# Fast 3D Gaussian Splatting Rendering via Easily Integrable Improvements

Laurens Diels, Michiel Vlaminck, Wilfried Philips, *Senior Member, IEEE,* and Hiep Luong

***Abstract*—The recently introduced 3D Gaussian Splatting and subsequent methods have achieved significantly reduced inference times for novel view synthesis. To reduce this rendering time even further, in this paper we propose four improvements which are fully compatible with the high-level Gaussian Splatting formulation and can thus be incorporated into most methods based on this paradigm. Most notably, we alter the way Gaussians are duplicated across tiles by allowing for non-square axis-aligned Gaussian bounding boxes whose sizes take into account the Gaussian's opacity information. Our experiments demonstrate that we can decrease the 3D Gaussian Splatting rendering times by up to a factor of almost 4.**

***Index Terms*—3D rendering, computational efficiency, Gaussian Splatting**

## I. Introduction

Novel view synthesis has received increased interest in the last few years thanks to the revolutionary methods of Neural Radiance Fields (NeRF) [1] and 3D Gaussian Splatting [2]. NeRFs represent a scene implicitly, encoded into the parameters of a neural network. While this method produces high quality renders, it is very slow, both at training and inference time. Although subsequent papers [3], [4], [5] have been able to reduce these significantly, real-time rendering on commodity hardware is still not attainable without resolution compromises.

3D Gaussian Splatting (3DGS) takes a different, more explicit approach, representing a scene as a collection of Gaussian functions with opacity and color information. For a given camera pose, these are then splatted together to form a new image. In this manner 3DGS achieves competitive training times, and extremely fast rendering. It has consequently spawned a class of follow-up methods. Most of these, including [6], [7], [8], [9], directly build upon 3DGS's open-source `diff-gaussian-rasterization` package. Any improvement to the core Gaussian Splatting rendering process will, therefore, automatically help such derived methods.

We propose the following improvements.

- We alter the way 2D Gaussian axis-aligned bounding boxes are created, allowing for non-square rectangles taking into account opacity information. This significantly reduces the number of duplicated Gaussians across tiles.
- We immediately and explicitly filter out Gaussians whose projection falls too far outside of the screen. Additionally, we make sure to only launch GPU-threads for non-filtered out Gaussians, in order to reduce warp divergence.
- To speed up the evaluation of the opacity-scaled Gaussian functions in each tile, we precompute features common to the different pixels in this tile.
- We make the tile size increase with the rendering resolution.

We will go over these improvements in more detail in Section III, and will evaluate them experimentally in Section IV. But first we review the theory of 3D Gaussian Splatting rendering.

## II. Overview of 3D Gaussian Splatting rendering

In this section we provide a brief overview of the way 3DGS handles image rendering. For more details, we refer to the original paper [2].

A trained 3DGS model represents a scene as a collection $\{(\mathcal{G}_i, \tau_i, \boldsymbol{H}_i)\}_i$, where

- $\mathcal{G}_i$ is a 3D Gaussian function defined by a mean $\boldsymbol{m}_i \in \mathbb{R}^3$ and covariance matrix $\boldsymbol{\mathcal{C}}_i \in \mathbb{R}^{3\times3}$:
$$\mathcal{G}_i(\boldsymbol{x}; \boldsymbol{m}_i, \boldsymbol{\mathcal{C}}_i) = \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{m}_i)^T \boldsymbol{\mathcal{C}}_i^{-1}(\boldsymbol{x}-\boldsymbol{m}_i)\right)$$
for all $\boldsymbol{x} \in \mathbb{R}^3$;
- $\tau_i \in [0,1]$ is the associated opacity;
- $\boldsymbol{H}_i \in \mathbb{R}^{(d+1)^2\times 3}$ is a matrix of spherical harmonics coefficients up to degree $d$ (typically 3), used to represent viewpoint-dependent color.

To render an image for given camera intrinsics and extrinsics, the 3D Gaussians are first projected to image-space (or normalized device coordinates). By linearizing this projection, a 3D Gaussian $\mathcal{G}_i$ is transformed into a 2D Gaussian $G_i$, defined by a 2D mean $\boldsymbol{\mu}_i \in \mathbb{R}^2$ and covariance matrix $\boldsymbol{\Sigma}_i \in \mathbb{R}^{2\times2}$. The depth of the 3D Gaussian mean is also stored and used to filter out Gaussians in front of the near clipping plane. The spherical harmonics coefficients are evaluated to an RGB color $\boldsymbol{c}_i \in [0,1]^3$ (although technically the components are allowed to exceed 1 at this stage).

Next, to obtain a rendered image $\boldsymbol{R}$, the Gaussians are sorted based on depth and alpha-blended together. Concretely, at every pixel location $(u,v)$ we obtain the RGB color

$$\boldsymbol{R}(u,v) = \sum_j \alpha_j(u,v) T_j(u,v) \boldsymbol{c}_j \in [0,1]^3. \tag{1}$$

Here

$$\alpha_j(u,v) = \tau_j G_j\left((u,v)^T; \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j\right) \in [0,1], \tag{2}$$

All authors are with the imec research group of Image Processing and Interpretation (IPI) of Ghent University, as well as with the university's UAV Research Center (URC).

$$T_j(u,v) = \prod_{k<j} (1 - \alpha_k(u,v)) \tag{3}$$

is the transmittance, and Gaussians are ordered by increasing depth. For reasons of numerical stability, indices $j$ with $\alpha_j(u,v) < \epsilon := 1/255$ are skipped in (1) and (3).

As for any given pixel the contribution of most Gaussians will be negligible, summing over all Gaussians is terribly inefficient. Therefore, a tiling scheme is employed where the image to render is divided into an axis-aligned regular grid of tiles. By converting 2D Gaussians into a (not necessarily tight) axis-aligned bounding box, it can easily be determined which Gaussians affect which tiles. The rendering process described above is then handled independently for each tile, where for each pixel in a tile, we only sum over the sorted list of Gaussians affecting the tile. For massive parallelization on a GPU, each tile corresponds to a thread-block, and each pixel in a tile to a thread.

In summary, 3DGS rendering consists of three phases:

1) Projection: Project Gaussians to camera, filter
2) Tiling: Divide screen in grid of tiles, assign each (remaining) 2D Gaussian to tiles, duplicate Gaussians (one copy for each assigned tile), sort per tile on depth
3) Tile rendering: Render all pixels in a tile by alpha-blending the (duplicated) Gaussians assigned to this tile.

## III. Proposed improvements

We propose four improvements to the reference 3DGS implementation [2], available at https://github.com/graphdeco-inria/gaussian-splatting. Our changes do not affect the high-level description of the previous section. The most notable change is in the way how it is determined whether a Gaussian should be assigned to a tile (Subsection III-A).

In this section we will mostly consider a single Gaussian at a time. To not complicate the notation we will then drop the subscript of Gaussian index ($i$ or $j$).

### A. Tile intersections

The conversion of a 2D Gaussian into an axis-aligned bounding box is done as follows in the reference 3DGS implementation. First the 2D Gaussian is converted to a 99% confidence elliptical disk. Note that this requires (implicitly) adding normalization to the Gaussian function in order to obtain a probability density function. The length of the semi-major axis of the ellipse is approximately $r := 3\sqrt{\lambda}$, where $\lambda$ is the largest eigenvalue of $\mathbf{\Sigma}$ and where 3 is an approximation of the square root of 9.21, the 0.99-quantile of a $\chi^2$-distribution with two degrees of freedom. This $r$ is then used as the *radius* of a circle centered at the 2D Gaussian mean, the circumscribed square of which is used as the Gaussian's bounding box. This is illustrated in Fig.1a. Determining which tiles the bounding box intersects is now trivial.

In our proposed approach we replace the probabilistic argument and additionally allow for non-square bounding boxes. Since a Gaussian will be skipped at locations where its $\alpha$-value (2) is less than $\epsilon$, we will consider the Gaussian level set $\mathcal{E}$ at $\epsilon/\tau$, defined by $\boldsymbol{x} \in \mathcal{E}$ if and only if

$$(\boldsymbol{x} - \boldsymbol{\mu})^T \mathbf{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}) = 2 \ln \frac{\tau}{\epsilon} =: \gamma \tag{4}$$

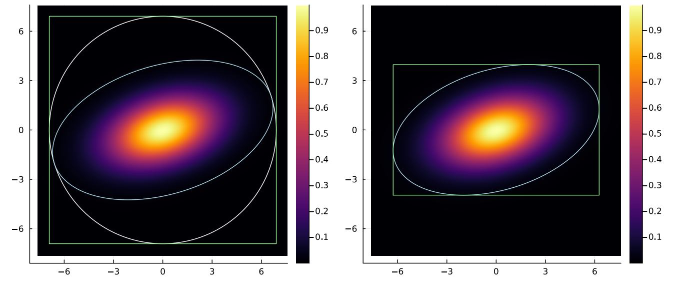

Fig. 1. The axis-aligned bounding box for a Gaussian with projected mean $\boldsymbol{\mu} = \mathbf{0}$, covariance matrix $\mathbf{\Sigma} = \begin{pmatrix} 5 & 1 \\ 1 & 2 \end{pmatrix}$ and opacity $\tau = 0.2$, for the original 3DGS implementation (a) and our proposed approach (b). The Gaussian ellipse ((a): boundary of 99% confidence elliptical disk of the associated Gaussian distribution, (b): $\frac{\epsilon}{\tau}$-level set, see also (4)) is drawn in light blue and the bounding box in light green. For the original approach (a) we also draw in white the circumscribed circle to the ellipse, illustrating their notion of *radius*.

for any $\boldsymbol{x} \in \mathbb{R}^2$. Note that $\mathcal{E} = \emptyset$ when $\gamma < 0$, i.e., $\tau < \epsilon$. From now on we will then assume that $\gamma \geq 0$. Using $s := x_1 - \mu_1$ and $t := x_2 - \mu_2$, and writing the entries of $\mathbf{\Sigma}^{-1}$ in terms of those of (the symmetric) $\mathbf{\Sigma}$, we obtain the equation

$$s^2 \Sigma_{22} - 2st\Sigma_{12} + t^2 \Sigma_{11} - \gamma \det \mathbf{\Sigma} = 0.$$

Interpreted as a quadratic equation in $s$, it has discriminant

$$\begin{aligned} D(t) &= 4t^2\Sigma_{12}^2 - 4t^2\Sigma_{11}\Sigma_{22} + 4\gamma\Sigma_{22} \det \mathbf{\Sigma} \\ &= 4\left(\gamma\Sigma_{22} - t^2\right) \det \mathbf{\Sigma}. \end{aligned}$$

As $\det \mathbf{\Sigma} > 0$ by virtue of $\mathbf{\Sigma}$ being positive semi-definite, our quadratic equation then has a solution in $s$ if and only if $t^2 \leq \gamma\Sigma_{22}$. Similarly, it has a solution in $t$ precisely when $s^2 \leq \gamma\Sigma_{11}$. Together, we obtain the tight bounding box

$$[\mu_1 - \sqrt{\gamma\Sigma_{11}}, \mu_1 + \sqrt{\gamma\Sigma_{11}}] \times [\mu_2 - \sqrt{\gamma\Sigma_{22}}, \mu_2 + \sqrt{\gamma\Sigma_{22}}].$$

A comparison between our bounding box and the original 3DGS one can be found in Fig.1.

### B. Explicit filtering

In the reference 3DGS implementation, Gaussians whose 2D projection falls too far outside of the screen, are not filtered out. Instead, for the purpose of calculating the 2D covariance matrix, and for this calculation alone, the projected mean is clamped to an enlarged screen boundary. In practice, this mostly results in these Gaussians hardly contributing to any on-screen pixels. Therefore, we propose to just immediately filter out such Gaussians. (In the CUDA part of the original implementation the code for this already exists, but is commented out.)

Additionally, in the reference implementation, filtering (based on depth) occurs implicitly, via `return` statements. While simple, this approach results in strongly differing workloads across different GPU-threads. We propose instead to first explicitly filter, writing the indices of the kept Gaussians into new arrays, and to subsequently spawn threads only for the filtered Gaussians. In this manner all threads have a similar workload and we reduce warp divergence.

### C. More rendering precomputations

In the final tile rendering phase, we need to compute $\alpha_j(u,v)$ extremely frequently: for each pixel $(u,v)$ in a tile, and each (duplicated) Gaussian $G_j$ affecting this tile. Therefore, if we can reduce the computations for obtaining each $\alpha_j(u,v)$, we expect to see some performance gains, even if we have to perform a few extra calculations beforehand. Below we will again focus on a single Gaussian and drop the subscript $j$.

Recall that

$$\alpha(u,v) = \tau \exp\left(-\frac{1}{2}\left(\begin{pmatrix}u\\v\end{pmatrix} - \boldsymbol{\mu}\right)^T \boldsymbol{\Sigma}^{-1}\left(\begin{pmatrix}u\\v\end{pmatrix} - \boldsymbol{\mu}\right)\right).$$

In the reference implementation this is handled by first precomputing and storing $\boldsymbol{\Sigma}^{-1} = \begin{pmatrix}p & q\\ q & r\end{pmatrix}$ in the projection phase, and then calculating $d_x = u - \mu_1$, $d_y = v - \mu_2$ and

$$\alpha(u,v) = \tau \cdot \exp\left(-0.5(pd_x^2 + rd_y^2) - qd_xd_y\right),$$

requiring 4 additions, 8 multiplications and one exp-evaluation. We propose to expand this exponent as a polynomial of $u$ and $v$, and also absorb the $\tau$ factor into the exp. We therefore first precompute the coefficients

$$\begin{aligned} z_1 &= -0.5p & z_4 &= p\mu_1 + q\mu_2 \\ z_2 &= -0.5r & z_5 &= q\mu_1 + r\mu_2 \\ z_3 &= -q & z_6 &= -0.5(p\mu_1^2 + r\mu_2^2) - q\mu_1\mu_2 + \ln\tau. \end{aligned}$$

For each pixel (GPU thread) we then calculate $u^2$, $v^2$ and $uv$ once, and finally compute $\alpha$ for each Gaussian as

$$\alpha(u,v) = \exp(z_1u^2 + z_2v^2 + z_3uv + z_4u + z_5v + z_6).$$

This requires 5 additions, 5 multiplications and one exp-evaluation.

For the best performance we should put this precomputation directly after the filtering step. However, $u^2$, $v^2$, $uv$, and $z_6$ can get quite large at high rendering resolutions, leading to numerical stability issues for 32-bit floats, as we illustrate in the Supplementary Material. But note that only the differences $u - \mu_1$ and $v - \mu_2$ are relevant. Consequently we can work in any shifted pixel coordinate system. We then propose to work in local tile coordinates, and perform the calculation of the $z_i$ while we are collaboratively loading the 2D Gaussian's attributes $\boldsymbol{\mu}$, $\boldsymbol{\Sigma}^{-1}$ and $\tau$ into a block's shared memory.

### D. Tile size

We found a larger tile size of $16\,\text{px} \times 32\,\text{px}$ to be more efficient at higher resolutions, compared to the reference $16\,\text{px} \times 16\,\text{px}$. We then propose to use the latter when the number of pixels to render is below 1.5 million, and the former otherwise. This threshold was determined experimentally. Note that our implementation uses column-major indexing, explaining why $16 \times 32$ will outperform $32 \times 16$. For row-major indexing this will be reversed.

The ideal threshold and tile size will likely depend on hardware architecture, but in any case we advise to not just blindly stick with $16\,\text{px} \times 16\,\text{px}$ in all situations.

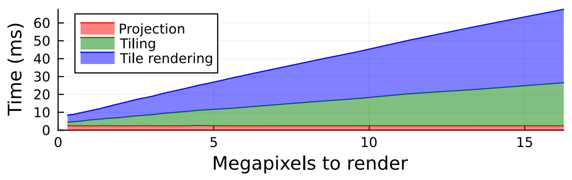

Fig. 2. Timings for the three 3DGS image rendering phases, as a function of the number of pixels to render. This uses our *Baseline* implementation to render the first image of the *Bicycle* scene at various resolutions.

## IV. Experiments and discussion

We implemented the rendering part of 3DGS, as well as our modifications, in the Julia programming language, making extensive use of CUDA.jl [10], [11]. We then compare the rendering times of the training images from the *Bicycle* and *Garden* scenes from the Mip-NeRF 360 dataset [12], as well as of those of the *Train* scene from Tanks and Temples [13], undistorted via COLMAP [14], [15], and the *Playroom* scene from Deep Blending [16]. We render at native resolution, except for the Mip-NeRF 360 scenes, where we additionally render at quarter resolution on each axis. We use the official pretrained models from 3DGS. These were trained on the low-resolution versions of *Bicycle* and *Garden*, on *Train* downscaled to half resolution on each axis, and on native resolution for *Playroom*. By timing the different implementations on the training image, we ensure that the models are capable of well-reconstructing the scene. We loop through the datasets 10 times for more accurate results. These are presented in Table I. We used an Intel i9-10900X CPU and an NVIDIA 3080 Ti GPU, in a machine running Ubuntu 22.04.

The original 3DGS method (in contrast to Mip-Splatting [6]) is intended to be used at the same resolution at training and inference time, explaining the worse quality at higher rendering resolutions. In general, the results from the reference 3DGS implementation in PyTorch and CUDA are included only for completeness, but should not directly be compared to, as it, in contrast to our versions, also stores relevant intermediate outputs required later for the backward pass. This also partially explains why we require less VRAM, though likely there is also some overhead from PyTorch involved.

The *Baseline* row in the table refers to our reimplementation in Julia, without any of our proposed modifications. It seems to subtly differ from the reference version. As can be seen from the included PSNR values at 8-bit precision, this does not come at the cost of rendering quality. In the Supplementary Material we provide a qualitative comparison. Figure 2 shows how the rendering time is distributed at different rendering resolutions for this baseline. It will be useful for analyzing the performance gains for the different improvements we proposed in Section III. The results for these improvements can be found in the next four rows of the table. Finally, *Ours (full)* is the implementation where we include all four improvements.

As the filtering changes occur in the projection phase, at which point there is no notion of rendering resolution (see also the red area in Fig.2), the speed-up for this row is mostly independent of resolution, instead being affected by the number of 3D Gaussians in the trained model. As a

TABLE I
RENDERING SPEED (SCENE AVERAGE $\pm$ STANDARD DEVIATION), AS WELL AS AVERAGE PSNR QUALITY AND PEAK VRAM USAGE FOR THE DIFFERENT COMPARED 3DGS RENDERING IMPLEMENTATIONS

| | **Bicycle** 4946 x 3286 | **Bicycle** 1237 x 822 | **Garden** 5187 x 3361 | **Garden** 1297 x 840 | **Train** 1959 x 1090 | **Playroom** 1264 x 832 |
|---|---|---|---|---|---|---|
| **Reference 3DGS (PyTorch + CUDA, [2])** | 60.3 $\pm$ 15.3 ms (20.3 dB; 8.5 GiB) | 15.3 $\pm$ 1.25 ms (25.7 dB; 4.5 GiB) | 48.4 $\pm$ 3.21 ms (20.6 dB; 7.1 GiB) | 14.0 $\pm$ 0.513 ms (29.4 dB; 4.3 GiB) | 11.0 $\pm$ 1.15 ms (23.6 dB; 2.1 GiB) | 8.19 $\pm$ 1.03 ms (35.0 dB; 2.5 GiB) |
| **Baseline (Julia, ours)** | 52.0 $\pm$ 14.1 ms (20.4 dB; 5.9 GiB) | 9.11 $\pm$ 1.25 ms (25.7 dB; 2.4 GiB) | 41.2 $\pm$ 3.24 ms (20.7 dB; 4.4 GiB) | 8.28 $\pm$ 0.533 ms (29.7 dB; 2.3 GiB) | 8.62 $\pm$ 1.17 ms (23.7 dB; 1.3 GiB) | 4.79 $\pm$ 1.02 ms (35.6 dB; 1.3 GiB) |
| **Baseline + filtering** | 51.2 $\pm$ 14.4 ms (20.3 dB; 5.9 GiB) | 8.31 $\pm$ 1.27 ms (25.7 dB; 2.5 GiB) | 40.6 $\pm$ 3.29 ms (20.7 dB; 4.4 GiB) | 7.64 $\pm$ 0.695 ms (29.6 dB; 2.3 GiB) | 8.12 $\pm$ 1.21 ms (23.0 dB; 1.3 GiB) | 4.49 $\pm$ 0.981 ms (35.4 dB; 1.3 GiB) |
| **Baseline + precomputation** | 49.7 $\pm$ 13.3 ms (20.4 dB; 5.9 GiB) | 8.81 $\pm$ 1.21 ms (25.7 dB; 2.4 GiB) | 39.5 $\pm$ 3.09 ms (20.7 dB; 4.4 GiB) | 8.03 $\pm$ 0.536 ms (29.7 dB; 2.3 GiB) | 8.31 $\pm$ 1.14 ms (23.7 dB; 1.3 GiB) | 4.63 $\pm$ 0.991 ms (35.6 dB; 1.3 GiB) |
| **Baseline + tile size** | 46.4 $\pm$ 12.0 ms (20.4 dB; 4.2 GiB) | 9.12 $\pm$ 1.27 ms (25.7 dB; 2.4 GiB) | 36.1 $\pm$ 2.54 ms (20.7 dB; 3.4 GiB) | 8.26 $\pm$ 0.525 ms (29.7 dB; 2.3 GiB) | 7.43 $\pm$ 0.993 ms (23.7 dB; 1.0 GiB) | 4.80 $\pm$ 1.02 ms (35.6 dB; 1.3 GiB) |
| **Baseline + tile intersections** | 15.3 $\pm$ 1.85 ms (20.4 dB; 3.0 GiB) | 5.74 $\pm$ 0.628 ms (25.7 dB; 2.2 GiB) | 16.2 $\pm$ 0.705 ms (20.7 dB; 2.8 GiB) | 6.13 $\pm$ 0.437 ms (29.7 dB; 2.1 GiB) | 3.45 $\pm$ 0.439 ms (23.7 dB; 0.82 GiB) | 2.63 $\pm$ 0.648 ms (35.6 dB; 1.2 GiB) |
| **Ours (full)** | 13.8 $\pm$ 1.87 ms (20.3 dB; 2.8 GiB) | 4.71 $\pm$ 0.690 ms (25.7 dB; 2.3 GiB) | 14.6 $\pm$ 0.719 ms (20.7 dB; 2.7 GiB) | 5.35 $\pm$ 0.517 ms (29.6 dB; 2.2 GiB) | 3.17 $\pm$ 0.738 ms (23.0 dB; 0.79 GiB) | 2.23 $\pm$ 0.726 ms (35.4 dB; 1.2 GiB) |

consequence, the relative improvement is greatest at lower resolutions. The quality of the renders is occasionally slightly reduced. But this is to be expected, as the models were trained for the purely depth-filtering reference implementation. The other modifications have no effect on the image quality.

The performance gains from the precomputation are greatest at higher rendering resolutions, as there are more duplicated Gaussians and hence $\alpha$-computations. On the low-resolution *Bicycle* scene, using $16 \times 32$ tiles increases the average render time to 9.77 ms, illustrating why we only increase the tile size at higher resolutions.

Our modification to the tile intersections makes a massive difference, especially at higher resolutions, since we need to sort and render many fewer duplicated Gaussians, as we will quantify below. Also note that the render times are much more consistent, in particular on *Bicycle* at native resolution. This can be explained by the presence of large, mostly transparent 3D Gaussians. In the original approach these will affect a significant number of tiles for certain views, leading to higher render times. But as we take into account the low opacity, our approach is much more effective at (not) rendering these.

Finally, combining all modifications decreases the rendering times further. Note that the tile intersection and filtering improvements synergize, as we can also immediately filter out Gaussians whose opacity $\tau$ falls below the $\alpha$-threshold of $\epsilon$, as in this case the axis-aligned bounding box is empty. In total, on the *Bicycle* scene at native resolution we decrease the render time and GPU memory usage by factors of 3.77 and 2.11.

## A. *Number of duplicated Gaussians*

To further quantify why our new axis-aligned bounding boxes work much better than the original ones, and to investigate whether the largest gains are due to our new conversion from a 2D Gaussian to an elliptical disk, or due to allowing non-square bounding boxes, we consider the total number of duplicated Gaussians in the first image of *Bicycle* (`_DSC8679.JPG`) at native resolution for $16 \times 16$ tiles. For the original bounding boxes we had 87 067 709 duplicated Gaussians. Using our opacity-sensitive ellipse, but keeping the bounding box square, this reduces to 55 648 264. Allowing for non-square boxes, it further drops to 18 466 556. Finally, if we would have pixel-perfect tile-Gaussian intersections, i.e., no longer approximate a 2D Gaussian by an axis-aligned bounding box, we would only need 10 976 269. Finding these exact intersections is computationally expensive, however. Our approach then provides a good trade-off between the approximation quality of the Gaussian-tile intersections, and the time needed to compute each of these, with a low total rendering time as a result.

Increasing the tile size to $16 \times 32$ of course also decreases the number of duplicated Gaussians, but now this is not a guaranteed speed-up. As the tile grid becomes coarser, we will assign a Gaussian to more pixels, leading to a reduction of duplicated Gaussians by only less than half (using the original tile intersections: from 87 067 709 to 48 012 778 at high resolution; from 10 207 165 to 6 988 817 at low resolution). Furthermore, each tile then needs to (sequentially) process more Gaussians. Additionally, the relation between thread-block size and GPU performance is complicated: increasing the block size certainly does not automatically improve performance.

## V. FURTHER WORK

Currently, our tile size modification uses a somewhat arbitrary threshold. We intend to further investigate the relation between tile size, rendering resolution and rendering time, to hopefully obtain a more principled formula.

We will also reimplement 3DGS training, as we believe our improvements will lead to similar performance gains. We expect this to also eliminate any quality degradation arising from our modifications, as the new models will be optimized for the new setting. When this implementation is finished, we intend to make it open-source.

## References

[1] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "NeRF: representing scenes as neural radiance fields for view synthesis," *Communications of the ACM*, vol. 65, no. 1, pp. 99–106, 2021.

[2] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis, "3D Gaussian splatting for real-time radiance field rendering," *ACM Trans. Graph.*, vol. 42, no. 4, pp. 139–1, 2023.

[3] S. Fridovich-Keil, A. Yu, M. Tancik, Q. Chen, B. Recht, and A. Kanazawa, "Plenoxels: radiance fields without neural networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 5501–5510.

[4] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant neural graphics primitives with a multiresolution hash encoding," *ACM transactions on graphics (TOG)*, vol. 41, no. 4, pp. 1–15, 2022.

[5] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman, "Zip-NeRF: anti-aliased grid-based neural radiance fields," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2023, pp. 19 697–19 705.

[6] Z. Yu, A. Chen, B. Huang, T. Sattler, and A. Geiger, "Mip-Splatting: alias-free 3D Gaussian Splatting," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 19 447–19 456.

[7] Z. Yang, X. Gao, W. Zhou, S. Jiao, Y. Zhang, and X. Jin, "Deformable 3D Gaussians for high-fidelity monocular dynamic scene reconstruction," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 20 331–20 341.

[8] H. Matsuki, R. Murai, P. H. Kelly, and A. J. Davison, "Gaussian Splatting SLAM," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 18 039–18 048.

[9] P. Papantonakis, G. Kopanas, B. Kerbl, A. Lanvin, and G. Drettakis, "Reducing the memory footprint of 3D Gaussian Splatting," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 7, no. 1, pp. 1–17, 2024.

[10] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[11] T. Besard, V. Churavy, A. Edelman, and B. De Sutter, "Rapid software prototyping for heterogeneous and distributed platforms," *Advances in Engineering Software*, vol. 132, pp. 29–46, 2019.

[12] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, and P. Hedman, "Mip-NeRF 360: unbounded anti-aliased neural radiance fields," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 5470–5479.

[13] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun, "Tanks and Temples: benchmarking large-scale scene reconstruction," *ACM Transactions on Graphics (ToG)*, vol. 36, no. 4, pp. 1–13, 2017.

[14] J. L. Schönberger and J.-M. Frahm, "Structure-from-motion revisited," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[15] J. L. Schönberger, E. Zheng, M. Pollefeys, and J.-M. Frahm, "Pixelwise view selection for unstructured multi-view stereo," in *European Conference on Computer Vision (ECCV)*, 2016.

[16] P. Hedman, J. Philip, T. Price, J.-M. Frahm, G. Drettakis, and G. Brostow, "Deep blending for free-viewpoint image-based rendering," *ACM Transactions on Graphics (ToG)*, vol. 37, no. 6, pp. 1–15, 2018.