Gaussian Point Splatting

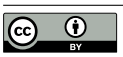
JORIS RIJSDIJK, Delft University of Technology, The Netherlands
CHRISTOPH PETERS, Delft University of Technology, The Netherlands
MICHAEL WEINMANN, Delft University of Technology, The Netherlands
RICARDO MARROQUIM, Delft University of Technology, The Netherlands

Fig. 1. Large-scale scene composed of four copies of a village, rendered interactively using a total of 425 M Gaussians. Frame times on an NVIDIA RTX 4070 Ti SUPER at 1920×1080 are 65.6 ms, 37.5 ms (red) and 24.3 ms (yellow) for the three views, respectively. We achieve this high throughput with a stochastic approach (the images shown here are converged) but avoid approximations, level-of-detail mechanisms and complex data structures.

We propose Gaussian point splatting, a stochastic method to render Gaussian splats that scales extremely well to scenes with many Gaussians. Our core idea is to sample pixel-sized, opaque points from the Gaussians and to splat them to a framebuffer using 64-bit atomics. Through parallel programming primitives, we achieve an even distribution of the workload across millions of threads. Since these threads splat points independently, multiple points may splat to the same pixel. That makes it non-trivial to determine how many points should be splatted for a Gaussian or how they should be distributed to achieve the desired opacity. We successfully formalize and solve these problems, thus keeping our renders faithful to the original Gaussian splatting. To further accelerate our method, we employ hierarchical frustum and occlusion culling. Our method renders hundreds of millions of Gaussians in real time. The only differences compared to the original Gaussian splatting are slight noise and differences in aliasing.

CCS Concepts: • **Computing methodologies** → **Point-based models**; *Visibility*; *Massively parallel algorithms*; • **Mathematics of computing** → *Stochastic processes.*

Authors' Contact Information: Joris Rijsdijk, Delft University of Technology, Delft, The Netherlands, j.a.rijsdijk@tudelft.nl; Christoph Peters, Delft University of Technology, Delft, The Netherlands, c.j.p.peters@tudelft.nl; Michael Weinmann, Delft University of Technology, Delft, The Netherlands, m.weinmann@tudelft.nl; Ricardo Marroquim, Delft University of Technology, Delft, The Netherlands, r.marroquim@tudelft.nl.



ACM 1557-7368/2026/7-ART45
https://doi.org/10.1145/3811272

ACM Reference Format:
Joris Rijsdijk, Christoph Peters, Michael Weinmann, and Ricardo Marroquim. 2026. Gaussian Point Splatting. *ACM Trans. Graph.* 45, 4, Article 45 (July 2026), 11 pages. https://doi.org/10.1145/3811272

1 Introduction

Gaussian splatting [Kerbl et al. 2023] has revolutionized novel view synthesis. Its scene representation is more efficient to render than neural radiance fields (NeRFs) [Mildenhall et al. 2020], suitable for inverse rendering, easy to exchange, and expressive enough to capture a wide variety of scenes. The scale, complexity and detail of the represented scenes are limited only by the number of Gaussians that one can afford to handle. In principle, entire villages can be captured while preserving small details, but doing so requires hundreds of millions of Gaussians (Fig. 1).

The number of Gaussians that can be rendered in real time is limited by the amount of available VRAM, but existing software rasterizers will drop to non-interactive frame rates long before this bound is reached. Hierarchies of Gaussians can be used for level of detail, reducing the number of Gaussians that need to be rendered each frame, but they require additional preprocessing and may deteriorate the quality of the original asset [Kerbl et al. 2024; Yang et al. 2025]. When attempting to rasterize large numbers of Gaussians directly, the software rasterization approach becomes inefficient [Schütz et al. 2025]: It requires sorting, which scales sublinearly. Every Gaussian in the view frustum, no matter how small, will be assigned to at least one tile in screen space, taking up resources

there. Thread groups for tiles that contain particularly many Gaussians will run much longer than others, thus dominating the overall frame time. Occlusion culling can only be used at a fairly late stage in the pipeline or not at all, since all Gaussians are transparent.

We propose Gaussian point splatting, a fundamentally different approach to render Gaussian splats, that scales much more favorably to scenes with many Gaussians. Our method stochastically samples pixel-sized points from Gaussians and renders them with depth-buffering using 64-bit atomics [Schütz et al. 2021] (Sec. 3.1). The number of points sampled from a Gaussian corresponds to its size in screen space. We use parallel compute primitives to assign the point splatting work to threads in such a way that each thread only handles a small, fixed number of points (Sec. 3.2). This independent sampling is key to making our method massively parallel. However, it makes it non-trivial to faithfully reproduce the opacity of Gaussians, since multiple points sampled for a single Gaussian may get splatted to the same pixel. That reduces the effective opacity. We compensate for this effect exactly by increasing the overall point count and the density near the center of Gaussians (Secs. 3.3 and 3.4). As a result, our method renders Gaussian splatting scenes in an unbiased fashion, except for differences in how aliasing manifests compared to the original Gaussian splatting.

The computational burden of our method is output sensitive, in that smaller Gaussians will splat fewer points. Nonetheless, a lot of work may be spent on splatting points that will eventually be occluded by closer points. To mitigate this, we exploit that our method also produces a noisy depth buffer and use it for occlusion culling (Sec. 3.5). Our rendering system (Sec. 3.6) achieves consistent real-time frame rates on extremely large scenes, where existing methods drop to non-interactive frame rates or fail entirely. On small scenes, its performance is similar to existing methods (Sec. 4.2). In spite of the different approach to rendering, its output is faithful to the original (except for differences in aliasing) and noise is moderate at the sample counts that we use (Sec. 4.1). Thus, we open the door to Gaussian splatting for scenes of unprecedented scale and detail. Full source code is available as supplemental material.

2 Related Work

Implicit neural scene representations such as neural radiance fields (NeRFs) and their extensions [Barron et al. 2022; Mildenhall et al. 2020; Müller et al. 2022], that model scenes in terms of continuous volumetric fields, allow for remarkable reconstruction quality and photorealism in synthesized views. However, their reliance on dense ray sampling and repeated neural network evaluations results in high computational cost, slow rendering, and limited support for real-time interaction or scene editing. These limitations have been addressed by 3D Gaussian splatting [Kerbl et al. 2023]. In this framework, a set of optimized 3D anisotropic Gaussian primitives is rendered using a tile-based software rasterizer, thereby enabling real-time rendering while maintaining photorealistic quality.

Extensions of Gaussian splatting focus on numerous aspects, including the handling of appearance changes [Dahmani et al. 2025; Kulhanek et al. 2024; Zhang et al. 2025b] the embedding of semantics [Cen et al. 2025; Li et al. 2024; Qin et al. 2024], multi-spectral and hyper-spectral scene representation [Sinha et al. 2025; Thirgood et al. 2025] as well as dynamic scenes [Kim et al. 2024]. Stuart et al. [2025] turn Gaussians into a point cloud, similar to our method. However, their point clouds are static and rendering them introduces substantial bias. Most closely related to our work are extensions that focus on improvements regarding render times and scalability, as discussed in the following.

To improve rendering efficiency and scalability while preserving the high visual fidelity, several works focus on accelerating the rasterization pipeline. Examples are improvements of tile-based rendering by tightening screen-space bounds and reducing overdraw through more precise Gaussian–tile intersection tests [Hanson et al. 2025a; Lee et al. 2024; Schütz et al. 2025] or adaptive screen-space extents [Wang et al. 2024] as well as hybrid preprocessing to reduce overdraw and improve GPU utilization [Huang et al. 2025]. Further hardware-oriented rasterization techniques [Wang et al. 2025] leverage axis-oriented rasterization to pre-compute and reuse shared terms along both screen-space axes. Others focus on system-level optimizations of differentiable Gaussian rasterization, improving memory access patterns, kernel fusion, and workload scheduling to better exploit GPU parallelism [Feng et al. 2025]. Tile grouping and scheduling strategies further reduce redundant sorting and improve load balancing by sharing computation across spatially adjacent tiles [Jo and Park 2025; Schütz et al. 2025].

Another strategy to improve rendering efficiency is to reduce the number of active primitives through pruning and to use compact representations. Sensitivity-based pruning, e.g., according to a second-order approximation of the reconstruction error on the training views with respect to the spatial parameters of each Gaussian, allows to remove Gaussians with a low impact on reconstruction quality [Hanson et al. 2025b], thereby eliminating a large fraction of primitives without compromising visual fidelity. Prioritizing Gaussians with high contributions in comparison to Gaussians with low contributions has also been explored by others such as contribution-aware rasterization approaches [Huang et al. 2025]. Further extensions incorporate temporal coherence and motion-aware grouping to stabilize pruning and reduce redundancy in dynamic scenes [Tu et al. 2025]. Beyond pruning, several methods focus on compressing Gaussian parameters to alleviate memory bandwidth and storage constraints. Respective approaches include vector quantization and quantization-aware training [Niedermayr et al. 2024], the quantization of per-Gaussian attributes and opacity via latent quantization frameworks [Girish et al. 2025] as well as combinations of guided pruning of low-significance Gaussians based on global significance scores with the reduction of the complexity of spherical harmonics through truncation or knowledge distillation [Chen et al. 2025; Fan et al. 2024].

Efficient handling of transparency, depth ordering, and visibility is a further strategy for accelerating Gaussian splatting, as semi-transparent Gaussians require correct compositing to avoid visual artifacts. Since strict sorting and sequential alpha blending as applied in standard Gaussian splatting [Kerbl et al. 2023] scales poorly with millions of semi-transparent primitives, several works explored approximate and hybrid strategies to reduce or bypass full sorting. Hierarchical sorting of Gaussians [Lee et al. 2024] uses an initial approximate sorting of Gaussians into chunks sorted by their approximate depth relative to the view, and a subsequent accurate

sorting per chunk when a chunk becomes active for rasterization. Hybrid transparency blends only a limited number of closest Gaussians in depth order while accumulating distant contributions in an order-independent approximate manner [Hahlbohm et al. 2025]. Hou et al. [2025] use weighted-sum formulations to approximate alpha blending through order-independent accumulation, inspired by weighted blended order-independent transparency [McGuire and Bavoil 2013], thereby removing the need for sorting. Duplex-GS [Liu et al. 2025] integrates order-independent transparency (OIT) directly with splatting through a proxy-guided weighted blending technique to bypass expensive radix sorting almost entirely, yielding substantial speedups in dense scenes.

While these approaches substantially reduce sorting overhead, they may introduce temporal or view-dependent artifacts. This is addressed by view-consistent hierarchical rasterization and resorting [Radl et al. 2024] and combining hierarchical depth ordering with tile-based rasterization and analytic Gaussian evaluation to further improve rendering quality [Steiner et al. 2025]. Furthermore, stochastic formulations relax deterministic ordering entirely. Exemplary approaches include stochastic ray tracing of Gaussians [Sun et al. 2025] and stochastic sampling and accumulation directly in the splatting domain to approximate correct alpha compositing through randomized evaluation and temporal averaging [Kheradmand et al. 2025]. Further work replaces the conventional sorting process with a neural sorting in terms of neural-network-based predictions of order-independent blending weights [Wang et al. 2025].

Additional efficiency gains can be achieved by occlusion-aware rendering, where proxy geometry or coarse depth representations [Gao et al. 2025] and hierarchical depth tests [Zhang et al. 2025a] are used to cull occluded Gaussians. Additional strategies address redundancy introduced by adaptive densification in the original formulation [Kerbl et al. 2023], employing opacity-aware pruning and structured cross-section-oriented splitting or cloning operations to reduce overlapping Gaussians [Zheng et al. 2025].

Level-of-detail (LoD) approaches address scalability to large scenes by adapting representation complexity to factors like the complexity of local scene structures, the distance of scene structures to the camera, or the camera's viewing frustum. Scene partitioning into spatially adjacent blocks based on divide-and-conquer strategies enables parallel training and selective activation of Gaussians [Liu et al. 2024]. Artifacts occurring in the vicinity of the transition between adjacent cells for too few overlapping cameras can further be mitigated based on progressive data partitioning based on assigning training views and point clouds to different cells [Lin et al. 2024], as well as further extensions for recursive load balancing across blocks [Chen and Lee 2024]. Furthermore, content-aware scene partitioning and visibility-aware block optimization [Wu et al. 2025] allow the embedding of occlusion relationships between blocks and reduce supervision mismatch during independent block optimization. Approaches tailored for LoD rendering include multi-resolution 3D Gaussian representations with different levels of Gaussians at varying resolutions [Cui et al. 2024] or tree-based hierarchical representations [Kerbl et al. 2024; Ren et al. 2025; Yang et al. 2025; Zhu et al. 2026]. The latter organizes Gaussians into tree structures based on local geometric and volumetric properties and dynamically selects levels or cuts through the tree based on the view to balance rendering efficiency and visual fidelity.

Thus, most existing methods depend on pruning or LoD approaches to achieve real-time performance when the number of Gaussians in a scene approaches the hardware memory limit. When possible, it is preferable to render scenes without LoD to avoid the required additional processing and loss of detail. In contrast, our method is the first to interactively render such extreme numbers in full, without any such approximations. While LoD combined with out-of-core rendering may facilitate even larger scenes with our method, we leave this to future work.

3 Gaussian Point Splatting

We now describe our method for rendering Gaussian splatting scenes. It has commonalities with methods based on stochastic transparency [Enderton et al. 2010] in that Gaussians splat points to each pixel with a probability matching their opacity. However, stochastic transparency accomplishes this by generating candidate fragments per pixel for each Gaussian that covers the pixel and then rejecting them stochastically. We instead sample points with the desired probability and splat them directly to the framebuffer using the method of Schütz et al. [2021] (Sec. 3.1). This way, we avoid overhead for rejected fragments and achieve massive parallelism by distributing this work evenly across millions of threads (Sec. 3.2). Since points of a Gaussian are being sampled independently, they may collide on the same pixel and thus fail to contribute to the opacity of the Gaussian. We model this issue using Poisson distributions and devise a way to compensate for it exactly, thus achieving the desired effective opacity (Secs. 3.3 and 3.4). Finally, we further accelerate the method using frustum and occlusion culling (Sec. 3.5) and summarize how our renderer works (Sec. 3.6).

3.1 Point Splatting

Our core idea is to splat opaque, pixel-sized points stochastically sampled from Gaussians instead of splatting Gaussians with alpha blending. Therefore, we need a massively parallel and efficient method for point splatting. We rely on the technique of Schütz et al. [2021], which we briefly summarize next. The framebuffer consists of one 64-bit unsigned integer per pixel. The most significant bits of each integer, in our case 28 bits, store the pixel depth. The remaining $3 \cdot 12$ least significant bits store an sRGB color. This way, we can handle sRGB values across the range $[0, 16)$ with $\frac{1}{256}$ increments, which is important since Gaussian splatting may use colors with brightness exceeding one. At the same time, we get sufficient precision for depth values using fixed-point quantization of the view-space depth between the near- and far-clipping planes.

To splat a point, we first convert it to such a 64-bit unsigned integer. Then, we compute which pixel the point falls into by rounding its screen-space coordinates. Finally, we use an atomic minimum operation to splat it to the framebuffer. If the depth of the new point is less than the currently stored depth, the stored depth and color will be overwritten. Otherwise, the framebuffer remains unchanged, since the point is occluded by a previously splatted closer point. In this way, we benefit from the fast atomic operations of modern GPU hardware [Schütz et al. 2021].

Due to its stochastic nature, our method produces noisy results. We will also see that a high resolution is beneficial for accuracy (Sec. 3.3). Therefore, we employ supersampling, i.e., we multiply the horizontal and vertical resolution of the framebuffer by a user-defined integer factor (typically 2×2 subpixels). Once all points have been splatted, we resolve the supersampled framebuffer: For each pixel, we take the arithmetic mean of the sRGB colors of its subpixels and store it in a framebuffer with 32-bit floats. We experimented with a Gaussian reconstruction filter instead of this box filter, but found that it introduces additional blur compared to the original Gaussian splatting. When the camera is static, we additionally use temporal accumulation such that noise diminishes with each frame. Our implementation includes basic temporal reprojection for denoising (Sec. 3.6), but we leave the application of more sophisticated spatiotemporal denoisers [Hofmann et al. 2021] for future work.

3.2 Work Distribution

To be efficient, the point splatting has to be massively parallel. We want millions of threads, each loading attributes of a single Gaussian and then sampling and splatting a single point. For each Gaussian, we can compute the intended point count based on its screen-space size and opacity in the current frame (Secs. 3.3 and 3.4). Based on this input, we need an efficient method for each thread to determine which Gaussian it should sample from. At first, we tried using massively-parallel construction of alias tables [Lehmann et al. 2022] each frame, which allows threads to sample a Gaussian index in time $O(1)$ once they have been constructed. However, we eventually developed a method that achieves even higher throughput and produces sorted Gaussian indices, which benefits cache coherence during the point splatting stage (Fig. 2).

As input, we receive an array of desired point counts $n_0, \dots, n_{G-1}$ for each of the $G \in \mathbb{N}$ Gaussians. We compute exclusive prefix sums $t_g := \sum_{i=0}^{g-1} n_i$ for this array (using Thrust). Then the threads with indices $t \in \{t_g, \dots, t_{g+1} - 1\}$ should sample points from Gaussian $g \in \{0, \dots, G-1\}$. Next, we want to reverse this mapping, i.e., construct an array mapping thread indices t to Gaussian indices g_t. We impose an upper bound T on how many points can be splatted in a single frame and zero-initialize an array $g_0, \dots, g_{T-1} := 0$. A scatter operation that sets $g_{t_g} := g$ whenever $n_g \neq 0$ assigns the correct value for the thread splatting the first point of Gaussian g. Finally, we perform an inclusive maximum scan operation on this array to make the index sequence monotonic, thus setting the remaining entries correctly: $g_t := \max\{g_0, \dots, g_t\}$.

At this point, thread t can read its Gaussian index from g_t. In this manner, we accomplish the work distribution task with two scan operations and a scatter operation, all of which achieve immense throughput on modern GPUs. The actual total number of points to be splatted $\sum_{i=0}^{G-1} n_i$ is a side product of the prefix sum and we spawn threads and adjust the range for the maximum scan accordingly. If it exceeds T, points will be missing, but that never happened in our experiments.

3.3 Opacity Correction

Now that we can splat points sampled from Gaussians in a massively parallel fashion, we use this mechanism to render Gaussian splatting scenes. In general, our implementation is designed to be as similar as possible to the original Gaussian splatting [Kerbl et al. 2023]. Thus, the conversion from a 3D Gaussian into a colored 2D Gaussian based on a perspective projection is exactly the same. In particular, we also add $\left(\begin{smallmatrix} 0.3 & 0 \\ 0 & 0.3 \end{smallmatrix}\right)$ to the screen-space covariance matrix. This enforces a minimal standard deviation of $\sqrt{0.3} \approx 0.55$ pixels along x and y, as 3DGS scenes are optimized with this in mind. Thin features such as the bike spokes in Fig. 6 may also disappear without it. Furthermore, we exactly reproduce the mechanisms for truncating Gaussians: If we sample a point outside the truncation range, we reject it.

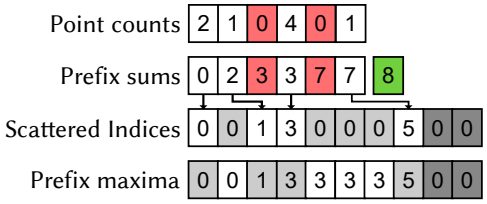


Fig. 2. The workload distribution pipeline illustrated for 6 Gaussians, $T = 10$, and 8 points in total being splatted. Cells colored red indicate a mask value of 0 such that their indices will not be scattered. The total number of points calculated as a byproduct of the exclusive prefix sum (green cell) is used to constrain the workload of the inclusive prefix max operation. Gray cells indicate that it has not been explicitly updated in that step.

To complete our method, we have to answer two questions: How do we determine the point count of a Gaussian, and how do we sample a point from it? The Gaussian is characterized by its symmetric, positive-definite covariance matrix $\Sigma \in \mathbb{R}^{2\times2}$, its mean $\mu \in \mathbb{R}^2$ (both given in screen-space coordinates) and the opacity at its mean $\alpha \in [0, 1]$. Its opacity at a screen-space coordinate $q \in \mathbb{R}^2$ is

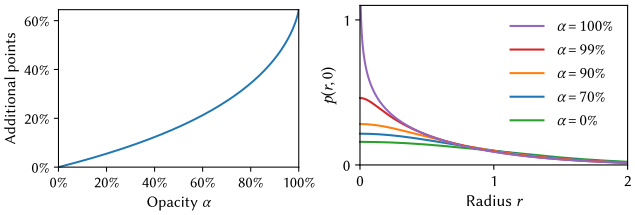
$$\alpha(q) := \alpha \exp\left(-\frac{1}{2}(q-\mu)^\mathsf{T}\Sigma^{-1}(q-\mu)\right).$$

Sampling points proportional to $\alpha(q)$ is easy, but (perhaps surprisingly) does not give us the desired result. Each thread samples points independently of other threads, which is key to the massive parallelism of our method. It is not at all unlikely that two threads sample two points for the same Gaussian that also splat to the same subpixel. Though, the outcome of splatting the color of this Gaussian twice is the same as splatting it once. Effectively, the second point has not contributed to the opacity of the Gaussian. We have to splat additional points to compensate for these collisions. Furthermore, points are more likely to collide in regions where the Gaussian has high opacity, i.e. near its mean. Thus, the effective opacity is diminished more in these regions. We have to adjust the sampled density, sampling more points near the mean to achieve the desired distribution for the opacity (Fig. 3).

To alleviate this problem, we fully embrace independent sampling. We choose the number of points n_g to be splatted for a Gaussian at random, using a Poisson distribution. Below, we determine its rate parameter $\lambda > 0$ from parameters of the Gaussian. Determining the point count like that and then taking independent samples simulates a Poisson point process [Streit 2010, Sec. 2.3]. Therefore, we know that the number of points splatted to any particular pixel will also be Poisson-distributed [Streit 2010, Sec. 2.4]. Let $p(q)$ denote the

(a) Without correction (b) With correction (c) Reference

Fig. 3. A black Gaussian on white background with opacity $\alpha = 100\%$. We render it with point splatting and show results of a single pass and converged results. Since splatted points may collide on the same pixel, sampling points from the Gaussian gives a too low opacity, especially at the mean. We splat additional points and concentrate their distribution towards the mean to correct for that. This way, our results match the rasterized reference.



(a) Point count increase (b) Corrected density

Fig. 4. Opacity correction increases the expected point count for a Gaussian by a factor of $\frac{\mathrm{Li}_2(\alpha)}{\alpha}$ (left). The density increases most in the middle of the Gaussian (right, shown here for $\Sigma = \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$ and $\mu = 0$). For $\alpha = 100\%$, the density at $r = 0$ approaches infinity, but still integrates to one. For $\alpha = 0\%$, the corrected density approaches the original Gaussian density.

corrected probability density with higher density around the mean that we use to sample points for the Gaussian. Let $A > 0$ be the screen-space area of a subpixel and let $q \in \mathbb{R}^2$ be the center of a subpixel. Then we can approximate the integral of the density over the subpixel footprint by $Ap(q)$ (Sec. 4.3 discusses the implications of this approximation). The probability of splatting exactly $k \in \mathbb{N}_0$ points to this pixel is given by a Poisson distribution:

$$\frac{(\lambda A p(q))^k}{k!} \exp(-\lambda A p(q)). \tag{1}$$

To obtain the desired effective opacity, we have to ensure that the probability of splatting $k = 0$ points is exactly $1 - \alpha(q)$:

$$\begin{aligned} &\frac{(\lambda A p(q))^0}{0!} \exp(-\lambda A p(q)) = 1 - \alpha(q) \\ \Leftrightarrow\ & p(q) = -\frac{1}{\lambda A} \ln(1 - \alpha(q)) \end{aligned} \tag{2}$$

This is the solution to our problem, although we still have to make it practical. The rate λ acts as a normalization factor for the probability density $p(q)$ and will be determined as such (Eq. 3). Then everything on the right-hand side of Eq. 2 is known. Fig. 4b shows how opacity correction increases the density of sampled points near the center of a Gaussian, especially when it has high opacity there.

3.4 Sampling Strategies

Two closely-related problems remain: We have to compute the normalization factor λ for the density p and we have to figure out how to sample points in proportion to p. These problems are tractable for the same reason that sampling of a 2D Gaussian is tractable: After an affine transform, the density is radially symmetric. Specifically, such a transform is $o := L^{-1}(q - \mu)$, where $L \in \mathbb{R}^{2\times 2}$ denotes a Cholesky decomposition $LL^\mathsf{T} = \Sigma$. With this substitution, we obtain

$$\alpha(q) = \alpha \exp\left(-\frac{o^\mathsf{T} o}{2}\right).$$

In Appendix A, we derive the following solution for the integral that gives the unnormalized cumulative distribution function (CDF) of p as function of radius $r := \|o\|$:

$$\begin{aligned} F_\alpha(r) &:= \int_{\|o\|<r} -\ln\left(1 - \alpha \exp\left(-\frac{o^\mathsf{T} o}{2}\right)\right) \mathrm{d}o \\ &= 2\pi \,\mathrm{Li}_2(\alpha) - 2\pi \,\mathrm{Li}_2\left(\alpha \exp\left(-\frac{r^2}{2}\right)\right), \end{aligned}$$

where $\mathrm{Li}_2(\alpha) := \int_0^\alpha -\frac{\ln(1-s)}{s}\,\mathrm{d}s$ denotes the dilogarithm.

With that at hand, we obtain a formula for the normalization factor of the probability density p:

$$\lambda = \int_{\mathbb{R}^2} -\frac{1}{A} \ln(1 - \alpha(q))\,\mathrm{d}q = \frac{1}{A}|L|F_\alpha(\infty) = \frac{1}{A} 2\pi\sqrt{|\Sigma|}\,\mathrm{Li}_2(\alpha). \tag{3}$$

This is also the expected number of points that we will splat for the Gaussian. If we had ignored the issue of colliding splats, we would instead have used an expected point count of

$$\frac{1}{A}\int_{\mathbb{R}^2} \alpha(q)\,\mathrm{d}q = \frac{1}{A} 2\pi\sqrt{|\Sigma|}\alpha.$$

Fig. 4a shows that compensating for collisions increases the point count by at most $\frac{\pi^2}{6} - 1 \approx 64.5\%$.

To convince ourselves that this additional work and our derivations are necessary, we briefly study the alternative: Suppose we have a Gaussian with opacity $\alpha = 100\%$ and sample points using the uncorrected density and point count, i.e. $\lambda A p(q) = \alpha(q)$. Then according to Eq. 1, its effective opacity at the mean μ, i.e. the probability to splat more than $k = 0$ points there, would be

$$1 - \exp(-\lambda A p(\mu)) = 1 - \exp(-\alpha(\mu)) = 1 - \exp(-1) \approx 63.2\%.$$

That is a strong deviation from the intended opacity of 100%, which demonstrates the necessity for opacity correction.

To sample from the density p, we first apply inverse CDF sampling to the normalized CDF, to map a uniform random number $u_0 \in [0, 1)$ to a radius r. Appendix B shows that that amounts to

$$r = F_\alpha^{-1}(F_\alpha(\infty)u_0) = \sqrt{-2\ln\left(\frac{1}{\alpha}\,\mathrm{Li}_2^{-1}\left((1 - u_0)\,\mathrm{Li}_2(\alpha)\right)\right)}.$$

Then we construct the sampled point q in analogy to the Box-Muller transform that is commonly used for sampling 2D Gaussians (with

a second uniform random number $u_1 \in [0, 1)$):

$$o := (r\cos(2\pi u_1), r\sin(2\pi u_1))^\mathsf{T}, \qquad q := Lo + \mu.$$

To implement these solutions, we need numerical fits for the dilogarithm and its inverse. For Li_2^{-1}, we use a minimax polynomial fit [Press et al. 2007, Sec. 5.13] of degree 10. For Li_2 that does not work well since the derivative is unbounded. We instead fit a polynomial of degree 7 to $\mathrm{Li}_2(\alpha) - (1-\alpha)\ln(1-\alpha)$ for $\alpha \in [0, 1]$. These fits achieve maximal absolute errors of $7.07 \cdot 10^{-5}$ and $7.27 \cdot 10^{-5}$, respectively.

To sample the Poisson-distributed point count n_g, we use the $\tilde{Q}_{N3}$ approximation of Giles [2016]. As an optimization, we stochastically round the point count n_g to a multiple of a parameter $K \in \mathbb{N}$, which we typically set to the supersampling rate. This way, each thread can splat K independent points after reading the data of a Gaussian and therefore this cost is amortized better. This is an optional deviation from our theory of Poisson point processes, but we found the visual impact to be small and the speedup to be substantial (Fig. 7).

3.5 Culling

Our Gaussian point splatting scales favorably to large scenes because the overhead per Gaussian is low, few points are splatted for small Gaussians and it distributes work evenly across many threads. However, the number of points that need to be splatted can still grow arbitrarily when there are many Gaussians occupying the same regions of the image. A lot of effort may be wasted on surfaces that are entirely occluded by other surfaces. Therefore, we incorporate occlusion culling as additional optimization. Although all Gaussians are semi-transparent, our stochastic approach uses a depth buffer that provides information about which Gaussians can be culled. We also use frustum culling.

The output is an array with 1 bit per Gaussian encoding its visibility (packed into 32-bit integers). The work distribution method (Sec. 3.2) uses this array to skip writing, reading and scattering of point counts for culled Gaussians. To ensure that occlusion culling does not introduce any errors, we use a two-phase approach [Aaltonen and Haar 2015]. In the first phase, we render all Gaussians that do not get culled based on the depth buffer from the previous frame. In the second phase, we render all Gaussians that do not get culled based on the depth buffer from the first phase and were not rendered there. Usually, the point count in the second phase is low, but it guarantees that we never miss anything. We experimented with temporal reprojection of the depth buffer from the previous frame but found that the overhead does not pay off.

The occlusion culling itself uses a hierarchical depth buffer, i.e. we repeatedly downsample the depth buffer by a factor of 2×2, keeping the maximal depth value [Akenine-Möller et al. 2018, Sec. 19.7.2]. To test visibility of a primitive, we compute its screen-space axis-aligned bounding box (AABB) and its minimal depth. Then we identify the highest-resolution level of the hierarchy where this AABB covers only 2×2 pixels. If these four depths are all less than the minimal depth, the primitive gets culled.

To limit the overhead of culling in the presence of extremely large numbers of Gaussians, we employ a two-level hierarchy. We sort Gaussians based on either 64-bit Morton codes of their mean or a depth-first traversal of a bounding volume hierarchy (built with the surface area heuristic using tinyBVH). Then we produce one million groups, each with roughly equal numbers of consecutive Gaussians and store a 3D AABB per group. Note that this only needs to happen once per scene. Culling first operates on these groups and then culls the individual Gaussians in the remaining groups.

3.6 Our Renderer

Our renderer is implemented in CUDA. To render a frame, it runs the following kernels: First, there is culling at the granularity of groups using one thread per group. Then, a Gaussian preprocessing kernel performs multiple tasks using one thread per Gaussian: It checks if the Gaussian itself is culled and, if not, computes its point count (Sec. 3.4) and the color, based on the spherical harmonics (SH) coefficients. After that, the prefix sum, scattering and maximum scan take care of work distribution (Sec. 3.2), which is followed by the point splatting itself (Sec. 3.1). The resulting framebuffer gets resolved, blending subpixels together (second phase only), and its depth buffer gets extracted and then downsampled repeatedly. Finally, all of these kernels run again in the second phase to render Gaussians that were falsely culled in the first phase (Sec. 3.5).

To save on bandwidth and to maximize the number of Gaussians that fit into VRAM, we quantize their data. Most attributes use fixed-point quantization relative to minimal and maximal values computed per scene. We use 23, 22 and 23 bits for the x-, y- and z-coordinate of the mean, respectively. The logarithm of the standard deviation along each principal axis is quantized into 10 bits. The principal axes themselves are stored as a quaternion using quantization into 30 bits [Reynolds 2017]. Together, all these attributes take up 128 bits. The opacity is stored separately in 8 bits. We store the three SH coefficients for band 0 as 32-bit floats, but divide all other coefficients by the respective coefficient in band 0 and use 8-bit fixed-point quantization [Sloan and Silvennoinen 2020]. When SH are not needed, we only store colors with 10 bits per sRGB channel.

Overall, a Gaussian takes up 21 bytes without SH or 81 bytes with SH. Point counts for work distribution take another 4 bytes per Gaussian (the scans operate in situ) and culling masks take 2 bits. If we can spare 10 GiB of VRAM for Gaussians, 425 million Gaussians will fit without SH (Fig. 1) and 125 million will fit with SH.

To reduce noise, we use temporal accumulation for static cameras and temporal reprojection for moving cameras. Accumulation progressively takes the arithmetic mean of all frames. Reprojection uses view-projection matrices of the previous and current frame to project a pixel position to the previous frame [Yang et al. 2020, Eq. 1]. We compute the pixel position based on the minimal depth of all subpixels. Then we clamp the color from the previous frame to an axis-aligned RGB box constructed from a 3×3-neighborhood in the current frame. The reprojected and clamped color enters into an exponential moving average with weight 0.9. Our reprojection is effective against noise but prone to ghosting and loss of detail. We only use it in our supplemental video and expect that more sophisticated volume denoising would perform considerably better [Hofmann et al. 2021].

4 Results

We evaluate our method using an NVIDIA RTX 4070 with 12 GiB VRAM (with the exception of Fig. 1). Our test scenes are generated

Table 1. A comparison of image quality between the original 3DGS and our method at various SPP using 1×1 supersampling and $K = 1$. The results are averaged across scenes from Mip-NeRF360 [Barron et al. 2022], Deep Blending [Hedman et al. 2018] and Tanks&Temples [Knapitsch et al. 2017], trained using 3DGS.

	3DGS	Ours (SPP)					
		1	4	16	64	256	1024
PSNR	25.71	16.94	21.23	24.01	25.23	25.63	25.73
SSIM	0.759	0.241	0.435	0.613	0.712	0.750	0.761
LPIPS	0.178	0.845	0.526	0.304	0.196	0.166	0.171

using 3DGS [Kerbl et al. 2023]: truck (2.5 M Gaussians), bicycle (6.1 M Gaussians), Opole campus (25 M Gaussians), St. Sebastian church (30 M Gaussians), and Jastrzębia Góra (106 M Gaussians). Jastrzębia Góra is the only scene without SH. We compare our method to several related works: The original 3DGS [Kerbl et al. 2023] is the reference solution since our scenes are optimized for it. StochasticSplats [Kheradmand et al. 2025] rasterizes Gaussians with stochastic transparency [Enderton et al. 2010] to avoid sorting, similar to our method. Splatshop [Schütz et al. 2025] is similar in nature to 3DGS, but optimized extensively, e.g. through changes in the sorting and more aggressive culling of fragments. Some of these optimizations introduce visual differences. Most notably, Gaussians below a certain screen-space size get culled completely. This culling makes the method considerably more effective for distant views but also introduces substantial errors. We first evaluate how our method compares to 3DGS in terms of quality (Sec. 4.1), then investigate its scalability and efficiency (Sec. 4.2) and finally discuss limitations (Sec. 4.3).

4.1 Quality

The main way our results deviate from 3DGS is noise, which diminishes at higher supersampling rates (Fig. 5, Table 1). Rendering at a supersampling rate of 2×2 roughly doubles frame times, although the point count quadruples. With such a sample count, the residual noise is already modest and 4×4 supersampling further diminishes it. Since this noise arises from stochastic transparency, the noise characteristics are the same as for other techniques that rely on it [Kheradmand et al. 2025]. We use 2×2 supersampling as default in our experiments, unless otherwise mentioned. Our supplemental material includes more images at different supersampling rates.

Fig. 6 compares our results at ca. 10^5 samples per pixel (spp) to 3DGS. We observe slight bias in high-frequency regions of the image, e.g. on the spokes of the bike. These discrepancies occur because aliasing manifests differently in the two techniques. While 3DGS samples the opacity of each Gaussian at the pixel center, the results of our method are more akin to prefiltering with a box filter. Sec. 4.3 discusses this aspect in more detail.

Sec. 3.4 proposes a parameter $K \in \mathbb{N}$, which is the number of points splatted by a single thread for a single Gaussian. That entails rounding the Poisson-distributed point count to a multiple of K, which is a deviation from our theory and introduces bias. Fig. 7 demonstrates that this bias is relatively small in practice. At the same time, rendering with $K = 1$ instead of $K = 4$ increases the frame

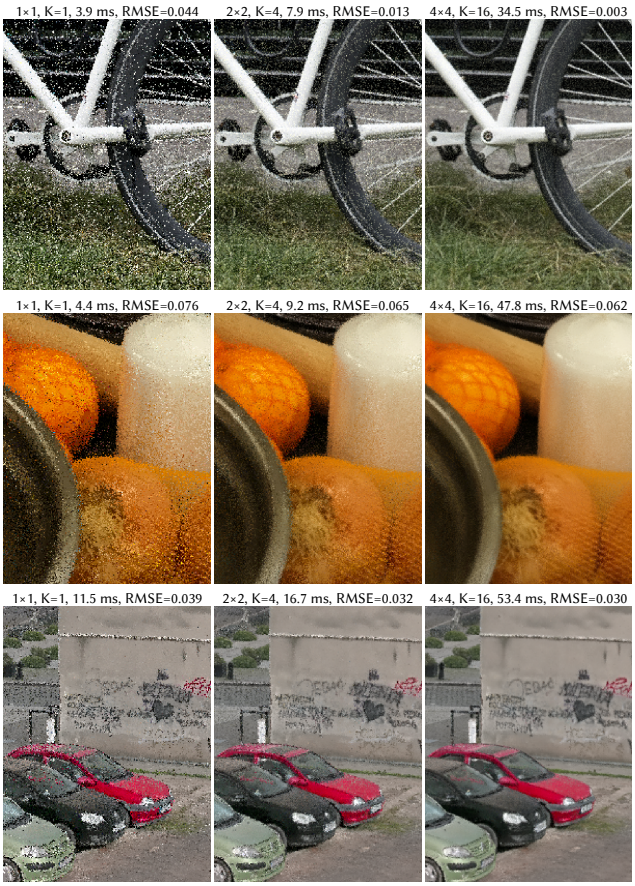


Fig. 5. An example of how noise reduces at growing supersampling rates on the bicycle (top), counter (center), and St. Sebastian church (bottom) scenes. Frame times and RMSEs pertain to full 1920×1080 frames and RMSEs compare to the 3DGS reference.

Fig. 6. Converged results of our method (left), 3DGS (middle) and per-pixel RMSEs between the two images (right). Differences in aliasing cause slight bias in high-frequency regions.

Fig. 7. A comparison between $K = 1$ (left) and $K = 4$ (middle), both with converged images and 2×2 supersampling. To the right, we show the per-pixel RMSE between these two images. The full 1920×1080 frames take 13.3 ms and 9.7 ms to render, respectively.

time by 37%. Therefore, we consider this optimization justifiable and use $K = 4$ by default for 2×2 supersampling.

4.2 Scalability

To evaluate the efficiency and scalability of various techniques, we have animated a camera path for each of our test scenes. This way, our occlusion culling gets access to a depth buffer for a similar but different view from the previous frame, as in real-time rendering. The camera paths generally start out close to the most detailed parts of the scene (e.g. near the truck as shown in Fig. 13) and move upwards and away over time, which leads to a significantly higher number of Gaussians in the camera frustum. Our supplemental video shows results of our technique for all camera paths, using 2×2 supersampling, $K = 4$ and both with and without temporal reprojection for denoising. All timings are medians across 10 runs and refer to a resolution of 1920×1080 (but many figures are cropped).

Fig. 8 shows our results. When graphs are missing, it means that the corresponding techniques either crashed when trying to render this scene or that the timings were orders of magnitude worse than for our method. For example, 3DGS renders St. Sebastian church two orders of magnitude slower (the renderer uses more than 12 GiB of VRAM for this scene). Our method renders all scenes successfully and achieves the most consistent frame times. With culling and 2×2 supersampling, the frame rate never drops below 24 Hz. 3DGS and Splatshop without culling of small Gaussians perform poorly when many Gaussians fall into a small screen-space region, i.e. towards the end of our camera paths. For example, for the frame in Fig. 10a, some tiles used for rasterization contain more than 10,000 Gaussian fragments. The load balancing of 3DGS and Splatshop is correspondingly bad. Contrary to that, our method achieves perfect load balancing (Sec. 3.2) and becomes faster as the point count diminishes. StochasticSplats scale quite poorly to scenes with more Gaussians, e.g. St. Sebastian church.

Our method is the only one capable of rendering Jastrzębia Góra fully on an RTX 4070. Splatshop with culling of small Gaussians achieves similar timings, but as shown in Fig. 9, this culling introduces strong visual differences as only 8% of the Gaussians are being rendered. We can push our method even further, rendering four copies of Jastrzębia Góra simultaneously for a total of 425 M Gaussians. The results in Fig. 1 use 1×1 supersampling for the

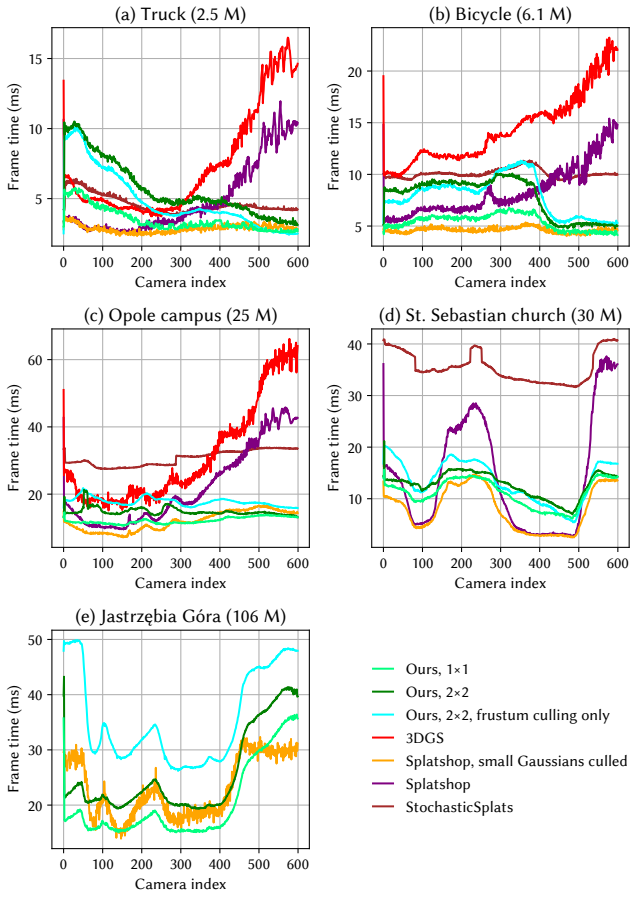


Fig. 8. Frame times of different techniques for our camera paths in five different scenes. Our method successfully renders even the largest scene and achieves real-time frame rates consistently.

overview and 2×2 for the close-up views, $K = 4$ and a resolution of 1920×1080. We used temporal accumulation to obtain converged results. Splatshop cannot handle this scene. For even larger scenes, our VRAM is exhausted, but Gaussian point splatting should be able to scale to a billion Gaussians and beyond on more potent GPUs.

We also observe that occlusion culling becomes more effective in larger scenes (Fig. 8). Fig. 11 investigates how it works for the frame shown in Fig. 10b, recorded in the St. Sebastian church scene at camera index 5. By looking at the scene from a different perspective, while rendering only the Gaussians that were not culled in the original view, we find that most Gaussians that are hidden by the facades are successfully culled. The difference between the frame times for our method with and without occlusion culling in Fig. 8 (d) at camera index 5 reflects this successful culling.

4.3 Limitations

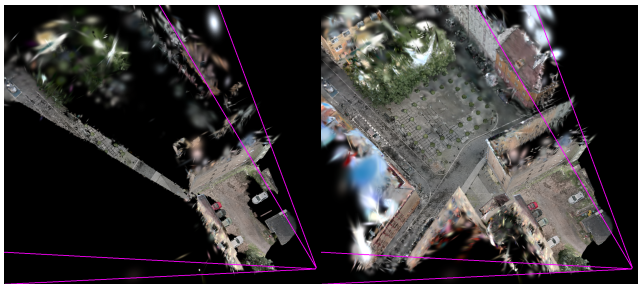
As observed in Sec. 4.1, our converged results differ from 3DGS because aliasing manifests differently. Eq. 1 assumes that the density of a Gaussian is nearly constant across the footprint of a pixel. For

(a) Ours (b) Splatshop, small Gaussians culled

Fig. 9. The optional culling of small Gaussians in Splatshop is beneficial to its scalability but introduces strong artifacts. In this distant view, most Gaussians are small and thus get culled. Only 8.2 M out of the 106 M Gaussians in the scene get rendered. Without culling of small Gaussians, Splatshop cannot render this scene. Our method splats points for every single Gaussian with the right probability.

(a) Opole campus, cam. index 600 (b) St. Sebastian church, cam. index 5

Fig. 10. Example frames from our camera paths. They end with large overview shots of the scenes. The path for St. Sebastian church starts with a view where a building occludes a public square behind it.



(a) With occlusion culling (b) Without occlusion culling

Fig. 11. We study the situation in Fig. 10b from a different perspective. The original frustum is shown by magenta lines. Occlusion culling successfully culls most of the hidden Gaussians in the town square behind the facades. Note that frustum culling is enabled in both cases.

Gaussians at the scale of a pixel, this assumption is violated. 3DGS samples Gaussians at pixel centers, which makes it prone to aliasing artifacts. Reproducing this behavior exactly with Gaussian point splatting would be challenging. Since we sample points from a continuous distribution, optionally with supersampling, and then splat them into pixels, the behavior of our method is more akin to prefiltering with a box filter. Arguably, this behavior is better for antialiasing, but since inverse rendering optimizes scenes to look right with the aliasing of 3DGS, our results may be slightly farther from the reference. In any case, the differences are small (Fig. 6). Furthermore, Fig. 12 demonstrates that this bias goes away when we zoom in heavily until Gaussians are larger than a pixel.

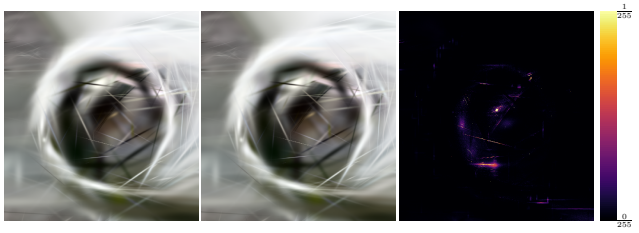


Fig. 12. Converged results of a high resolution render of the crank bolt of the bicycle using our method (left), 3DGS (middle) and per-pixel RMSEs between the two images (right). Gaussians cover more pixels due to the higher resolution, reducing aliasing differences, resulting in a maximum error under 1/255, compared to 5/255, as seen in Figure 6.

(a) Normal viewpoint (b) Close-up

Fig. 13. The truck scene rendered using our method, from a viewpoint used during optimization (left), and up close (right), without occlusion culling, at 2×2 supersampling with $K = 4$. Our method renders the left image using 38 million points at 10.2 ms, and the right image using 168 million points at 29.1 ms.

Our method offers consistent frame times and scales well to large scenes, but 3DGS may be faster for small scenes (Fig. 8). For example, the truck (Fig. 13 left) has a high concentration of Gaussians for this central object. When the camera gets close, it can result in excessive point counts, especially when individual Gaussians come extremely close to the camera (Fig. 13 right). Though, such views look strange anyway and our frame times remain generally competitive. We alleviate the worst cases by clamping the point count for a single Gaussian so that it launches at most one thread per pixel and by placing the near-clipping plane reasonably far away.

It is not obvious how our method could be used for differentiable rendering. Thus, optimization of scenes using our renderer is currently not possible. We rely on existing implementations of 3DGS

to obtain scenes and only use our method for rendering. This is also the reason why we invest effort to replicate the behavior of 3DGS faithfully, including artifacts such as popping. In principle, we could sample points from a 3D distribution to eliminate popping, but with existing scenes that would deteriorate overall quality.

5 Conclusions

Gaussian point splatting achieves unprecedented scalability. On our tested GPU configuration, any scene that fits into VRAM can be rendered interactively. Given how compute throughput, bandwidth and memory capacity scale, we expect the same for larger GPUs. The key to this performance is a novel Monte Carlo approach and a fundamentally different approach to parallelism. Threads splat points independently, yet we guarantee that each pixel receives a point with the right probability. In this first application of Gaussian point splatting, we have been careful to reproduce the idiosyncrasies of 3DGS. However, we believe that our method would be even more compelling for unbiased volume rendering. Popping artifacts would be eliminated in the most natural way and the image formation model would simplify. The only challenge is the inverse rendering.

Acknowledgments

This work was supported by the Dutch Research Council (NWO) under the project *VR Retrofit-4U* (Grant ID: https://doi.org/10.61686/EIHMV70145). We thank Andrii Shramko and the team at Teleportour for providing the 3D scanning data used in Figs. 1, 5 (bottom), 9, 10 and 11 and authors of prior work for the remaining scenes [Barron et al. 2022; Hedman et al. 2018; Knapitsch et al. 2017].

References

Sebastian Aaltonen and Ulrich Haar. 2015. Advances in real time rendering, part II: GPU-Driven Rendering Pipelines. In *ACM SIGGRAPH 2015 Courses*. doi:10.1145/2776880.2787702 https://advances.realtimerendering.com/s2015/.

Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. 2018. Real-Time Rendering (4th ed.). A K Peters/CRC Press.

Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. 2022. Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/CVPR52688.2022.00539

Jiazhong Cen, Jiemin Fang, Chen Yang, Lingxi Xie, Xiaopeng Zhang, Wei Shen, and Qi Tian. 2025. Segment Any 3D Gaussians. *Proceedings of the AAAI Conference on Artificial Intelligence* 39, 2 (2025). doi:10.1609/aaai.v39i2.32193

Jiaqi Chen, Xinhao Ji, Yuanyuan Gao, Hao Li, Yuning Gong, Yifei Liu, Dan Xu, Zhihang Zhong, Dingwen Zhang, and Xiao Sun. 2025. ExGS: Extreme 3D Gaussian Compression with Diffusion Priors. arXiv:2509.24758 [cs.CV]

Yu Chen and Gim Hee Lee. 2024. DOGS: Distributed-Oriented Gaussian Splatting for Large-Scale 3D Reconstruction Via Gaussian Consensus. In *Advances in Neural Information Processing Systems*, Vol. 37. Curran Associates. doi:10.52202/079017-1087

Jiadi Cui, Junming Cao, Fuqiang Zhao, Zhipeng He, Yifan Chen, Yuhui Zhong, Lan Xu, Yujiao Shi, Yingliang Zhang, and Jingyi Yu. 2024. LetsGo: Large-Scale Garage Modeling and Rendering via LiDAR-Assisted Gaussian Primitives. *ACM Trans. Graph.* 43, 6 (2024). doi:10.1145/3687762

Hiba Dahmani, Moussab Bennehar, Nathan Piasco, Luis Roldão, and Dzmitry Tsishkou. 2025. SWAG: Splatting in the Wild Images with Appearance-Conditioned Gaussians. In *Computer Vision – ECCV 2024*. Springer. doi:10.1007/978-3-031-73116-7_19

Eric Enderton, Erik Sintorn, Peter Shirley, and David Luebke. 2010. Stochastic transparency. In *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. doi:10.1145/1730804.1730830

Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, and Zhangyang Wang. 2024. LightGaussian: unbounded 3D Gaussian compression with 15x reduction and 200+ FPS. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*. Curran Associates. doi:10.52202/079017-4447

Guofeng Feng, Siyan Chen, Rong Fu, Zimu Liao, Yi Wang, Tao Liu, Boni Hu, Lining Xu, Zhilin Pei, Hengjie Li, Xiuhong Li, Ninghui Sun, Xingcheng Zhang, and Bo Dai. 2025. FlashGS: Efficient 3D Gaussian Splatting for Large-scale and High-resolution Rendering. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/CVPR52734.2025.02482

Yuanyuan Gao, Yuning Gong, Yifei Liu, Li Jingfeng, Zhihang Zhong, Dingwen Zhang, Yanci Zhang, Dan Xu, and Xiao Sun. 2025. Proxy-GS: Unified Occlusion Priors for Training and Inference in Structured 3D Gaussian Splatting. arXiv:2509.24421 [cs.CV]

Michael B. Giles. 2016. Algorithm 955: Approximation of the Inverse Poisson Cumulative Distribution Function. *ACM Trans. Math. Softw.* 42, 1 (2016). doi:10.1145/2699466

Sharath Girish, Kamal Gupta, and Abhinav Shrivastava. 2025. EAGLES: Efficient Accelerated 3D Gaussians with Lightweight Encodings. In *Computer Vision – ECCV 2024*. Springer. doi:10.1007/978-3-031-73036-8_4

Florian Hahlbohm, Fabian Friederichs, Tim Weyrich, Linus Franke, Moritz Kappel, Susana Castillo, Marc Stamminger, Martin Eisemann, and Marcus Magnor. 2025. Efficient Perspective-Correct 3D Gaussian Splatting Using Hybrid Transparency. *Computer Graphics Forum* 44, 2 (2025). doi:10.1111/cgf.70014

Alex Hanson, Allen Tu, Geng Lin, Vasu Singla, Matthias Zwicker, and Tom Goldstein. 2025a. Speedy-Splat: Fast 3D Gaussian Splatting with Sparse Pixels and Sparse Primitives. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/CVPR52734.2025.02006

Alex Hanson, Allen Tu, Vasu Singla, Mayuka Jayawardhana, Matthias Zwicker, and Tom Goldstein. 2025b. PUP 3D-GS: Principled Uncertainty Pruning for 3D Gaussian Splatting. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/CVPR52734.2025.00558

Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. 2018. Deep Blending for Free-viewpoint Image-based Rendering. 37, 6 (2018). doi:10.1145/3272127.3275084

Nikolai Hofmann, Jon Hasselgren, Petrik Clarberg, and Jacob Munkberg. 2021. Interactive Path Tracing and Reconstruction of Sparse Volumes. *Proc. ACM Comput. Graph. Interact. Tech.* 4, 1 (2021). doi:10.1145/3451256

Qiqi Hou, Randall Rauwendaal, Zifeng Li, Hoang Le, Farzad Farhadzadeh, Fatih Porikli, Alexei Bourd, and Amir Said. 2025. Sort-free Gaussian Splatting via Weighted Sum Rendering. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=y8uPsxR8PN

Xiaotong Huang, He Zhu, Zihan Liu, Weikai Lin, Xiaohong Liu, Zhezhi He, Jingwen Leng, Minyi Guo, and Yu Feng. 2025. SeeLe: A Unified Acceleration Framework for Real-Time Gaussian Splatting. In *2026 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. arXiv:2503.05168 [cs.GR]

Joongho Jo and Jongsun Park. 2025. GS-TG: 3D Gaussian Splatting Accelerator with Tile Grouping for Reducing Redundant Sorting while Preserving Rasterization Efficiency. In *Proceedings of the 62nd Annual ACM/IEEE Design Automation Conference*. IEEE Press. doi:10.1109/DAC63849.2025.11133283

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkuehler, and George Drettakis. 2023. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Trans. Graph.* 42, 4 (2023). doi:10.1145/3592433

Bernhard Kerbl, Andreas Meuleman, Georgios Kopanas, Michael Wimmer, Alexandre Lanvin, and George Drettakis. 2024. A Hierarchical 3D Gaussian Representation for Real-Time Rendering of Very Large Datasets. *ACM Trans. Graph.* 43, 4 (2024). doi:10.1145/3658160

Shakiba Kheradmand, Delio Vicini, George Kopanas, Dmitry Lagun, Kwang Moo Yi, Mark Matthews, and Andrea Tagliasacchi. 2025. StochasticSplats: Stochastic Rasterization for Sorting-Free 3D Gaussian Splatting. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. arXiv:2503.24366 [cs.CV]

Mijeong Kim, Jongwoo Lim, and Bohyung Han. 2024. 4D Gaussian splatting in the wild with uncertainty-aware regularization. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*. Curran Associates. doi:10.52202/079017-4104

Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. 2017. Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction. In *ACM SIGGRAPH 2017 Conference Proceedings*. doi:10.1145/3072959.3073599

Jonas Kulhanek, Songyou Peng, Zuzana Kukelova, Marc Pollefeys, and Torsten Sattler. 2024. WildGaussians: 3D Gaussian splatting in the wild. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*. Curran Associates. doi:10.52202/079017-0670

Junseo Lee, Seokwon Lee, Jungi Lee, Junyong Park, and Jaewoong Sim. 2024. GSCore: Efficient Radiance Field Rendering via Architectural Support for 3D Gaussian Splatting. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. doi:10.1145/3620666.3651385

Hans-Peter Lehmann, Lorenz Hübschle-Schneider, and Peter Sanders. 2022. Weighted Random Sampling on GPUs. arXiv:2106.12270 [cs.DS]

Linfei Li, Lin Zhang, Zhong Wang, and Ying Shen. 2024. GS3LAM: Gaussian Semantic Splatting SLAM. In *Proceedings of the 32nd ACM International Conference on Multimedia*. doi:10.1145/3664647.3680739

Jiaqi Lin, Zhihao Li, Xiao Tang, Jianzhuang Liu, Shiyong Liu, Jiayue Liu, Yangdi Lu, Xiaofei Wu, Songcen Xu, Youliang Yan, and Wenming Yang. 2024. Vast-Gaussian: Vast 3D Gaussians for Large Scene Reconstruction. In *Proceedings*

of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR). doi:10.1109/CVPR52733.2024.00494

Weihang Liu, Yuke Li, Yuxuan Li, Jingyi Yu, and Xin Lou. 2025. Duplex-GS: Proxy-Guided Weighted Blending for Real-Time Order-Independent Gaussian Splatting. arXiv:2508.03180 [cs.CV]

Yang Liu, Chuanchen Luo, Lue Fan, Naiyan Wang, Junran Peng, and Zhaoxiang Zhang. 2024. CityGaussian: Real-Time High-Quality Large-Scale Scene Rendering with Gaussians. In *Computer Vision – ECCV 2024*. Springer. doi:10.1007/978-3-031-72640-8_15

Morgan McGuire and Louis Bavoil. 2013. Weighted Blended Order-Independent Transparency. *Journal of Computer Graphics Techniques (JCGT)* 2, 2 (2013). http://jcgt.org/published/0002/02/09/

Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. 2020. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In *Computer Vision – ECCV 2020*. Springer. doi:10.1007/978-3-030-58452-8_24

Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. 2022. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.* 41, 4 (2022). doi:10.1145/3528223.3530127

Simon Niedermayr, Josef Stumpfegger, and Rüdiger Westermann. 2024. Compressed 3D Gaussian Splatting for Accelerated Novel View Synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/CVPR52733.2024.00985

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 2007. Numerical Recipes: The Art of Scientific Computing (3rd ed.). (2007).

Minghan Qin, Wanhua Li, Jiawei Zhou, Haoqian Wang, and Hanspeter Pfister. 2024. LangSplat: 3D Language Gaussian Splatting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/CVPR52733.2024.01895

Lukas Radl, Michael Steiner, Mathias Parger, Alexander Weinrauch, Bernhard Kerbl, and Markus Steinberger. 2024. StopThePop: Sorted Gaussian Splatting for View-Consistent Real-time Rendering. *ACM Trans. Graph.* 43, 4 (2024). doi:10.1145/3658187

Kerui Ren, Lihan Jiang, Tao Lu, Mulin Yu, Linning Xu, Zhangkai Ni, and Bo Dai. 2025. Octree-GS: Towards Consistent Real-time Rendering with LOD-Structured 3D Gaussians. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2025). doi:10.1109/TPAMI.2025.3568201

Marc B. Reynolds. 2017. Quaternion Quantization: part 1. https://marc-b-reynolds.github.io/quaternions/2017/05/02/QuatQuantPart1.html. Blog post.

Markus Schütz, Bernhard Kerbl, and Michael Wimmer. 2021. Rendering Point Clouds with Compute Shaders and Vertex Order Optimization. *Computer Graphics Forum* 40, 4 (2021). doi:10.1111/cgf.14345

Markus Schütz, Christoph Peters, Florian Hahlbohm, Elmar Eisemann, Marcus Magnor, and Michael Wimmer. 2025. Splatshop: Efficiently Editing Large Gaussian Splat Models. *Computer Graphics Forum* 44, 8 (2025). doi:10.1111/cgf.70214

Saptarshi Neil Sinha, Holger Graf, and Michael Weinmann. 2025. SpectralGaussians: Semantic, spectral 3D Gaussian splatting for multi-spectral scene representation, visualization and analysis. *ISPRS Journal of Photogrammetry and Remote Sensing* 227 (2025). doi:10.1016/j.isprsjprs.2025.06.008

Peter-Pike Sloan and Ari Silvennoinen. 2020. Precomputed Lighting Advances in Call of Duty: Modern Warfare. In *SIGGRAPH 2020 Course: Advances in Real-Time Rendering in 3D Graphics and Games*. https://advances.realtimerendering.com/s2020/

Michael Steiner, Thomas Köhler, Lukas Radl, Felix Windisch, Dieter Schmalstieg, and Markus Steinberger. 2025. AAA-Gaussians: Anti-Aliased and Artifact-Free 3D Gaussian Rendering. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. arXiv:2504.12811 [cs.GR]

Roy L. Streit. 2010. *Poisson Point Processes: Imaging, Tracking, and Sensing*. Springer. doi:10.1007/978-1-4419-6923-1

Lewis A. G. Stuart, Andrew Morton, Ian Stavness, and Michael P. Pound. 2025. 3DGS-to-PC: 3D Gaussian Splatting to Dense Point Clouds. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*. doi:10.1109/ICCVW69036.2025.00395

Xin Sun, Iliyan Georgiev, Yun (Raymond) Fei, and Milos Hasan. 2025. Stochastic Ray Tracing of Transparent 3D Gaussians. In *Eurographics Symposium on Rendering*. doi:10.2312/sr.20251191

Christopher Thirgood, Oscar Mendez, Erin Ling, Jon Storey, and Simon Hadfield. 2025. HyperGS: Hyperspectral 3D Gaussian Splatting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/CVPR52734.2025.00560

Allen Tu, Haiyang Ying, Alex Hanson, Yonghan Lee, Tom Goldstein, and Matthias Zwicker. 2025. SpeeDe3DGS: Speedy Deformable 3D Gaussian Splatting with Temporal Pruning and Motion Grouping. arXiv:2506.07917 [cs.GR]

Xinzhe Wang, Ran Yi, and Lizhuang Ma. 2024. AdR-Gaussian: Accelerating Gaussian Splatting with Adaptive Radius. In *SIGGRAPH Asia 2024 Conference Papers*. doi:10.1145/3680528.3687675

Zhican Wang, Guanghui He, Dantong Liu, Lingjun Gao, Shell Xu Hu, Chen Zhang, Zhuoran Song, Nicholas Lane, Wayne Luk, and Hongxiang Fan. 2025. Accelerating 3D Gaussian Splatting with Neural Sorting and Axis-Oriented Rasterization. arXiv:2506.07069 [cs.GR]

Yongchang Wu, Zipeng Qi, Zhenwei Shi, and Zhengxia Zou. 2025. BlockGaussian: Efficient Large-Scale Scene Novel View Synthesis via Adaptive Block-Based Gaussian Splatting. arXiv:2504.09048 [cs.CV]

Lei Yang, Shiqiu Liu, and Marco Salvi. 2020. A Survey of Temporal Antialiasing Techniques. *Computer Graphics Forum* 39, 2 (2020). doi:10.1111/cgf.14018

Xijie Yang, Linning Xu, Lihan Jiang, Dahua Lin, and Bo Dai. 2025. Virtualized 3D Gaussians: Flexible Cluster-based Level-of-Detail System for Real-Time Rendering of Composed Scenes. In *SIGGRAPH 2025 Conference Papers*. doi:10.1145/3721238.3730602

Dayou Zhang, Zhicheng Liang, Zijian Cao, Dan Wang, and Fangxin Wang. 2025a. SRBF-Gaussian: Streaming-Optimized 3D Gaussian Splatting. In *2025 IEEE Conference Virtual Reality and 3D User Interfaces (VR)*. doi:10.1109/VR59515.2025.00070

Dongbin Zhang, Chuming Wang, Weitao Wang, Peihao Li, Minghan Qin, and Haoqian Wang. 2025b. Gaussian in the Wild: 3D Gaussian Splatting for Unconstrained Image Collections. In *Computer Vision – ECCV 2024*. Springer. doi:10.1007/978-3-031-73116-7_20

Zhiyu Zheng, Dake Zhou, Yiming Shao, and Xin Yang. 2025. EGU-GS: Efficient Gaussian utilization for real-time 3D Gaussian splatting. *Image and Vision Computing* 162 (2025). doi:10.1016/j.imavis.2025.105687

He Zhu, Zheng Liu, Xingyang Li, Anbang Wu, Jieru Zhao, Fangxin Liu, Yiming Gan, Jingwen Leng, and Yu Feng. 2026. Nebula: Infinite-Scale 3D Gaussian Splatting in VR via Collaborative Rendering and Accelerated Stereo Rasterization. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. doi:10.1145/3779212.3790190

A Solving the CDF Integral

We solve the integral for the CDF using polar coordinates:

$$F_\alpha(r_{\max}) = 2\pi \int_0^{r_{\max}} -r \ln\left(1 - \alpha \exp\left(-\frac{r^2}{2}\right)\right) \mathrm{d}r.$$

Now we employ the substitution

$$s := \alpha \exp\left(-\frac{r^2}{2}\right) \quad \text{with} \quad \frac{\partial s}{\partial r} = -r\alpha \exp\left(-\frac{r^2}{2}\right) = -rs$$

and exploit that we know the derivative of Li_2:

$$\begin{aligned} F_\alpha(r_{\max}) &= 2\pi \int_0^{r_{\max}} \frac{-rs}{s} \ln\left(1 - \alpha \exp\left(-\frac{r^2}{2}\right)\right) \mathrm{d}r \\ &= 2\pi \int_\alpha^{\alpha \exp\left(-\frac{r_{\max}^2}{2}\right)} \frac{\ln(1-s)}{s} \,\mathrm{d}s \\ &= 2\pi \,\mathrm{Li}_2(\alpha) - 2\pi \,\mathrm{Li}_2\left(\alpha \exp\left(-\frac{r_{\max}^2}{2}\right)\right). \end{aligned}$$

B Inverting the CDF Integral

To implement inverse CDF sampling, we solve the following equation for $r > 0$:

$$\begin{aligned} \frac{F_\alpha(r)}{F_\alpha(\infty)} = u_0 \quad &\Leftrightarrow \quad \mathrm{Li}_2(\alpha) - \mathrm{Li}_2\left(\alpha \exp\left(-\frac{r^2}{2}\right)\right) = \mathrm{Li}_2(\alpha) u_0 \\ &\Leftrightarrow \quad \mathrm{Li}_2\left(\alpha \exp\left(-\frac{r^2}{2}\right)\right) = (1-u_0)\,\mathrm{Li}_2(\alpha) \\ &\Leftrightarrow \quad -\frac{r^2}{2} = \ln\left(\frac{1}{\alpha}\,\mathrm{Li}_2^{-1}((1-u_0)\,\mathrm{Li}_2(\alpha))\right) \\ &\Leftrightarrow \quad r = \sqrt{-2\ln\left(\frac{1}{\alpha}\,\mathrm{Li}_2^{-1}((1-u_0)\,\mathrm{Li}_2(\alpha))\right)}. \end{aligned}$$