SGSS: Streaming 6-DoF Navigation of Gaussian Splat Scenes

Mufeng Zhu
Rutgers University
Piscataway, NJ, USA
mz526@rutgers.edu

Mingju Liu
University of Maryland, College Park
College Park, MD, USA
mliu9867@umd.edu

Cunxi Yu
University of Maryland, College Park
College Park, MD, USA
cunxiyu@umd.edu

Cheng-Hsin Hsu
National Tsing Hua University
Hsin-Chu, Taiwan
chsu@cs.nthu.edu.tw

Yao Liu
Rutgers University
Piscataway, NJ, USA
yao.liu@rutgers.edu

Abstract

3D Gaussian Splatting (3DGS) is an emerging approach for training and representing real-world 3D scenes. Due to its photorealistic novel view synthesis and fast rendering speed (e.g., over 100 FPS), it has the potential to transform how scenes that can be explored in 6 degrees-of-freedom (6-DoF) are represented. However, a limiting factor of 3DGS is its large size, which requires high network bandwidth for streaming reconstructed real-world 3D scenes.

In this paper, we propose *SGSS* for optimizing the streaming transmission of 3DGS scenes during 6-DoF navigation. Since not all Gaussians in the full scene are needed for rendering a user's view, *SGSS* uses view-adaptive streaming, enabled by optimized spatial partitioning of the scene, for achieving network transmission savings. For each spatial partition, *SGSS* uses an importance-based Gaussians pre-sorting scheme to enhance the initial view quality and reduce the user-perceived scene loading time. We further design a client-side view-adaptive streaming algorithm that features lightweight visibility checking, prioritized streaming, incremental processing, and stream pausing/resuming schemes. We implement *SGSS* with JavaScript and WebGL2. Evaluation results show that the quality of views rendered with *SGSS* streaming is consistently higher than or on par with state-of-the-art approaches. Furthermore, the view-adaptive streaming of *SGSS* can result in high savings in network transmission without impacting the view quality.

CCS Concepts

• **Information systems** → **Multimedia streaming**; • **Computing methodologies** → **Point-based models**; • **Human-centered computing** → **Mixed / augmented reality** .

Keywords

3D Gaussian Splatting, View-Adaptive Streaming, Visual Quality

ACM Reference Format:
Mufeng Zhu, Mingju Liu, Cunxi Yu, Cheng-Hsin Hsu, and Yao Liu. 2025. *SGSS*: Streaming 6-DoF Navigation of Gaussian Splat Scenes. In *ACM Multimedia Systems Conference 2025 (MMSys '25), March 31-April 4, 2025, Stellenbosch, South Africa.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3712676.3714437

MMSys '25, Stellenbosch, South Africa

ACM ISBN 979-8-4007-1467-2/25/03
https://doi.org/10.1145/3712676.3714437

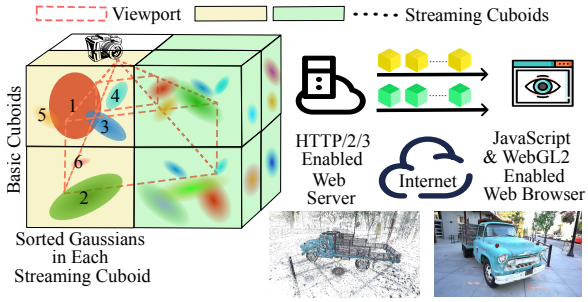


Figure 1: Illustration of our proposed SGSS streaming system.

1 Introduction

In recent years, both immersive applications and VR/AR devices such as Apple Vision Pro have gained increased popularity among users. This has resulted in an increasing demand for recording, reconstruction, and streaming of real-world 3D scenes for providing immersive experiences that can be navigated in 6 degrees-of-freedom (6-DoF) to users. Traditional representations of 3D scenes include point clouds and triangular meshes. To capture real-world 3D scenes, raw point clouds can be obtained by fusing data recorded sets of cameras and/or other sensors such as LiDAR sensors [1, 2]. Due to noises inherent in data produced by the sensors [3] and the size of rendered points, point clouds can suffer from inaccurate geometry and "holes" in rendered views [4]. Reconstructing accurate triangular meshes from point clouds is also challenging [5].

Recent advances in 3D reconstruction and novel-view synthesis have seen the emergence of learned representations such as NeRF [6] and 3D Gaussian Splatting (3DGS) [7], and much effort has been devoted to follow-up works of these representations, e.g., [8–11]. Both NeRF and 3DGS represent 3D scenes via parameters trained via gradient descent, given input images of the scene recorded from different perspectives. While NeRF and its variants can achieve photorealistic novel view synthesis, their main issue is with the high computation overhead associated with volumetric view rendering. As a result, it is presently infeasible to achieve high-quality view rendering with high frames-per-second (FPS). Compared to NeRF, 3DGS can achieve over 100 FPS rendering speed, an order of magnitude faster than NeRF-like approaches [12].

3DGS is a point-based representation. It uses anisotropic 3D Gaussians to represent the geometry of the 3D scenes. Unlike point clouds that may suffer from "holes" in rendered views, each 3D Gaussian is projected to a 2D splat (e.g., a circular/elliptical disc) during rendering, which can occupy a larger extent than a single

point's projection. 3DGS has been shown to provide photorealistic view synthesis with real-time rendering speed. Its main drawback, however, is the large representation size. Each 3D Gaussian occupies over 200 bytes of space for storing its attributes such as position, scale, rotation, and opacity. It also uses 48 spherical harmonics (SH) coefficients for representing view-dependent color. To produce a good representation of a scene, millions of Gaussians may be needed, resulting in significant storage requirements. For example, the trained 3DGS representation for the `garden` scene [11] contains over 5.8 million Gaussians and is 1.4 GBytes.

To address the significant representation size of 3DGS, approaches have been proposed to reduce the per-Gaussian size by only retaining the DC components of the SH coefficients while removing the rest (45) coefficients. This can yield 180 bytes savings out of 248 bytes (72%) of each Gaussian [13]. Even so, due to the large number of Gaussians in a scene, the total scene size is still large, e.g., hundreds of MBytes. Recent research also proposed to reduce the number of Gaussians without significantly impacting the view quality. For example, Fan et al. propose to prune Gaussians based on a global significance score calculated during training [10].

As an immersive media representation, 3DGS includes all the information needed for viewing the scene from any position in any orientation (thereby supporting 6-DoF navigation). However, at a time, only part of the scene is visible in the user's viewport. Downloading the full scene while only part of the data will be needed inevitably results in wasted data. Furthermore, after requesting the scene, the user expects to see a high-quality view of the scene as soon as possible. So Gaussians within the viewport should be transmitted first before others. To the best of our knowledge, however, no existing work has considered optimizing the transmission of Gaussians needed for rendering the user's view. Some solutions require the full 3DGS scene to be downloaded (a long time) before views are rendered to the users [14]. Others support progressive downloading and rendering of Gaussians, allowing the user to view the scene before it is fully downloaded [13, 15]. Nonetheless, both types of solutions end up downloading all Gaussians of the scene.

In this paper, we propose *SGSS* – **S**treaming 6-DoF navigation of **G**aussian **S**platt **S**cenes – for optimizing streaming of 3DGS scenes. Figure 1 provides a high-level overview of the streaming system.

Optimized Spatial Partitioning for View-Adaptive Streaming. *First*, since not all Gaussians are needed for rendering a view of the scene, we propose view-adaptive streaming of Gaussians, which can save network transmission. Due to the vast number (millions) of Gaussians within a 3DGS scene, it is infeasible to use individual Gaussians as the transmission scheduling unit. Instead, we propose to spatially partition the 3DGS scene into non-overlapping cuboids and use them to support view-adaptive streaming. To minimize the expected network downloading under a view distribution pattern while taking into account the overhead associated with using small cuboids, we formulate an optimization problem to find the best way to spatially partition the scene into cuboids.

Importance-Based Gaussians Pre-Sorting. *Second*, we propose a ranking scheme for pre-sorting Gaussians within each cuboid to enhance the initial view quality and reduce the user-perceived scene loading time. The need to reduce the user-perceived scene loading time is motivated by the "largest contentful paint" (LCP) metric for measuring how quickly the main content of a webpage is downloaded and becomes visible to the user [16]. Based on the observation that Gaussians with higher opacity and larger scale have a greater impact on the rendered view, our ranking scheme prioritizes the streaming and rendering of these important Gaussians, allowing meaningful views to be rendered faster.

***SGSS* Streaming Algorithm.** *Third*, we design a view-adaptive streaming algorithm for the client-side. It includes a lightweight cuboid visibility test for checking which cuboids are inside the viewport. Since all visible cuboids are streamed in a multiplexed connection in modern HTTP protocols, we design a practical scheme for assigning priority weights to streams, based on each visible stream cuboid's density of Gaussian ellipsoids, contribution to the viewport, and distance from the camera origin. The streaming algorithm also supports incremental response (i.e., Gaussians) downloading and processing, fully realizing the benefits of pre-sorting Gaussians inside cuboids for faster initial visual quality improvements.

We implemented *SGSS* in JavaScript with WebGL2 and conducted extensive experiments for evaluation. Results show that views rendered with *SGSS* streaming are consistently of visual quality higher or on par with state-of-the-art approaches. Furthermore, the view-adaptive transmission of *SGSS* can result in 32.9% savings on average in network transmission. In summary, this paper makes the following contributions:

- We propose optimized spatial partitioning of the 3DGS scene into cuboids to support the viewport-adaptive transmission.
- We further propose importance-based Gaussians pre-sorting within cuboids for enhancing the initial view quality.
- We design a viewport-adaptive streaming algorithm for 6-DoF navigation of 3DGS scenes and implement our proposed solution in JavaScript with WebGL2.
- Extensive experiment results show that *SGSS* can improve the visual quality during the initial streaming phase and save network bandwidth compared to state-of-the-art approaches.

2 Background and Related Works

Representation of each 3D Gaussian. A 3DGS scene is represented by a set of 3D Gaussians with their parameters optimized through a training procedure via backpropagation. Each 3D Gaussian is represented using information about its position (i.e., mean) $\boldsymbol{\mu} = (\mu_x, \mu_y, \mu_z) \in \mathbb{R}^3$, a covariance matrix $\Sigma \in \mathbb{R}^{3\times3}$, color $\mathbf{c}$, and opacity $o \in \mathbb{R}$. To facilitate optimization of parameters using gradient descent, the covariance matrix Σ is equivalently represented as the scale and rotation of an ellipsoid [7]. That is, $\Sigma = RSS^TR^T$, where $S \in \mathbb{R}^{3\times3}$ is a diagonal matrix, representing the scaling along each of the 3 dimensions; and $R \in \mathbb{R}^{3\times3}$ is a 3D rotation matrix. S is stored as $\mathbf{s} = (s_x, s_y, s_z) \in \mathbb{R}^3$, $S = \text{diag}(\mathbf{s})$; and the rotation matrix R is stored as quaternion $\mathbf{q} \in \mathbb{R}^4$. 3DGS further uses spherical harmonics (SH) for representing view-dependent color. For a single color channel, with k degrees of freedom, $(k+1)^2$ SH coefficients are needed. Given that 3 degrees of view direction freedom is required, a total of $16 \times 3 = 48$ SH coefficients are needed for R/G/B color channels. Overall, each 3D Gaussian needs to store 59 floating point numbers (236 Bytes) for representing the position $\boldsymbol{\mu} \in \mathbb{R}^3$, scale $\mathbf{s} \in \mathbb{R}^3$, rotation quaternion $\mathbf{q} \in \mathbb{R}^4$, opacity $o \in \mathbb{R}$, and view-dependent color $\mathbf{c} \in \mathbb{R}^{3\times(3+1)^2}$. 3D Gaussians are stored in a `.ply` file, with each line of the file containing attributes of a 3D Gaussian.

3D Gaussian "splatting". During rendering, 3D Gaussians are projected to 2D splats, such as circular or elliptical discs. This effectively avoids the appearance of "holes" in the rendered views, an issue with the traditional point-based representation like point clouds. The projection is done by $\Sigma' = JW\Sigma W^T J^T$, where $\Sigma' \in \mathbb{R}^{2\times 2}$ is the 2D covariance matrix, W is the view transformation matrix, and J is the affine transform approximation of the perspective projection transformation [7, 17, 18].

Pixel rendering. Given a pixel i, a list of overlapping Gaussians can be obtained, transformed to the camera space, and sorted based on their depth. With $\mathcal{N}$ sorted Gaussians, the color of pixel i can be obtained via alpha blending:

$$C_i = \sum_{n\leq \mathcal{N}} \Big(c_n \cdot \alpha_n \cdot \prod_{m<n} (1-\alpha_m) \Big),$$

where α_n is computed using the 2D covariance matrix Σ' and opacity parameter o_n [17].

Compression of 3DGS representations. Existing literature in point cloud compression has explored efficient data structures such as octrees (e.g., G-PCC [19, 20]) and kd-trees (e.g., Draco [21]) for compressing the geometry information (i.e., point positions). For compressing attributes (e.g., color) of point clouds, region-adaptive hierarchical transform (RAHT) [22], quantization, and entropy coding can be used. For dynamic scenes (i.e., point cloud video), video-based point cloud compression (V-PCC) leverages existing 2D video codec [19]. Since 3DGS is a point-based representation, it can also apply point cloud compression for compressing the geometry (i.e., positions of Gaussians) and other attributes, e.g., [23]. Since the emergence of 3DGS, several 3DGS compression schemes have been proposed. For example, Compact3DGS [24] proposes to i) reduce the number of Gaussians by identifying non-essential Gaussians with a masking strategy; and ii) compress the attributes by using a grid-based neural field and vector quantization. LightGaussian [10] proposes Gaussian pruning based on a global significance score obtained during training and an SH distillation approach for reducing the attribute size. Compressed3DGS [25] uses vector quantization [26, 27] for compression. It proposes sensitivity-aware vector clustering for creating a compact codebook. While compression is orthogonal to the tasks of our paper, we note that these compression solutions can be integrated with our proposed approaches.

Transporting 3DGS over the network. `gsplat` [14] is one of the earliest demo of 3DGS web viewers. However, it does not support streaming of 3DGS, and users must wait for the full representation to be downloaded to view the scene. `GaussianSplats3D` [15] is another 3DGS web viewer based on JavaScript and WebGL. It pre-sorts 3D Gaussians in the scene based on their distance from the world origin (i.e., $(0, 0, 0)$ in the world coordinate system). That is, pre-sorting based on the L2 norm of $\boldsymbol{\mu}$, $||\boldsymbol{\mu}||$. In this way, Gaussians are downloaded from the world origin going outward. However, this scheme works well only if the viewport is centered at the world origin. `splat` [13] also supports streaming, and it pre-sorts Gaussians based on a metric considering both the scale and opacity. This ensures that Gaussians with large volumes and high opacity are downloaded first during streaming. Despite the pre-sorting, these approaches download all Gaussians in the full scene regardless of the user's viewport, wasting network bandwidth. Unlike these approaches, in this paper, we propose spatial partitioning of the 3DGS scene to enable view-adaptive streaming, saving network bandwidth while maintaining or improving the quality of rendered views. ViVo [28] is a state-of-the-art point cloud streaming approach. It can save streaming bandwidth by adjusting the point density of each spatial cell based on its visibility in the viewport. However, it does not work well for 3DGS streaming due to the special characteristics of 3D Gaussians (Section 5.5).

3 Design of *SGSS*

We design *SGSS* to both enhance the visual quality during scene streaming and achieve network bandwidth savings. The core of *SGSS* is a view-adaptive streaming solution. Minimizing network bandwidth requires only Gaussians within the viewport to be downloaded. However, individually checking the visibility of millions of Gaussians and scheduling transmission is a daunting task. On the other hand, transmitting the full scene without spatial partitioning can result in the least visibility testing and scheduling overhead but can lead to a significant amount of unused data to be downloaded. To balance these two conflicting objectives, we propose **optimized spatial partitioning** of the 3DGS scene in Section 3.1.

Since 3DGS is a representation with parameters of Gaussians trained via a gradient descent approach, each Gaussian can contribute differently to the final view, depending on its scale and opacity, and Gaussians are not uniformly distributed in the scene. This is different from traditional point clouds. Based on these unique characteristics of 3D Gaussians, *SGSS* uses an **importance score** metric (Section 3.2) to measure the importance of each Gaussian within the streaming cuboid and pre-sorts the Gaussians accordingly. This allows the most important Gaussians to be streamed first. *SGSS* further implements an **efficient viewport-adaptive streaming algorithm** for the client-side (Section 3.3). It uses a lightweight scheme visibility checking of cuboids. It then prioritizes the streaming of cuboids that contribute more to the rendering result, based on a carefully-designed priority weight metric. In addition, *SGSS* leverages modern HTTP protocol capabilities and supports incremental cuboid downloading and processing, as well as pausing and resuming unfinished downloads for enhancing the initial view quality and quality improvement.

3.1 Optimized Spatial Partitioning of the Scene

We first generate an Axes Aligned Bounding Box (AABB) for the full 3DGS scene, based on the extent of the Gaussians' positions in the world coordinate system. We then partition it into **basic cuboids** with the same size. The total number of basic cuboids in a scene depends on the shape of the scene's AABB, ensuring that each basic cuboid approximates a cube. We define a **streaming cuboid** as a partition of the scene that can be scheduled for streaming transmission. It is constructed by one or more basic cuboids. Suppose that along the three dimensions (i.e., width, height, depth) of the AABB, we divide the scene into i, j, and k partitions, respectively. Then the total number of basic cuboids is $i \cdot j \cdot k$, and the total number of possible, valid, streaming cuboids can be obtained using binomial coefficients as:

$$num\_valid\_streaming\_cuboids = \binom{i+1}{2} \cdot \binom{j+1}{2} \cdot \binom{k+1}{2}. \quad (1)$$

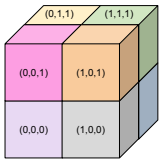


Figure 2: This figure shows a simple example where an Axes Aligned Bounding Box (AABB) of a full 3DGS scene is partitioned into 2x2x2 basic cuboids. With this partitioning, a total of 27 possible cuboids can be created (shown in Figure 3).

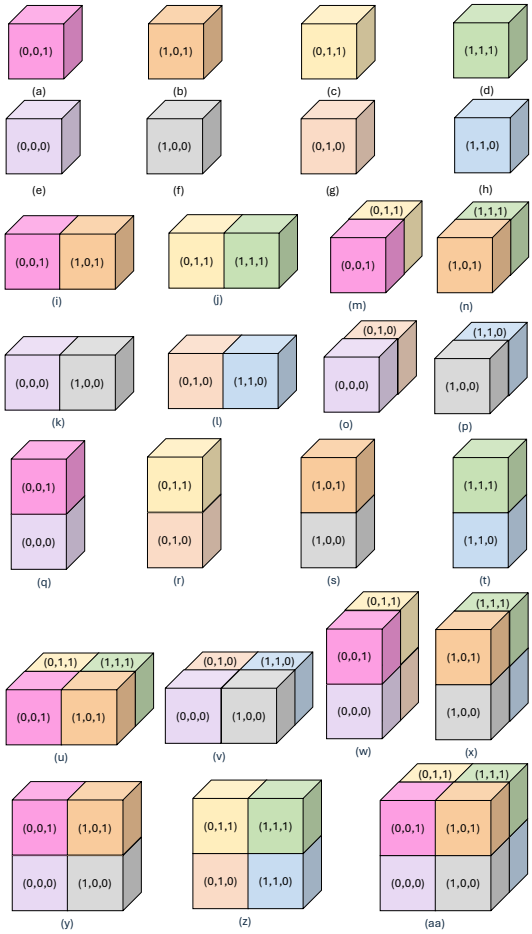


Figure 3: For an example scene with 2x2x2 basic cuboids, a total of 27 possible cuboids can be created. In general, given $i \times j \times k$ basic cuboids, the number of valid streaming cuboids can be obtained via Equation 1.

Figure 2 shows a simple example where a 3DGS scene is partitioned into 8 (2x2x2) basic cuboids. In this case, 27 possible streaming cuboids (labeled as `(a)`, `(b)`, ..., `(z)`, and `(aa)` in Figure 3) can be constructed from the 8 basic cuboids. To simplify the process of checking whether a cuboid is within the viewport (Section 3.3), we require that the constructed streaming cuboid be a regular cuboid. For example, a combination of basic cuboids $(0, 0, 0)$, $(1, 0, 0)$, $(0, 0, 1)$, and $(1, 0, 1)$ is valid. On the other hand, a combination of the following three basic cuboids $(0, 0, 0)$, $(1, 0, 0)$, and $(0, 0, 1)$ is invalid.

To formulate an optimization problem to determine the best spatial partitioning of the scene, we use a binary vector $\overrightarrow{x}$ to represent the solution of selected streaming cuboids. Given a 2x2x2 example shown in Figure 2, $\overrightarrow{x}$ would be a binary vector with a length of

Position	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	aa
(0,0,0)	0	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	1	1	0	1	0	1
(0,1,0)	0	0	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	1	0	0	0	1	1	0	0	1	1
(0,0,1)	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	1	0	0	0	1	0	1	0	1	0	1
(0,1,1)	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	1	0	1	0	0	0	1	1
(1,0,0)	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	1	0	0	1	0	1	0	1	1	1
(1,1,0)	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	1	0	1	0	1	0	1	0	1
(1,0,1)	0	1	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	1	0	0	1	1	0	1
(1,1,1)	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0	1	0	1	1	1

Figure 4: This table shows an example binary matrix A when a scene is partitioned into 2x2x2 basic cuboids. This matrix encodes how each of the 27 possible valid streaming cuboids includes different sets of these 8 basic cuboids.

27 that shows the presence of each streaming cuboid in the solution. For ease of presentation, we use the label of the basic cuboids along with a subscript to indicate the binary value. For example, in $[a_1,b_0,c_1,d_0,...,l_0,m_1,n_0...z_0,aa_0]$, a_1 represents streaming cuboid a is present in the solution, b_0 indicates streaming cuboid b is not present in the solution, and the solution only includes 3 streaming cuboids, i.e., a, c, and m.

A valid solution $\overrightarrow{x}$ should include streaming cuboids that cover all the basic cuboids while ensuring no basic cuboids are covered more than once. For example, $[...,o_1,p_1,...,u_1,...]$ is a valid solution including a streaming cuboid u constructed with 2x2 basic cuboids, two streaming cuboids constructed with 1x2 basic cuboids. On the other hand, $[...,m_1,...,o_1,p_1,u_1,...]$ is invalid, since basic cuboids $(0, 0, 1)$ and $(0, 1, 1)$ are covered by two different streaming cuboids (m and u). $[...,m_1,...,o_1,p_1,...]$ is also invalid, since basic cuboids $(1, 0, 1)$ and $(1, 1, 1)$ are not included in any of the three selected streaming cuboids (i.e., m, o, and p).

We also build a vector $\overrightarrow{C_{num}}$ to represent the number of Gaussians inside each streaming cuboid. It has the same length as binary vector $\overrightarrow{x}$.

Our optimization objective aims to minimize a cost function given a distribution of views. This distribution of views can be obtained from a dataset of prior users' views of the same scene. For each view v, a set of basic cuboids is needed to render the view. We thus use d_v, a binary vector, to represent all possible streaming cuboids that cover the basic cuboids required by view v. In the simple 2x2x2 basic cuboids example, given a viewport where two basic cuboids $(0, 0, 1)$ and $(1, 0, 1)$ are visible, then d_v is $[a_1, b_1, ..., i_1, ..., m_1, n_1, ..., q_1, s_1, ..., u_1, ..., w_1, x_1, y_1,..., aa_1]$. Given a valid solution $\overrightarrow{x}$, the total number of Gaussians inside streaming cuboids selected in $\overrightarrow{x}$ is:

$$(\overrightarrow{x})^\top \mathrm{diag}(d_v)\overrightarrow{C_{num}}. \tag{2}$$

However, the actual number of Gaussians observed by view v is typically smaller than the total in all the selected streaming cuboids. Thus, if the size of a streaming cuboid is too large, it will include Gaussians outside the viewport (i.e., invisible), resulting in downloading bandwidth waste. Therefore, we use C_v to represent the actual number of Gaussians visible to v, and calculate the ratio $\overrightarrow{R_{waste}}$ between the total number of Gaussians inside all selected streaming cuboids and the actual number of visible Gaussians:

$$R_{waste}^{(v)} = \frac{p_v \cdot \overrightarrow{x}^\top \mathrm{diag}(d_v)\overrightarrow{C_{num}}}{C_v}, \tag{3}$$

where p_v represents the viewing probability of v, and $\sum_{p_{v\in \mathbf{v}}} = 1$.

The smaller R_{waste}, the fewer wasted Gaussians. However, if we simply minimize R_{waste}, the optimal solution will be all the basic cuboids, meaning the largest number of streaming cuboids. With the increasing amount of streaming cuboids, the amount of time needed for checking if each streaming cuboid is within the user's viewport also increases, i.e., the "cuboid visibility test" against the view frustum. For a valid solution $\overrightarrow{x}$, the total times for visibility test C_d is calculated as:

$$C_d = \overrightarrow{C_{num'}}^{\top} \cdot \overrightarrow{x}, \tag{4}$$

where $\overrightarrow{C_{num'}}$ is a binary version of the original $\overrightarrow{C_{num}}$ by keeping all the zero elements and replacing all the non-zero elements with 1, which can filter the empty cuboids in the solution. The larger the total times for the visibility test, the longer time we need to wait before downloading the required cuboids. Therefore, we need to minimize the total visibility test times, while in the meantime, maintaining the smallest number of wasted Gaussians. Here we use a new vector $\overrightarrow{x_{base}}$ to represent a solution of $\overrightarrow{x}$ when all the basic cuboids are selected. Then the maximum visibility test times are:

$$C_{dmax} = \overrightarrow{C_{num'}}^{\top} \cdot \overrightarrow{x_{base}}. \tag{5}$$

Similar to R_{waste}, we introduce R_{vtest} to represent the ratio between the visibility test times of the current solution and the maximum visibility test time:

$$R_{vtest} = \frac{C_d}{C_{dmax}}. \tag{6}$$

Finally, we build a binary matrix A to encode the constraints of the optimization. Each row in A represents how a specific basic cuboid is included in the possible set of streaming cuboids, and each column encodes which basic cuboids are included in a specific streaming cuboid. For the simple 2x2x2 basic cuboids example, we show the content of matrix A (an 8x27 matrix) in Figure 4. For instance, the value at row $(1, 0, 0)$ and column s is 1, since streaming cuboid s includes basic cuboid $(1, 0, 0)$.

Our ILP formulation for this problem is defined as follows:

$$\text{minimize: } R_{vtest} + \sum_{v\in \mathbf{v}} R^{(v)}_{waste}$$
$$\text{subject to: } A\overrightarrow{x} = \overrightarrow{1}.$$

Here, $\overrightarrow{1}$ is a vector of 1's with the same size as the binary solution vector of streaming cuboids $\overrightarrow{x}$. The solution vector $\overrightarrow{x}$ determines a set of streaming cuboids for downloading that can lead to a minimized total number of wasted Gaussians and detection time of all the streaming cuboids.

3.2 Importance Sorting in Each Streaming Cuboid

In addition to spatial partitioning, *SGSS* also leverages features of modern HTTP protocols, i.e., HTTP/2 [29] and HTTP/3 [30], for improving scene loading quality for the users. These features include stream multiplexing where multiple requests/responses share the same transport-layer connection. When combined with responses that can be "incrementally" processed (e.g., chunks of a JPEG image or progressive JPEG), it can reduce the user-perceived webpage loading time. To fully take advantage of these features, *SGSS* presorts Gaussians in each streaming cuboid based on an **importance score**. That is, during scene streaming, requests/responses for multiple streaming cuboids within the viewport are multiplexed in the same connection. This ensures that each visible streaming cuboid contributes to the initial rendered view. The most important Gaussians in each streaming cuboid in the viewport are streamed and rendered first, enhancing both the initial viewport quality and the speed of quality improvement.

Importance score. Unlike traditional point cloud where points are typically rendered as spheres or cubes of the same size, in 3DGS, each Gaussian can have different scale and opacity attributes. The size of the Gaussian "splat" on the rendered view can vary based on its scale, and its contribution to the final pixel color can be different depending on its opacity. A Gaussian with a very small scale and low opacity (i.e., nearly transparent) will be barely visible compared to a Gaussian with a larger scale and full opacity. This means by streaming and processing Gaussians that have a greater impact on the rendering first, both the initial quality and the speed of quality improvement can be enhanced. This motivated us to pre-sort Gaussians within each streaming cuboid based on an importance score calculated based on the scale and opacity of a Gaussian. We define the **importance score** (`IS`) as

$$IS^{(i)} = o^{(i)} \cdot \left(\frac{s_x^{(i)} \cdot s_y^{(i)} \cdot s_z^{(i)}}{S_{\text{max\_}n}} \right)^{\beta}, \tag{7}$$

where $o^{(i)}$ represents the opacity of the i^{th} 3D Gaussian, and $s_x^{(i)}, s_y^{(i)}, s_z^{(i)}$ represent its scale. $S_{\text{max\_}n}$ represents the n^{th} percentile scale product value (i.e., $s_x \cdot s_y \cdot s_z$) of all 3D Gaussians in the scene. $n = 90$ and $\beta = 0.1$. This importance score was inspired by the global significance score proposed by Fan et al. [10]. We show in Section 5.2 that pre-sorting Gaussians based on the defined importance score can effectively improve view quality during the initial scene loading.

3.3 *SGSS* Streaming Algorithm

During streaming, the client first obtains information about all the streaming cuboids for the 3DGS scene that is available at the server in a `.json` file. *SGSS* supports view-adaptive streaming of 3DGS scenes by first performing cuboid visibility tests to determine a list of visible streaming cuboids for downloading. It designs a novel scheme for assigning priority to the streaming cuboids based on their Gaussian density, contribution to the viewport, and distance from the camera origin. It also fully supports incremental cuboid downloading and processing, allowing downloaded cuboids to be rendered promptly. A simplified version of the *SGSS* streaming algorithm is shown in Algorithm 1.

Cuboid Visibility Test. Viewport-aware streaming requires the streaming client to check if a streaming cuboid is inside the user's viewport, i.e., if the cuboid intersects with the view frustum. Using simple collision detection algorithms for AABB or OBB (Oriented Bounding Boxes) detection [31] is not accurate enough and can lead to false positives, resulting in unnecessary data downloading. Instead, we use the Separating Axis Theorem [32] to detect whether a streaming cuboid collides with the current user's view frustum. Since the view frustum's near plane is usually set to a very small

Algorithm 1 SGSS Streaming Algorithm

```
Initialize downloading manager
if the viewport has changed then
    ▷ Obtain a list of visible cuboids for the current view
    cuboid_list ← VISIBILITY_TEST(all_cuboids, viewport)
    ▷ Calculate priority weight
    for cuboid in cuboid_list do
        calculate priority weight P based on Equations 9 and 10
    ▷ Streaming of cuboids
    for cuboid in cuboid_list do
        if cuboid is not yet fully downloaded then
            range ← DOWNLOADING_MANAGER(cuboid)
            if cuboid is not being downloaded then
                request cuboid with priority and range
            else if priority ≠ current_priority then
                update request cuboid with new priority/range
    for cuboids that are downloading but no longer visible do
        pause the downloading
```

value – much smaller than the shape of a basic cuboid, we simplify the view frustum into a pyramid shape. For each streaming cuboid, we first convert all eight vertices' coordinates from world coordinates to camera coordinates, using the following equation

$$[x_c, y_c, z_c] = (R_c)^T * ([x_w, y_w, z_w] - [c_x, c_y, c_z]), \tag{8}$$

where $[x_w,y_w,z_w]$ represents a vertex's position in the world coordinate system, R_c is the camera's rotation matrix, and $[c_x,c_y,c_z]$ is the camera's position in the world coordinate system.

We then project the cuboid to three planes, i.e., xy, xz, and yz planes of the camera's coordinate system, respectively, resulting in three polygons. We also project the view pyramid to the same three planes, resulting in two triangles and one rectangle. To decide if a cuboid is inside the view pyramid, we check if two polygons intersect using point-in-polygon detection and edge intersection detection. If the projection of the cuboid intersects with the projection of the view pyramid on all three planes, the cuboid is visible.

Prioritized Streaming. *SGSS* uses modern HTTP protocols for downloading all visible streaming cuboids in HTTP/2/3 streams. With HTTP/2/3 stream multiplexing, the downloading of streaming cuboids share the same transport-layer connection. While this allows important Gaussians in all streaming cuboids within the viewport frustum to be downloaded and displayed, it can inevitably lead to inefficient use of network bandwidth during the initial starting phase of the streaming. For example, not all visible streaming cuboids contribute equally to the overall rendering quality. Some visible streaming cuboids may be either not completely visible in the viewport or far away from the viewport. Among these streaming cuboids, some with a high density of Gaussians may use a high percentage of available network bandwidth while contributing little to the increase of visual quality.

To address the above challenge, we define a priority weight for each streaming cuboid given a viewport. We use this priority weight with the Extensible Prioritization Scheme for HTTP [33]. This weight considers both the density of Gaussians inside and the cuboid's contribution to the rendered viewport. The priority weight PW of a streaming cuboid c given a viewport v is calculated as:

$$PW_v^{(c)} = \frac{\rho^{(c)} * V_v^{(c)}}{z_v^{(c)} + \varepsilon}. \tag{9}$$

Here, $\rho^{(c)}$ represents the density of Gaussian ellipsoids of inside streaming cuboid c, $V_v^{(c)}$ is the visibility factor, which is the calculated as the ratio of pixels in the rendered viewport v that are occupied by streaming cuboid c, and $z_v^{(c)}$ represents the z-distance between the center of streaming cuboid c and viewport v. We set a very small value ε to avoid the distance to be 0. Since PW can range from thousands to values smaller than 1, we apply a logarithmic function to compress the range to approximately -10 to 10. The final priority weight P for a streaming cuboid is

$$P_v^{(c)} = log_{10}(PW_v^{(c)}). \tag{10}$$

We then map the priority P to the priority values supported by the `Fetch Priority API` [34]. For example,

$$\texttt{fetchpriority} = \begin{cases} \text{high} & P \geq 1 \\ \text{low} & P < 1 \end{cases}$$

In addition, due to the high computational cost of pixel-level visibility factor calculations, we set the `fetchpriority` to *low* for cuboids with a density lower than 10, regardless of their visibility factor and distance.

Supporting Incremental Cuboids Downloading/Processing. Gaussians received in each HTTP/2/3 stream are incrementally parsed, loaded to the GPU memory, and rendered. These streams, however, may not be aware of each Gaussian's memory boundary during their transport over the connection. That is, while information on each Gaussian is saved in a certain order, e.g., position, scale, color, and rotation (as adopted by *SGSS*), the actual data delivery may break the per-Gaussian boundary, leading to incomplete Gaussian information being downloaded and incrementally parsed. For example, a data chunk might contain only Y bytes of a Gaussian of X bytes, where $Y < X$. When a new data chunk for a different cuboid arrives, its position and attributes could be mistaken for the missing bytes from the previous Gaussian. This can result in misaligned information of Gaussians being sent to the WebGL vertex shader [35], rendering incorrect views. To address this issue, we design a **breakpoint handler** that keeps track of the exact byte offset where data for each cuboid was left incomplete. When a new data chunk for the same cuboid arrives, the handler checks how many bytes were missing from the last incomplete data chunk of this cuboid and fills in the missed value. It then calculates how many Gaussians are in the remaining bytes of the current data chunk and allocates memory for them. With the breakpoint handler, *SGSS* fully supports incremental cuboid downloading and processing while ensuring correct view rendering.

Stream Pause and Resumption. When the viewport changes, the visible streaming cuboids and the visibility factor of each streaming cuboid may change as well. Therefore, we use a downloading manager to record how many bytes have been downloaded for each streaming cuboid. For the cuboids to become invisible, we abort the current HTTP stream, ceasing transmission of the invisible cuboid. If the cuboid later becomes visible again, we will send a new HTTP range request with its `Range` header field set to bytes

recorded by the downloading manager to resume the downloading. For streaming cuboids with priority changes, we will first abort the current downloading and then send a new request with updated priority and `Range` field in the header.

4 Implementation

We implemented *SGSS* with JavaScript and WebGL2 within a fork of the `splat` [13] repository. In addition to the sorting worker designed in `splat`, we implemented two additional web workers to support our viewport-adaptive streaming[1]. A **visibility and priority** web worker is used for testing the visibility of streaming cuboids and calculating the priority of each visible streaming cuboid. It sends the information about visible streaming cuboids to the **streaming** web worker, which sends requests to the server and receives the Gaussians in HTTP responses. We create a simple HTTP server using `H2O` [36], which supports HTTP/2/3 protocols.

Each pre-trained 3DGS scene is pre-processed offline by a Python script. Given an existing distribution of user views for a scene, it generates spatially-partitioned cuboids based on ILP results. For solving the ILP, we used Gurobi Optimization [37]. For each streaming cuboid, we pre-sort the Gaussians according to their importance score ranking. For each Gaussian, we store its information with its original float32 type in the following order: position (12 bytes), scale (12 bytes), SH coefficients, opacity (4 bytes), and rotation (16 bytes). We then store all the streaming cuboids in the HTTP server. We also store the spatial information of each streaming cuboid, including the maximum and minimum spatial coordinates and the density of Gaussian in a `.json` file, which will be shared with the client at the beginning of the streaming session. We have implemented WebGL vertex shaders that can support rendering Gaussians with spherical harmonics of varying degrees (SH0, SH1, SH2, and SH3). These SH coefficients are converted to RGB color inside the vertex shader. We only present SH0 results in the paper due to space.

5 Performance Evaluation

5.1 Setup

3DGS scenes. We use 12 pre-trained 3DGS scenes from the original 3DGS paper [7] in our experiments. We did not include the `counter` scene as this scene is the smallest among all, and the full scene can be downloaded in a very short amount of time. The 12 scenes are from the MipNeRF360 dataset [11], the Tanks&Temples dataset [38], and the Deep Blending dataset [39]. These datasets contain a variety of scene settings, including indoor, outdoor, and natural environments.

Baseline approaches. We use six baseline approaches in our experiments, summarized in Table 1. We first compare *SGSS* with three state-of-the-art third-party web-viewers implemented based on WebGL. **FD** refers to `gsplat` [14] that waits for full scene downloading before view rendering. **DWO** refers to `Gaussian-Splat3D` [15], which pre-sorts 3D Gaussians based on their distance to the origin $(0, 0, 0)$ of the world coordinate system. It supports 3DGS scene streaming, but visible content is expanded from the world origin as Gaussians are downloaded. **IM** refers to `splat` [13], which pre-sorts all 3D Gaussians based on their scale and opacity values. 3D Gaussians with large scale and opacity values are streamed first. None of the above baseline approaches are view-adaptive.

[1]The repository of this project is at https://github.com/symmru/SGSS.

Table 1: Baseline approaches used in the evaluation.

FD [14]	**F**ull scene **D**ownloading before rendering
DWO [15]	Rendering while streaming; Gaussians are pre-sorted based on **D**istance from **W**orld **O**rigin
IM [13]	Rendering while streaming; Gaussians are pre-sorted based on an **I**mportance **M**etric
ViVo [28]	Bandwidth-efficient point cloud streaming through visibility-based point density adjustment
Ours-`{w/o OSP}`	*SGSS* but without optimized spatial partitioning, i.e., with basic cuboids only
Ours-`{w/o IS}`	*SGSS* but without importance sorting

We also compare our method with state-of-the-art viewport-adaptive volumetric scene streaming method, **ViVo** [28]. To support bandwidth-efficient streaming, ViVo streams fixed-size cells (a concept very similar to basic cuboids in our design) and adjusts the point density of each cell based on its visibility. To evaluate the effectiveness of our optimized spatial partitioning (OSP), we design a baseline approach **Ours**-`{w/o OSP}` where only basic cuboids are used for scene streaming. We also compare with **Ours**-`{w/o IS}`, where Gaussians inside the streaming cuboids are not sorted.

Two sets of viewing traces. For our experiments, we generate two sets of user viewing traces derived from the camera poses of images in the raw dataset of the 12 3DGS scenes. The first set of viewing traces simulates an ***orbital shot***, emulating the user behavior over 80 seconds. During this period, the user navigates through 40 camera poses from the scene's training dataset, one every 2 seconds. Between two adjacent camera poses, we uniformly interpolate 119 frames, enabling smooth 60 FPS rendering. These 40 camera poses allow the user to explore the full scene. With this viewing trace, we aim to demonstrate how visual quality evolves over time as more Gaussians are downloaded and the user's viewport moves. We also design a ***dolly shot*** for evaluation of total downloaded data. This shot only navigates a portion of the scene. To do so, we use the first two training camera poses for the scene and uniformly interpolate 1,199 frames between them. The user's viewport moves slowly and smoothly from one camera pose to the other in 20 seconds.

For each experiment, we replay the generated viewing trace and record the rendering result in WebGL when the viewport is located at a training camera pose. We use the original training images as ground truth for calculating the visual quality of rendered views.

Evaluation metrics. For visual quality evaluation, we use three metrics: peak signal-to-noise ratio (**PSNR**), structural similarity (**SSIM**) [40], and perceptual video quality metric **VMAF** [41, 42] for comparing visual quality of views rendered to the user and ground truth camera-recorded images. We also record the **total downloaded data** amount (e.g., in MBytes) to show the effectiveness of our approach in reducing wasted network transmission.

Network bandwidth settings. To evaluate how our proposed *SGSS* approach and baseline approaches perform under different available network bandwidth, we choose three different bandwidth settings in our experiments: 100 Mbit/s, 150 Mbit/s, and 200 Mbit/s. We use the `tc` [43] tool to emulate different available bandwidth between the server and the 3DGS viewing client and set the maximum client-server network latency to 20ms round-trip. For ***orbital shot***

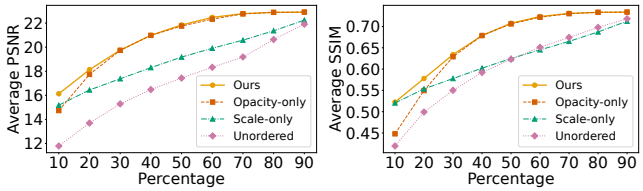


Figure 5: Comparison of different pre-sorting criteria.

traces, we evaluate the visual quality of all four methods under three bandwidth settings. For ***dolly shot traces***, focusing on the bandwidth savings, we only evaluate the performance under 200 Mbit/s. This ensures that all visible content in the scenes can be fully downloaded within 20 seconds of the viewing traces.

5.2 Pre-Sorting Criteria Results

To show the effectiveness of our importance sorting, we compare it with three other sorting criteria: `scale-only`, `opacity-only`, and `unordered`. `scale-only` ranks Gaussians solely based on their scales, `opacity-only` ranks Gaussians based on their opacity only, while `unordered` preserves the original order of Gaussians as they are trained. We evaluate 12 distinct 3DGS scenes. For each criterion, we pre-sort all Gaussians in the full `.ply` file according to the respective criterion. We then save the first 10%, 20%, ..., 90% of the sorted Gaussians and use them for rendering two different views in WebGL. These two views are also the same as the first two camera poses used to generate ***orbital shot traces***, which can be used to show the visual quality of initial views during streaming. We calculate the average visual quality results across scenes for each percentage, using the original camera images as the ground truth.

Figure 5 shows the average PSNR and SSIM values from 12 scenes. Our pre-sorting criteria demonstrate both higher initial visual quality and faster improvement in both metrics and significantly outperform non-sorting methods, demonstrating the necessity of this step. By incorporating both opacity and scale features in our sorting criteria, we achieve better results compared to approaches that consider these features independently. These findings indicate that the visual quality will increase the fastest with our pre-sorting criteria.

5.3 Spatial Partitioning Results

Table 2 shows the optimized spatial partitioning results from integer linear programming (ILP). For large scenes, we partition the 3DGS scenes into more basic cuboids. `Truck` is a scene with both small AABB size and file size, we therefore partition it into 10x10x3 basic cuboids in total. We are unable to find an existing dataset of users exploring these 12 3DGS scenes in 6-DoF. So we use the camera poses from training images of the original datasets as distribution of views, with each camera pose sharing the same view probability to set up the ILP. For each scene, the number of camera poses varies from 125 to 311. In Table 2, *Non-Empty* basic cuboids are cuboids with at least one 3D Gaussian. However, there exist many non-empty cuboids with only a few hundreds of Gaussians or even only one Gaussian. *Low-Density* represents the cuboids with a Gaussian density (as described in Equation 9) of less than 1. As discussed in Section 3.1, each of these cuboids takes the same time for the view frustum "visibility test". Our formulated optimization problem aims to reduce the number of selected streaming cuboids. Table 2 indicates that the ILP effectively works for our purpose. We can reduce the number of selected streaming cuboids to roughly 4% to 40% of non-empty basic cuboids.

Table 2: Spatial partitioning results.

Scene	# of Basic Cuboids			# of Solved/Selected	
	Total	Non-Empty	Low-Density	Streaming Cuboids	Low-Density
bicycle	2000	243	134	32	5
bonsai	1000	243	87	42	0
drjohnson	1000	285	79	112	3
flowers	1000	144	75	19	3
garden	2000	269	71	32	1
kitchen	2000	267	98	16	2
playroom	1000	431	94	139	2
room	1000	109	47	23	1
stump	2000	329	221	55	14
train	2000	237	177	9	4
treehill	2000	186	148	22	6
truck	300	108	52	16	2

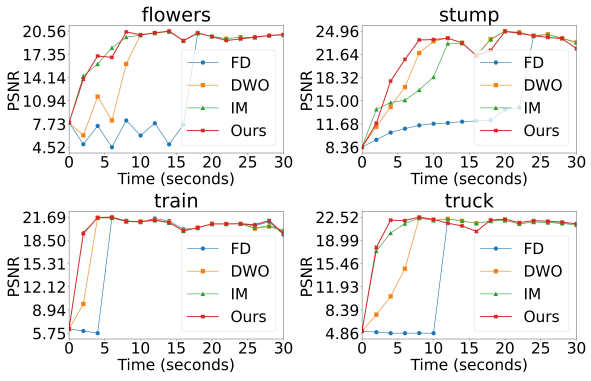
5.4 *SGSS* vs. Non-Viewport-Adaptive

We first compare *SGSS* with three state-of-the-art 3D Gaussian web-viewers, namely, **FD**, **DWO**, and **IM**.

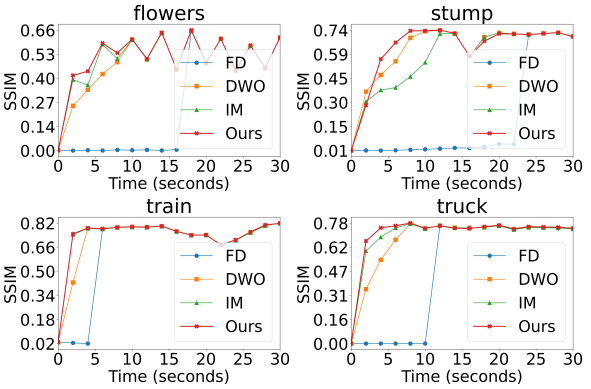
Visual quality results. None of the three baseline approaches support viewport-adaptive streaming. Even with our smallest bandwidth setting, they can finish full scene downloading within the first 30 seconds. We thus only show the results in the first 30 seconds. Benefiting from spatial partitioning, *SGSS* only downloads the streaming cuboids inside the viewport during the first 30 seconds. The total downloaded bytes are thus barely affected by the bandwidth change. Therefore, we present the comparison results under the smallest bandwidth setting, 100 Mbit/s.

Figure 6 shows the PSNR and SSIM results of three baseline approaches and *SGSS*. Compared to **FD** and **DWO**, our method achieves significantly better initial visual quality. **FD** takes a long time to download the scene. For large scenes like `stump` and `flowers`, it takes more than 20 seconds to achieve a reasonable visual quality. **DWO** performs better when the initial viewport looks at the scene's world origin. However, when the initial viewport is not looking at the world origin in the first 2 seconds, **DWO** leaves a huge blank in the viewport. Compared to **DWO**, our method downloads 3D Gaussians inside the user's viewport first, yielding better visual quality, shown in Figure 7. Besides, the density of Gaussians around world origin also affects the performance of **DWO**. In `truck`, the density of Gaussians close to the world origin is very high, therefore, it takes a long time for **DWO** to download these Gaussians, leaving the part close to viewport edges blank.

SGSS also yields better or similar initial visual quality results compared to **IM**. This is because both methods stream the 3D Gaussians with higher opacity and scale first. Different from our method, **IM** pre-sorts Gaussians over the full scenes, while we sort the Gaussians inside each streaming cuboid. If 3D Gaussians with higher importance metrics are concentrated in the background of the initial viewport, **IM** can potentially perform better than us in terms of the objective quality metrics in the first few seconds. This is because *SGSS* prioritizes streaming of cuboids closer to the viewport. However, if the Gaussians with higher importance are distributed uniformly around the scene, such as `stump`, the initial quality and



(a) PSNR (dB)↑ comparisons



(b) SSIM↑ comparisons

Figure 6: Visual quality comparison: *SGSS* vs. non-viewport-adaptive approaches under 100 Mbit/s and *orbital shot traces*.

(a) View rendered with the `DWO` approach. PSNR = 10.71 dB **(b) View rendered with our `SGSS` approach. PSNR = 30.66 dB**

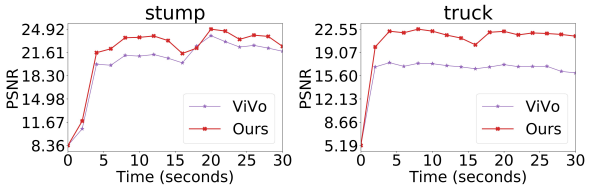
Figure 7: Side-by-side comparison: rendering result of `drjohnson` at the 6th second under 100 Mbit/s bandwidth.

the growth rate of visual quality will be severely affected. Compared to **IM**, our streaming method only focuses on the Gaussians inside the viewport, limiting the bandwidth wasted due to transmissions of Gaussians outside the viewport with higher significance scores.

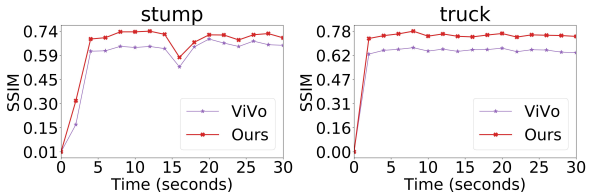
Bandwidth savings results. Table 3 shows the total bandwidth consumption in 20 seconds under 200 Mbit/s bandwidth based on the ***dolly shot traces***. With limited navigation of the viewport, only part of the scene can be viewed in 20 seconds. **FD**, **DWO**, and **IM** all download full scenes. However, our method only downloads the Gaussians in visible streaming cuboids. Notably, our method can save up to 71.47% bandwidth while streaming the `drjohnson` scene. Sometimes 3D Gaussians may be concentrated in one streaming cuboid, such as the table in the `garden` scene, and have a higher chance of being viewed, we have to download them completely.

Table 3: Total downloaded data under 200 Mbit/s and *dolly shot traces*.

Scene	DWO/IM/FD	*SGSS* (Ours)
bicycle	343.4 MB	**223.8 MB (-34.82%)**
bonsai	69.7 MB	**36.2 MB (-48.06%)**
drjohnson	190.7 MB	**54.4 MB (-71.47%)**
flowers	203.6 MB	**162.5 MB (-20.19%)**
garden	326.7 MB	**241.6 MB (-26.05%)**
kitchen	103.7 MB	**90.1 MB (-13.11%)**
playroom	142.6 MB	**59.2 MB (-58.49%)**
room	89.2 MB	**77.1 MB (-13.57%)**
stump	277.9 MB	**152.4 MB (-45.16%)**
train	57.5 MB	**53.3 MB (-7.30%)**
treehill	211.9 MB	**179.0 MB (-15.53%)**
truck	142.3 MB	**83.6 MB (-41.25%)**



(a) PSNR (dB)↑ comparison with *ViVo*



(b) SSIM↑ comparison with with *ViVo*

Figure 8: Visual quality comparison: *SGSS* vs. *ViVo* under 200 Mbit/s and *orbital shot traces*.

Therefore, the amount of saved bandwidth may vary. Nevertheless, on average, *SGSS* can achieve 32.9% network bandwidth savings.

5.5 *SGSS* vs. *ViVo*

Figure 8 shows that the visual quality of *SGSS* is significantly better than *ViVo*. This is because *ViVo* decreases the density of the cells occluded by others and far away from the view origin. However, due to the special geometry features of a 3DGS scene, Gaussians are not uniformly distributed across the cell. This results in the "fake occlusion" problem, where a cell is occluded by other cells while the Gaussians inside are not. As a result, due to density control for further-away cells used in *ViVo*, not all Gaussians required for view rendering may be downloaded. While this allows *ViVo* to use less network bandwidth compared to *SGSS*, the visual quality of *ViVo* suffers significantly. In contrast, benefiting from optimized spatial-partitioning, *SGSS* will download all the streaming cuboids which are visible in the viewport and decrease the wasted bandwidth. Figure 9 shows an example comparison. In the highlight of figures, due to the "fake occlusion" issue, required Gaussians in the background are not completely downloaded.

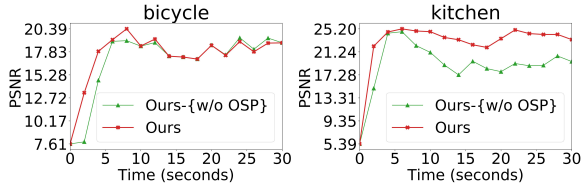
5.6 *SGSS* vs. *SGSS-w/o OSP* vs. *SGSS-w/o IS*

To show how *SGSS* can benefit from optimized spatial partitioning (OSP) and importance sorting (IS), we conduct experiments streaming basic cuboids only (i.e., *w/o OSP*) and streaming solved cuboids

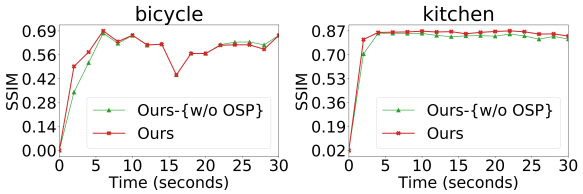
(a) View rendered with the ViVo approach. PSNR = 17.48 dB

(b) View rendered with our SGSS approach. PSNR = 22.21 dB

Figure 9: Side-by-side comparison: rendering result of garden at the 22th second under 200 Mbit/s bandwidth.

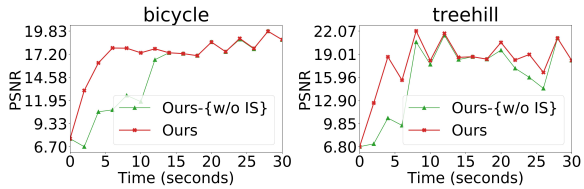


(a) PSNR (dB)↑ comparison with *SGSS-w/o OSP*

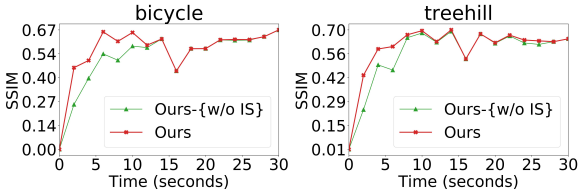


(b) SSIM↑ comparison with *SGSS-w/o OSP*

Figure 10: Visual quality comparison: *SGSS* vs. *SGSS-w/o OSP* under 200 Mbit/s and *orbital shot traces*.



(a) PSNR (dB)↑ comparison with *SGSS-w/o IS*



(b) SSIM↑ comparison with *SGSS-w/o IS*

Figure 11: Visual quality comparison: *SGSS* vs. *SGSS-w/o IS* under 100 Mbit/s and *orbital shot traces*.

without sorting Gaussians inside (i.e., *w/o IS*), both with their respective priority weights. The visual quality results are shown in Figures 10 and 11. For comparison with *SGSS-w/o OSP*, since basic cuboids with low density can benefit more from higher bandwidth, we show results under 200 Mbit/s bandwidth setting. Figure 10 shows that *SGSS* with OSP can achieve a more rapid increase at the beginning. Since OSP will merge some basic cuboids, more spatial cuboids will be assigned high priority, which promotes rapid growth in visual quality. Figure 11 shows that with importance sorting, *SGSS* allows the visual quality to improve faster than *SGSS-w/o IS*. This is

Table 4: VMAF results for all scenes across all baseline approaches under 100 Mbit/s bandwidth and *orbital shot traces*.

Scene	FD	DWO	IM	ViVo	Ours-{w/o IS}	Ours-{w/o OSP}	Ours
bicycle	27.46	84.14	88.19	53.29	91.47	92.08	**94.33**
bonsai	82.22	92.55	94.63	50.75	92.79	94.93	**96.04**
drjohnson	54.68	85.60	94.41	77.68	93.15	92.45	**94.63**
flowers	52.61	86.66	90.16	56.21	89.31	89.30	**91.86**
garden	24.58	80.86	86.87	60.16	87.58	83.63	**89.82**
kitchen	74.57	91.70	94.06	67.73	93.71	90.95	**95.34**
playroom	64.08	88.77	**95.15**	89.41	86.70	86.08	89.35
room	76.59	92.04	**95.20**	66.34	93.45	92.60	94.13
stump	39.61	90.13	77.79	75.09	90.91	90.85	**91.84**
train	83.91	92.72	94.62	79.74	94.78	93.83	**96.43**
treehill	47.60	81.72	91.76	72.18	86.52	91.77	**92.54**
truck	65.17	87.81	91.59	58.99	88.05	91.33	**94.47**

because Gaussians within a same streaming cuboid that contribute more to the rendering results are streamed and displayed earlier. We can also see that the visual qualities of the three methods are nearly the same after 10 seconds. This is because the camera in the ***orbital shot*** dataset moves slowly, causing the number of newly visible cuboids and the total Gaussians to be downloaded to remain similar, leading to almost the same visual quality.

5.7 Perceptual Video Quality

We report the perceptual visual quality results of all scenes under 100 Mbits/s bandwidth and ***orbital shot traces***, using the VMAF metric in Table 4. Since we do not have raw camera video footage of these scenes, we use a video rendered using all Gaussians (i.e., without streaming) in each scene as reference for computing the VMAF scores. *SGSS* consistently achieves high VMAF scores compared to baseline approaches. It performs the best in 10 out of 12 scenes, demonstrating the effectiveness of *SGSS*. As discussed earlier, when Gaussians with higher importance are concentrated in the initial viewport, **IM** produces results similar to those of *SGSS*. Moreover, since **IM** is a non-viewport-adaptive approach, it can achieve good visual quality performance once the entire scene is fully downloaded, as no additional Gaussians need to be downloaded for the later viewports. In contrast, *SGSS* may need to download new visible streaming cuboids. This has caused **IM** to perform slightly better than *SGSS* in VMAF results in two scenes.

6 Conclusion

3D Gaussian Splatting has the potential to transform real-world 3D scene representations. In this paper, we made a first effort to optimize the streaming transmission of 3D Gaussians to improve initial viewing quality while saving network bandwidth. We designed *SGSS* for view-adaptive streaming of optimized spatial partitions of the full 3DGS scene. Additionally, we implemented a pre-sorting scheme to quickly enhance initial visual quality. Furthermore, we developed an efficient streaming strategy that is compatible with modern HTTP protocols. Experiment results show that our approach is effective in achieving savings in network transmission without impacting the quality of views.

Acknowledgments

We appreciate constructive comments from anonymous referees. This work is partially supported by NSF under grants CNS-2200042 and CNS-2200048.

References

[1] Rafael Pagés, Konstantinos Amplianitis, David Monaghan, Jan Ondřej, and Aljosa Smolić. Affordable content creation for free-viewpoint video and vr/ar applications. *Journal of Visual Communication and Image Representation*, 53:192–201, 2018.

[2] Iro Armeni, Sasha Sax, Amir R Zamir, and Silvio Savarese. Joint 2d-3d-semantic data for indoor scene understanding. *arXiv preprint arXiv:1702.01105*, 2017.

[3] Branislav Jenco. Virtual lidar error models in point cloud compression. Master's thesis, 2022.

[4] Mufeng Zhu, Yuan-Chun Sun, Na Li, Jin Zhou, Songqing Chen, Cheng-Hsin Hsu, and Yao Liu. Dynamic 6-dof volumetric video generation: Software toolkit and dataset. In *2024 IEEE 26th International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–6. IEEE, 2024.

[5] Raphael Sulzer, Renaud Marlet, Bruno Vallet, and Loic Landrieu. A survey and benchmark of automatic surface reconstruction from point clouds. *arXiv preprint arXiv:2301.13656*, 2023.

[6] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[7] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4):1–14, 2023.

[8] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. Tensorf: Tensorial radiance fields. In *European Conference on Computer Vision*, pages 333–350. Springer, 2022.

[9] Sara Fridovich-Keil, Giacomo Meanti, Frederik Rahbæk Warburg, Benjamin Recht, and Angjoo Kanazawa. K-planes: Explicit radiance fields in space, time, and appearance. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12479–12488, 2023.

[10] Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, and Zhangyang Wang. Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps. *arXiv preprint arXiv:2311.17245*, 2023.

[11] Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5470–5479, 2022.

[12] Maximum Wilder-Smith, Vaishakh Patil, and Marco Hutter. Radiance fields for robotic teleoperation, 2024.

[13] Github: antimatter15/splat. https://github.com/antimatter15/splat.

[14] gsplat — 3D Gaussian Splatting WebGL viewer . https://gsplat.tech/.

[15] Github: mkkellogg/GaussianSplats3D. https://github.com/mkkellogg/GaussianSplats3D.

[16] Defining the Core Web Vitals metrics thresholds. https://web.dev/articles/defining-core-web-vitals-thresholds.

[17] Vickie Ye and Angjoo Kanazawa. Mathematical supplement for the `gsplat` library. *arXiv preprint arXiv:2312.02121*, 2023.

[18] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. Ewa splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):223–238, 2002.

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

[20] D Graziosi, O Nakagami, S Kuma, A Zaghetto, T Suzuki, and A Tabatabai. An overview of ongoing point cloud compression standardization activities: Video-based (v-pcc) and geometry-based (g-pcc). *APSIPA Transactions on Signal and Information Processing*, 9:e13, 2020.

[21] Draco 3D. https://google.github.io/draco/.

[22] Ricardo L De Queiroz and Philip A Chou. Compression of 3d point clouds using a region-adaptive hierarchical transform. *IEEE Transactions on Image Processing*, 25(8):3947–3956, 2016.

[23] Yuan-Chun Sun, Yuang Shi, Wei Tsang Ooi, Chun-Ying Huang, and Cheng-Hsin Hsu. Multi-frame bitrate allocation of dynamic 3d gaussian splatting streaming over dynamic networks. In *Proceedings of the 2024 SIGCOMM Workshop on Emerging Multimedia Systems*, pages 1–7, 2024.

[24] Joo Chan Lee, Daniel Rho, Xiangyu Sun, Jong Hwan Ko, and Eunbyung Park. Compact 3d gaussian representation for radiance field. *arXiv preprint arXiv:2311.13681*, 2023.

[25] Simon Niedermayr, Josef Stumpfegger, and Rüdiger Westermann. Compressed 3d gaussian splatting for accelerated novel view synthesis. *arXiv preprint arXiv:2401.02436*, 2023.

[26] Towaki Takikawa, Alex Evans, Jonathan Tremblay, Thomas Müller, Morgan McGuire, Alec Jacobson, and Sanja Fidler. Variable bitrate neural fields. In *ACM SIGGRAPH 2022 Conference Proceedings*, pages 1–9, 2022.

[27] Lingzhi Li, Zhen Shen, Zhongshu Wang, Li Shen, and Liefeng Bo. Compressing volumetric radiance fields to 1 mb. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4222–4231, 2023.

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

[29] Martin Thomson and Cory Benfield. HTTP/2. RFC 9113, June 2022.

[30] Mike Bishop. HTTP/3. RFC 9114, June 2022.

[31] Christer Ericson. *Real-time collision detection*. Crc Press, 2004.

[32] Johnny Huynh. Separating axis theorem for oriented bounding boxes. *URL: jkh. me/files/tutorials/Separating% 20Axis% 20Theorem% 20for% 20Oriented% 20Bounding% 20Boxes. pdf*, 2009.

[33] Kazuho Oku and Lucas Pardue. Extensible Prioritization Scheme for HTTP. RFC 9218, June 2022.

[34] Optimize resource loading with the Fetch Priority API . https://web.dev/articles/fetch-priority.

[35] WebGL Shaders and GLSL . https://webglfundamentals.org/webgl/lessons/webgl-shaders-and-glsl.html.

[36] H2O - an optimized HTTP server with support for HTTP/1.x, HTTP/2 and HTTP/3. https://github.com/h2o/h2o/?tab=readme-ov-file/.

[37] Gurobi Optimization. https://www.gurobi.com/.

[38] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics (ToG)*, 36(4):1–13, 2017.

[39] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (ToG)*, 37(6):1–15, 2018.

[40] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[41] VMAF: The Journey Continues. https://medium.com/netflix-techblog/vmaf-the-journey-continues-44b51ee9ed12.

[42] VMAF - Video Multi-Method Assessment Fusion. https://github.com/Netflix/vmaf.

[43] tc(8) — Linux manual page . https://man7.org/linux/man-pages/man8/tc.8.html.