overfitting.

Optimization: We use AdamW optimizer with base learning rate $\eta_{\text{base}} = 10^{-4}$. The OneCycleLR scheduler increases the learning rate to $\eta_{\max} = 10^{-3}$ over the first 30% of training, then decreases it following a cosine annealing schedule. The maximum learning rate is 10× the base rate for stable convergence.

Gradient clipping: We clip gradient norms to 1.0 to prevent exploding gradients common in temporal models.

Batch size and training time: We use batch size 32 with sequences processed in parallel. Training on a single NVIDIA A6000 GPU with 1,000 training sequences takes approximately 1.4 hours for 100 epochs. Early stopping with patience 15 epochs is employed based on validation loss.

C.4 Model Configurations and Ablations

Table B.1 shows performance across different model configurations:

Configuration	Layers	Channels	ADE (m)	FDE (m)	Coverage (%)
Small (real-time)	3	64	1.8	3.9	82
Medium	4	96	1.4	3.2	85
Large (default)	4	128	1.2	2.8	87
XLarge	5	128	1.1	2.5	88

The Large configuration provides the best accuracy-efficiency tradeoff, with inference latency 18ms on CPU (sufficient for real-time operation with 12 Hz input rate). The XLarge configuration provides only marginal improvement at 2× computational cost.

Ablation studies: Removing the LSTM decoder and directly outputting predictions from the TCN encoder increases ADE to 1.7m and FDE to 3.8m, demonstrating the importance of the autoregressive decoder for multi-step prediction. Using a standard RNN (GRU with 128 units) instead of TCN increases ADE to 2.3m and training time by 3×, confirming TCN's superiority for this task. Removing probabilistic outputs (outputting only means) maintains similar ADE/FDE but degrades coverage to 82% as the optimization algorithm cannot account for prediction uncertainty when prioritizing tiles.

Input feature ablation: We also tested including explicit velocity as additional input features (8D state instead of 4D). This provided no improvement in prediction accuracy (ADE 1.2m, same as 4D) while increasing inference time by 15%, confirming that velocity information is already implicitly captured by the temporal sequence of positions.

C.5 Real-time Inference Pipeline

During deployment, the prediction model runs in a separate Web Worker thread from the rendering loop to avoid blocking frame generation. The pipeline operates as follows:

1. Input buffering: Maintain a circular buffer of the last 60 camera states (5 seconds at 12 Hz) in shared memory 2. Prediction trigger: Trigger prediction every 2-3 seconds or when camera velocity changes by >20% 3. Forward pass: Run TCN encoder-decoder to generate 30-second prediction (18ms on CPU, 6ms on GPU) 4. Tile mapping: For each predicted camera state, compute visible tiles via frustum-OBB intersection 5. Deadline computation: Compute $t_{\text{first}}(i)$ and t_i for each predicted tile 6. Optimization handoff:

Pass predicted tiles, deadlines, and priorities to quality selection optimizer

The optimizer (Section 3.5) runs synchronously with prediction updates, solving the dynamic programming problem in <100ms. Together, the prediction and optimization pipeline adds <120ms overhead, which is imperceptible given that predictions are valid for 2-3 seconds before re-computation.

Handling prediction failures: If the prediction model has not yet accumulated sufficient history (first 5 seconds after page load), we fall back to a simple heuristic: fetch all tiles within 100m radius at Medium quality. Once sufficient history is available, the learned predictor takes over. Similarly, if prediction confidence is very low (high $\boldsymbol{\sigma}_k$ values), we blend predicted tiles with the radius-based heuristic to ensure baseline coverage.

D WebGL Rendering Pipeline Details

D.1 Texture Layout

For a tile containing n primitives, we allocate textures with width $W_{\text{tex}} = 2048$ as follows: The position texture T_{pos} uses RGBA32F format with dimensions $2048 \times \lceil n/2048 \rceil$. Each texel stores $(x, y, z, 1)$ in world coordinates.

The covariance texture T_{cov} uses RGBA16F format with dimensions $2048 \times \lceil 2n/2048 \rceil$. Each Gaussian requires two texels to store the six unique elements of the symmetric 3×3 covariance matrix: first texel stores $(\Sigma_{11}, \Sigma_{12}, \Sigma_{13}, \Sigma_{22})$ and second texel stores $(\Sigma_{23}, \Sigma_{33}, 0, 0)$. The color texture T_{col} uses RGBA8UI format with dimensions $2048 \times \lceil n_{\text{SH}} n/2048 \rceil$ where n_{SH} is the number of SH coefficients per primitive. Each texel packs four coefficient values. For SH degree 3 (48 coefficients), each primitive requires 12 texels.

D.2 Vertex Shader

The vertex shader receives only a `splatIndex` attribute and a corner index (0-3 for quad corners). It performs the following steps: 1. Fetch position $\boldsymbol{\mu}$ from T_{pos} 2. Fetch covariance $\boldsymbol{\Sigma}$ from T_{cov} (reconstructing from 6 values) 3. Compute view-space position: $\boldsymbol{\mu}_{\text{view}} = \mathbf{V}\boldsymbol{\mu}$ where $\mathbf{V}$ is view matrix 4. Compute projection Jacobian: $\mathbf{J} = \frac{\partial \pi}{\partial \mathbf{x}}|_{\boldsymbol{\mu}_{\text{view}}}$ 5. Project covariance: $\boldsymbol{\Sigma}' = \mathbf{J}\boldsymbol{\Sigma}\mathbf{J}^T$ 6. Compute eigenvalues λ_1, λ_2 and eigenvectors of $\boldsymbol{\Sigma}'$ 7. Generate quad corner position: $\mathbf{p}_{\text{corner}} = \boldsymbol{\mu}' + 3\sqrt{\lambda_1}\mathbf{e}_1 \cdot c_x + 3\sqrt{\lambda_2}\mathbf{e}_2 \cdot c_y$, where $(c_x, c_y) \in \{(-1,-1), (1,-1), (1,1), (-1,1)\}$ are corner offsets and $\mathbf{e}_1, \mathbf{e}_2$ are eigenvectors. The factor of 3 ensures 3σ coverage.

D.3 Fragment Shader

Following the way how 3DGS is rasterized in Appendix A, the fragment shader receives the screen-space position $\mathbf{x}$ and evaluates:

$$\alpha_{\text{final}} = \alpha \cdot \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}')^T(\boldsymbol{\Sigma}')^{-1}(\mathbf{x} - \boldsymbol{\mu}')\right) \tag{23}$$

where α is the primitive opacity fetched from texture. The color is computed by evaluating spherical harmonics with view direction as input. For efficiency, we precompute SH basis functions in a lookup texture indexed by viewing angle.

D.4 Depth Sorting

We maintain a shared array buffer B_{depth} containing depth keys and a corresponding index buffer B_{index}. When sorting is triggered, we transfer these buffers to a Web Worker that performs radix sort with 8-bit passes (4 passes total for 32-bit keys). The worker uses the following algorithm: 1. Create 256 histogram bins for the first 8 bits 2. Compute prefix sum to get output positions 3. Rearrange elements based on prefix positions 4. Repeat for next 8 bits, alternating between two buffers

D.5 Frustum Culling

We represent the view frustum as six planes $\{\mathbf{n}_p, d_p\}_{p=1}^{6}$ where $\mathbf{n}_p$ is the normal and d_p is the distance. For each tile's OBB, we compute the eight corner vertices $\{\mathbf{v}_k\}_{k=1}^{8}$ in world space. The frustum-OBB intersection test marks a tile visible if any OBB corner is on the positive side of all frustum planes. For hierarchical culling, we traverse the tile tree depth-first and skip entire subtrees when parents are culled.