Novel View Synthesis

Joe Lin   Michael Song   Alexander Chien

February 23, 2024

Intro + Problem Statement

Problem: I'm bored and I want to re-create 3D scenes from 2D images.

Solution: **Novel View Synthesis**, which aims to generate new scene perspectives given images at known viewpoints.

Useful applications in:

- Computational photography
- Virtual reality

Classical Approach

Light Fields are 4D representation of radiance as a function of light ray position and direction.[1]

- **Light Slabs** ($L(u, v, s, t)$) used to represent the light field.
- Less computational heavy and mappings from (x, y) -> (u, v, s, t) is a projection map.
- Create light field from views by mapping (x, y) to (u, v, s, t).
- Views are essentially 2D slices of the 4D light field and to get novel views we can extract and resample slices of the light field (inverse mapping from (u, v, s, t) to (x, y)).

but wait…

I don't have enough photos from different perspectives 😭. This classical approach isn't for me! 💩 🚽

Classical Approach Challenges

- Sampling density must be high to avoid bluriness (i.e. many ground truth views are required).
- Heavy computation and memory usage.
- Observer is restricted to regions of space free of occluders.
- Illumination must be fixed.

Solution: Deep Learning approaches… **neural networks** learn mapping between sparse input views and high quality 3D scene representations.

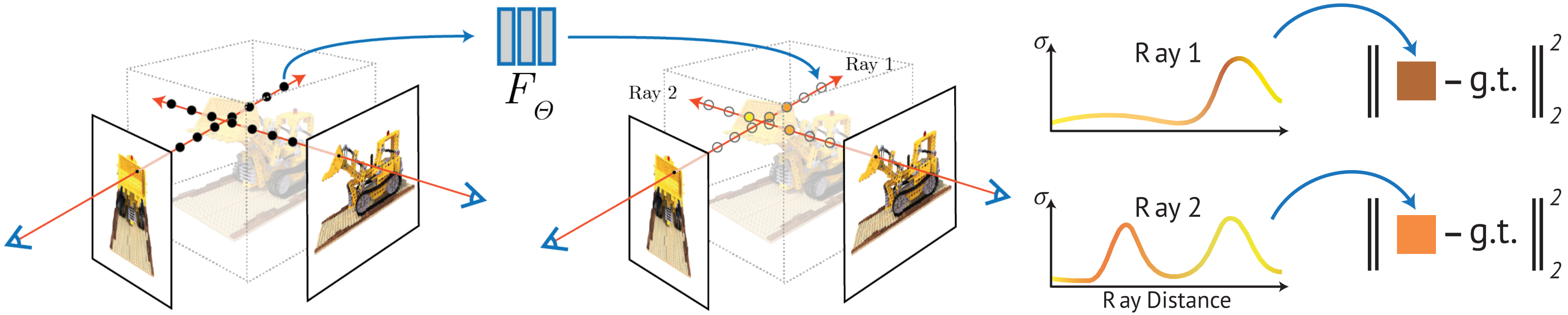
Neural Radiance Fields (NeRFs)[2]

Algorithm Overview

Goal is to optimize a continuous **volumetric scene function** F_θ where:

- input is a spatial location (x, y, z) and viewing direction (θ, φ)
- output is volume density σ and view-dependent emitted radiance

Basic NeRFs essentially encode 3D scene with a neural network.



Analysis of Approach

Benefits:

- Capable of representing complex real-world geometry.
- Differentiable $\Rightarrow$ can be optimized with gradient-based methods.

Analysis of Approach

Benefits:

- Capable of representing complex real-world geometry.
- Differentiable $\Rightarrow$ can be optimized with gradient-based methods.

but wait...

Analysis of Approach

Benefits:

- Capable of representing complex real-world geometry.
- Differentiable $\Rightarrow$ can be optimized with gradient-based methods.

but wait...

I don't have enough computational power! I also want a good rendering in less than a day... 🥴

Analysis of Approach (cont.)

Problems:

- Computationally expensive. Single scene can take days to train.
- Produces low FPS renders.
- Implicit representation (hard to interpret and edit scene).

Solution: 3D Gaussian Splatting

3D Gaussian Splatting[3]

Computer Graphics

Rasterization is the process of drawing graphical data onto the screen.

Objects are oftentimes represented by its polygonal faces (commonly triangles). Each of these polygons are decomposed into pixels and rasterized into a **raster image**.

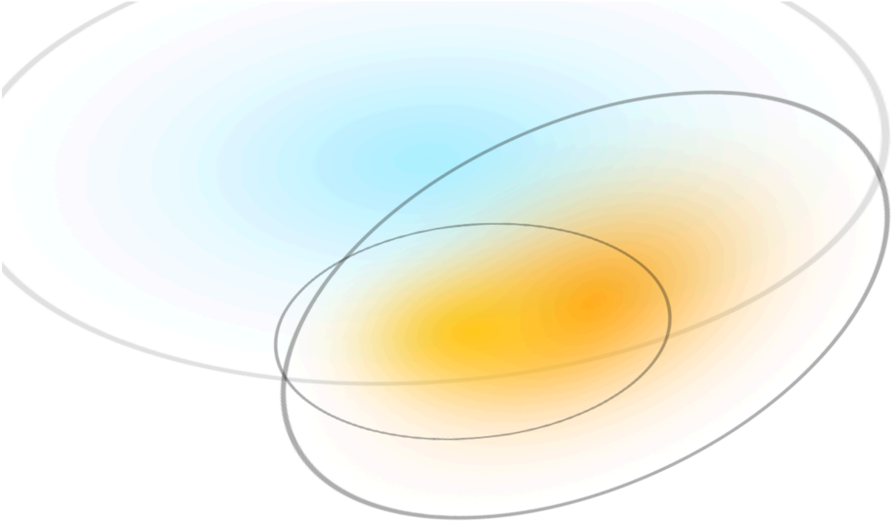
We can think of Gaussian Splatting as a rasterization technique.

Rasterizing Gaussians

$$G(x) = e^{\left(-\frac{1}{2}\right)(x-\mu)^T \Sigma^{-1} (x-\mu)}$$

Essential parameters:

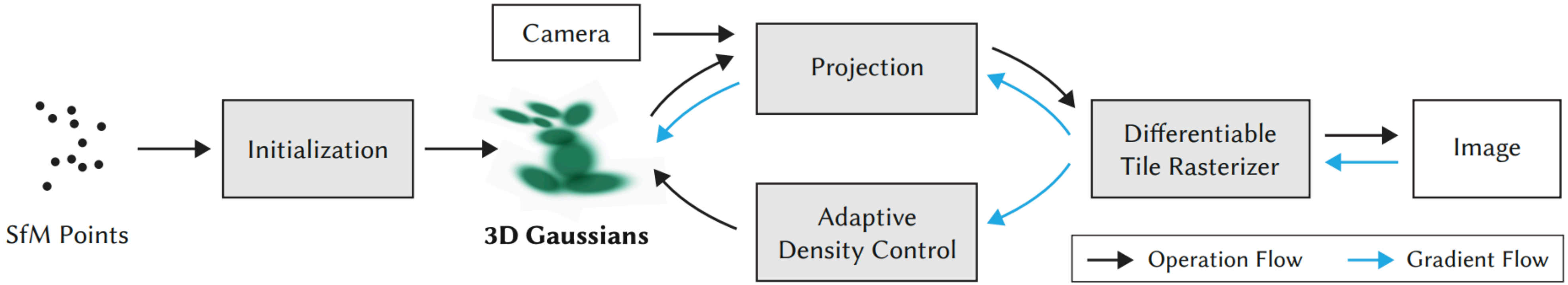
- position – mean, μ
- splat size – covariance matrix, Σ
- color – spherical harmonics
- opacity – α



Algorithm Overview

1. Given a set of images, estimate a pointcloud with **Structure from Motion (SfM)**.
2. Initialize isotropic Gaussians at each point.
3. Train 3D scene representation.
   a. Rasterize Gaussians with a **differentiable rasterizer**.
   b. Compute loss between raster image and ground truth image.
   c. Adjust Gaussian parameters with SGD.
   d. Densify or prune Gaussians.

Algorithm Overview



Differentiable Rasterizer

1. Project each Gaussian into 2D camera perspective.
    a. Covariances in camera coordinates are $\Sigma' = JW\Sigma W^T J^T$
        i. J is the Jacobian of the affine approximation to the projective transform[4]
2. Sort Gaussians by order of depth.
3. Perform front-to-back blending of Gaussians at each pixel.

Optimizing Gaussians

- Mean parameter μ
- Covariance parameter Σ
  - *Initial Thought*: Optimize Σ directly.
    - Problem: Doesn't guarantee Σ remains positive semi-definite.
  - *What can we do instead?*
    - Gaussians can be seen as an **ellipsoid** rotated and stretched.
    - Decompose into $\Sigma = RSS^T R^T$ and optimize R and S.[5]
- Spherical Harmonics coefficients c
- Opacity parameter α

Quaternions

Traditionally, rotations are represented with Euler angles (3 axis angles). In practice, rotation matrix R is encoded with **quaternions**.

$$\mathbb{H} = \{s + xi + yj + zk \mid s, x, y, z \in \mathbb{R}\}$$

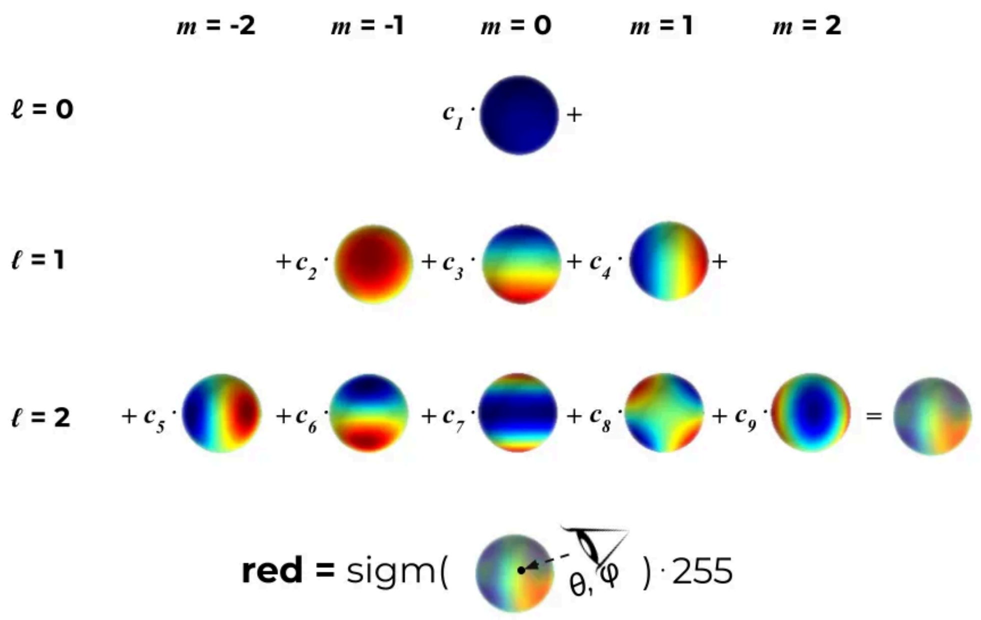
where i, j, k obeys $i^2 = j^2 = k^2 = ijk = -1, ij = k, ji = -k$

Quaternions prevent **gimbal lock**.

Spherical Harmonics

Why not just RGB? Problem: No view dependence.

Spherical harmonics encode view-dependent colors as a linear combination of a basis of **harmonic functions.**[5]



Optimization Details

Use **Stochastic Gradient Descent** algorithm.

Activation Functions

- Opacity parameters (α): $\sigma(x) = \frac{1}{1+e^{-x}}$
  - smooth gradients
  - keep $\alpha \in [0, 1)$
- Scale parameters (s): $\exp(x)$
  - similar reasons as above

Initialization of Gaussians: Estimate initial covariance matrix as an **isotropic** Gaussian with axes equal to the mean of the distances to the 3-NNs (recall the K-NN)

Exponential decay scheduling for positions.

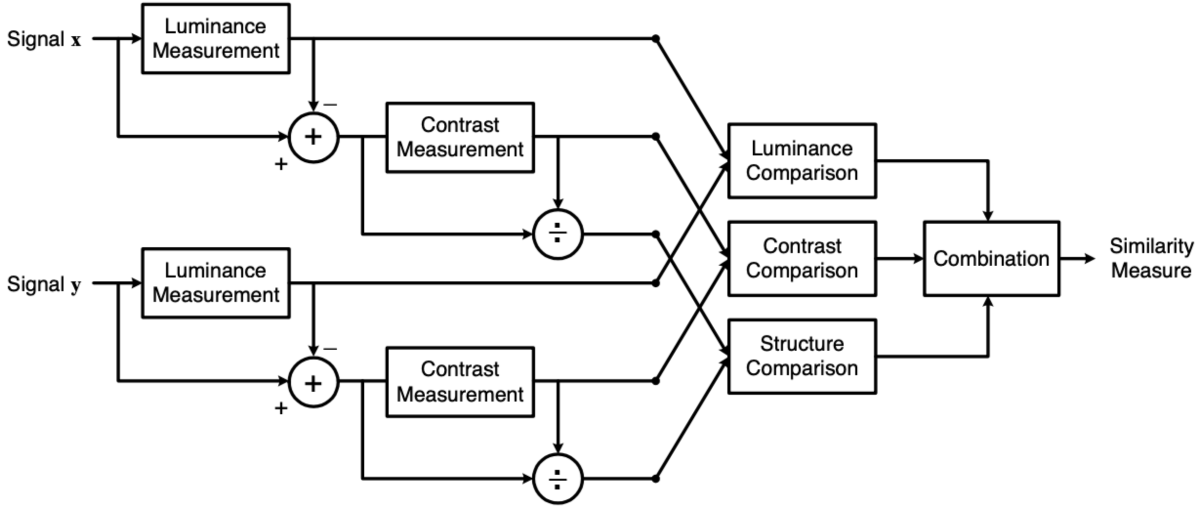
Loss Function

$$\mathcal{L} = (1 - \lambda)\mathcal{L}_1 + \lambda\mathcal{L}_{\text{D-SSIM}}$$

- Comparing predicted and ground truth intensities.
- $\lambda = 0.2$
- **L1 loss** ($\mathcal{L}_1$): Mean Absolute Error
- D-SSIM ($\mathcal{L}_{\text{D-SSIM}}$) = 1 - SSIM (**Structural Similarity Index Metric**)[6]
  - SSIM compares luminance, contrast, and structure

Optimization Details (cont.)

$$\mathrm{SSIM}(x,y) = \frac{\left(2\mu_x\mu_y + C_1\right)\left(2\sigma_{xy} + C_2\right)}{\left(\mu_x^2 + \mu_y^2 + C_1\right)\left(\sigma_x^2 + \sigma_y^2 + C_2\right)}$$ [7]



Controlling Gaussians

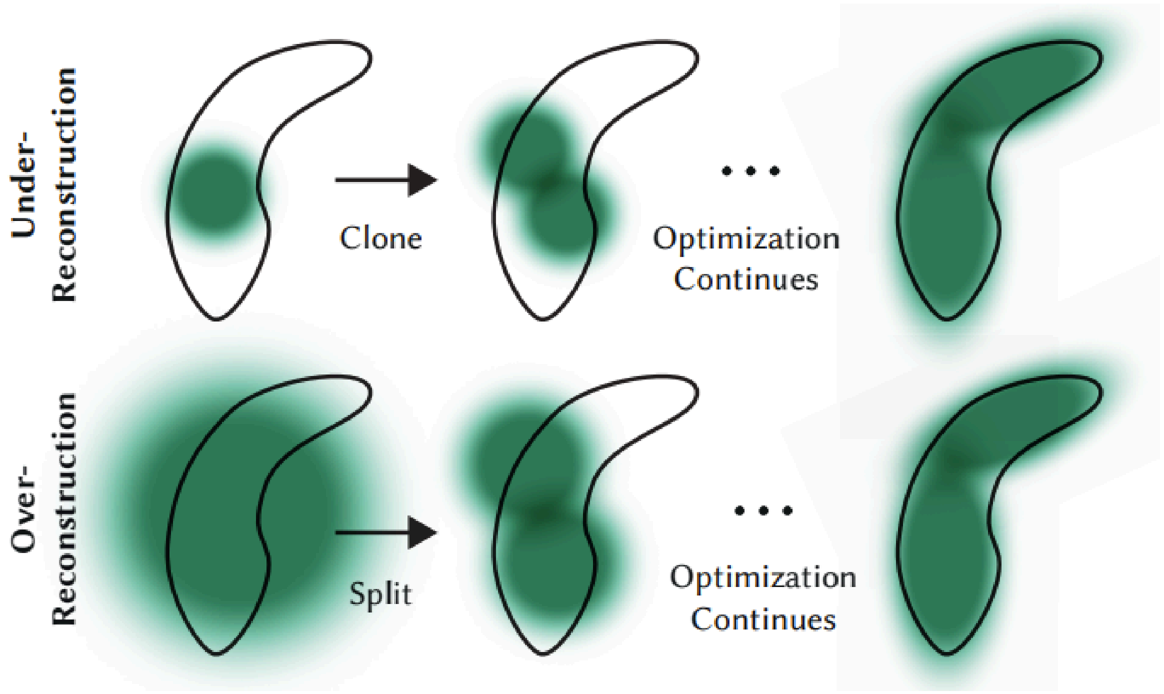
Problem: Areas with missing geometric features (**under-reconstruction**; no Gaussians 😫) and regions where Gaussians cover large areas in the scene (**over-reconstruction**): moral of the story is area not well-reconstructed.

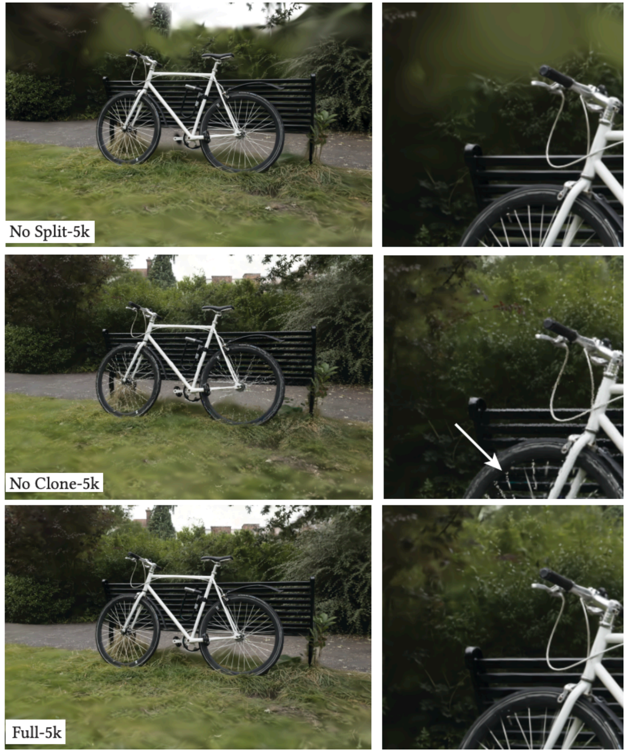
- Both scenarios result in large view-space **positional gradients**.

Solution: **Densification**

1. Small Gaussians in under-constructed regions: clone and move clone in direction of positional gradient. (Increase total volume of system and Gaussians)
2. Large Gaussians in regions of high variance: replace Gaussian with 2 new ones and divide scale by factor of $\varphi = 1.6$ (value determined experimentally)
   a. Initialize position by using original Gaussian as a probability density function for sampling.
   b. Conserve total volume of system but increase number of Gaussians.

Controlling Gaussians (cont.)





Controlling Gaussians (cont.)

Another Issue Arises: Floaters close to input cameras cause unjustified increase in Gaussian density.

Solution: Set α close to 0 every 3000 iterations.

Optimization increases α for Gaussians where this is needed, while culling Gaussians with an alpha less than ε_α.

Miscellaneous:

- Remove Gaussians large in world-space and those with a large footprint in view-space
- Gaussians remain primitives in Euclidean space
  - Don't require space compaction, warping, or projection strategies for distant and large Gaussians.

Controlling Gaussians (cont.)

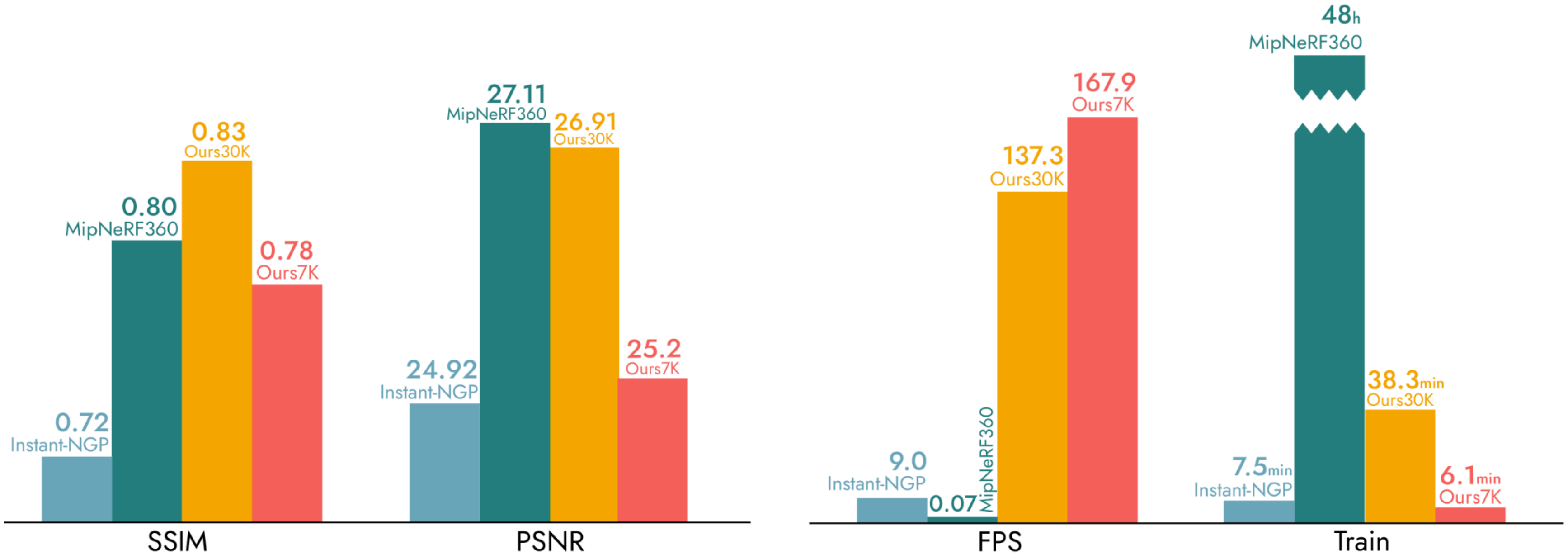
if IsRefinementIteration(i) **then**
    for all Gaussians (μ, Σ, c, α) **in** (M, S, C, A) **do**
        if $\alpha < \epsilon$ or IsTooLarge(μ, Σ) **then** ▷ Pruning
            RemoveGaussian()
        end if
        if $\nabla_p L > \tau_p$ **then** ▷ Densification
            if $\|S\| > \tau_S$ **then** ▷ Over-reconstruction
                SplitGaussian(μ, Σ, c, α)
            else ▷ Under-reconstruction
                CloneGaussian(μ, Σ, c, α)
            end if
        end if
    end for
end if

Fast Differentiable Rasterizz

1. Split into 16×16 tiles.
2. Cull 3D Gaussians with **frustum culling**.
3. Initialize Gaussians and give each a key.
4. Fast **radix sort** based on key (roughly $O(n)$ algorithm).
5. Parallelize **alpha blending** per tile (one thread block per tile).
6. Stops thread block for tile when pixels reach certain alpha level.
7. Back-to-front traversal to compute gradients.

Analysis of Approach

Performance Metrics



Analysis of Approach (cont.)

Problems:

- Artifacts still exist (remember floaters?); especially elongated ones
  - Popping artifacts may arise due to trivial culling of rasterizz
- Significantly more memory than methods like NeRF

Current/Future Works:

- Few shot 3D Gaussian Splatting
- Mesh reconstruction with Gaussians
- Reducing memory consumption

BUT WAIT…

Ha. It's over. We don't have any more to offer… for the time being. 😉

Bibliography

1. Levoy M, Hanrahan P. Light Field Rendering. In: *Seminal Graphics Papers: Pushing the Boundaries, Volume 2.* 1st ed. Association for Computing Machinery; 2023. https://doi.org/10.1145/3596711.3596759

2. Mildenhall B, Srinivasan P P, Tancik M, Barron J T, Ramamoorthi R, Ng R. NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis. In: *ECCV.* ; 2020.

3. Kerbl B, Kopanas G, Leimkühler T, Drettakis G. 3D Gaussian Splatting for Real-Time Radiance Field Rendering. *ACM Transactions on Graphics.* 2023;42(4). https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

4. Zwicker M, Pfister H, Baar J van, Gross M. EWA volume splatting. In: *Proceedings Visualization, 2001. VIS '01.* ; 2001:29-538. doi:10.1109/VISUAL.2001.964490

5. Yurkova K. A Comprehensive Overview of Gaussian Splatting. Published online December 2023. https://towardsdatascience.com/a-comprehensive-overview-of-gaussian-splatting-e7d570081362

6. Wang Z, Bovik A, Sheikh H, Simoncelli E. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*. 2004;13(4):600-612. doi:10.1109/TIP.2003.819861

7. Datta P. All about Structural Similarity Index (SSIM): Theory + Code in PyTorch. Published online September 2020. https://medium.com/srm-mic/all-about-structural-similarity-index-ssim-theory-code-in-pytorch-6551b455541e