Radiance Meshes for Volumetric Reconstruction

Alexander Mai [1] alexandertmai@gmail.com
Trevor Hedstrom [1] tjhedstr@ucsd.edu
George Kopanas [2,3] gkopanas@google.com
Janne Kontkanen [2] jkontkanen@google.com
Falko Kuester [1] fkuester@ucsd.edu
Jonathan T. Barron [2] barron@google.com
[1]University of California, San Diego [2]Google [3]Runway ML

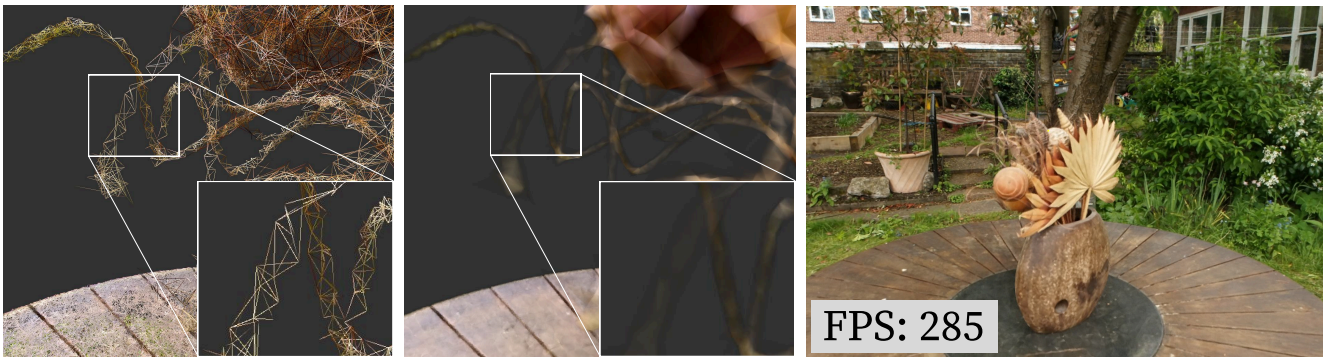

Figure 1. A radiance mesh parameterizes a radiance field using the tetrahedra produced by a Delaunay triangulation of a set of points, where each tetrahedron has a constant density and a linearly-varying color. This representation allows us to use traditional mesh processing (left) while still enabling semi-transparent rendering (middle), which allows our model to be rendered using a hardware triangle rasterizer very quickly on a wide variety of platforms (right). See the supplement for an interactive web demo.

Abstract

We introduce radiance meshes, a technique for representing radiance fields with constant density tetrahedral cells produced with a Delaunay tetrahedralization. Unlike a Voronoi diagram, a Delaunay tetrahedralization yields simple triangles that are natively supported by existing hardware. As such, our model is able to perform exact and fast volume rendering using both rasterization and ray-tracing. We introduce a new rasterization method that achieves faster rendering speeds than all prior radiance field representations (assuming an equivalent number of primitives and resolution) across a variety of platforms. Optimizing the positions of Delaunay vertices introduces topological discontinuities (edge flips). To solve this, we use a Zip-NeRF-style backbone which allows us to express a smoothly varying field even when the topology changes. Our rendering method exactly evaluates the volume rendering equation and enables high quality, real-time view synthesis on standard consumer hardware. Our tetrahedral meshes also lend themselves to a variety of exciting applications including fisheye lens distortion, physics-based simulation, editing, and mesh extraction.

1. Introduction

Radiance fields have become the standard paradigm for 3D reconstruction since NeRF [43]. It pioneered the use of differentiable volume rendering for directly optimizing a radiance field to explain a set of captured images. A fundamental tradeoff persists in radiance field model design: representations that are fast to render are difficult to optimize, such as opaque triangle meshes. Conversely, representations with well-behaved optimization dynamics, like MLPs, tend to be slow to render.

A particularly effective point on this trade-off continuum is 3D Gaussian Splatting (3DGS) [31]. It is fast to render and relative easy to make backwards-compatible with the existing graphics ecosystem. But 3DGS has a variety of limitations: 1) The volume rendering approximations that allow the high frame-rates introduce popping artifacts. 2) Splatting is challenging with complicated camera models like ultra-wide lenses. 3) Splats can be difficult to edit. 4) Though 3DGS is faster than most radiance field models, it

is significantly slower than conventional mesh-based real-time rendering techniques.

The recently introduced Radiant Foam [19] partitions space using an optimizable 3D Voronoi diagram. While this approach strikes a good balance between speed and quality, it relies on ray tracing arbitrarily complex convex *polyhedra* (the Voronoi cells). While GPU-based rasterization of these cells is theoretically possible, it would be prohibitively costly: Voronoi cells generally possess an average of 15.5 faces [42]. Rasterizing a single cell would require tessellating these faces into dozens of triangles. Even accounting for the fact that the dual Delaunay graph contains multiple tetrahedra per Voronoi site, the total triangle count required to rasterize the Voronoi diagram is nearly double that of the Delaunay representation. In addition, to find the transparency of each cell, we need to find the distance from the front face to the back face. However finding the distance to the back face of the polyhedra requires iterating over all of it's neighbors within the fragment shader, which is very expensive.

We introduce a method that is able to perform high quality view synthesis using tetrahedral meshes - specifically, a Delaunay tetrahedralization. The Delaunay tetrahedralization partitions space into tetrahedral cells, to which we assign a constant density and linear color (radiance) to create what we call a *radiance mesh*. The use of Delaunay tetrahedralization enables unconstrained optimization of the vertex locations, as we can simply recompute the triangulation after the points have moved. RadFoam had considered the use of a Delaunay tetrahedralization, as it is the dual of the Voronoi diagram. However, they chose not to pursue it because of the discontinuous mapping from vertex positions to mesh topology. These present difficulties during optimization, which we overcome.

The resulting radiance mesh can be sorted front to back very efficiently [13, 29, 41] using a radix sort over the power of the circumsphere. This, combined with the fixed number of faces per cell, enables fast rasterization, even faster than Gaussian splatting, but without the popping. Since the tetrahedral mesh produced is comprised entirely of triangles, it is possible to utilize the hardware triangle rasterizer to accelerate rendering, and to utilize the hardware triangle intersector to accelerate ray tracing. For this purpose, we introduce a novel hardware rasterization method that interpolates the exit intersection and uses mesh shaders to collaboratively load data onto vertices. We achieve rendering speeds 32% faster than the original 3DGS implementation at 1440p resolutions and higher quality than RadFoam. For ray tracing, we achieve a 17% speed increase compared to RadFoam on MipNeRF360 indoor scenes. Crucially, unlike 3DGS, our method utilizes exact visibility. This guarantees that radiance meshes are free from the temporal popping and photometric inconsistencies caused by the sorting errors inherent to splatting.

Because radiance meshes consist of semi-transparent triangles, they are natively compatible with the real-time graphics ecosystem. For example, we can support physics-based simulations using Extended Position Based Dynamics [39], which we demonstrate as an interactive application. Additionally, we can extract a watertight opaque surface triangle mesh by thresholding our primitives according to their contribution to the rendered views. We provide reference implementations for both the rasterizer on desktop and web, as well as the ray tracer.

2. Related Work

Neural Volume Rendering . Neural Radiance Fields use numerical quadrature to integrate emission-only volume rendering parameterized by a neural network [43]. Many improvements to NeRF have since been proposed, such as different filtering approaches [3–5] and different field representations [6, 15, 38, 46, 63]. Adaptive Voronoi NeRFs proposed partitioning a scene into multiple radiance fields using a Voronoi diagram [14], and Radiant Foam [19] partition a scene into constant density Voronoi cells to enable fast ray-tracing.

Particle-Based Radiance Fields . Early works [1, 33, 62] explored particle based representations as an alternative to fields. 3DGS [31] popularized them by using Gaussian primitives. Many other primitives have since been explored [8, 22, 24–27, 34, 37, 50, 67], largely based on the same splatting-based alpha compositing technique as 3DGS. Most splatting-based methods make different approximations to volume rendering that result in instabilities and "popping". StopThePop [51] and 3DGRT [44] use per-ray-sorting to improve popping. SVRaster [56] is able to perform high speed rasterization of voxel primitives using few quadrature samples within each primitive. EVER [40] is the first primitive-based representation able to perform fast and exact integration via ray-tracing.

Mesh-Based View synthesis . Because meshes are the dominant representation used in computer graphics, many have explored how to best use meshes to accelerate radiance field rendering. Popular approaches include optimizing a field that has a dual interpretation and can be used both as a radiance field and a surface [61, 64, 65]. Another way to obtain mesh-like properties while retaining the image quality of 3DGS is to attach Gaussians to a mesh [10, 16, 17, 36, 49, 55, 59, 60, 68], convert Gaussians to a mesh using a Delaunay triangulation [20, 66], or binarize during the optimization a volumetric field such that it can be converted to a mesh [52]. Meshes can also be directly optimized via gradient descent [9, 30, 35], though these approaches are limited due to edge discontinuities, which cause optimization issues.

Tetrahedralizations . Tetrahedra has been widely studied as a volumetric representation in graphics, specifically for solids. To accelerate rendering, power sorting was first introduced by Karasick et al. [29]. Even further acceleration were introduced by [18, 47] and Tricard [57] utilized geometry shaders for further acceleration. Tetrahedra have also been used for view synthesis in [21, 58].

3. Preliminaries

Neural Radiance Fields A radiance field is an emission-only volumetric field consisting of a density field $\sigma(\mathbf{x})$ and a color field $\mathbf{c}(\mathbf{x}, \mathbf{d})$, where color may depend on viewing direction $\mathbf{d}$. Each pixel's color is rendered by setting up a ray $\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$ with origin $\mathbf{o}$ and direction $\mathbf{d}$ through the pixel, and ray marching using standard volume rendering integral (also known as the radiative transfer equation [7]):

$$\int_0^\infty \mathbf{c}(\mathbf{r}(t), \mathbf{d})\sigma(\mathbf{r}(t)) \exp\left(-\int_0^t \sigma(\mathbf{r}(s))\, ds\right) dt\,. \quad (1)$$

Delaunay Triangulation Given a set of points, the Delaunay triangulation [12] is the triangulation defined by the empty sphere property: that the circumscribed sphere (circumsphere) corresponding to every tetrahedron contains no other vertices. A Delaunay tetrahedralization can be sorted from front to back [13, 41] with respect to a camera origin $\mathbf{o}$ according to the power of each circumsphere with respect to the camera origin:

$$P(T) = \|\mathbb{O}(T) - \mathbf{o}\|_2^2 - \mathbb{R}(T)^2\,, \quad (2)$$

where $\mathbb{R}(T)$ and $\mathbb{O}(T)$ are the radius and center, respectively, of the circumsphere of T. We offer a visual intuition of why this works using the radical axis in the supplement while the full proof can be found in [29]. This sort order is particularly useful because it is with respect to the ray origin, which means it is valid for fisheye lens and other such distorted projections.

Zip-NeRF Downweighting To mitigate aliasing, Zip-NeRF [5] introduced a querying scheme where the output of the instant-NGP is convolved with a gaussian. For a given query with σ denoting the filter variance, it multiplies each level ℓ with a grid resolution of n_ℓ by the following factor:

$$\phi(\sigma) = \mathrm{erf}\left(1/\sqrt{8\sigma^2 n_\ell^2}\right), \quad (3)$$

Together with a regularizer that pulls all features towards 0, this reduces the impact of higher resolution features for larger sized queries.

4. Method

Our model takes as input a set of posed images and sparse initial point cloud. During optimization, we represent our scene using a set of 3D points $\{\mathbf{v}_i\}$, initialized from the COLMAP SfM reconstruction [54], and an Instant-NGP datastructure G_θ. Every 10 optimization iterations, the Delaunay triangulation of points $\{\mathbf{v}_i\}$ is computed to yield a set of non-overlapping tetrahedra $\{T_k\}$. Because $\{\mathbf{v}_i\}$ move during optimization, tetrahedra may appear and disappear. We therefore use the Instant-NGP data structure to parameterize the non-spatial properties of each tetrahedron. The output of our model is a set of tetrahedra $\{T_k\}$, each with a constant density σ_k, a base color given by spherical harmonics $\mathbf{c}_k^0$, and a linear color gradient $\nabla\mathbf{c}_k$, all of which are produced by querying G_θ with each T_k (see Section 4.2).

Given our parameterization, the color at any interior point $\mathbf{p} \in T_k$ is:

$$\mathbf{c}_k(\mathbf{p}) = \mathbf{c}_k^0 + \nabla\mathbf{c}_k \cdot \big(\mathbf{p} - \mathbb{O}_k\big)\,, \quad (4)$$

where $\mathbb{O}_k$ is either the circumcenter or centroid of T_k.

4.1. Rendering

Before rendering, we sort $\{T_k\}$ utilizing power sorting (Equation 2). For any given ray, we can now iterate over each primitive and compute the emission only volume rendering equation across the interval bounded by the primitive. Let t_k^{in} and t_k^{out} be the distances where the ray $\mathbf{r}(t)$ enters and exits the k'th tetrahedron (computed using the Cyrus-Beck algorithm [11]). The color contribution of tetrahedron k to the ray, premultiplied by alpha, has the form:

$$\mathbf{\Delta C}_k = \int_{t_k^{\text{in}}}^{t_k^{\text{out}}} e^{-\sigma_k(t - t_k^{\text{in}})}\, \mathbf{c}_k\big(\mathbf{r}(t)\big)\, \sigma_k\, dt\,. \quad (5)$$

This integral has a closed form solution that depends only on the constant density and the color of the tetrahedron at its entry and exit points. This solution for the color premultiplied by the alpha is as follows:

$$\mathbf{\Delta C}_k = \left(1 - \frac{\alpha_k}{d_k}\right)\mathbf{c}_k^{\text{in}} + \left(\frac{\alpha_k}{d_k} - e^{-d_k}\right)\mathbf{c}_k^{\text{out}}\,, \quad (6)$$

$$\begin{aligned} d_k &= \big(t_k^{\text{out}} - t_k^{\text{in}}\big)\sigma_k & \alpha_k &= 1 - e^{-d_k} \\ \mathbf{c}_k^{\text{in}} &= \mathbf{c}_k\big(\mathbf{r}\big(t_k^{\text{in}}\big)\big) & \mathbf{c}_k^{\text{out}} &= \mathbf{c}_k\big(\mathbf{r}\big(t_k^{\text{out}}\big)\big) \end{aligned} \quad (7)$$

To render the radiance mesh, we rasterize the tetrahedra front-to-back, using Eq. 6 and Eq. 7 to compute the color and opacity of each tetrahedron, respectively. The final pixel color is found by blending the colors of the pre-sorted tetrahedra along the ray as follows:

$$\mathbf{C} = \sum_k w_k \mathbf{\Delta C}_k\,, \qquad w_k = \prod_{l=1}^{k-1}(1 - \alpha_\ell)\,, \quad (8)$$

which we efficiently compute using hardware rasterization. For each pixel, the $\mathbf{\Delta C}_k$ and α_k values are computed once

per tetrahedra within the fragment shader, and the product of opacities w_k is efficiently computed using hardware blending operations.

4.2. Querying Tetrahedron Properties

As vertices move during optimization, tetrahedra might appear and disappear after recomputing the Delaunay Tetrahedralization. Attaching density and appearance features to vertices works poorly, as we demonstrate in our ablation study in Tab. 2. We therefore optimize a field, parameterized by an Instant-NGP datastructure [46] and the mip-NeRF 360 contraction function [4], which we query with each tetrahedron to produce density and appearance features.

Because tetrahedra are volumes but Instant-NGP expects points as input, we query the NGP using an approach similar to that of Zip-NeRF [5]. We evaluated two strategies as the center $\mathbb{O}_k$ of a tetrahedra: the circumcenter and the centroid. The circumcenter has the advantage of making the method continuous. This is because,the circumcenters are guaranteed to be the same for tetrahedra involved in edge-flipping as visualized in Fig. 2, resulting in a continuous function over the Delaunay Tetrahedralization. However, the circumcenter has poor numerical conditioning, as it can move very quickly when a tetrahedron is skinny. The circumradius $\mathbb{R}_k$ is used to prefilter the NGP to address skinny tetrahedra. As a tetrahedron becomes more co-planar, the system of linear equations used to find the circumcenter becomes poorly conditioned (nearly singular), and the circumradius becomes larger. By filtering the NGP using this radius, using downweighing as in Zip-NeRF [5], we attempt to mitigate this issue. Our second strategy, the centroid, has the disadvantage of not being continuous, but has much better numerical conditioning. We attempt to counteract this lack of continuity with the same filtering strategy. We compare these two methods in the ablation and show that the centroid has better performance.

Each individual attribute is produced by the Instant NGP G_θ with downweighting function ϕ as follows:

$$\mathbf{b}_k \leftarrow \phi(\mathbb{R}_k) \circ G_\theta(\mathbb{O}_k)\,, \tag{9}$$

$$\sigma_k \leftarrow \exp(h_\sigma(\mathbf{b}_k))\,, \tag{10}$$

$$\mathbf{c}_k^0 \leftarrow \text{softplus}_{\beta=10}(h_{sh}(\mathbf{b}_k) \cdot Y(d_k))\,, \tag{11}$$

$$\nabla \mathbf{c}_k \leftarrow \frac{\min \mathbf{c}_k}{\mathbb{R}_k} \frac{h_\Delta(\mathbf{b}_k))}{\sqrt{1 + \|h_\Delta(\mathbf{b}_k))\|_2^2}}\,, \tag{12}$$

where each of h_σ, h_Δ, h_c, and h_{sh} are the different shallow neural network heads that take the Instant-NGP bottleneck feature $\mathbf{b}$ as input, and where $Y(\mathbf{d}_k)$ are the spherical harmonics basis functions evaluated in the direction $\mathbf{d}_k$.

The purpose of the activation on the color gradient $\nabla \mathbf{c}_k$ is to prevent the color from going negative within the tetrahedron. Since the vertices of the tetrahedron lie within a ball of radius $\mathbb{R}_k$, combining Eq. 12 into Eq. 4 yields a maximum negative change of $-\min \mathbf{c}_k$. We need to use the minimum across channels because we reuse the color gradient across all color channels for the purpose of efficiency, instead of giving each channel it's own gradient. Using a monochrome gradient saves 30% of memory loads in the shaders.

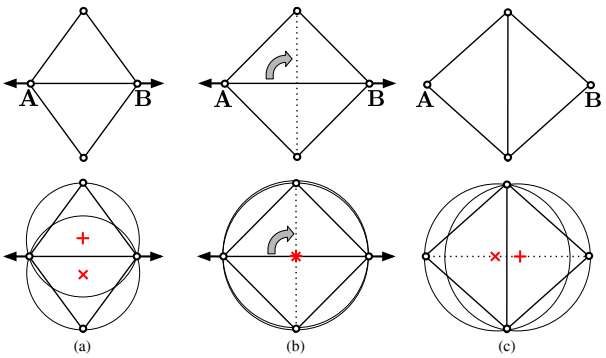

Figure 2. **Top**: Vertices **A** and **B** are moved slightly as we go from left to right, illustrating the the problem with Delaunay Triangulation. This small modification causes a topological flip. Thus naively assigning colors to vertices and using barycentric interpolation would not give a smooth optimization objective. **Bottom**: Our circumcenter smoothing. We color each triangle based on the position of their circumcenters (+ and ×). This works, because at the moment of the flip the centers are co-located.

4.3. Mesh Densification

Particle-based radiance field methods like 3DGS use heuristics to densify or prune particles based on their contribution to rendered images. We use only densification, done by adding points to $\{\mathbf{v}_i\}$ during optimization. We conceptualize the densification process as choosing a tetrahedron to "split", then choosing a point within the selected tetrahedron as the insertion point. Every 500 iterations, we iterate over a sample of the training dataset of size M and collect two metrics: an SSIM split score and a total variance split score, select primitives to "split" according to these scores, then chose a point to add to $\{\mathbf{v}_i\}$ from each selected primitive, before continuing with optimization.

The first and most important selection method, the SSIM split $\mathcal{S}_k$, is based on Error Based Densification [53]. We select all primitives with a score $\mathcal{S}_k > 0.5$ for densification. The idea is to back-project the image error associated with each pixel, specifically SSIM, onto all of the primitives that contributed to that pixel according to the proportion they contributed, or the weight. These are accumulated on the primitives across the image to obtain an error score, then we take the mean of the top two error scores across the sample

images $\{\pi_n\}_{n=1}^M$:

$$\nu_k^{\pi_n} = \left[\sum_i \{\mathbb{1}[w_k\alpha_k > 0]\}_i\right]_{\pi_n} \tag{13}$$

$$\mathcal{S}_k = \text{mean}\left(\underset{\pi_n}{\text{top2}}\left[\frac{1}{\nu_k^{\pi_n}}\sum_i \{w_k\alpha_k s\}_i\right]_{\pi_n}\right)$$

The brackets $[\{\cdot\}_i]_{\pi_n}$ indicate that everything within takes place within camera π_n at pixel i. So $[\sum_i]_{\pi_n}$ sums over all pixels in image π_n, and s is the SSIM error at pixel i within camera π_n. The top two scores are used because at least two views are needed to triangulate a primitive.

This first metric drives the majority of the densification decisions, but fails to detect regions that are low density to begin with. For regions with thin structures represented by low density early in optimization, we devise a second selection strategy we call the total variance split score $\mathcal{T}_k$, with the selection of all primitives $\mathcal{T}_k > 2.0$ for densification. We treat each tetrahedron as an estimator of radiance within its volume and analyze the variance of its residuals. We compute the variance of the error at each tetrahedron to measure whether the radiance is being represented well. This variance, σ_k^2, is then multiplied by the weighted sum of the total error, as follows:

$$\sigma_k^2 = \mathbb{E}[\boldsymbol{\delta}^2] - \mathbb{E}[\boldsymbol{\delta}]^2\,, \quad \mathbb{E}[\mathbf{x}] = \frac{\sum_{\pi_n}[\sum_i \{\alpha_k w_k \mathbf{x}\}_i]_{\pi_n}}{\sum_{\pi_n}[\sum_i \{\alpha_k w_k\}_i]_{\pi_n}}$$

$$\mathcal{T}_k = \sigma_k^2 \sum_{\pi_n}\left[\sum_i \{\alpha_k w_k\}_i\right]_{\pi_n} \tag{14}$$

where we use the same sum and bracket notation, and $\boldsymbol{\delta} = \mathbf{C} - \mathbf{C}_{\text{gt}}$ is the pixel residual.

Next, we need to select the locations of the new points within the selected tetrahedra. We note that uniform sampling within a tetrahedron is ineffective for this purpose. For example, it is common during initialization that a large tetrahedron just touches the surface that needs to be densified. In such case, a random point would almost always fall into the empty space rather than at the surface.

To address this, we select two rays corresponding to the views with highest error scores, and use their approximate intersection as the new point, falling back to a random point within the tetrahedron should the intersection lie outside of the tetrahedron. We use the mean ray for each of the two views, computed by finding the line segment between the entrance and exit centroids, $\boldsymbol{\mu}_k^{\text{in}}$ and $\boldsymbol{\mu}_k^{\text{out}}$. For a given camera π_n and primitive k, those are computed as follows:

$$\boldsymbol{\mu}_k^{\text{in}} = \frac{\sum_i \{w_k\alpha_k \mathbf{r}(t_k^{\text{in}})\}_i}{\sum_i \{w_k\alpha_k\}_i}, \quad \boldsymbol{\mu}_k^{\text{out}} = \frac{\sum_i \{w_k\alpha_k \mathbf{r}(t_k^{\text{out}})\}_i}{\sum_i \{w_k\alpha_k\}_i} \tag{15}$$

We then take the two segments corresponding to the top two SSIM error cameras and find the approximate intersection. This consists of finding the midpoint between the shortest line between the two segments. Finally, this midpoint is the new vertex added to $\{\mathbf{v}_i\}$.

4.4. High Speed Rendering

We introduce a novel method for rasterizing a tetrahedral volume using hardware rasterization. A previous approach emitted 2-3 triangles to represent the tetrahedron [57], then interpolated the transparency across each triangle. However, emitting variable amounts of geometry is slow, so instead we emit duplicate but static geometry. This includes duplicating each face twice and each vertex as many times as it is connected to a tetrahedron. We then rely on the rasterization pipeline to filter out what it needs. An overview of the approach can be see in Fig. 3. Our approach can be thought of as treating each tetrahedron as a distinct primitive, for which we output an individually wound tetrahedron. Then, instead of interpolating transparency, we interpolate the distance to the back faces. Details of how to interpolate this distance can be found in the Supplement.

When mesh shaders are not available, we use instanced shaders, drawing an instance of a triangle strip for each tetrahedron, where the vertex shader uses the instance index to load the tetrahedron attributes and plane equations for each tetrahedron. This means that we load each tetrahedron 4 times: once for each of its vertices. These duplicated loads can be removed by utilizing mesh shaders. Mesh shaders replace vertex shaders, allowing an entire thread block to collaboratively generate mesh data which is passed directly to the hardware rasterizer. We leverage this feature to allow vertices to share the data they load within each tetrahedron using warp shuffle operations, so we only need to load each tetrahedron once.

The differentiable PyTorch renderer is implemented as a tile based renderer in Slang.D [2] to enable tracking of derivatives, which is difficult across the vertex and fragment shaders. The Vulkan renderer is implemented in C++ and Slang, which allows it to support mesh shaders, our fastest rendering approach. Finally, the web viewer implements instanced rendering in JavaScript and WebGPU, as mesh shaders are not yet available.

5. Results

We evaluate radiance meshes on the nine scenes introduced in Mip-NeRF 360 [4] and four scenes from DeepBlending [23] and Tanks&Temples [32]. We compare against several recent state-of-the-art radiance field representations including 3DGS [31], EVER [40], SVRaster [56], Radiant Foam [19], and Triangle Splatting [25]. Results are shown in Table 1.

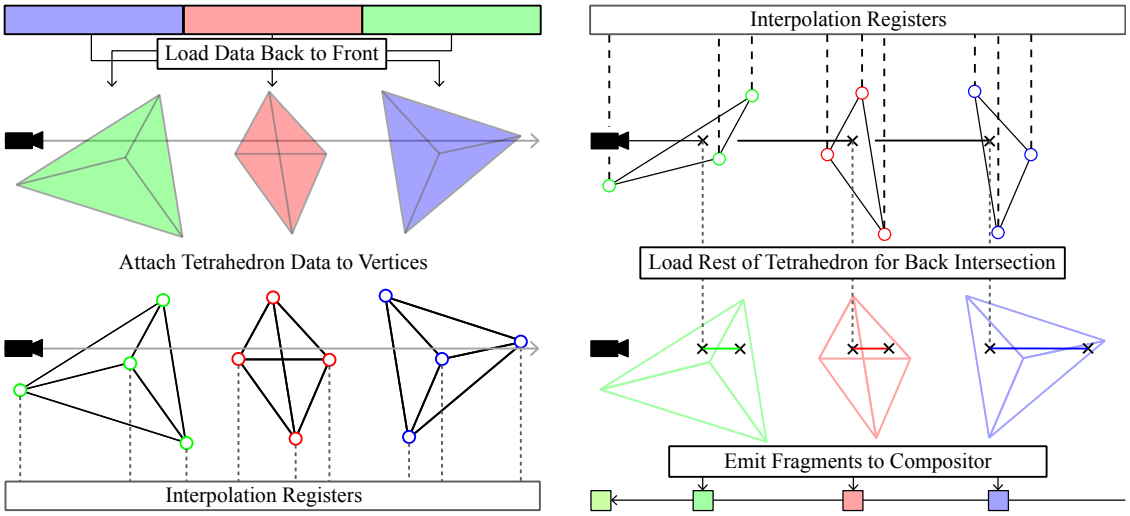

Figure 3. Radiance meshes can be efficiently rendered using the hardware rasterization on commodity GPUs. We sort all tetrahedra using the power of their circumspheres relative to the camera origin, and load them into the vertex shader from back to front. We then traverse the tets from back to front and for each, we emit 4 vertices with the tetrahedra data attached to each to the interpolation registers. This data is then read by the fragment shader, which must then read back the whole tetrahedron attached to each triangle it hit to reconstruct each back intersection. With this information, the shader can finally integrate across the interval and emit the fragment to the compositor, which composites the fragment according to the load order, so back to front.

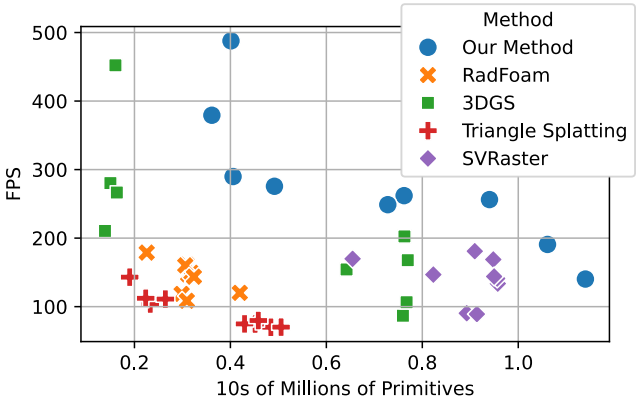

Figure 4. Rendering speed vs. primitive count. Each point represents a different scene from the Mip-NeRF 360 dataset, plotted by the number of primitives used for reconstruction (x-axis) against the average rendering speed across test views (y-axis). Performance is measured at standard test set resolution.

Our method, while generally faster than all baselines, also achieves an improvement in terms of quality compared to RadFoam [19], which is the most similar baseline to ours. While 3DGS [31] achieves higher quality, it suffers from popping artifacts and generally slower rendering speeds. EVER [40] and SVRaster [56] have accurate volumetric rendering and high quality, but suffer from even slower rendering speeds and very limited portability, especially compared to ours. To further investigate performance, we show in Figure 4 that our method clearly defines the pareto front in terms of performance per primitive across a wide range of baselines.

Another important metric that ensures practicality is VRAM consumption. Radiant Foam has high VRAM requirements; when running it on an RTX 4090 (24GB), we observed that it runs out of memory at ∼3.3 million cells. In contrast, our method supports ∼15 million tetrahedra on the same hardware.

Our optimization also proves to be more robust than Radiant Foam. In Tanks&Temples+DeepBlending dataset RadFoam fails to reconstruct them in a catastrophic way, while our method achieves competitive results with all the other baselines. We hypothesize that the RadFoam tracing algorithm fails because of numerical stability issues, as a single incorrect intersection, such as on a nearly parallel face, can cause a ray to diverge.

In Table 2 we present our ablation study, run on the *bicycle* and *room* scenes, from Mip-NeRF360. We investigate different approaches for splitting and colorizing primitives, and different Instant-NGP querying approaches. We see that omitting the SSIM based variance based densification ("SSIM Splitting") decreases performance significantly. Omitting the total dataset variance based densification ("Total Var Splitting") decreases performance marginally, and qualitatively mostly affects thin structures, shown in Fig. 5. Assigning each tetrahedron a single color ("Constant Color") instead of using a linear color results in a marginal decrease in quality. Ablating the Instant-NGP ("No Instant-NGP") and directly assigning properties

	mip-NeRF360 Outdoor [4]			mip-NeRF360 Indoor [4]			T&T [32]+DB [23]		
	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓
3DGS [31]	24.63	0.729	0.271	31.05	0.924	0.238	26.65	0.876	0.263
EVER [40]	24.73	0.744	0.239	30.98	0.926	0.227	26.09	0.889	0.234
Triangle Splatting [24]	24.16	0.720	0.248	30.79	0.923	0.207	26.59	0.875	0.243
SVRaster [56]	24.70	0.738	0.234	30.67	0.927	0.203	26.52	0.874	0.229
RadFoam [19]	23.90	0.663	0.366	30.66	0.906	0.251	19.20	0.590	0.543
Our Model	24.64	0.735	0.272	30.31	0.923	0.234	26.53	0.884	0.263
	GPU-hr ↓	#Prim.	FPS ↑	GPU-hr ↓	#Prim.	FPS ↑	GPU-hr ↓	#Prim.	FPS ↑
3DGS [31]	0.609	7.4 M	145	0.451	1.5 M	251	0.336	5.0 M	535
EVER [40]	1.077	7.4 M	36	1.005	1.5 M	66	0.861	5.0 M	41
Triangle Splatting [24]	0.612	4.7 M	87	0.558	2.3 M	137	0.313	2.2 M	200
SVRaster [56]	0.194	9.4 M	193	0.245	8.2 M	137	0.160	7.6 M	327
RadFoam [19]	1.587	3.3 M	234	1.640	2.9 M	162	2.816	2.4 M	353
Our Model	4.515	9.3 M	240	3.457	4.2 M	384	2.575	4.8 M	475

Table 1. Results on the 5 outdoor scenes and 4 indoor scenes from the mip-NeRF 360 dataset [4], and 4 scenes from a combined Deep-Blending [23] and Tanks&Temples [32] dataset. "GPU-hr" indicates the number of GPU hours required to train a model on an NVIDIA RTX4090. FPS results are computed at the test resolution of each dataset. Note that our FPS results include time to display the results to the frame buffer and indicate actual performance in use, while all others capture only render time.

	PSNR ↑	SSIM ↑	LPIPS ↓
No Total Var Splitting	27.59	0.825	0.312
No SSIM Splitting	26.92	0.751	0.417
Constant Color	27.68	0.813	0.324
No Instant-NGP	27.25	0.787	0.360
No Downweighting	27.68	0.826	0.307
No Centroid	27.79	0.826	0.310
Our Method	27.89	0.830	0.301

Table 2. We evaluate the impact of different splitting criteria, the benefit of linear color ramps (versus constant color), and various methods for assigning attributes to primitives.

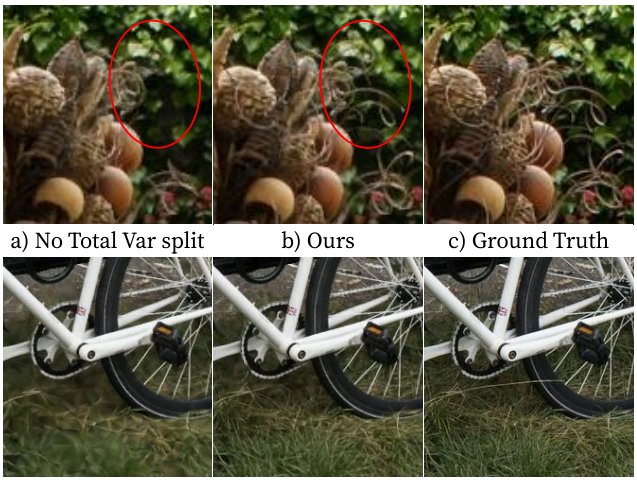

Figure 5. Qualitative results of our ablation over our splitting methods. SSIM splitting handles texture, while total variance splitting handles thin structures. Red circles highlight missing detail

to the vertices works poorly due to topology flips, as expected. Ablating the downweighting used after in interpolation ("No Downweighting") hurts slightly. ("No Centroid") is a test where we used the circumcenter instead of the centroid when querying the iNGP to get the value of each tet. As described in Sec. 4, the circumcenter is mathematically smooth but has worse numerical conditioning, a tradeoff that this experiment shows to be disadvantageous.

In Figure 6 we analyze different shader implementations to measure the impact of each choice. "Tile-based" is a naive tile-based renderer that we adapted from the 3DGS SlangTorch codebase, and "Instanced" is a hardware triangle rasterizer with duplicated loads. We see that switching from the tile-based renderer to the instanced hardware triangle rasterizer that utilizes vertex and fragment shaders yields a significant performance boost. As detailed in Sec. 4, the standard rasterization pipeline incurs redundant primitive loads in the vertex shader. We see that using our mesh shader pipeline eliminates this redundancy and achieves another significant performance gain.

5.1. Capabilities

Our model enables a variety of qualitative capabilities, all of which are shown in Figure 7. We also provide a visual demonstration of Boolean-based editing (removing a cube, then adding one), in Fig 8.

Lens Models & Ray Tracing Because our representation is highly general, we were able to straightforwardly implement a ray-tracer based on OptiX, by leveraging the transparency method from 3DGRT [44]. Crucially, our tetrahedral mesh allows us to utilize built-in hardware triangle intersection. We achieve 84 FPS on the MipNeRF 360 out-

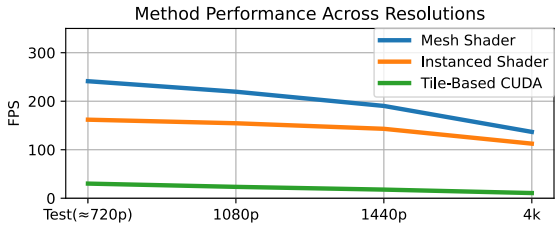

Figure 6. We plot the frames per second across a wide range of resolutions that are common in modern hardware. We show how rendering tetrahedra with a tile-based approach similar to 3DGS is significantly slower than hardware-based rasterizers. We also show how mesh shaders improve performance compared to instance-based rendering by reducing the number of duplicate memory loads.

door scenes and 190 FPS on the indoor scenes, at test resolution (roughly 720p on outdoor scenes, less than 1080p on indoor scenes). At comparable primitive counts, we achieve a 17% speed increase relative to Radiant Foam's ray tracer.

Surface mesh extraction Our volumetric representation can be converted into a surface mesh using a straightforward procedure: We measure the peak color contribution of each tetrahedron across all pixels and all training views. We use this quantity to then threshold the radiance mesh by removing the tetrahedrons with a smaller peak color contribution than 0.1. We then identify the connected components of all tetrahedra and construct a surface mesh for each one.

Deformation and Simulation Finite element analysis and various simulation techniques like Position Based Dynamics (PBD) [39, 45] are proven to be very useful and practical in interactive graphics applications. We demonstrate in Fig. 7 that a radiance mesh can directly use PBD and xPDB techniques to create interactive physical simulators, we also provide recorded interactive sessions in our supplemental video. This is a direct consequence of the nature of radiance meshes, as we are using the primal form of the Voronoi diagram — a simple Tetrahedral mesh. Previous work [19] that use the dual form of a Voronoi diagram only store, optimize, and render the generators of each cell. Animating these generator sites would cause non-trivial topological changes to the cell shapes.

6. Conclusion

We have presented radiance meshes, a novel hybrid radiance field representation that leverages the Delaunay triangulation to surpass the speed and flexibility of fast particle-based approaches, while still providing the physical consistency, and the lack of temporal artifacts of much slower neural field approaches. To accomplish this, we have presented a novel way to parameterize the attributes of the tetrahedra yielded by a Delaunay triangulation using an Instant-NGP. We additionally presented techniques for tetrahedral densification and hardware accelerated tetrahedral rendering. We have shown that this flexible backwards-compatible representation enables a variety of standard editing applications and physics-based simulation approaches, in addition to supporting arbitrary lens distortions. We believe that our approach represents a compelling combination of highly desirable properties for radiance field models, and we hope that this work will bring the community closer to widespread adoption of radiance field techniques in user-facing computer graphics applications.

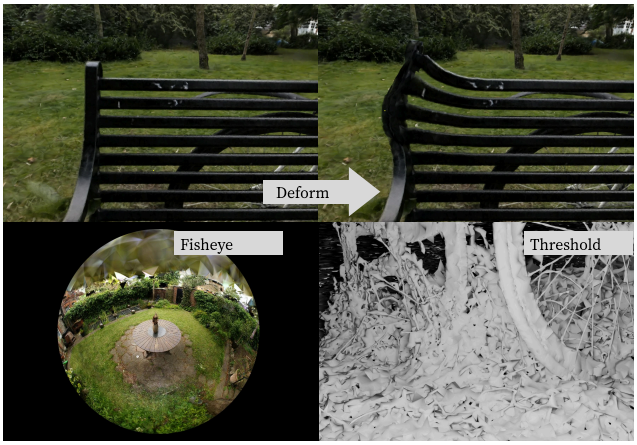

Figure 7. Radiance meshes enable a variety of useful capabilities. Bottom left: our splatting method supports fisheye during training because power sorting supports any camera where rays share a common origin. Bottom right: Blender rendering of a surface mesh of the *bicycle* scene obtained by thresholding the scene. Top right: To simulate physics in realtime, we apply xPBD [39] to the radiance mesh vertices.

Figure 8. Editing demonstration. Left: Original. Middle: Delete the same cube. Right: Add a red cube. We perform these edits by adding points to the surface of the edit volume, re-triangulating the scene, then modifying the cell values.

Acknowledgements AM is funded in part by the USACE Engineer Research and Development Center Cooperative Agreement W9132T-22-2-0014. We would like to thank the GraphDeco team at Inria, for providing a stimulating research environment during the final three months of this work.

References

[1] Kara-Ali Aliev, Artem Sevastopolsky, Maria Kolos, Dmitry Ulyanov, and Victor Lempitsky. Neural Point-Based Graphics. *ECCV*, 2020. 2

[2] Sai Bangaru, Lifan Wu, Tzu-Mao Li, Jacob Munkberg, Gilbert Bernstein, Jonathan Ragan-Kelley, Fredo Durand, Aaron Lefohn, and Yong He. SLANG.D: Fast, Modular and Differentiable Shader Programming. *SIGGRAPH Asia*, 2023. 5

[3] Jonathan T. Barron, Ben Mildenhall, Matthew Tancik, Peter Hedman, Ricardo Martin-Brualla, and Pratul P. Srinivasan. Mip-NeRF: A Multiscale Representation for Anti-Aliasing Neural Radiance Fields. *ICCV*, 2021. 2

[4] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields. *CVPR*, 2022. 4, 5, 7

[5] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Zip-NeRF: Anti-Aliased Grid-Based Neural Radiance Fields. *ICCV*, 2023. 2, 3, 4

[6] Eric R Chan, Connor Z Lin, Matthew A Chan, Koki Nagano, Boxiao Pan, Shalini De Mello, Orazio Gallo, Leonidas J Guibas, Jonathan Tremblay, Sameh Khamis, et al. Efficient Geometry-Aware 3D Generative Adversarial Networks. *CVPR*, 2022. 2

[7] Subrahmanyan Chandrasekhar. *Radiative Transfer*. Courier Corporation, 1960. 3

[8] Danpeng Chen, Hai Li, Weicai Ye, Yifan Wang, Weijian Xie, Shangjin Zhai, Nan Wang, Haomin Liu, Hujun Bao, and Guofeng Zhang. PGSR: Planar-based Gaussian Splatting for Efficient and High-Fidelity Surface Reconstruction. *Visualization and Computer Graphics*, 2024. 2

[9] Zhiqin Chen, Thomas Funkhouser, Peter Hedman, and Andrea Tagliasacchi. MobileNeRF: Exploiting the Polygon Rasterization Pipeline for Efficient Neural Field Rendering on Mobile Architectures. *CVPR*, 2023. 2

[10] Jaehoon Choi, Yonghan Lee, Hyungtae Lee, Heesung Kwon, and Dinesh Manocha. MeshGS: Adaptive Mesh-Aligned Gaussian Splatting for High-Quality Rendering. *ACCV*, 2024. 2

[11] Mike Cyrus and Jay Beck. Generalized Two- and Three-Dimensional Clipping. *Computers & Graphics*, 1978. 3, 2

[12] B Delaunay. Sur la Sphère Vide. *Bulletin de l'Académie des Sciences de l'URSS, Classe des sciences mathématiques et naturelles*, 1934. 3

[13] Herbert Edelsbrunner. An acyclicity theorem for cell complexes in d dimensions. *Symposium on Computational geometry*, 1989. 2, 3

[14] Tim Elsner, Victor Czech, Julia Berger, Zain Selman, Isaak Lim, and Leif Kobbelt. Adaptive Voronoi NeRFs. *arXiv:2303.16001*, 2023. 2

[15] Sara Fridovich-Keil, Alex Yu, Matthew Tancik, Qinhong Chen, Benjamin Recht, and Angjoo Kanazawa. Plenoxels: Radiance fields without neural networks. *CVPR*, 2022. 2

[16] Lin Gao, Jie Yang, Bo-tao Zhang, Jia-mu Sun, Yu-jie Yuan, Hongbo Fu, and Yu-kun Lai. Real-time Large-scale Deformation of Gaussian Splatting. *ACM Transactions on Graphics (TOG)*, 2024. 2

[17] Xiangjun Gao, Xiaoyu Li, Yiyu Zhuang, Qi Zhang, Wenbo Hu, Chaopeng Zhang, Yao Yao, Ying Shan, and Long Quan. Mani-GS: Gaussian Splatting Manipulation with Triangular Mesh. *CVPR*, 2025. 2

[18] Joachim Georgii and Rudiger Westermann. A Generic and Scalable Pipeline for GPU Tetrahedral Grid Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 2006. 3

[19] Shrisudhan Govindarajan, Daniel Rebain, Kwang Moo Yi, and Andrea Tagliasacchi. Radiant Foam: Real-Time Differentiable Ray Tracing. *ICCV*, 2025. 2, 5, 6, 7, 8

[20] 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. *SIGGRAPH Asia*, 2025. 2

[21] Minghao Guo, Bohan Wang, Kaiming He, and Wojciech Matusik. TetSphere Splatting: Representing High-Quality Geometry with Lagrangian Volumetric Meshes. *arXiv:2405.20283*, 2024. 3

[22] 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. *CVPR*, 2024. 2

[23] Peter Hedman, Julien Philip, True Price, Jan-Michael Frahm, George Drettakis, and Gabriel Brostow. Deep blending for free-viewpoint image-based rendering. *ACM Transactions on Graphics (TOG)*, 2018. 5, 7

[24] Jan Held, Renaud Vandeghen, Adrien Deliege, Abdullah Hamdi, Silvio Giancola, Anthony Cioppa, Andrea Vedaldi, Bernard Ghanem, Andrea Tagliasacchi, and Marc Van Droogenbroeck. Triangle Splatting for Real-Time Radiance Field Rendering. *arXiv:2505.19175*, 2025. 2, 7

[25] Jan Held, Renaud Vandeghen, Abdullah Hamdi, Adrien Deliege, Anthony Cioppa, Silvio Giancola, Andrea Vedaldi, Bernard Ghanem, and Marc Van Droogenbroeck. 3D Convex Splatting: Radiance Field Rendering with 3D Smooth Convexes. *CVPR*, 2025. 5

[26] Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. 2D Gaussian Splatting for Geometrically Accurate Radiance Fields. *SIGGRAPH 2024*, 2024.

[27] Yi-Hua Huang, Ming-Xian Lin, Yang-Tian Sun, Ziyi Yang, Xiaoyang Lyu, Yan-Pei Cao, and Xiaojuan Qi. Deformable Radial Kernel Splatting. *CVPR*, 2025. 2

[28] Ferran Hurtado, Marc Noy, and Jorge Urrutia. Flipping Edges in Triangulations. *Symposium on Computational Geometry*, 1996. 1

[29] Michael S Karasick, Derek Lieber, Lee R. Nackman, and VT Rajan. Visualization of Three-Dimensional Delaunay Meshes. *Algorithmica*, 1997. 2, 3

[30] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3D Mesh Renderer. *CVPR*, 2018. 2

[31] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Transactions on Graphics (TOG)*, 2023. 1, 2, 5, 6, 7

[32] Arno Knapitsch, Jaesik Park, Qian-Yi Zhou, and Vladlen Koltun. Tanks and Temples: Benchmarking Large-Scale

Scene Reconstruction. *ACM Transactions on Graphics (TOG)*, 2017. 5, 7

[33] Christoph Lassner and Michael Zollhofer. Pulsar: Efficient Sphere-Based Neural Rendering. *CVPR*, 2021. 2

[34] Haolin Li, Jinyang Liu, Mario Sznaier, and Octavia Camps. 3D-HGS: 3D Half-Gaussian Splatting. *arXiv:2406.02720*, 2024. 2

[35] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Transactions on Graphics (TOG)*, 2018. 2

[36] Ancheng Lin, Yusheng Xiang, Paul Kennedy, and Jun Li. Direct Learning of Mesh and Appearance via 3D Gaussian Splatting. *arXiv:2405.06945*, 2024. 2

[37] Rong Liu, Dylan Sun, Meida Chen, Yue Wang, and Andrew Feng. Deformable Beta Splatting. *SIGGRAPH*, 2025. 2

[38] Stephen Lombardi, Tomas Simon, Gabriel Schwartz, Michael Zollhoefer, Yaser Sheikh, and Jason Saragih. Mixture of volumetric primitives for efficient neural rendering. *ACM TOG*, 2021. 2

[39] Miles Macklin, Matthias Müller, and Nuttapong Chentanez. Xpbd: position-based simulation of compliant constrained dynamics. *International Conference on Motion in Games*, 2016. 2, 8

[40] Alexander Mai, Peter Hedman, George Kopanas, Dor Verbin, David Futschik, Qiangeng Xu, Falko Kuester, Jonathan T Barron, and Yinda Zhang. EVER: Exact Volumetric Ellipsoid Rendering for Real-time View Synthesis. *ICCV*, 2025. 2, 5, 6, 7

[41] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar Functions. *Workshop on Volume Visualization*, 1990. 2, 3

[42] J. L. Meijering. Interface area, edge length, and number of vertices in crystal aggregates with random nucleation. *Philips Research Reports*, 1953. 2

[43] 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. *ECCV*, 2020. 1, 2

[44] 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. *SIGGRAPH Asia*, 2024. 2, 7

[45] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 2007. 8

[46] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant Neural Graphics Primitives with a Multiresolution Hash Encoding. *ACM Transactions on Graphics (TOG)*, 2022. 2, 4

[47] Ken Museth and Santiago Lombeyda. TetSplat Real-Time Rendering and Volume Clipping of Large Unstructured Tetrahedral Meshes. *IEEE Visualization*, 2004. 3

[48] Udo Pachner. Pl homeomorphic manifolds are equivalent by elementary shellings. *European Journal of Combinatorics*, 1991. 1

[49] Shenhan Qian, Tobias Kirschstein, Liam Schoneveld, Davide Davoli, Simon Giebenhain, and Matthias Nießner. GaussianAvatars: Photorealistic Head Avatars with Rigged 3D Gaussians. *CVPR*, 2024. 2

[50] Haoxuan Qu, Zhuoling Li, Hossein Rahmani, Yujun Cai, and Jun Liu. DisC-GS: Discontinuity-aware Gaussian Splatting. *NeurIPS*, 37, 2024. 2

[51] Lukas Radl, Michael Steiner, Mathias Parger, Alexander Weinrauch, Bernhard Kerbl, and Markus Steinberger. StopThePop: Sorted Gaussian Splatting for View-Consistent Real-time Rendering, 2024. 2

[52] 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 Transactions on Graphics (TOG)*, 2024. 2

[53] Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. Revising Densification in Gaussian Splatting. *ECCV*, 2024. 4

[54] Johannes Lutz Schönberger and Jan-Michael Frahm. Structure-from-motion revisited. *CVPR*, 2016. 3

[55] Zhijing Shao, Zhaolong Wang, Zhuang Li, Duotun Wang, Xiangru Lin, Yu Zhang, Mingming Fan, and Zeyu Wang. SplattingAvatar: Realistic Real-Time Human Avatars with Mesh-Embedded Gaussian Splatting. *CVPR*, 2024. 2

[56] Cheng Sun, Jaesung Choe, Charles Loop, Wei-Chiu Ma, and Yu-Chiang Frank Wang. Sparse Voxels Rasterization: Real-time High-fidelity Radiance Field Rendering. *CVPR*, 2025. 2, 5, 6, 7

[57] Thibault Tricard. Interval shading: using mesh shaders to generate shading intervals for volume rendering. *Computer Graphics and Interactive Techniques*, 2024. 3, 5

[58] Nicolas von Lützow and Matthias Nießner. LinPrim: Linear Primitives for Differentiable Volumetric Rendering. *arXiv:2501.16312*, 2025. 3

[59] Joanna Waczyńska, Piotr Borycki, Sławomir Tadeja, Jacek Tabor, and Przemysław Spurek. GaMeS: Mesh-Based Adapting and Modification of Gaussian Splatting. *arXiv:2402.01459*, 2024. 2

[60] Jiepeng Wang, Yuan Liu, Peng Wang, Cheng Lin, Junhui Hou, Xin Li, Taku Komura, and Wenping Wang. GausSurf: Geometry-Guided 3D Gaussian Splatting for Surface Reconstruction. *arXiv:2411.19454*, 2024. 2

[61] P Wang, L Liu, Y Liu, C Theobalt, T Komura, and WP Wang. NeuS: Learning Neural Implicit Surfaces by Volume Rendering for Multi-view Reconstruction. *Neurips*, 2021. 2

[62] Olivia Wiles, Georgia Gkioxari, Richard Szeliski, and Justin Johnson. SynSin: End-to-end View Synthesis from a Single Image. *CVPR*, 2020. 2

[63] Qiangeng Xu, Zexiang Xu, Julien Philip, Sai Bi, Zhixin Shu, Kalyan Sunkavalli, and Ulrich Neumann. Point-NeRF: Point-based Neural Radiance Fields. *CVPR*, 2022. 2

[64] Lior Yariv, Jiatao Gu, Yoni Kasten, and Yaron Lipman. Volume rendering of neural implicit surfaces. *NeurIPS*, 2021. 2

[65] 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. *SIGGRAPH*, 2023. 2

[66] Zehao Yu, Torsten Sattler, and Andreas Geiger. Gaussian Opacity Fields: Efficient Adaptive Surface Reconstruction in Unbounded Scenes. *ACM Transactions on Graphics (ToG)*, 2024. 2

[67] Ziyu Zhang, Binbin Huang, Hanqing Jiang, Liyang Zhou, Xiaojun Xiang, and Shunhan Shen. Quadratic Gaussian Splatting for Efficient and Detailed Surface Reconstruction. *arXiv:2411.16392*, 2024. 2

[68] Chengwei Zheng, Lixin Xue, Juan Zarate, and Jie Song. GauSTAR: Gaussian Surface Tracking and Reconstruction. *CVPR*, 2025. 2