Hardware-Rasterized Ray-Based Gaussian Splatting

Samuel Rota Bulò, Nemanja Bartolovic, Lorenzo Porzi, Peter Kontschieder
Meta Reality Labs, Zürich

Abstract

We present a novel, hardware-rasterized rendering approach for ray-based 3D Gaussian Splatting (RayGS), obtaining both fast and high-quality results for novel view synthesis. Our work contains a mathematically rigorous and geometrically intuitive derivation about how to efficiently estimate all relevant quantities for rendering RayGS models, structured with respect to standard hardware rasterization shaders. Our solution is the first enabling rendering RayGS models at sufficiently high frame rates to support quality-sensitive applications like Virtual and Mixed Reality. Our second contribution enables alias-free rendering for RayGS, by addressing MIP-related issues arising when rendering diverging scales during training and testing. We demonstrate significant performance gains, across different benchmark scenes, while retaining state-of-the-art appearance quality of RayGS.

1. Introduction

The advent of recently introduced image-based reconstruction methods like Neural Radiance Fields (NeRFs) [19] and 3D Gaussian Splatting (3DGS) [12] has paved the way for a new era of photorealistic novel view synthesis in Virtual and Mixed Reality applications. Radiance fields can capture subtle nuances of real-world scenes, including fine-grained texture details, complex lighting phenomena, and transparent surfaces. In particular, 3DGS has a number of interesting properties including interpretability, flexibility, and efficiency for both training and real-time rendering.

Once the scene models are reconstructed, fast and real-time rendering capabilities are required for applications like novel view synthesis. While the original 3DGS paper is already significantly faster compared to NeRFs, further works have addressed improving rendering speed [5, 6, 23]. However, many of their target applications still center on rendering for 2D consumption or on small (mobile) screens, which allows for certain glitches in terms of rendering artifacts and overall quality. In contrast, we observed that VR applications set the bar significantly higher when it comes to rendering quality, otherwise breaking the immersion and thus the overall experience.

A number of works have contributed to further improving 3DGS' reconstruction quality, *e.g.*, by using a better densification procedure [2], removing popping artifacts [26], or leveraging a ray casting based approach for computing ray-Gaussian intersections [9, 10, 39] rather than using the originally proposed splatting formulation. Many of these improvements are complementary in nature, but particularly ray-tracing based volume rendering of 3D Gaussians (RayGS) [10] has shown superior quality. The quality gain is mostly due to eliminating some of the approximations needed in traditional 3DGS, however, this comes at increased computational costs making it unsuitable for high frame rate applications based on consumer-grade hardware.

In our work we propose a novel and substantially faster renderer based on hardware-rasterization, while retaining the high quality of ray-based Gaussian Splatting. Hardware rasterization pipelines have been successfully demonstrated for standard 3DGS [30], enabling cross-platform usage based on using standard graphics pipeline components. For standard 3DGS, vertex shaders were used to determine the support area of each Gaussian primitive by means of the quad enclosing the minimum area of the corresponding, projected ellipse on the image plane. The fragment shader then computes the primitive's opacity information, given the positional information computed by the vertex shader. Providing the analogous quantities for the RayGS case is non-trivial, and forms the core contribution of our paper. We propose a solution that yields the smallest enclosing quads in 3D space, which turns out to be approximately as fast as the hardware-rasterized variant for standard 3DGS, while retaining the higher quality of RayGS. We discuss the challenges of selecting the computationally most efficient solution out of infinitely many valid ones.

The second contribution of our paper addresses MIP-related issues in a RayGS formulation. Our solution enables alias-free rendering of images at diverging test and training scales, preventing undesirable artifacts. For a given and normalized ray, we marginalize the Gaussian 3D distribution on a plane orthogonal to the ray and intersecting

https://github.com/facebookresearch/vkraygs

its point of maximum density. This yields a 2D Gaussian distribution which can be locally smoothed to approximate the integral over the pixel area, and to further compute the rendered opacity at each point in the pixel area. Besides the theoretically correct solution, we derive an approximated one that can be efficiently integrated into our renderer.

To summarize, our work proposes a mathematically rigorous and geometrically intuitive derivation for deriving all quantities required for high-quality and fast, hardware rasterized, ray-based Gaussian Splatting. We additionally introduce a solution for handling MIP-related issues in RayGS, demonstrated by qualitative and side-by-side comparisons. Finally, we provide quantitative evaluation results, demonstrating that we can retain state-of-the-art appearance performance of RayGS-based models while obtaining on average approximately $40\times$ faster rendering performance on scenes from the MipNeRF360 [1] and Tanks&Temples [13] benchmark datasets.

2. Related Works

3D Gaussian Splatting (3DGS) was initially presented in [12], and has since become a fundamental tool in Computer Vision and Graphics. Thanks to its speed and ease of use, 3DGS has been applied to a wide variety of downstream tasks, including text-to-3D generation [3, 29, 37], photo-realistic human avatars [14, 15, 27, 41], dynamic scene modeling [16, 32, 35], Simultaneous Localization and Mapping [11, 18, 34, 40], and more [8, 33, 36]. In this section, we focus on two areas of research that are most closely related to our work: re-formulating 3DGS as ray casting, and improving its rendering performance.

3DGS as ray casting. A few different works [9, 10, 17, 20, 26, 39] have shown that rendering 3D gaussian primitives using ray casting can be a preferable alternative to splatting. These work concurrently showed that ray-splat intersections can be calculated efficiently and, importantly, *exactly*, as opposed to the approximation introduced by the original rasterization formulation in [12]. The works in [39] and [10] steer the training process towards learning 3D representations that more accurately follow the real geometry of the scene, by introducing regularization losses that exploit the more meaningful depth and normals that can be computed with the ray-splat intersection formulation. Similarly, the works in [26] and [9] exploit ray-splat intersection to compute per-pixel depth values, that can be used to locally re-order the splats and avoid "popping" artifacts. While the previous works implemented ray casting in the same CUDA software rasterization framework of [12], a few others [17, 20] exploited Nvidia hardware to implement 3DGS rendering as a full ray-tracing procedure. While generally slower than the others, these approaches unlock an entire new range of possibilities, *e.g.*, physically accurate simulations of reflections and shadows, by tracing light propagation through a 3DGS scene.

Improving 3DGS performance. One of the main advantages of 3DGS compared to previous photorealistic 3D reconstruction approaches such as Neural Radiance Fields (NeRF) [19], is its render-time performance. Even when compared to NeRF approaches specifically tuned for speed over quality [22], 3DGS can still run up to one order of magnitude faster. Nonetheless, real-time rendering complex 3DGS scenes using the original CUDA implementation from [12] can be infeasible when very high output resolutions are required, if computational budget is limited, or both (*e.g.* on VR headsets). Because of this, optimizing 3DGS rendering performance has been an active area of research in the past years, focusing on two main directions: model pruning and compression [4, 21, 24, 25], and fine-tuning the rendering logic [5, 6, 30].

In Radsplat [25], Niemeyer *et al.* propose a strategy to prune splats that don't significantly contribute to image quality, considerably reducing how many need to be rendered and thus increasing speed. Since 3DGS is generally bottlenecked by GPU memory bandwidth, model compression can be exploited to both reduce model storage size and increase rendering speed, *e.g.* by organizing splat parameters in coherent 2D grids to be compressed using standard image compression algorithms [21], or by developing specific parameter quantization approaches [4, 24].

In FlashGS [6], Feng *et al.* present an in-depth analysis of the original differentiable 3DGS CUDA renderer, proposing many small optimizations which together contribute substantial speed improvements, particularly at training time. The largest increase in rendering performance, however, can generally be achieved by abandoning the CUDA-based software rasterization paradigm (and thus the ability to differentiate through the renderer) in favour of hardware rasterization, in a way reminiscent of older works on rendering quadratic 3D surfaces [28, 31]. To the best of our knowledge, this approach to 3DGS rendering has not been formally described in computer vision literature, but many different HW-rasterization implementations of 3DGS are available as open source software, such as [5, 30].

3. Preliminaries: Gaussian Splatting

We provide a brief introduction to Gaussian Splatting (GS) [12] and its ray-based variant (RayGS) [39], including implementation details of hardware-rasterized GS.

Scene representation. Gaussian Splatting introduces a scene representation expressed in terms of (3D Gaussian) *primitives* $\mathcal{S} \coloneqq \{(\boldsymbol{\mu}_i, \Sigma_i, o_i, \boldsymbol{\xi}_i)\}_{i=1}^{N}$, each consisting of a center $\boldsymbol{\mu}_i \in \mathbb{R}^3$, a covariance matrix $\Sigma_i \in \mathbb{R}^{3\times3}$, a *prior* opacity scalar $o_i \in [0, 1]$ and a feature vector $\boldsymbol{\xi}_i \in \mathbb{R}^d$ (*e.g.* RGB color). The covariance matrix Σ_i is typically

parametrized with a rotation $\mathtt{R}_i$ and a positive-definite, diagonal matrix $\mathtt{S}_i$ as follows: $\Sigma_i \coloneqq \mathtt{R}_i \mathtt{S}_i^2 \mathtt{R}_i^\top$. We assume center and covariance to be expressed in the camera frame.

Scene rendering. Rendering a scene $\mathcal{S}$ on a given camera is formulated as a per-pixel, convex linear combination of primitives' features, *i.e*.

$$\mathcal{R}(\boldsymbol{x}; \mathcal{S}) \coloneqq \sum_{i=1}^{N} \boldsymbol{\xi}_{\nu_i} \omega_{\nu_i}(\boldsymbol{x}; \mathcal{S}) \prod_{j=1}^{i-1} [1 - \omega_{\nu_j}(\boldsymbol{x}; \mathcal{S})] \,, \quad (1)$$

where ν is a permutation of primitives that depends on the camera pose and camera ray $\boldsymbol{x}$, typically yielding an ascending ordering with respect to depth of the primitive's center. The term $\omega_i(\boldsymbol{x}; \mathcal{S}) \in [0, 1]$ can be regarded as the *rendered* primitive opacity value, which is given by

$$\omega_i(\boldsymbol{x}; \mathcal{S}) \coloneqq o_i \exp\left[-\frac{1}{2} \mathcal{D}(\boldsymbol{x}; \boldsymbol{\mu}_i, \Sigma_i)\right] \,, \quad (2)$$

where $\mathcal{D}(\boldsymbol{x}; \mu_i, \Sigma_i)$ is a divergence of $\boldsymbol{x}$ from the primitive. This divergence takes different forms depending on the type of model we consider, namely GS or RayGS.

Support of a primitive and its boundary. Given a primitive $(\boldsymbol{\mu}, \Sigma, o, \boldsymbol{\xi})$, the set of camera rays $\boldsymbol{x}$ for which the rendered opacity as per Eq. (2) is above a predefined, cut-off probability value p_{min} is called the *support* of the primitive. The support is camera-specific and can be characterized in terms of $\mathcal{D}$ as the set of rays satisfying

$$\mathcal{D}(\boldsymbol{x}; \boldsymbol{\mu}, \Sigma) \leq \kappa \,, \quad (3)$$

where $\kappa \coloneqq -2 \log\left(\frac{p_{\text{min}}}{o}\right)$. Indeed, the relation holds if and only if $\omega_i(\boldsymbol{x}; \mathcal{S}) \geq p_{\text{min}}$. The set of rays for which equality holds in Eq. (3) forms the *boundary* of the primitive's support. Finally, if $\kappa \leq 0$, the support of the primitive is a null set and, hence, the primitive can be skipped since it is not visible given the provided cut-off probability. For this reason, we assume $\kappa > 0$ in the rest of the paper.

3.1. Gaussian Splatting

Let $\pi(\boldsymbol{x})$ be the camera projection function mapping a 3D point in camera space to the corresponding 2D pixel in image space. In GS the divergence is given by

$$\mathcal{D}_{\text{gs}}(\boldsymbol{x}; \boldsymbol{\mu}, \Sigma) \coloneqq (\pi(\boldsymbol{x}) - \pi(\boldsymbol{\mu}))^\top \Sigma_\pi^{-1} (\pi(\boldsymbol{x}) - \pi(\boldsymbol{\mu})) \,, \quad (4)$$

where $\Sigma_\pi \coloneqq \mathtt{J}_\pi \Sigma \mathtt{J}_\pi^\top \in \mathbb{R}^{2\times 2}$ with $\mathtt{J}_\pi \in \mathbb{R}^{2\times 3}$ being the Jacobian of the projection function π evaluated at $\boldsymbol{\mu}$. This is the Mahalanobis distance between the pixel corresponding to $\boldsymbol{x}$ and the 2D Gaussian distribution that is obtained from the primitive's 3D Gaussian distribution transformed via a linerization of π at $\boldsymbol{\mu}$. One advantage of this approximation is that the support of the primitive spans a 2D ellipse in pixel-space and can be rasterized in hardware. The disadvantage is that the support is misplaced with respect to the 3D Gaussian density, yielding unexpected artifacts, and the approximation assumes a pinhole camera model.

3.2. Ray-Based Gaussian Splatting

RayGS improves the rendering quality by dropping the approximation in GS due to the local linearization, which causes unexpected behavior (see Fig. 5). The idea is to render a primitive by considering the point of maximum Gaussian density along each camera ray. By doing so, the density of the rendered primitive behaves as expected, but the computational cost is higher. The divergence function underlying RayGS takes the following form

$$\mathcal{D}_{\text{ray}}(\boldsymbol{x}; \boldsymbol{\mu}, \Sigma) \coloneqq (\tau(\boldsymbol{x})\boldsymbol{x} - \boldsymbol{\mu})^\top \Sigma^{-1} (\tau(\boldsymbol{x})\boldsymbol{x} - \boldsymbol{\mu}) \,, \quad (5)$$

where $\tau(\boldsymbol{x}) \coloneqq \frac{\boldsymbol{x}^\top \Sigma^{-1} \boldsymbol{\mu}}{\boldsymbol{x}^\top \Sigma^{-1} \boldsymbol{x}}$. Geometrically, $\tau(\boldsymbol{x})\boldsymbol{x}$ is the point of maximum density along ray $\boldsymbol{x} \in \mathbb{R}^3 \setminus \{\mathbf{0}\}$ (see Prop. A.1), and the divergence corresponds to the Mahalanobis distance between this point and the primitive's 3D Gaussian.

Skipping cases. Primitives for which $c^2 \leq \kappa$ holds with

$$c \coloneqq \sqrt{\boldsymbol{\mu}^\top \Sigma^{-1} \boldsymbol{\mu}} \quad (6)$$

have a support that spans the entire image (see Prop. A.2). This intuitively happens because the camera is positioned inside the primitive. For this reason, we skip those cases and, therefore, we assume $c^2 > \kappa$ in the rest of the section.

3.3. Hardware-Rasterized GS

One advantage of GS over RayGS is the straightforward mapping of the algorithm onto a traditional hardware-accelerated rasterization pipeline with programmable vertex and fragment shading stages. The idea is simple. Since the support of a primitive yields a 2D ellipse on the image plane, it is possible to enclose it with a *quad*, *i.e*. a 4-sided polygon, spanning a minimum area. The role of the vertex shader is to compute the positions of the quad vertices and initialize vertex-specific features that are interpolated and transformed by the fragment shader to deliver a per-pixel RGBA color. A standard alpha-blending pipeline configuration combines the RGBA color from multiple pre-sorted quads to mimic the actual rendering equation in Eq. (1). Below, we review the vertex and fragment shaders and refer the reader to the code of [30] for more details.

Vertex shader. Given the eigendecomposition, $\Sigma_\pi = \mathtt{U}_{gs} \Lambda_{gs} \mathtt{U}_{gs}^\top$, the vertices of the quad enclosing a primitive's support in image-space are given by

$$\mathtt{V}_{\text{gs}} \coloneqq \mathtt{T}_{\text{gs}} \mathcal{H}(\mathtt{Z}_{\text{gs}}) \,, \quad (7)$$

where $\mathtt{T}_{\text{gs}} \coloneqq \begin{bmatrix} \mathtt{U}_{\text{gs}} \Lambda_{\text{gs}}^{\frac{1}{2}} & \pi(\boldsymbol{\mu}) \end{bmatrix} \in \mathbb{R}^{2\times 3}$, function $\mathcal{H}$ turns each column of the argument matrix into homogeneous coordinates, $\mathtt{Z}_{\text{gs}} \coloneqq \sqrt{\kappa}\mathtt{O} \in \mathbb{R}^{2\times 4}$ and $\mathtt{O} \coloneqq \begin{bmatrix} -1 & -1 & 1 & 1 \\ -1 & 1 & 1 & -1 \end{bmatrix}$ are the vertices of the *canonical quad*, namely a centered 2D square.

Fragment shader. The goal of the fragment shader is to compute the rendered primitive's opacity in Eq. (2) by using hardware interpolation capabilities over vertex-specific quantities. Here, the relevant part of the opacity computation is $\mathcal{D}_{\text{gs}}(\boldsymbol{x};\boldsymbol{\mu},\Sigma)$, which changes over pixels. Given any ray $\boldsymbol{x}$ intersecting the primitive's quad, there exist interpolating coefficients $\boldsymbol{\alpha} \in \mathbb{R}^4$ such that $\mathcal{D}_{\text{gs}}(\boldsymbol{x};\boldsymbol{\mu},\Sigma) = \mathcal{D}_{\text{gs}}(\mathcal{H}(\mathtt{V}_{\text{gs}}\boldsymbol{\alpha}))$. Moreover, for any such $\boldsymbol{\alpha}$ we have that

$$\mathcal{D}_{\text{gs}}(\mathcal{H}(\mathtt{V}_{\text{gs}}\boldsymbol{\alpha})) = \|\mathtt{Z}_{\text{gs}}\boldsymbol{\alpha}\|^2 .$$

Accordingly, it is sufficient to interpolate $\mathtt{Z}_{\text{gs}}$ and use a simple dot product in the fragment shader to compute $\mathcal{D}_{\text{gs}}(\boldsymbol{x};\boldsymbol{\mu},\Sigma)$ over the quad area. The resulting quantity is then fed to a pixel-independent function to get the rendered opacity as per Eq. (2).

4. Hardware-Rasterized RayGS

In this section, we introduce the main contribution of the paper, namely showing how hardware rasterization can be used to efficiently render Gaussian primitives under a ray-based formulation. This allows to retain the advantage of GS, *i.e.* faster rendering, and the better quality of RayGS, due to the dropped linear approximation.

Thanks to the linear approximation of the projection function, the support of a primitive in GS spans a 2D ellipse on the image plane and an optimal enclosing quad can be efficiently computed in the same space, as shown in Sec. 3.3. However, when it comes to RayGS, where we drop this approximation, the support of primitives can also span half-hyperbolas making the approximation with quads *on the image plane* more complex and less efficient.

To sidestep this limitation, we drop the restriction of seeking quads on the image plane and approximate the support of primitives with quads placed directly in 3D space. In fact, any quad intersecting all camera rays in the support of a primitive is a valid solution and there are infinitely-many ones. But, two valid quads do not necessarily share the same computational efficiency, so making the right choice here makes the difference.

In this section, we present a solution strategy using quads lifted in 3D space that can be computed efficiently as we will show later in the experimental section. Mimicking Sec. 3.3, we discuss how the vertex and fragment shaders are implemented in our solution, which are the distinctive parts of our contribution, while we omit the rest of the pipeline (*e.g.* alpha-blending logic), for it is shared with the GS hardware-rasterized implementation [30].

4.1. Vertex shader

Consider a primitive with center $\boldsymbol{\mu}$ and covariance $\Sigma := \mathtt{R}_p\mathtt{S}^2\mathtt{R}_p^\top$ that factorizes in terms of the scale matrix $\mathtt{S}$ and rotation matrix $\mathtt{R}_p$. For a given camera view, we compute a quad that encloses the 3D points of maximum Gaussian density that we find along rays belonging to the boundary of the primitive's support, *i.e.* the points belonging to the following set (see, Fig. 1a):

$$\mathcal{E} := \left\{\tau(\boldsymbol{x})\boldsymbol{x} \,:\, \mathcal{D}_{\text{ray}}(\boldsymbol{x};\boldsymbol{\mu},\Sigma) = \kappa,\, \boldsymbol{x} \in \mathbb{R}^3 \setminus \{\mathbf{0}\}\right\} .$$

Determining such a quad is possible because $\mathcal{E}$ forms a 2D ellipse embedded in 3D space and, hence, is isomorphic to the unit circle $\mathbb{S}_1$. To grasp the geometrical intuition of why this is the case, we describe how we can map $\mathcal{E}$ to $\mathbb{S}_1$ and provide in Fig. 1 a schematic overview.

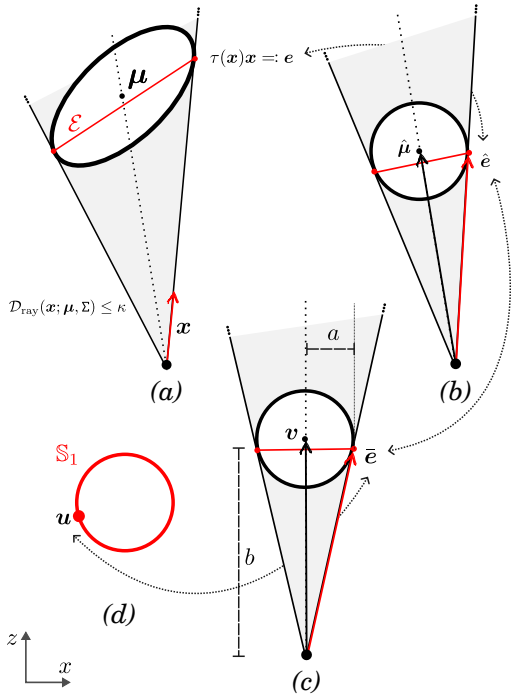

Figure 1. Schematic overview of the isomorphism between $\mathcal{E}$ and the unit circle $\mathbb{S}_1$, shown from the (x, z)-plane perspective.

Isomorphism Φ between $\mathcal{E}$ and $\mathbb{S}_1$. We start with the primitive in its original space in Fig. 1a. By Prop. A.5 we have that

$$\boldsymbol{e}^\top\Sigma^{-1}\boldsymbol{e} = \boldsymbol{e}^\top\Sigma^{-1}\boldsymbol{\mu} = c^2 - \kappa\,, \tag{8}$$

holds for all $\boldsymbol{e} \in \mathcal{E}$ with c as defined in Eq. (6). By setting $\hat{\boldsymbol{\mu}} := \frac{1}{c}\mathtt{S}^{-1}\mathtt{R}_p^\top\boldsymbol{\mu}$ and $\hat{\boldsymbol{e}} := \frac{1}{\sqrt{c^2-\kappa}}\mathtt{S}^{-1}\mathtt{R}_p^\top\boldsymbol{e}$, we can rewrite the right-most equality in Eq. (8) as

$$\hat{\boldsymbol{e}}^\top\hat{\boldsymbol{\mu}} = b := \sqrt{1 - \frac{\kappa}{c^2}}\,. \tag{9}$$

The result of this space transformation is shown in Fig. 1b. Since both $\hat{\boldsymbol{e}}$ and $\hat{\boldsymbol{\mu}}$ are points of the unit sphere $\mathbb{S}_2$, we have that all $\hat{\boldsymbol{e}}$ satisfying Eq. (9) live on a circle embedded in 3D space. To make this mapping explicit, we rotate the space to align $\hat{\boldsymbol{\mu}}$ with the z-axis $\boldsymbol{v} = \begin{bmatrix}0 & 0 & 1\end{bmatrix}^\top$. To this end,

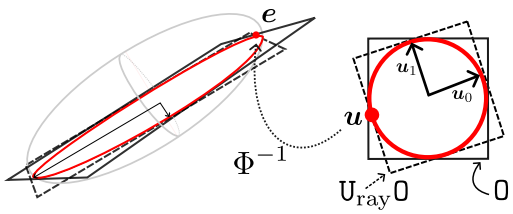

Figure 2. Examples of 3D quads obtained by mapping 2D squares via the isomorphism Φ^{-1}.

we define a rotation matrix $\mathtt{R}_{\hat{\boldsymbol{\mu}}\leftarrow\boldsymbol{v}}$ (see, Appendix A.1) such that $\hat{\boldsymbol{\mu}} = \mathtt{R}_{\hat{\boldsymbol{\mu}}\leftarrow\boldsymbol{v}}\boldsymbol{v}$. By setting $\bar{\boldsymbol{e}} \coloneqq \mathtt{R}_{\hat{\boldsymbol{\mu}}\leftarrow\boldsymbol{v}}\hat{\boldsymbol{e}}$, we have that

$$\bar{\boldsymbol{e}}^\top \boldsymbol{v} = b\,. \tag{10}$$

This transformation is depicted in Fig. 1c. Since $\bar{\boldsymbol{e}}$ is still a vector of the unit sphere $\mathbb{S}_2$, and by Eq. (10) its z-coordinate has to be b, we have that

$$\bar{\boldsymbol{e}} = b\mathcal{H}\left(a\boldsymbol{u}\right) \tag{11}$$

holds with $a \coloneqq \sqrt{\frac{\kappa}{c^2-\kappa}}$ and for a specific element $\boldsymbol{u} \in \mathbb{S}_1$ of the unit circle, which corresponds to the (x, y)-subvector of $\bar{\boldsymbol{e}}$ normalized to unit length (see Fig. 1d). In summary, we have described a transformation chain

$$\boldsymbol{e} \in \mathcal{E} \rightleftarrows \hat{\boldsymbol{e}} \in \mathbb{S}_2 \rightleftarrows \bar{\boldsymbol{e}} \in \mathbb{S}_2 \rightleftarrows \boldsymbol{u} \in \mathbb{S}_1$$

mapping elements of $\mathcal{E}$ to the unit circle, which is both linear and invertible, thus showing that the two sets are isomorphic. From this we infer that all points in $\mathcal{E}$ actually live on a 2D ellipse embedded in 3D space and, therefore, can be enclosed with a quad placed on the same 3D plane the 2D ellipse belongs to. We denote by Φ the isomorphism from $\mathcal{E}$ to $\mathbb{S}_1$. The 3D center of the ellipse spanned by $\mathcal{E}$ can be computed as $\Phi^{-1}(\mathbf{0})$, *i.e.* by back-mapping the center of the unit disk $\mathbf{0} \in \mathbb{R}^2$.

Quad vertices and optimality. Determining the vertices of a quad enclosing $\mathcal{E}$ becomes easy given the mapping Φ, because it is sufficient to enclose the unit circle $\mathbb{S}_1$ with a 2D quad and map it back to 3D space with Φ^{-1} (see, Fig. 2a). The explicit form of Φ^{-1} can be obtained by traversing backwards the transformations from the previous paragraph, with some terms rearranged:

$$\Phi^{-1}(\boldsymbol{u}) \coloneqq \frac{c^2-\kappa}{c} \underbrace{\mathtt{R}_p\mathtt{S}\mathtt{R}_{\hat{\boldsymbol{\mu}}\leftarrow\boldsymbol{v}}}_{=:\mathtt{Q}} \mathcal{H}(a\boldsymbol{u})\,. \tag{12}$$

Unfortunately, there are infinitely-many ways we can approximate the unit circle with a quad, two examples being given in Fig. 2, but not all are efficient for the sake of rendering. In particular, their projection on the image plane can span different area sizes, potentially introducing a waste of compute on irrelevant pixels. Ideally, we would like to position the quad in a way to minimize the spanned area *on the image plane*, but this requires additional complexity in the way the vertices are computed, to the detriment of the overall rendering speed. What we found out to be a good compromise is to position the vertices of the enclosing quad in a way to span the smallest area *in 3D space*, by forming a tight rectangle aligned with the 2D ellipse's axes (see Fig. 2b). To this end, we identify one vertex of the ellipse $\mathcal{E}$, *i.e.* one of the two endpoints along its major axis, by localizing it first on the unit circle. This is achieved by solving the following optimization problem

$$\boldsymbol{u}_1 \in \arg\max_{\boldsymbol{u}\in\mathbb{S}_1} \|\Phi^{-1}(\boldsymbol{u}) - \Phi^{-1}(\mathbf{0})\|^2\,, \tag{13}$$

which finds the point $\boldsymbol{u}_1$ on the unit circle whose counterpart $\Phi^{-1}(\boldsymbol{u})$ on the ellipse $\mathcal{E}$ maximizes the distance to the ellipse's center, which can computed as $\Phi^{-1}(\mathbf{0})$. By substituting Eq. (12) into Eq. (13), and dropping constant scalar factors that do not change the maximizers, the objective takes a standard quadratic form (see, Appendix A.2):

$$\boldsymbol{u}_1 \in \arg\max_{\boldsymbol{u}\in\mathbb{S}_1} \boldsymbol{u}^\top \mathtt{B}\boldsymbol{u}\,. \tag{14}$$

Here, $\mathtt{B} \coloneqq \mathtt{Q}_{0:2}^\top\mathtt{Q}_{0:2}$, where $\mathtt{Q}_{0:2} \in \mathbb{R}^{3\times2}$ denotes $\mathtt{Q}$ restricted to the first two columns. The solution $\boldsymbol{u}_1$ is an eigenvector of $\mathtt{B}$ with maximum eigenvalue, which can be computed in closed-form for 2×2 matrices. Similarly, $\boldsymbol{u}_0$, *i.e.* the point corresponding to the minor axis, is an eigenvector of $\mathtt{B}$ with minimum eigenvalue. Since B has only two eigenvectors and are mutually orthogonal, $\boldsymbol{u}_0$ can be computed by simply rotating $\boldsymbol{u}_1$ by 90 degrees anticlockwise. By doing so we ensure to preserve a consistent orientation of the quad surface. We stack $\boldsymbol{u}_0$ and $\boldsymbol{u}_1$ to form a 2×2 matrix $\mathtt{U}_{\text{ray}} \coloneqq (\boldsymbol{u}_0, \boldsymbol{u}_1)$, which we use to rotate the vertices of the canonical quad $\mathtt{O}$ before mapping it back to 3D space via Φ^{-1} (see, Fig. 2). Although this mapping would already provide a valid 3D quad for rendering, we also scale it by the factor $\frac{c^2}{c^2-\kappa}$. This scaling operation preserves the support of camera rays, making the scaled quad equivalent to the original one from a rendering perspective, but it enables the computation of vertices in a more efficient and stable way. In fact, the final set of quad vertices $\mathtt{V}_{\text{ray}}$ can be computed as follows (see Appendix A.3), mimicking Eq. (7):

$$\mathtt{V}_{\text{ray}} \coloneqq \mathtt{T}_{\text{ray}}\mathcal{H}(\mathtt{Z}_{\text{ray}})\,, \tag{15}$$

where $\mathtt{Z}_{\text{ray}} \coloneqq \frac{\sqrt{\kappa}}{b}\mathtt{O}$ and $\mathtt{T}_{\text{ray}} \coloneqq \begin{bmatrix}\mathtt{Q}_{0:2}\mathtt{U}_{\text{ray}} & \boldsymbol{\mu}\end{bmatrix}$.

A note on near plane clipping. The proposed formulation works under the assumption that near plane clipping is disabled. Indeed, if a quad intersecting the near plane is clipped, we obtain undesired effects like visible discontinuities (see Fig. 3). Nonetheless, frustum culling of primitives based on a near plane is still applicable without consequences, for the whole quad is removed in that case.

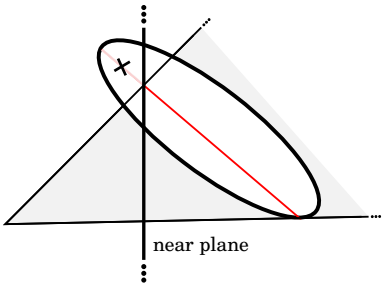

Figure 3. Undesired effects of near-plane clipping. Sharp discontinuities might be visible if a quad intersects the clipping plane.

4.2. Fragment shader

Our goal is to compute the rendered primitive's opacity in Eq. (2) by exploiting hardware interpolation of quantities specified at the quad's vertices. Again, the relevant part of the computation is $\mathcal{D}_{\text{ray}}(\boldsymbol{x};\boldsymbol{\mu},\Sigma)$ since it changes over pixels. Similarly to GS, we have that for any ray $\boldsymbol{x}$ intersecting the primitive's quad, there exist interpolating coefficients $\boldsymbol{\alpha}\in\mathbb{R}^4$ such that $\mathcal{D}_{\text{ray}}(\boldsymbol{x};\boldsymbol{\mu},\Sigma)=\mathcal{D}_{\text{ray}}(\mathtt{V}_{\text{ray}}\boldsymbol{\alpha};\boldsymbol{\mu},\Sigma)$. Moreover, for any such $\boldsymbol{\alpha}$ the following holds (see, Appendix A.4):

$$\mathcal{D}_{\text{ray}}(\mathtt{V}_{\text{ray}}\boldsymbol{\alpha};\boldsymbol{\mu},\Sigma)=\left\{c^{-2}+\|\mathtt{Z}_{\text{ray}}\boldsymbol{\alpha}\|^{-2}\right\}^{-1}. \tag{16}$$

Hence, it is sufficient to interpolate $\mathtt{Z}_{\text{ray}}$ and apply a simple scalar function to get $\mathcal{D}_{\text{gs}}(\boldsymbol{x};\boldsymbol{\mu},\Sigma)$ over the quad area. The resulting quantity is then fed to a pixel-independent function to get the rendered opacity as per Eq. (2).

5. MIP for RayGS

In this section we focus on MIP-related issues, which should be addressed to ensure higher-quality renderings in particular for VR applications, where we are free of moving in the scene. The problem arises from the fact that when we render we approximate a pixel with a single camera ray in the center, instead of considering the whole pixel area. Imagine to have a primitive that spans less than a pixel when rendered. If the center of the pixel overlaps with the support of the primitive, the full pixel will take the primitive's rendered color. Otherwise it will show the background color. We would instead expect the pixel color to be the combination of primitive and background color, depending on the size of the primitive's support.

In the context of GS, there have been works suggesting ways to address MIP issues [38], but to our knowledge no work has explicitly addressed the matter for RayGS. To fill this gap, we introduce a novel formulation that can be easily integrated into our fast renderer. Given a (normalized) ray $\boldsymbol{x}$, the idea is to marginalize the Gaussian 3D distribution on a plane that is orthogonal to $\boldsymbol{x}$ and passing through the point of maximum density $\tau(\boldsymbol{x})\boldsymbol{x}$. This yields a 2D Gaussian distribution on the same plane, which can be smoothed using a properly-sized isotropic 2D Gaussian distribution to approximate the integral over the pixel area. By applying this idea, we end up with the following per-primitive distribution over (normalized) camera rays (see, Appendix A.5 for a detailed derivation):

$$P_{\text{MIP}}(\boldsymbol{x})\propto\frac{\tau^2(\boldsymbol{x})}{\sqrt{|\hat{\Sigma}_x|\boldsymbol{x}^\top\hat{\Sigma}_x^{-1}\boldsymbol{x}}}\exp\left(-\frac{1}{2}\mathcal{D}_{\text{ray}}(\boldsymbol{x};\boldsymbol{\mu},\hat{\Sigma}_x)\right),$$

where $\hat{\Sigma}_x\coloneqq\Sigma+\sigma_x^2\tau(\boldsymbol{x})^2\mathtt{I}$ is a pixel-dependent 3D covariance matrix and σ_x^2 represents the area of the pixel at unit distance along the ray, which is in principle also dependent on the ray because the plane we are projecting onto is not necessarily parallel to the image plane. Akin to MIP-Splatting [38], we can compute the rendered opacity by considering the exponential term in $P_{\text{mip}}(\boldsymbol{x})$ and modulating the prior opacity so that the total opacity matches the one of standard RayGS, where the modulating factor is given by $\sqrt{\frac{|\Sigma|c^2}{|\hat{\Sigma}_x|\boldsymbol{x}^\top\hat{\Sigma}_x^{-1}\boldsymbol{x}}}$. The resulting formula is structurally similar to the one from [38], differences being that we do not require separate 2D and 3D filters and the modulation factor is pixel-dependent. Unfortunately, the latter fact, despite being theoretically more correct, poses challenges when it comes to having a fast rasterizer integrating it. For this reason, we introduce the following approximations in our implementation. We start considering σ_x constant and regard $\hat{\Sigma}$ as the resulting pixel-invariant counterpart of $\hat{\Sigma}_x$. Next, we assume $\boldsymbol{x}^\top\hat{\Sigma}^{-1}\boldsymbol{x}\approx\hat{c}^2\coloneqq\boldsymbol{\mu}^\top\hat{\Sigma}^{-1}\boldsymbol{\mu}$. This approximation is accurate when primitives are sufficiently small to have rays concentrated around the mean, which accounts for most of the cases. Conversely, when primitives are sufficiently large along both (x,y)-directions then the modulation factor is close to 1 for all $\boldsymbol{x}$, so the approximation still works. The cases when the approximation is less accurate are more rare, *i.e.* when primitives are thin less than a pixel upon projection, but sufficiently elongated to span a wide cone of rays. The final form of our MIP formulation is thus

$$\omega^{\text{mip}}(\boldsymbol{x})\approx o_i\sqrt{\frac{|\Sigma|c^2}{|\hat{\Sigma}|\hat{c}^2}}\exp\left(-\frac{1}{2}\mathcal{D}_{\text{ray}}(\boldsymbol{x};\boldsymbol{\mu},\hat{\Sigma})\right), \tag{17}$$

which can be efficiently integrated in our fast renderer.

6. Experiments

We have implemented our hardware-rasterized renderer for RayGS on top of VKGS [30], which is based on Vulkan. Accordingly, we refer our method to as *VKRayGS*. However, other OpenGL implementations are potentially possible based on what we described in the paper. Our experimental evaluation targets different goals:

Rendering speed. We want to show that our contribution unlocks significantly higher rendering speed for RayGS

models than the best publicly-available, open-sourced alternative, which at the time of writing is the CUDA-based renderer from Gaussian Opacity Field (GOF) [39]. Moreover, to put the numbers in the right perspective, we provide in Appendix B.1 a speed comparison between the original VKGS implementation of GS and the CUDA-based implementation of Gaussian Splatting (GS) from INRIA [7], in its most recent version integrating speed optimizations.

Rendering quality. The way primitives are rendered in our formulation is mathematically equivalent to [39], but there are still factors that can influence the final rendering quality, like differences occurring in the rest of the pipeline (*e.g.* how primitives are clipped). For this reason, we report quality metrics in addition to speed measurements. However, given that we use models trained with the differentiable renderer from the competitors, it is reasonable to expect a bias in their favor. Since quality drops that are not directly ascribable to our contribution are expected to show up also when comparing VKGS against its CUDA counterpart, we decided to include in Appendix B.1 also the latter comparison despite not being directly related to our contribution.

6.1. Evaluation Protocol

We evaluate the performance of our renderer on scenes from two benchmark datasets that have been often used in the context of novel-view synthesis with GS (see *e.g.* [2, 12]), namely MipNerf360 [1] and Tanks and Temples [13]. Those are real-world captures that span both indoor and outdoor scenarios. For each scene, every 8th image is set aside to form a test set, on which three quality metrics are reported, namely peak signal-to-noise ratio (PSNR), structural similarity (SSIM) and the perceptual metric from (LPIPS) using VGG. Following the standard protocol from [1], we evaluate MipNerf360 indoor/outdoor scenes using the $2\times$/$4\times$ downsampling factors, respectively. Moreover, we use downsampled images provided by the authors of the dataset. For Tanks&Temples we evaluate the results at $2\times$ downsampling factor, akin to previous methods. The same images used for quality evaluations are also used to compute the rendering speed in terms of Frames-Per-Second (FPS). We opted to run experiments on an RTX2080, which is a mid-range GPU, sufficiently powerful and CUDA-based to run the implementations from the competitors. Nevertheless, our implementation can run on any GPU supporting Vulkan and, more in general, enables implementations in OpenGL, which broadens up the applicability spectrum significantly. All scene models used for the experiments are either provided by the authors of the competing methods or, if not available, have been trained using the code they provide. Finally, the quantitative evaluations of our model are run with MIP disabled, because the scene models from [39] already incorporate an additive factor on the 3D covariance and are trained without the MIP opacity modulation. We nevertheless provide qualitative examples showing the importance of the MIP formulation.

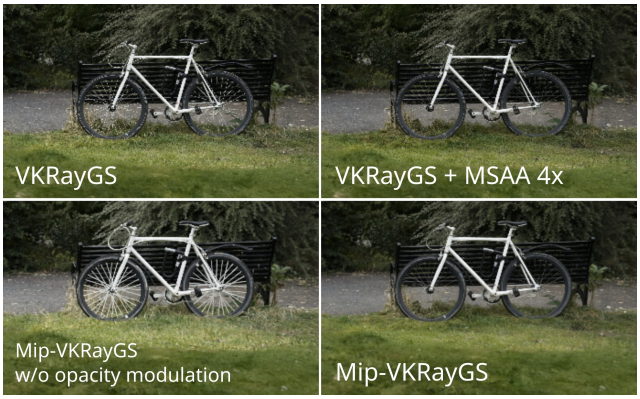

Figure 4. Benefits of our MIP formulation for RayGS. Best viewed with digital zoom. See text for details.

6.2. Results

Before delving into the results obtained by our fast renderer against GOF, and following our previous discussion about quality expectations, we discuss a quantitative comparison of VKGS against GS. In addition, we provide some qualitative results and discussion about MIP.

GS versus VKGS. In Appendix B.1, we report results obtained by GS versus the Vulkan counterpart VKGS on the benchmark datasets, which highlight a clear speed advantage of the Vulkan implementation over the CUDA-based one from GS, being on average $2\times$ faster. Quality metrics, instead, are not significantly different, excepting a few cases. However, the fact that there are differences indicates potential misalignment between the implementations and the results favor GS because the model has been trained with the same renderer.

GOF versus VKRayGS. In Tab. 1, we report the results obtained by our renderer against GOF. We have split the table into two sections to distinguish scenes from MipNerf360 (top) and Tanks&Temples (bottom). For each scene, we report the size of the model in terms of number of primitives N and report left-to-right speed comparisons in terms of FPS and quality metrics in terms of PSNR, SSIM and LPIPS, averaged over all the test views. we observe huge gains in terms of speed, with an average speed up of $40\times$, highlighting the relevance of our contribution, for it unlocks real-time rendering with the better RayGS models (as reflected by the overall worse perceptual metrics obtained with GS in Tab. B.1). If we shift our focus on the qualitative metrics, we observe a slight quality drop with some cases where our renderer delivers even better perceptual scores. This drop is motivated by implementation misalignments between GOF and VKRayGS that are not ascribable to our method, as confirmed by similar, if not larger, gaps

Figure 5. Artifacts one might experience with GS models (top) as opposed to RayGS (bottom), while moving close to objects. Renderings are from the *room* scene of MipNeRF360 and obtained with VKGS and our VKRayGS, respectively.

between VKGS and GS (see, Tab. B.1). Besides the quantitative analysis, qualitative comparisons are provided in Appendix C, where perceptual differences are barely visible.

on RTX2080 Scene	N	FPS ↑		PSNR ↑		SSIM ↑		LPIPS ↓	
		GOF	VKRayGS	GOF	VKRayGS	GOF	VKRayGS	GOF	VKRayGS
bicycle	5.35M	4	**177**	25.47	25.31	0.784	0.784	0.206	0.203
bonsai	1.07M	6	**341**	31.60	31.46	0.937	0.930	0.240	0.204
counter	0.82M	6	**292**	28.69	28.57	0.901	0.896	0.258	0.230
flowers	3.28M	6	**191**	21.67	21.61	0.632	0.631	0.309	0.310
garden	4.40M	4	**172**	27.46	27.34	0.866	0.865	0.122	0.117
kitchen	1.07M	5	**242**	30.74	30.63	0.915	0.911	0.168	0.153
room	1.09M	5	**305**	30.81	30.55	0.915	0.906	0.281	0.245
stump	4.97M	6	**186**	26.96	26.94	0.790	0.792	0.223	0.22
treehill	4.02M	5	**184**	22.40	22.48	0.638	0.636	0.325	0.326
barn	0.83M	10	**312**	28.99	28.30	0.892	0.883	0.190	0.192
caterpillar	1.43M	7	**247**	23.68	23.12	0.820	0.810	0.241	0.246
ignatius	2.60M	7	**203**	22.76	22.42	0.826	0.815	0.186	0.188
meetingroom	0.95M	7	**351**	25.50	24.71	0.881	0.867	0.235	0.230
truck	2.14M	7	**195**	25.80	25.30	0.892	0.881	0.153	0.141

Table 1. Comparison between GOF and VKRayGS on scenes from MipNeRF360 and Tanks&Temples.

Benefits of MIP. To understand why it is important to address MIP-related issues, in particular in real-time viewers, we provide in Fig. 4 an example from the MipNeRF360 *bicycle* scene. We show crops of the bike from images rendered by a far away camera. Top-left, we provide the rendering with plain VKRayGS, which exhibits strong aliasing artifacts. Top-right, we adopt a $4\times$ multi-sampling anti-aliasing (MSAA) approach, which consists in averaging 4 rays per pixel, but also this solution solves only partially the issue at a higher computational cost. Bottom-left, we show renderings from VKRayGS with our MIP formulation, when we set $\sigma^2 \coloneqq 0.1$ but neglect the opacity modulation factor. This solution solves the aliasing problem, but renders primitives unnaturally thick (see *e.g.* the bike wheel rays). Finally, bottom-right, we have VKRayGS with the full MIP formulation, which produces an antialiased output without artifacts at a negligible computational overhead.

GS versus RayGS. In Fig. 5 we highlight some artifacts that typically occur when using a GS renderer like VKGS as opposed to a RayGS one like ours. We show three frames of a camera that moves along a linear trajectory in the *room* scene of MipNeRF360. The camera moves close to objects in the scene on purpose, as this is typically the setting under which artifacts occur. On the top row, we present the results with VKGS, which exhibit spikes inconsistent with the scene geometry. On the bottom row, we report results obtained with VKRayGS, which despite being RayGS-based is executed on a GS scene model. As we can see, the artifacts afflicting the GS renderer are not there. This is because in RayGS models the rendered opacity is geometrically more consistent as opposed to GS.

7. Conclusions

We have presented a novel approach to rendering ray-based 3D Gaussian Splatting (RayGS) using hardware rasterization, achieving both fast and high-quality results for novel view synthesis. Our method leverages the advantages of RayGS, which provides superior quality compared to traditional 3DGS, while obtaining significant rendering speed gains on all tested scenes. We have demonstrated that our approach can render high-quality images at frame rates suitable for VR and MR applications. Our contributions include a mathematically rigorous and geometrically intuitive derivation of how to efficiently estimate all relevant quantities for rendering of RayGS models, as well as a solution to MIP-related issues in a RayGS formulation, enabling alias-free rendering of scenes at diverging test and training scales.

Our work has shown how to significantly speedup RayGS models at test time, but it would be interesting to employ hardware rasterization to improve training time as well. It is not trivial how this can be achieved and we leave this to future work.

References

[1] Jonathan T Barron, Ben Mildenhall, Dor Verbin, Pratul P Srinivasan, and Peter Hedman. Mip-nerf 360: Unbounded anti-aliased neural radiance fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5470–5479, 2022. 2, 7

[2] Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. Revising densification in gaussian splatting. In *European Conference on Computer Vision*, 2024. 1, 7

[3] Zilong Chen, Feng Wang, and Huaping Liu. Text-to-3d using gaussian splatting. *arXiv preprint arXiv:2309.16585*, 2023. 2

[4] Zhiwen Fan, Kevin Wang, Kairun Wen, Zehao Zhu, Dejia Xu, and Zhangyang Wang. Lightgaussian: Unbounded 3d gaussian compression with 15x reduction and 200+ fps. *arXiv preprint arXiv:2311.17245*, 2023. 2

[5] fast-gauss. Fast gaussian rasterization. GitHub, 2024. 1, 2

[6] Guofeng Feng, Siyan Chen, Rong Fu, Zimu Liao, Yi Wang, Tao Liu, Zhilin Pei, Hengjie Li, Xingcheng Zhang, and Bo Dai. Flashgs: Efficient 3d gaussian splatting for large-scale and high-resolution rendering. *arXiv preprint arXiv:2408.07967*, 2024. 1, 2

[7] gs-inria. INRIA Gaussian Splatting. https://github.com/graphdeco-inria/gaussian-splatting, 2024. 7

[8] Antoine Guédon and Vincent Lepetit. Sugar: Surface-aligned gaussian splatting for efficient 3d mesh reconstruction and high-quality mesh rendering. *arXiv preprint arXiv:2311.12775*, 2023. 2

[9] Florian Hahlbohm, Fabian Friederichs, Tim Weyrich, Linus Franke, Moritz Kappel, Susana Castillo, Marc Stamminger, Martin Eisemann, and Marcus Magnor. Efficient perspective-correct 3d gaussian splatting using hybrid transparency. *arXiv preprint arXiv:2410.08129*, 2024. 1, 2

[10] Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. 2d gaussian splatting for geometrically accurate radiance fields. In *ACM SIGGRAPH 2024 Conference Papers*, pages 1–11, 2024. 1, 2

[11] Nikhil Keetha, Jay Karhade, Krishna Murthy Jatavallabhula, Gengshan Yang, Sebastian Scherer, Deva Ramanan, and Jonathon Luiten. Splatam: Splat, track & map 3d gaussians for dense rgb-d slam. *arXiv preprint arXiv:2312.02126*, 2023. 2

[12] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42 (4), 2023. 1, 2, 7

[13] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: Benchmarking large-scale scene reconstruction. *ACM Transactions on Graphics (ToG)*, 36 (4):1–13, 2017. 2, 7

[14] Muhammed Kocabas, Jen-Hao Rick Chang, James Gabriel, Oncel Tuzel, and Anurag Ranjan. Hugs: Human gaussian splats. *arXiv preprint arXiv:2311.17910*, 2023. 2

[15] Jiahui Lei, Yufu Wang, Georgios Pavlakos, Lingjie Liu, and Kostas Daniilidis. Gart: Gaussian articulated template models. *arXiv preprint arXiv:2311.16099*, 2023. 2

[16] Jonathon Luiten, Georgios Kopanas, Bastian Leibe, and Deva Ramanan. Dynamic 3d gaussians: Tracking by persistent dynamic view synthesis. *arXiv preprint arXiv:2308.09713*, 2023. 2

[17] Alexander Mai, Peter Hedman, George Kopanas, Dor Verbin, David Futschik, Qiangeng Xu, Falko Kuester, Jon Barron, and Yinda Zhang. Ever: Exact volumetric ellipsoid rendering for real-time view synthesis. *arXiv preprint arXiv:2410.01804*, 2024. 2

[18] Hidenobu Matsuki, Riku Murai, Paul HJ Kelly, and Andrew J Davison. Gaussian splatting slam. *arXiv preprint arXiv:2312.06741*, 2023. 2

[19] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020. 1, 2

[20] Nicolas Moenne-Loccoz, Ashkan Mirzaei, Or Perel, Riccardo de Lutio, Janick Martinez Esturo, Gavriel State, Sanja Fidler, Nicholas Sharp, and Zan Gojcic. 3D Gaussian Ray Tracing: Fast tracing of particle scenes. *ACM Transactions on Graphics and SIGGRAPH Asia*, 2024. 2

[21] Wieland Morgenstern, Florian Barthel, Anna Hilsmann, and Peter Eisert. Compact 3d scene representation via self-organizing gaussian grids. *arXiv preprint arXiv:2312.13299*, 2023. 2

[22] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):102:1–102:15, 2022. 2

[23] KL Navaneet, Kossar Pourahmadi Meibodi, Soroush Abbasi Koohpayegani, and Hamed Pirsiavash. Compact3d: Smaller and faster gaussian splatting with vector quantization. *arXiv preprint arXiv:2311.18159*, 2023. 1

[24] Simon Niedermayr, Josef Stumpfegger, and Rüdiger Westermann. Compressed 3d gaussian splatting for accelerated novel view synthesis. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10349–10358, 2024. 2

[25] Michael Niemeyer, Fabian Manhardt, Marie-Julie Rakotosaona, Michael Oechsle, Daniel Duckworth, Rama Gosula, Keisuke Tateno, John Bates, Dominik Kaeser, and Federico Tombari. Radsplat: Radiance field-informed gaussian splatting for robust real-time rendering with 900+ fps. *arXiv preprint arXiv:2403.13806*, 2024. 2

[26] Lukas Radl, Michael Steiner, Mathias Parger, Alexander Weinrauch, Bernhard Kerbl, and Markus Steinberger. Stopthepop: Sorted gaussian splatting for view-consistent real-time rendering. *ACM Transactions on Graphics (TOG)*, 43(4):1–17, 2024. 1, 2

[27] Shunsuke Saito, Gabriel Schwartz, Tomas Simon, Junxuan Li, and Giljoo Nam. Relightable gaussian codec avatars. *arXiv preprint arXiv:2312.03704*, 2023. 2

[28] Christian Sigg, Tim Weyrich, Mario Botsch, and Markus H Gross. Gpu-based ray-casting of quadratic surfaces. In *PBG@ SIGGRAPH*, pages 59–65, 2006. 2

[29] Jiaxiang Tang, Jiawei Ren, Hang Zhou, Ziwei Liu, and Gang Zeng. Dreamgaussian: Generative gaussian splatting for effi-

cient 3d content creation. *arXiv preprint arXiv:2309.16653*, 2023. 2

[30] vkgs. VulKan Gaussian Splatting. https://github.com/jaesung-cs/vkgs, 2024. 1, 2, 3, 4, 6

[31] Tim Weyrich, Simon Heinzle, Timo Aila, Daniel B Fasnacht, Stephan Oetiker, Mario Botsch, Cyril Flaig, Simon Mall, Kaspar Rohrer, Norbert Felber, et al. A hardware architecture for surface splatting. *ACM Transactions on Graphics (TOG)*, 26(3):90–es, 2007. 2

[32] Guanjun Wu, Taoran Yi, Jiemin Fang, Lingxi Xie, Xiaopeng Zhang, Wei Wei, Wenyu Liu, Qi Tian, and Xinggang Wang. 4d gaussian splatting for real-time dynamic scene rendering. *arXiv preprint arXiv:2310.08528*, 2023. 2

[33] Tianyi Xie, Zeshun Zong, Yuxin Qiu, Xuan Li, Yutao Feng, Yin Yang, and Chenfanfu Jiang. Physgaussian: Physics-integrated 3d gaussians for generative dynamics. *arXiv preprint arXiv:2311.12198*, 2023. 2

[34] Chi Yan, Delin Qu, Dong Wang, Dan Xu, Zhigang Wang, Bin Zhao, and Xuelong Li. Gs-slam: Dense visual slam with 3d gaussian splatting. *arXiv preprint arXiv:2311.11700*, 2023. 2

[35] Zeyu Yang, Hongye Yang, Zijie Pan, Xiatian Zhu, and Li Zhang. Real-time photorealistic dynamic scene representation and rendering with 4d gaussian splatting. *arXiv preprint arXiv:2310.10642*, 2023. 2

[36] Mingqiao Ye, Martin Danelljan, Fisher Yu, and Lei Ke. Gaussian grouping: Segment and edit anything in 3d scenes. *arXiv preprint arXiv:2312.00732*, 2023. 2

[37] Taoran Yi, Jiemin Fang, Guanjun Wu, Lingxi Xie, Xiaopeng Zhang, Wenyu Liu, Qi Tian, and Xinggang Wang. Gaussiandreamer: Fast generation from text to 3d gaussian splatting with point cloud priors. *arXiv preprint arXiv:2310.08529*, 2023. 2

[38] Zehao Yu, Anpei Chen, Binbin Huang, Torsten Sattler, and Andreas Geiger. Mip-splatting: Alias-free 3d gaussian splatting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 19447–19456, 2024. 6

[39] Zehao Yu, Torsten Sattler, and Andreas Geiger. Gaussian opacity fields: Efficient and compact surface reconstruction in unbounded scenes. *arXiv preprint arXiv:2404.10772*, 2024. 1, 2, 7

[40] Vladimir Yugay, Yue Li, Theo Gevers, and Martin R Oswald. Gaussian-slam: Photo-realistic dense slam with gaussian splatting. *arXiv preprint arXiv:2312.10070*, 2023. 2

[41] Wojciech Zielonka, Timur Bagautdinov, Shunsuke Saito, Michael Zollhöfer, Justus Thies, and Javier Romero. Drivable 3d gaussian avatars. *arXiv preprint arXiv:2311.08581*, 2023. 2