VoGE: A Differentiable Volume Renderer using Gaussian Ellipsoids for Analysis-by-Synthesis

Anonymous authors
Paper under double-blind review

Abstract

Differentiable rendering allows the application of computer graphics on vision tasks, e.g. object pose and shape fitting, via analysis-by-synthesis, where gradients at occluded regions are important when inverting the rendering process. To obtain those gradients, state-of-the-art (SoTA) differentiable renderers Liu et al. (2019); Yifan et al. (2019) use rasterization to collect a set of nearest components for each pixel and aggregate them based on the viewing distance. In this paper, we propose VoGE, which uses ray tracing to capture nearest components with their volume density distributions on the rays and aggregates via integral of the volume densities based on Gaussian ellipsoids, which brings more efficient and stable gradients. To efficiently render via VoGE, we propose an approximate close-form solution for the volume density aggregation and a coarse-to-fine rendering strategy. Finally, we provide a CUDA implementation of VoGE, which gives a competitive rendering speed in comparison to PyTorch3D Ravi et al. (2020). Quantitative and qualitative experiment results show VoGE outperforms SoTA counterparts when applied to various vision tasks, *e.g.*, object pose estimation, shape/texture fitting, and occlusion reasoning. The VoGE code will be publicly available.

1 Introductions

Recently, significant advances in computer vision tasks have been achieved, through the integration of deep learning and computer graphics, *e.g.* in pose estimation Wang et al. (2020a), 3D reconstruction Zhang et al. (2021), or texture estimation Bhattad et al. (2021). Different from traditional graphics applications, the differentiability of the rendering process is of utmost importance for inverse graphics applications in computer vision. Differentiable renderers compute the gradients w.r.t. the image formation process, and hence enable to broadcast cues from 2D images towards the parameters of computer graphics models, such as the camera parameters, or the object geometry and texture. Such an ability is also essential when embedding graphics models into deep neural networks. In this work, we focus on developing a differentiable renderer using explicit object representations, which can be either used separately for image generation or for serving as 3D aware neural network layers.

The traditional rendering process typically involves a naive rasterization Kato et al. (2018), which projects components onto the image plane and only captures the nearest components for each pixel. However, this process eliminates the cues from the occluded components and blocks gradients toward them. Liu *et al.* Liu et al. (2019) provide a solution to this problem by tracking a set of nearest components for each image pixel instead of only one, and aggregates based on the viewing distance. However such an approach is sensitive to the settings of near and far distance, while also introducing incorrect transparencies when components are near to each other. This limitation is introduced by the ambiguity during the rasterization process, which assumes components do not overlap with each other and are ordered front to back along the viewing direction Zwicker et al. (2001). Such assumption raise a paradox for differentiable rendering since during optimization components are necessary to overlap with each other when they change the order along viewing direction. In order to resolve this assumption, we propose to record the volume density distributions instead of simply recording the viewing distance, since such distributions provide cues on occlusion and interaction of components when they overlapped.

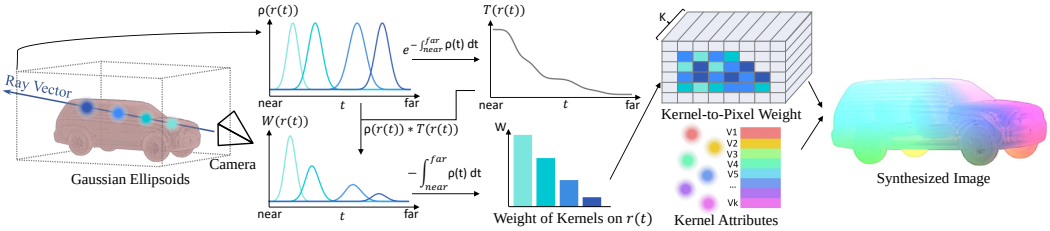


Figure 1: VoGE conducts ray tracing and volume density aggregation. Given the Gaussian Ellipsoids, *i.e.* a set of anisotropic 3D Gaussian reconstruction kernels, VoGE first samples rays $r(t)$. And along each ray, VoGE traces the density distribution of each ellipsoid $\rho_k(r(t))$ respectively. Then occupancy $T(r(t))$ is accumulated via density aggregation along the ray. The observation of each Gaussian ellipsoid kernels W_k is computed via integral of reweighted per-kernel volume density $W_k(r(t))$. Finally, VoGE synthesizes the image using the computed W_k on each pixel to interpolate per kernel attributes. In practice, the density aggregation process is bootstrapped via approximate close-form solutions.

Recent works Mildenhall et al. (2020); Schwarz et al. (2020) show the power of volume rendering with high-quality occlusion reasoning and differentiability, which benefits from the usage of ray tracing volume densities Kajiya & Von Herzen (1984). However, the rendering process in those works relies on implicit object representations which limits the modifiability and interpretability. Back in the 90s, graphics researchers developed the splatting method Westover (1990); Zwicker et al. (2001) which reconstruct objects using volumetric Gaussian kernels and renders based on a simplification of the ray tracing volume densities method. Unfortunately, splatting methods were designed for graphics rendering without considering the differentiability and approximate the ray tracing volume densities using rasterization. Inspired by both approaches Zwicker et al. (2001); Liu et al. (2019), we propose VoGE using 3D Gaussians to represent objects, which give soften boundary of components. Specifically, VoGE traces components along viewing rays as a density function, which gives a probability of observation along the viewing direction.

In VoGE rendering pipeline, the ray tracing method is designed to replace rasterization, and a better aggregation function is developed based on integral of traced volume densities functions. As Figure 1 shows, VoGE uses a set of Gaussian ellipsoids to reconstruct the object in 3D space. Each Gaussian ellipsoid is indicated with a center location $\mathbf{M}$, and a spatial variance $\mathbf{\Sigma}$. During rendering, we first sample viewing rays with the camera configuration. Then for each ray, we trace the volume density of each ellipsoid as a function of distance along the ray respectively. We further compute the occupancy along the ray via an integral of the volume density and reweight the contribution of each ellipsoid. Finally, we interpolate the attribute of each reconstruction kernel with the kernel-to-pixel weights into an image. In practice, we propose an approximate close-form solution, which makes it feasible to compute the density aggregation without computational heavy operation, *e.g.*, integral, cumulative sum. Benefitting from the advanced differentiability, VoGE obtain satisfied performance on various vision tasks.

In summary, the contribution of VoGE includes:

1. A ray tracing method that traces each component along viewing ray as density functions. VoGE ray tracing is a replacement for rasterization.

2. An aggregation function that integral the traced density functions to reason the occlusion between components on each viewing ray, with differentiability toward both visible and invisible components.

3. A differentiable CUDA implementation with competitive rendering speed. VoGE can be inserted into neural networks via our PyTorch APIs.

4. Exceptional performance on various vision tasks. Quantitative results demonstrate that VoGE significantly outperforms concurrent state-of-the-art differentiable renderer on in-wild object pose estimation tasks.

Table 1: Comparison with state-of-the-art differentiable renderers. Similar to NeRF but different from previous graphics renderers, VoGE uses ray tracing to record volume densities on each ray for each component, and aggregate them via integral of volume densities.

Method	Representation	Component	Component Tracking	Aggregation
NMR Kato et al. (2018)	explicit	mesh	rasterization	none
SoftRas Liu et al. (2019)	explicit	mesh	rasterization	distance
PyTorch3D Ravi et al. (2020)	explicit	mesh/points	rasterization	distance
DSS Yifan et al. (2019)	explicit	2D Gaussian	rasterization	distance
Pulsar Lassner & Zollhofer (2021)	explicit	sphere	rasterization	distance
VoGE (ours)	explicit	3D Gaussian	ray tracing	volume density
NeRF Mildenhall et al. (2020)	implicit	—	ray tracing	volume density

2 Related Works

Volume Rendering. In the 1980s, Blinn Blinn (1982) introduces the volume density representation, which simulates the physical process of light interacting with matter. Kajiya and Herzen Kajiya & Von Herzen (1984) develop the ray tracing volume density aggregation algorithm, which renders the volume density via light scattering equations. However, obtaining the contiguous volume density function is infeasible in practice. Current approaches Mildenhall et al. (2020); Niemeyer et al. (2020); Mescheder et al. (2019); Genova et al. (2020) use implicit functions, *e.g.*, neural networks, as object representations. Though those implicit representations give a satisfying performance, such representations are lacking interpretability and modifiability, which may limit their usage in analysis tasks. In this work, we provide a solution that utilizes explicit representation while rendering with the ray tracing volume density aggregation.

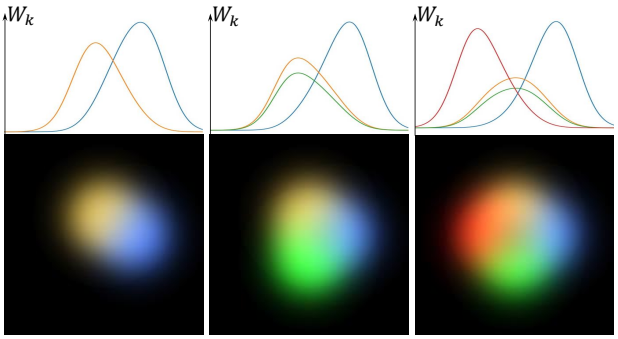
Kernel Reconstruction of 3D Volume. Westover Westover (1990) introduces the volume reconstruction kernel, which decomposes a 3D volume into a sum of homogeneous components. Zwicker *et al.*Zwicker et al. (2001) introduces the elliptical Gaussian kernel and show such reconstruction gives satisfied shape approximation. However, both approaches conduct non-differentiable rendering and use rasterization to approximate the ray tracing process.

Differentiable Renderer using Graphics. Graphics renderers use explicit object representations, which represent objects as a set of isotropic components. As Table 1 shows, concurrent differentiable graphics renderers use rasterization to track components. In order to compute gradients across components boundaries, some approaches Loper & Black (2014); Kato et al. (2018); Liu et al. (2017) manually create the gradients while others Liu et al. (2019); Yifan et al. (2019) use components with soft boundaries to allow gradients flow. Whereas to differentiate toward those occluded components, current differentiable renders Liu et al. (2019); Yifan et al. (2019); Lassner & Zollhofer (2021) aggregate tracked components via viewing distance. However, all existing graphics renderers ignore the density distributions when conducting aggregation, which will introduce confusion while limiting differentiability.

Renderer for Deep Neural Features. Recent works demonstrate exceptional performance for rendering deep neural features. Specifically, works on object pose estimation Wang et al. (2020a); Iwase et al. (2021) demonstrate rendering on deep neural features benefits the optimization process in render-and-compare. Niemeyer *et al.*Niemeyer & Geiger (2021) show rendering deep neural features also helps image generation tasks. In our work, we show VoGE benefits rendering using deep neural features via a better reconstruction of the spatial distribution of deep neural features.

3 Volume Renderer for Gaussian Ellipsoids

In this section, we describe VoGE rendering pipeline that renders explicit 3D volume representations into images with a certain camera configuration. Section 3.1 introduces the volume rendering. Section 3.2 describes the kernel reconstruction of the 3D volume using Gaussian ellipsoids. In Section 3.3, we propose the rendering pipeline for Gaussian ellipsoids via an approximate closed-form solution of ray tracing and volume density aggregation. Section 3.4 discusses embedding the proposed pipeline into deep neural networks.



(a) K=2 (b) K=3 (c) K=4

Figure 2: Rendering with increasing numbers of Gaussian Ellipsoids. Top: the kernel-to-pixel weight along the median row on the image, the colors demonstrate each corresponded Gaussian ellipsoids. Bottom: the rendered RGB image. Note VoGE resolves occlusion naturally in a contiguous way.

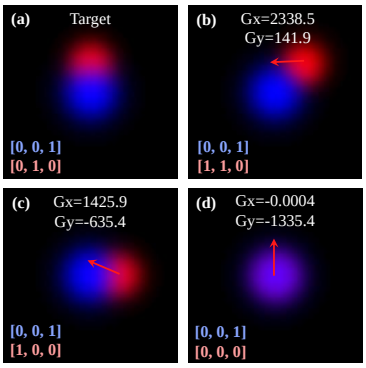


Figure 3: Computing gradient of $\mathbf{M}$ when rendering two ellipsoids. The colored numbers below indicate the $\mathbf{M}$ of each ellipsoids. The red arrow and G_x, G_y show the $\frac{\partial(\mathbf{I}-\hat{\mathbf{I}})^2}{\partial \mathbf{M}_{\mathbf{red}}}$.

3.1 Volume Rendering

Different from the surface based shape representations, in volume rendering, objects are represented using continuous volume density functions. Specifically, for each point in the volume, we have a corresponded density $\rho(x, y, z)$ with emitted color $c(x, y, z) = (r, g, b)$, where (x, y, z) donates location of the point in the 3D space. During the volume rendering process, light scattering equation for volume density Kajiya & Von Herzen (1984) provides a mechanism to compute the observed color $C(\mathbf{r})$ along a ray $\mathbf{r}(t) = (x(t), y(t), z(t))$:

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\rho(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t))dt, \text{where } T(t) = \exp\left(-\tau \int_{t_n}^{t} \rho(\mathbf{r}(s))ds\right) \tag{1}$$

where τ is a coefficient that determines the rate of absorption, t_n and t_f donates the near and far bound alone the ray.

3.2 Gaussian Ellipsoid Reconstruction Kernel

Due to the difficulty of obtaining contiguous function of the volume density and enormous computation cost when calculating the integral, kernel reconstruction is introduced to conduct volume rendering in a computationally efficient way Westover (1990). The reconstruction decomposes the contiguous volume into a set of homogeneous kernels, while each kernel can be described with a simple density function. We use volume ellipsoidal Gaussians as the reconstruction kernels. Specifically, we reconstruct the volume with a sum of ellipsoidal Gaussians:

$$\rho(\mathbf{X}) = \sum_{k=1}^{K} \frac{1}{\sqrt{2\pi \cdot ||\mathbf{\Sigma}_k||_2}} e^{-\frac{1}{2}(\mathbf{X}-\mathbf{M}_k)^T \cdot \mathbf{\Sigma}_k^{-1} \cdot (\mathbf{X}-\mathbf{M}_k)} \tag{2}$$

where K is the total number of Gaussian kernels, $\mathbf{X} = (x, y, z)$ is an arbitrary location in the 3D space. The $\mathbf{M}_k$, a 3×1 vector, is the center of k-th ellipsoidal Gaussians kernel. Whereas the $\mathbf{\Sigma}_k$ is a 3×3 spatial variance matrix, which controls the direction, size and shape of k-th kernel. Also, following previous works Zwicker et al. (2001), we assume that the emitted color is approximately constant inside each reconstruction kernel $\mathbf{c}(\mathbf{r}(t)) = \mathbf{c}_k$.

The anisotropic Gaussian ellipsoid kernels can reconstruct arbitrary 3D shapes, which makes it feasible to convert common representations into Gaussian ellipsoids. When converting meshes, we compute $\mathbf{\Sigma}_k$ based on the distance from k-th vertex to its neighbors. Point clouds can be easily converted via homogeneous isotropic Gaussians. Refer to Appendix A.4 for details.

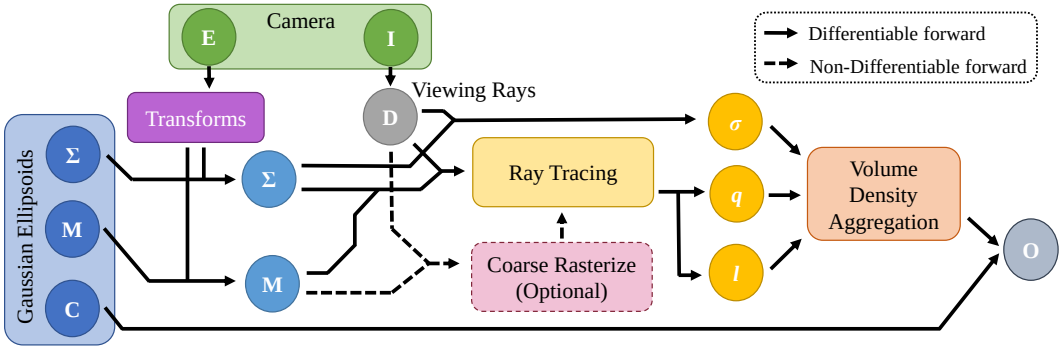


Figure 4: The forward process for VoGE rendering. The camera described with the extrinsic matrix $\mathbf{E}$ composed with $\mathbf{R}$ and $\mathbf{T}$, as well as the intrinsic matrix $\mathbf{I}$ composed with F and O_x, O_y. Given Gaussian Ellipsoids, VoGE renderer synthesizes an image $\mathbf{O}$.

3.3 Render Gaussian Ellipsoids

Figure 4 shows the rendering process for VoGE. VoGE takes inputs of a perspective camera and Gaussian ellipsoids to render images, while computing gradient towards both camera and Gaussian ellipsoids (shows in Figure 2 and 3).

Viewing transformation utilize the extrinsic configuration $\mathbf{E}$ of the camera to transfer the Gaussian ellipsoids from the object coordinate to the camera coordinate. Let $\mathbf{M}_k^o$ donate centers of ellipsoids in the object coordinate. Following the standard approach, we compute the centers in the camera coordinate:

$$\mathbf{M}_k = \mathbf{R} \cdot \mathbf{M}_k^o + \mathbf{T} \tag{3}$$

where $\mathbf{R}$ and $\mathbf{T}$ are the rotation and translation matrix included in $\mathbf{E}$. Since we consider 3D Gaussian Kernels are ellipsoidal, observations of the variance matrices are also changed upon camera rotations:

$$\mathbf{\Sigma}_k^{-1} = \mathbf{R}^T \cdot (\mathbf{\Sigma}_k^o)^{-1} \cdot \mathbf{R} \tag{4}$$

Perspective rays indicate the viewing direction in the camera coordinate. For each pixel, we compute the viewing ray under the assumption that the camera is fully perspective:

$$\mathbf{r}(t) = \mathbf{D} * t = \begin{bmatrix} \frac{i-O_y}{F} & \frac{j-O_x}{F} & 1 \end{bmatrix}^T * t \tag{5}$$

where $p = (i, j)$ is the pixel location on the image, O_x, O_y is the principal point of the camera, F is the focal length, $\mathbf{D}$ is the ray direction vector.

Ray tracing observes the volume densities of each ellipsoid along the ray $\mathbf{r}$ respectively. Note the observation of each ellipsoid is a 1D Gaussian function along the viewing ray (for details, refer to Appendix A.1):

$$\rho_m(\mathbf{r}(s)) = \exp(q_m - \frac{(s - l_m)^2}{2 \cdot \sigma_m^2}) \tag{6}$$

where $l_m = \frac{\mathbf{M}_m^T \cdot \mathbf{\Sigma}_m^{-1} \cdot \mathbf{D} + \mathbf{D}^T \cdot \mathbf{\Sigma}_m^{-1} \cdot \mathbf{M}_m}{2 \cdot \mathbf{D}^T \cdot \mathbf{\Sigma}_m^{-1} \cdot \mathbf{D}}$ is the length along the ray that gives peak activation for m-th kernel. $q_m = -\frac{1}{2}\mathbf{V}_m^T \cdot \mathbf{\Sigma}_m^{-1} \cdot \mathbf{V}_m, where\ \mathbf{V}_m = \mathbf{M}_m - l_m \cdot \mathbf{D}$ computes peak density of m-th kernel alone the ray. The 1D variance is computed via $\frac{1}{\sigma_m^2} = \mathbf{D}^T \cdot \mathbf{\Sigma}_m^{-1} \cdot \mathbf{D}$. Thus, when tracing along each ray, we only need to record l_m, q_m and σ_m for each ellipsoid respectively.

Volume density aggregation computes the observation along the ray $\mathbf{r}$. As the Figure 1 shows, different from other generic renderers, which only consider the viewing distance for aggregation, VoGE aggregates all observations based on integral of volume densities along the ray. However, computing the integral using brute force is so computational inefficient that even infeasible for concurrent computation power. To resolve this, we propose an approximate closed-form solution, which conducts the computation in a both accurate and effective way.

We use the Error Function erf to compute the integral of Gaussian, since it can be computed via numerical approach directly. Specifically, with Equation 13 and Equation 5, we can calculate $T(t)$ as (for computation detail, refer to Appendix A.2):

$$T(t) = \exp(-\tau \int_{-\infty}^{t} \rho(\mathbf{r}(s))ds) = \exp(-\tau \sum_{m=1}^{K} e^{q_m} \frac{\mathrm{erf}((t - l_m)/\sigma_m) + 1}{2}) \tag{7}$$

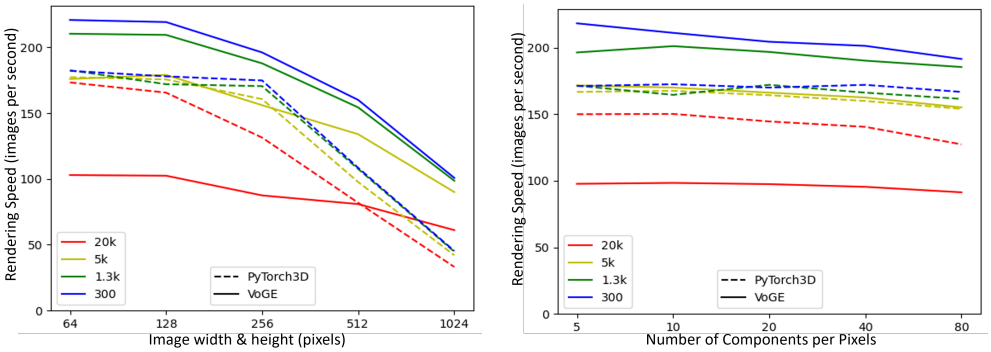


Figure 5: Comparison for rendering speeds of VoGE and PyTorch3D, reported in images per second (higher better). We evaluate the rendering speed using cuboids with different number of components (verts, ellipsoids), which illustrated using different colors, also different image sizes and numbers of component per pixel.

Now, to compute closed-form solution of the outer integral in Equation 11, we use the $T(t), t = l_k$ at the peak of $\rho(\mathbf{r}(t))$ alone the rays. Here we provide the closed-form solution for $C(\mathbf{r})$:

$$C(\mathbf{r}) = \int_{-\infty}^{\infty} T(t)\rho(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t))dt = \sum_{k=1}^{K} T(l_k)e^{q_k}\mathbf{c}_k \tag{8}$$

Note based on the assumption that distances from camera to ellipsoids are significantly larger than ellipsoid sizes, it is equivalent to set $t_n = -\infty$ and $t_f = \infty$.

Coarse-to-fine rendering. In order to improve the rendering efficiency, we implement VoGE rendering with a coarse-to-fine strategy. Specifically, VoGE renderer has an optional coarse rasterizing stage that, for each ray, selects only 10% of all ellipsoids (details in Appendix A.3). Besides, the ray tracing to the volume density aggregation also works in a coarse-to-fine manner. VoGE aggregates $K^{'}$ nearest ellipsoids among all traced kernels that gives $e^{q_k} > thr = 0.01$. We provide a CUDA NVIDIA et al. (2022) implementation of VoGE with both forward and backward function. The CUDA-VoGE is packed with an easily used "autogradable" PyTorch API.

3.4 VoGE in Neural Networks

VoGE can be easily embedded into neural networks by serving as neural sampler and renderer. As a sampler, VoGE extracts attributes α_k (*e.g.*, deep neural features, textures) from images or feature maps into kernel-correspond attributes, which is conducted via reconstructing their spatial distribution in the screen coordinates. When serving as a renderer, VoGE converts kernel-correspond attributes into images or feature maps. Since both sampling and rendering give the same spatial distribution of feature/texture, it is possible for VoGE to conduct geometry-based image-to-image transformation.

Here we discuss how VoGE samples deep neural features. Let $\mathbf{\Phi}$ donates observed features, where ϕ_p is the value at location p. Let $\mathbf{A} = \bigcup_{k=1}^{K}\{\alpha_k\}$ donates the per kernel attribute, which we want to discover during sampling. With a given object geometry $\mathbf{\Gamma} = \bigcup_{k=1}^{K}\{\mathbf{M}_k, \mathbf{\Sigma}_k\}$ and viewing rays $\mathbf{r}(p)$. The the observation formulated with conditional probability regarding α_k:

$$\phi^{'}(p) = \sum_{k=1}^{K} \mathcal{P}(\alpha_k|\mathbf{\Gamma}, \mathbf{r}(p), k)\alpha_k \tag{9}$$

Since $\mathbf{\Phi}$ is a discrete observation of a continuous distribution $\phi(p)$ on the screen, the synthesis can only be evaluated at discrete positions, *i.e.*the pixel centers. As the goal is to make $\mathbf{\Phi}^{'}$ similar as $\mathbf{\Phi}$ on all observable locations, we resolve via an inverse reconstruction:

$$\alpha_k = \sum_{p=1}^{P} \mathcal{P}(\phi(p)|\mathbf{\Gamma}, \mathbf{r}(p), p)\phi(p) = \frac{\sum_{p=1}^{P} \mathbf{W}_{p,k} * \phi_p}{\sum_{p=1}^{P} \mathbf{W}_{p,k}} \tag{10}$$

where $\mathbf{W}_{p,k} = T(l_k)e^{q_k}$ is the kernel-to-pixel weight as described in 3.1.

Table 2: Pose estimation results on the PASCAL3D+ and the Occluded PASCAL3D+ dataset. Occlusion level L0 is the original images from PASCAL3D+, while Occlusion Level L1 to L3 are the occluded PASCAL3D+ images with increasing occlusion ratios. NeMo is an object pose estimation pipeline via neural feature level render-and-compare. We compare the object pose estimation performance using different renderers, *i.e.* VoGE, Soft Rasterizer, DSS, PyTorch3D (which is used in NeMo originally).

Evaluation Metric	$ACC_{\frac{\pi}{6}} \uparrow$				$ACC_{\frac{\pi}{18}} \uparrow$				*MedErr* $\downarrow$			
Occlusion Level	L0	L1	L2	L3	L0	L1	L2	L3	L0	L1	L2	L3
Res50-General	88.1	70.4	52.8	37.8	44.6	25.3	14.5	6.7	11.7	17.9	30.4	46.4
Res50-Specific	87.6	73.2	58.4	43.1	43.9	28.1	18.6	9.9	11.8	17.3	26.1	44.0
StarMap	89.4	71.1	47.2	22.9	59.5	34.4	13.9	3.7	9.0	17.6	34.1	63.0
NeMo+SoftRas	85.3	75.2	63.0	44.3	59.7	46.7	32.1	16.8	9.1	14.8	24.0	39.3
NeMo+DSS	81.1	71.9	56.8	38.7	33.5	30.4	23.0	14.1	16.1	19.8	25.8	40.4
NeMo+PyTorch3D	86.1	76.0	63.9	46.8	61.0	46.3	32.0	17.1	8.8	13.6	20.9	36.5
NeMo+VoGE(Ours)	**90.1**	**83.1**	**72.5**	**56.0**	**69.2**	**56.1**	**41.5**	**24.8**	**6.9**	**9.9**	**15.0**	**26.3**

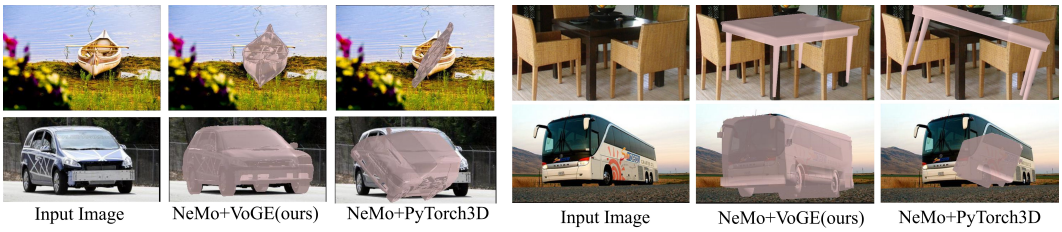


Figure 6: Qualitative object pose estimation results on PASCAL3D+ dataset. We visualize the predicted object poses from NeMo+VoGE and standard NeMo. Specifically, we use a standard mesh renderer to render the original CAD model under the predicted pose and superimpose onto the input image.

4 EXPERIMENT

We explore several applications of VoGE. In section 4.1, we study the object pose estimation using VoGE in a feature level render-and-compare pose estimator. In section 4.2, we explore texture extraction ability of VoGE. In section 4.4, we demonstrate VoGE can optimize the shape representation via multi-viewed images. Visualizations of VoGE rendering are included in Appendix B.

Rendering Speed. As Figure 5 shows, CUDA-VoGE provides a competitive rendering speed compare to state-of-the-art differentiable generic renderer when rendering a single cuboidal object.

4.1 OBJECT POSE ESTIMATION IN WILD

We evaluate the ability of VoGE when serving as a feature sampler and renderer in an object pose estimation pipeline, NeMo Wang et al. (2020a), an in-wild category-level object 3D pose estimator that conducts render-and-compare on neural feature level. NeMo utilizes PyTorch3D Ravi et al. (2020) as the feature sampler and renderer, which converts the feature maps to vertex corresponded feature vectors and conducts the inverse process. In our NeMo+VoGE experiment, we use VoGE to replace the PyTorch3D sampler and renderer via the approach described in Section 3.4.

Dataset. Following NeMo, we evaluate pose estimation performance on the PASCAL3D+ dataset Xiang et al. (2014), the Occluded PASCAL3D+ dataset Wang et al. (2020b) and the ObjectNet3D dataset Xiang et al. (2016). The PASCAL3D+ dataset contains objects in 12 man-made categories with 11045 training images and 10812 testing images. The Occluded PASCAL3D+ contains the occluded version of same images, which is obtained via superimposing occluder cropped from MS-COCO dataset Lin et al. (2014). The dataset includes three levels of occlusion with increasing occlusion rates. In the experiment on ObjectNet3D, we follow NeMo to test on 18 categories.

Evaluation Metric. We measure the pose estimation performance via accuracy of rotation error under given thresholds and median of per image rotation errors. The rotation error is defined as the difference between the predicted rotation matrix and the ground truth rotation matrix: $\Delta\left(R_{pred}, R_{gt}\right) = \frac{\left\|\log m\left(R_{pred}^{T} R_{gt}\right)\right\|_{F}}{\sqrt{2}}$

Table 3: Pose estimation results on the ObjectNet3D dataset. Evaluated via pose estimation accuracy for error under $\frac{\pi}{6}$ (higher better).

$ACC_{\frac{\pi}{6}} \uparrow$	bed	shelf	calculator	cellphone	computer	cabinet	guitar	iron	knife
StarMap	40.0	72.9	21.1	41.9	62.1	79.9	38.7	2.0	6.1
NeMo+PyTorch3D	56.1	53.7	57.1	28.2	**78.8**	**83.6**	38.8	32.3	9.8
NeMo+VoGE(Ours)	**76.8**	**83.2**	**77.8**	**50.7**	**78.8**	**83.6**	**54.6**	**45.4**	**12.1**
$ACC_{\frac{\pi}{6}} \uparrow$	oven	pen	pot	rifle	slipper	stove	toilet	tub	wheelchair
StarMap	86.9	12.4	45.1	3.0	13.3	79.7	35.6	46.4	17.7
NeMo+PyTorch3D	90.3	3.7	66.7	13.7	6.1	85.2	74.5	61.6	**71.7**
NeMo+VoGE(Ours)	**94.9**	**13.5**	**77.8**	**30.8**	**22.2**	**89.8**	**81.9**	**68.9**	68.4

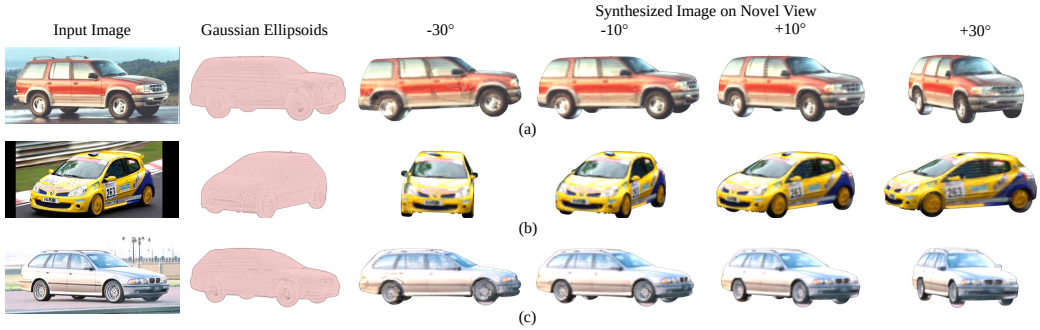


Figure 7: Sampling texture and rerendering on novel view. The inputs include a single RGB image and the Gaussian Ellipsoids with corresponded pose. Note the result is produced **without** any training or symmetrical information.

Baselines. We compare our VoGE for object pose estimation with other state-of-the-art differentiable renderers, *i.e.* Soft Rasterizer, DSS, and PyTorch3D. For comparison, we use the same training and inference pipeline, and same hyper-parameters for all 4 experiments. Our baselines also includes Res50-General/Specific which converts object pose estimation into a bin classification problem, and StarMap Zhou et al. (2018) which first detect keypoints and conduct pose estimation via the PnP method.

Experiment Details. Following the experiment setup in NeMo, we train the feature extractor 800 epochs with a progressive learning rate. During inference, for each image, we sample 144 starting poses and optimizer 300 steps via an ADAM optimizer. We convert the meshes provided by NeMo using the method described Section 3.2.

Results. Figure 6 and Table 2 show the qualitative and quantitative results of object pose estimation on PASCAL3D+ and the Occluded PASCAL3D+ dataset. Results in Table 2 demonstrate significant performance improvement using VoGE compared to Soft RasterizerLiu et al. (2019), DSSYifan et al. (2019) and PyTorch3DRavi et al. (2020). Moreover, both qualitative and quantitative results show our method a significant robustness under partial occlusion and out distributed cases. Also, Figure 6 demonstrates our approach can generalize to those out distributed cases, *e.g.*, a car without front bumper, while infeasible for baseline renderers. Table 3 shows the results on ObjectNet3D, which demonstrates a significant performance gain compared to the baseline approaches. The ablation study is included in Appendix C.1.

4.2 Texture Extraction and Rerendering

As Figure 7 shows, we conduct the texture extraction on real images and rerender the extracted textures under novel viewpoints. The qualitative results is produced on PASCAL3D+ dataset. The experiment is conducted on each image independently that there is no training included. Specifically, for each image, we have only three inputs, *i.e.* the image, the camera configuration, the Gaussian ellipsoids converted from the CAD models provided by the dataset. Using the method proposed in 3.4, we extract the RGB value for each kernel on the Gaussian ellipsoids using the given groundtruth camera configuration. Then we rerender Gaussian ellipsoids with the extracted texture under a novel view, that we increase or decrease the azimuth of the viewpoint (horizontal rotation). The qualitative

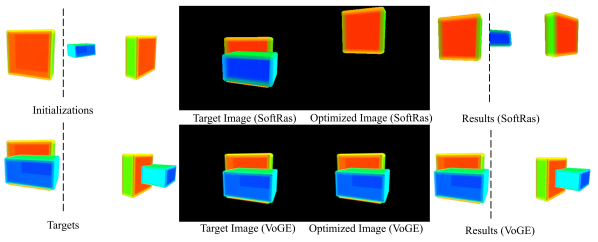


Figure 8: Reasoning multi-object occlusions for single view optimization of object location. Left: initialization and target locations. Middle: target images, note the target image generate via each rendering method. Right: results. Videos: VoGE, SoftRas.

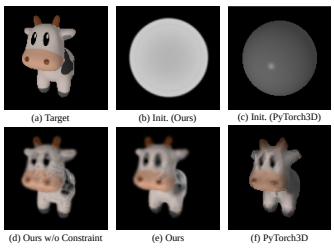


Figure 9: Shape fitting results with 20 multi-viewed images following the PyTorch3D Ravi et al. (2020) official tutorial.

results demonstrate a satisfying texture extraction ability of VoGE, even with only a single image. Also, the details (*e.g.*, numbers on the second car) are retained in high quality under the novel views.

4.3 Occlusion Reasoning of Multiple Objects

Figure 8 shows differentiating the occlusion reasoning process between two objects. Specifically, a target image, and the colored cuboid models and initialization locations, are given to the method. Then we render and optimize the 3D locations of both the cuboids. In this experiment, we find both SoftRas and VoGE can successfully optimize the locations when the occludee (blue cuboid) is near the occluder (red cuboid), which is 1.5 scales behind the occluder as the thickness of the occluder is 0.6 scales. However, when the the occludee is far behind the occluder (5 scales), SoftRas fails to produce correct gradient to optimize the locations, whereas VoGE can still successfully optimize the locations. We think such advantage benefits from the better volume density aggregation compared to the distance based aggregation used in SoftRas.

4.4 Shape Fitting via Inverse Rendering

Figure 9 shows the qualitative results of multi-viewed shape fitting. In this experiment, we follows the setup in *fit a mesh with texture via rendering* from PyTorch3D official tutorial pyt. First, a standard graphic renderer is used to render the cow CAD model in 20 different viewpoints under a fixed light condition, which are used as the optimization targets. For both baseline and ours, we give a sphere object geometry with 2562 vertices and optimize toward target images using the same configuration, *e.g.*, iterations, learning rate, optimizer, loss function. During the shape optimization process, we compute MSE loss on both silhouettes and RGB values between the synthesized images and the targets. The vertices locations and colors are gradiently updated with an ADAM optimizer Kingma & Ba (2014). We conduct the optimization for 2000 iterations, while in each iteration, we randomly select 5 out of 20 images to conduct the optimization. In Figure 9 (e) and (f), we use the normal consistency, edge and Laplacian loss Nealen et al. (2006) to constrain the object geometry, while in (d) no additional loss is used. From the results, we can see that VoGE has a competitive ability regarding shape fit via deformation. Specifically, VoGE gives better color prediction and a smoother object boundary. Also, we observe the current geometry constrain losses do not significantly contribute to our final prediction. We argue those losses are designed for surface triangular meshes, that not suitable for Gaussian ellipsoids. The design of geometry constraints that are suitable for Gaussian ellipsoids is an interesting topic but beyond scope of this paper.

5 Conclusion

In this work, we propose VoGE, a differentiable volume renderer using Gaussian Ellipsoids. Experiments on in-wild object pose estimation and neural view matching show VoGE an extraordinary ability when applied on neural features compare to the concurrent famous differential generic renderers. Texture extraction and rerendering experiment shows VoGE the ability on feature and texture sampling, which potentially benefits downstream tasks. Overall, VoGE demonstrates better differentiability, which benefits vision tasks, while retains competitive rendering speed.

REFERENCES

Pytorch3d official tutorials: Fit a mesh via rendering. https://pytorch3d.org/tutorials/fit_textured_mesh. Accessed: 2021-11-10.

Anand Bhattad, Aysegul Dundar, Guilin Liu, Andrew Tao, and Bryan Catanzaro. View generalization for single image textured 3d models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 6081–6090, 2021.

James F Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *Acm Siggraph Computer Graphics*, 16(3):21–29, 1982.

Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pp. 303–312, 1996.

Kyle Genova, Forrester Cole, Avneesh Sud, Aaron Sarna, and Thomas Funkhouser. Local deep implicit functions for 3d shape. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

Shun Iwase, Xingyu Liu, Rawal Khirodkar, Rio Yokota, and Kris M Kitani. Repose: Fast 6d object pose refinement via deep texture rendering. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021.

James T Kajiya and Brian P Von Herzen. Ray tracing volume densities. *ACM SIGGRAPH computer graphics*, 18(3):165–174, 1984.

Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3d mesh renderer. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Christoph Lassner and Michael Zollhofer. Pulsar: Efficient sphere-based neural rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*. Springer, 2014.

Guilin Liu, Duygu Ceylan, Ersin Yumer, Jimei Yang, and Jyh-Ming Lien. Material editing using a physically based rendering network. In *Proceedings of the IEEE International Conference on Computer Vision*, 2017.

Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019.

Matthew M Loper and Michael J Black. Opendr: An approximate differentiable renderer. In *European Conference on Computer Vision*. Springer, 2014.

Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *European conference on computer vision*. Springer, 2020.

Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. Laplacian mesh optimization. In *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, 2006.

Michael Niemeyer and Andreas Geiger. Giraffe: Representing scenes as compositional generative neural feature fields. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021.

Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3d representations without 3d supervision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 11.2.89, 2022. URL `https://developer.nvidia.com/cuda-toolkit`.

Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3d deep learning with pytorch3d. *arXiv:2007.08501*, 2020.

Katja Schwarz, Yiyi Liao, Michael Niemeyer, and Andreas Geiger. Graf: Generative radiance fields for 3d-aware image synthesis. *Advances in Neural Information Processing Systems*, 33, 2020.

Angtian Wang, Adam Kortylewski, and Alan Yuille. Nemo: Neural mesh models of contrastive features for robust 3d pose estimation. In *International Conference on Learning Representations*, 2020a.

Angtian Wang, Yihong Sun, Adam Kortylewski, and Alan L. Yuille. Robust object detection under occlusion with context-aware compositionalnets. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, June 2020b.

Lee Westover. Footprint evaluation for volume rendering. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pp. 367–376, 1990.

Yu Xiang, Roozbeh Mottaghi, and Silvio Savarese. Beyond pascal: A benchmark for 3d object detection in the wild. In *IEEE Winter Conference on Applications of Computer Vision*, 2014.

Yu Xiang, Wonhui Kim, Wei Chen, Jingwei Ji, Christopher Choy, Hao Su, Roozbeh Mottaghi, Leonidas Guibas, and Silvio Savarese. Objectnet3d: A large scale database for 3d object recognition. In *European Conference Computer Vision*, 2016.

Wang Yifan, Felice Serena, Shihao Wu, Cengiz Öztireli, and Olga Sorkine-Hornung. Differentiable surface splatting for point-based geometry processing. *ACM Transactions on Graphics*, 38(6): 1–14, 2019.

Jason Zhang, Gengshan Yang, Shubham Tulsiani, and Deva Ramanan. Ners: Neural reflectance surfaces for sparse-view 3d reconstruction in the wild. *Advances in Neural Information Processing Systems*, 34, 2021.

Xingyi Zhou, Arjun Karpur, Linjie Luo, and Qixing Huang. Starmap for category-agnostic keypoint and viewpoint estimation. In *Proceedings of the European Conference on Computer Vision*, 2018.

Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. Ewa volume splatting. In *Proceedings Visualization, 2001. VIS'01*. IEEE, 2001.

A ADDITIONAL DETAILS OF VOGE RENDERER

In this section we provide more detailed discussion for the math of ray tracing volume densities in VoGE (section A.1 and A.2), coarse-to-fine rendering strategy (section A.3), and the converters (section A.4).

A.1 RAY TRACING

In this section, we provide the detailed deduction process for Equations 6 in the main text. First, let's recall the formula of Ray tracing volume densities Kajiya & Von Herzen (1984):

$$\begin{aligned} C(\mathbf{r}) &= \int_{t_n}^{t_f} T(t)\rho(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t))dt, \\ \text{where } T(t) &= \exp\left(-\tau \int_{t_n}^{t} \rho(\mathbf{r}(s))ds\right) \end{aligned} \tag{11}$$

where $T(t)$ is the occupancy function alone viewing ray $\mathbf{r}(t)$, as we describe in Equation 5 in main text:

$$\mathbf{r}(t) = \mathbf{D} * t \tag{12}$$

where $\mathbf{D}$ is the normalized direction vector of the viewing ray.

Also, as we describe in Section 3.2, we reconstruct the volume density function $\rho(\mathbf{r}(t))$ via the sum of a set of ellipsoidal Gaussians:

$$\rho(\mathbf{X}) = \sum_{k=1}^{K} \frac{1}{\sqrt{2\pi \cdot ||\mathbf{\Sigma}_k||_2}} e^{-\frac{1}{2}(\mathbf{X}-\mathbf{M}_k)^T \cdot \mathbf{\Sigma}_k^{-1} \cdot (\mathbf{X}-\mathbf{M}_k)} \tag{13}$$

where K is the total number of Gaussian kernels, $\mathbf{X} = (x, y, z)$ is an arbitrary location in the 3D volume. $\mathbf{M}_k$ is the center of k-th ellipsoidal Gaussians kernel:

$$\mathbf{M}_k = (\mu_{k,x}, \mu_{k,y}, \mu_{k,z}) \tag{14}$$

whereas the $\mathbf{\Sigma}_k$ is the spatial variance matrix:

$$\mathbf{\Sigma}_k = \begin{bmatrix} \sigma_{k,xx} & \sigma_{k,xy} & \sigma_{k,xz} \\ \sigma_{k,yx} & \sigma_{k,yy} & \sigma_{k,yz} \\ \sigma_{k,zx} & \sigma_{k,zy} & \sigma_{k,zz} \end{bmatrix} \tag{15}$$

Note that $\mathbf{\Sigma}_k$ is a symmetry matrix, e.g., covariance $\sigma_{k,xy} = \sigma_{k,yx}$.

Occupancy Function. Based on Equation 13 and 11, $T(t)$ can be computed via:

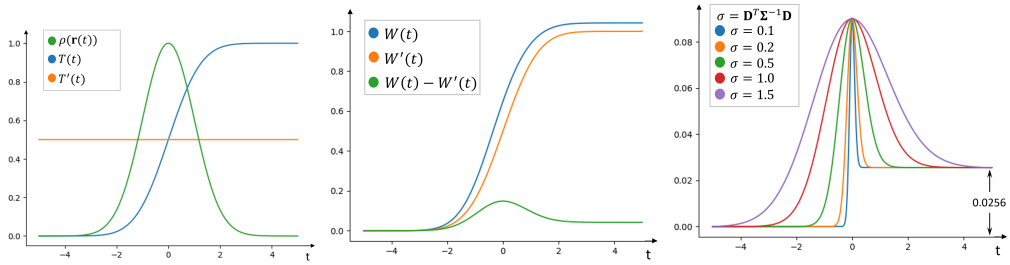
$$\begin{aligned} T(t) &= \exp\left(-\tau \int_{t_n}^{t} \rho(\mathbf{r}(s))ds\right) \\ &= \exp(-\tau \int_{t_n}^{t} \sum_{k=1}^{K} \frac{1}{\sqrt{2\pi \cdot ||\mathbf{\Sigma}_k||_2}} e^{-\frac{1}{2}(s\mathbf{D}-\mathbf{M}_k)^T \cdot \mathbf{\Sigma}_k^{-1} \cdot (s\mathbf{D}-\mathbf{M}_k)} ds) \end{aligned} \tag{16}$$

Now, let $\mathbf{M}_k = l_k\mathbf{D} + \mathbf{V}_k$, where l_k is a length along the viewing ray, $\mathbf{V}_k = \mathbf{M}_k - l_k\mathbf{D}$ is the vector from location $l_k\mathbf{D}$ on the ray to the vertex $\mathbf{M}_k$ (we will discuss a solution for $\mathbf{V}_k$ and l_k later). Equation 16 can be simplified as:

$$\begin{aligned} T(t) = \exp(-\tau \int_{t_n}^{t} \sum_{k=1}^{K} \frac{1}{\sqrt{2\pi \cdot ||\mathbf{\Sigma}_k||_2}} e^{-\frac{1}{2}(s-l_k)^2 \mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D}} \\ e^{-\frac{1}{2}(s-l_k)(\mathbf{V}_k^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D} + \mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{V}_k)} e^{-\frac{1}{2}\mathbf{V}_k^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{V}_k} ds) \end{aligned} \tag{17}$$

In order to further simplify $T(t)$, we take $\mathbf{V}_k$ that makes:

$$\mathbf{V}_k^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D} + \mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{V}_k = 0 \tag{18}$$



(a) (b) (c)

Figure 10: Approximate computation of integral along the viewing ray for a single kernel. (a) $T(t)$ is the real occupancy function along the ray, $T^{'}(t)$ means we use the occupancy at l_k of $\rho(\mathbf{r}(t))$, since $\rho(\mathbf{r}(t))$ are mainly concentrate near l_k. (b) shows $W(t) = \int_{-\infty}^{t} T(s) \cdot \rho(\mathbf{r}(s))ds$, $W^{'}(t) = \int_{-\infty}^{t} T^{'}(s) \cdot \rho(\mathbf{r}(s))ds$ and the difference $W(t) - W^{'}(t)$. Since we use the infinite integral with t, only error at end of t axis need to be consider. (c) shows the accumulative $W(t) - W^{'}(t)$ using different $\sigma = \mathbf{D}^T \cdot \mathbf{\Sigma}^{-1} \cdot \mathbf{D}$. Interestingly, the final error gives a fix value which is independent from σ. Note that the final error is 0.0256 which can be ignored when compared to the integral result $W = 1$.

which can be solve using $\mathbf{V}_k = \mathbf{M}_k - l_k\mathbf{D}$:

$$\begin{aligned} (\mathbf{M}_k - l_k\mathbf{D})^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D} + \mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot (\mathbf{M}_k - l_k\mathbf{D}) = 0 \\ \mathbf{M}_k^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D} + \mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{M}_k - 2l_k\mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D} = 0 \\ l_k = \frac{\mathbf{M}_k^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D} + \mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{M}_k}{2 \cdot \mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D}} \end{aligned} \tag{19}$$

Note that l_k is also the length that gives the maximum density $\rho_k(\mathbf{r}(t))$ along the ray for k-th kernel. To proof this, we compute:

$$\begin{aligned} \frac{\partial}{\partial t}\rho_k(\mathbf{r}(t)) &= \frac{\partial}{\partial t} \frac{1}{\sqrt{2\pi \cdot ||\mathbf{\Sigma}_k||_2}} e^{-\frac{1}{2}(t\mathbf{D}-\mathbf{M}_k)^T \cdot \mathbf{\Sigma}_k^{-1} \cdot (t\mathbf{D}-\mathbf{M}_k)} \\ &= \frac{1}{4\sqrt{2\pi||\mathbf{\Sigma}_k||_2}} (t\mathbf{D} - \mathbf{M}_k)^T \mathbf{\Sigma}_k^{-1} (t\mathbf{D} - \mathbf{M}_k) \\ &\quad \cdot (\mathbf{M}_k^T \mathbf{\Sigma}_k^{-1} \mathbf{D} + \mathbf{D}^T \mathbf{\Sigma}_k^{-1} \mathbf{M}_k - 2t\mathbf{D}^T \mathbf{\Sigma}_k^{-1} \mathbf{D}) \\ &\quad \cdot e^{-\frac{1}{2}(t\mathbf{D}-\mathbf{M}_k)^T \cdot \mathbf{\Sigma}_k^{-1} \cdot (t\mathbf{D}-\mathbf{M}_k)} \end{aligned} \tag{20}$$

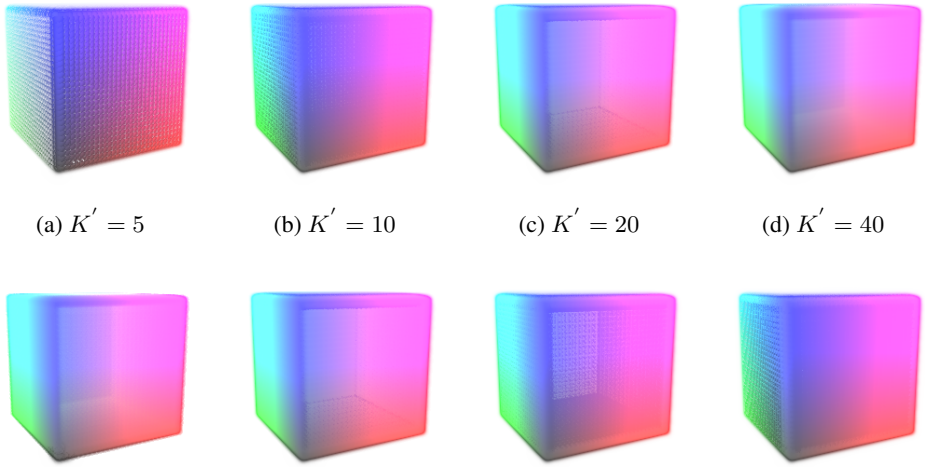
Obviously, the solve for $\frac{\partial}{\partial t}\rho_k(\mathbf{r}(t)) = 0$ is:

$$t = \frac{\mathbf{M}_k^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D} + \mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{M}_k}{2 \cdot \mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D}} = l_k \tag{21}$$

Now the density function of the k-th ellipsoid along the viewing ray $\mathbf{r}(s)$ gives an 1D Gaussian function:

$$\begin{aligned} \rho_k(\mathbf{r}(s)) &= \frac{e^{-\frac{1}{2}\mathbf{V}_k^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{V}_k}}{\sqrt{2\pi \cdot ||\mathbf{\Sigma}_k||_2}} e^{-\frac{1}{2}(s-l_k)^2 \mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D}} \\ &= \frac{1}{\sqrt{2\pi \cdot ||\mathbf{\Sigma}_k||_2}} \cdot \exp(q_k - \frac{(s-l_k)^2}{2 \cdot \sigma_k^2}) \end{aligned} \tag{22}$$

where $q_k = -\frac{1}{2}\mathbf{V}_k^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{V}_k$, $\frac{1}{\sigma_k^2} = \mathbf{D}^T \cdot \mathbf{\Sigma}_k^{-1} \cdot \mathbf{D}$. Thus, when tracing along each ray, we only need to record l_k, q_k and σ_k for each ellipsoid respectively.



(a) $K^{'} = 5$ (b) $K^{'} = 10$ (c) $K^{'} = 20$ (d) $K^{'} = 40$

(e) $\eta = 0.1$ (f) $\eta = 0.01$ (g) $\eta = 0.001$ (h) $\eta = 0.0001$

Figure 11: Rendering cuboid using different $K^{'}$ and η. (a) to (d) shows the rendering result using different $K^{'}$, the threshold is fix as $\eta = 0.01$. (e) to (h) shows the rendered cuboid with different η, while fixing $K^{'} = 20$.

A.2 Volume Density Aggregation

Since q_k is independent from t, the Equation 16 can be further simplified:

$$\begin{aligned} T(t) &= \exp(-\sum_{k=1}^{K} e^{q_k} \int_{-\infty}^{t} \frac{1}{\sqrt{2\pi \cdot ||\mathbf{\Sigma}_k||_2}} e^{-(s-l_k)^2/\sigma_k^2} ds) \\ &= \exp(-\sum_{k=1}^{K} e^{q_k} \frac{\mathrm{erf}((t-l_k)/\sigma_k)+1}{2}) \end{aligned} \tag{23}$$

where erf is the error function, that concurrent computation platforms, *e.g.*, PyTorch, Scipy, have already implemented.

Scattering Equation. Now we compute the final color observation $C(\mathbf{r})$. As we describe in Section 3.2, we assume each kernel has a homogeneous $\mathbf{C}_k$. Thus, here we compute:

$$\begin{aligned} \mathbf{W}_k(t) &= \int_{t_n}^{t_f} T(t)\rho(\mathbf{r}(t))dt \\ &= \int_{-\infty}^{\infty} T(t) \sum_{k=1}^{K} \frac{1}{\sqrt{2\pi \cdot ||\mathbf{\Sigma}_k||_2}} e^{-\frac{1}{2}(\mathbf{X}-\mathbf{M}_k)^T \cdot \mathbf{\Sigma}_k^{-1} \cdot (\mathbf{X}-\mathbf{M}_k)} dt \end{aligned} \tag{24}$$

where $\mathbf{X} = t\mathbf{D}$. Similar to previous simplifications, we use q_k and l_k to replace $\mathbf{M}_k$ in Equation 24:

$$\mathbf{W}_k(t) = \sum_{k=1}^{K} e^{q_k} \int_{-\infty}^{\infty} T(t) \frac{1}{\sqrt{2\pi \cdot ||\mathbf{\Sigma}_k||_2}} e^{-(t-l_k)^2/\sigma_k^2} dt \tag{25}$$

Due to the error function is already a complex function, it is infeasible to compute the integral of $T(t)$. We propose an approximate solution that we use $T(l_k)$ to replace $T(t)$ inside the integral. Now the final closed-form solution for $\mathbf{W}_k(t)$ is computed by:

$$\begin{aligned} \mathbf{W}_k(t) &= \sum_{k=1}^{K} T(l_k) e^{q_k} \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi \cdot ||\mathbf{\Sigma}_k||_2}} e^{-(t-l_k)^2/\sigma_k^2} dt \\ &= \sum_{k=1}^{K} T(l_k) e^{q_k} \end{aligned} \tag{26}$$

Because of the complexity when computing integral of the erf function, here we prove that in practice such approximate gives high enough accuracy. To simplify the problem, we study the case that the volume only contains a single Gaussian ellipsoid kernel. We further suggest that in the multi-kernel cases, the errors between different kernels introduced by the approximation will be lower. Because m-th kernel has a low $\rho_m(\mathbf{r}(t))$ at l_k, which makes the corresponded $T(t)$ more flatten, thus the approximation fits better. As Figure 10 (a) shows, we plot density function along the ray. Specifically, we sample 10k points on the ray, and for each point, we plot its density, the real occupancy, and the approximate occupancy. Figure 10 (b) shows the real weight W which is computed via the cumulative sum along the ray, and the approximate weight $W^{'}$ which is computed via our proposed approximate closed-from solution. We also show the difference between $W^{'}$ and W with the green line, which is significantly smaller compare to W. Interestingly, as Figure 10 (c) shows, we find the error $W(t) - W^{'}(t)$ is independent from $\mathbf{\Sigma}^{-1}$ and $\mathbf{D}$, that always converge to a same value: 0.0256. Though we cannot give a mathematical explanation regarding this phenomenon, we argue the result is already enough to draw the conclusion that such approximation gives satisfying accuracy.

A.3 Coarse-to-Fine Rendering with Kernel Selection

As we discussed in Section 3.3 in the main text, in order to efficiently render Gaussian ellipsoids, we design the coarse-to-fine rendering strategy. Specifically, we gradually reduce the number of ellipsoids that interact with viewing rays. Following PyTorch3D, we develop a optional coarse rasterization stage, which select 10% of all ellipsoids and feed them into the ray tracing stage. Specifically, we project the center of each ellipsoid onto the screen coordinate via standard object-to-camera transformation, then for each ellipsoids, we compute the height b_h and width b_w of a maximum bounding box of the ellipsoids in 2D screen coordinate. The height and width are computed via:

$$[b_h \quad b_w \quad .] = \frac{\log(-\eta)}{d_z} \cdot \mathbf{\Omega} \cdot \mathbf{\Sigma}^{-1} \cdot \mathbf{\Omega} \tag{27}$$

where d_z is the distance from camera to the center of ellipsoid, η is the threshold for maximum volume density, $\mathbf{\Omega}$ is the projection matrix from camera coordinate to screen:

$$\mathbf{\Omega} = \begin{bmatrix} \frac{2 \cdot F}{h} & 0 & 0 \\ 0 & \frac{2 \cdot F}{w} & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{28}$$

Then we rasterize the bounding boxes to produce a pixel-to-kernels assignment in a low resolution (8 times smaller compared to the image size), which indicates the set of ellipsoid kernels for each pixel to trace.

Similarly, the ray tracing stage is also select only part of all Gaussian ellipsoids to feed into the aggregation stage. When conducting ray tracing, we only trace $K^{'}$ nearest kernels that has non-trivial contributions regarding its final weight $\mathbf{W}_k$. Specifically, we first record all ellipsoids that gives a maximum density $e^{q_k} > \eta$. For all the recorded kernels, we sort them via the length to the 1D Gaussian center l_k and select $K^{'}$ nearest ellipsoids. In the experiment, we find $K^{'}$ has a significant impact on the quality of rendered images, while the threshold η has relatively low impact, but needs to be fit with $K^{'}$. Here we provide default settings that give a satisfying quality with low computation cost: $K^{'} = 20, \eta = 0.01$.

Figure 11 shows the rendered cuboids using different $K^{'}$ and η. Here the results demonstrate that inadequate $K^{'}$ will lead to some dark region around the boundary of kernels, which we think is caused by the hard cutoff of the boundary. On the other hand, decreasing the threshold η could make the object denser (less transparent), but need more kernels (higher $K^{'}$) to avoid the artifacts.

A.4 Mesh & Point Cloud Converter

We develop a simple mesh converter, which converts triangular meshes into isotropic Gaussian ellipsoids, and a point cloud converter. In the mesh converter, we retain all original vertices on the mesh and compute the $\mathbf{\Sigma}_k$ using the distance between each vertex and its connected neighbors.

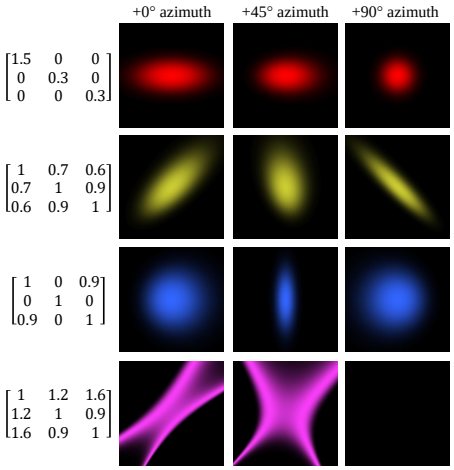


Figure 12: Rendering single anisotropic ellipsoidal Gaussian. Left column shows the $\mathbf{\Sigma}$ for the kernel. We render the kernel under 3 different viewpoint: 0° azimuth, 45° azimuth, and 90° azimuth.

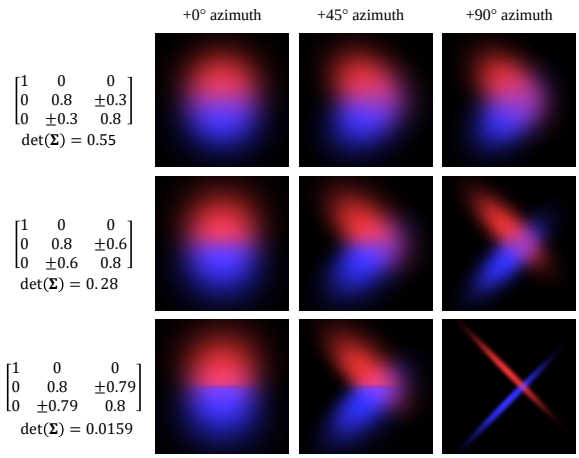


Figure 13: Rendering flattened Gaussian ellipsoids to approximate the 2D Gaussian ellipses. For each row, we show two Gaussian ellipsoids viewed in 3 different viewpoints. From top to bottom: decrease $\det(\mathbf{\Sigma})$ to flatten Gaussian ellipsoids.

Specifically, for each vertex, we compute the average length d_k of edges connected to that vertex. Then $\mathbf{\Sigma}_k$ is computed via:

$$\mathbf{\Sigma}_k = \begin{bmatrix} \sigma_k & 0 & 0 \\ 0 & \sigma_k & 0 \\ 0 & 0 & \sigma_k \end{bmatrix} \tag{29}$$

where σ_k is computed via the coverage rate ζ and d_k,

$$\sigma_k = \frac{(d_k/2)^2}{\log(1/\zeta)} \tag{30}$$

Similarly, in the point cloud converter, the $\mathbf{\Sigma}_k$ is controlled with the same function, but the d_k is determined by the distance to m nearest points of the target points.

Since the concurrent mesh converter does not consider the shape of the triangles, admittedly we think this could be improved via converting each triangle into an anisotropic Gaussian ellipsoid, which we are still working on.

B Additional Rendering Results

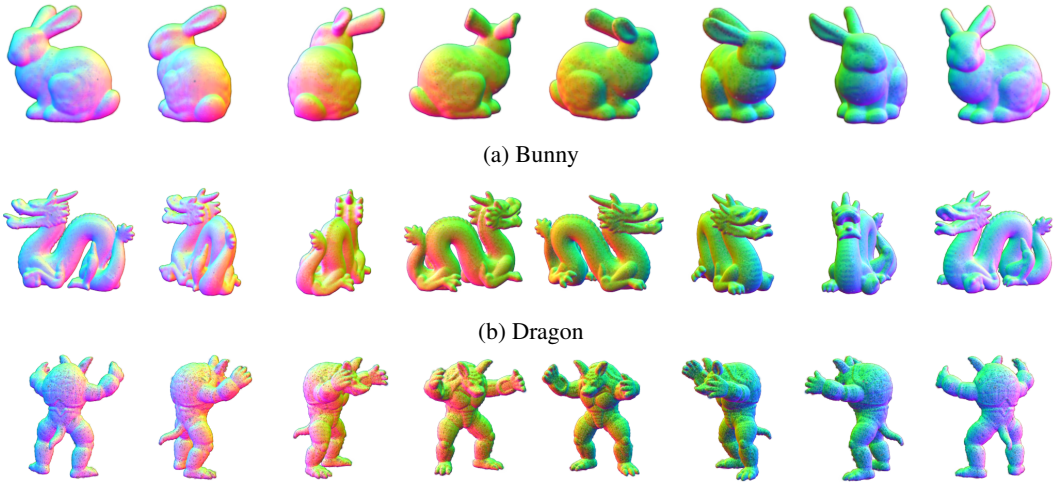
B.1 Rendering Anisotropic Gaussian Ellipsoids

As Figure 12 shows, VoGE rendering pipeline natively supports anisotropic ellipsoidal Gaussian kernels, where for each kernel the spatial variance is represented via the 3×3 symmetric matrix $\mathbf{\Sigma}_k$. Note that, the spatial covariances, *e.g.*, $\sigma_{k,xy}$, cannot exceed square root of dot product of the two variances, *e.g.*, $\sqrt{\sigma_{k,xx}\sigma_{k,yy}}$, otherwise, the kernel will become hyperbola instead of ellipsoids (as the last row in Figure 12 shows).

On the other hand, we suggest that ellipsoidal Gaussian kernels can also **approximate the 2D Gaussian ellipses** (the representation used in DSS Yifan et al. (2019)), which can be simply done by set $\det(\mathbf{\Sigma}) \to 0$, where $\det$ is the determinant of matrix. Figure 13 shows the rendering result using flattened Gaussian ellipsoids. As we demonstrated in the third row, VoGE rendering pipeline allows rendering the surface-liked representations in a stable manner.

B.2 Rendering Surface Normal

As Figure 14 shows, we render CAD models provided by *The Stanford 3D Scanning Repository* Curless & Levoy (1996). Specifically, we use our mesh converter to convert the meshes provide by



(a) Bunny

(b) Dragon

(c) Armadillo

Figure 14: Rendering the surface normal using VoGE under 8 different viewpoints. The Gaussian ellipsoids are converted from meshes provided by *The Stanford 3D Scanning Repository*.

the dataset into Gaussian ellipsoids. In detail, the Bunny contains 8171 vertices, the Dragon contains 22998 vertices, and the Armadillo contains 33792 vertices. During rendering, we compute surface normals via PyTorch3D and use the normals as the RGB value of each vertex. Then we render the Gaussian Ellipsoids in 8 different viewpoints, and interpolate the RGB value into images.

B.3 Rendering Quality vs Number of Gaussians

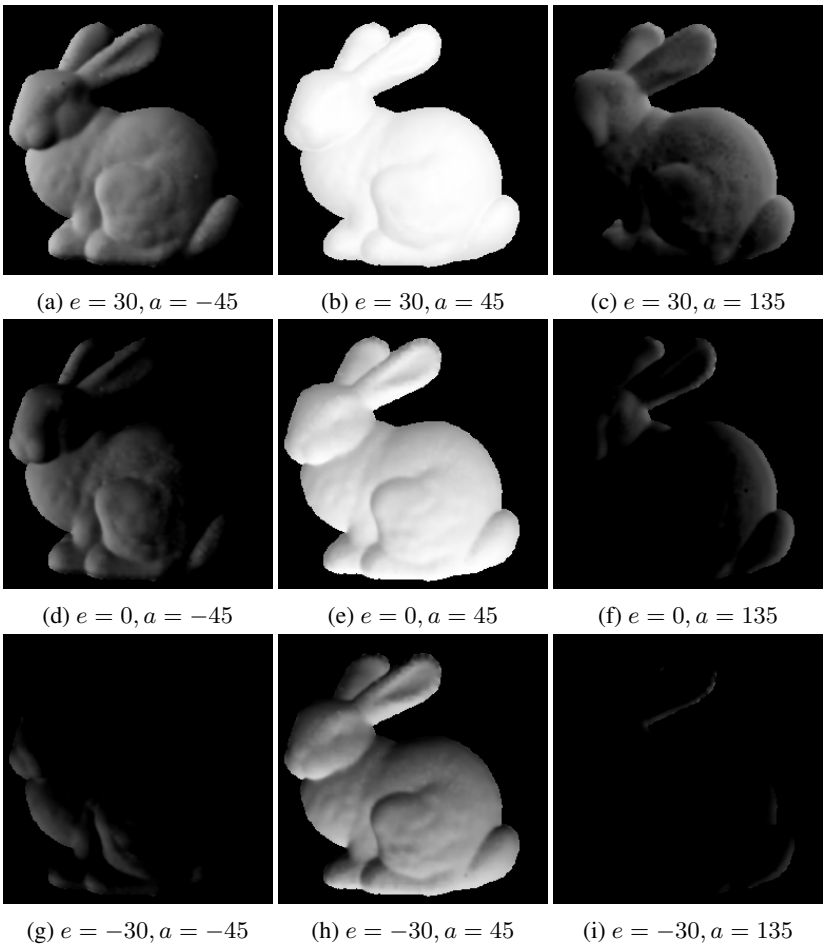
Figure 16 shows surface normal rendering quality of VoGE using different number of Gaussians. We also include comparison of rendering quality of VoGE vs PyTorch3D mesh renderer. In each image, we control a same number of Gaussians vs mesh vertices, which gives similar number of parameters that $9 * N_{Gauss}$ vs $3 * N_{verts} + 3 * N_{faces}$. Here we observe that increasing number of Gaussians will significant improve rendering quality. Admittedly, VoGE renderer gives slight fuzzier boundary compare to mesh renderer.

B.4 Lighting with External Normals

Although Gaussian ellipsoids do not contain surface normal information (since they are represented as volume), VoGE still can utilize surface normal via processing them as an extra attribute in an external channel as we describe in section B.2. Once the surface normals are rendered, the light diffusion method in the traditional shader can be used to integrate lighting information into VoGE rendering pipeline. Figure 15 shows the results that integrate lighting information when rendering the Stanford bunny mesh using VoGE. Specifically, we first render the surface normals computed via PyTorch3D into an image-liked map (same as the process in section B.2). Then we use the diffuse function (*PyTorch3D.renderer.lighting*), to compute the brightness of the rendered bunny under a point light. In the visualization, we place the light source at variant locations, while using a fully white texture on the bunny.

B.5 Rendering Point Clouds

Figure 17 shows the point clouds rendering results using VoGE and PyTorch3D. We follow the *Render a colored point cloud* from PyTorch3D official tutorial **?**. Specifically, we use the Pittsburgh-Bridge point cloud provided by PyTorch3D, which contains 438544 points with RGB color for each point respectively. We first convert the point cloud into Gaussian ellipsoids using the method described in A.4. Then we render the Gaussian ellipsoids using the same configuration (Except the camera. As the tutorial uses orthogonal camera, which concurrently we don't support, we alter-



(a) $e = 30, a = -45$ (b) $e = 30, a = 45$ (c) $e = 30, a = 135$

(d) $e = 0, a = -45$ (e) $e = 0, a = 45$ (f) $e = 0, a = 135$

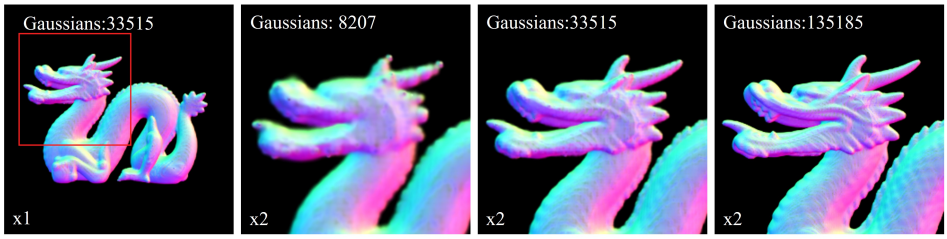
(g) $e = -30, a = -45$ (h) $e = -30, a = 45$ (i) $e = -30, a = 135$

Figure 15: Lighting rendered mesh using external normals. We first render the surface normals of the bunny mesh using VoGE. Then we use the light diffusion functions provided by PyTorch3D to light the render surface normal. For each image, we place a point light source in the object space using different elevations (e) and azimuth (a). The distance from the light source to the object center is fixed as 1.

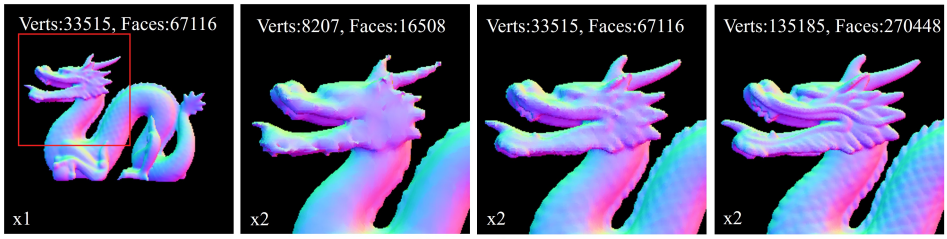
Table 4: (Left) Ablation study for object pose estimation on PASCAL3D+. We control the coverage rate ζ when computing $\mathbf{\Sigma}$, higher ζ gives larger values in $\mathbf{\Sigma}$. w/o grad $T(\mathbf{r})$ means we block the gradient from $T(\mathbf{r})$, while w/o grad $\rho(\mathbf{r})$ means gradient on e^{q_k} in Equation 8 is blocked.

Exp. Setup	$ACC_{\frac{\pi}{6}}$	$ACC_{\frac{\pi}{18}}$	*MedErr*
$\zeta = 0.2$	89.9	68.7	7.0
$\zeta = 0.5$ (standard)	90.1	69.2	6.9
$\zeta = 0.8$	90.3	64.7	8.5
w/o grad $T(\mathbf{r})$	47.1	18.9	39.1
w/o grad $\rho(\mathbf{r})$	31.7	8.0	48.1

nate the camera using a PerspectiveCamera with a similar viewing scope). The qualitative results demonstrate VoGE a better quality with smoother boundaries.



(a) VoGE



(b) PyTorch3D

Figure 16: Comparison of rendering quality with number of components using VoGE vs PyTorch3D hard mesh renderer.

(a) PyTorch3D 1x

(b) PyTorch3D 4x

(c) PyTorch3D 16x

(d) VoGE 1x

(e) VoGE 4x

(f) VoGE 16x

Figure 17: Rendering point clouds using VoGE and PyTorch3D. In the 1x visualization, we render the image with camera with standard focal length. In 4x and 16x we zoom-in the camera by increase the focal length by 4x and 16x.

Table 5: Per category result for in-wild object pose estimation results on PASCAL3D+. Results are reported in Accuracy (percentage, higher better) and Median Error (degree, lower better).

		aero	bike	boat	bottle	bus	car	chair	table	mbike	sofa	train	tv	Mean
$\uparrow ACC_{\frac{\pi}{6}}$	Res50-General	83.0	79.6	73.1	87.9	96.8	95.5	91.1	82.0	80.7	97.0	94.9	83.3	88.1
	Res50-Specific	79.5	75.8	73.5	90.3	93.5	95.6	89.1	82.4	79.7	96.3	**96.0**	84.6	87.6
	StarMap	85.5	**84.4**	65.0	**93.0**	98.0	97.8	**94.4**	82.7	85.3	**97.5**	93.8	**89.4**	89.4
	NeMo+SoftRas	80.8	79.2	70.3	88.0	89.1	98.4	85.6	74.9	82.0	95.7	76.2	82.3	85.3
	NeMo+DSS	77.2	69.3	65.4	83.7	91.4	96.5	80.9	67.8	71.0	89.9	76.3	77.5	81.1
	NeMo+PyTorch3D	82.2	78.4	68.1	88.0	91.7	98.2	87.0	76.9	85.0	95.0	83.0	82.2	86.1
	NeMo+VoGE(ours)	**89.7**	82.6	**77.7**	88.2	**98.1**	**99**	90.5	**84.8**	**87.5**	94.9	89.2	83.9	**90.1**
$\uparrow ACC_{\frac{\pi}{18}}$	Res50-General	31.3	25.7	23.9	35.9	67.2	63.5	37.0	40.2	18.9	62.5	51.2	24.9	44.6
	Res50-Specific	29.1	22.9	25.3	39.0	62.7	62.9	37.5	42.0	19.5	57.5	50.2	25.4	43.9
	StarMap	49.8	34.2	25.4	**56.8**	90.3	81.9	**67.1**	57.5	27.7	**70.3**	69.7	40.0	59.5
	NeMo+SoftRas	47.5	26.2	36.2	49.9	85.5	94.5	46.7	50.7	29.8	59.5	63.9	42.6	59.7
	NeMo+DSS	22.8	10.2	23.7	37.8	52.8	38.9	23.1	15.9	12.1	31.7	18.7	25.7	27.8
	NeMo+PyTorch3D	49.7	29.5	37.7	49.3	89.3	94.7	49.5	52.9	29.0	58.5	70.1	42.4	61.0
	NeMo+VoGE(ours)	**61.4**	**40.3**	**51.2**	53.9	**93.8**	**96.7**	58.6	**70.8**	**39.6**	63.8	**79.3**	**47.9**	**69.2**
$\downarrow MedErr$	Res50-General	13.3	15.9	15.6	12.1	8.9	8.8	11.5	11.4	16.6	8.7	9.9	15.8	11.7
	Res50-Specific	14.2	17.3	15.4	11.7	9.0	8.8	12.0	11.0	17.1	9.2	10.0	14.9	11.8
	StarMap	10.0	14.0	19.7	**8.8**	3.2	4.2	**6.9**	8.5	14.5	**6.8**	6.7	12.1	9.0
	NeMo+SoftRas	10.6	17.3	15.1	10.0	3.3	3.4	10.4	9.9	14.9	8.4	6.1	12.3	9.1
	NeMo+DSS	16.6	23.0	19.5	13.1	9.3	11.7	16.2	18.7	21.9	14.2	20.5	18.2	16.1
	NeMo+PyTorch3D	10.1	16.3	14.9	10.2	3.2	3.2	10.1	9.3	14.1	8.6	5.4	12.2	8.8
	NeMo+VoGE(ours)	**7.5**	**12.8**	**9.8**	9.1	**2.6**	**2.9**	8.6	**5.8**	**12.5**	7.7	**4.3**	**10.5**	**6.9**

Figure 18: Additional qualitative in-wild object pose estimation results for NeMo+VoGE and NeMo+PyTorch3D.

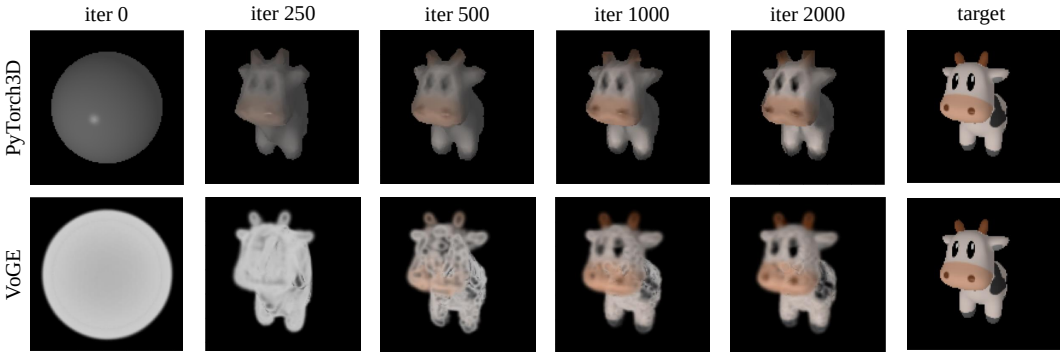


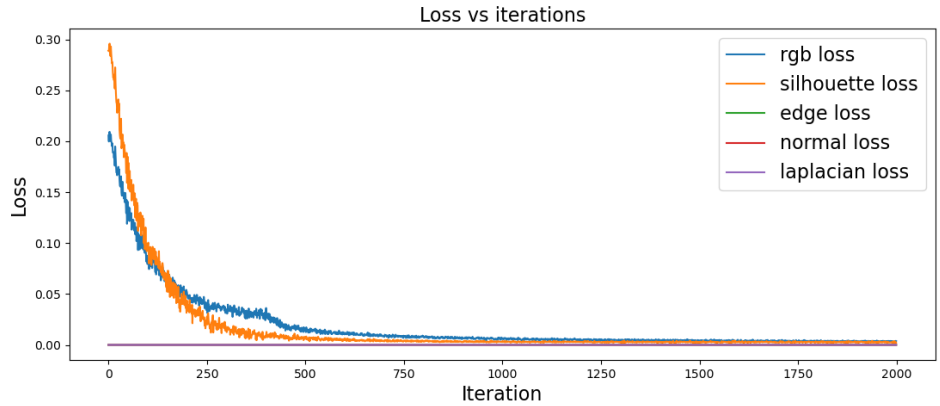
Figure 19: The shape fitting process regarding optimization iterations. We visualize both VoGE and PyTorch3D with all constraints.

C Additional Experiment Results

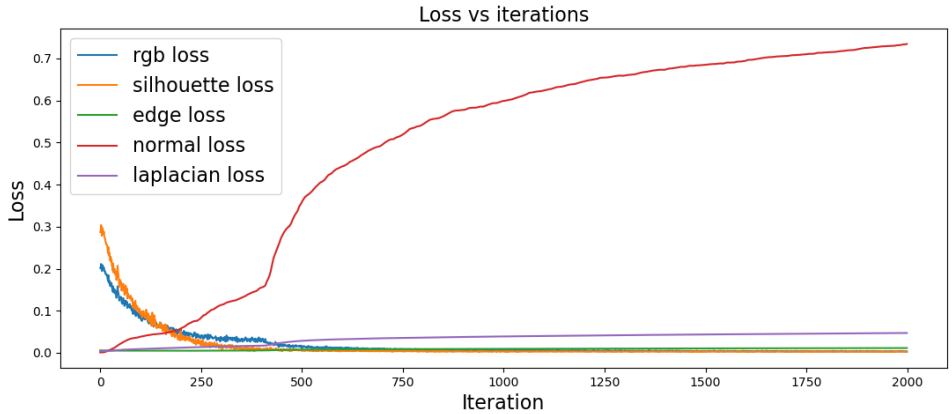
C.1 In-wild Object Pose Estimation

Ablation Study. As Table 4 shows, we conduct controlled experiments to validate the effects of different components. Using the method we described in 3.2, we develop tools that convert triangle meshes to Gaussian ellipsoids, where a tunable parameter, coverage rate, is used to control the intersection rate between nearby Gaussian ellipsoids. Specifically, the higher coverage rate gives the large $\mathbf{\Sigma}$, which makes the feature more smooth but also fuzzy, vice versa. As the results demonstrate, increasing $\mathbf{\Sigma}$ can increase the rough performance under $\frac{\pi}{6}$, while reducing it can improve the performance under the more accurate evaluation threshold. We also ablate the affect regarding block part of the gradient in Equation 8. Specifically, we conduct two experiments on all kernels, we block the gradient on $T(l_k)$ and e^{q_k} respectively. The results show blocking either term leads significant negative impact on the final performance.

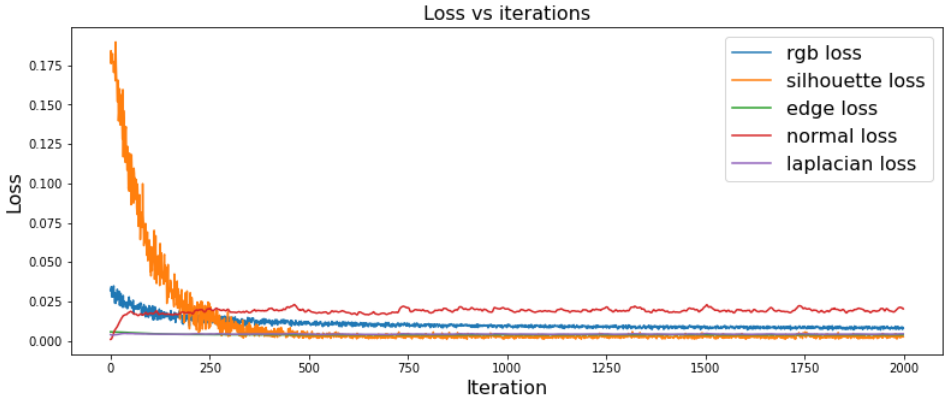
Additional Results. Table 5 shows the per-category object pose estimation results on PASCAL3D+ dataset (L0). All NeMo Wang et al. (2020a) baseline results and ours are conducted using the single cuboid setting described in NeMo. Specifically, Gaussian ellipsoids used in VoGE is converted from the same single cuboid mesh models provided by NeMo (coverage rate $\zeta = 0.5$).



(a) VoGE with all constraints.



(b) VoGE without constraints.



(c) PyTorch3D with all constraint.

Figure 20: Losses in the shape fitting experiment. Note that for the VoGE without constraint, we only calculate the geometry loss but not compute gradient with those losses.

Figure 18 shows the additional qualitative results of the object pose estimation. In the visualization, we use a standard graphic renderer to render the original CAD models provide by PASCAL3D+ dataset under the predicted pose, and superimpose the rendered object onto the input image.

C.2 TEXTURE EXTRACTION AND RERENDERING

Figure 21 shows the additional texture extraction and rerendering results on car, bus and boat images from PASCAL3D+ dataset. Interestingly, Figure 21 (g) shows the texture extraction using VoGE demonstrate stratifying generation ability on those out distributed cases.

C.3 SHAPE FITTING VIA INVERSE RENDERING

Figure 20 shows the losses in the multi-viewed shape fitting experiment. Specifically, we plot the losses regarding optimization iterations using the method provided by *fit a mesh with texture via rendering* from PyTorch3D official tutorial pyt. Note the geometry constraint losses except normal remain relatively low in VoGE without constraints experiment. We think such results demonstrate the optimization process using VoGE can give correct gradient toward the optimal solution effectively, that even without geometry constraint the tightness of the Gaussian ellipsoids is still retained. As for the normal consistency loss, since we use the volume Gaussian ellipsoids, the surface normal directions are no longer informative.

Figure 21: Additional results for texture extraction experiment on car, bus and boat category in PASCAL3D+ dataset. We extract texture using a in-wild image and Gaussian ellipsoids with corresponded viewpoint, and render under novel viewpoint.