Triangle Splatting for Real-Time Radiance Field Rendering

Jan Held* [1,2] Renaud Vandeghen* [1] Adrien Deliege [1] Abdullah Hamdi [4]
Silvio Giancola [2] Daniel Rebain [3] Anthony Cioppa [1] Bernard Ghanem [2]
Andrea Vedaldi [4] Andrea Tagliasacchi [5,6,7] Marc Van Droogenbroeck [1]

[1] University of Liège [2] KAUST [3] University of British Columbia [4] University of Oxford
[5] Simon Fraser University [6] University of Toronto [7] Google DeepMind

* Equal contribution

Figure 1. We propose a new representation for differentiable rendering based on the most classical of 3D primitives, the *triangle*. We show how a triangle soup (*i.e.* unstructured, disconnected triangles) can be optimized effectively, generating state-of-the-art novel view synthesis images while being immediately compatible with classical rendering pipelines. The figure shows the final rendered output (left), a visualization of soft blending (middle), and the rendering of a random subset of triangles to highlight their structure (right).

Abstract

The field of computer graphics was revolutionized by models such as NeRF and 3D Gaussian Splatting, displacing triangles as the dominant representation for photogrammetry. In this paper, we argue for a triangle comeback. We develop a differentiable renderer that directly optimizes triangles via end-to-end gradients. We achieve this by rendering each triangle as differentiable splats, combining the efficiency of triangles with the adaptive density of representations based on independent primitives. Compared to popular 2D and 3D Gaussian Splatting methods, our approach achieves competitive rendering and convergence speed, and demonstrates high visual quality. On the Mip-NeRF360 dataset, our method outperforms concurrent non-volumetric primitives in visual fidelity and achieves higher perceptual quality than the state-of-the-art Zip-NeRF on indoor scenes. Triangles are simple, compatible with standard graphics stacks and GPU hardware, and highly efficient. Our results highlight the efficiency and effectiveness of triangle-based representations for high-quality novel view synthesis. Triangles bring us closer to mesh-based optimization by combining classical computer graphics with modern differentiable rendering frameworks. The project page is https://trianglesplatting.github.io/

1. Introduction

One of the enduring challenges in 3D vision and graphics is identifying a truly *universal* primitive for representing 3D content in a differentiable form, enabling gradient-based optimization of geometry and appearance. Despite extensive research, no single data structure has emerged as a silver bullet. Instead, researchers have explored a variety of approaches, including neural fields [29], hash tables [30], convex primitives [8, 14], and Gaussians [19], among others. Conversely, in conventional graphics pipelines, the triangle remains the undisputed workhorse. Game engines and other real-time systems primarily rely on triangles, as GPUs feature dedicated hardware pipelines for ultra-efficient triangle processing and rendering. Although other primitives exist (*e.g.*, quads in 2D or tetrahedra in 3D), they can always be subdivided into triangles. Moreover, surface reconstruction in 3D vision and graphics predominantly relies on triangle meshes to represent continuous, watertight geometry in an efficient, renderable form [18].

Despite their ubiquity, triangles are difficult to optimize in differentiable frameworks due to their discrete nature. Early attempts at differentiable optimization softened the non-differentiable occlusion at polygon edges, enabling gradients from image loss to flow into geometry and appearance parameters [17, 25]. However, these methods require a predefined mesh template, making them unsuitable when the scene's topology is unknown a priori. As a result, they struggle to capture fine geometric details and adapt to novel structures. To address these challenges, researchers adopted volumetric primitives, such as anisotropic 3D Gaussians in 3DGS [19], which can be optimized for high-quality novel view synthesis. However, the unbounded support of Gaussians makes it difficult to define the "surface" of the representation, and their inherent smoothness hinders accurate modeling of sharp details. Surface structures can be partially restored using 2D Gaussian Splatting [15] or 3D convex polytopes [14]. Yet, a pivotal question remains:
Can triangles themselves be optimized directly?

Learning to optimize a "triangle soup" (*i.e.* unstructured, disconnected triangles), as shown in the right side of Figure 1, via gradient-based methods could represent a major step towards the goal of template-free mesh optimization. Such an approach leverages decades of GPU-accelerated triangle processing and the mature mesh processing literature, making it easier to integrate these techniques within differentiable rendering pipelines.

In this work, we introduce **Triangle Splatting**, a real-time differentiable renderer that splats a soup of triangles into screen space while enabling end-to-end gradient-based optimization. Triangle Splatting merges the adaptability of Gaussians with the efficiency of triangle primitives, surpassing 3D Gaussian Splatting (3DGS), 2D Gaussian Splatting (2DGS), and 3D Convex Splatting (3DCS) in visual fidelity, training speed, and rendering throughput. The optimized triangle soup is directly compatible with any standard mesh-based renderer. To our knowledge, Triangle Splatting is the first splatting-based approach to directly optimize triangles for novel-view synthesis and 3D reconstruction, delivering state-of-the-art results while bridging classical rendering pipelines with modern differentiable frameworks.

Contributions. **(i)** We propose Triangle Splatting, a novel approach that directly optimizes triangles, bridging computer graphics and radiance fields. **(ii)** We introduce a differentiable window function for soft triangle boundaries, enabling effective gradient flow. **(iii)** Triangle Splatting delivers high visual quality and rendering speed, surpassing Zip-NeRF on indoor scenes in perceptual quality (LPIPS) and structural similarity (SSIM). **(iv)** The optimized triangles are compatible with standard mesh-based renderers, enabling seamless integration into traditional graphics pipelines.

2. Related work

Neural radiance fields have become the de-facto standard for image-based 3D reconstruction [28]. A large body of work has since addressed NeRF's slow training and rendering by introducing multi-resolution grids or hybrid representations [5, 10, 23, 30], or baking procedures for real-time playback [7, 13, 31]. Improvements in robustness include anti-aliasing [1–3], handling unbounded scenes [2, 37], and few-shot generalization [4, 9, 16]. Despite their success, implicit fields still require costly volume integration at render time. Our Triangle Splatting sidesteps this by optimizing *explicit* triangles that are traced once per pixel, leading to comparable fidelity but orders-of-magnitude faster rendering. For example, our triangles render ten times faster than Instant-NGP [30], while matching its optimization speed and achieving significantly higher visual fidelity.

Primitive-based differentiable rendering.

Differentiable renderers back-propagate image loss to scene parameters, enabling end-to-end optimization of explicit primitives such as points [11, 17], voxels [10], meshes [17, 25, 26], and Gaussians [19]. 3D Gaussian Splatting [19] demonstrated that millions of Gaussians can be fitted in minutes and rendered in real time. Follow-up work improved anti-aliasing [36], or offered exact volumetric integration [27] Because Gaussians have infinite support and inherently smooth fall-off, they struggle with sharp creases and watertight surfaces; recent extensions therefore experiment with alternative kernels [**?**], learnable basis functions [6], or linear primitives [35]. Convex Splatting [14] replaced Gaussians with smooth convexes, capturing hard edges more faithfully, but at the cost of slow optimization time and larger memory footprints. Compared

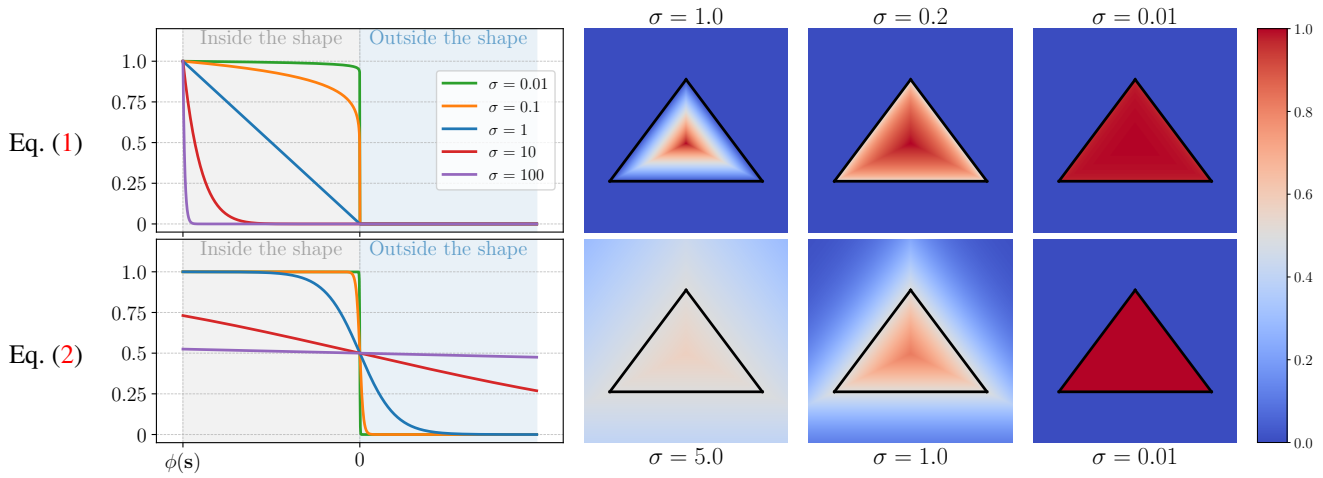

Figure 2. **Triangle window function (1D and 2D)**. We visualize the window functions of prior works [8, 14] (bottom) vs. the one introduced in our paper (top) in both 1D (left) and 2D (right). We show how the window function changes as we vary the smoothness control parameter σ. As σ decreases, note that both can approximate the window function of a triangle. However, as σ increases, the support of Eq. (2) exceeds the footprint of the triangle, making it unsuitable for *rasterization* workloads. In the limit, Eq. (2) becomes globally supported, with a window value of 0.5 everywhere, causing every triangle to contribute to the color of every pixel in the image.

with Gaussian [15, 19] or Convex Splatting [14], which explored either *volumetric* (*e.g.* Gaussian, voxel) or *solid* (*e.g.* convex, tetrahedral) primitives, Triangle Splatting proposes *surface* primitives, aligning with the surface of solid objects most typically found in real-world scenes. In extensive experiments, we show that Triangle Splatting surpasses 3DGS [19], 2DGS [15], and 3DCS [14] in visual quality and speed of rendering and optimization.

3. Method

We address the problem of reconstructing a photorealistic 3D scene from multiple images. We propose a scene representation that enables efficient, differentiable rendering and can be directly optimized by minimizing a rendering loss. Similar to prior work, the scene is represented by a large collection of simple geometric primitives. Where 3DGS used 3D Gaussians [19], 3DCS used 3D convex hulls [14], and 2DGS used 2D Gaussians [15], we propose the simplest and most efficient primitive, *triangles*. First, Sec. 3.1 explains how these triangles are rendered on an image. Then, Sec. 3.2 describes how we adaptively prune and densify the triangle representation. Finally, Sec. 3.3 describes how to optimize the triangles' parameters to fit the input images.

3.1. Differentiable rasterization

Our primitives are 3D triangles T_{3D}, each defined by three vertices $\mathbf{v}_i \in \mathbb{R}^3$, a color $\mathbf{c}$, a smoothness parameter σ and an opacity o. The vertices can move freely during optimization. To render a triangle, we first project each vertex $\mathbf{v}_i$ to the image plane using a standard pinhole camera model. The projection involves the intrinsic camera matrix $\mathbf{K}$ and the camera pose (parameterized by rotation $\mathbf{R}$ and translation $\mathbf{t}$): $\mathbf{q}_i = \mathbf{K}(\mathbf{R}\mathbf{v}_i + \mathbf{t})$, with $\mathbf{q}_i \in \mathbb{R}^2$ forming the projected triangle T_{2D} in the 2D image space. Instead of rendering the triangle as fully opaque, we weigh its influence smoothly using a window function I mapping pixels $\mathbf{p}$ to values in the $[0, 1]$ range. As we discuss below, the choice of this function is of critical importance. Once the triangles are projected, the color of each image pixel $\mathbf{p}$ is computed by accumulating contributions from all overlapping triangles, in depth order, treating the value $I(\mathbf{p})$ as opacity. The rendering equation is the same as the one used in prior works [14, 19], and refer the reader to [34] for its derivation.

A new window function. We first describe how the window function I is defined, which is one of our core contributions. We start by defining the *signed distance field* (SDF) ϕ of the 2D triangle in image space. This is given by:

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

where $\mathbf{n}_i$ are the unit normals of the triangle edges pointing outside the triangle, and d_i are offsets such that the triangle is given by the zero-level set of the function ϕ. The signed distance field ϕ thus takes positive values outside the triangle, negative values inside, and equals zero on its boundary. Let $\mathbf{s} \in \mathbb{R}^2$ be the *incenter* of the projected triangle T_{2D} (*i.e.*, the point inside the triangle with minimum signed distance). With this, we define our new window function I as:

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

$$\text{such that} \quad I(\mathbf{p}) \begin{cases} = 1 & \text{at the triangle incenter,} \\ = 0 & \text{at the triangle boundary,} \\ = 0 & \text{outside the triangle.} \end{cases} \tag{1}$$

Here, the parameter $\sigma > 0$ controls the *smoothness* of the window function. $\phi(\mathbf{p})$ is negative inside the triangle, and $\phi(\mathbf{s})$ is its smallest (most negative) value, so the ratio $\phi(\mathbf{p})/\phi(\mathbf{s})$ is positive inside the triangle, equal to one at the incenter, and equal to zero at the boundary. This formulation has three important properties: (i) there is a point (the incenter) inside the triangle where the window function obtains the maximum value of one; (ii) the window function is zero at the boundary and outside the triangle so that its support tightly fits the triangle; and (iii) a single parameter can easily control the smoothness of the window function.

Figure 2 shows that all three properties are satisfied for different values of σ. For $\sigma \to 0$ our window function converges to the window function of the triangle. For larger values, the window function transitions smoothly from zero at the boundary to one in the middle, and for $\sigma \to \infty$ the window function becomes a delta function at the incenter.

Window function alternatives. Related works [8, 14] use the $\mathrm{LogSumExp}$ function to approximate $\max$ in the signed distance field: $\phi(\mathbf{p}) = \log \sum_{i=1}^{3} \exp L_i(\mathbf{p})$. However, we observed that, for small triangles, this $\max$ approximation is poor, to the point that *only one* of the three vertices has any influence on the final shape. We thus opted to use the actual $\max$ function which, while not smooth everywhere, accurately defines the signed distance field. Further, related work [8, 14, 25] also use a different definition for the window function I based on sigmoid:

$$I(\mathbf{p}) = \mathrm{sigmoid}(-\sigma^{-1}\,\phi(\mathbf{p}))$$
$$\text{such that} \quad I(\mathbf{p}) \begin{cases} > \frac{1}{2} & \text{inside the triangle,} \\ = \frac{1}{2} & \text{at the triangle boundary,} \\ < \frac{1}{2} & \text{outside the triangle.} \end{cases} \tag{2}$$

This definition fails to meet properties (i) and (ii) above, as the maximum can be less than 1, and the support of the window function can be significantly larger than the triangle. This is illustrated in Figure 2, where $\sigma \to \infty$ results in a constant value everywhere in $\mathbb{R}^2$.

Simpler depth-dependent scaling. In 3D Gaussian Splatting, each 3D Gaussian is defined in world space by a full covariance matrix, which is mapped to image space by accounting for the projective transformation, resulting in a 2D covariance matrix inversely proportional to depth. The effect is a 2D Gaussian whose size scales with depth. In Convex Splatting [14], the 2D convex hull scales automatically with depth, but not the window function smoothness parameter σ. Because the latter is defined in pixel units, it must be scaled "manually" to achieve a depth-consistent effect. In our case, this is unnecessary because of the normalization in Eq. (1): σ results in consistently scaled 2D window functions for all depth values.

A simpler and more efficient primitive. Triangle Splatting reduces complexity compared to concurrent approaches. Unlike 3DGS, Triangle Splatting avoids the need for EWA, a core component of 3DGS that is complex to implement and whose quality degrades when moving away from the optical center. Also, compared to 3D Convex Splatting, Triangle Splatting avoids the complex and costly kernel required to compute active edges and the need to calculate the convex hull per primitive, simplifying implementation and improving speed.

Figure 3. **Triangle pruning**. To reduce floaters, we prune triangles seen in fewer than two views with over one pixel of coverage, removing those overfitted to a single training view.

3.2. Adaptive pruning and splitting

Triangles have a compact spatial domain (and, therefore, a compact gradient); hence, we need a mechanism to control coverage of the spatial domain by the triangles, modulating their density and thus representation power at different locations. This is achieved by pruning and densification routines (respectively decreasing and increasing the representation power), analogously to 3DGS [19].

Pruning. During rasterization, we calculate the maximum volume rendering blending weight $T \cdot o$ (where T is transmittance, and o is opacity) for each triangle, and prune all triangles whose maximum weight is less than a defined threshold τ_{prune} across all training views. Additionally, we prune all triangles that are not rendered at least *twice* with more than one pixel. In other words, we remove triangles that explain small amounts of data within a single view and are therefore likely to have overfitted to the training data. Figure 3 illustrates the impact of this pruning strategy.

Densification. Instead of relying on manually tuned heuristics for adding shapes, we adopt the probabilistic framework based on MCMC introduced by Kheradmand et al. [20]. At each densification step, we sample from a probability distribution to guide where new shapes should be added. Kheradmand et al. [20] stochastically allocates new Gaussians proportionally to the *opacity*, and we extend this idea to our representation by incorporating the sharpness parameter σ. Since both opacity and σ are learned during training, we build the probability distribution directly from these parameters by alternating between using the inverse of σ and the opacity for Bernoulli sampling. In particular, we preferentially sample triangles with low σ values, *i.e. solid* triangles. Because of our window function, the triangle's influence is bounded by its projected geometry, and the diffusion

remains confined within the triangle itself. In high-density regions, many triangles overlap at each pixel, allowing each shape to adopt a higher σ and thus a softer contribution. In contrast, in low-density regions, where fewer triangles influence a pixel, each triangle must contribute more to the reconstruction. As a result, it adopts a lower σ to increase the contribution across its interior, ensuring maximal coverage within the geometric bounds and producing a more solid appearance.

Further, taking inspiration from Kheradmand et al. [20], we design updates to avoid disrupting the sampling process. In particular, we require that the probability of the state (*i.e.* the current set of parameters of all triangles) remains *unchanged* before and after the transition, allowing it to be interpreted as a move between equally probable samples, preserving the integrity of the Markov chain. To preserve a consistent representation across sampling steps, we apply *midpoint subdivision* to the selected triangles. Each triangle is split into four smaller ones by connecting the midpoints of its edges, ensuring that the combined area and spatial region of the new triangles match that of the original. As in our parametrization, a triangle is defined by 3D vertices, making this operation straightforward to perform. Finally, if a triangle is smaller than the threshold τ_{small}, we do not split it. Instead, we clone it and add random noise along the triangle's plane orientation.

3.3. Optimization

Our method starts from a set of images and their corresponding camera parameters, calibrated via SfM [32], which also produces a sparse point cloud. We create a 3D triangle for each 3D point in the sparse point cloud. We optimize the 3D vertex positions $\{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3\}$, sharpness σ, opacity o, and spherical harmonic color coefficients $\mathbf{c}$ of all such 3D triangles by minimizing the rendering error from the given posed views. The initialization is done as follows. Let $\mathbf{q} \in \mathbb{R}^3$ be a SfM 3D point and let d be the average Euclidean distance to its three nearest neighbors. We initialize the corresponding 3D triangle to be approximately equilateral, randomly oriented, and with a size proportional to d. To do this, we sample uniformly at random three vertices $\{\mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3\}$ from the unit sphere, we scale them by d, and we add $\mathbf{q}$ to center them at the point $\mathbf{q}$: $\mathbf{v}_i = \mathbf{q} + k \cdot d \cdot \mathbf{u}_i$, where $k \in \mathbb{R}$ is a scaling constant. Our training loss combines the photometric $\mathcal{L}_1$ and $\mathcal{L}_{\text{D-SSIM}}$ terms [19], the opacity loss $\mathcal{L}_o$ [20], and the distortion $\mathcal{L}_d$ and normal $\mathcal{L}_n$ losses [15]. Finally, we add a size regularization term $\mathcal{L}_s = 2\left\|(\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0)\right\|_2^{-1}$, to encourage larger triangles. The final loss $\mathcal{L}$ is given by:

$$\mathcal{L} = (1-\lambda)\mathcal{L}_1 + \lambda\mathcal{L}_{\text{D-SSIM}} + \beta_1\mathcal{L}_o + \beta_2\mathcal{L}_d + \beta_3\mathcal{L}_n + \beta_4\mathcal{L}_s\,. \tag{3}$$

The full list of thresholds and hyperparameters is detailed in the *Supplementary Material*.

4. Experiments

We compare our method to concurrent approaches on the standard benchmarks Mip-NeRF360 [2] and Tanks and Temples (T&T) [22]. We consider 3DCS [14], which is the most closely related method, as well as to non-volumetric primitives such as BBSplat [33] and 2DGS [15]. We also consider primitive-based volumetric methods, including 3DGS [19], 3DGS-MCMC [20], and DBS [24]. Additionally, we evaluate against implicit neural rendering methods such as Instant-NGP [30], Mip-NeRF360 [2], and the state-of-the-art in novel view synthesis Zip-NeRF [3]. We evaluate the visual quality using standard metrics: SSIM, PSNR, and LPIPS. We report training time, rendering speed, and memory usage, measured on an NVIDIA A100.

Implementation details. We use spherical harmonics of degree 3, resulting in 59 parameters per triangle, matching the number of parameters for a single 3D Gaussian [19].

4.1. Novel-view synthesis

Table 1 presents the quantitative results on the T&T dataset, as well as on the indoor and outdoor scenes from the Mip-NeRF360 dataset. In comparison with planar primitive methods, Triangle Splatting achieves a higher visual quality, with a significant improvement in LPIPS (the metric that best correlates with human visual perception). Specifically, Triangle Splatting improves over 2DGS and BBSplat by 25% and 19% on Mip-NeRF360, respectively. Triangle Splatting achieves consistently better LPIPS scores in outdoor scenes, surpassing both 2DGS and BBSplat. Similarly, on the T&T dataset, Triangle Splatting yields a substantial improvement over 2DGS and BBSplat.

While our method yields slightly lower PSNR values, this metric does not fully capture visual quality due to its inherent limitations (PSNR generally rewards overly smooth reconstructions that regress to the mean). As a result, smooth representations, such as Gaussian-based primitives, tend to perform better under PSNR, whereas sharper transitions from solid shapes may be penalized. Figure 4 illustrates this limitation of PSNR: although our reconstruction looks visually superior to that of 2DGS, the PSNR *in the highlighted region* is 3 PSNR higher for 2DGS.

Against volumetric primitive methods, Triangle Splatting achieves high visual quality, with a significant improvement in LPIPS. Specifically, Triangle Splatting improves over 3DGS and 3DCS by 10% and 7% respectively on Mip-NeRF 360. Compared to implicit methods, Triangle Splatting matches the visual quality of the state-of-the-art Zip-NeRF, with only a 0.002 difference in LPIPS, while delivering over 500× faster rendering performance.

Triangles work particularly well in indoor or structured outdoor scenes with walls, cars, and other well-defined surfaces, where they can closely approximate geometry.

	Outdoor Mip-NeRF 360			Indoor Mip-NeRF 360			Aver. Mip-NeRF 360			Tanks & Temples			
	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	LPIPS ↓	PSNR ↑	SSIM ↑	FPS ↑
	Implicit Methods												
Instant-NGP [30]	-	-	-	-	-	-	0.331	25.59	0.699	0.305	21.92	0.745	14.4
Mip-NeRF360 [2]	0.283	24.47	0.691	0.179	31.72	0.917	0.237	27.69	0.792	0.257	22.22	0.759	0.14
Zip-NeRF [3]	0.207	25.58	0.750	0.167	32.25	0.926	0.189	28.54	0.828	-	-	-	-
	Volumetric Primitives												
3DGS [19]	0.234	24.64	0.731	0.189	30.41	0.920	0.214	27.21	0.815	0.183	23.14	0.841	154
3DGS-MCMC [20] ‡	0.210	25.51	0.76	0.208	31.08	0.917	0.210	27.98	0.829	0.19	24.29	0.86	129
DBS [24] †	0.246	25.10	0.751	0.22	32.29	0.936	0.234	28.29	0.833	0.140	24.85	0.870	150
3DCS [14]	0.238	24.07	0.700	0.166	31.33	0.927	0.207	27.29	0.802	0.156	23.94	0.851	33
	Non-Volumetric Primitives												
BBSplat [33] †	0.281	23.55	0.669	0.178	30.62	0.921	0.236	26.69	0.781	0.172	**25.12**	**0.868**	66
2DGS [15]	0.246	**24.34**	0.717	0.195	30.40	0.916	0.252	27.04	0.805	0.212	23.13	0.831	122
Triangle Splatting	**0.217**	24.27	**0.722**	**0.160**	**30.80**	**0.928**	**0.191**	**27.16**	**0.814**	**0.143**	23.14	0.857	**165**

Table 1. **Quantitative results**. We evaluate our method on both indoor and outdoor scenes, achieving state-of-the-art performance on the *indoor* benchmark. Across all datasets, our approach consistently outperforms other non-volumetric primitives. Bold scores indicate the best results among *non-volumetric* methods. † denotes reproduced results, while ‡ marks results reported in [24].

his makes Triangle Splatting especially effective indoors, achieving state-of-the-art performance and outperforming 3DCS and Zip-NeRF. Unstructured outdoor scenes are more challenging for planar primitives due to sparse or ambiguous geometry, making it harder to maintain visual consistency across views. Even so, Triangle Splatting narrows the performance gap and surpasses 3DGS and 3DCS on the T&T dataset, achieving a lower LPIPS.

Figure 5 presents a qualitative comparison between Triangle Splatting, 3DCS, and 2DGS. We consistently produce sharper reconstructions, particularly in high-frequency regions. For instance, in the `Bicycle` scene, it more accurately captures fine details, as highlighted.

Speed & Memory. Table 2 compares the memory consumption and rendering speed of concurrent methods. Although BBSplat uses fewer primitives, it suffers from considerably slower training and inference. Triangle Splatting shows strong scalability, despite using more primitives, it renders 4× faster than BBSplat and achieves a 40% speedup over 2DGS. Triangle Splatting significantly outperforms 3DCS, achieving 2× faster training and 4× faster inference.

	Train ↓	FPS ↑	Memory ↓
ZipNerf	5h	0.18	569MB
3DGS	42m	134	734MB
3DCS	87m	25	666MB
BBSplat	96m	25	**175MB**
2DGS	**29m**	64	484MB
Ours	39m	**97**	795MB

Table 2. **Speed & Memory**. Triangle Splatting scales efficiently, achieving faster training and rendering even with more primitives.

Unlike 3DCS, Triangle Splatting does not require computing a 2D convex hull and rendering is more efficient. Triangle Splatting computes the signed distance for only three lines per pixel, whereas 3DCS requires calculations for six lines, effectively doubling the per-pixel computational cost.

5. Ablations

Loss terms. Table 3 shows the impact on performance when removing different components of our pipeline on Mip-NeRF360. The opacity regularization $\mathcal{L}_o$ is the most

Figure 4. **Limitations of PSNR**. Due to its smoothness, Gaussians tends to perform better on the PSNR metric, which evaluates pixel-wise differences, despite being blurrier. In the highlighted areas, our method achieves a PSNR of 18.41, compared to 21.27 for 2DGS.

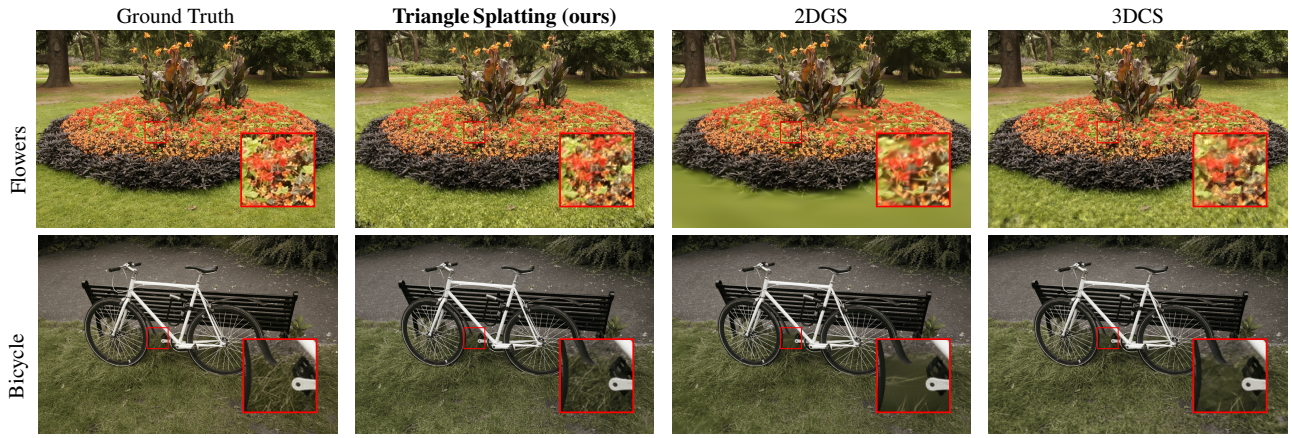

Figure 5. **Qualitative results**. We visually compare our method to 3DCS [14] and 2DGS [15]. Triangle Splatting captures finer details and produces more accurate renderings of real-world scenes, with less blurry results than 2DGS, and a higher visual quality than 3DCS [14].

	LPIPS ↓	PSNR ↑	SSIM ↑
Triangle Splatting	**0.191**	**27.14**	**0.814**
w/o $\mathcal{L}_s$	**0.191**	26.97	0.812
w/o σ^{-1} sampling	0.193	27.03	0.811
w/o o sampling	0.193	27.02	0.811
w/o $\mathcal{L}_d$ & $\mathcal{L}_n$	0.194	27.11	0.811
w/o $\mathcal{L}_o$	0.207	26.38	0.794

Table 3. **Ablation study**. We isolate the impact of each component by removing them individually.

impactful, encouraging lower opacity values so that unnecessary triangles become transparent and are reallocated. The regularization term $\mathcal{L}_s$ encourages larger triangles, increasing PSNR, particularly in indoor scenes. The initial point cloud is often extremely sparse along walls, frequently with few or no initial triangles. Without $\mathcal{L}_s$, triangles expand too slowly to reach scene boundaries. By promoting larger shapes, this regularization enables faster growth, allowing triangles to extend into underrepresented regions. Sampling based on σ^{-1} or opacity alone yields similar performance, while combining both leads to improved results.

Window functions. Figure 6 compares the sigmoid-based window function with our proposed version. In regions with sparse initial point cloud, particularly in the background, the sigmoid function fails to recover the scene structure. Since the sigmoid is not bounded by its geometry's vertices and can grow arbitrarily large, the optimizer tends to increase the sigma values instead of moving vertices to cover empty areas. This results in small yet very smooth shapes, making them difficult to optimize. In contrast, our normalized window function enforces spatial bounds, which encourages vertices to move and fill underrepresented regions. As the size of each shape is explicitly defined, the optimization process becomes more stable and effective.

Triangle *vs.* convex splatting. In 3DCS, each convex is defined by six vertices. When the number is reduced to three, the shape degenerates into a triangle. In Figure 7, we present a visual comparison between Triangle Splatting and 3DCS using triangles. Triangle Splatting does not produce line artifacts, which often appear in 3DCS when handling degenerate triangles. Furthermore, Triangle Splatting obtains a higher visual quality with an improvement of 0.05 LPIPS, 0.61 PSNR and 0.045 SSIM on Mip-NeRF360.

Figure 6. **Ablation study (window function)**. We compare against the Sigmoid function (left) which fails to recover background regions accurately, while ours doesn't (right).

Figure 7. **Ablation study (triangles as convexes)**. We compare our method (left) with 3DCS using three-vertex convexes (right), which results in degenerate geometry, as seen in the zoom-ins.

6. Triangle splatting in a traditional renderer

While the main goal of this work is to introduce triangles as a novel primitive for high-quality novel view synthesis, their benefits extend beyond this task. To obtain a model with mesh-like topology, we set opacity to 1 and σ to 0, yielding opaque, hard triangles that require only vertex positions and per-triangle colors to be stored in a standard mesh format (*e.g.*, .off), with no post-processing. This marks a sig-

nificant improvement over 2DGS. The final triangle soup can be integrated into *any* mesh renderer. Alternatively, the full triangle splatting model can be incorporated into traditional rendering pipelines using order-independent transparency to correctly handle semi-transparent triangles. To demonstrate practicality, we evaluate triangle splatting in a traditional renderer across both use cases.

Stochastic Triangle Splatting renderer. As demonstrated by StochasticSplats [21], alpha blending-based models like 3DGS can be implemented with z-buffer rasterization by employing stochastic transparency. This approach is particularly well suited to Triangle Splatting, as it relies on invoking the fragment shader many times for each final pixel to reduce noise. Unlike Gaussian-based models, which must evaluate the Gaussian PDF in the fragment shader, Triangle Splatting can perform almost all computations in the vertex shader, lowering the cost of additional samples. We include a link in the *Supplementary Material* to a browser-based demo of this method, which includes a naive implementation of temporal filtering for noise reduction. It should be noted that more sophisticated temporal antialiasing pipelines are a standard feature in modern game engines. Table 4 shows that even a consumer laptop with an Iris Xe renders 4.3M triangles at high speed, while high-end GPUs like the RTX 4090 exceed 1000 FPS. As shown in Figure 8, Stochastic Triangle Splatting also preserves high visual quality when integrated into a traditional renderer.

Mesh-based novel-view synthesis. Table 5 compares mesh-based novel view synthesis between Triangle Splatting and 2DGS. For 2DGS, the mesh was extracted via TSDF fusion and textured with a neural color field trained for 5k iterations following 30k iterations of main training (see MILo [12] for more details). In contrast, Triangle Splatting requires no additional processing, only the opacity is set to 1 and σ to 0. Compared to Triangle Splatting, 2DGS relies on post-processing, lacks direct game engine compatibility, and achieves lower visual quality. For Triangle Splatting we simply set the opacity to 1 and σ to 0, without any additional processing. 2DGS requires post-processing, lacks direct game engine compatibility, and delivers lower visual quality. Triangle Splatting improves PSNR by 37%, SSIM by 12%, and reduces LPIPS by 2.7%, yielding better visuals without post-processing.

Triangle Splatting uses far fewer triangles ($\approx$ 1M vs. $\approx$ 7M). It reconstructs background regions more accurately, while 2DGS tends to focus on the object center and often leaves backgrounds sparse or missing, reducing visual quality. Triangle Splatting opens the door to future work, such as developing training strategies specifically for high visual quality in game engines (for example, gradually pushing opacity and σ toward 1 and 0, respectively), or connecting triangles during training to obtain a watertight mesh.

Hardware	OS	TFLOPS	HD	Full HD	4k
Iris Xe	Windows	2	33	23	15
MacBook M4	MacOS	8	120	90	40
RTX4090	Linux	48	1058	759	327

Table 4. **Stochastic renderer: FPS across different hardware and resolutions**. Evaluated on *Garden* ($\approx$ 4.3M triangles) with 1 sample per pixel. OS stands for operating system.

Figure 8. **Qualitative rendering results with Stochastic Triangle Splatting**. Most computations run in the vertex shader, enabling real-time rendering in a game engine without quality loss.

	Post-processing	PSNR ↑	LPIPS ↓	SSIM ↑	#T
2DGS†	Yes	15.36	0.474	0.498	≈7M
Ours	No	**21.05**	**0.462**	**0.558**	**≈1M**

Table 5. **Mesh-based novel view synthesis**. Comparison between Triangle Splatting and 2DGS on MipNeRF 360. We achieve higher image quality (higher PSNR/SSIM, lower LPIPS), and require far fewer triangles (#T). † results obtained from [12].

7. Conclusions

We have introduced **Triangle Splatting**, a novel differentiable rendering technique that directly optimizes triangle primitives for novel-view synthesis. Leveraging the same primitive used in classical mesh representations, our method bridges the gap between neural rendering and traditional graphics pipelines. We have shown that Triangle Splatting outperforms concurrent methods in visual quality while retaining the efficiency, and compatibility of the triangle. Moreover, it opens several directions for future research, including connecting triangles during training to obtain watertight meshes and optimizing σ and opacity for high visual quality in game engines. These results establish Triangle Splatting as a promising step toward mesh-aware neural rendering, unifying decades of GPU-accelerated graphics with modern differentiable frameworks.

Acknowledgments. J. Held, A. Deliege and A. Cioppa are funded by the F.R.S.-FNRS. The research reported in this publication was supported by funding from KAUST Center of Excellence on GenAI, under award number 5940. This work was also supported by KAUST Ibn Rushd Postdoc Fellowship program. The present research benefited from computational resources made available on Lucia, the Tier-1 supercomputer of the Walloon Region, infrastructure funded by the Walloon Region under the grant agreement n°1910247.

References

[1] 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. In *IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, pages 5835–5844, Montréal, Can., 2021. Inst. Electr. Electron. Eng. (IEEE). 2

[2] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Mip-NeRF 360: Unbounded anti-aliased neural radiance fields. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 5460–5469, New Orleans, LA, USA, 2022. Inst. Electr. Electron. Eng. (IEEE). 2, 5, 6

[3] Jonathan T. Barron, Ben Mildenhall, Dor Verbin, Pratul P. Srinivasan, and Peter Hedman. Zip-NeRF: Anti-aliased grid-based neural radiance fields. In *IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, pages 19640–19648, Paris, Fr., 2023. Inst. Electr. Electron. Eng. (IEEE). 2, 5, 6

[4] Eric R. Chan, Connor Z. Lin, Matthew A. Chan, Koki Nagano, Boxiao Pan, Shalini de Mello, Orazio Gallo, Leonidas Guibas, Jonathan Tremblay, Sameh Khamis, Tero Karras, and Gordon Wetzstein. Efficient geometry-aware 3D generative adversarial networks. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 16102–16112, New Orleans, LA, USA, 2022. Inst. Electr. Electron. Eng. (IEEE). 2

[5] Anpei Chen, Zexiang Xu, Andreas Geiger, Jingyi Yu, and Hao Su. TensoRF: Tensorial radiance fields. In *Eur. Conf. Comput. Vis. (ECCV)*, pages 333–350, Tel Aviv, Israël, 2022. Springer Nat. Switz. 2

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

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

[8] Boyang Deng, Kyle Genova, Soroosh Yazdani, Sofien Bouaziz, Geoffrey Hinton, and Andrea Tagliasacchi. CvxNet: Learnable convex decomposition. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 31–41, Seattle, WA, USA, 2020. Inst. Electr. Electron. Eng. (IEEE). 2, 3, 4

[9] Yilun Du, Cameron Smith, Ayush Tewari, and Vincent Sitzmann. Learning to render novel views from wide-baseline stereo pairs. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 4970–4980, Vancouver, Can., 2023. Inst. Electr. Electron. Eng. (IEEE). 2

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

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

[12] 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. *arXiv*, abs/2506.24096, 2025. 8

[13] Peter Hedman, Pratul P. Srinivasan, Ben Mildenhall, Jonathan T. Barron, and Paul Debevec. Baking neural radiance fields for real-time view synthesis. In *IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, pages 5855–5864, Montréal, Can., 2021. Inst. Electr. Electron. Eng. (IEEE). 2

[14] Jan Held, Renaud Vandeghen, Abdullah Hamdi, Adrien Deliège, Anthony Cioppa, Silvio Giancola, Andrea Vedaldi, Bernard Ghanem, and Marc Van Droogenbroeck. 3D convex splatting: Radiance field rendering with 3D smooth convexes. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 21360–21369, Nashville, TN, USA, 2025. Inst. Electr. Electron. Eng. (IEEE). 2, 3, 4, 5, 6, 7

[15] Binbin Huang, Zehao Yu, Anpei Chen, Andreas Geiger, and Shenghua Gao. 2D Gaussian splatting for geometrically accurate radiance fields. In *ACM SIGGRAPH Conf. Pap.*, pages 1–11, Denver, CO, USA, 2024. ACM. 2, 3, 5, 6, 7

[16] Ajay Jain, Matthew Tancik, and Pieter Abbeel. Putting NeRF on a diet: Semantically consistent few-shot view synthesis. In *IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, pages 5865–5874, Montréal, Can., 2021. Inst. Electr. Electron. Eng. (IEEE). 2

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

[18] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *Symp. Geom. Process.*, pages 61–70, Cagliari, Italy, 2006. Eurographics Assoc. 2

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

[20] Shakiba Kheradmand, Daniel Rebain, Gopal Sharma, Weiwei Sun, Jeff Tseng, Hossam Isack, Abhishek Kar, Andrea Tagliasacchi, and Kwang Moo Yi. 3D Gaussian splatting as Markov chain Monte Carlo. In *Adv. Neural Inf. Process. Syst. (NeurIPS)*, pages 80965–80986, Vancouver, Can., 2024. Curran Assoc. Inc. 4, 5, 6

[21] Shakiba Kheradmand, Delio Vicini, George Kopanas, Dmitry Lagun, Kwang Moo Yi, Mark Matthews, and Andrea Tagliasacchi. StochasticSplats: Stochastic rasterization for sorting-free 3D Gaussian splatting. *arXiv*, abs/2503.24366, 2025. 8

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

[23] Jonas Kulhanek and Torsten Sattler. Tetra-NeRF: Representing neural radiance fields using tetrahedra. In *IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, pages 18412–18423, Paris, Fr., 2023. Inst. Electr. Electron. Eng. (IEEE). 2

[24] Rong Liu, Dylan Sun, Meida Chen, Yue Wang, and Andrew Feng. Deformable beta splatting. *arXiv*, abs/2501.18630, 2025. 5, 6

[25] Shichen Liu, Weikai Chen, Tianye Li, and Hao Li. Soft rasterizer: A differentiable renderer for image-based 3D reasoning. In *IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, pages 7707–7716, Seoul, South Korea, 2019. Inst. Electr. Electron. Eng. (IEEE). 2, 4

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

[27] 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. *arXiv*, abs/2410.01804, 2024. 2

[28] 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 *Eur. Conf. Comput. Vis. (ECCV)*, pages 405–421, Virtual conference, 2020. Springer Int. Publ. 2

[29] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. NeRF. *Commun. ACM*, 65(1):99–106, 2021. 2

[30] Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.*, 41(4):1–15, 2022. 2, 5, 6

[31] Christian Reiser, Rick Szeliski, Dor Verbin, Pratul Srinivasan, Ben Mildenhall, Andreas Geiger, Jon Barron, and Peter Hedman. MERF: Memory-efficient radiance fields for real-time view synthesis in unbounded scenes. *ACM Trans. Graph.*, 42(4):1–12, 2023. 2

[32] Johannes L. Schonberger and Jan-Michael Frahm. Structure-from-motion revisited. In *IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 4104–4113, Las Vegas, NV, USA, 2016. Inst. Electr. Electron. Eng. (IEEE). 5

[33] David Svitov, Pietro Morerio, Lourdes Agapito, and Alessio Del Bue. BillBoard splatting (BBSplat): Learnable textured primitives for novel view synthesis. *arXiv*, abs/2411.08508, 2024. 5, 6

[34] Andrea Tagliasacchi and Ben Mildenhall. Volume rendering digest (for NeRF). *arXiv*, abs/2209.02417, 2022. 3

[35] Nicolas von Lützow and Matthias Nießner. LinPrim: Linear primitives for differentiable volumetric rendering. *arXiv*, abs/2501.16312, 2025. 2

[36] Zehao Yu, Anpei Chen, Binbin Huang, Torsten Sattler, and Andreas Geiger. Mip-splatting: Alias-free 3D Gaussian splatting. In *IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 19447–19456, Seattle, WA, USA, 2024. Inst. Electr. Electron. Eng. (IEEE). 2

[37] Kai Zhang, Gernot Riegler, Noah Snavely, and Vladlen Koltun. NeRF++: Analyzing and improving neural radiance fields. *arXiv*, abs/2010.07492, 2020. 2