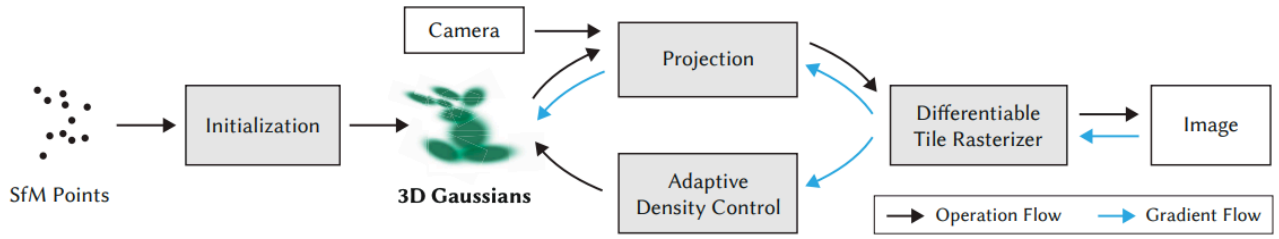
#cs-101 #radiance-fields #3d-deep-learning



Why are NeRFs bad? Stochastic sampling

- is costly
- results in noise

Why is GS good?

- **Random initialization of Gaussians yields high quality** immediately
  - Pipeline: same inputs as NeRF (images -> SfM), initialize Gaussians at each p_i from SfM
- 3-D Gaussians are a good representation
  - Differentiable
    - Easy to optimize properties
  - Easy to rasterize with 2-D projection + alpha compositing, as in NeRF
  - About 1-5 million sufficient
- Easy rendering on GPU
  - Anistropic splatting is easy because of sorting and alpha blending
    - The point clouds are anistropic = points are not spheres, they're Gaussians that can be stretched into ellipsoids

Representing Gaussians for differentiability

Define a Gaussian with covariance Σ in world space centered at μ as

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

Ideally, just optimize Σ to get 3-D Gaussians representing the RF. But this might break positive semi-definiteness and create invalid covariance matrices.

But we know the covariance matrix is symmetric! So it admits a spectral decomposition

$$\Sigma = Q\Lambda Q^T$$

Where

- The orthogonal matrix Q's columns are the eigenvectors of the covariance matrix, which represent the principal axes of the 3-D ellipsoid
- Λ is a diagonal scaling matrix

For physical validity, we define scaling and rotation matrices S and R which we can optimize

$$\Sigma = RSS^TR^T$$

- We want Λ to only have positive scaling values, so take the square of the values in the scaling matrix S
- Rotation matrix R = Q
  Store R as a (unit) quaternion and S as a 3-D scaling vector $S = ()$

The paper explicitly derives gradients of and with respect to the projection (as defined below)

Projecting Gaussians to 2-D space

The covariance matrix in camera coordinates (2-D space) will be

$$\Sigma = \Sigma^{TT}$$

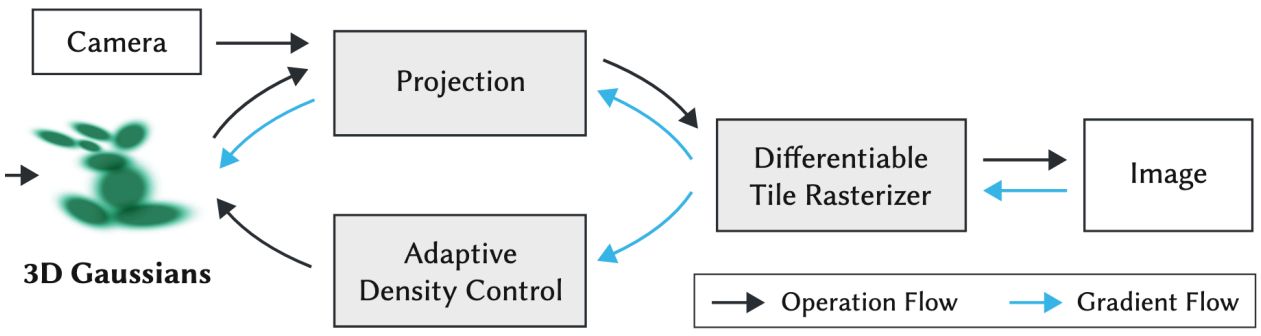
1. If we apply a transformation to x to get x the new covariance matrix is Σ^T: covariance matrix of a random variable under linear transformations
2. Still in 3-D space, project it to 2-D coordinates. So apply the Jacobian (2 x 3 matrix) of the affine (=local linear) approximation of the projective transformation
   Equivalent to $()\Sigma()^T = \Sigma^{TT}$

Optimizing 3-D Gaussians

Apply differentiation of existing Gaussian parameters + adaptive control

For each Gaussian, we ned to optimize

1. Σ (as represented by and)
2. Position p
3. Opacity
4. Spherical harmonic coefficients for the color of each Gaussian



Implementation details

- , R need to be physical
  - Set = sigmoid of learnable value and cap it to $1)$
  - Exponentiate the scale to keep positive scales
  - Normalize for rotations
- Exponential decay scheduling for learning rate of p
- Combined L1 and reconstruction loss

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

Adaptive control

SfM gives us a sparse set of Gaussians, we need to densify. So after initial warmup, every 100 iterations:

1. Densify
2. Remove Gaussians with a very low

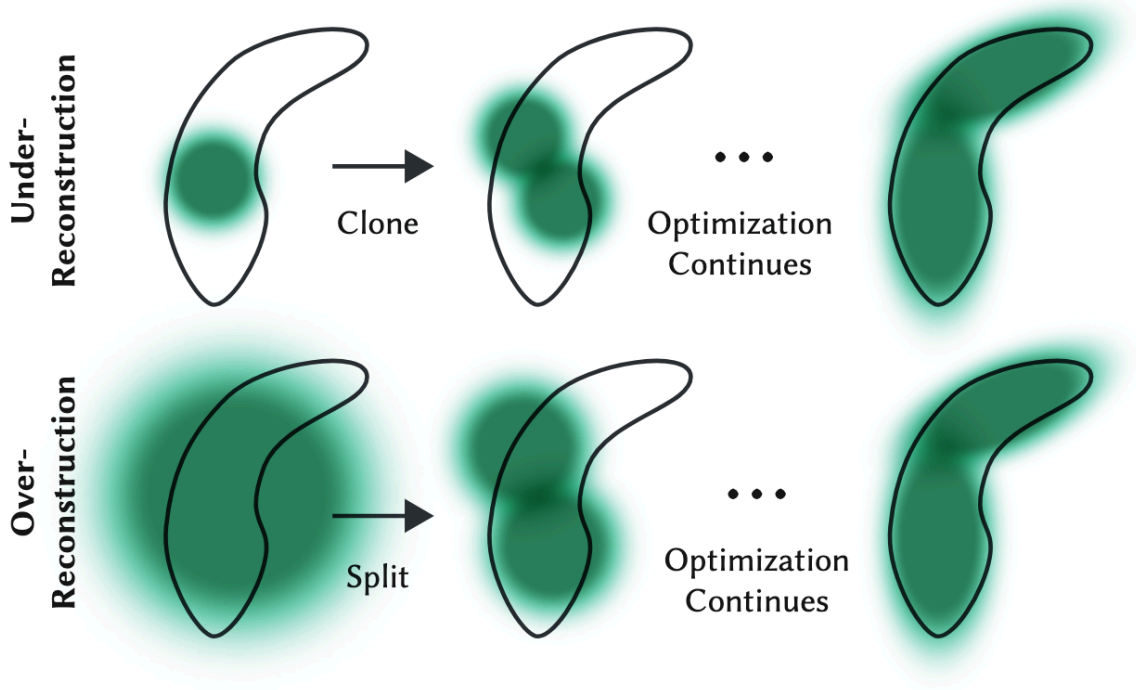
How does densification work?
Large gradients (bad reconstructions) emerge in view space in 2 kinds of regions:

1. Under-reconstruction (missing features)
2. Over-reconstruction (Gaussian covers a large area)

So:

1. Create a copy of Gaussians in under-reconstructed regions and move it in the direction of positional gradient
2. Densify Gaussians which have an average gradient above some threshold $= 2$
   - Split into 2 new Gaussians
     - Sample positions from PDF

- Reduce scale factor by $= 1$



Hacks for experimental issues:

- Gaussians close to camera get stuck (results in a spurious increase in Gaussian density)
  - set near 0 every 3000 iterations, important Gaussians will re-gain gradients, unimportant ones will get culled
- Periodically remove Gaussians very large in worldspace that have a big footprint in view space

For stability: Start in 4x smaller image resolution, upsample twice after 250 and 500 iterations

Color with SH:

- Color is view-dependent - can't just store RGB scalars
- Store a set of Spherical harmonic coefficients per color channel
  - To get the RGB pixel given the viewing angle, just take the linear combination of harmonic basic functions for that angle with learned coefficients
- 0th order = sphere represent diffuse color, higher order represent light wrt to surface
- Doesn't make sense to try to learn higher order before lower order. So introduce one band of SH every 1000 iterations until all bands are represented

Differentiable rasterization

Split the screen into 16x16 tiles

- Cull 3-D gaussians against view frustum and each tile
- Reject Gaussians close to near place + far outside frustum

For a given Gaussian, 'instantiate' it for each tile

- Confusing term but used in GPU stuff

We have an unordered set of Gaussians, now assign each Gaussian a key so we can use radix sort
Key=

- Tile ID for that Gaussian (high bits)
- View space depth (low bits)

After sorting, we get a list for each tile by getting the first and last depth sorted entry in that tile

Launch one thread block per tile for rasterization

- Load packets of Gaussians into shared memory
- For each pixel, accumulate color and by traversing the list
  - Stop the thread when we reach a target saturation from the pixel
  - Stop the entire thread when all pixels have saturated

Algorithm 2 GPU software rasterization of 3D Gaussians
w, h: width and height of the image to rasterize
M, S: Gaussian means and covariances in world space
C, A: Gaussian colors and opacities
V: view configuration of current camera

```
function RASTERIZE(w, h, M, S, C, A, V)
    CullGaussian(p, V)                          ▷ Frustum Culling
    M', S' ← ScreenspaceGaussians(M, S, V)      ▷ Transform
    T ← CreateTiles(w, h)
    L, K ← DuplicateWithKeys(M', T)             ▷ Indices and Keys
    SortByKeys(K, L)                            ▷ Globally Sort
    R ← IdentifyTileRanges(T, K)
    I ← 0                                       ▷ Init Canvas
    for all Tiles t in I do
        for all Pixels i in t do
            r ← GetTileRange(R, t)
            I[i] ← BlendInOrder(i, L, r, K, M', S', C, A)
        end for
    end for
     return I
end function
```

Gradients:

- Traverse the lists back to front
  - Start from last point that affected any pixel

    > Each pixel will only start overlap testing and processing of points if their depth is lower than or equal to the depth of the last point that contributed to its color during the forward pass.

Numerical stability. During the backward pass, we reconstruct the intermediate opacity values needed for gradient computation by repeatedly dividing the accumulated opacity from the forward pass by each Gaussian's α. Implemented naïvely, this process is prone to numerical instabilities (e.g., division by 0). To address this, both in the forward and backward pass, we skip any blending updates with $\alpha < \epsilon$ (we choose ϵ as $\frac{1}{255}$) and also clamp α with 0.99 from above. Finally, **before** a Gaussian is included in the forward rasterization pass, we compute the accumulated opacity if we were to include it and stop front-to-back blending **before** it can exceed 0.9999.

-