Stochastic Gaussian Splatting

MSc CS Thesis project

J.A. Rijsdijk

Stochastic Gaussian Splatting

by

J.A. Rijsdijk

Joris
Rijsdijk

Thesis Advisor:	R. Marroquim
Daily co-supervisor:	C. Peters
Project Duration:	November, 2024 - July, 2025
Research Group:	Computer Graphics & Visualisation, Delft

Summary

3D Gaussian splatting (3DGS) is an appealing implementation of novel view synthesis, with fast training and render times compared to related methods. However, per-frame sorting and front-to-back alpha compositing lead to a significant decline in performance for scenes with a high number of Gaussians. In particular, the alpha compositing step in the original implementation inefficiently handles overdraw. Therefore, dense clusters of Gaussians observed at a distance significantly increase frame times.

In this thesis, we propose a stochastic rendering approach that significantly improves performance in large scenes, by integrating principles from stochastic transparency and compute-based point cloud rendering. Each frame, we first compute a weight for each Gaussian based on its screen footprint and opacity. These weights are then used to construct an alias table on the GPU. We then splat points, each sampled from a Gaussian selected in proportion to its weight using the alias table, onto an interleaved depth-and-color buffer. By using an atomic operation, we ensure that the points closest to the camera are kept without sorting. To construct the final image, we average multiple samples per pixel, obtained across multiple point cloud passes. Although our approach introduces noise at low sample counts and minor artifacts due to the independent and identically distributed nature of the sampling, these effects can be mitigated using temporal anti-aliasing and rejection sampling respectively.

Our evaluations indicate that the render time of our method depends primarily on the density of Gaussians near the camera, rather than the total number in the scene. This allows our method to efficiently handle large-scale scenes.

Contents

1 Introduction

Novel view synthesis is a task in computer graphics that involves generating images from viewpoints not in the original dataset, which typically consists of photographs of 3D scans captured by a camera or depth sensor. Historically, novel view synthesis tackled the challenge of generating unseen views using techniques such as image-based view synthesis [28], which often struggled with occlusion and computational efficiency. Recent advances, such as neural radiance fields (NeRF) [14] have made this task more achievable by introducing the concept of learning a continuous volumetric representation from a set of images, at high computational costs.

Gaussian splatting [9] is a differentiable rendering technique that approximates a scene using 3D Gaussian primitives. It achieves real-time novel view synthesis, outperforming NeRF in render times while maintaining visual quality.

Gaussian splatting uses a set of 3D Gaussian primitives to approximate a scene, where each Gaussian features a view-dependent color. In the original implementation, Gaussians are sorted based on distance to the camera, and each Gaussian is projected onto the view plane of the camera. To divide the workload better among GPU threads, the screen is subdivided into small tiles, and each tile collects the list of projected Gaussians that intersect it. Then, per tile each gaussian is rasterized and the alpha blended.

This approach has some issues. The color of a pixel is computed by iterating over all Gaussians in the tile. Therefore, overdraw becomes a serious problem. When the entire scene is seen at a distance, the frame rate eventually drops below real-time, as more Gaussians start to occupy the same pixels. Moreover, as the number of Gaussians in a scene grows, the sorting becomes increasingly problematic, as the Gaussians have to be resorted every frame in accordance with the current view matrix.

We address these issues by introducing a stochastic approach following principles from stochastic transparency [5] and compute-based point cloud rendering [23]. Stochastic transparency is a form of order-independent transparency, thus forgoing the need for sorting. It replaces the alpha blending stage by a stochastic process. Fragments are discarded with probability $(1 - \alpha)$, ensuring that after sufficient samples, the result converges to the sorted alpha blending outcome. The use of stochastic transparency not only eliminates the need for expensive sorting but also opens the door for novel sampling strategies that adapt to the scene's complexity.

For our approach, we propose to generate only the fragments that are not discarded, in the form of a point cloud. Gaussians that are large on screen will have more points dedicated to them than small Gaussians. To this end, we first compute the weight of a Gaussian every frame (see Section 3.2). Based on these weights, we compute an alias table (see Section 3.3) on the GPU [11], to enable fast selection of Gaussians proportional to their weight.

Then, we generate and directly splat a point cloud onto the screen, with the number of points proportional to the sum of weights (see Section 3.4). Each point is independently and identically distributed, and first selects a particular Gaussian using the alias table. The position of the point is sampled from the selected Gaussian projected onto the screen. The point is splatted onto an interleaved depth-and-color buffer using an atomic min operation [23]. The atomic operation will efficiently and reliably provide the closest point per view ray.

However, our approach does suffer from some limitations (see Section 3.5). Due to the independently and identically distributed nature of the points, points drawn from the same Gaussian may splat to the same pixel. Furthermore, we cannot guarantee that every pixel expected to reach full opacity will receive enough points to do so. Using rejection sampling, we mostly mitigate this issue. Moreover, as we use a stochastic process, our results are noisy. By increasing the number of samples per pixel, noise is reduced. We also use progressive rendering and TAA to reduce the noise further.

Compared to related work, our method has several promising qualities. Firstly, it deals extremely well with overdraw. Large clusters of Gaussians seen at a distance are rendered extremely quickly, whereas existing techniques will experience a slowdown in frame times due to a large number of Gaussians occupying the same pixel. Moreover, the render time of our technique correlates primarily with the density of Gaussians near the camera, rather than with the overall number of Gaussians. Finally, we propose some additional performance optimizations that results indicate to be beneficial when dealing with extremely large scenes. Therefore, our technique is a promising approach to rendering vast Gaussian splatting scenes.

2 Related Work

2.1. Novel View Synthesis

Novel view synthesis is a fundamental task in computer graphics, aiming to generate photorealistic images from viewpoints that are new or not included in the dataset. Traditional techniques such as image-based view synthesis [28] and view synthesis with multiplane images [24] struggle with handling occlusions or view-dependent effects such as reflections. Recent advances have significantly improved the quality and efficiency, enabling more use cases such as applications in virtual reality.

2.1.1. NeRF

Neural radiance fields (NeRF) [14] introduced a new approach to novel synthesis by learning a continuous volumetric representation of a scene. It uses a neural network to map spatial coordinates and viewing directions to color and density values of a radiance field, unlike traditional voxel-based methods. The method is based on direct volume rendering (DVR), where it accumulates radiance along camera rays, resulting in high-quality images with smooth view transitions. However, the training requirements and render times associated with NeRF limit its applicability to real-time scenarios.

2.1.2. Gaussian Splatting

The original 3D Gaussian Splatting (3DGS) paper by Kerbl et al. [9] achieves novel view synthesis with much faster render and training times compared to other state-of-the-art techniques such as NeRF. It represents a scene as a collection of 3D-Gaussian distributions, each with a mean $\mu \in \mathbb{R}^3$, rotation $r \in \mathbb{R}^4$ represented as a quaternion, scale $s \in \mathbb{R}^3$, a view-dependent RGB color (spherical harmonics [15], with 48 coefficients), and an opacity. Unlike NeRF, 3DGS uses rasterization instead of direct volume rendering. The Gaussians are projected onto the view plane, and an axis-aligned bounding box (AABB) around them is rasterized directly onto the screen –similar to billboard rendering– with α values computed based on the distance to the mean of the Gaussian. The Gaussians are transparent, and the value of each pixel is composited using alpha blending. To this end, the Gaussians have to be sorted based on the depth to the camera.

2.1.3. Improvements

Since the release of 3DGS there have been many subsequent works improving the technique. The original approach has major popping issues, where small changes in the camera's transformations displace the render order of Gaussians. This issue can be solved by implementing alternative transparency methods, such as hybrid transparency or other order independent transparency techniques, that do not rely on global sorting sorting of the Gaussian primitives [7].

NeRF excels in image quality by volume rendering a radiance field directly. To this end, Mai et al. employ ray tracing to directly render a volume of constant-density ellipsoids that represent the scene, similar to 3DGS, rather than the rasterization pipeline [12]. Despite modern hardware, this still has a worse computational performance compared to the original implementation.

Our approach can approximate direct volume rendering and thus resolve popping artifacts by splatting points in 3D, as discussed in Section 3.5. As we currently do not optimize scenes with this in mind, we instead splat points onto 2D Gaussians projected onto the screen to better approach the results from 3DGS.

2.1.4. Stochastic Splats

A key limitation of 3DGS is its reliance on depth-sorted rasterization for accurate alpha compositing. Kheradmand et al. [10] propose StochasticSplats, an alternative forward and backward rendering method that removes the need for sorting by leveraging an unbiased Monte Carlo estimator of the volume rendering equation. Much like 3DGS, it generates fragments on the AABB around each projected Gaussian. However, instead of alpha blending, it uses stochastic transparency. Fragments are discarded proportionally to their alpha value. Compared to 3DGS, it achieves up to 4× faster rendering. However, resulting images suffer from noticeable noise at lower sample counts. This can be mitigated using temporal anti-aliasing and denoising.

Our approach shares similarities with StochasticSplats, but we aim to generate only the fragments that would not be discarded. To this end, our render pipeline is vastly different than theirs.

2.2. Transparency Rendering

Transparency is a challenge in computer graphics because it requires correctly combining multiple surface layers that may be partially visible. Transparency as a property of a surface is commonly expressed as an α value $\in [0,1]$, where $\alpha = 0$ means the surface is completely transparent, and $\alpha = 1$ means that it is completely opaque. Given N fragments sorted by ascending distance to the camera for a particular pixel p, we compute its color C_p based on the colors C_i and opacities α_i of the fragments:

$$C_p = \sum_{i=0}^{N-1} \left(C_i * \alpha_i * \prod_{j=0}^{i-1} (1 - \alpha_j) \right).$$

As fragments are generated in arbitrary order during rasterization, a depth buffer is used to ensure that the fragment closest to the camera for a given screen pixel is rendered. When a fragment is drawn to the screen, the depth buffer is updated occluding any fragments further away in the process. Since transparent objects only partially occlude other geometry, the depth buffer cannot be used as effectively, as it traditionally only stores one depth value per pixel in the render target.

To render the transparency correctly, some approaches rely on sorting [19]. When the fragments are sorted, they can be blended perfectly using the formula introduced prior. However, sorting introduces a large overhead to the rendering, as it has to be repeated every frame due to changing camera parameters or geometry.

2.2.1. Order-Independent Transparency

Alternatively, order-independent transparency (OIT) techniques aim to render transparent objects without relying on sorting, thereby mitigating one of the key computational bottlenecks in traditional transparency methods [26].

Hybrid Transparency

Hybrid transparency [13] combines the strengths of sorted and unsorted transparency. In this approach, per-fragment depth-sorted transparency is applied to the K fragments closest to the camera in each pixel to ensure that the most significant contributions are composited accurately. The remaining fragments are then accumulated using a weighted sum, which approximates their combined color without the overhead of full sorting.

Moment-Based Transparency

Moment-based transparency [16] approximates the cumulative effects of transparency along each viewing ray using a compact set of statistical moments. In an initial additive render pass, the method computes the moments of the logarithmic transmittance per view ray, leveraging the logarithm to transform the multiplicative accumulation of transmittance into an additive formulation. In a second pass, fragments are blended directly using this approximation of transmittance at any depth. For more complicated scenes with many transparent surfaces, trigonometric moments improve robustness by better capturing transparency variations across depth. Overall, this approach effectively avoids numerical instability and popping artifacts, even in complex scenes.

Multi-Layer Alpha Blending

Multi-layer alpha [22] achieves order-independent transparency by maintaining a fixed-size blending array B per pixel, which stores the color, transmittance and depth values of transparent fragments ordered by increasing depth to the camera. New fragments are inserted based on their depth values. Since array size is fixed, the algorithm merges two adjacent rows to make room for a new fragment. This operation preserves the combined color contribution and transmittance of the rows, and keeps the closest depth. The resulting image is rendered using front-to-back compositing on the final values in B, and compositing that result with the opaque background using the product of the transmittance in B.

Stochastic Transparency

Our approach uses stochastic transparency [5]. Instead of rendering transparent surfaces using alpha blending, all fragments are rendered as either fully opaque or fully transparent. To achieve the desired level of transparency, fragments are discarded with $1 - \alpha$ probability. This stochastic process introduces noise. By taking multiple samples per pixel and averaging them in a later pass, this noise is mitigated, and the final composited values approach the reference values rendered using per-fragment sorted alpha blending.

Several techniques, such as depth sampled stochastic transparency and temporal anti-aliasing [27], can help to reduce this noise.

2.3. Point Cloud Rendering

For our approach, instead of blending billboards using stochastic transparency, we sample points per Gaussian, and render those. This requires an efficient way to render points. Schütz et al. [23] came up with an optimized GPU-compute-based pipeline to efficiently do this, vastly outclassing the performance of rendering points using OpenGL primitives. In a first pass, they use a 64-bit interleaved buffer, with one 64-bit integer per pixel, that stores depth and color, where the last 24 bits represent a color with eight bits per channel. Using a single atomic min operation, they write to this buffer for all points. Then, in a second pass, they transfer the results from this storage buffer into a texture.

They directly use the linear depth value for the depth buffer, as no interpolation is required. As atomic operations can be slow if many points occupy the same pixel, an early depth test is executed, essentially comparing the computed value with the currently stored value. Only if the current value is lower is the atomic min operation executed.

3
Method

3.1. Overview

Our approach presents an alternative compute-based renderer for Gaussian splatting scenes, optimized using 3DGS, which builds upon the principles of point cloud rendering and stochastic transparency. Rather than projecting Gaussians onto the view plane, rasterizing them as billboards and discarding fragments stochastically with probability $(1 - \alpha)$, we generate only the necessary fragments in the form of a point cloud that is regenerated every frame.

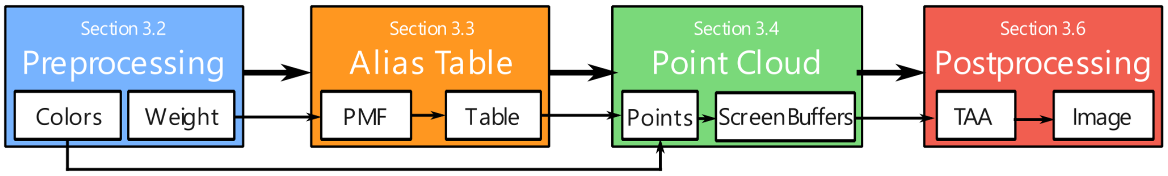

Figure 3.1: The rendering pipeline of our approach.

Our rendering pipeline has four primary stages: In the first preprocessing stage, all Gaussians are processed to compute their weight, based on opacity, and the size of the projected 2D-Gaussian in the view plane. Furthermore, the color of each Gaussian is computed based on the current camera position relative to the mean of the Gaussian and its spherical harmonics constants.

In the second stage, the array of Gaussian weights is used to construct an alias table. This data structure facilitates sampling a probability mass function (PMF) in constant time, thus enabling the efficient stochastic selection of Gaussians with respect to their weights.

In the third stage, we generate a point cloud and splat it into a single interleaved depth-color buffer. The total point count is directly proportional to the image resolution and the sum of all weights. Per point, we select one of the Gaussians with probability proportional to their weight in that frame. The selected Gaussian is projected onto the plane parallel to the view plane, positioned at the same depth as the mean of the Gaussian. The point is then sampled independently and identically (i.i.d.) from this projected Gaussian distribution, after which it will overwrite the value in the screen buffer at its screen position if its distance to the camera is smaller than the current value present. We perform r_p render passes, corresponding to the number of samples per pixel (SPP), to mitigate noise introduced by this stochastic method.

In the fourth and final stage, render passes are combined and written to the final screen buffer. We also apply temporal accumulation techniques to improve the image quality over multiple frames.

3.2. Gaussian Preprocessing

3.2.1. Gaussian Weight

To accurately distribute points across all Gaussians, each Gaussian needs a weight value interpreted as sampling probability. E.g., a Gaussian close to the camera requires more points than a Gaussian of the same size but further away, due to perspective projection. Each Gaussian is geometrically defined by a mean $\mu_g \in \mathbb{R}^3$, a scale $s_g \in \mathbb{R}^3$, and a rotation $r_g \in \mathbb{R}^4$, represented as a unit quaternion. Note that r_g and s_g can also be represented as transformation matrices $R_g \in \mathbb{R}^{3\times3}$ and $S_g \in \mathbb{R}^{3\times3}$, respectively. To compute the weight of the Gaussian, we evaluate the integral of the projected 2D-Gaussian defined by its covariance matrix $\Sigma' \in \mathbb{R}^{2\times2}$. The normalization factor $\sqrt{|2\pi\Sigma'|}$ is the result of this integral, representing the area under the Gaussian, which we use as a scaling factor for determining its weight:

$$\sqrt{|2\pi\Sigma|} = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \exp\left(-\frac{1}{2}\left([u,v]^T - m\right)^T \Sigma^{-1}\left([u,v]^T - m\right)\right)\mathrm{d}u\mathrm{d}v. \tag{3.1}$$

We take the integral from $(-\infty,\infty)$, as we want to sample points from anywhere in the Gaussian. Gaussians that are visibly cut off by the screen boundary, i.e. the projected mean of the Gaussian is less than 3 standard deviations outside the boundary, do not need to be corrected, as they will generate points increasingly more outside the screen, the more cut off they are. This means Gaussians only generate the correct number of points if their weight is treated as if a Gaussian is fully visible.

To obtain Σ', we use the local affine approximation of the projective transformation as introduced by Zwicker et al. [29], and as defined for use in Gaussian splatting by Kerbl et al. [9]. Firstly, compute the Jacobian of the projection transform:

$$J = \begin{pmatrix} \frac{1}{t_z} & 0 & -\left(\frac{1}{t_z}\right)^2 \cdot t_x \\ 0 & \frac{1}{t_z} & -\left(\frac{1}{t_z}\right)^2 \cdot t_y \end{pmatrix}. \tag{3.2}$$

Secondly, compute the view projection matrix $T \in \mathbb{R}^{2\times3}$, and project the covariance matrix $\Sigma = R_g \cdot S_g \cdot S_g^T \cdot R_g^T$ by:

$$T = J \cdot W, \qquad \Sigma' = T \cdot \Sigma \cdot T^T. \tag{3.3}$$

Here, t is the mean of the Gaussian projected into view space using the affine-inverse model matrix of the camera $V \in \mathbb{R}^{4\times4}$, and W is the top-left 3×3 block of V.

Finally, based on Equation 3.1, the weight of each Gaussian equals

$$w_g = \sqrt{|2\pi\,\Sigma'|} \cdot \alpha_g, \tag{3.4}$$

where α_g is the opacity of the Gaussian. To prevent unstable weight values caused by projection, Gaussians are culled if their mean is in front of the near-plane. Moreover, we implement frustum culling by computing the axis-aligned bounds of each Gaussian at three standard deviations in Normalized Device Coordinates (NDC) and culling them if it does not intersect the clipbox: $x, y \in [-1,1]$. In either case, we can trivially cull a Gaussian by assigning it zero weight. The bounds are computed as follows [9]:

$$p_x = t_x \pm 3\sqrt{\Sigma_{0,0}},\ p_y = t_y \pm 3\sqrt{\Sigma_{1,1}}. \tag{3.5}$$

3.2.2. Color

In the original 3D Gaussian splatting approach, spherical harmonics are used to represent view-dependent colors. Spherical harmonics are functions capable of encoding low-frequency signals over the surface of a sphere [15]. The resulting signal is calculated as a weighted sum of spherical harmonics basis functions evaluated at a specific angle, with the weights determined by a coefficient corresponding to each basis function.

For Gaussian splatting, we commonly use spherical harmonics of degree $L = 3$, which results in $(L+1)^2 = 4^2 = 16$ coefficients per color channel, totaling 48 coefficients per Gaussian to encode color information. The colors are computed based on the view direction, i.e. the difference between the mean of the Gaussian and the camera position. This results in uniform colors on the surface of any Gaussian given an arbitrary camera position, and thus, colors can be precomputed once per Gaussian per frame.

In contrast, opacity is independent of the view direction and is represented by a single scalar value per Gaussian.

3.3. Alias Table

Our implementation requires the ability to select a Gaussian proportional to its weight. Essentially, we want to sample the following probability mass function (PMF):

$$P(X = i) = \frac{w_i}{\sum_{k=0}^{N-1} w_k}. \tag{3.6}$$

To select a Gaussian efficiently, we make use of an alias table [25]. Once built, this data structure enables sampling a discrete PMF in constant time while requiring only two memory reads. This is achieved by computing two arrays, a split s and an alias a array. The former dictates the probability s_i that the element i should be chosen, and a_i provides an index to an alternative element.

The sampling process works as follows:

1. Generate a random uniform index $i \in \{0, ..., N-1\}$, where N is the size of the original PMF.
2. Choose i with probability s_i; otherwise, select the index given by a_i.

As the weights and, consequently, the PMF depend on the camera position, the alias table has to be rebuilt every frame. Most implementations rely on splitting the input data set into two sets, a light and a heavy set, based on whether the weight of an element is below or above the average weight. However, these algorithms are challenging to implement on GPUs, as they rely on iterative loops that process the light and heavy sets until one set is empty.

Lehmann et al. provide a way to efficiently build an alias table on the GPU [11]. Their approach splits the building step into many subproblems, which can then be solved in parallel. It achieves building speeds on consumer hardware (RTX 2080) in the order of $\approx 10^9 \frac{\text{weights}}{\text{s}}$, making it fast enough to be integrated into a rendering pipeline.

3.4. Point Cloud Splatting

Each frame, we stochastically generate a point cloud to represent the scene geometry and we directly splat the points onto a screen-space buffer with interleaved depth and color channels. In total, we generate $W \cdot r_w \cdot r_h \cdot r_p$ points, where W is the sum of all Gaussian weights, r_w, r_h are the resolution width and height respectively, and r_p is the number of samples per pixel.

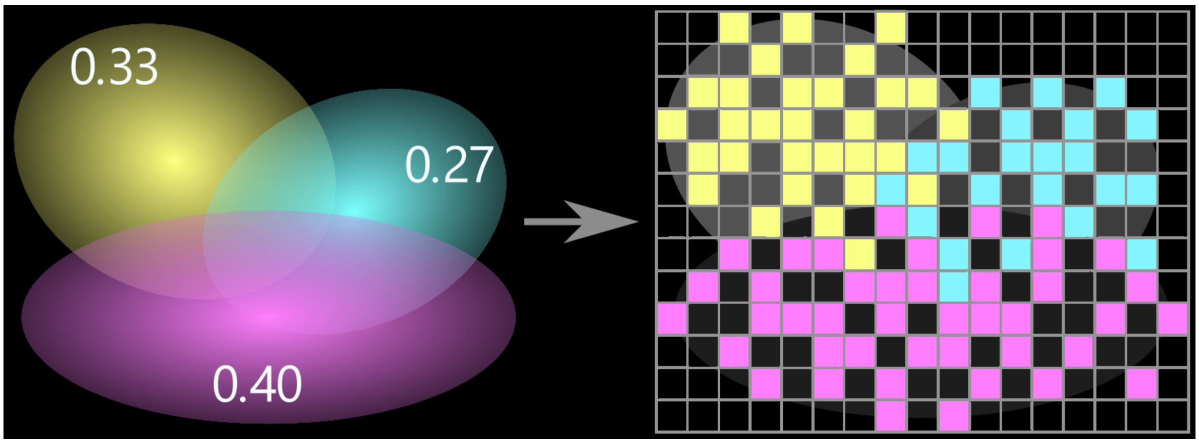

Figure 3.2: Illustration of the splatting process. For each point, a Gaussian is selected proportional to its projected size and opacity, illustrated here by the number within each Gaussian. Then, the position of the point is drawn from the selected Gaussian distribution. The depth value of each point equals the depth of the mean of the selected Gaussian. If this depth value is smaller than the current depth value in the screen buffer, the point is splatted. This process is repeated independently for every render pass, and averaged to obtain the final image.

For each point, we select a single Gaussian randomly. By generating a random uniform variate vector $u \in [0,1)^2$ using PCG2D [8], we use u to select a Gaussian using the alias table computed prior. For our primary implementation, we compute the position of a point p to be on a plane through the mean of the Gaussian parallel to the view plane, to best match the 3DGS implementation. p is computed as follows: First, we compute the covariance matrix of the selected Gaussian Σ, and project it into $\Sigma' \in \mathbb{R}^{2\times 2}$ once again, as described in section 3.2.1. We also compute the projected mean of the gaussian $\mu_{NDC} \in \mathbb{R}^3$ in NDC using the view-projection $M_{VP} \in \mathbb{R}^{4\times 4}$ matrix:

$$\mu_{NDC} = \frac{M_{VP} \cdot \mu_g}{(M_{VP} \cdot \mu_g)_3} \tag{3.7}$$

Then, we sample a vector of standard Gaussian variates $z \in \mathbb{R}^2$ ($\mu = 0$, $\sigma = 1$) using another PCG2D uniform variate transformed by the Box Muller algorithm [3]:

$$z = \begin{pmatrix} \sqrt{-2\ln(u_0)}\,\cos(2\pi u_1) \\ \sqrt{-2\ln(u_0)}\,\sin(2\pi u_1) \end{pmatrix}. \tag{3.8}$$

We compute the Cholesky decomposition of Σ': $L \in \mathbb{R}^{2\times 2}$, and transform the standard Gaussian variates into ones sampled from the selected Gaussian using $z_g \in \mathbb{R}^2 = L \cdot z$ [6]. Then, we compute the final screen-buffer position of the point $p = \begin{pmatrix} z_{g0} & z_{g1} & 0 \end{pmatrix} + \mu_{NDC}$, and transform p from NDC to integer screen-pixel coordinates. To remain faithful to the 3DGS implementation, we truncate the Gaussians in an axis-aligned bounding box around the mean, computed in Section 3.2.1. We achieve this using rejection sampling.

For each point, we construct a 64-bit value, using 28 bits to represent the linearly quantized Z-coordinate of $\mu_{NDC} \in [-1,1)$, concatenated by the previously computed color in 36 bits. By using 36 bits, 12 bits are used per color channel. As we take multiple samples per pixel, p is combined with the sample index i, to compute the index of the image buffer $j = r_p(p_0 + p_1 \cdot r_w) + i$, ordered using Morton order [23]. We splat 64-bit point values into the image buffer using an atomic min operation, as proposed by Schütz et al. [23].

3.5. Limitations of Point Cloud Splatting

Fundamental Discrepancy in Sampling Approaches

As our points are independently and identically distributed (i.i.d.), points sampled from the same Gaussian may occupy the same pixel in the buffer. Furthermore, it is impossible to guarantee that at least one of the points will fall into a given pixel, so 100% opacity cannot be reached. This is in contrast with stochastic transparency, which considers every fragment exactly once. This is essentially identical to stratified Bernoulli sampling in a discrete grid. Let the density value of a Gaussian at a position be $g(x, y)$. Then $G(i, j)$ is the probability to splat into pixel i, j:

$$G(i, j) = \int_{i}^{i+1} \int_{j}^{j+1} g(x, y) \mathrm{d}x \mathrm{d}y. \tag{3.9}$$

Let n be the total number of points we would sample for a particular Gaussian, given its weight and the rendering resolution. The number of i.i.d. points N that fall into a pixel i, j equals

$$N = \sum_{m=0}^{n-1} X_m, \tag{3.10}$$

where

$$X_i := \begin{cases} 1 & \text{with probability } G(i, j) \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

The probability of points overlapping can be computed using the binomial distribution:

$$P(N = k) = \binom{n}{k} G(i, j)^k \left(1 - G(i, j)\right)^{n-k} \tag{3.12}$$

To compare stochastic transparency with our method, let O be the pixel occupancy:

$$O := \begin{cases} 1 & N > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{3.13}$$

For i.i.d. points: $E[O_{\mathrm{P}}] = P(N > 0) = 1 - (1 - G(i, j))^n$.

Stochastic transparency uses an alpha value per pixel, computed at the pixel center. For large Gaussians: $\alpha = g(i + 0.5, j + 0.5) \approx G(i, j)$. For this method, N is either 0 or 1, as each pixel is considered exactly once using a Bernoulli trial with $p = \alpha$. Therefore $E[O_{\mathrm{ST}}] = \alpha \approx G(i, j)$.

To match stochastic transparency, we want the same equation to hold for i.i.d. points $E[O_{\mathrm{P}}] = G(i, j)$. However, this is only trivially true if either $G(i, j)$ equals 0 or 1 –resulting in a completely opaque, or completely transparent surface– or if $n = 1$, i.e. when the Gaussian is very transparent or small. We could bias $G(i, j)$ to approximate $E[O_{\mathrm{P}}] = G(i, j)$ by weighting pixels closer to the mean of a Gaussian higher, but it would come at the cost of greatly increasing the weight of a Gaussian. As this would directly increase the number of points in the point cloud, we choose to use other methods.

Mitigating Sampling Artifacts

When we just consider a single Gaussian, we can take multiple samples occupying one pixel into account by accumulating their color values if the existing depth at the pixel is the same as the newly computed point. Just as before, if the depth of the new point is larger, we discard it. If it is smaller, we overwrite the current value. This approach ensures all points are considered when computing the average like in stochastic transparency. Therefore, we obtain the expected pixel value of $\alpha \cdot c_g$, where c_g is the color of the Gaussian. Extending the use of the atomic min operation by Schütz et al. [23], we achieve this using an atomic operation written using the atomic compare and swap (CAS) operation, as seen in Algorithm 1.

However, when multiple Gaussians overlap in screen-space, the expected value of a pixel containing both is still incorrect. Stochastic transparency relies on occluding fragments to get the correct expected value. As sampled points may overlap, a larger than expected number of samples from occluded Gaussians may pass through, resulting in the final pixel being weighted more by occluded Gaussian than alpha blending dictates. We refer to this phenomenon as porous Gaussians, as they appear to

Algorithm 1 Atomic operation to handle combining points into the screen buffer, taking the possibility of multiple samples occupying the same pixel into account.

```
bool atomicAddColor(uint64_t* address, uint64_t value) {
    const uint64_t MASK_DEPTH = 0xFFFFFFF000000000ULL;  // Upper 28 bits
    const uint64_t MASK_COLOR = 0x0000000FFFFFFFFFULL;  // Lower 36 bits
        const uint64_t target_depth = value & MASK_DEPTH;
        uint64_t old = *address, assumed;
        do {
                assumed = old;
                uint64_t assumed_depth = (assumed & MASK_DEPTH);
                uint64_t new_value = value;
                if (target_depth > assumed_depth) {
                        return true;
                }
                if (target_depth == assumed_depth) {
                        uint64_t new_lower = (assumed & MASK_COLOR) + (value & MASK_COLOR);
                        new_value = (assumed & MASK_DEPTH) | (new_lower & MASK_COLOR);
                }
                if (target_depth < assumed_depth) {
                        new_value = value;
                }
                old = atomicCAS(address, assumed, new_value);
        } while (assumed != old);
        return true;
}
```

have many more small holes in the center compared to the control rendered using standard stochastic transparency, as seen in Figure 3.3.

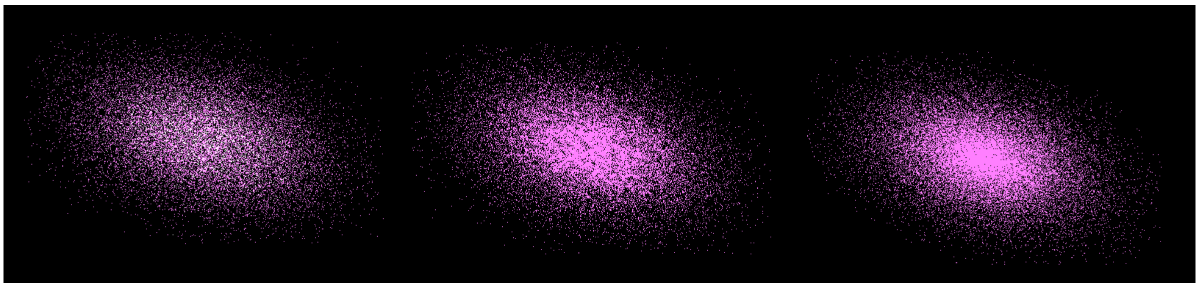

Figure 3.3: Example of the porosity of a Gaussian with uniform color (1, 0.5, 1). Left is rendered using our technique without rejections. Middle is rendered using our technique with $N = 10$ attempts, but with the same number of points as left. Right is a reference rendered using stochastic transparency. Notice how the middle Gaussian is a lot closer to the reference, and has much fewer overlapping (white) pixels.

To largely resolve this discrepancy, we introduce a rejection sampling step. After the initial attempt, we jitter the computed point by the projected covariance of the selected Gaussian, scaled down to 10% its original size, up to N times until a spot with unequal depth is found. While this introduces minor blur, it significantly reduces the porosity of Gaussians, even at $N = 2$ attempts. If a point has been rejected $(N - 1)$ times, the final attempt is allowed to add to the existing color as described previously. Both the scale-down factor and the number of attempts have been empirically chosen based on experiments, see Figures 3.5 and 3.6. Figure 3.4 shows how the porosity of a Gaussian affects its ability to occlude Gaussians behind it, and how rejection sampling helps to mitigate it.

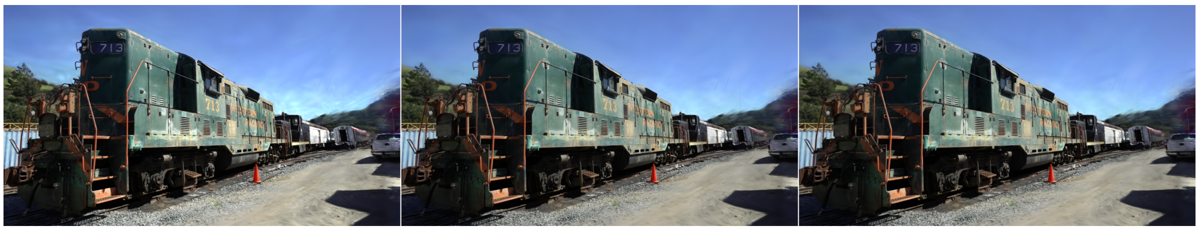

Figure 3.5: The train scene rendered with varying numbers of rejection attempts: $N = 1$ (left), $N = 2$ (middle), and $N = 10$ (right). The $N = 1$ case exhibits overexposure in the sky and on the train due to excessive point overlap. As the difference between $N = 2$ and $N = 10$ is unnoticeable, our approach uses $N = 2$ attempts to balance quality and efficiency.

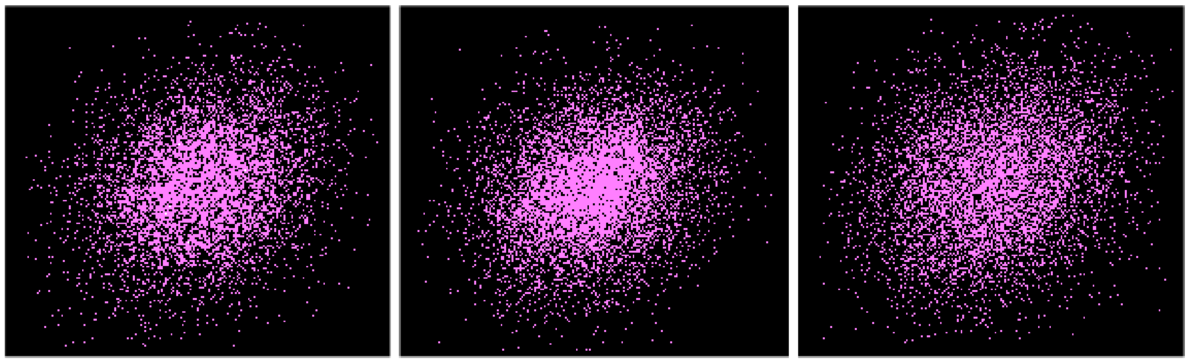

Figure 3.6: Effect of jittering scales on Gaussian rejection sampling. For the second attempt, points are, points are jittered using a Gaussian with 1% (left), 10% (middle), and 100% (right) the size of the projected covariance matrix of the original Gaussian. Points beyond 3 standard deviations are discarded. For our approach we choose a jitter scale of 10%—higher values push the points away from the mean, fundamentally altering the Gaussian distribution, while lower values cause small Gaussians to suffer from porosity.

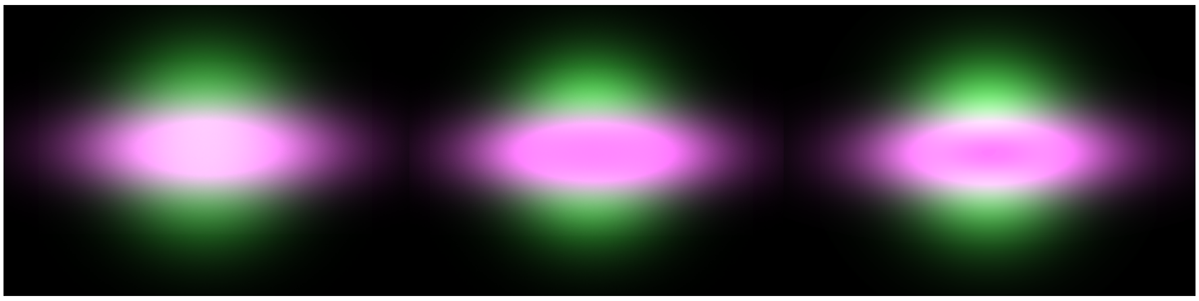

Figure 3.4: Example of how the porosity of a Gaussian affect its ability to occlude other Gaussians. Left is rendered using our stochastic splatting technique without rejections. Middle is rendered using our technique with $N = 10$ attempts, but with the same number of points as left. Right is a reference rendered using 3DGS, i.e. using alpha blending on depth-sorted Gaussians. The middle case is closer to the reference, as it better occludes the generated points from the other Gaussian behind it.

Splatting Points in 3D

These limitations and modifications are in place to come close to the original Gaussian splatting approach. Ideally, we sample the position of a point p from the selected 3D Gaussian directly, and project that point into NDC, i.e.

$$p_{NDC} = M_{VP}\left(R_g(S_g \cdot z) + \mu_g\right). \tag{3.14}$$

This would approximate direct volume rendering of the Gaussian scene, as Gaussians will also blend in the Z-axis of view-space. Additionally, the popping issue that plagues many implementations of Gaussian splatting would be elegantly resolved. However, this method would require the scenes to be optimized with a backward renderer that includes this additional blending. We have therefore chosen to stick to the splatting points in the flat Gaussians projected onto the view plane for this thesis project, but consider 3D splatting promising future work.

3.6. Postprocessing

Once the point cloud has been generated and written into the screen buffer, we need to convert it into a format displayable to the screen. Per screen space pixel $p_{u,v}$, we iterate through the render targets $i \in [0, r_p)$, and extract the last 36 bits per value stored in the buffer. We convert these 36 bits to a color vector $c_{u,v,i} \in \mathbb{R}^3$. Our final color per pixel is the arithmetic mean of the r_p samples per screen pixel pixel coordinate:

$$\frac{1}{r_p} \sum_{i=0}^{r_p - 1} c_{u,v,i}. \tag{3.15}$$

Furthermore, we increase the quality of the renderer when the camera is stationary using progressive rendering, i.e. adding samples over time and displaying the average color per pixel.

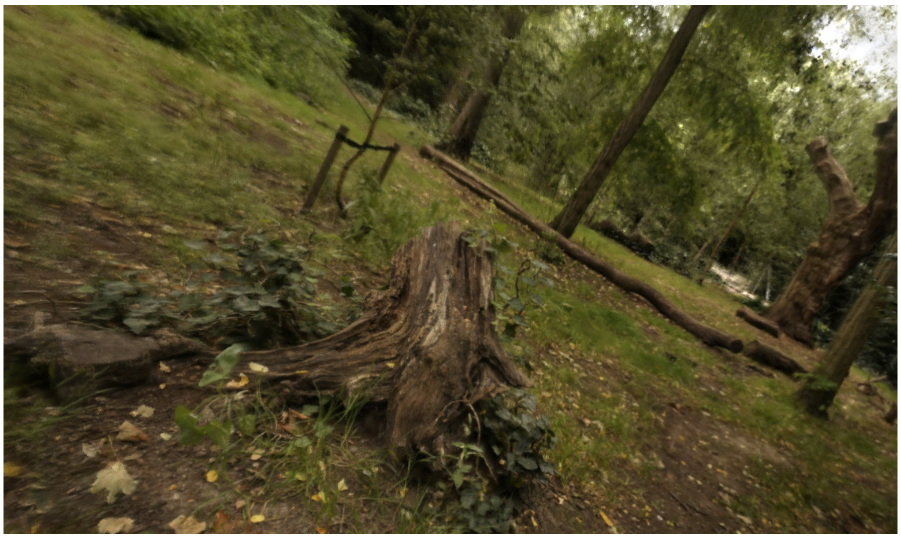

Figure 3.7: Example of a scene rendered at 4 SPP and with TAA enabled. The camera has been moving forward for over 100 frames. Some ghosting around high frequencies areas can be observed.

Additionally, we use temporal anti aliasing (TAA) [27] to reproject and accumulate samples from previous frames to handle camera motion. TAA uses a delta matrix $D = M_i \cdot M_{i-1}^{-1}$, constructed from the current and previous view projection matrices M_i and M_{i-1} respectively, to reposition pixels from the previous frame. This process requires depth values for computing normalized device coordinates, which we obtain as the arithmetic mean from multiple samples per pixel. We then blend both the color and depth per pixel using an exponential moving average, yielding a smoother depth buffer that is more resilient to noise, such as from occlusions by mostly transparent Gaussians. Figure 3.7 shows an example of a scene rendered with TAA enabled.

3.7. Performance Considerations

Several aspects of our render pipeline require some extra attention to be optimized. Here are some decisions we have made to increase the computational performance of our approach significantly.

3.7.1. Alias Table

For larger scenes ($N > 10^7$), building the alias table becomes increasingly computationally expensive.

Compaction

Many of the Gaussians in the scene may be frustum culled by the preprocessing step, by setting their weight to zero. It is wasteful to include these weights in the building step of the alias table, as their respective Gaussians will not be able to be sampled. Therefore, we apply a stream compaction [2] preprocessing step to the alias table-building process. We create a stencil for each weight, assigning 0 if

≤ 0, and 1 otherwise. We then compute the exclusive prefix sum of this stencil. Non-zero weights are written to their new index, and we store a mapping from the new to the old index using the result of the prefix sum as a mapping. The alias table size is now reduced to the sum of the computed stencil vector.

Grouping
In large scenes, the number of Gaussians may approach or even exceed the point count. After compaction, we accelerate alias table construction by grouping weights into clusters of size k. We define the aggregated weight for each group g:

$$s_g = \sum_{j=0}^{k-1} w_{(gk+j)}. \tag{3.16}$$

Using the alias table, we sample a group proportional to s_g. We use inverse CDF sampling to sample a weight proportionally to the weights within the group. As build times scale linearly with input size, this effectively cuts the building time by k, at a slight increase in sampling cost.

Sampling
During the point cloud generation, we sample the alias table to select a Gaussian. However, as the input to the alias table is uniform between 0 and 1, threads within a warp often access memory locations scattered across VRAM. This randomness inhibits effective memory caching, causing significant latency due to long scoreboard stalls. To enhance memory coherence, we adopt a jittered sampling approach [20]. Rather than selecting rows in the alias table using a uniformly distributed index $u \in [0, 1)$, we remap the uniform variate to a smaller section given the thread block id i of the active thread. Specifically, given n thread blocks:

$$u_x = \frac{i + u_x}{n}. \tag{3.17}$$

3.7.2. Deferred Spherical Harmonics

In the initial processing step, the color of each Gaussian is computed based on the camera position relative to its mean and its spherical harmonics coefficients. This step is skipped for any Gaussians that are culled in that frame, but when many Gaussians are fully occluded by Gaussians closer to the camera, computational effort might be wasted. Similar to deferred rendering [18], we render the index of the Gaussian instead of the color to the interleaved depth color buffer. In the postprocessing pass, we load the coefficients corresponding to the index of the Gaussian of the pixel and compute the color as explained in Section 3.2.2. We expect this method to improve performance over preprocessing the spherical harmonics if the size of the buffer is smaller than the expected number of processed –not frustum culled– Gaussians: $r_w \cdot r_h \cdot r_p \leq E_{\text{visible}}[G]$.

3.7.3. Data Quantization

Uncompressed, each Gaussian takes up 236 bytes, mostly due to the 48 SH coefficients. During the point cloud generation stage we sample Gaussians in a less memory-coherent manner than in 3DGS, due to the i.i.d. sampling. As our approach samples Gaussian randomly based on the alias table, we expect the memory throughput to be lower due to fewer cache hits. To facilitate a higher Gaussian processing rate, we compress each Gaussian using quantization. An approach has been explored and open sourced by Niantic Labs [17]. However, we use a slightly different approach, as ours is intended to optimize for fast memory reads rather than storage space. The resulting difference in storage size is illustrated in Figure 3.8.

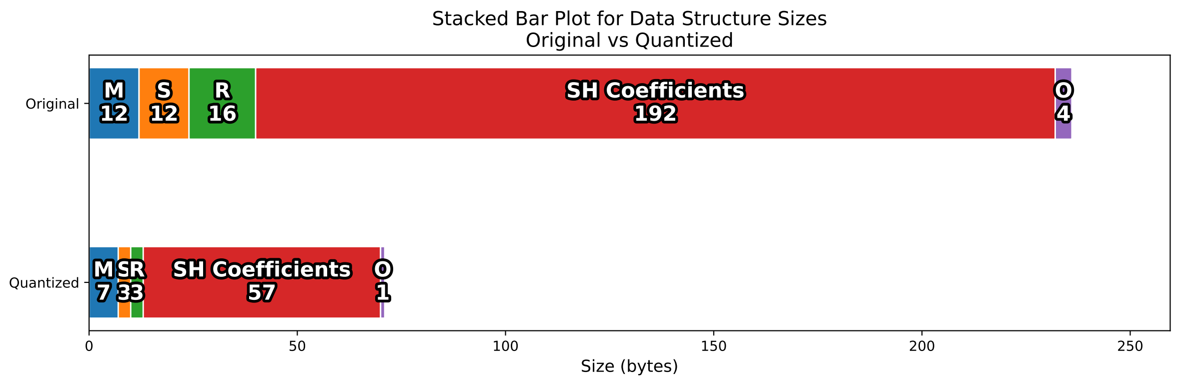

Figure 3.8: Visual comparison of the size in bytes of the original and quantized Gaussian structs. Blue represents the mean, orange scale, green rotation, red SH coefficients, and purple opacity.

As discussed before, the Gaussian is comprised of three parts that describe its geometry; a mean $\mu_g \in \mathbb{R}^3$, a scale $s_g \in \mathbb{R}^3$, and a rotation $r_g \in \mathbb{R}^4$, stored as a unit quaternion.

The mean is quantized linearly. First, we compute the scene bounds by finding the minimum and maximum for each spatial dimension. We then normalize each coordinate to the unit interval and map it to 7 bytes, yielding 19 bits of precision for the x and z coordinates and 18 bits for the y coordinate.

For the scale, we first encode each component in logarithmic space and then apply a similar linear quantization using 8 bits per component. This logarithmic encoding aids in capturing the vast dynamic range expected from the scales, while using only 8 bits.

Unit quaternions have a redundancy: only three components need to be stored, as the fourth can be reconstructed via the normalization constraint, i.e. quaternion q can be destructed into v and w:

$$q = (v, w) \quad \text{with } v \in \mathbb{R}^3 \text{ and } w \in \mathbb{R}. \tag{3.18}$$

The remaining components cannot be directly linearly quantized, as this leads to extremely low precision when w is small. Instead, we apply an exponential mapping [21]. The forward exponential mapping $f_{\text{fem}} : \mathbb{R}^4 \to \mathbb{R}^3$ is given by:

$$f_{\text{fem}}(q) = \frac{2}{\pi} \frac{\arctan\left(\frac{\sqrt{1-w^2}}{w}\right)}{\sqrt{1-w^2}} v\,. \tag{3.19}$$

Similarly, for a vector $v \in \mathbb{R}^3$, the inverse exponential mapping $f_{\text{iem}} : \mathbb{R}^3 \to \mathbb{R}^4$ is defined as:

$$f_{\text{iem}}(v) = \left(\frac{\sin\left(\frac{\pi}{2}\|v\|\right)}{\|v\|} v,\ \cos\left(\frac{\pi}{2}\|v\|\right) \right). \tag{3.20}$$

To quantize the quaternion, the forward exponential mapping is applied, resulting in a vector $v \in \mathbb{R}^3$. This vector is then mapped from $[-1, 1]$ to $[0, 255]$, quantized and stored in three bytes, one per component. The inverse mapping is applied to retrieve the original unit quaternion.

To further optimize memory accesses, we store the geometry of the quantized Gaussian in an array of structs. This reduces the number of memory lookup operations as all relative data is now located next to each other in memory. This aids performance as we always require the mean, rotation and scale at the same time.

Besides geometric properties, opacity and spherical harmonics (SH) values are quantized as well. Opacity is linearly quantized within [0,1] and stored as an unsigned byte. For the SH values, the base coefficients are maintained at full precision, while the remaining coefficients are normalized relative to the base and stored as bytes. Since these values are only needed during the initial processing stage, and not during point cloud splatting, they reside in dedicated buffers. Moreover, if the weight of a Gaussian falls below a given threshold, it will not be rendered that frame. Therefore we skip computing its color, and also forgo the memory reads associated with it.

4 Results

We implemented our proposed render pipeline from scratch in CUDA 12.6. We refer to this as our stochastic splatter. As discussed in section 3.5, we use 2 rejection steps to approximate 3DGS in terms of rendering quality by reducing Gaussian porosity, while maintaining high performance.

Additionally, we created a simplified compute-based stochastic rasterizer similar to the one by Kheradmand et al. [10]. It generates a billboard for each Gaussian and stochastically discards fragments based on their alpha value. The purpose of this secondary implementation is to validate the correctness our algorithms, and thus show that the deviations between our results and the original implementation likely originate from the independently and identically distributed points.

All our testing has been done on a single machine equipped with a 10 GB VRAM NVIDIA RTX 3080, at stock settings. We render all the scenes in full HD 1920 × 1080 resolution.

Test Scenes
As our renderer is designed to match the output of the original implementations as closely as possible, we have used scenes optimized using 3DGS, provided by the authors of the original paper, trained on the MipNerf360 dataset [1]. More specifically, we have used the following eight scenes (in order of increasing Gaussian count): train (1.0M), counter (1.2M), bonsai (1.2M), room (1.6M), kitchen (1.9M), stump (5.0M), garden (5.8M), bicycle (6.1M).

We have created an additional scene to gauge how our implementation scales to even larger scenes. We separated the train from the original train scene, and repeated it 5 × 5 times horizontally, which was the largest configuration we could fit within the 3DGS renderer on our GPU. This resulted in a new scene *repeated_train* with 17.8M Gaussians, as shown in Figure 4.1.

4.1. Quality

To measure the quality of our renders, we compare renders of our results with the original implementation. Our renderer is designed to match the output given the same scenes as the original. Figure 4.2 shows renders at different samples per pixel (SPP) for two scenes. The resulting images of our stochastic

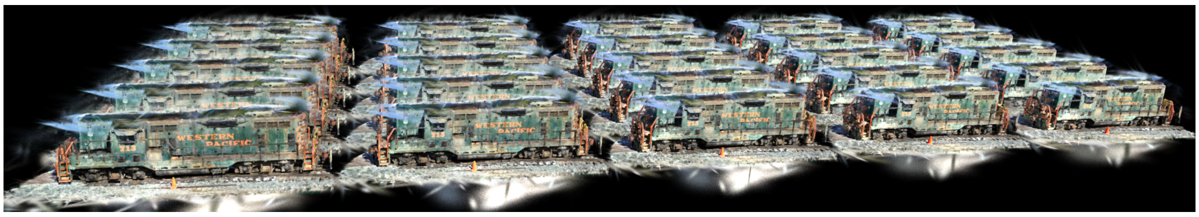

Figure 4.1: The repeated train scene (17M Gaussians) rendered using our implementation. It is manually created by extracting the train from the train scene, and repeating it.

rasterizer closely match the original. Our stochastic splatter slightly deviates from the original, primarily with respect to high-frequency details. Note how the tips of some grass blades lack a strong specular highlight. Flat surfaces such as the side of the train are virtually identical to the reference.

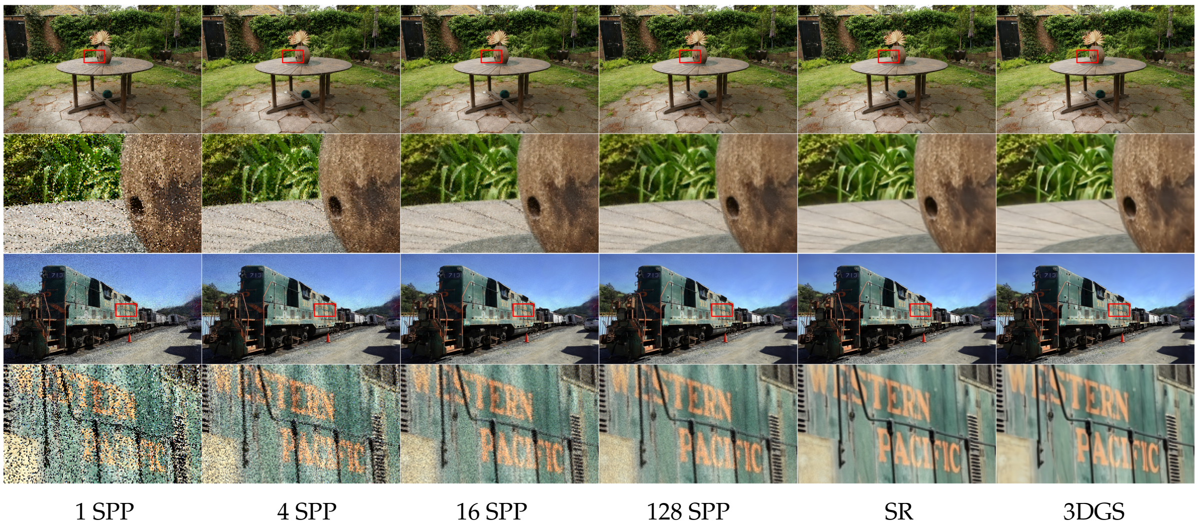

Figure 4.2: Quality Comparison of our renderer compared to the reference in the garden (top) and train (bottom) scenes. From left to right, the images depict our stochastic splatter at different samples per pixel (SPP): 1 SPP, 4 SPP, 16 SPP, 128 SPP. The second to last column contains results of our reference stochastic rasterizer (SR) at 1024 SPP, and the last column contains a reference from the original Gaussian splatting implementation by Kerbl et al.

Figure 4.3 gives an impression of the rendering quality at different samples per pixel, measured as the root mean squared error (RMSE) between our stochastic splatting method and 3DGS. Noise steadily decreases as the number of samples increases.

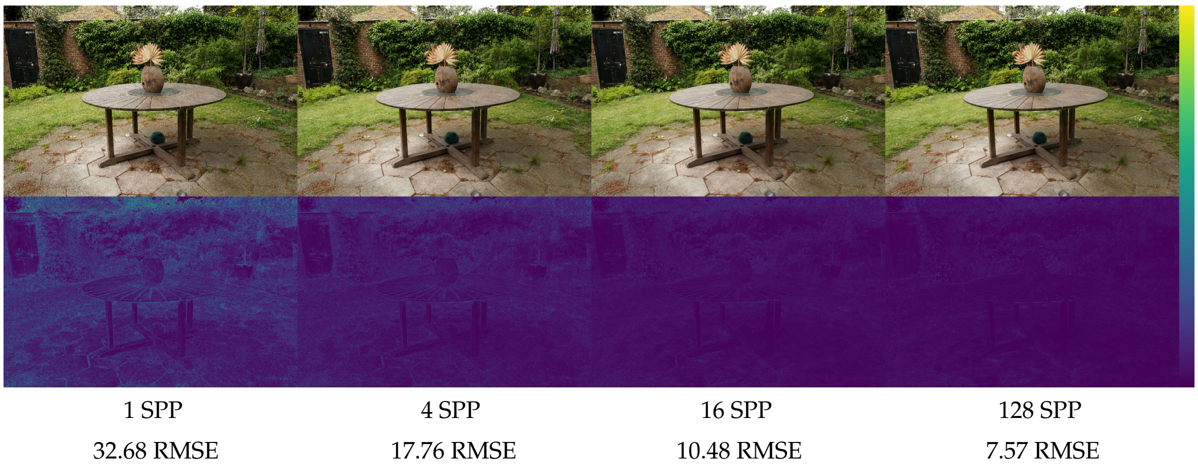

Figure 4.3: Quality comparison of results from our stochastic splatting renderer compared to the 3DGS reference in the garden scene. From left to right, the images depict the root mean squared error (RMSE) at 1, 4, 16, and 128 SPP.

4.2. Experiments

Frame Time Comparison

To compare our approach to 3DGS, we use the original viewpoints from the dataset used to generate the scenes. We average the frame times over multiple viewpoints for each scene. For a fair comparison, we disable Gaussian quantization in our renderer during these tests. Nevertheless, in a later experiment discussed in Section 4.3 we show the quantization does not have a profound effect on the overall runtime. We ran both the 3DGS forward renderer implementation and our stochastic Gaussian splatting implementation on the same machine on a variety of scenes.

We present our measurements in Figure 4.4, sorted horizontally by increasing scene size. We see that our render method at one sample per pixel always beats 3DGS, even by a factor of two for some larger scenes. The bicycle scene is the largest original scene tested at ≈ 6.1M Gaussians. The original approach renders it roughly twice as slow compared to the smaller train and counter scenes, both having ≈ 1M Gaussians. Meanwhile, our renderer at 1 SPP performs similarly in all MipNerf360 scenes. Finally, our method renders the *repeated_train* roughly twice as fast, at both 1 and 4 spp.

We can observe a positive correlation between the number of Gaussian in a scene and frame times of 3DGS. The absence of this correlation from the results of our method indicates that our method is less sensitive to the number of Gaussians in a scene compared to the original.

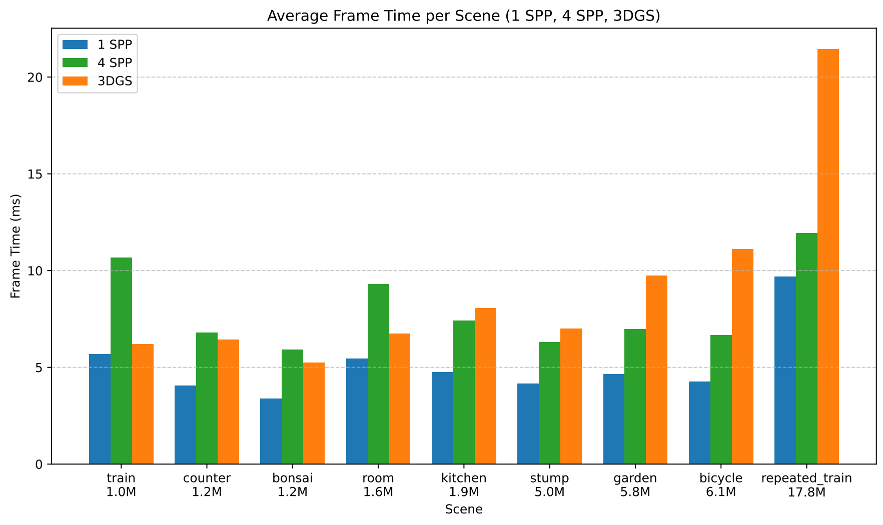

Figure 4.4: Comparison Performance between our (at 1 SPP & 4 SPP) and the 3DGS (original) implementation over multiple scenes, scenes order by increasing Gaussian count. Note how ours performs better at 1 sample per pixel (SPP) for all scenes compared to the original. Also note how ours lacks a clear correlation between frame time and overall gaussian count.

Frame Time Analysis

Further analysis also supports that our method is relatively unaffected by the number of Gaussians in a scene. Recall from Section 3.1 that our render pipeline has 4 primary stages, preprocessing, alias table construction, point cloud generation and postprocessing. As demonstrated in Figure 4.5, the frame time is primarily determined by the point cloud generation step. This step also has the greatest variance, primarily depending on the sum of the weight of all projected Gaussians in the view frustum. For the larger scenes we have tested, the point cloud generation step is actually cheaper. This is likely caused by Gaussians being more spread out, forming a skybox of sorts, rather than being densely concentrated on a small part of the scene.

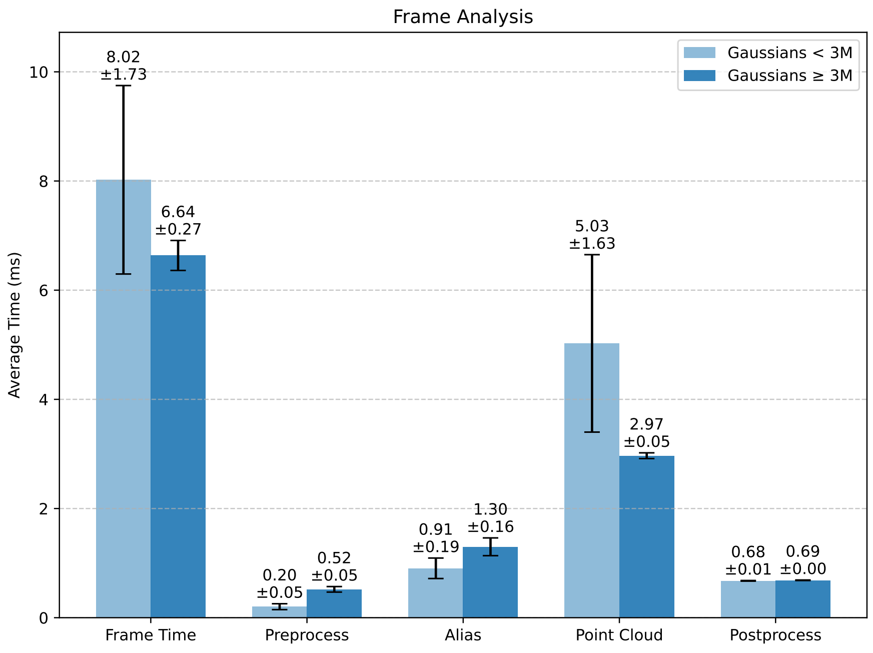

Figure 4.5: Breakdown of the frame time for 4 SPP aggregated over two groups of scenes, under and over 3M Gaussians –excluding the *repeated_train* scene– into the 4 primary stages of our render pipeline; preprocessing, alias table construction, point cloud generation and postprocessing. Furthermore, each bar displays an error in terms of 1 standard deviation. The frame time statistic is the time between frames measured on the CPU, and may be slightly larger than the sum of the other metrics, which are measured using the CUDA API.

Our method experiences moderate variance in frame times between scenes. Figure 4.7 illustrates how there is not necessarily a correlation between the number of Gaussian and the point count, and that it varies per camera angle. As the point count in the bicycle scene is consistently low despite featuring many Gaussians, evidence suggests that the placement or density of Gaussians in a scene has a larger influence on render times with our method. For instance, many Gaussians may be frustum culled, or further away in the bicycle scene, directly resulting in a relatively lower point cloud size.

The room scene has some notable outliers at camera indices 0 and 5 and 8. Camera 0 is closest to the sofa, which is a highly dense object in the scene. Camera positions 5 and 8 have the best view of the garden without having the top of scene in view, like in positions 4 and 7, resulting in fewer Gaussians in the frame. The same can be said for the train scene, where outliers at camera positions 2 and 9 are primarily caused by the train –the densest part of the scene– taking up a majority of the image or being very close respectively.

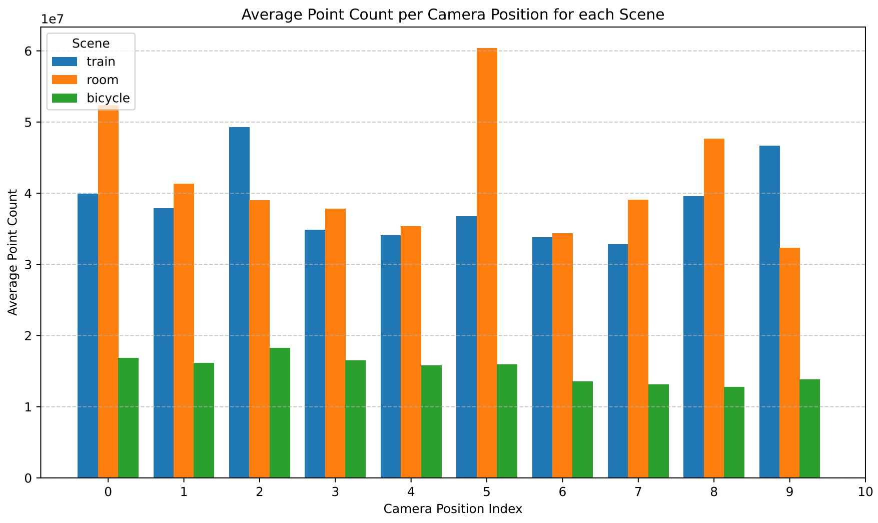

Figure 4.6: Measured point count at 1 SPP at the first 10 camera positions from the data set for the train scene (1.0M Gaussians), the room scene (1.6M Gaussians) and the bicycle scene (6.1M Gaussians). Note how there is some notable variance within each scene, and that the bicycle scene has a lower point count, despite containing many more Gaussians than the others.

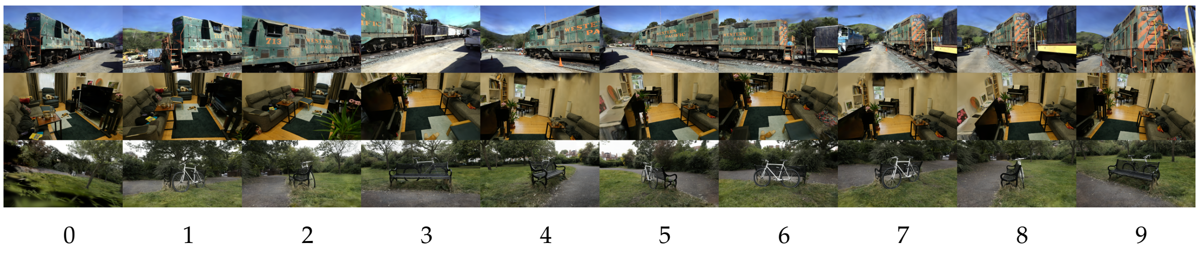

Figure 4.7: The camera positions per scene. Each row is one scene (train, room, bicycle), each column is one camera angle $(0, 1, \ldots 9)$

Frame Times Over Distance

The performance of the original 3DGS implementation suffers from overdraw. Larger scenes, or scenes composited from multiple Gaussian optimized objects naturally exhibit more overlapping Gaussians. To simulate this, we compare the performance of multiple scenes at increasing viewpoint distances from the origin, while facing the origin. This slowly increases the number of Gaussians within the frustum, and Gaussians that overlap per pixel, thus increasing overdraw. An example of what these viewpoints look like can be seen in Figure 4.9.

For all scenes, at 256 units removed from the origin, all the Gaussians are in view, but not yet culled by the far plane.

In Figure 4.8, we present our measurements. At close distances to the origin, the total number of Gaussians in view is limited, and the original Gaussian splatting render method renders faster. However, at further distances from the origin, we notice that our method has much lower frame times. The frame times of our method primarily scale with point cloud size, which is much larger when many Gaussians are viewed up close.

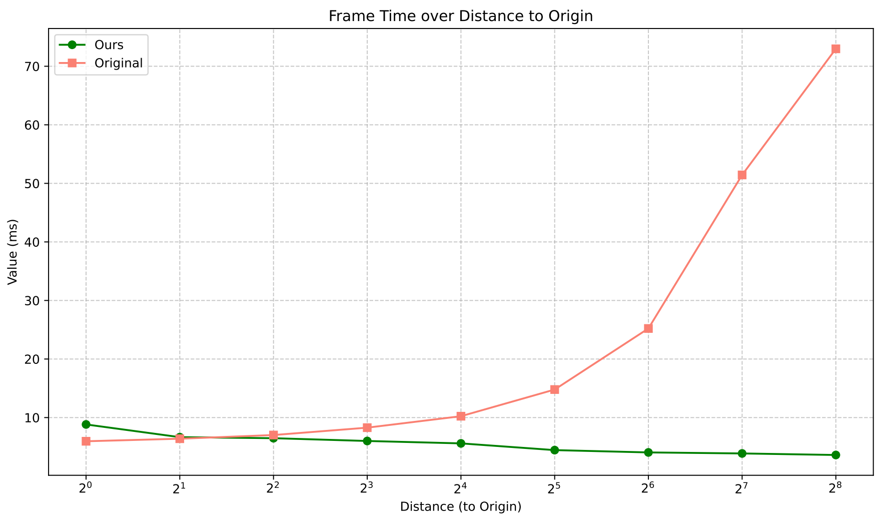

Figure 4.8: Comparison frame time (ms) between our approach at 4 SPP and 3DGS at different viewpoint distances from the origin, with the viewpoint facing the origin. Frame time is averaged over the same scenes as in figure 4.4. Note how ours performs better at larger distances.

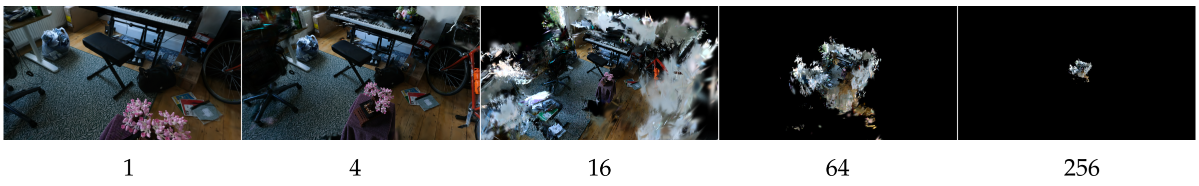

Figure 4.9: Example of viewpoints used to compare ours and the original implementation at different distances. From left to right, the image is taken at 1, 4, 16, 64, and 256 unit distance from the origin, all pointing towards the origin.

StochasticSplats

Since no source code has been published, we only provide a limited comparison between our approach and StochasticSplats [10]. We have additionally ran our implementation on the same scenes from the MipNerf360 dataset as StochasticSplats, and averaged the timings over all viewpoints and frames. Compared to their reported NVIDIA RTX 3090 statistics, our frame times on an NVIDIA RTX 3080 are higher for lower samples per pixel, but competitive for four and eight. Some increase in render time is expected due to differences in hardware [4]. With this in mind, our method is within the same ballpark as StochasticSplats. Overall it is hard to judge which method scales better to larger scenes. It may be that the scenes used in the evaluation of StochasticSplats were less densely populated by Gaussians than the scenes we used.

Table 4.1: Comparison of runtime performance. StochasticSplats ran on an RTX 3090 compared to our method running on an RTX 3080.

SPP	RTX 3090 (Paper) [ms]	RTX 3080 (Ours) [ms]	Ratio (Ours/StochasticSplats)
1 SPP	3.25	4.39	1.35
2 SPP	4.18	5.24	1.25
4 SPP	6.42	7.05	1.10
8 SPP	15.31	13.79	0.90

4.3. Optimizations

To test the effectiveness of our optimizations, we compare the performance of the implementation with them disabled and enabled. We ran all our tests on same scenes as before –the MipNerf360 scenes, excluding the *repeated_train* scene– and hardware, at 4 SPP. The base case for all experiments had quantization, jittered sampling, and alias table compaction enabled, and alias table grouping and deferred spherical harmonics disabled, unless specified otherwise.

Quantized Gaussians

The quantization is used to compress the memory size of the Gaussians. As seen in Figure 4.10, this has a meaningful positive impact on the preprocessing time of the Gaussians, and a small effect on the point cloud step. However, the primary use for quantization is to increase the number of Gaussians that can fit into VRAM, enabling us to render larger scenes.

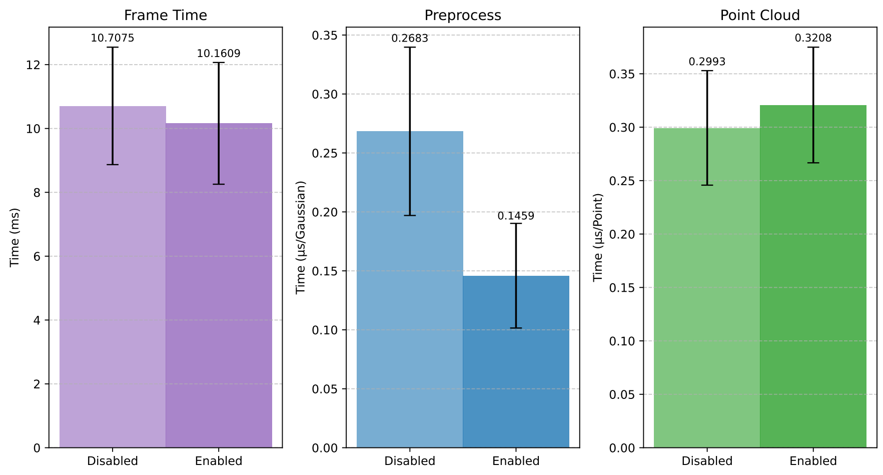

Figure 4.10: Comparison of the effect of quantization disabled (left), and enabled (right). Results are averaged over all view points on the original eight MipNerf360 scenes.

Jittered Sampling

We jitter the input into the alias table to aim for higher memory read coherency for threads within a warp. As seen in Figure 4.11, this has an extremely significant positive impact on render times, reducing the point cloud generation time per frame by ≈ 4 times on average.

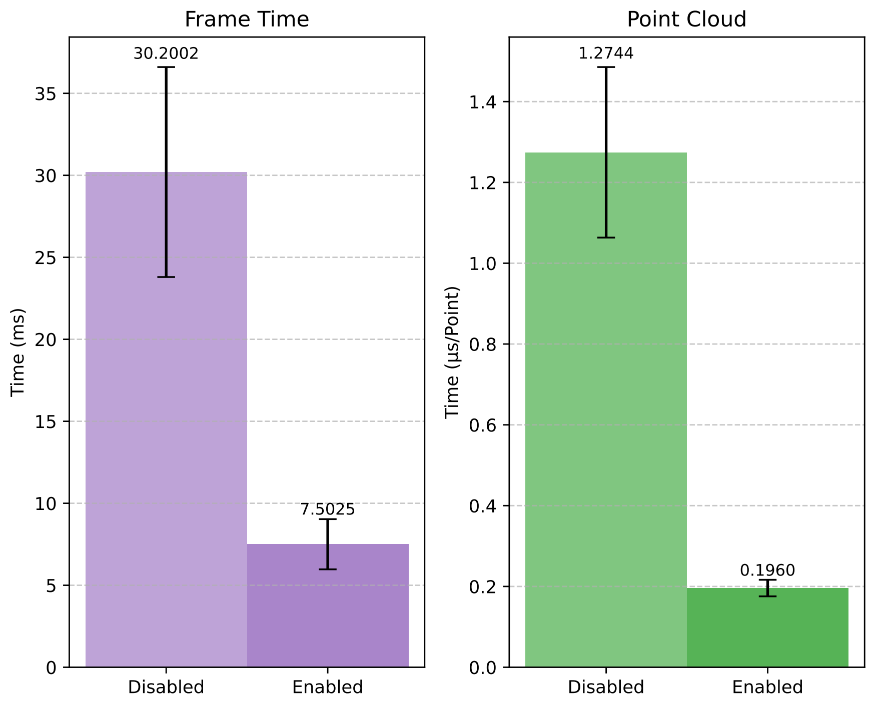

Figure 4.11: Comparison of the effect of jittered sampling disabled (left), and enabled (right). Results are averaged over all view points on the original eight MipNerf360 scenes.

Alias Table Compaction

We use stream compaction to remove all Gaussians that are frustum culled from the alias table construction step. Depending on the viewpoint, this may greatly decrease the time it takes to build the alias table, at the cost of an extra memory read required to map the alias table result back to the right Gaussian. Figure 4.12 shows this. Without alias compaction, the point cloud generation times are slightly lower, at the cost of a more expensive alias table construction step. Unless a majority of Gaussians are expected to always be within the frustum, this method is effective at reducing frame times, especially in larger scenes.

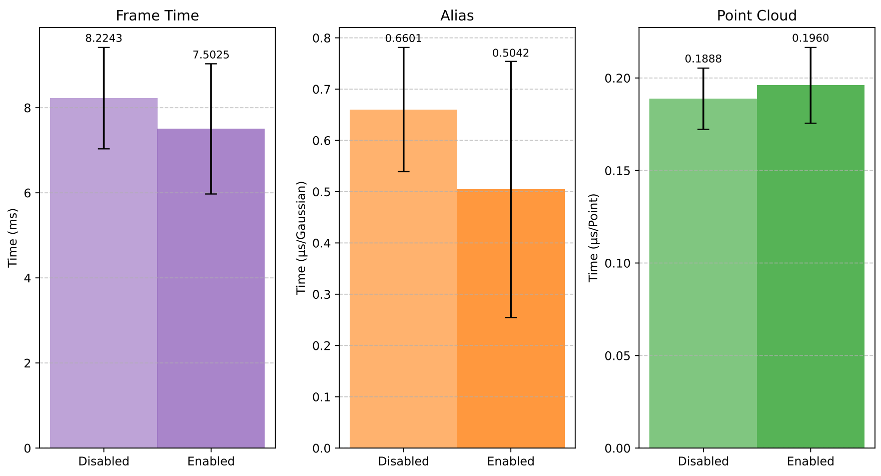

Figure 4.12: Comparison of the effect of alias compaction disabled (left), and enabled (right). Results are averaged over all view points on the original eight MipNerf360 scenes.

Alias Table Grouping

Besides compaction, we group weights during alias table construction. In all prior testing, the group size K has been exactly one, meaning this step was omitted. As seen in Figure 4.13, group size $K = 4$ results in slightly slower frame times. This is primarily due to the additional inverse CDF sampling step per point, which involves a linear search of size $K = 4$. While the overhead of grouping the weights is lower than the time saved by constructing an alias table at a quarter of the original size, the point cloud processing step outweighs the benefits for smaller scenes.

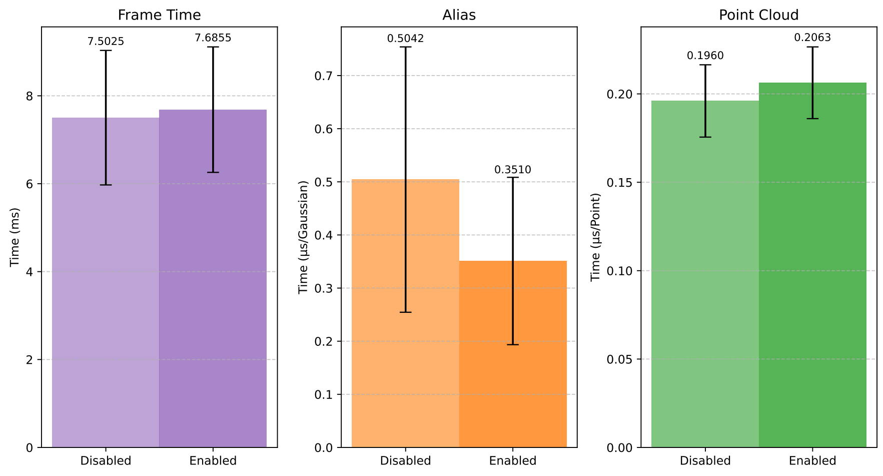

Figure 4.13: Comparison of the effect of alias grouping disabled (left), and enabled (right). Results are averaged over all view points on the original eight MipNerf360 scenes.

At a larger number of Gaussians, such as in the *repeated_train* scene, the technique is more effective. Figure 4.14 shows a similar increase in point cloud splatting cost, but a larger decrease in alias table construction times. This indicates that this technique is useful to render massive scenes, by cutting down on the overhead associated with rebuilding the alias table every frame.

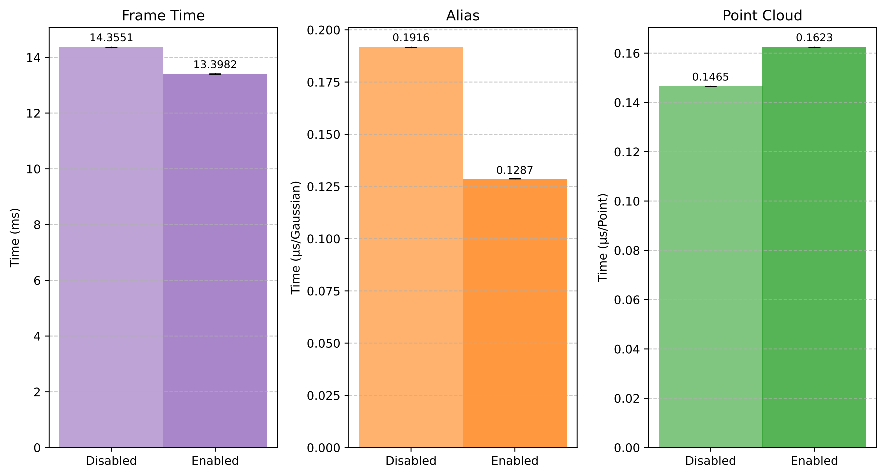

Figure 4.14: Comparison of the effect of alias grouping disabled (left), and enabled (right). Results are averaged over all view points on the *repeated_train* scene.

Deferred Spherical Harmonics

Deferred spherical harmonics postpone the computation of colors to the post processing step. As seen in Figure 4.15, on the original scenes optimized by 3DGS, enabling deferred spherical harmonics results in worse frame times. As these scenes have fewer Gaussians than the amount of pixels in the render passes, the more expensive postprocessing stage outweighs the cheaper preprocessing stage.

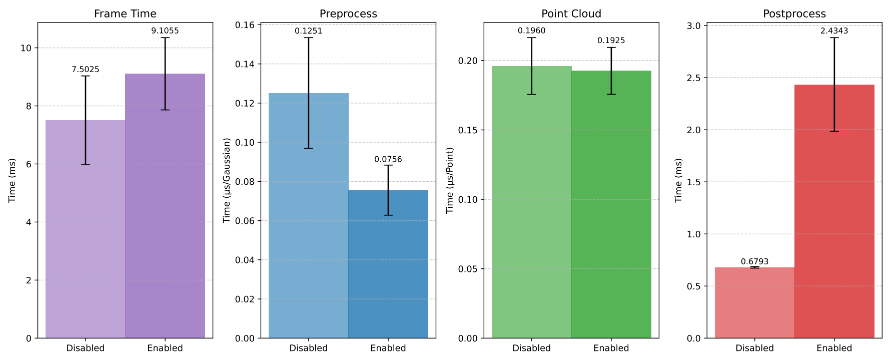

Figure 4.15: Comparison of the effect of deferred sh disabled (left), and enabled (right). Results are averaged over all view points on the original eight MipNerf360 scenes.

However, when many Gaussians are in view, for instance in the *repeated_train* scene, deferred spherical harmonics helps to reduce the preprocess time significantly. For very large scenes, this approach could therefore be a viable optimization technique.

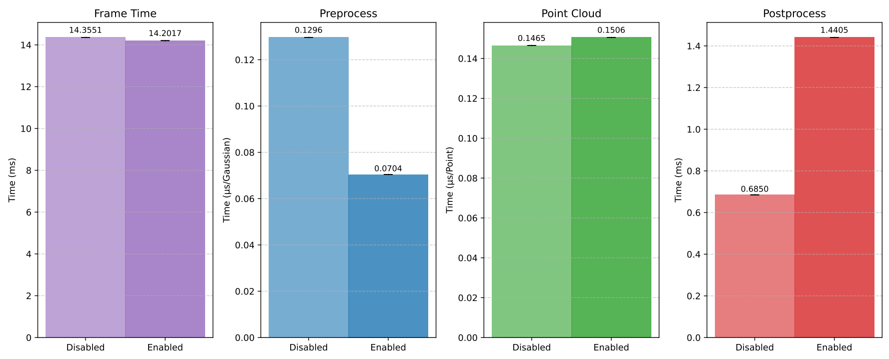

Figure 4.16: Comparison of the effect of deferred sh disabled (left), and enabled (right). Results are averaged over all view points on the *repeated_train* scene.

4.4. Limitations

As discussed in Section 3.5, the differences between stochastic transparency and our i.i.d. point rendering approach lead to minor discrepancies between the images produced by our stochastic splatter and those from the reference stochastic rasterizer. In Figure 4.17, we show the nature of noise between the two different render approaches. The resulting image of stochastic splatting exhibits more structured error, which supports our analysis of the differences between i.i.d. point cloud rendering and stochastic transparency in Section 3.5.

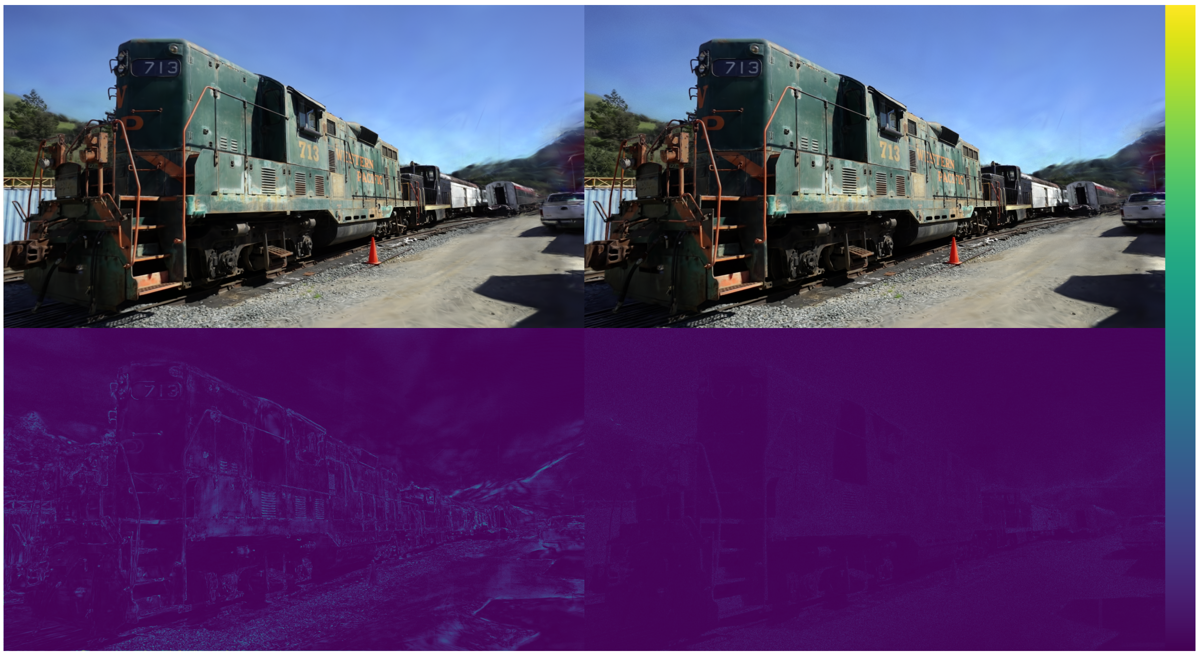
Stochastic Splatting (1028 SPP)
Stochastic Rasterizer (1028 SPP)
2.50 RMSE
1.73 RMSE
Figure 4.17: Quality comparison of both of our renderers in the train scene. The images depict the root mean squared error (RMSE) a resulting image of our stochastic splatting approach (left) and our stochastic rasterizer (right) compared to 3DGS.

Figure 4.18 displays the same camera viewpoint rendered using both stochastic splatting (top) and stochastic rasterization (bottom). In the red box, we notice that the black background bleeds through slightly more with the stochastic splatting approach. In the blue box, our method produces slightly brighter results. In the green box, some light blue Gaussians in particular appear to have higher opacity with our method. Finally, a slight blur is clearly perceivable in the orange box, with reflections and specular highlights being less pronounced around the train's lamps.

Therefore, our stochastic splatting method does not produce an image faithful to the original approach. Issues such as blur can be traced back to our rejection sampling approach, as explained in Section 3.5. Nonetheless, our images remain very similar to the ground truth overall. We expect that scenes optimized with our renderer in mind –by implementing a suitable backward renderer– will be even closer, e.g. by using additional Gaussians to cover an opaque surface.

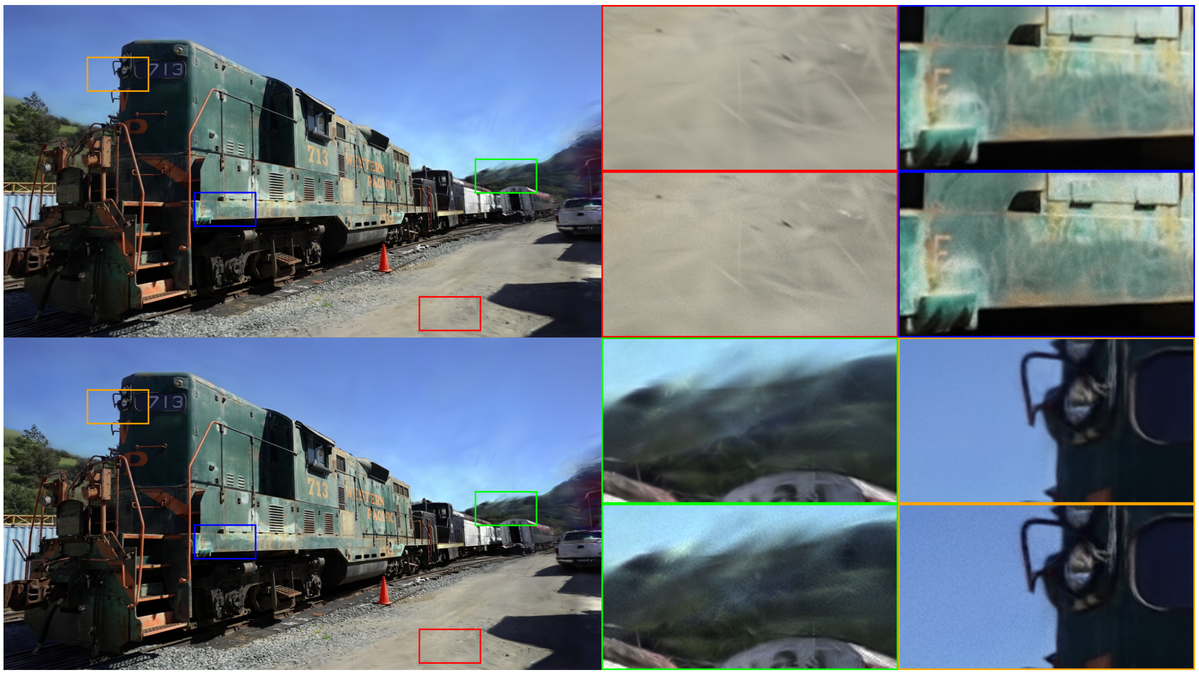

Figure 4.18: Comparison between a resulting image of our primary approach stochastic splatting (top) and our reference stochastic rasterization (bottom).

4.5. Discussion

Overall, results indicate that our method scales well to larger scenes. Unlike 3DGS, the frame time of our method does not directly scale with the number of Gaussians in a scene. Instead, the performance of our method is primarily dictated by the point cloud splatting pass. As its compute time depends primarily on the density of the scene close to the camera, our method is able to handle a vast amount of Gaussians on screen, unlike existing implementations like 3DGS.

Further experiments indicate that our optimizations are mostly effective. Jittered sampling, alias table compaction and compression using quantization of Gaussians all positively affect the runtime of our implementation, regardless of the scene rendered. Deferred spherical harmonics and alias table grouping negatively impact performance across all tested MipNerf360 scenes. However, results in the larger *repeated_train* scene indicate that these techniques may prove useful when scaling to extremely large scenes with a higher number of Gaussians.

5 Conclusion

Overall, our approach offers a competitive rendering method for Gaussian splatting, with promising scalability to larger scenes. While our method requires the reconstruction of an alias table every frame, stream compaction helps to reduce its computational burden. Nevertheless, the cost is relatively small in scenes with the number Gaussians only in the millions. Other optimizations such as alias table grouping and deferred spherical harmonics can support our method to render larger scenes.

Compared to the original 3DGS implementation, our renderer scales better to larger scenes. The frame time of our method is relatively stable with respect to the overall number of Gaussians in a scene. Moreover, our method handles overdraw with ease compared to 3DGS, meaning that dense clusters of Gaussians seen from further away hardly affect render times, unlike with 3DGS. Therefore, our proposed method is a suitable candidate to render vast Gaussian scenes.

Future Work
Our method would be further enhanced if scenes were optimized specifically for it, which requires a backward render pass. While our current sampling strategy leads to some deviations between images resulting from our approach and the references from 3DGS, we expect that the images rendered using these scenes would come closer to the ground truth. Moreover, this would enable sampling points directly from the Gaussian primitives, rather than their projected counterparts, resulting in a more accurate approximation of volume rendering. This approach would also elegantly resolve popping issues that plague many Gaussian splatting algorithms.

Furthermore, the render quality may be improved by implementing a more robust TAA solution, or by using a lightweight denoising algorithm designed for volumes.

References

[1] Jonathan T. Barron et al. "Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields". In: *CVPR* (2022).

[2] Markus Billeter, Ola Olsson, and Ulf Assarsson. "Efficient stream compaction on wide SIMD many-core architectures". In: Aug. 2009, pp. 159–166. DOI: `10.1145/1572769.1572795`.

[3] G. E. P. Box and Mervin E. Muller. "A Note on the Generation of Random Normal Deviates". In: *The Annals of Mathematical Statistics* 29.2 (1958), pp. 610–611. DOI: `10.1214/aoms/1177706645`. URL: `https://doi.org/10.1214/aoms/1177706645`.

[4] NVIDIA Corporation. *NVIDIA Ampere GA102 GPU Architecture Whitepaper*. 2020. URL: `https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf`.

[5] Eric Enderton et al. "Stochastic transparency". In: *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '10. Washington, D.C.: Association for Computing Machinery, 2010, pp. 157–164. ISBN: 9781605589398. DOI: `10.1145/1730804.1730830`. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/1730804.1730830`.

[6] James E Gentle. *Computational statistics*. Springer 2009, 2009. URL: `https://catalogue.library.cern/literature/g749k-13t02`.

[7] Florian Hahlbohm et al. "Efficient Perspective-Correct 3D Gaussian Splatting Using Hybrid Transparency". In: *Computer Graphics Forum* n/a.n/a (), e70014. DOI: `https://doi.org/10.1111/cgf.70014`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.70014`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.70014`.

[8] Mark Jarzynski and Marc Olano. "Hash Functions for GPU Rendering". In: *Journal of Computer Graphics Techniques (JCGT)* 9.3 (Oct. 2020), pp. 20–38. ISSN: 2331-7418. URL: `http://jcgt.org/published/0009/03/02/`.

[9] Bernhard Kerbl et al. "3D Gaussian Splatting for Real-Time Radiance Field Rendering". In: *ACM Transactions on Graphics* 42.4 (July 2023). URL: `https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/`.

[10] Shakiba Kheradmand et al. *StochasticSplats: Stochastic Rasterization for Sorting-Free 3D Gaussian Splatting*. 2025. arXiv: `2503.24366 [cs.CV]`. URL: `https://arxiv.org/abs/2503.24366`.

[11] Hans-Peter Lehmann, Lorenz Hübschle-Schneider, and Peter Sanders. *Weighted Random Sampling on GPUs*. 2021. eprint: `2106.12270`.

[12] Alexander Mai et al. *EVER: Exact Volumetric Ellipsoid Rendering for Real-time View Synthesis*. 2024. arXiv: `2410.01804 [cs.CV]`. URL: `https://arxiv.org/abs/2410.01804`.

[13] Marilena Maule et al. "Hybrid transparency". In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '13. Orlando, Florida: Association for Computing Machinery, 2013, pp. 103–118. ISBN: 9781450319560. DOI: `10.1145/2448196.2448212`. URL: `https://doi.org/10.1145/2448196.2448212`.

[14] Ben Mildenhall et al. "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis". In: *ECCV*. 2020.

[15] Claus Müller. *Spherical Harmonics*. Vol. 17. Lecture Notes in Mathematics. Springer Berlin, Heidelberg, 1966. ISBN: 978-3-540-03600-5. DOI: `10.1007/BFb0094775`.

[16] Cedrick Münstermann et al. "Moment-Based Order-Independent Transparency". In: *Proc. ACM Comput. Graph. Interact. Tech.* 1.1 (July 2018). DOI: `10.1145/3203206`. URL: `https://doi-org.tudelft.idm.oclc.org/10.1145/3203206`.

[17] *Open-sourcing .SPZ: it's .JPG for 3D Gaussian splats.* https://scaniverse.com/news/spz-gaussian-splat-open-source-file-format. Accessed: 29 April 2025. 2024.

[18] Alexandru-Lucian Petrescu et al. "Analyzing Deferred Rendering Techniques". In: *Control Engineering and Applied Informatics* 18 (Mar. 2016), pp. 30–41.

[19] Thomas Porter and Tom Duff. "Compositing digital images". In: *SIGGRAPH Comput. Graph.* 18.3 (Jan. 1984), pp. 253–259. ISSN: 0097-8930. DOI: 10.1145/964965.808606. URL: https://doi.org/10.1145/964965.808606.

[20] Ravi Ramamoorthi et al. "A theory of monte carlo visibility sampling". In: *ACM Trans. Graph.* 31.5 (Sept. 2012). ISSN: 0730-0301. DOI: 10.1145/2231816.2231819. URL: https://doi.org/10.1145/2231816.2231819.

[21] Marc B. Reynolds. *Quaternion Quantization – Part 1.* https://marc-b-reynolds.github.io/quaternions/2017/05/02/QuatQuantPart1.html. Accessed: 29 April 2025. 2017.

[22] Marco Salvi and Karthik Vaidyanathan. "Multi-layer alpha blending". In: *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '14. San Francisco, California: Association for Computing Machinery, 2014, pp. 151–158. ISBN: 9781450327176. DOI: 10.1145/2556700.2556705. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/2556700.2556705.

[23] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. "Rendering Point Clouds with Compute Shaders and Vertex Order Optimization". In: *Computer Graphics Forum* 40.4 (2021), pp. 115–126. DOI: https://doi.org/10.1111/cgf.14345. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14345. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14345.

[24] Richard Tucker and Noah Snavely. "Single-view View Synthesis with Multiplane Images". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2020.

[25] Alastair J. Walker. "An Efficient Method for Generating Discrete Random Variables with General Distributions". In: *ACM Trans. Math. Softw.* 3.3 (Sept. 1977), pp. 253–256. ISSN: 0098-3500. DOI: 10.1145/355744.355749. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/355744.355749.

[26] Chris Wyman. "Exploring and Expanding the Contiuum of OIT Algorithms". In: *ACM/EG Symposium on High Performance Graphics (HPG)*. June 2016, pp. 1–11. DOI: 10.2312/hpg.20161187.

[27] Lei Yang, Shiqiu Liu, and Marco Salvi. "A Survey of Temporal Antialiasing Techniques". In: *Computer Graphics Forum* 39.2 (July 2020), pp. 607–621. DOI: 10.1111/cgf.14018.

[28] Ce Zhu and Shuai Li. "Depth Image Based View Synthesis: New Insights and Perspectives on Hole Generation and Filling". In: *IEEE Transactions on Broadcasting* 62.1 (2016), pp. 82–93. DOI: 10.1109/TBC.2015.2475697.

[29] M. Zwicker et al. "EWA splatting". In: *IEEE Transactions on Visualization and Computer Graphics* 8.3 (2002), pp. 223–238. DOI: 10.1109/TVCG.2002.1021576.