EnvGS: Modeling View-Dependent Appearance with Environment Gaussian

Supplementary Material

In the supplementary material, we provide more qualitative and quantitative results and per-scene breakdowns to demonstrate the effectiveness and robustness of our method (Sec. A). We also provide additional ablation studies to further analyze the key components of our method (Sec. B). Furthermore, we provide details on the gradient computation of our Gaussian tracer (Sec. D).

Accurate and smooth reflection reconstruction and rendering are key advantages of our method. We strongly encourage readers to view the rendered continuous videos in the supplementary material for a more comprehensive understanding of its performance.

A. Additional Results

A.1. Comparison on Reflective Regions

To demonstrate the improvements in the reflective and near-field reflection regions using our environment Gaussian representation, we additionally annotate a reflection mask to compute metrics specifically for the reflective region and a near-field mask to evaluate near-field reflections on the Ref-Real [37] and NeRF-Casting [38] datasets. As shown in Tab. 2 and Fig. 6, our method achieves a significant improvement of over 1.0 PSNR ↑ improvement on the reflective region and 2.0 PSNR ↑ in the near-field regions compared to using an environment map. These results highlight the effectiveness of our approach in capturing and rendering complex reflective and near-field phenomena.

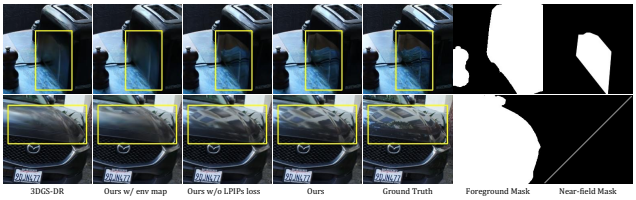

Figure 6. **Qualitative comparison on reflective foreground and near-field reflection regions.** We also provide visualizations of the foreground and near-field region mask we annotated.

The reflective masks mentioned above are obtained through the following steps. First, we train our EnvGS on each scene, then export the trained Gaussian and remove the Gaussian points in 3D space except for those in the foreground reflective region. We render the remaining Gaussian to generate an accumulated alpha map. Finally, we binarize this alpha map to obtain the foreground reflective masks. We manually annotate the near-field masks as they are difficult to define in 3D space.

The quantitative results in Tab. 2 are evaluated only in the masked regions, following NeRF-Casting [38], we compute these masked metrics by blending the masked regions onto a white background.

A.2. Comparison on Real-World Shiny Scenes

We present additional qualitative comparisons on the NeRF-Casting Shiny Scenes [38], including both indoor and outdoor real-world scenes featuring complex reflections. As shown in Fig. 9, our method significantly outperforms previous approaches in reflection fidelity and overall rendering quality, particularly excelling in near-field reflections and high-frequency reflection details.

We also provide per-scene breakdowns of Ref-Real [37] and NeRF-Casting Shiny Scenes [38] in Tab. 7. These results are consistent with the averaged results in the paper. All metrics are evaluated at the original resolution downsampled by a factor of 4, following prior works [38]. Notably, our method is more general and does not rely on manually estimated bounding boxes for foreground objects, which are essential for 3DGS-DR [51] to prevent optimization failure.

A.3. Comparison on Shiny Blender [37]

In Tab. 8, Fig. 10 and Fig. 11, we present additional quantitative and qualitative comparisons on the Shiny Blender dataset [38], which is rendered with environment maps under distant lighting assumption. The results show that although being designed for robustness on real-world data, our method effectively reconstructs accurate distant specular reflections, performing on par with or surpassing prior methods GaussianShader [15] and 3DGS-DR [18] specifically designed for environment map lighting scenarios. EnvGS considerably outperforms these methods in capturing near-field reflections caused by self-occlusions, as illustrated in the zoomed-in regions of the "toaster" scene. Moreover, our method reconstructs more accurate geometry, as shown in Fig. 11.

A.4. Comparison on Mip-NeRF 360 [3]

We perform additional comparisons on the Mip-NeRF 360 dataset [3], which consists of large-scale real-world scenes with primarily diffuse appearance and complex geometry. As shown in Tab. 9, our method is not limited to reflective scenes and can achieve comparable or superior performance to both state-of-the-art implcit [4] and explicit [17] methods.

Methods	Ref-Real [37] and NeRF-Casting [38]			
	PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑
ENVIDR	15.890	0.416	0.607	0.058
NDE	19.399	0.422	0.593	0.083
Ours	**27.947**	**0.794**	**0.189**	**26.221**

Table 5. **Quantitative comparison with object-level methods.**

Figure 7. **Qualitative comparison with object-level methods.**

A.5. Additional Baselines

We also compare our method with object-baselines including ENVIDR [22] and NDE [45]. While object-level methods perform well on synthetic data, they often struggle with real-world scenes and cannot real-time rendering speed on scenes with background, as shown in Tab. 5 and Fig. 7.

B. Additional Ablation Studies

B.1. Environment Representation Comparison

As described in Sec. 5.4, the environment representation plays a crucial role in capturing complex reflections. In Fig. 8, we provide additional qualitative comparisons between our *environment Gaussian* representation and the environment map representation. The "*w/* env. map 128" and "*w/* env. map 256" variants replace our core *environment Gaussian* representation with environment maps using six cubemaps at resolutions of 128 and 256, respectively. The results demonstrate that both environment map variants fail to capture the near-field reflections and tend to blur high-frequency reflection details, whereas our *environment Gaussian* representation excels at capturing complex reflections with high fidelity.

B.2. Speed Analysis

We conduct additional speed ablations on the "*hatchback*" from the NeRF-Casting Shiny Scenes [38] with resolution 3504 × 2336 (which we downsample by a factor of 4, as done in all baselines and experiments). The results are listed in Tab. 6.

Differentiable Gaussian tracing. As discussed in Sec. 4.2, rendering the *environment Gaussian* primitives with rasterization is impractical due to the uniqueness of each reflected ray. To validate this, we compare two alternative rendering strategies: (1) manually computing the ray-primitive intersections using PyTorch in a chunk-based manner ("*w/* PyTorch"), and (2) rasterizing the *environment Gaussian* primitives with a modified 3DGS [17] rasterizer using 1x1 tiles ("*w/* 1x1-tile rasterizer"). All three methods, including our Gaussian tracer, apply the same volume rendering equation as in Eq. 2. Quantitative results in Tab. 6 reveal that both alternative strategies take over an hour to render a single frame, whereas our Gaussian tracer achieves real-time rendering speeds, leveraging hardware-accelerated ray tracing.

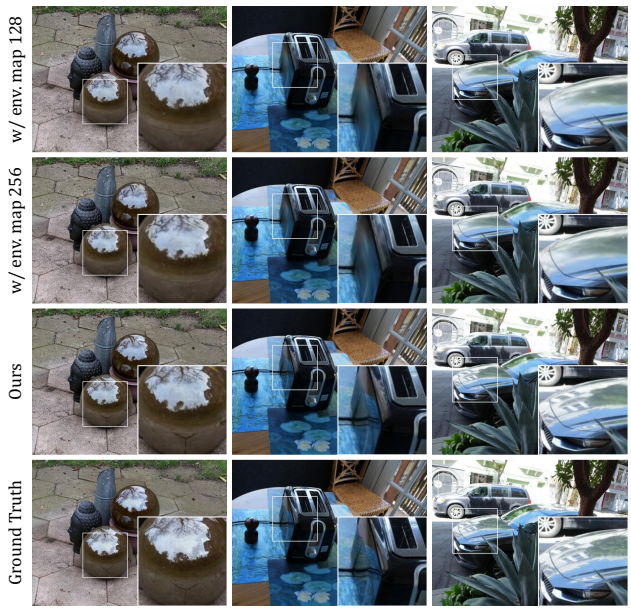

Figure 8. **Qualitative comparison between the environment map representation and our environment Gaussian representation**. Replacing the environment map with our environment Gaussian representation significantly improves the rendering quality, especially in capturing near-field reflections and high-frequency reflection details.

	PSNR ↑	SSIM ↑	LPIPS ↓	FPS ↑
w/ PyTorch	-	-	-	1/6157.613
w/ 1x1-tile rasterizer	-	-	-	1/11902.431
w/ 50% weight filting	27.137	0.824	0.192	36.216
w/ 75% weight filting	27.104	0.823	0.192	41.824
w/ 80% weight filting	27.033	0.822	0.193	44.215
w/ 90% weight filting	26.695	0.816	0.197	**47.149**
Ours	**27.220**	**0.838**	**0.177**	32.259

Table 6. **Runtime analysis of the proposed method on the *hatchback* of NeRF-Casting Shiny Scenes [38].** Rasterization or PyTorch-based ray tracing is impractical for rendering the *environment Gaussian* primitives. The acceleration techniques lead to minimal quality changes as shown by the cell.

Rendering speed analysis. As mentioned in Sec. 4.1, the rendering of our method consists of two main rounds: rasterization of the *base Gaussian* and ray tracing of the *en-*

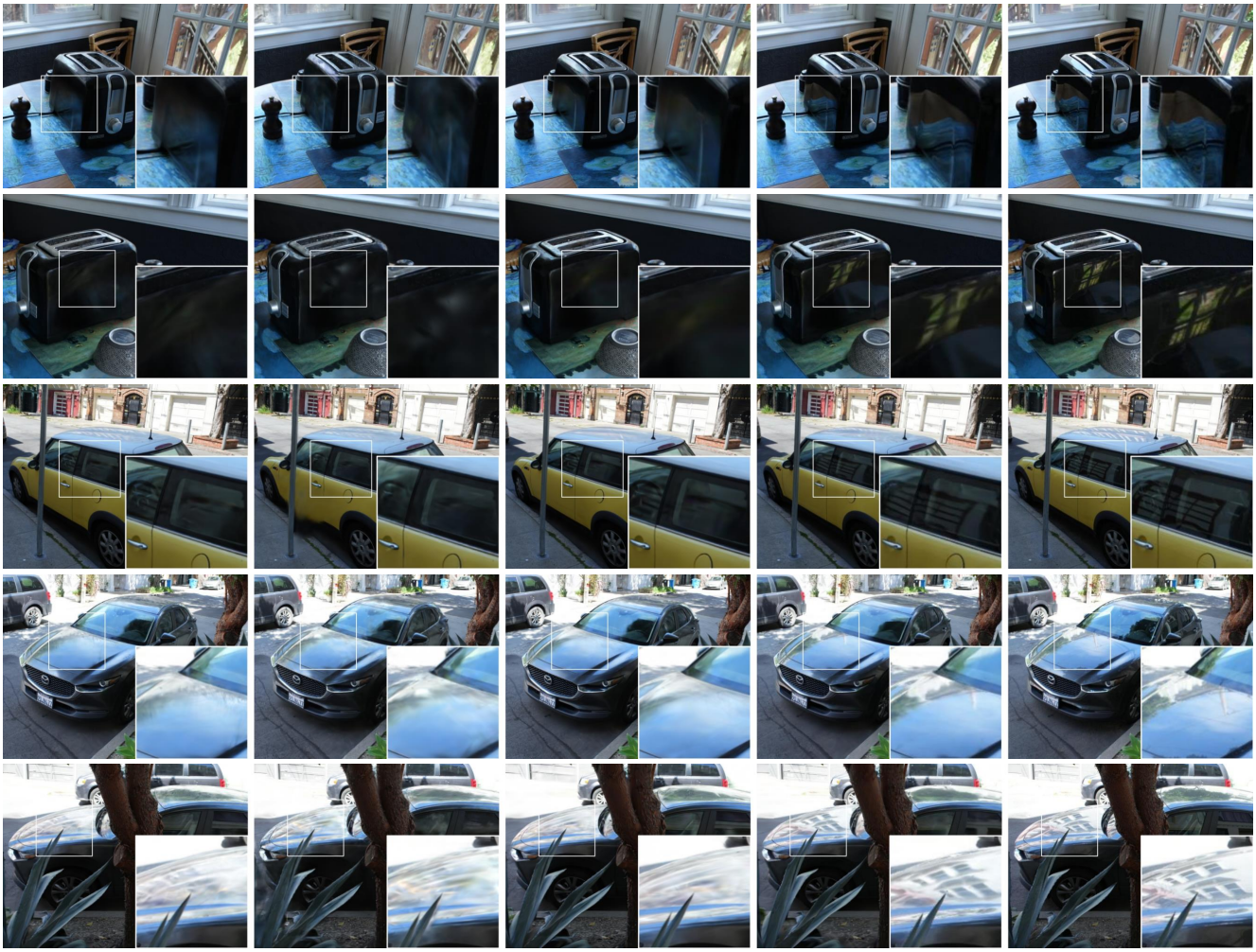

Figure 9. **Qualitative comparison on real scenes.** Our method significantly improves rendering quality over previous approaches, particularly in producing more detailed reflections. Zoom in for more details.

vironment Gaussian, and the final color is the blending of the two. Based on the fact that only a small portion of the scene surface contains strong specular reflections, we can further accelerate the rendering process by only tracing rays with high blending weights, which are only made possible by our tracing-based renderer. We ablate the effectiveness and quality impact of this acceleration technique, results are shown in Tab. 6.

B.3. Environment Gaussian Design

The Necessity of using a separate environment Gaussian primitives. To evaluate the decision to use separate Gaussian primitives for reflection modeling, we perform an experiment using a single set of Gaussian primitives for both reflection and base scene modeling. We first trace a camera ray to obtain the base color, normal and rendering weight, then trace a secondary ray to render the reflection color, ultimately combining these results using Eq. (5) to get the final color. However, we found that the experiment consistently failed to converge due to unavoidable interference between suboptimal geometry during optimization and incorrectly hitting Gaussian primitives from erroneous reflection directions, leading to an unstable training process.

C. Details of Environment Gaussian

We provide more details of our *base Gaussian* and *environment Gaussian*. The SH coefficients of both *base Gaussian* and *environment Gaussian* are set to two for the best results. The *environment Gaussian* is jointly optimized with the *base Gaussian*, and *environment Gaussian* constitutes around 15% of the *base Gaussian*, of average 300k Gaus-

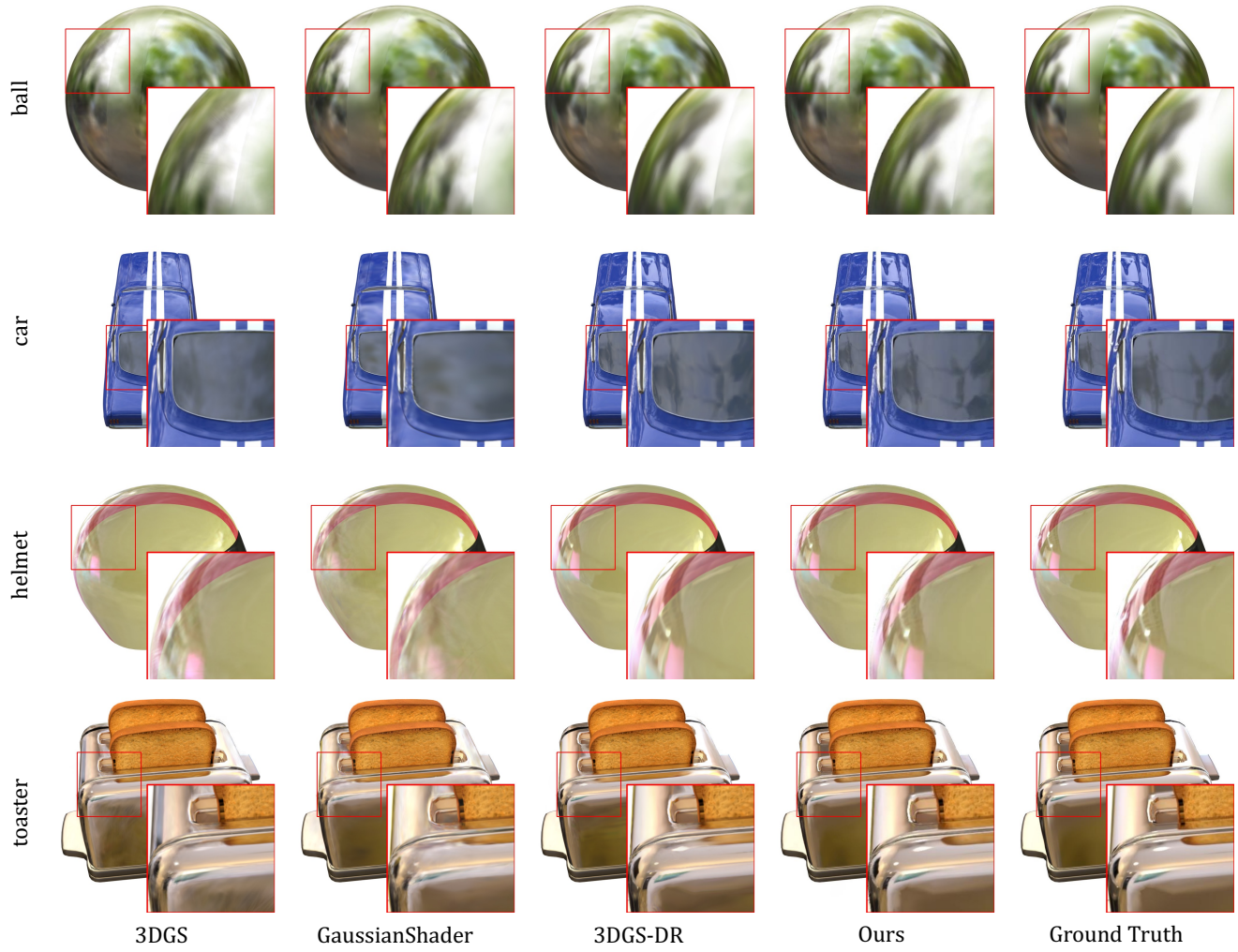

Figure 10. **Qualitative comparison on synthetic scenes.** Despite being designed for robustness on real-world data, our method effectively reconstructs accurate distant specular reflections and effectively captures near-field reflections caused by self-occlusions.

sian primitives using 70MB after training. For pruning, we follow the pruning method in the original 2DGS [13] and keep at most the top 630k *environment Gaussian* primitives based on rendering weights.

D. Details of Gradient Computation

To enable the joint optimization of *base Gaussian* and *environment Gaussian* primitives, which is essential for accurate geometry recovery and reflection reconstruction (as demonstrated in Sec. 5.4), our Gaussian tracer must be fully differentiable. This requires computing gradients with respect to the input reflected ray origin, $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{o}}$, and direction, $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{d}}$. These gradients are backpropagated through the surface position $\mathbf{x}$ and normal $\mathbf{n}$, obtained during the first rasterization stage, to the *base Gaussian* parameters for joint optimization.

Consider an input ray with origin $\mathbf{o}$ and direction $\mathbf{d}$, and a intersected triangle primitive i with vertices $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$. During ray traversal, the OptiX kernel utilizes the GPU's RT core to determine the intersection depth t_i, which is then used to compute the interaction position as $\mathbf{x}_i = \mathbf{o} + t_i\mathbf{d}$. This position is subsequently transformed into the local tangent plane of the corresponding 2D Gaussian, yielding $\mathbf{u}_i$ via Eq. 6 for Gaussian value evaluation. Note that the ray-triangle intersection depth can be manually computed as:

$$t_i = \frac{\mathbf{n}_i^\top(\mathbf{v}_1 - \mathbf{o})}{\mathbf{n}_i^\top \mathbf{d}}, \tag{12}$$

where $\mathbf{n}_i = (\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1)$ is the normal direction of the triangle. Then, we can apply the chain rule to calculate

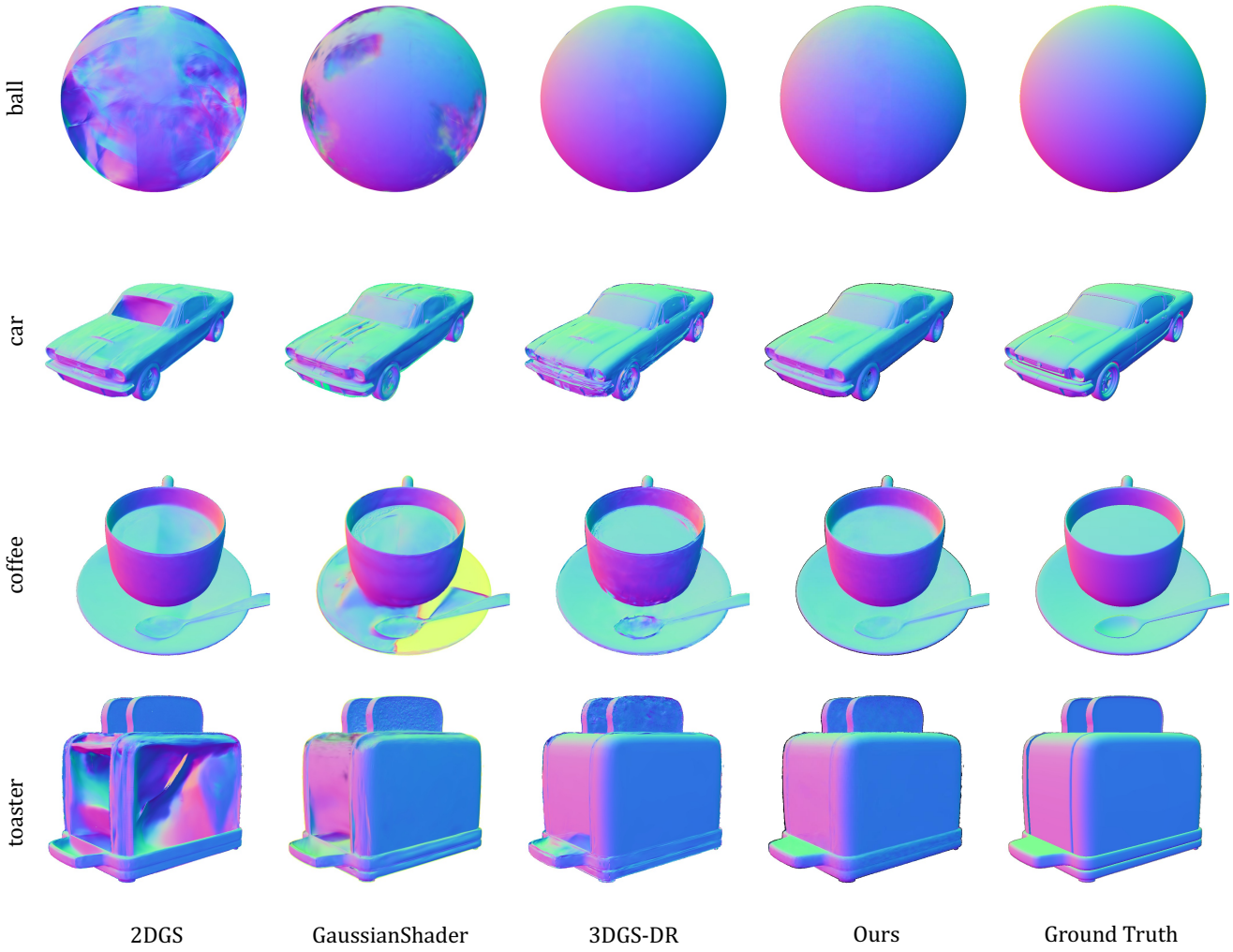

Figure 11. **Qualitative comparisons of normal produced by different methods.**

the derivatives w.r.t. the ray origin and direction:

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{o}} &= \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{x}_i}\frac{\mathrm{d}\mathbf{x}_i}{\mathrm{d}\mathbf{o}} + \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}t}\frac{\mathrm{d}t}{\mathrm{d}\mathbf{o}} \\ &= \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{x}_i} + \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}t}\cdot\frac{-\mathbf{n}_i}{\mathbf{n}_i^\top\mathbf{d}},\end{aligned} \tag{13}$$

and

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{d}} &= \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{x}_i}\frac{\mathrm{d}t}{\mathrm{d}\mathbf{d}} + \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}t}\frac{\mathrm{d}t}{\mathrm{d}\mathbf{d}} \\ &= \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{x}_i}\cdot t_i + \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}t}\cdot\frac{-\mathbf{n}_i^\top(\mathbf{v}_1-\mathbf{o})}{\mathbf{n}_i\cdot(\mathbf{n}_i^\top\mathbf{d})^2}.\end{aligned} \tag{14}$$

This gradient flow enables the joint optimization of the reflection appearance of *environment Gaussian* alongside the geometry and base appearance of *base Gaussian*, enhancing both geometry accuracy and reflection fidelity.

PSNR ↑	Methods	Ref-Real [37]			NeRF-Casting Shiny Scenes [38]			
		sedan	*toycar*	*spheres*	*compact*	*grinder*	*hatchback*	*toaster*
Non real-time	Ref-NeRF* [37]	25.390	22.750	21.120	30.550	33.910	25.210	32.660
	UniSDF [39]	24.680	24.150	22.270	29.720	33.720	27.010	32.900
	ZipNeRF [4]	25.850	23.410	21.770	31.100	34.670	27.780	33.410
	NeRF-Casting [38]	26.770	24.200	23.040	29.730	34.000	27.490	32.870
Real-time	3DGS [17]	25.240	23.910	21.950	28.945	30.885	26.201	29.410
	2DGS [13]	25.065	24.282	22.064	28.415	30.164	25.893	28.630
	GaussianShader [14]	24.081	23.137	21.408	27.474	26.572	24.959	26.641
	3DGS-DR [51]	25.445	23.582	21.539	28.692	30.129	25.985	29.141
	Ours	26.156	24.746	22.949	29.608	33.331	27.220	31.618

SSIM ↑	Methods	Ref-Real [37]			NeRF-Casting Shiny Scenes [38]			
		sedan	*toycar*	*spheres*	*compact*	*grinder*	*hatchback*	*toaster*
Non real-time	Ref-NeRF* [37]	0.721	0.612	0.542	0.907	0.880	0.842	0.932
	UniSDF [39]	0.700	0.639	0.567	0.895	0.879	0.845	0.937
	ZipNeRF [4]	0.733	0.626	0.545	0.913	0.887	0.870	0.944
	NeRF-Casting [38]	0.739	0.641	0.597	0.884	0.882	0.853	0.938
Real-time	3DGS [17]	0.713	0.636	0.573	0.877	0.864	0.838	0.928
	2DGS [13]	0.704	0.662	0.595	0.857	0.854	0.819	0.917
	GaussianShader [14]	0.668	0.625	0.573	0.851	0.799	0.805	0.884
	3DGS-DR [51]	0.714	0.635	0.571	0.857	0.849	0.813	0.914
	Ours	0.727	0.667	0.619	0.871	0.895	0.838	0.938

LPIPS ↓	Methods	Ref-Real [37]			NeRF-Casting Shiny Scenes [38]			
		sedan	*toycar*	*spheres*	*compact*	*grinder*	*hatchback*	*toaster*
Non real-time	Ref-NeRF* [37]	0.270	0.257	0.257	0.105	0.123	0.156	0.111
	UniSDF [39]	0.309	0.245	0.243	0.122	0.132	0.160	0.107
	ZipNeRF [4]	0.260	0.243	0.238	0.096	0.111	0.130	0.082
	NeRF-Casting [38]	0.254	0.246	0.238	0.148	0.114	0.155	0.096
Real-time	3DGS [17]	0.301	0.237	0.248	0.154	0.181	0.179	0.123
	2DGS [13]	0.344	0.246	0.254	0.193	0.217	0.215	0.147
	GaussianShader [14]	0.371	0.293	0.278	0.189	0.289	0.217	0.169
	3DGS-DR [51]	0.322	0.249	0.251	0.196	0.219	0.228	0.146
	Ours	0.287	0.208	0.229	0.159	0.151	0.177	0.110

Table 7. **Ref-Real [37] and NeRF-Casting [38] per-scene breakdowns.** All metrics are evaluated at the original resolution downsample by a factor of 4 as prior works [38].

PSNR ↑	Methods	Shiny Blender Scenes [37]					
		ball	*car*	*coffee*	*helmet*	*teapot*	*toaster*
Non real-time	Ref-NeRF [37]	47.460	30.820	34.210	29.680	47.900	25.700
	UniSDF [39]	44.100	29.860	33.170	38.840	48.760	26.180
	NeRF-Casting [38]	45.460	30.450	33.180	39.100	49.980	26.190
Real-time	3DGS [17]	27.650	27.260	32.300	28.220	45.710	20.990
	2DGS [13]	25.990	26.730	32.360	27.300	44.940	20.272
	GaussianShader [14]	29.081	26.940	31.147	28.883	43.379	23.584
	3DGS-DR [51]	33.533	30.236	34.580	31.518	47.038	26.823
	3iGS [36]	27.640	27.510	32.580	28.210	46.040	22.690
	Ours	32.567	30.598	34.312	31.470	46.582	27.427

SSIM ↑	Methods	Shiny Blender Scenes [37]					
		ball	*car*	*coffee*	*helmet*	*teapot*	*toaster*
Non real-time	Ref-NeRF [37]	0.995	0.955	0.974	0.958	0.998	0.922
	UniSDF [39]	0.993	0.954	0.973	0.990	0.998	0.945
	NeRF-Casting [38]	0.994	0.964	0.973	0.988	0.999	0.950
Real-time	3DGS [17]	0.937	0.931	0.972	0.951	0.996	0.894
	2DGS [13]	0.935	0.932	0.973	0.952	0.997	0.892
	GaussianShader [14]	0.955	0.930	0.969	0.955	0.996	0.907
	3DGS-DR [51]	0.979	0.957	0.976	0.971	0.997	0.943
	3iGS [36]	0.938	0.930	0.973	0.951	0.997	0.908
	Ours	0.971	0.958	0.974	0.968	0.997	0.945

LPIPS ↓	Methods	Shiny Blender Scenes [37]					
		ball	*car*	*coffee*	*helmet*	*teapot*	*toaster*
Non real-time	Ref-NeRF [37]	0.059	0.041	0.078	0.075	0.004	0.095
	UniSDF [39]	0.039	0.047	0.078	0.021	0.004	0.072
	NeRF-Casting [38]	0.044	0.033	0.074	0.018	0.002	0.073
Real-time	3DGS [17]	0.162	0.047	0.079	0.081	0.008	0.125
	2DGS [13]	0.155	0.051	0.080	0.080	0.008	0.126
	GaussianShader [14]	0.145	0.066	0.085	0.086	0.011	0.105
	3DGS-DR [51]	0.104	0.038	0.076	0.050	0.006	0.082
	3iGS [36]	0.156	0.045	0.076	0.073	0.006	0.099
	Ours	0.138	0.037	0.085	0.052	0.006	0.077

Table 8. **Quantitative results on Shiny Blender Scenes [37].**

PSNR ↑	Methods	Mip-NeRF 360 [3]								
		bicycle	*bonsai*	*counter*	*flowers*	*garden*	*kitchen*	*room*	*stump*	*treehill*
Non real-time	Ref-NeRF* [37]	24.910	32.290	26.020	21.630	27.450	31.610	31.680	25.910	21.790
	UniSDF [39]	24.670	32.860	29.260	21.830	27.460	31.730	31.250	26.390	23.510
	ZipNeRF [4]	25.800	34.460	29.380	22.400	28.200	32.500	32.650	27.550	23.890
	NeRF-Casting [38]	24.920	33.810	28.840	21.750	27.310	32.260	31.660	25.640	23.220
Real-time	3DGS [17]	25.250	31.980	28.700	21.520	27.410	30.320	30.630	26.550	22.490
	2DGS [13]	24.741	31.246	28.107	21.131	26.723	30.372	30.679	26.123	22.427
	GaussianShader [14]	23.103	29.278	26.639	20.267	26.290	27.125	24.098	24.668	20.552
	3DGS-DR [51]	24.869	31.232	27.730	21.116	27.142	28.999	30.068	25.473	21.344
	Ours	25.209	31.946	29.017	21.551	27.709	31.660	31.020	25.423	22.686

SSIM ↑	Methods	Mip-NeRF 360 [3]								
		bicycle	*bonsai*	*counter*	*flowers*	*garden*	*kitchen*	*room*	*stump*	*treehill*
Non real-time	Ref-NeRF* [37]	0.723	0.935	0.875	0.592	0.845	0.922	0.914	0.731	0.634
	UniSDF [39]	0.737	0.939	0.888	0.606	0.844	0.919	0.914	0.759	0.670
	ZipNeRF [4]	0.769	0.949	0.902	0.642	0.860	0.928	0.925	0.800	0.681
	NeRF-Casting [38]	0.747	0.945	0.887	0.605	0.836	0.924	0.911	0.749	0.653
Real-time	3DGS [17]	0.771	0.938	0.905	0.605	0.868	0.922	0.914	0.775	0.638
	2DGS [13]	0.734	0.931	0.893	0.575	0.844	0.917	0.907	0.756	0.618
	GaussianShader [14]	0.700	0.917	0.875	0.541	0.842	0.888	0.839	0.701	0.579
	3DGS-DR [51]	0.740	0.933	0.889	0.578	0.852	0.908	0.904	0.750	0.607
	Ours	0.734	0.933	0.899	0.589	0.854	0.923	0.910	0.726	0.621

LPIPS ↓	Methods	Mip-NeRF 360 [3]								
		bicycle	*bonsai*	*counter*	*flowers*	*garden*	*kitchen*	*room*	*stump*	*treehill*
Non real-time	Ref-NeRF* [37]	0.256	0.182	0.213	0.317	0.132	0.121	0.206	0.261	0.294
	UniSDF [39]	0.243	0.184	0.206	0.320	0.136	0.124	0.206	0.242	0.265
	ZipNeRF [4]	0.208	0.173	0.185	0.273	0.118	0.116	0.196	0.193	0.242
	NeRF-Casting [38]	0.231	0.176	0.203	0.312	0.142	0.118	0.216	0.244	0.273
Real-time	3DGS [17]	0.205	0.205	0.204	0.336	0.103	0.129	0.220	0.210	0.317
	2DGS [13]	0.267	0.227	0.229	0.374	0.145	0.146	0.243	0.258	0.374
	GaussianShader [14]	0.275	0.242	0.243	0.380	0.131	0.170	0.307	0.277	0.394
	3DGS-DR [51]	0.254	0.230	0.231	0.368	0.135	0.151	0.247	0.248	0.375
	Ours	0.233	0.180	0.194	0.339	0.112	0.120	0.207	0.262	0.347

Table 9. **Quantitative results on Mip-NeRF 360 [3].** The results in "Non Real-time" are borrowed from NeRF-Casting [38], and Ref-NeRF* is an improved version of Ref-NeRF [37] that uses Zip-NeRF's [4] geometry model.