This CVPR paper is the Open Access version, provided by the Computer Vision Foundation.
Except for this watermark, it is identical to the accepted version;
the final published version of the proceedings is available on IEEE Xplore.

MeshSplatting: Differentiable Rendering with Opaque Meshes

Jan Held[1,2] Sanghyun Son[3] Renaud Vandeghen[1]
Daniel Rebain[4] Matheus Gadelha[6] Yi Zhou[6] Anthony Cioppa[1]
Ming C. Lin[3] Marc Van Droogenbroeck[1] Andrea Tagliasacchi[2,5,7]

[1]University of Liège [2]Simon Fraser University [3]University of Maryland
[4]University of British Columbia [5]University of Toronto [6]Adobe Research [7]Wayve

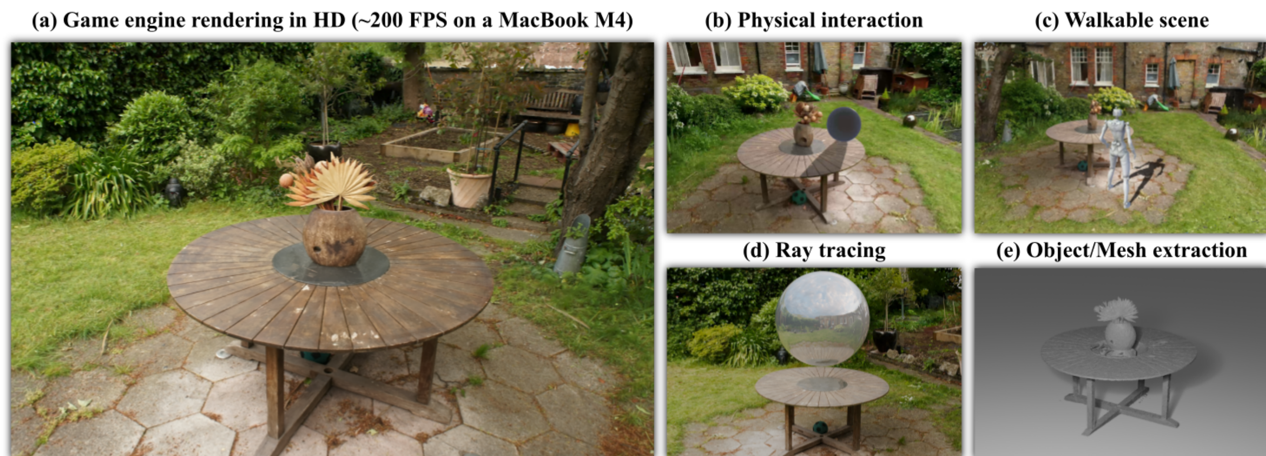

Figure 1. **MeshSplatting** produces a *connected mesh* composed only of *opaque triangles*, achieving high-quality novel view synthesis through end-to-end optimization, with a 2× training speed-up and 2× lower memory usage over current state-of-the-art methods. (a) Our representation is compatible with standard game engines, requiring no a-posteriori conversion and/or custom rendering routines for transparency, and natively supports (b) physical interactions, (c) interactive walkthroughs, and (d) ray tracing. (e) MeshSplatting enables straightforward object extraction, allowing scene elements to be directly exported and imported into game engines.

Abstract

Primitive-based splatting methods like 3D Gaussian Splatting (3DGS) have revolutionized novel view synthesis with real-time rendering. However, their point-based representations remain incompatible with mesh-based pipelines that power AR/VR and game engines. We present MeshSplatting, a mesh-based reconstruction approach that jointly optimizes geometry and appearance through differentiable rendering. By enforcing connectivity via restricted Delaunay triangulation and refining surface consistency, MeshSplatting creates end-to-end smooth, high-fidelity meshes that render efficiently in real-time engines. On Mip-NeRF360 and Tanks&Temples, it boosts PSNR by +0.69 dB, while training 2× faster and using 2× less memory, bridging neural rendering and interactive 3D graphics for seamless real-time scene interaction. The project page is available at https://meshsplatting.github.io/.

1. Introduction

Recent advances in novel view synthesis, like 3D Gaussian Splatting [24], have enabled photo-realistic reconstruction of extremely complex scenes. 3D Gaussian Splatting encodes scenes with millions of 3D Gaussian primitives, achieving real-time rendering while also training efficiently. However, while 3DGS renders with high visual fidelity, Gaussian primitives are not immediately compatible with classical graphics pipelines used in simulators, games, and AR/VR applications, as these are typically based on *polygonal meshes*.

Therefore, integrating 3DGS in classical pipelines either requires *engineering* rendering engines [28, 35, 45] and simulators [34] to support them. However, this is non-trivial since 3DGS relies on *sorting* and *alpha blending*, preventing the use of standard techniques like depth buffers and occlusion culling [1, 21]. Another line of work is *converting*

Gaussian radiance fields into meshes [14, 19, 50]. Although somewhat effective, all of these conversion approaches rely on complex post-processing and typically lead to loss of visual quality, as the conversion step is non-differentiable. Rather than relying on conversion a-posteriori, we consider optimizing a mesh directly by making the rasterization process differentiable. Early approaches, such as Kato et al. [23] and Liu et al. [30], enabled differentiable optimization of solid polygonal surfaces, but they required very careful initialization for optimization to converge effectively and are mainly limited to object-level setups rather than large realistic scenes.

Instead, Held et al. [17] recently proposed to optimize (potentially transparent) triangles via volumetric rendering, therefore effectively replacing Gaussians with *triangles*. However, when their triangles are rendered in a game engine, a noticeable drop in visual quality occurs, as game engines assume triangles to be opaque. Moreover, Held et al. [17] outputs a *soup of triangles*, rather than a connected polygonal mesh, often needed for physics-based simulation.

Key Contributions. We introduce *MeshSplatting* to address all the aforementioned limitations: **(i)** an end-to-end optimization of mesh-based scene representations that retains visual quality while training $2\times$ faster than current state-of-the-art methods; **(ii)** rather than a polygon soup, we generate a connected mesh by refining the vertex locations of a restricted Delaunay triangulation; **(iii)** triangles are naturally connected to each other, and quantities stored within vertices are smoothly interpolated across each triangle; **(iv)** the optimization is aware that the triangles should be opaque, and therefore allowing direct high-quality rendering in standard game engines (see Fig. 1), opening the door for classical techniques like the use of depth buffers and occlusion culling [1, 21].

MeshSplatting achieves higher visual fidelity and captures finer geometric detail compared to modern *mesh-based* novel-view synthesis approaches [14, 19, 49, 50]. It is the first method to reconstruct large-scale real-world meshes end-to-end, directly producing connected, opaque, and colored triangle meshes *without* post-hoc extraction. Our representation can be *directly* imported into standard game engines, enabling a wide range of downstream applications including physics-based simulation, interactive walkthroughs, ray tracing, and scene editing, some of which are illustrated in Figure 1.

2. Related work

Differentiable rendering enables end-to-end optimization by propagating image-based losses back to scene parameters, allowing for the learning of explicit representations such as point clouds [12, 23], voxel grids [10], polygonal meshes [23, 30, 31], and more recently, Gaussian primitives [24]. The advent of 3D Gaussian Splatting [24] showed that it is possible to fit millions of anisotropic Gaussians in minutes, enabling real-time rendering with high fidelity. Since then, various directions have been explored to improve the Gaussian primitive, including the use of 2D Gaussians [19], generalized Gaussians [15, 42], alternative kernels [20], and learnable basis functions [6]. Other works moved beyond Gaussians entirely, investigating different primitives such as smooth 3D convexes [16], linear primitives [43], sparse voxel fields [41], or radiance foams [11]. More recently, Held et al. [17] advocated for the comeback of triangles, the most classical primitive in computer graphics. Several researchers have explored this direction, proposing triangle-based representations for efficient scene modeling [4, 22]. However, existing triangle-based methods usually result in an unstructured *triangle soup*, with no connectivity between adjacent triangles, and they fail to produce opaque triangles at the end of training, limiting usability in downstream applications. We propose a method that enforces *connectivity*, resulting in a mesh of opaque triangles directly compatible with game engines.

Mesh reconstruction from images. Implicit and explicit methods have made significant progress in reconstructing 3D scenes, but they remain largely incompatible with traditional game engines that primarily rely on mesh-based rasterization with depth buffers. Some methods propose strategies to convert implicit radiance fields into meshes. BakedSDF [47] learns a neural signed distance field and appearance and then bakes them into a textured triangle mesh. Binary Opacity Fields [37] drives densities toward near binary opacities so surfaces can be extracted as a mesh, and MobileNeRF [7] distills a NeRF into a compact set of textured polygons. However, these methods introduce overhead and increase the overall training time.

Several methods have built upon 3DGS and proposed ways to extract a mesh from an optimized Gaussian scene. 2DGS [19] and RaDe-GS [50] rely on Truncated Signed Distance Fields for mesh extraction. Other approaches extract meshes by sampling a surface-aligned Gaussian level set followed by Poisson reconstruction [13], or by defining a Gaussian opacity level set and applying Marching Tetrahedra on Gaussian-induced tetrahedral grids [49]. All of these, however, treat mesh extraction as a *separate* post-processing step, decoupled from the optimization process.

More recently, MiLo [14] integrates surface mesh extraction directly into the optimization, jointly refining both the mesh and the Gaussian representation. However, while MiLo optimizes the mesh geometry during training, color still has to be learned separately. Another line of work employs differentiable meshes for 3D reconstruction [39, 40], but these methods primarily target synthetic objects and do not generalize to real-world scenes. In contrast, MeshSplatting directly optimizes opaque triangles together with their

vertex colors, making the result immediately compatible with any game engine without additional post-processing steps.

3. Methodology

We now overview the key components of our method. Section 3.1 reviews Triangle Splatting [17], which is used as volume-renderable primitives for differentiable rendering in this work. We give an overview of our MeshSplatting representation in Section 3.2, and describe the optimization stages to convert the triangle soup into a connected mesh in Section 3.3. For the representation to be *natively* compatible with game engines, triangles need to be *opaque*, and the process to achieve this objective is detailed in Section 3.4. We conclude by detailing other optimization details in Section 3.5, such as densification/pruning, and training losses.

3.1. Background

In Triangle Splatting [17], each triangle $\mathbf{T_m}$ is defined by three vertices $\mathbf{v}_i \in \mathbb{R}^3$, a color $\mathbf{c_m}$, a smoothness parameter σ_m and an opacity o_m. The rasterization process begins by projecting each 3D vertex $\mathbf{v}_i$ of a triangle $\mathbf{T_m}$ onto the image plane using a standard pinhole camera model[1]. To determine the influence of a triangle on a pixel $\mathbf{p}$ and make the splatting process differentiable, the *signed distance field* ϕ of the 2D triangle in image space is defined as:

$$\phi(\mathbf{p}) = \max_{i \in \{1,2,3\}} L_i(\mathbf{p}), \quad L_i(\mathbf{p}) = \mathbf{n}_i \cdot \mathbf{p} + d_i, \tag{1}$$

where $\mathbf{n}_i$ are the unit normals of the triangle edges pointing outside the triangle, and d_i are offsets such that the triangle is given by the zero-level set of the function ϕ. The signed distance field ϕ thus takes positive values outside the triangle, negative values inside, and equals zero on its boundary. The window function I is then defined as:

$$I(\mathbf{p}) = \left(\text{ReLU} \left(\frac{\phi(\mathbf{p})}{\phi(\mathbf{s})} \right) \right)^{\sigma} \tag{2}$$

with $\mathbf{s} \in \mathbb{R}^2$ be the *incenter* of the projected triangle (*i.e.*, the point inside the triangle with minimum signed distance). Note how the indicator evaluates to 1 at the triangle incenter, 0 at the boundary and 0 outside the triangle. σ is a smoothness parameter that controls the transition between the incenter and boundary of the triangle. More specifically, as $\sigma \to 0$, the representation converges to a solid triangle, while larger values of σ yield a smooth window function that gradually increases from zero at the boundary to one at the center. This triangle parameterization has two main limitations: triangles remain isolated without vertex sharing, and treating σ and o as independent free parameters prevents them from becoming fully opaque after training.

[1]Unlike 3DGS, linearization of Gaussian projections is required [51], since perspective projection preserves linearity, 3D triangles remain triangles in 2D screen space.

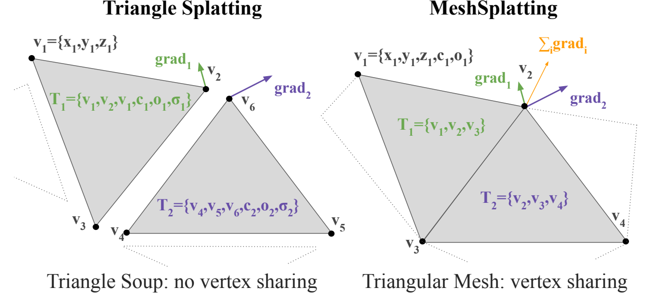

Figure 2. **Mesh parametrization.** (left) In a triangle soup, each triangle $\mathbf{T}_m$ is defined independently by three vertices $\mathbf{v}_i$, $\mathbf{v}_j$, $\mathbf{v}_k$, a color $\mathbf{c_m}$, a smoothness parameter σ_m, and an opacity o_m, without sharing vertices with neighboring triangles. (right) MeshSplatting parameterize a triangle $\mathbf{T}_m$ through a shared vertex set, where each vertex $\mathbf{v}_i$ stores x_i, y_i, z_i, c_i, and o_i. Each triangle is defined by the three indices in the vertex set that compose it. During the backward pass, gradients from all adjacent triangles are accumulated at shared vertices. The smoothness parameter σ is shared across all triangles.

3.2. Vertex-sharing triangle representation

In MeshSplatting, we define our mesh vertices as

$$\mathcal{V} = \{\mathbf{v}_i \in \mathbb{R}^3 \mid i = 1, \ldots, N\}, \tag{3}$$

with N denoting their cardinality. Similarly to Son et al. [40] each vertex is parameterized as $\mathbf{v}_i = (x_i, y_i, z_i, c_i, o_i)$, where $(x_i, y_i, z_i) \in \mathbb{R}^3$ denotes its 3D position, $\mathbf{c}_i \in \mathbb{R}^3$ the vertex color and $o_i \in [0, 1]$ the vertex opacity.[2] A triangle is defined by a triplet of vertices $\mathbf{T}_m = \{v_i, v_j, v_k\}$, its opacity is set to $o_{\mathbf{T}_m} = \min(o_i, o_j, o_k)$, and its color at a point inside the triangle is obtained by interpolating vertex colors with *barycentric coordinates*. During differentiation, vertex positions, colors, and opacities therefore receive the accumulated gradients from all triangles connected to it, as shown in Figure 2 (right).

3.3. From soups to meshes

Our optimization executes in two stages that *gradually* convert an unstructured into a structured representation, as illustrated in Figure 3. This design exploits the fact that, early in training, unstructured representations are *easier to optimize*, as they impose less constraints on the representation.

Stage 1. Triangle soup optimization . We start by taking as input a set of posed images and corresponding camera parameters obtained from structure-from-motion (SfM) [38], which also provides a sparse point cloud. For each 3D point, we initialize an equilateral triangle centered at that point, with its size proportional to the average distance to its three nearest neighbors, and random orientation. All triangles are initially defined as semi-transparent (we selected o_i=0.28 via hyper-parameter tuning).

[2]After training, the opacity parameter is *discarded*, and all triangles are treated as fully opaque for compatibility with standard game engines.

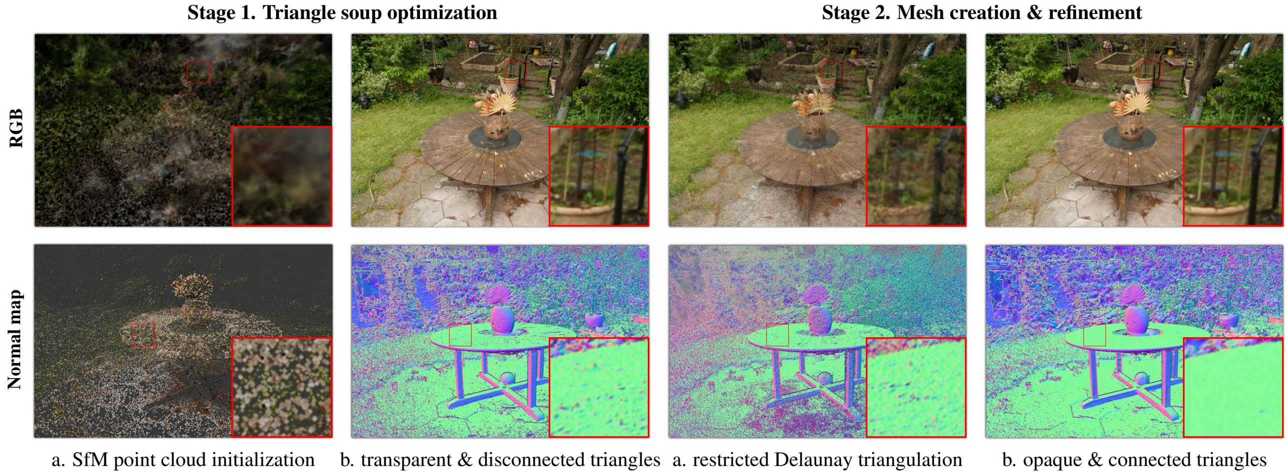

Figure 3. **From triangle soups to meshes.** (1a) We initialize semi-transparent triangles and scale them based on local density. (1b) We optimize a semi-transparent triangle soup without shared vertices, leading to disconnected triangles. (2a) Applying restricted Delaunay triangulation restores global connectivity but introduces geometric artifacts and a loss of visual quality, as vertex colors no longer accurately align with the underlying geometry. (2b) The final fine-tuning stage refines the connected mesh, producing smooth surfaces, accurate geometry, and restoring the visual fidelity lost during triangulation. Using only opaque triangles, our method achieves high visual quality compared to the semi-transparent and isolated triangle soup.

We begin the optimization with this *unstructured* triangle soup, i.e. without any connectivity or manifold constraints between triangles. Each triangle is optimized independently and can move freely under the influence of image-space gradients. This unconstrained formulation behaves similarly to point-based splatting, enabling rapid coverage of the visible scene and fast adaptation to local geometry and appearance. Note that, while similar, this is not the same as executing Held et al. [17], as we optimize *interpolated* vertex quantities within a triangle, rather than keeping them uniform.

Stage 2. Mesh creation & refinement. To transform our triangle soup into a connected mesh, we execute a restricted Delaunay triangulation on the triangle soup [8]. This operation first compute a standard Delaunay tetrahedralization, and then identifies tetrahedral faces whose dual Voronoi edges intersect the surface of the input triangle soup. The restricted Delaunay triangulation generates a mesh that *approximates the surface*, while at the same time, *maintaining Delaunay properties* (high quality meshing) locally restricted to it. Note that, in contrast to other reconstruction [26, 32, 33], restricted Delaunay triangulation does not introduce new vertices or modify their position. Instead, it *reuses* the optimized vertices directly, preserving both spatial accuracy and learned appearance.

Given the mesh connectivity from our restricted Delaunay triangulation, we then continue optimization so to fine-tune both vertex locations, as well as their appearance. As vertices are shared among adjacent triangles, gradients from neighboring faces are accumulated at shared vertices, ensuring that each vertex is updated consistently according to all incident triangles; see Figure 2. We do not need to introduce additional facets or vertices, as the unstructured triangle soup optimization has already produced a sufficiently dense set of triangles to capture all spatial regions accurately. After this fine-tuning stage, we obtain a fully opaque *mesh* capable to render the scene at *high photometric quality* in conventional rendering engines. In the final iterations of training, we enable supersampling, allowing even small triangles to receive gradients and be properly optimized.

3.4. Optimizing meshes with opaque triangles

Optimizing meshes composed only of *opaque* triangles introduces new challenges compared to classical NeRF/3DGS optimization. To enable gradient propagation through occlusions and allow the representation to be optimized effectively, the representation must remain semi-transparent in the early phases of training. There are two degrees of freedom that we can control in this regard: the per-vertex opacity parameter o, and the smoothness parameter σ.

Opacity parameter scheduling. We optimize opacity freely during the initial $5k$ iterations. Subsequently, we reparameterize opacity so that the optimizer is encouraged to make triangles more opaque. We achieve this by reparameterizing opacity values as

$$o'(o) = O_t + (1 - O_t) \cdot \mathrm{sigm}(o), \tag{4}$$

where $\mathrm{sigm}(.)$ is the sigmoid function, and scheduling O_t over time. Note that if $O_t = 0$ the sigmoid parameterizes opacities smoothly between 0 and 1. Conversely, if $O_t = 1$ all the opacities are mapped to a value of 1 (fully opaque triangles). By linearly increasing O_t from 0 to 1 during optimization, we can control this behavior smoothly.

Window parameter scheduling. We initialize the window parameter σ=1.0, corresponding to a linear transition

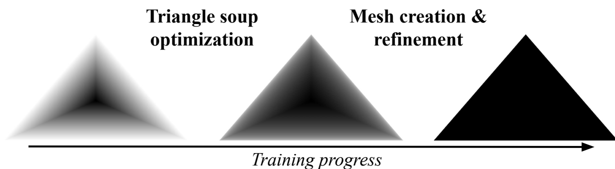

Figure 4. **Window parameter scheduling.** To ensure stable gradient flow during training, we begin with smooth triangles (σ=1.0, left) and linearly decrease σ throughout training, resulting in sharper triangles by the end. We visualize σ for a prototypical triangle at the *beginning* and *end* of each optimization stage.

from the incenter to the triangle boundary, and treat it as a *single parameter*, shared across all triangles. Throughout the training stages detailed in Section 3.3, σ is linearly annealed from $1.0 \rightarrow 0.0001$, ensuring strong gradient flow in the early stages while converging to opaque triangles at the end of the optimization process; see Figure 4. Note that this is in direct contrast to Triangle Splatting, as Held et al. [17] lets every triangle optimize its own window parameter.

3.5. Optimization details

We now detail our densification/pruning strategies, training losses, and observations about our rendering process.

Densification. As the initial triangle soup may not be sufficiently dense, we adapt the ideas from 3DGS-MCMC [27] to spawn additional triangles. At each densification step, candidate triangles are selected by sampling from a probability distribution constructed directly from their opacity o using Bernoulli sampling. Following Held et al. [17], new triangles are generated through *midpoint subdivision*: the midpoints of the three edges of a selected triangle are connected, splitting it into four smaller triangles. The new midpoints are added to the vertex set $\mathcal{V}$ and assigned the average color and opacity of their two adjacent vertices. We already exploit the connectivity in the early stages, as it drastically reduces the number of newly created vertices, resulting in only 6 new vertices after a split when connected, compared to 12 in a triangle-soup setting.

Pruning. At iteration $5k$ (Stage 1, just before opacity scheduling starts), we prune all triangles with opacity o<0.2, eliminating roughly 70% of primitives. During the rest of Stage 1, we monitor the *maximum* volume rendering blending weight w=$T \cdot o$ across views, and prune whenever w<O_t, hence eliminating *occluded* triangles as the representation becomes more opaque. While pruning is disabled during Stage 2, we perform a final pruning pass over all training views at the end of training to remove triangles that were never rendered.

Training losses. We optimize the 3D vertex positions $\mathbf{v}_i$, opacity o_i, and spherical harmonic color coefficients $\mathbf{c_i}$ of all vertices. Our training loss combines the photometric $\mathcal{L}_1$ and $\mathcal{L}_{\text{D-SSIM}}$ terms from 3DGS [24], the opacity loss $\mathcal{L}_o$ from Kheradmand et al. [27], the *depth alignment loss* $\mathcal{L}_z$ (detailed below), the normal loss $\mathcal{L}_n$ from [19], and the depth loss $\mathcal{L}_d$ from Kerbl et al. [25]:

$$\mathcal{L} = \mathcal{L}_{\text{3DGS}} + \beta_o \mathcal{L}_o + \beta_z \mathcal{L}_z + \beta_n \mathcal{L}_n + \beta_d \mathcal{L}_d \,. \quad (5)$$

We follow Kerbl et al. [25], and employ Depth Anything v2 [46] to align the predicted depths using their scale-and-shift procedure. The normal loss $\mathcal{L}_n$ can be supervised either by an external normal estimation network [18], or the self-supervised normal regularization from 2DGS [19], or both. We employ both supervision sources in all experiments, except for the mesh quality evaluation (Section 4.3), where we demonstrate that MeshSplatting also performs effectively even in a *self-supervised* setting.

Depth alignment loss. To promote the creation of manifolds, we align triangles to the observed depth map using a vertex-to-surface depth loss. To achieve this, for each rendered vertex v_i with predicted depth z_i and screen coordinates (x_i, y_i), we sample the predicted depth z_i^* from the rendered depth map, and (robustly via L1 losses) penalize the depth difference: $\mathcal{L}_z = \frac{1}{N} \sum_{i=1}^{N} |z_i - z_i^*|$. Unlike losses like normal consistency [3, 9] or Laplacian [3, 9, 44] that rely on local mesh connectivity, this formulation acts on each vertex independently.

Rendering equation. The final color of each image pixel $\mathbf{p}$ is computed by accumulating contributions from all overlapping triangles in depth order: $C(\mathbf{p}) = \sum_{n=1}^{N} \mathbf{c}_{T_n} o_{T_n} I(\mathbf{p}) \left(\prod_{i=1}^{n-1} (1 - o_{T_i} I(\mathbf{p})) \right)$. Analogously to Chen et al. [7], at the end of training, this simplifies to $C(\mathbf{p})$=$\mathbf{c}_{T_n} I(\mathbf{p})$, so that only a *single* evaluation per pixel is required, significantly accelerating the rendering process (i.e. *over-drawing is zero*).

4. Experiments

We compare our method to concurrent approaches on Mip-NeRF360 [2] and Tanks and Temples (T&T) [29]. We evaluate the visual quality using standard metrics: SSIM, PSNR, and LPIPS. Our the total number of used vertices and training times are reported on an NVIDIA A100 (40GB). Following Guédon et al. [14], we focus our evaluation on the task of *Mesh-Based Novel View Synthesis*, whose evaluation protocol attempts to measure how well reconstructed meshes reproduce complete scenes. Existing datasets such as DTU, MipNeRF360, and Tanks&Temples are limited: MipNeRF360 lacks ground-truth geometry, DTU only contains simple objects, and T&T provides sparse annotations with only foreground regions. As a solution, we measure novel view synthesis quality through the visual consistency between mesh renderings and reference images, capturing **(i)** geometric alignment and surface artifacts, **(ii)** mesh completeness, and **(iii)** background reconstruction, even when ground-truth geometry is unavail-

	Mesh	Color	Connect	Ready	Mip-NeRF360 dataset $PSNR^{\uparrow}$	$LPIPS^{\downarrow}$	$SSIM^{\uparrow}$	$\|V\|^{\downarrow}$	Tanks & Temples $PSNR^{\uparrow}$	$LPIPS^{\downarrow}$	$SSIM^{\uparrow}$	$\|V\|^{\downarrow}$
3DGS[24]	–	–	–	✗	27.21	0.214	0.815	–	23.14	0.183	0.841	–
Triangle Splatting [17]	-	-	-	✗	27.16	0.191	0.814	-	23.14	0.143	0.857	-
2DGS [19]	✗	✗	✓	✓	15.36	0.474	0.498	2M	14.23	0.485	0.569	16M
GOF [49]	✗	✗	✓	✓	20.78	0.465	0.573	33M	21.69	0.326	0.690	12M
RaDe-GS [50]	✗	✗	✓	✓	23.56	0.361	0.668	31M	20.51	0.344	0.659	10M
MiLo [14]	✓	✗	✓	✓	24.09	0.323	0.688	7M	21.46	0.348	0.706	4M
Triangle Splatting [17] †	✓	✓	✗	✓	21.05	0.462	0.558	3M	17.27	0.402	0.600	6M
MeshSplatting	✓	✓	✓	✓	24.78	0.310	0.728	3M	20.52	0.287	0.745	2M

Table 1. **Mesh-based novel view synthesis on the Mip-NeRF360 dataset.** MeshSplatting significantly outperforms all concurrent methods both in visual quality and in compactness, requiring far fewer vertices to achieve superior results. *Mesh* indicates whether a method directly produces a mesh (vs. requiring post-processing). *Color* denotes whether the mesh is already colored or requires some form of post-processing (e.g., coloring by fine-tuning). *Connect* specifies whether the final mesh consists of a connected component. *Ready* means the output is directly usable in standard game engines without custom rendering shaders. † with only opaque triangles.

able. Finally, to quantitatively assess the mesh quality of our method, we compute the Chamfer Distance on DTU.

Baselines. We compare our method against Triangle Splatting [17], and meshes derived from MiLo [14], 2DGS [19], Gaussian Opacity Fields (GOF) [49], and RaDe-GS [50]. For MiLo, surface mesh extraction is integrated into the optimization itself, so no additional post-processing is required. In contrast, 2DGS, GOF, and RaDe-GS rely on mesh extraction as a post-processing operation. Furthermore, *all* these methods require an additional post-processing stage to color the mesh, achieved by training a neural color field for $5k$ iterations; see MiLo for details [14]. For Triangle Splatting, we use the *opaque*† triangles version, as it produces *game engine* outputs without additional post-processing needed. For reference, we also compare against 3DGS [24] to highlight the contrast in rendering quality between volumetric and mesh-based novel view synthesis. Qualitatively, we compare against current state-of-the-art MiLo [14] and Triangle Splatting [17], which represents the closest line of work to ours.

Implementation details. We set the spherical harmonics to degree 3, which yields 51 parameters per vertex (48 from the SH coefficients and 3 from the vertex position) and 3 parameters per triangle (the vertex indices). In comparison, a single 3D Gaussian requires 59 parameters.

4.1. Mesh-based NVS – Table 1 and Figure 5

Table 1 reports quantitative results on the Mip-NeRF360 and Tanks & Temples datasets for mesh-based novel view synthesis. Compared to 2DGS and Triangle Splatting, our method uses a similar number of vertices, yet it achieves a 4–10 dB higher PSNR and a significantly lower LPIPS. Compared to GOF, RaDe-GS and MiLo, MeshSplatting uses 2 to 10 times fewer vertices while obtaining significantly higher SSIM and lower LPIPS. In terms of LPIPS (the metric that best correlates with human visual perception), MeshSplatting significantly outperforms all concurrent methods.

Method	Train ↓	FPS ↑ (HD)	FPS ↑ (Full HD)	Memory ↓
GOF	74m	OOM	OOM	1.5GB
RaDe-GS	84m	OOM	OOM	1.1GB
MiLo	106m	170	160	253MB
MeshSplatting	48m	220	190	100MB

Table 2. **Speed & memory on MipNeRF-360.** MeshSplatting achieves *faster training and lower memory usage* than concurrent methods. FPS were measured on a costumer MacBook M4.

On the Mip-NeRF360 dataset, our method achieves substantially higher PSNR than GOF, RaDe-GS and MiLo. On T&T, GOF and MiLo attain a higher PSNR but exhibit noticeably lower SSIM and higher LPIPS scores. This suggests that although GOF and MiLo produce highly detailed meshes, their renderings contain more artifacts, which degrade perceptual quality and lead to worse LPIPS and SSIM scores. We illustrate this qualitatively in Figure 5 and provide more examples in the *supplementary material*. MeshSplatting reconstructs fine structures and details more accurately and produces less noisy renderings.

Additionally, note that 2DGS, GOF, and RaDe-GS require two additional post-processing steps after training: first *extracting* a mesh, and then *coloring* it. MiLo directly outputs a mesh, but still relies on a post-processing stage to *texture* the mesh. These extra steps limit the practicality of such methods, and increase their overall complexity. In contrast, we *directly produce colored opaque meshes that are immediately compatible* with any game engine, *without* requiring additional steps.

4.2. Training speed & memory – Table 2

We report the average training time and final mesh size on the Mip-NeRF360 dataset. MeshSplatting trains in only 48 minutes on average, achieving a 35–55% speedup over concurrent mesh-yielding methods. Our optimized restricted Delaunay triangulation runs in under two minutes, contributing negligibly to the total training time. MiLo performs Delaunay triangulation at *every iteration*, whereas we run it only once. This results in a training time of

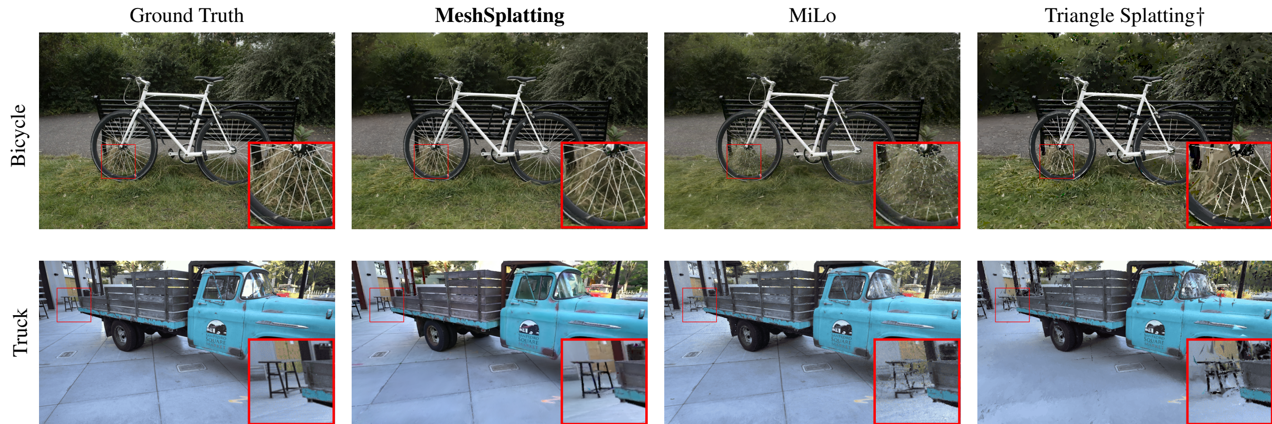

Figure 5. **Qualitative results.** Comparison of our method with ground truth, the current state-of-the-art MiLo [14] and opaque† Triangle Splatting [17]. Our approach produces renderings that are *closer to the ground truth, with sharper details and finer structures* (see the *Bicycle* spokes), and with fewer artifacts (see the table in the *Truck* scene). More visualizations are available in the *supplementary material*.

106 minutes for MiLo, compared to just 48 minutes for MeshSplatting. The final mesh representation produced by MeshSplatting is substantially more compact, with only 100 MB, which corresponds to a 2.5–15$\times$ reduction compared to concurrent methods. GOD and RaDe-GS require 1.5 GB and 1.1 GB of memory, respectively, making them impractical for real-world applications. This compactness enables MeshSplatting to run efficiently even on consumer hardware, significantly opening its applicability to lightweight rendering and real-time use-cases. MeshSplatting renders $\approx$25% faster on a consumer M4 MacBook compared to existing approaches. GOF and RaDe-GS exceed the device's memory capacity, resulting in an out-of-memory error.

4.3. Surface reconstruction

For evaluation on the DTU dataset, we use only self-supervised regularization (β_d to zero) to ensure a fair comparison with other methods and to show that MeshSplatting also performs well in a self-supervised setting. Although we designed MeshSplatting for mesh-based novel-view synthesis in large and complex real scenes, rather than surface reconstruction, it achieves mesh quality comparable to concurrent methods. Across the 15 scenes, MeshSplatting attains the lowest Chamfer distance in 5 of them. A detailed comparison table is provided in the *supplementary material*.

4.4. Applications

We now illustrate a few application demos that are quite difficult to implement with semi-transparent representations, yet almost trivial once the scene is represented with opaque meshes like in MeshSplatting.

Physical simulation. As our representation contains no semi-transparent primitives, the triangles may directly be interpreted as hard surfaces for the purpose of physical simulation. We demonstrate this by using an off-the-shelf non-convex mesh collider: the one provided in the *Unity game engine*. With no post-processing, our mesh can be loaded into Unity, and used for physics interaction with dynamic objects and characters. Figure 1 showcases a selection of downstream applications, with additional visualizations in the *supplementary material.*

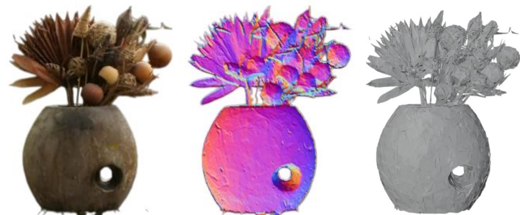

Figure 6. **Object segmentation.** With MeshSplatting, objects can be easily extracted or removed from a reconstructed scene, in this case the flower pot from the *Garden* scene. From left to right: generated RGB image of the object, estimated surface normals, and resulting mesh representation.

Object segmentation. Current 3D Gaussian Splatting methods for object extraction or removal [5, 48] face a fundamental challenge: a single pixel is rendered by accumulating the contributions of many primitives. This makes it non-trivial to decide whether a primitive belongs to a given object. To address this, prior work learns object associations during optimization [5, 48]. In contrast, MeshSplatting requires no additional tweak. Since each pixel is covered by exactly one triangle, mapping objects from image space to 3D space becomes straightforward: given a 2D mask of an object, all triangles contributing to pixels within that mask are directly identified as part of the object. By iterating over all training views, we recover the complete set of triangles belonging to the object. The object masks are generated using SAMv2 [36], which enables the selection of single or multiple objects that can then be removed or extracted from a scene with minimal complexity. Figure 6 shows a qualitative example of the extracted *flower pot* from the *Garden* scene of the Mip-NeRF360 dataset. Beyond object re-

	0	1	2	3	4+	Mean
Restricted Delaunay	1%	2%	5 %	11%	81%	6.1
With pruning	9%	22%	25%	19%	25%	2.5
Final mesh	2%	9%	16%	20%	53%	3.7

Table 3. **Connectivity.** Distribution of triangle connectivity on the *Garden* scene. The final mesh mostly consists of triangles connected to three or more neighboring triangles, indicating a well-connected mesh.

	PSNR ↑	LPIPS ↓	SSIM ↑
Baseline	**24.78**	**0.31**	**0.728**
w/o SH	-2.07	+0.06	-0.069
w/o $\mathcal{L}_d$	+0.05	-0.04	+0.006
w/o $\mathcal{L}_z$	+0.02	-0.01	+0.002
w/o $\mathcal{L}_n$	+0.10	-0.02	+0.004

Table 4. **Ablations (Mip-NeRF360).** We assess the impact of each design choice by removing each one of them individually.

moval and extraction, this capability allows the scene to be segmented into distinct sub-meshes by disconnecting triangles belonging to different objects, enabling structured and object-aware 3D scene editing.

4.5. Analysis

Mesh connectivity – Table 3. We report the distribution of triangles according to their connectivity with neighboring triangles. After the restricted Delaunay triangulation, most triangles are already well connected, with about 92% having at least three neighbors. If pruning is applied immediately after this step, a large number of triangles are removed because the Delaunay mesh contains many very small triangles that do not contribute to rendering, significantly reducing the overall connectivity. Instead, pruning is performed only after training. We make a final pass over all training views and remove all triangles that did not contribute to any rendered image. On average, each triangle is connected to approximately 3.7 triangles, and fewer than 2% of triangles remain isolated. Overall, most triangles are connected to exactly three neighboring triangles.

Ablations – Table 4 and Figure 7. While the regularization terms $\mathcal{L}_d$, $\mathcal{L}_z$, and $\mathcal{L}_n$ slightly reduce visual quality, they significantly improve geometric accuracy and yield smoother surfaces. The decrease in visual fidelity arises because the spherical harmonics representation cannot fully compensate for the reduced geometric flexibility, nor can it "cheat" by positioning triangles in non-physical configurations to better reproduce local colors or textures. This is further confirmed by replacing spherical harmonics with simple RGB colors, which results in an average drop of about 2 PSNR. This highlights that expressive color representations are essential for maintaining high visual fidelity with fully opaque triangles. Future work could decouple geometry and appearance. By employing a more expressive appearance model, such as neural textures, the representation

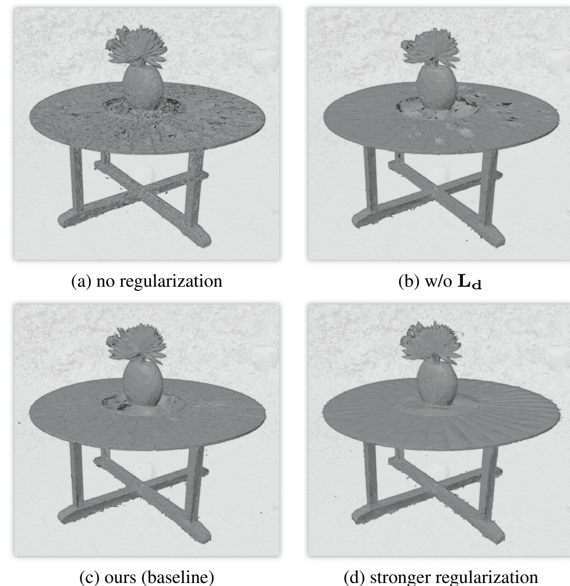

Figure 7. **Regularization *vs.* mesh quality.** (a) Without any regularization, the rendered views have high visual quality, but the underlying geometry is inaccurate. (b) The normal loss $\mathcal{L}_n$ encourages smoother surfaces, yet without the depth regularization $\mathcal{L}_d$, a few local regions show minor geometric inaccuracies. (c) Our baseline model achieves both smooth and geometrically consistent surfaces. (d) Increasing the regularization strength yields even smoother geometry, but the visual fidelity decreases as spherical harmonics fail to capture fine appearance details.

could achieve smoother and more accurate geometry without losing appearance information, thereby preserving high visual fidelity even under strong smoothness constraints. More ablations are provided in the *supplementary material.*

5. Conclusions

We introduce a differentiable framework for end-to-end optimization of mesh-based scene representations. By reformulating the triangle parameterization to enable vertex sharing, our method produces connected meshes while maintaining high visual fidelity. A redefined training strategy moves the optimization toward opaque triangles and connectivity, resulting in a unified representation that combines high quality appearance and accurate geometry within a compact, real-time-renderable mesh. MeshSplatting bridges radiance field optimization with traditional graphics pipelines, paving the way for the practical integration of neural scene representations into interactive VR applications, game engines, and simulation environments.

Acknowledgments. We thank Bernhard Kerbl and George Kopanas for their helpful feedback and for proofreading the paper. J. Held is funded by the F.R.S.-FNRS. The present research benefited from computational resources made available on Lucia, the Tier-1 supercomputer of the Walloon Region, infrastructure funded by the Walloon Region under the grant agreement n°1910247.

References

[1] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire. *Real-Time Rendering*. AK Peters/CRC Press, Boca Raton, Florida, USA, 4 edition, 2018. 1, 2

[2] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-NeRF 360: Unbounded anti-aliased neural radiance fields. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 5460–5469, New Orleans, LA, USA, 2022. 5

[3] Mark Boss, Zixuan Huang, Aaryaman Vasishta, and Varun Jampani. SF3D: Stable fast 3D mesh reconstruction with UV-unwrapping and illumination disentanglement. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 16240–16250, Nashville, TN, USA, 2025. 5

[4] Nathaniel Burgdorfer and Philippos Mordohai. Radiant triangle soup with soft connectivity forces for 3D reconstruction and novel view synthesis. *arXiv*, abs/2505.23642, 2025. 2

[5] Jiazhong Cen, Jiemin Fang, Chen Yang, Lingxi Xie, Xiaopeng Zhang, Wei Shen, and Qi Tian. Segment any 3D Gaussians. In *AAAI Conf. Artif. Intell.*, pages 1971–1979, Philadelphia, PA, USA, 2025. 7

[6] Haodong Chen, Runnan Chen, Qiang Qu, Zhaoqing Wang, Tongliang Liu, Xiaoming Chen, and Yuk Ying Chung. Beyond Gaussians: Fast and high-fidelity 3D splatting with linear kernels. *arXiv*, abs/2411.12440:1–14, 2024. 2

[7] Zhiqin Chen, Thomas Funkhouser, Peter Hedman, and Andrea Tagliasacchi. MobileNeRF: Exploiting the polygon rasterization pipeline for efficient neural field rendering on mobile architectures. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 16569–16578, Vancouver, Can., 2023. 2, 5

[8] Siu-Wing Cheng, Tamal Krishna Dey, and Jonathan Shewchuk. *Delaunay Mesh Generation*. Chapman and Hall/CRC, 2013. 4

[9] Jaehoon Choi, Rajvi Shah, Qinbo Li, Yipeng Wang, Ayush Saraf, Changil Kim, Jia-Bin Huang, Dinesh Manocha, Suhib Alsisan, and Johannes Kopf. LTM: Lightweight textured mesh extraction and refinement of large unbounded scenes for efficient storage and real-time rendering. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 5053–5063, Seattle, WA, USA, 2024. 5

[10] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 5491–5500, New Orleans, LA, USA, 2022. 2

[11] Shrisudhan Govindarajan, Daniel Rebain, Kwang Moo Yi, and Andrea Tagliasacchi. Radiant foam: Real-time differentiable ray tracing. *arXiv*, abs/2502.01157, 2025. 2

[12] Markus Gross and Hanspeter Pfister. *Point-Based Graphics*. Morgan Kauffmann Publ. Inc., San Francisco, CA, USA, 2007. 2

[13] Antoine Guédon and Vincent Lepetit. SuGaR: Surface-aligned Gaussian splatting for efficient 3D mesh reconstruction and high-quality mesh rendering. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 5354–5363, Seattle, WA, USA, 2024. 2

[14] Antoine Guédon, Diego Gomez, Nissim Maruani, Bingchen Gong, George Drettakis, and Maks Ovsjanikov. MILo: Mesh-in-the-loop Gaussian splatting for detailed and efficient surface reconstruction. *ACM Trans. Graph.*, 44(6):1–15, 2025. 2, 5, 6, 7

[15] Abdullah Hamdi, Luke Melas-Kyriazi, Jinjie Mai, Guocheng Qian, Ruoshi Liu, Carl Vondrick, Bernard Ghanem, and Andrea Vedaldi. GES: Generalized exponential splatting for efficient radiance field rendering. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 19812–19822, Seattle, WA, USA, 2024. 2

[16] Jan Held, Renaud Vandeghen, Abdullah Hamdi, Adrien Deliège, Anthony Cioppa, Silvio Giancola, Andrea Vedaldi, Bernard Ghanem, and Marc Van Droogenbroeck. 3D convex splatting: Radiance field rendering with 3D smooth convexes. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 21360–21369, Nashville, TN, USA, 2025. 2

[17] Jan Held, Renaud Vandeghen, Adrien Deliège, Daniel Hamdi, Abdullah Rebain, Silvio Giancola, Anthony Cioppa, Andrea Vedaldi, Bernard Ghanem, Andrea Tagliasacchi, and Marc Van Droogenbroeck. Triangle splatting for real-time radiance field rendering. In *Int. Conf. 3D Vis. (3DV)*, pages 1–10, Vancouver, Can., 2026. 2, 3, 4, 5, 6, 7

[18] Mu Hu, Wei Yin, Chi Zhang, Zhipeng Cai, Xiaoxiao Long, Hao Chen, Kaixuan Wang, Gang Yu, Chunhua Shen, and Shaojie Shen. Metric3D v2: A versatile monocular geometric foundation model for zero-shot metric depth and surface normal estimation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 46(12):10579–10596, 2024. 5

[19] Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. 2D Gaussian splatting for geometrically accurate radiance fields. In *ACM SIGGRAPH Conf. Pap.*, pages 1–11, Denver, CO, USA, 2024. 2, 5, 6

[20] Yi-Hua Huang, Ming-Xian Lin, Yang-Tian Sun, Ziyi Yang, Xiaoyang Lyu, Yan-Pei Cao, and Xiaojuan Qi. Deformable radial kernel splatting. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 21513–21523, Nashville, TN, USA, 2025. 2

[21] John Hughes, Andries van Dam, Morgan McGuire, David Sklar, James D. Foley, Steven Feiner, and Kurt Akeley. *Computer Graphics: Principles and Practice*. Addison-Wesley, 3 edition, 2014. 1, 2

[22] Changjian Jiang, Kerui Ren, Linning Xu, Jiong Chen, Jiangmiao Pang, Yu Zhang, Bo Dai, and Mulin Yu. HaloGS: Loose coupling of compact geometry and Gaussian splats for 3D scenes. *arXiv*, abs/2505.20267, 2025. 2

[23] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3D mesh renderer. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 3907–3916, Salt Lake City, UT, USA, 2018. 2

[24] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkuehler, and George Drettakis. 3D Gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.*, 42(4):1–14, 2023. 1, 2, 5, 6

[25] Bernhard Kerbl, Andreas Meuleman, Georgios Kopanas, Michael Wimmer, Alexandre Lanvin, and George Drettakis.

A hierarchical 3D Gaussian representation for real-time rendering of very large datasets. *ACM Trans. Graph.*, 43(4): 1–15, 2024. 5

[26] Muhammad Umar Karim Khan and Chong-Min Kyung. Poisson mixture model for high speed and low-power background subtraction. *Smart Sensors and Systems*, pages 1–23, 2020. 4

[27] Shakiba Kheradmand, Daniel Rebain, Gopal Sharma, Weiwei Sun, Jeff Tseng, Hossam Isack, Abhishek Kar, Andrea Tagliasacchi, and Kwang Moo Yi. 3D Gaussian splatting as Markov chain Monte Carlo. In *Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 80965–80986, Vancouver, Can., 2024. 5

[28] KIRI Innovations. 3DGS Render: 3D Gaussian splatting renderer for Blender. `https://github.com/Kiri-Innovation/3dgs-render-blender-addon`, 2025. 1

[29] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and temples: benchmarking large-scale scene reconstruction. *ACM Trans. Graph.*, 36(4):1–13, 2017. 5

[30] Shichen Liu, Weikai Chen, Tianye Li, and Hao Li. Soft rasterizer: A differentiable renderer for image-based 3D reasoning. In *IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, pages 7707–7716, Seoul, South Korea, 2019. 2

[31] Matthew M. Loper and Michael J. Black. OpenDR: An approximate differentiable renderer. In *Eur. Conf. Comput. Vis. (ECCV)*, pages 154–169, Zürich, Switzerland, 2014. 2

[32] Nissim Maruani, Roman Klokov, Maks Ovsjanikov, Pierre Alliez, and Mathieu Desbrun. VoroMesh: Learning watertight surface meshes with Voronoi diagrams. In *IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, pages 14519–14528, Paris, Fr., 2023. 4

[33] Nissim Maruani, Maks Ovsjanikov, Pierre Alliez, and Mathieu Desbrun. PoNQ: A neural QEM-based mesh representation. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 3647–3657, Seattle, WA, USA, 2024. 4

[34] Vismay Modi, Nicholas Sharp, Or Perel, Shinjiro Sueda, and David I. W. Levin. Simplicits: Mesh-free, geometry-agnostic elastic simulation. *ACM Trans. Graph.*, 43(4):1–11, 2024. 1

[35] Aras Pranckevičius. UnityGaussianSplatting: 3D Gaussian splatting in Unity. `https://github.com/aras-p/UnityGaussianSplatting`, 2023. 1

[36] Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, Eric Mintun, Junting Pan, Kalyan Vasudev Alwala, Nicolas Carion, Chao-Yuan Wu, Ross Girshick, Piotr Dollar, and Christoph Feichtenhofer. SAM 2: Segment anything in images and videos. In *Int. Conf. Learn. Represent. (ICLR)*, pages 1–44, Singapore, 2025. 7

[37] Christian Reiser, Stephan Garbin, Pratul Srinivasan, Dor Verbin, Richard Szeliski, Ben Mildenhall, Jonathan Barron, Peter Hedman, and Andreas Geiger. Binary opacity grids: Capturing fine geometric detail for mesh-based view synthesis. *ACM Trans. Graph.*, 43(4):1–14, 2024. 2

[38] Johannes L. Schonberger and Jan-Michael Frahm. Structure-from-motion revisited. In *IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 4104–4113, Las Vegas, NV, USA, 2016. 3

[39] Sanghyun Son, Matheus Gadelha, Yang Zhou, Zexiang Xu, Ming C. Lin, and Yi Zhou. DMesh: A differentiable mesh representation. In *Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 12035–12077, Vancouver, Can., 2024. 2

[40] Sanghyun Son, Matheus Gadelha, Yang Zhou, Matthew Fisher, Zexiang Xu, Yi-Ling Qiao, Ming C. Lin, and Yi Zhou. DMesh++: An efficient differentiable mesh for complex shapes. In *IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, pages 26590–26599, Honolulu, HI, USA, 2025. 2, 3

[41] Cheng Sun, Jaesung Choe, Charles Loop, Wei-Chiu Ma, and Yu-Chiang Frank Wang. Sparse voxels rasterization: Real-time high-fidelity radiance field rendering. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 16187–16196, Nashville, TN, USA, 2025. 2

[42] Maria Taktasheva, Lily Goli, Alessandro Fiorini, Zhen Li, Daniel Rebain, and Andrea Tagliasacchi. 3D Gaussian flats: Hybrid 2D/3D photometric scene reconstruction. In *Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 1–26, San Diego, CA, USA, 2025. 2

[43] Nicolas von Lützow and Matthias Nießner. LinPrim: Linear primitives for differentiable volumetric rendering. In *Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 1–27, San Diego, CA, USA, 2025. 2

[44] Nanyang Wang, Yinda Zhang, Zhuwen Li, Yanwei Fu, Wei Liu, and Yu-Gang Jiang. Pixel2Mesh: Generating 3D mesh models from single RGB images. *arXiv*, abs/1804.01654, 2018. 5

[45] XVERSE. XScene-UEPlugin: 3D Gaussian splatting plugin for Unreal Engine 5. `https://github.com/xverse-engine/XScene-UEPlugin`, 2024. 1

[46] Lihe Yang, Bingyi Kang, Zilong Huang, Zhen Zhao, Xiaogang Xu, Jiashi Feng, and Hengshuang Zhao. Depth anything V2. In *Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 1–37, Vancouver, Can., 2024. 5

[47] Lior Yariv, Peter Hedman, Christian Reiser, Dor Verbin, Pratul P. Srinivasan, Richard Szeliski, Jonathan T. Barron, and Ben Mildenhall. BakedSDF: Meshing neural SDFs for real-time view synthesis. In *ACM SIGGRAPH Conf. Proc.*, pages 1–9, Los Angeles, CA, USA, 2023. 2

[48] Mingqiao Ye, Martin Danelljan, Fisher Yu, and Lei Ke. Gaussian grouping: Segment and edit anything in 3D scenes. In *Eur. Conf. Comput. Vis. (ECCV)*, pages 162–179, 2024. 7

[49] Zehao Yu, Torsten Sattler, and Andreas Geiger. Gaussian opacity fields: Efficient adaptive surface reconstruction in unbounded scenes. *ACM Trans. Graph.*, 43(6):1–13, 2024. 2, 6

[50] Baowen Zhang, Chuan Fang, Rakesh Shrestha, Yixun Liang, Xiaoxiao Long, and Ping Tan. RaDe-GS: Rasterizing depth in Gaussian splatting. *arXiv*, abs/2406.01467, 2024. 2, 6

[51] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. EWA splatting. *IEEE Trans. Vis. Comput. Graph.*, 8(3):223–238, 2002. 3