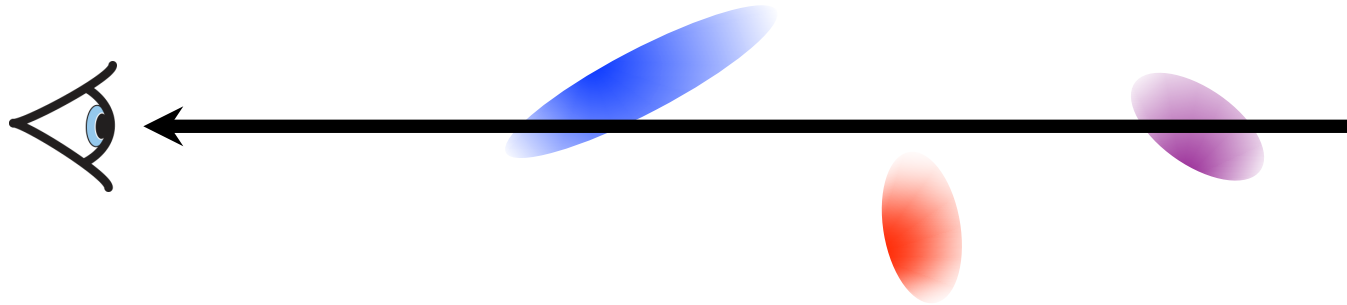


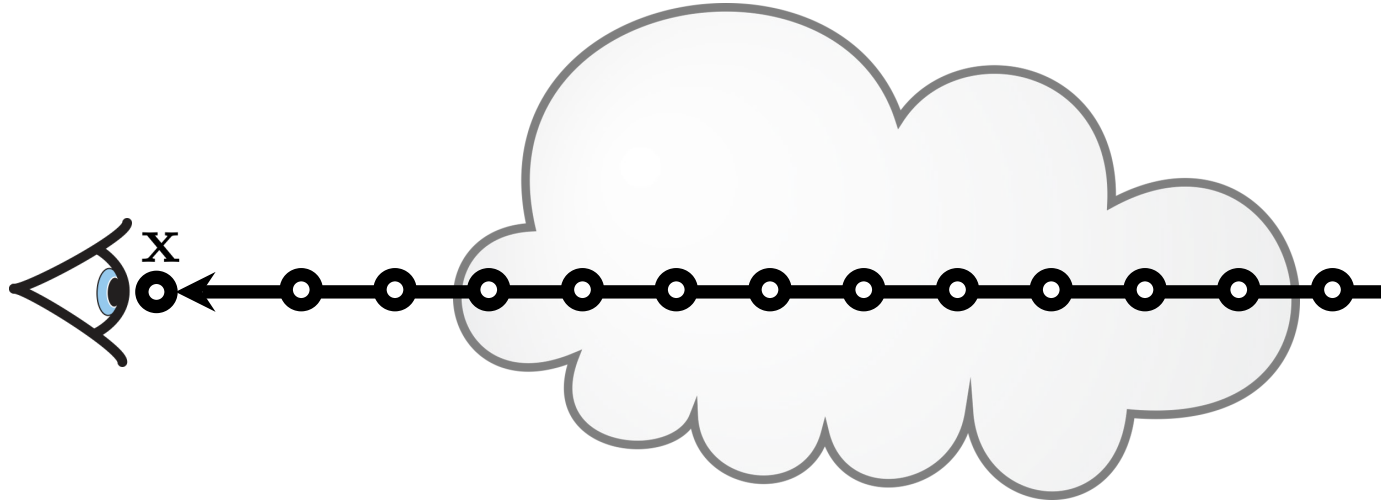
Neural rendering (continued)



Differentiable Primitive Rendering



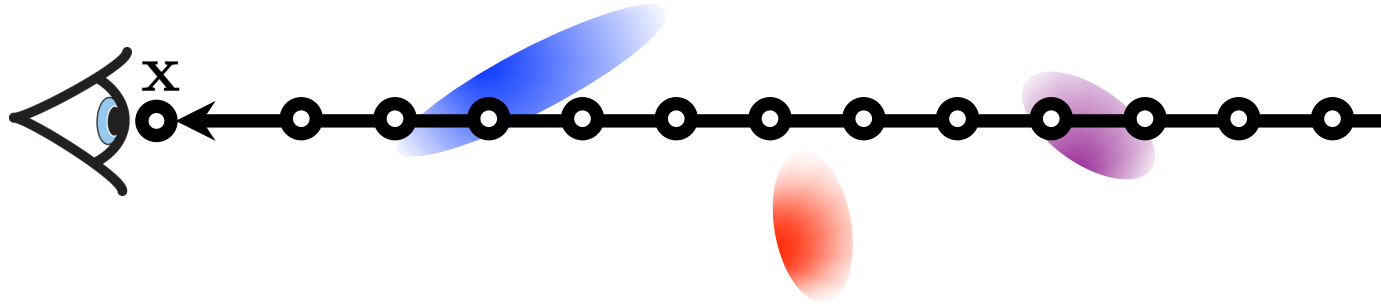
Recap: Rendering Volumes



1. Draw samples along the ray

2. Aggregate their contributions to render

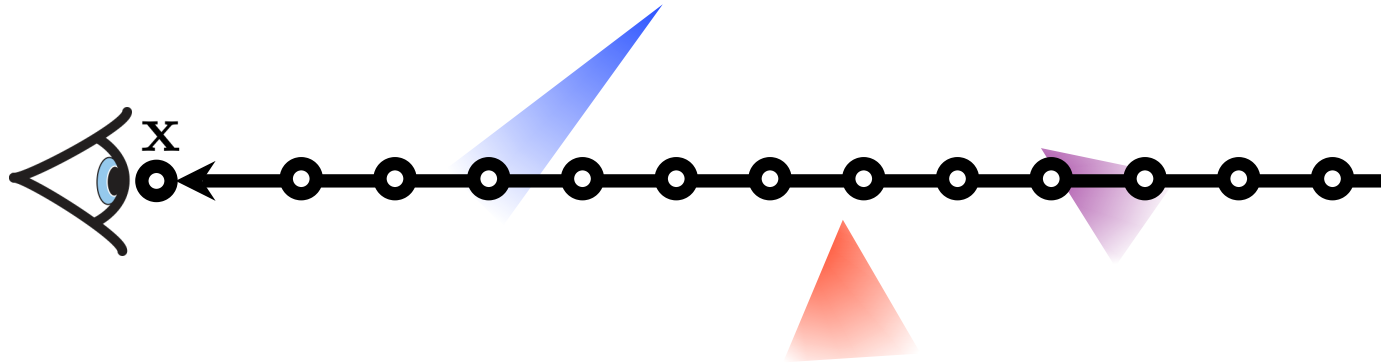
Rendering Primitives (e.g. Gaussians)



1. Draw samples along the ray

2. Aggregate their contributions to render

Rendering Primitives (e.g. Triangles/Meshes)

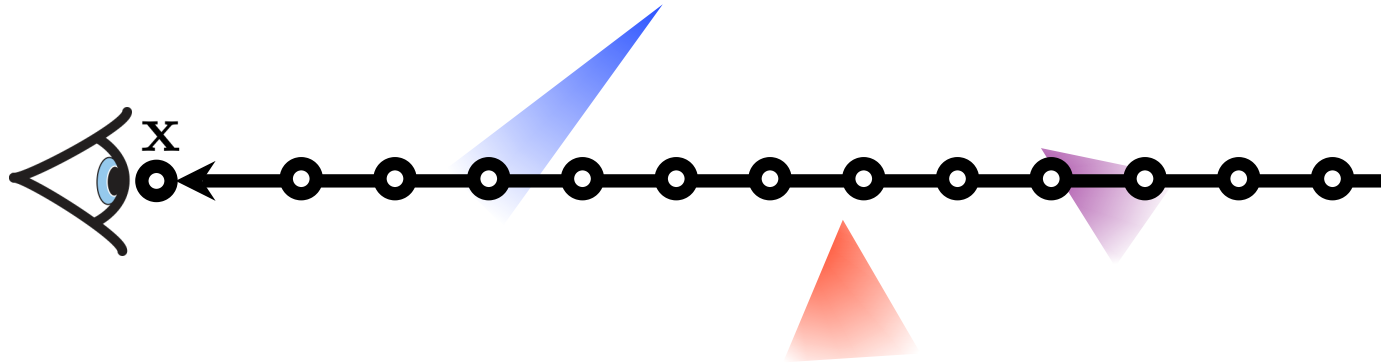


~~1. Draw samples along the ray~~

(wasteful — we know where the primitives are!)

2. Aggregate their contributions to render

Rendering Primitives



1. Find primitives that affect the ray/pixel
2. Aggregate their contributions to render

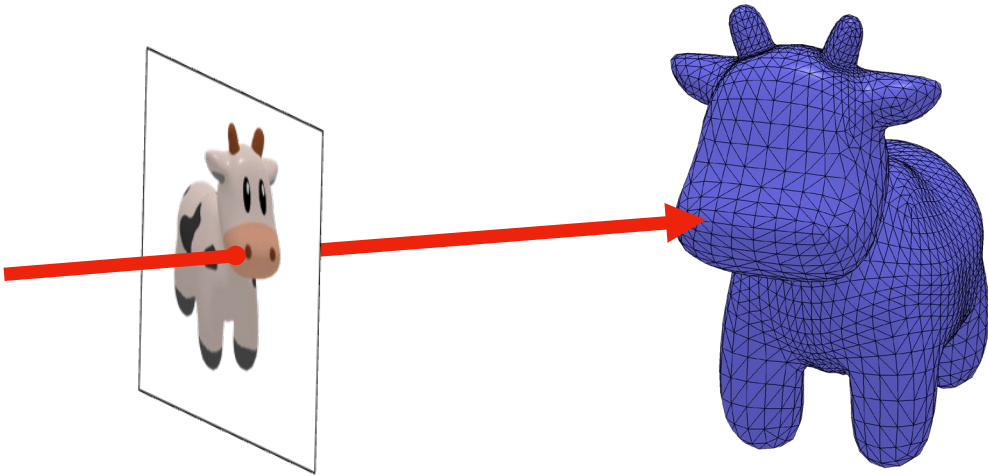
Rendering Primitives

Today's Goals

1. (Differentiable) Mesh Rendering via Ray-tracing
2. From Ray-tracing to Rasterization
3. Rendering via Gaussian Splatting (and some implementation details)

2. Aggregate their contributions to render

Mesh Rendering via Ray-casting

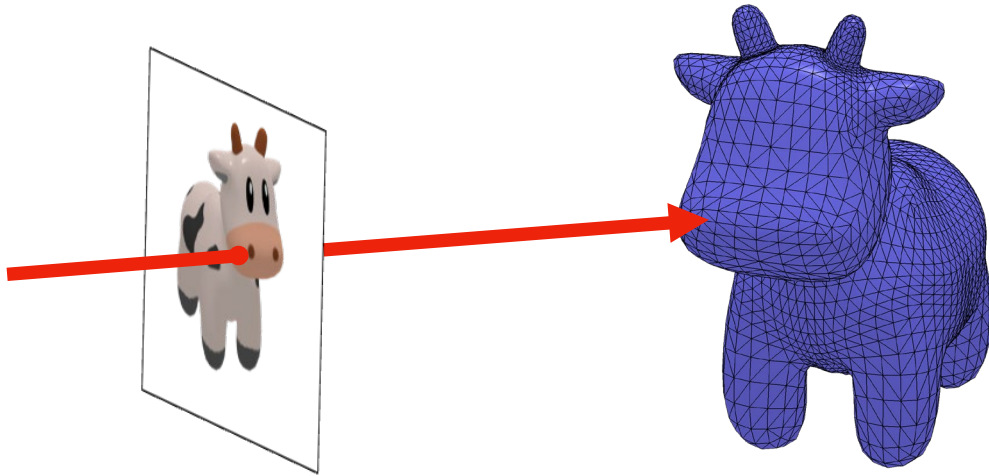


1. Identify the *closest* intersecting triangle

2. Compute appearance for intersection point

Note: 'mesh' is equivalent to a 'triangle set' for above algorithm!

Mesh Rendering: Ray-Triangle Intersection



1. Identify the *closest* intersecting triangle

Given a triangle ($\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$) and a ray ($\mathbf{x}_0, \mathbf{d}$):

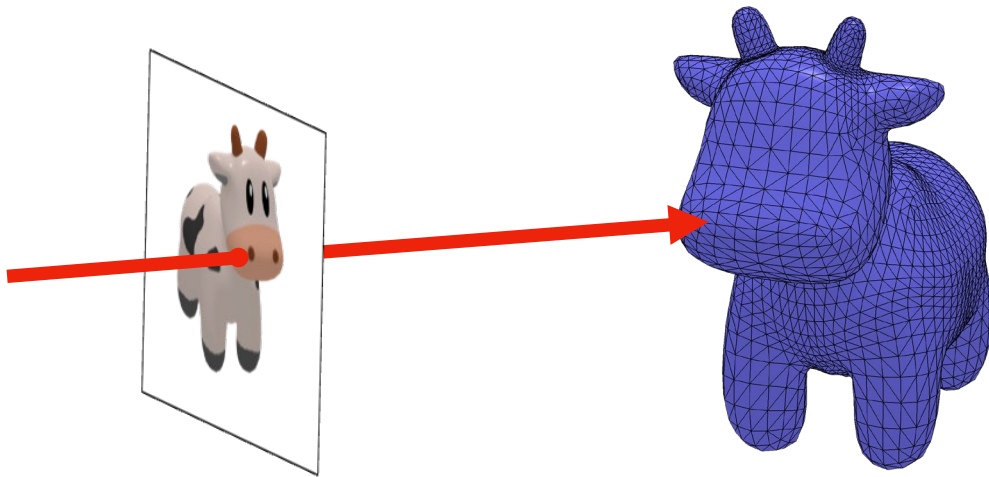
$$\begin{aligned}\mathbf{p} &= \mathbf{x}_0 + \lambda \mathbf{d} \\ &= \alpha \mathbf{v}_1 + \beta \mathbf{v}_2 + (1 - \alpha - \beta) \mathbf{v}_3\end{aligned}$$

Can solve a linear equation to obtain:

Barycentric Coordinates: (α, β)

Distance along the ray: λ

Mesh Rendering: Ray-Mesh Intersection



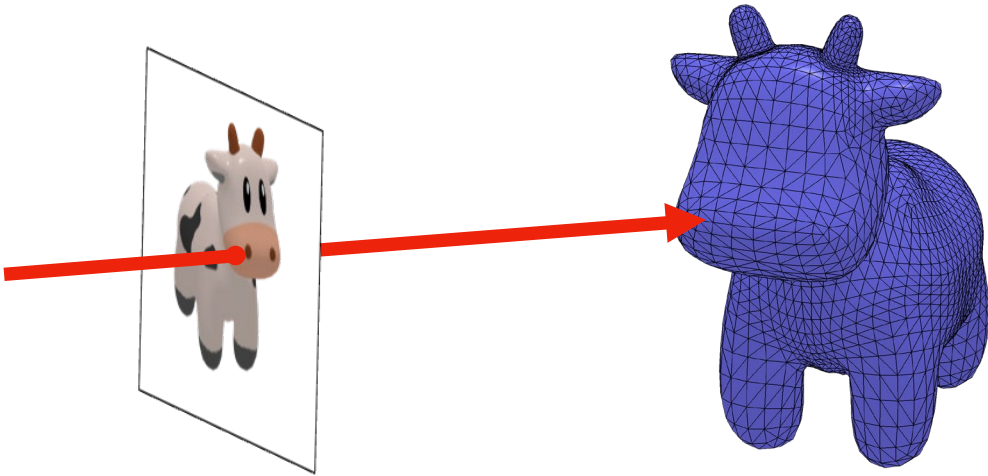
1. Identify the *closest* intersecting triangle

(*rough pseudo-code for illustration, not a complete executable function!)

```
def mesh_ray_intersection(mesh, ray)
    λ_min = ∞
    for triangle in mesh:
        (α,β,λ) = intersect(ray, triangle)
        if (0 < α,β,(1-α-β) < 1) and λ>0 and λ < λ_min:
            intersection = triangle, (α,β)
    return intersection
```

Exhaustive *for* loop not scalable, using spatial hierarchy!

Mesh Rendering



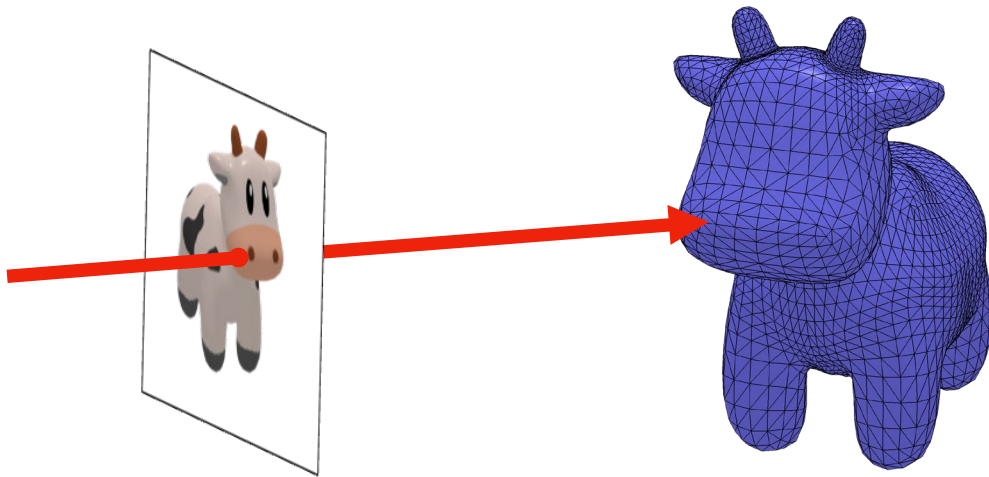
1. Identify the *closest* intersecting triangle

$\text{mesh, ray} \rightarrow \text{triangle}, (\alpha, \beta)$

2. Compute appearance

$\text{triangle}, (\alpha, \beta) \rightarrow \mathbf{c} \in \mathbb{R}^3$

Mesh Rendering: Appearance



Option a: Per-vertex color representation

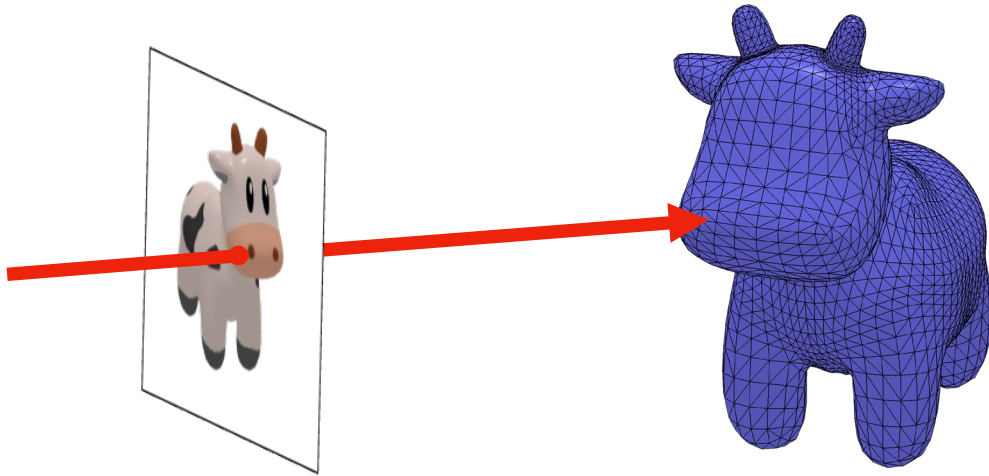
$$(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3) \rightarrow (\mathbf{c}_1, \mathbf{c}_2, \mathbf{c}_3)$$

$$\mathbf{c} = \alpha \mathbf{c}_1 + \beta \mathbf{c}_2 + (1 - \alpha - \beta) \mathbf{c}_3$$

Option b: Texture Atlas representation
 $A (C \times C)$ texture image per triangle.
 $\mathbf{c} = \text{bilinear_sample}(\text{atlas}, [\alpha, \beta])$

Can also extend to lighting-dependent appearance
(using albedo as stored texture + surface normals)

Mesh Rendering



1. Identify the *closest* intersecting triangle
mesh, ray $\rightarrow$ triangle, (α, β)

(α, β) are differentiable

2. Compute appearance
w.r.t. mesh vertex

triangle, $(\alpha, \beta) \rightarrow \mathbf{c} \in \mathbb{R}^3$
positions

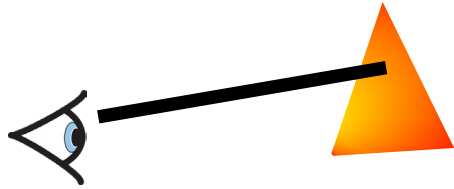
Color $\mathbf{c}$ is differentiable

Q: Is this differentiable?
w.r.t. barycentric

A: Yes, coordinates

Mesh Rendering: Differentiability

Rendered color is differentiable
w.r.t vertex positions of the visible
triangle

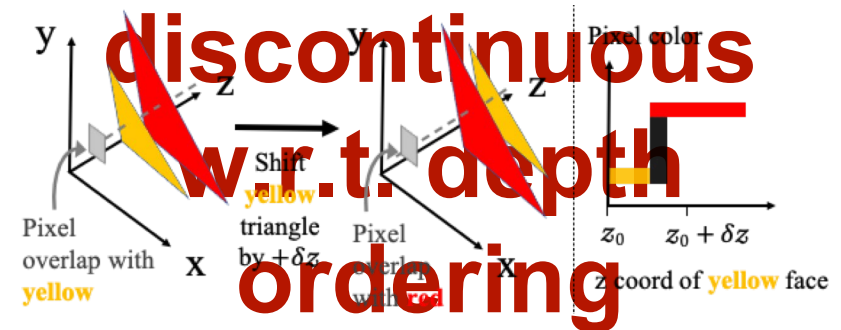


e.g., can make
the color more
red by changing
the position

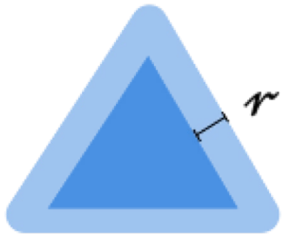
1. rendering is
discontinuous

w.r.t.
occlusion/dis-

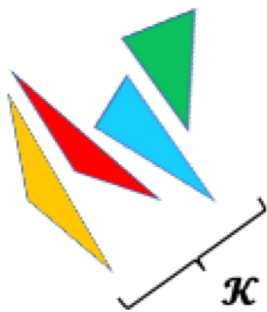
2. rendering is
discontinuous



Differentiable Mesh Rendering



1. Make triangles 'soft'



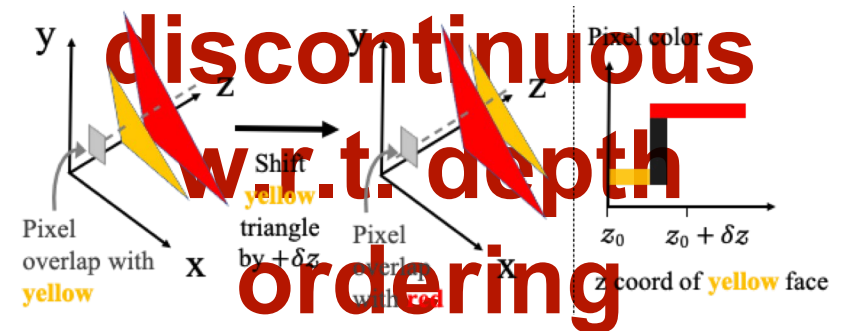
2. Aggregate from multiple triangles

1. rendering is discontinuous



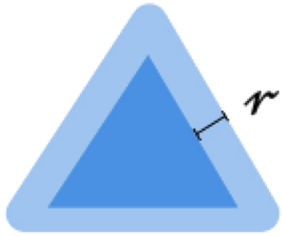
w.r.t. occlusion/dis-

2. rendering is discontinuous



Alternative: visibility-aware differentiable rendering

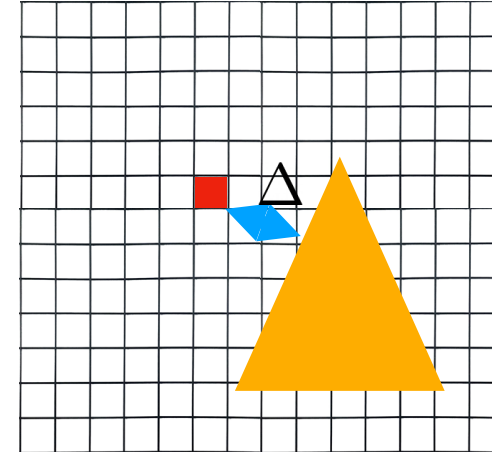
Differentiable Mesh Rendering



1. Make triangles 'soft'



2. Aggregate from multiple triangles



$$m = \text{sigmoid}\left(\frac{-\Delta}{\sigma}\right)$$

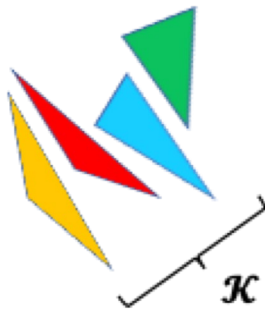
soft 'mask' of a triangle depending on
(pixel-space) signed distance

(with a manually-defined cutoff, e.g., 10σ)

Differentiable Mesh Rendering



1. Make triangles 'soft'



2. Aggregate from multiple triangles

1. Find K closest triangles that have a non-zero mask for the current pixel (similar code as finding 1 closest triangle)

2. Weigh each triangle based on mask and distance along ray

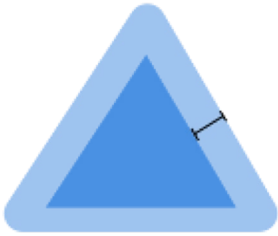
$$w_k \propto m_k e^{-\lambda_k / \gamma}$$

can also model 'background' with a small default weight w_{bg}

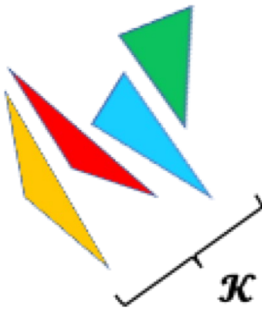
3. Combine per-triangle appearance

$$\mathbf{c} = \sum_k w_k \mathbf{c}_k$$

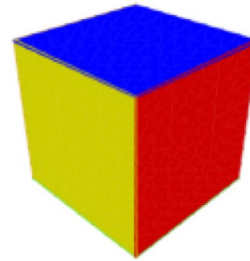
Differentiable Mesh Rendering



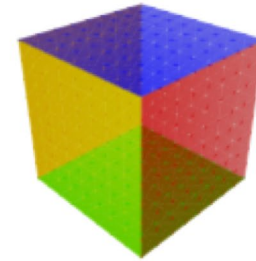
1. Make triangles 'soft'



2. Aggregate from multiple triangles



Standard rendering

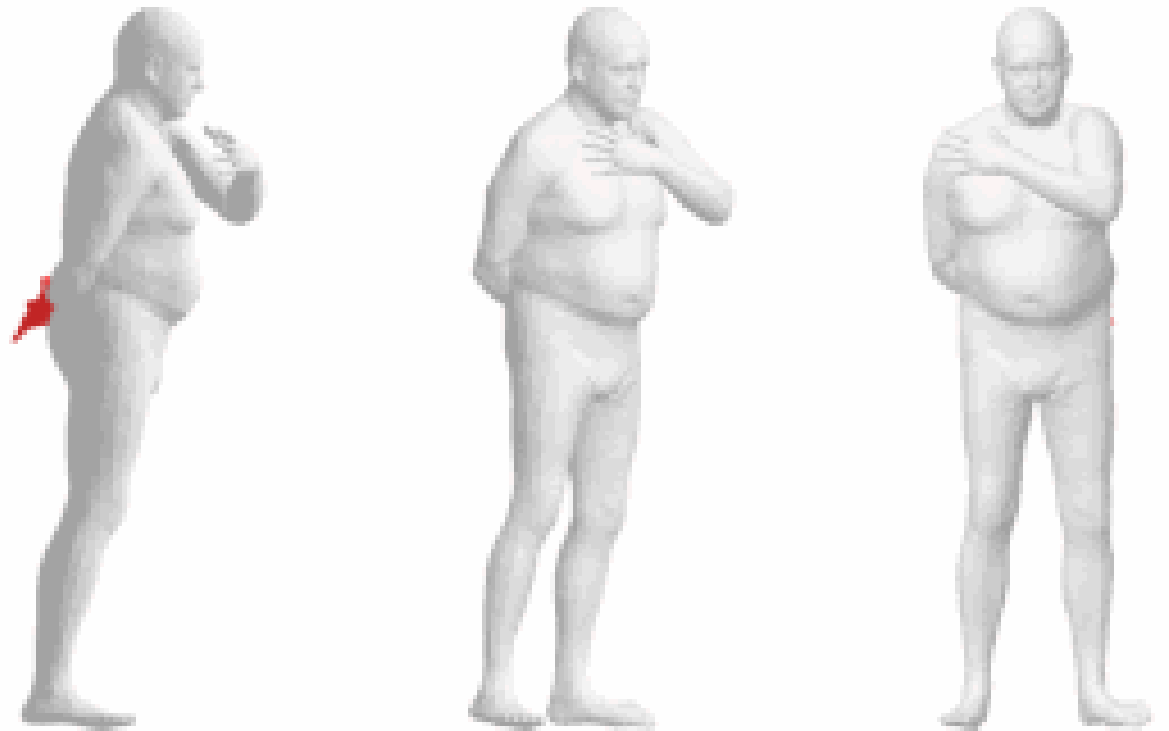
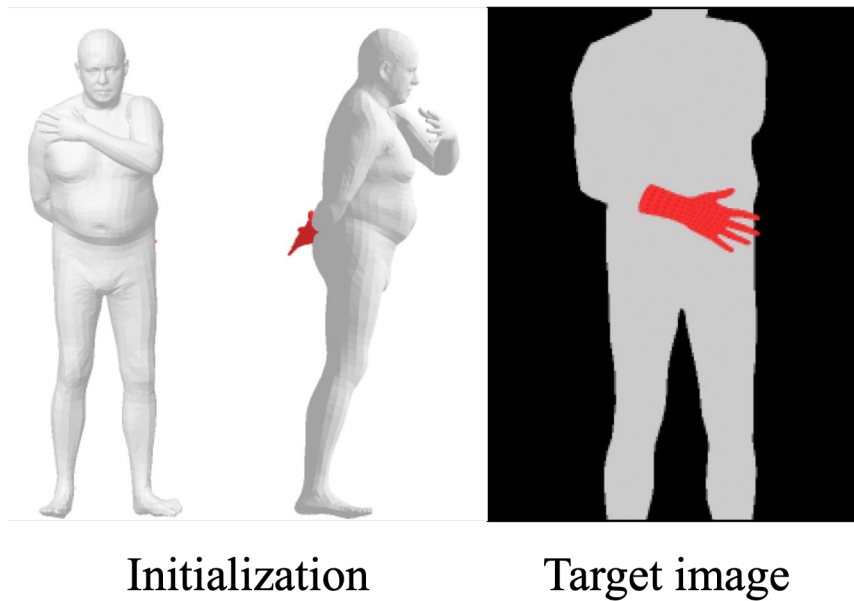


Rendered
w/ larger γ

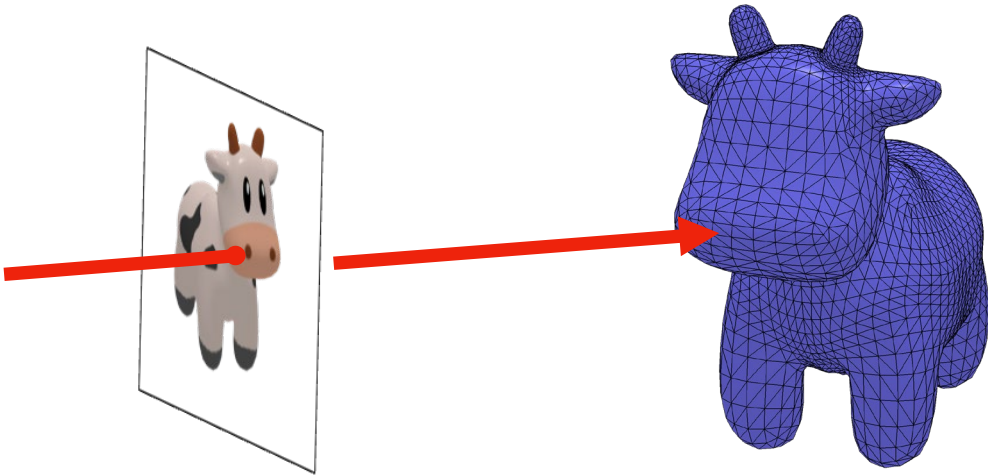


Rendered
w/ larger γ and σ

Differentiable Mesh Rendering

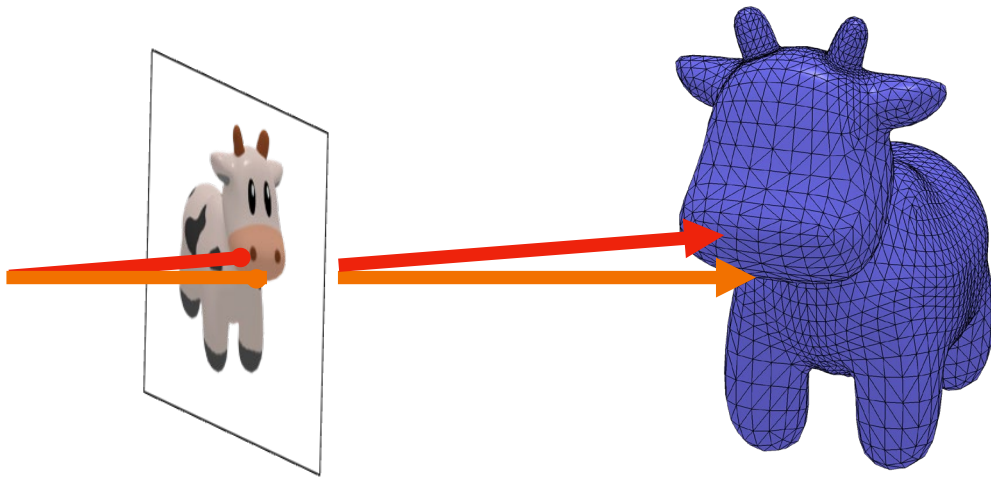


Differentiable Mesh Rendering via Ray tracing



1. Identify the closest K 'softly' intersecting triangles (and the corresponding pixel-space SDF)
2. Aggregate appearance using masks, depth-dependent weights, and per-triangle appearance

Differentiable Mesh Rendering via Ray tracing



Some redundant
computation across pixels

```
def render(mesh, camera):  
    ### Iterate over all pixels  
    for pixel in camera.grid:  
        ray = pixel2ray(camera, pixel)  
  
        ### Iterate over all triangles  
        for triangle in mesh:  
            #####  
            # Find current K closest intersecting triangles  
            #####  
  
        ### Iterate over triangles influencing this pixel  
        for triangle in sorted_k_closest:  
            #####  
            # Aggregate appearance  
            #####
```

Idea: iterate over triangles just once (and store what pixels they affect)

From Ray tracing to Rasterization

Rasterization

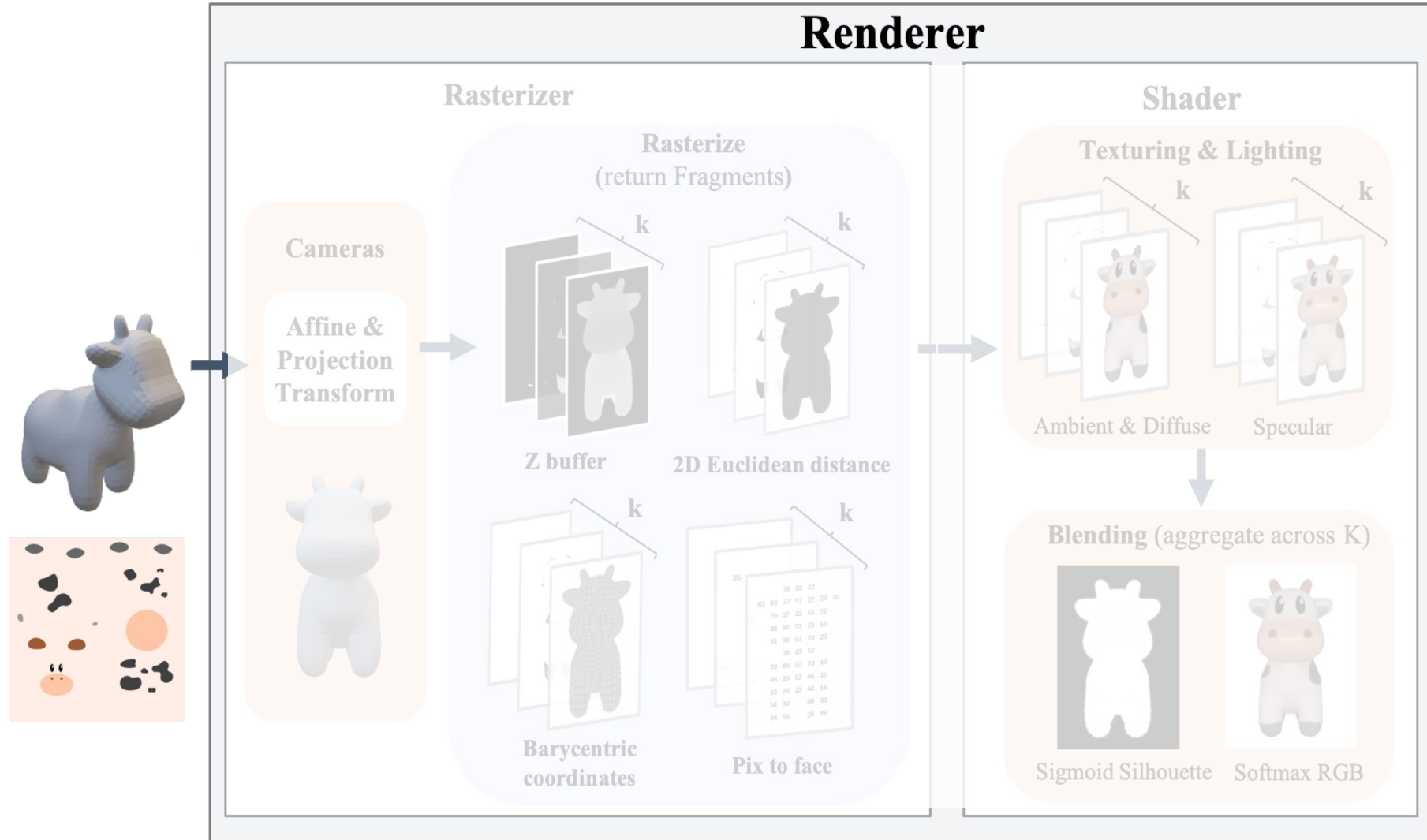


Blending (+ Shading)

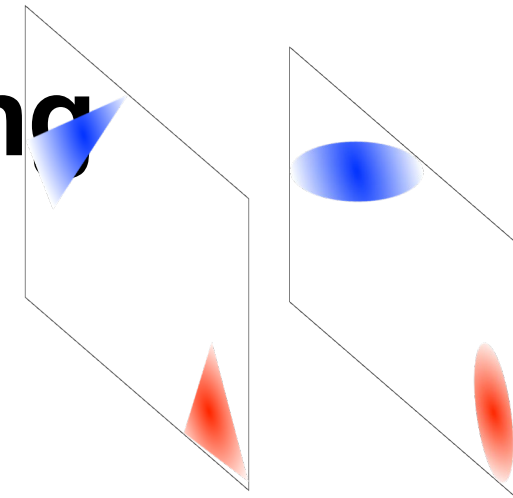


```
def render(mesh, camera):  
    ### some structure to store K triangles  
    ### for each pixel  
    sorted_k_closest_img = ...  
  
    ### Iterate over all triangles  
    for triangle in mesh:  
        tri_2d = project(triangle, image)  
        #####  
        # Update closest_K_img for pixels within tri_2d  
        #####  
  
    ### Iterate over all pixels  
    for pixel in camera.grid:  
        ### Iterate over triangles influencing this pixel  
        for triangle in sorted_k_closest_img[pixel]:  
            #####  
            # Aggregate appearance  
            #####|
```

From Ray tracing to Rasterization



Differentiable Primitive Rendering



```
def render(primitives, camera):  
    ###  
    # Initialize a rasterization data structure  
    # (records influencing primitives for each pixel)  
    ###
```

Rasterization



```
    for primitive in primitives:  
        prim2d = project(primitive, camera)  
        ###  
        # Update rasterization data structure  
        ###
```

Blending



```
    for pixel in camera.grid:  
        ###  
        # Aggregate appearance from influencing primitives  
        ###
```

Differentiable **Gaussian** Rendering

What is the representation
of a 3D Gaussian?

How to project to 2D
and rasterize?

How to model/aggregate
appearance?

Rasterization



Blending

```
def render(gaussians, camera):  
    ###  
    # Initialize a rasterization data structure  
    # (records influencing primitives for each pixel)  
    ###  
  
    for gaussian in gaussians:  
        gauss2d = project(gaussian, camera)  
        ###  
        # Update rasterization data structure  
        ###  
  
    for pixel in camera.grid:  
        ###  
        # Aggregate appearance from influencing gaussians  
        ###
```


Differentiable Gaussian Rendering

What is the representation
of a 3D Gaussian?

How to project to 2D
and rasterize?

How to model/aggregate
appearance?

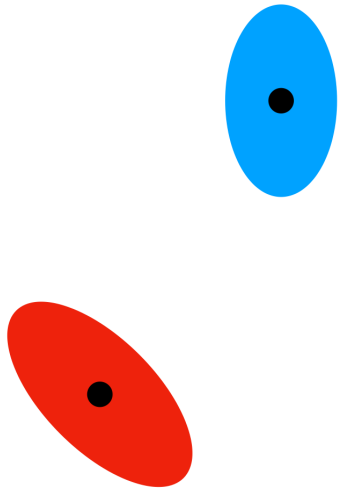


Differentiable Gaussian Rendering

What is the representation of a 3D Gaussian?

How to project to 2D and rasterize?

How to model/aggregate appearance?



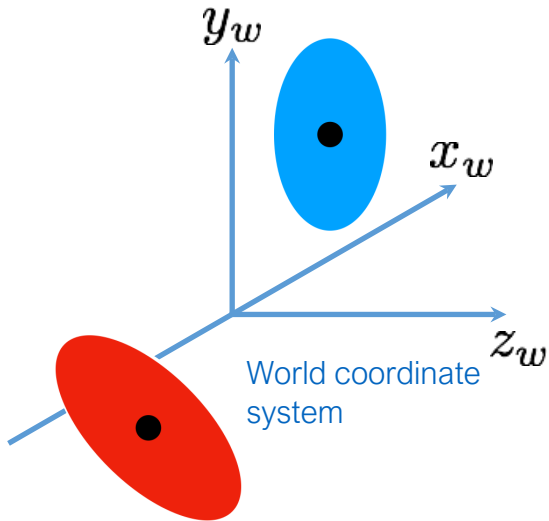
$$\mathcal{G}_{\mathbf{V}}(\mathbf{x} - \mathbf{p}) = \frac{1}{2\pi|\mathbf{V}|^{\frac{1}{2}}} e^{-\frac{1}{2}(\mathbf{x}-\mathbf{p})^T \mathbf{V}^{-1}(\mathbf{x}-\mathbf{p})}$$

Factorize as scale and rotation: $\mathbf{V} = \mathbf{R}\mathbf{S}\mathbf{S}^T\mathbf{R}^T$

Each Gaussian also has an opacity and view-dependent color (via SH coefficients): $\alpha, \mathbf{c}$

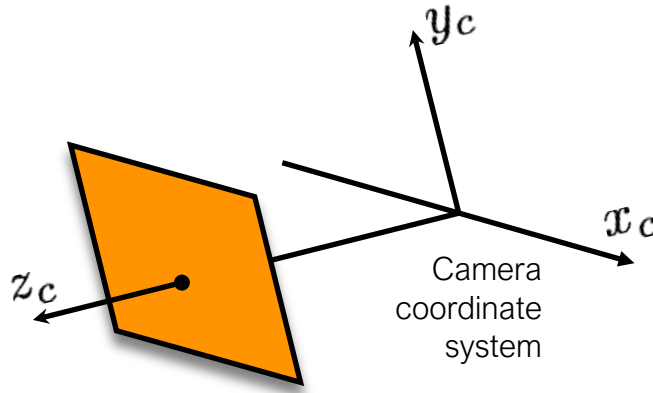
Differentiable Gaussian Rendering

What is the representation of a 3D Gaussian?



$\mathbf{p}, R, S$

How to project to 2D and rasterize?



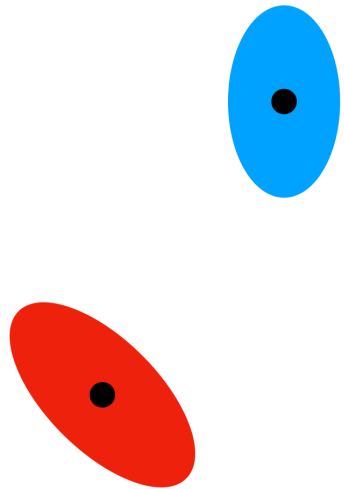
$\mathbf{p}', R', S$

How to model/aggregate appearance?

We can use the camera extrinsics to transform each 3D Gaussian to the camera frame

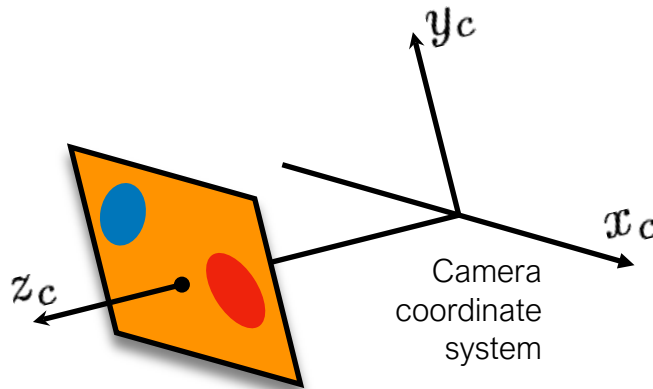
Differentiable Gaussian Rendering

What is the representation of a 3D Gaussian?



$\mathbf{p}', R', S$

How to project to 2D and rasterize?



Q: What is the image-space projection of a 3D Gaussian?

A: Can approximate as a 2D Gaussian!

How to model/aggregate appearance?

$$\pi(\mathbf{x}) = \mathbf{u} \quad z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

π : Projection function for mapping 3D points to pixels

2D mean: $\mu_{2D} = \pi(\mu_{3D})$

2D covariance:

$$J = \frac{\partial \pi}{\partial \mathbf{x}}(\mu_{3D})$$

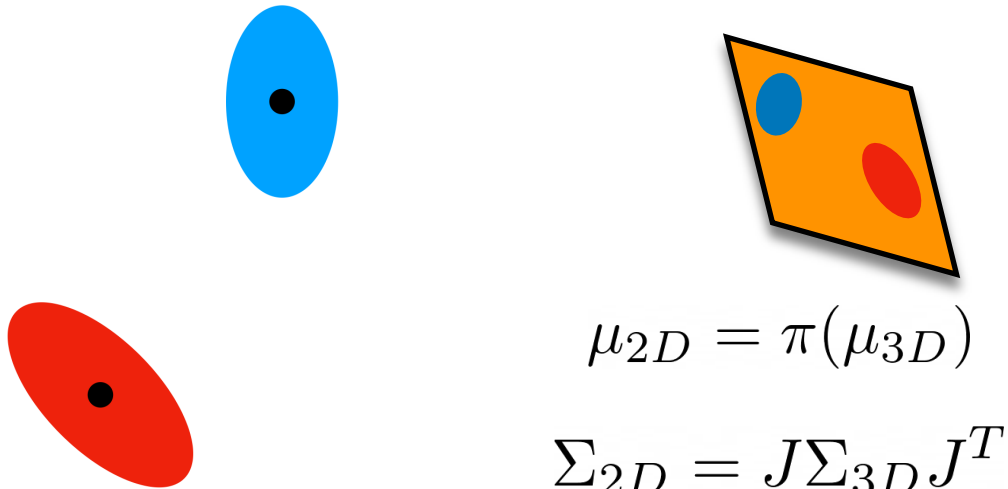
$$\Sigma_{2D} = J \Sigma_{3D} J^T$$

Differentiable Gaussian Rendering

What is the representation of a 3D Gaussian?

How to project to 2D and rasterize?

How to model/aggregate appearance?



1. Sort Gaussians from closest to furthest from the camera
2. For each pixel $\mathbf{u}$, compute opacity for each gaussian $\mathcal{G}_k$:

$$\bar{\alpha}_k = \alpha_k \frac{e^{-(\mathbf{u} - \mu_{2D}^k)^T (\Sigma_{2D}^k)^{-1} (\mathbf{u} - \mu_{2D}^k)}}{2\pi |\Sigma_{2D}^k|^{0.5}}$$

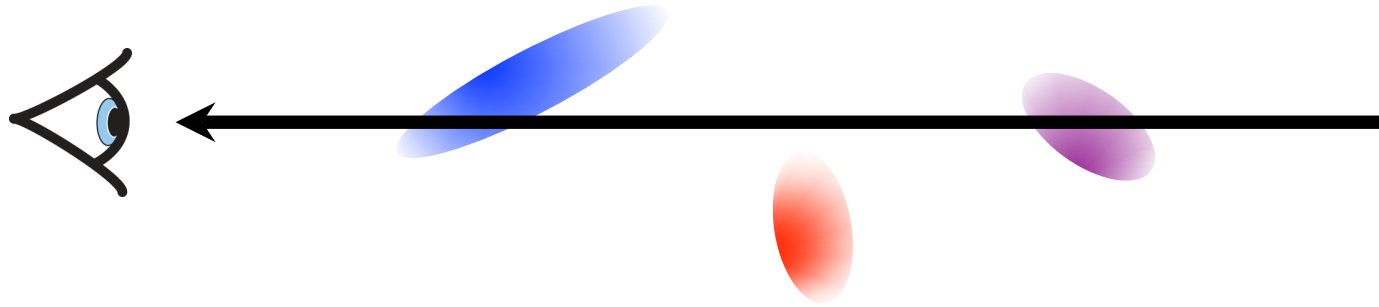
(In practice, can rasterize 'blocks' instead of entire image as not all Gaussians influence all blocks)

Differentiable Gaussian Rendering

What is the representation
of a 3D Gaussian?

How to project to 2D
and rasterize?

How to model/aggregate
appearance?



Compute per-Gaussian weights based on opacities of current and previous Gaussians:

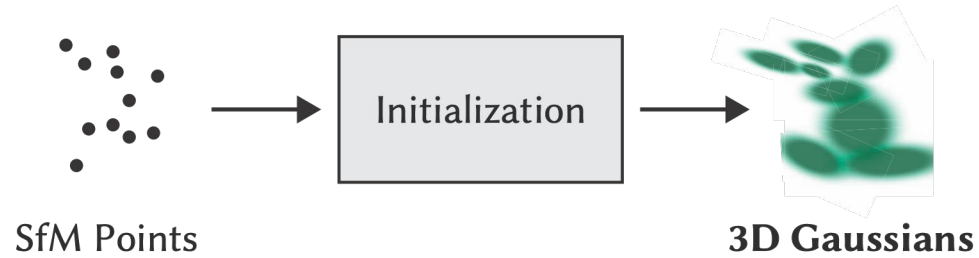
$$w_k = \bar{\alpha}_k \prod_{j=1}^{k-1} (1 - \bar{\alpha}_j)$$

Use per-Gaussian SH coefficients and ray
direction to get view-dependent color $\mathbf{c}_k$

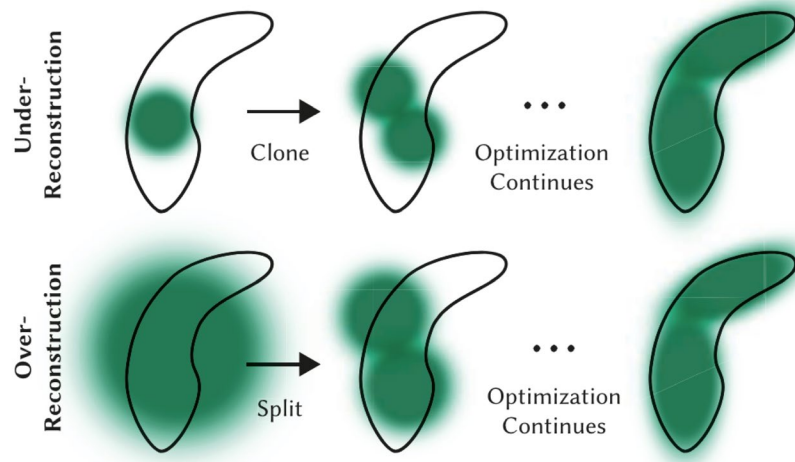
Aggregate to obtain pixel color:

$$\mathbf{c} = \sum_k w_k \mathbf{c}_k$$

Gaussian Splatting: Bells and Whistles



Initialize with sparse point cloud from SfM



Split/clone Gaussians based on heuristics





This video contains a voice-over

3D Gaussian Splatting for Real-Time Radiance Field Rendering

SIGGRAPH 2023
(ACM Transactions on Graphics)

Bernhard Kerbl*



Georgios Kopanas*



Thomas Leimkühler



George Drettakis



* Denotes equal contribution

Fuzzy Metaballs: Approximate Differentiable Rendering with Algebraic Surfaces

[Leonid Keselman](#) and [Martial Hebert](#)

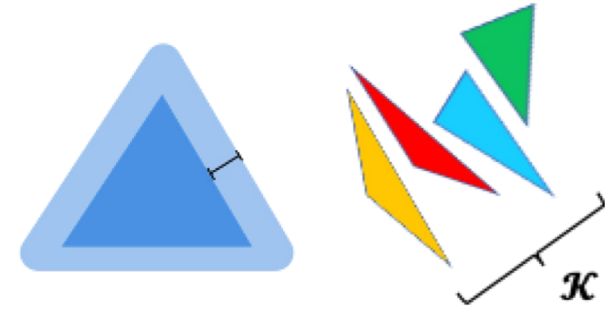
ECCV 2022 (Oral)



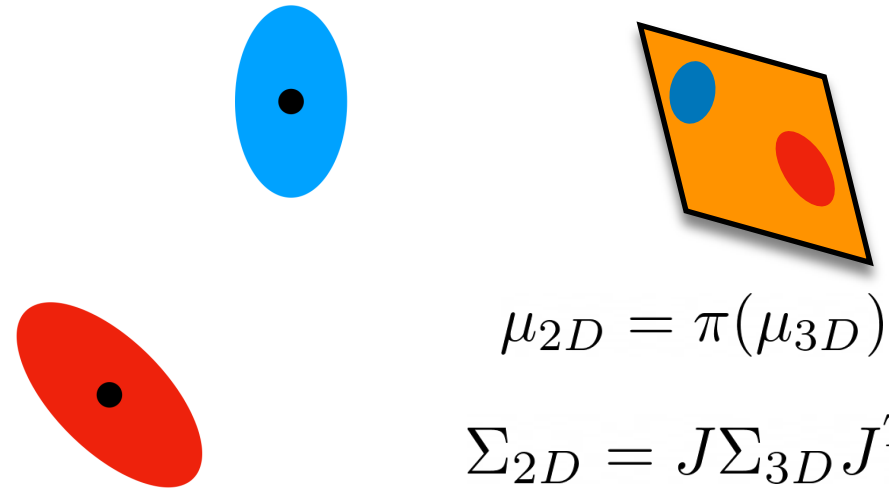
Summary

```
def render(primitives, camera):  
    ###  
    # Initialize a rasterization data structure  
    # (records influencing primitives for each pixel)  
    ###  
  
    for primitive in primitives:  
        prim2d = project(primitive, camera)  
        ###  
        # Update rasterization data structure  
        ###  
  
    for pixel in camera.grid:  
        ###  
        # Aggregate appearance from influencing primitives  
        ###
```

Rasterization + Blending for efficient rendering

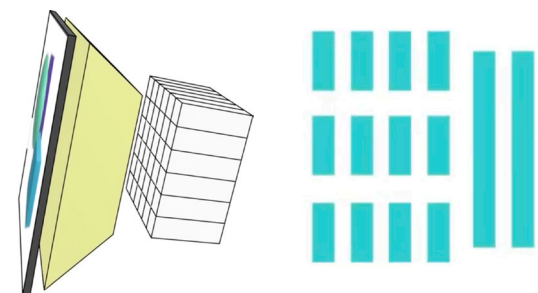
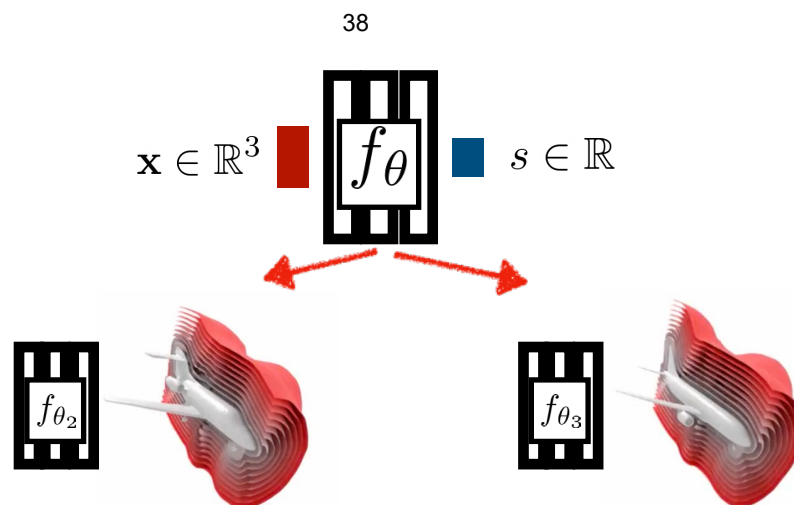
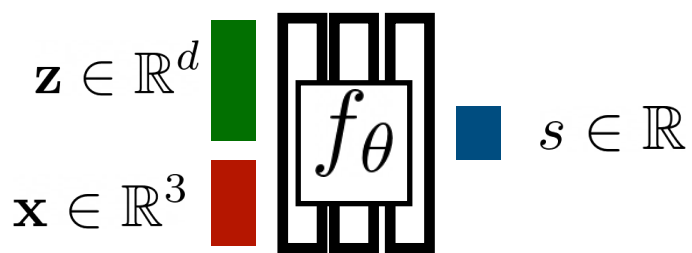


Aggregating 'soft' primitives
along a ray allows differentiable
rendering

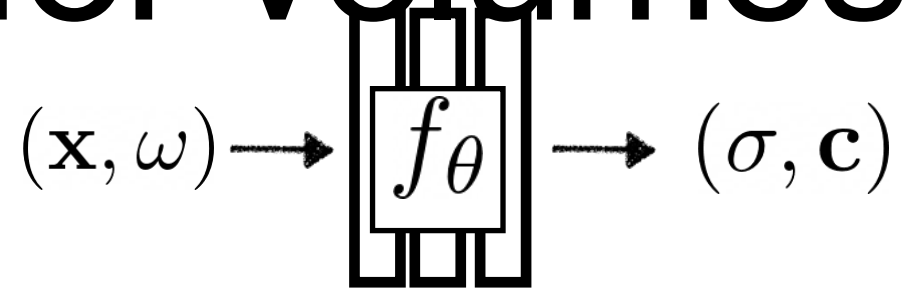
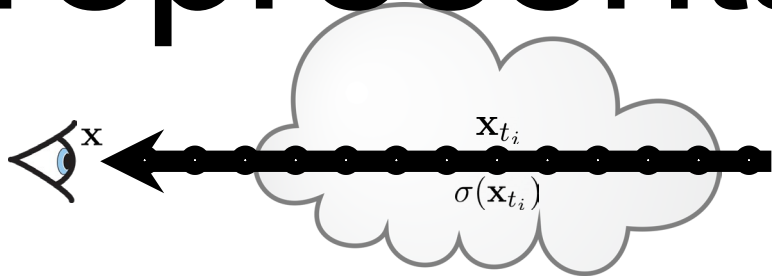


Projecting 3D Gaussians for 2D
rendering

Generalizable Neural Fields



Recap: Rendering and representation for volumes



39

$$\sigma_{t_i} \equiv \sigma(\mathbf{x}_{t_i}) \longrightarrow L(\mathbf{x}, \omega)$$
$$L_e(\mathbf{x}_{t_i}, \omega)$$

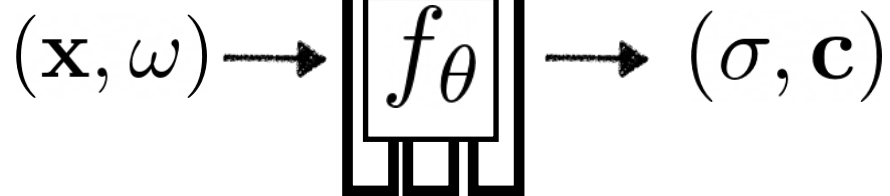
Rendering algorithm

$$\sigma \in \mathbb{R}^+$$

$$\mathbf{c} \in [0, 1]^3$$

Pointwise density
(and color)

Scene-specific neural representation



40

One neural network **per** scene (or object)

Scene-specific neural

r

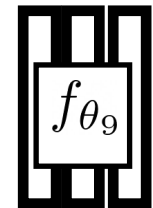
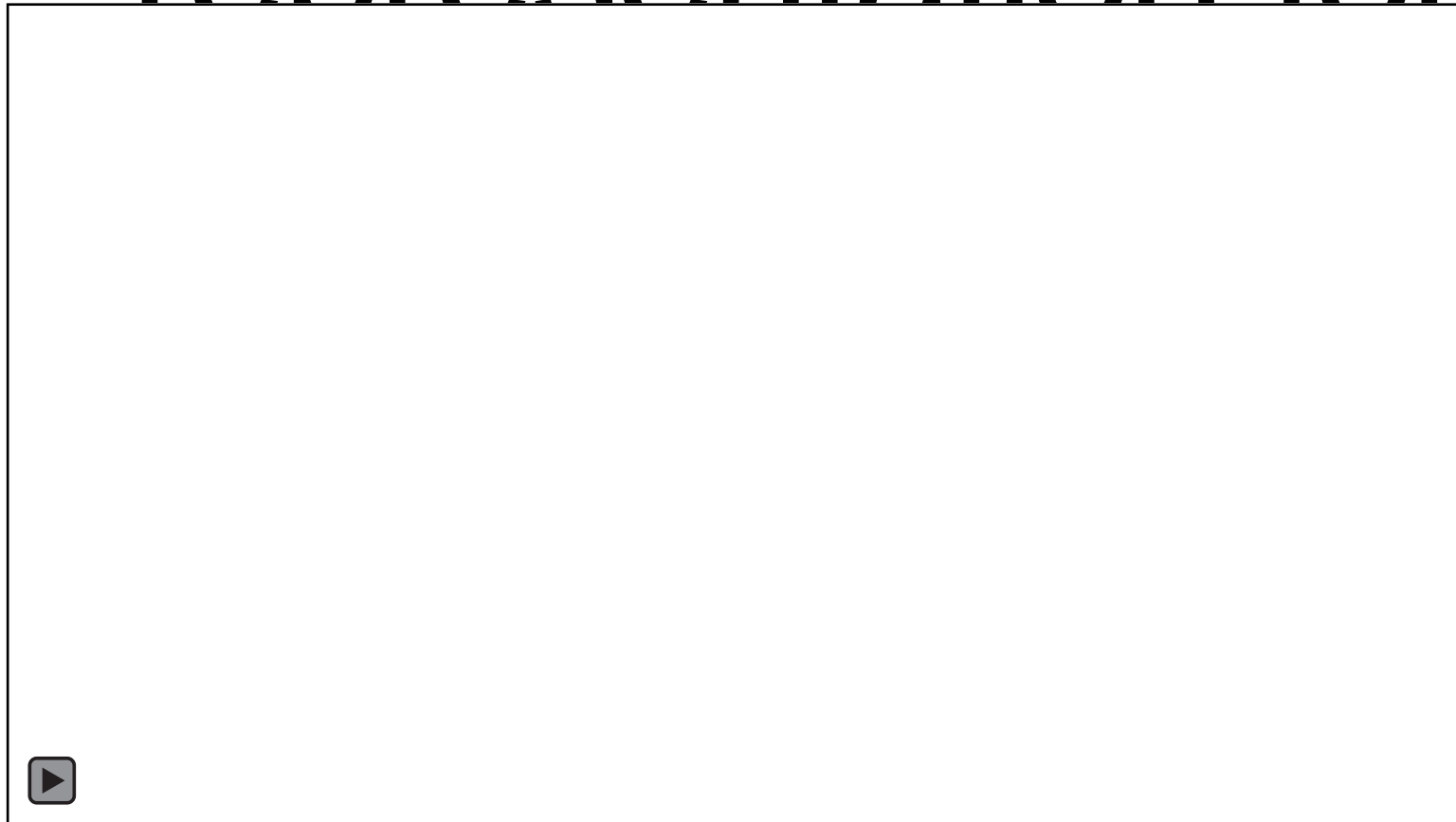


Scene-specific neural

r



Recap: Rendering and representation for volumes



No generalization

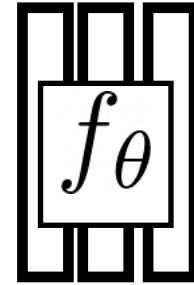
Scene-specific neural representation



100 training images



44



Instance-specific NN

Can capture detailed shape and appearance

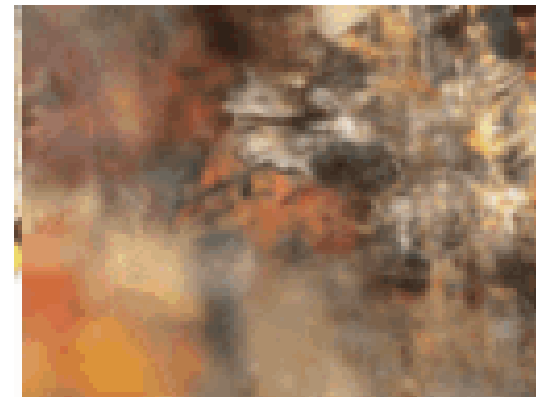
No priors *across* scenes/objects

Scene-specific neural representation



45

No mechanism to hallucinate unseen aspects (e.g. left/front of object)



Scene-specific neural representation



46

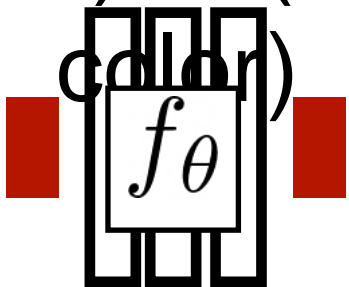
Goals

1. Architectures and mechanisms for **generalizable** neural fields (i.e., reusing/sharing parameters across scenes)
2. Review of different approaches and understand tradeoffs
3. Disclaimer: emerging research area (no 'right' way)

Neural fields: coordinates to values

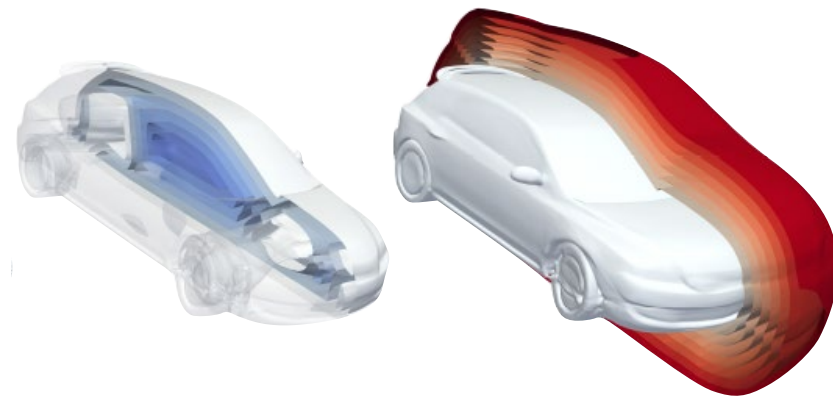
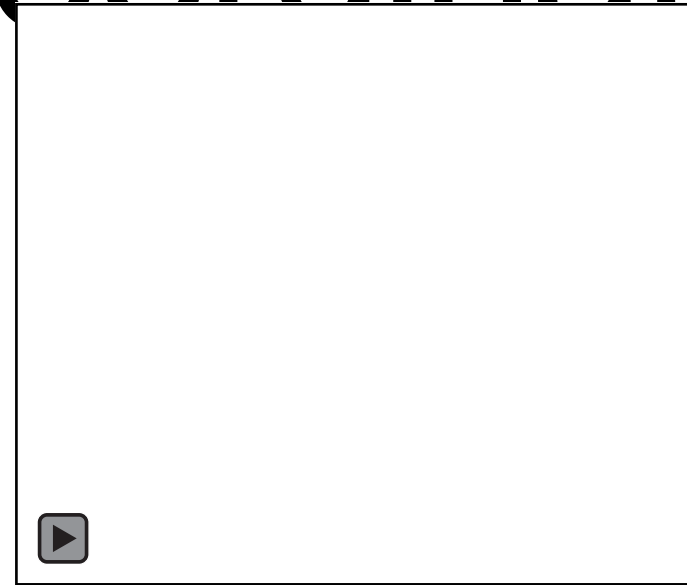


Radiance field: (position, direction) $\rightarrow$ (density,



implicit field: (position)
 $\rightarrow$ (implicit function)

47

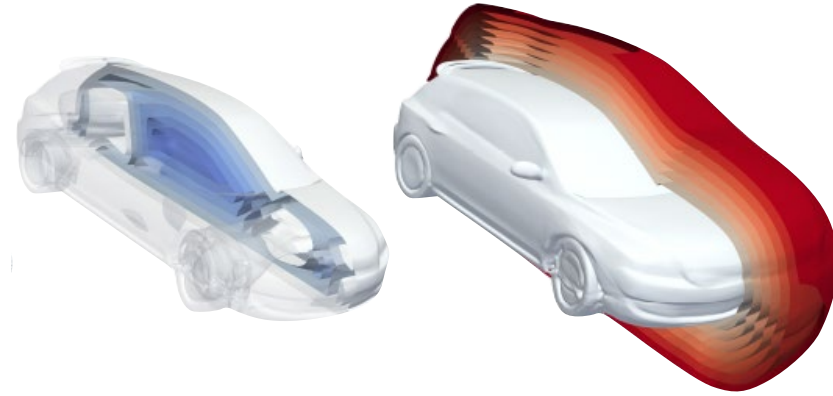


Generalizable neural implicit fields

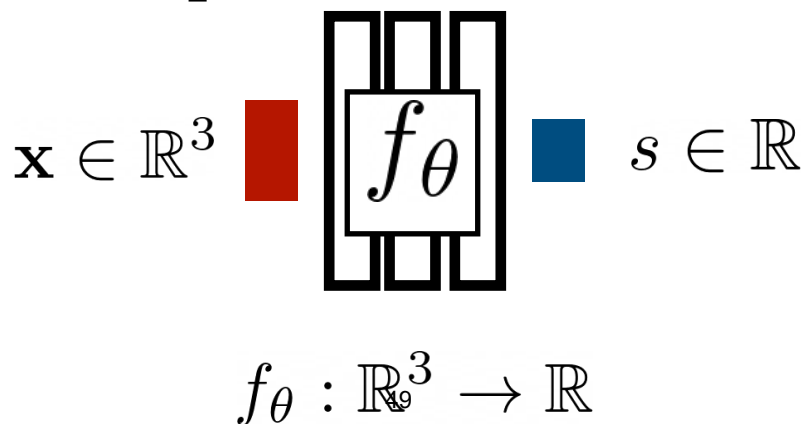
Implicit field: (position)
-> (implicit function)

Q: How to design generalizable implicit function fields?

(hopefully similar principles may apply to radiance fields)



Instance-specific neural implicit

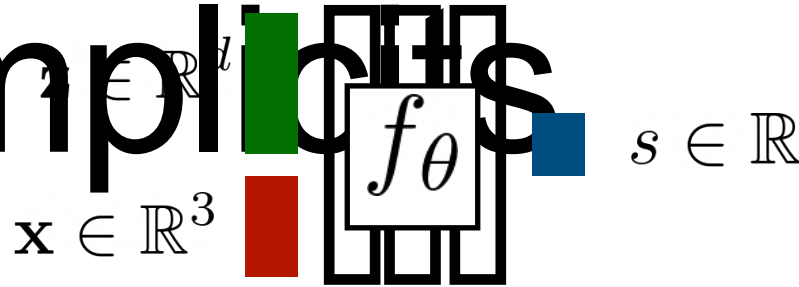


**Implicit field (for
a single
instance).**

For a 6-layer network with 1000-d feature space, about 6M parameters per instance!

(position) ->
(implicit function)

Latent conditioning-based neural implicit fields



$$f_\theta : \mathbb{R}^3 \times \mathbb{R}^d \rightarrow \mathbb{R}$$

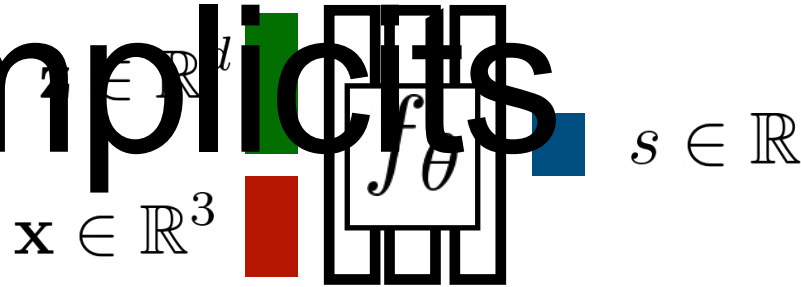
Generalizable implicit field:

Each object is represented by a corresponding
(latent code, position) $\rightarrow$
latent code (only d parameters per instance)

(implicit function)
The same neural net parameters across **all** objects

(similar to occupancy net)

Latent conditioning-based neural implicit



$$f_\theta : \mathbb{R}^3 \times \mathbb{R}^d \rightarrow \mathbb{R}$$

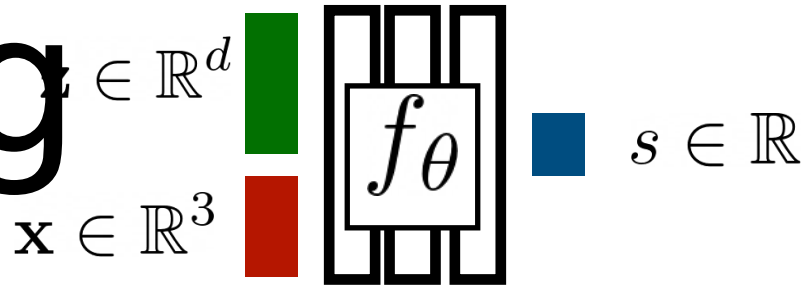
Generalizable implicit field:

Q: What is a good architecture?
(latent code, position) ->

Q: Given some training shapes, how to learn f ?
(implicit function)

Q: How do we infer shapes for new instances?

Learning via auto-decoding



$\mathbf{z}_1$



$f_\theta(\cdot, \mathbf{z}_1)$



$\mathbf{z}_2$

52



$f_\theta(\cdot, \mathbf{z}_2)$



$\mathbf{z}_3$



$f_\theta(\cdot, \mathbf{z}_3)$

Initialize a (random) latent variable per-scene

Jointly optimize neural network and per-

instance latent variable

Learning via auto-

$$\begin{array}{c} \mathbf{z} \in \mathbb{R}^d \\ \mathbf{x} \in \mathbb{R}^3 \end{array} \begin{array}{c} \text{green square} \\ \text{red square} \end{array} \begin{array}{c} \boxed{f_\theta} \\ \boxed{} \end{array} \begin{array}{c} \text{blue square} \\ \text{blue square} \end{array} \begin{array}{c} s \in \mathbb{R} \\ s \in \mathbb{R} \end{array}$$



$\mathbf{z}_1$



$f_\theta(\cdot, \mathbf{z}_1)$



$\mathbf{z}_2$

53



$f_\theta(\cdot, \mathbf{z}_2)$



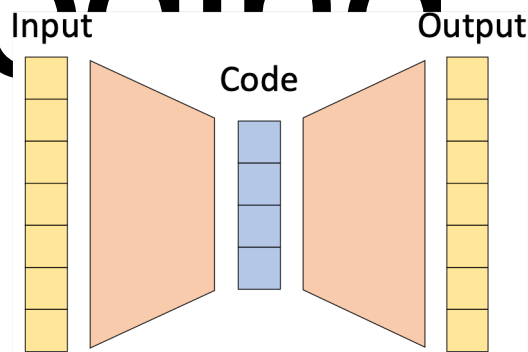
$\mathbf{z}_3$



$f_\theta(\cdot, \mathbf{z}_3)$

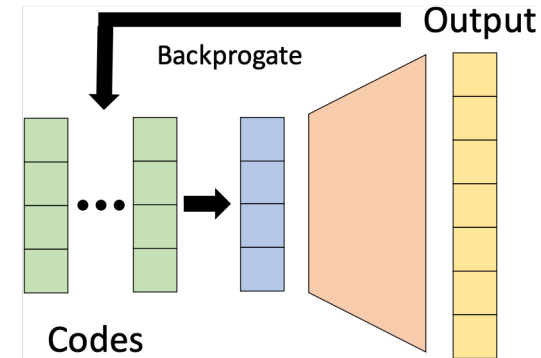
$$\arg \min_{\theta, \{\mathbf{z}_i\}_{i=1}^N} \sum_{i=1}^N \left(\overbrace{\sum_{j=1}^K \mathcal{L}(f_\theta(\mathbf{z}_i, \mathbf{x}_j), s_j)}^{\text{data term}} + \underbrace{\frac{1}{\sigma^2} \|\mathbf{z}_i\|_2^2}_{\text{Gaussian prior}} \right)$$

Auto-encoding vs auto-decoding



(a) Auto-encoder

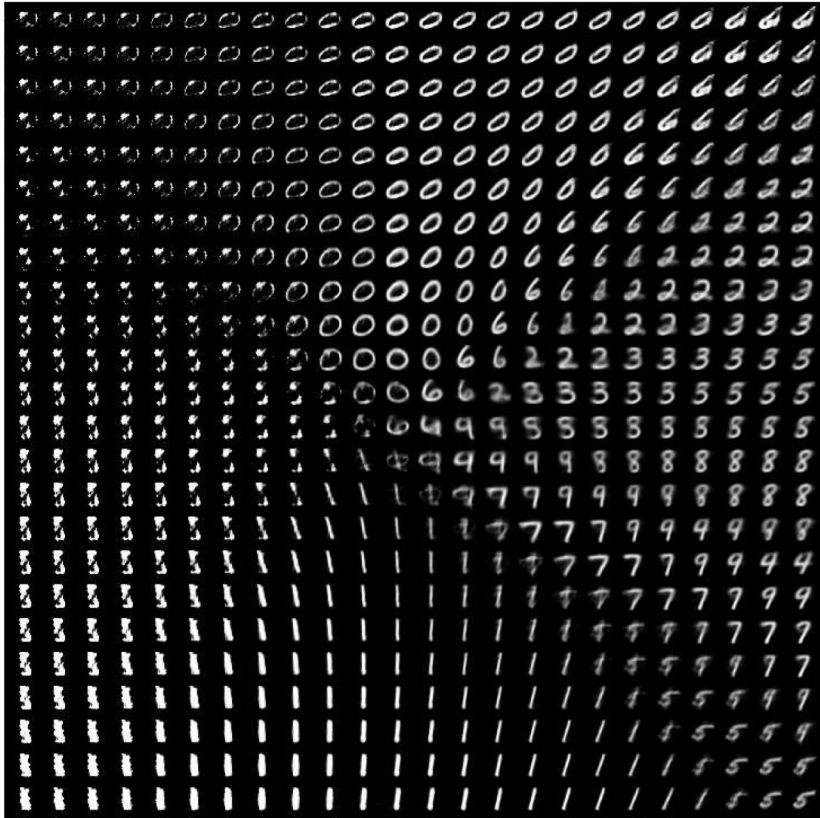
54



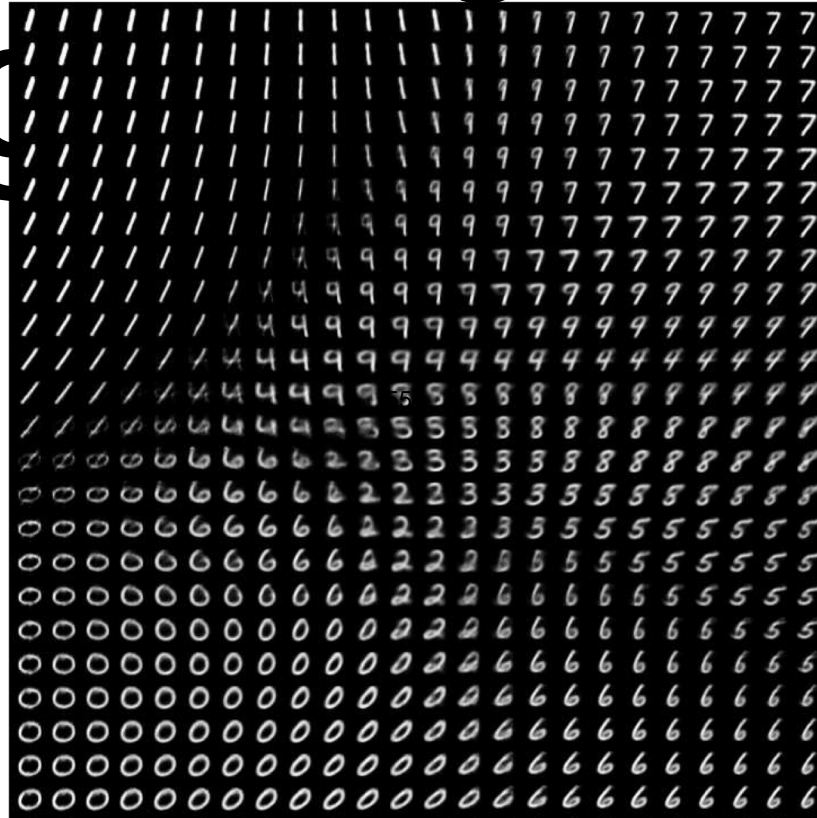
(b) Auto-decoder

Eventually, we will only use the decoder—a question of which optimization is easier

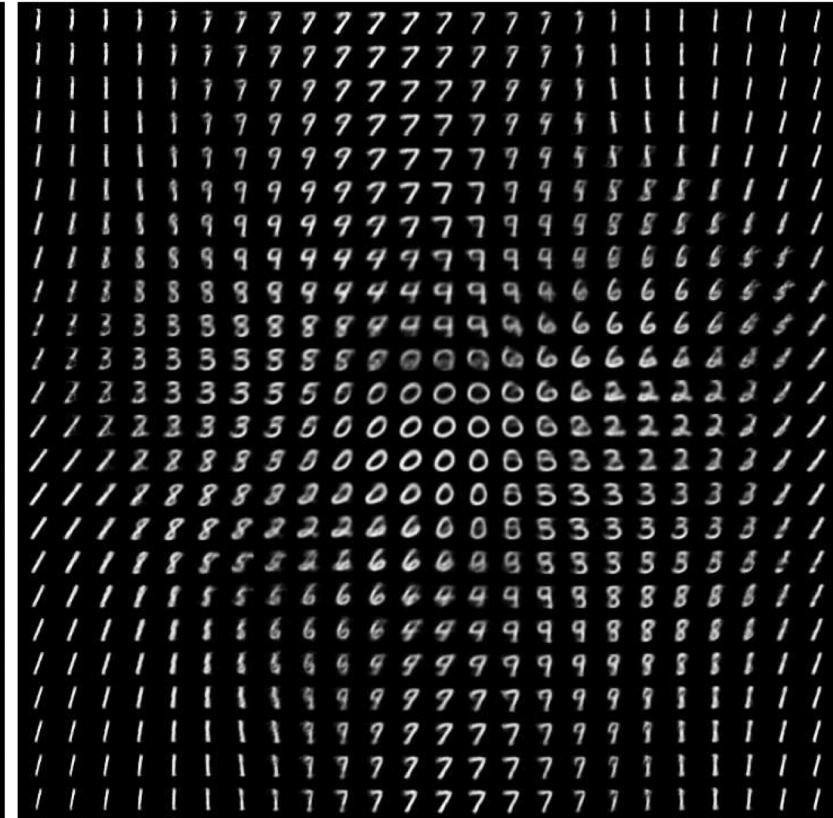
Auto-encoding vs auto-



(a) Auto Encoder (AE)



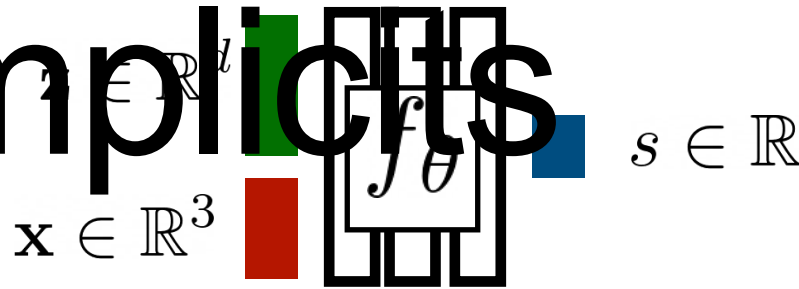
(b) Variational Auto Encoder (VAE)



(c) Auto Decoder (AD)

A Gaussian prior allows a meaningful latent space

Latent conditioning-based neural implicit



$$f_\theta : \mathbb{R}^3 \times \mathbb{R}^d \rightarrow \mathbb{R}$$

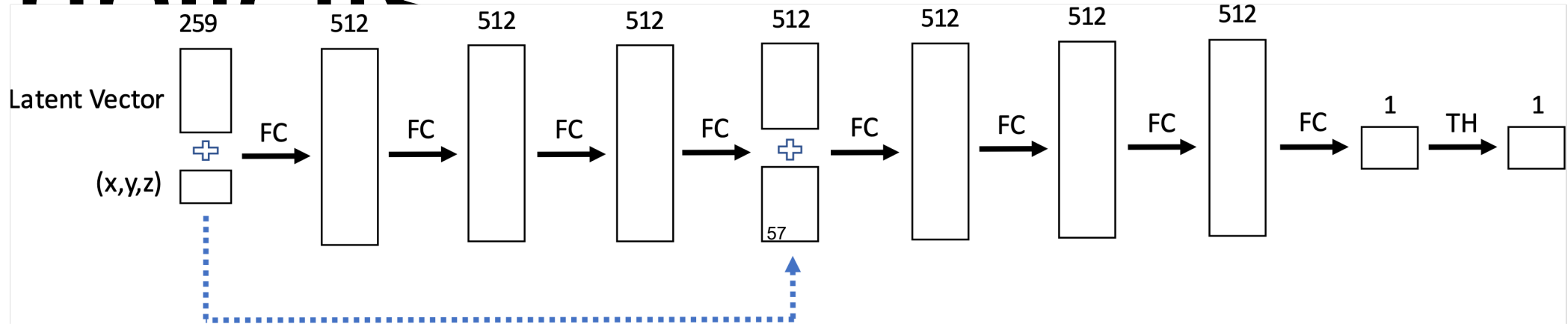
Generalizable implicit field:

Q: What is a good architecture?
(latent code, position) $\rightarrow$

Q: Given some training shapes, how to learn f ?
(implicit function)

Q: How do we infer shapes for new instances?

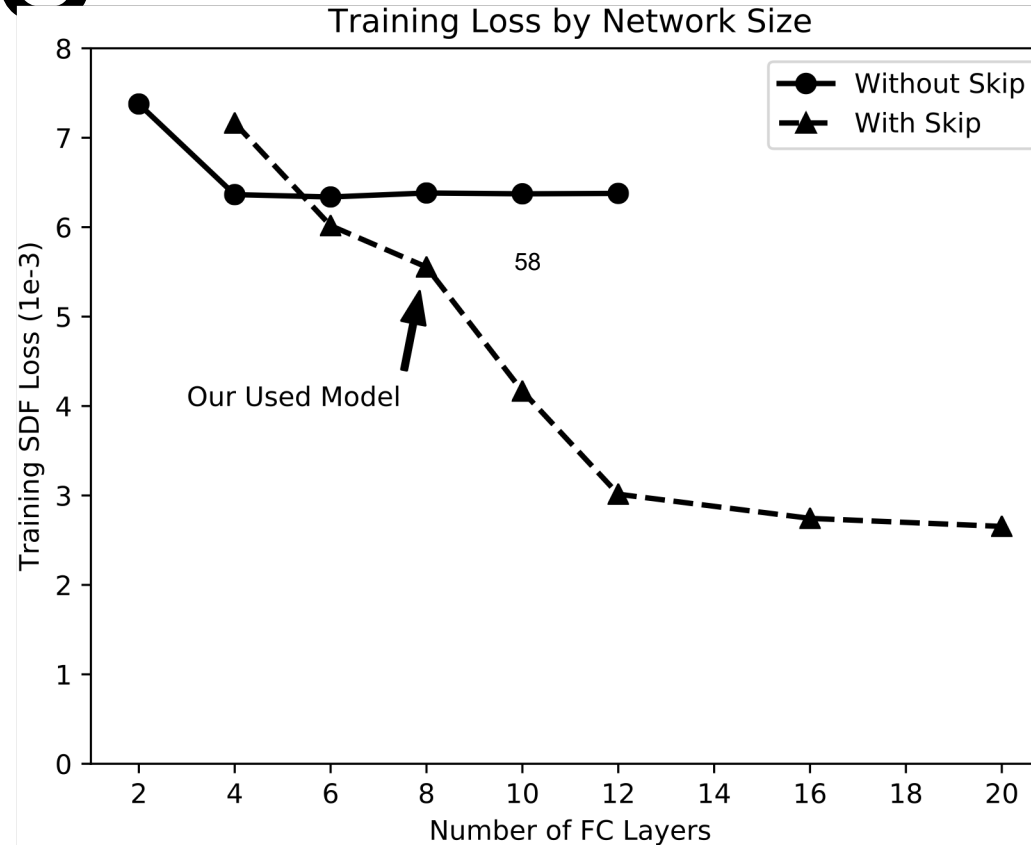
Conditional neural implicit



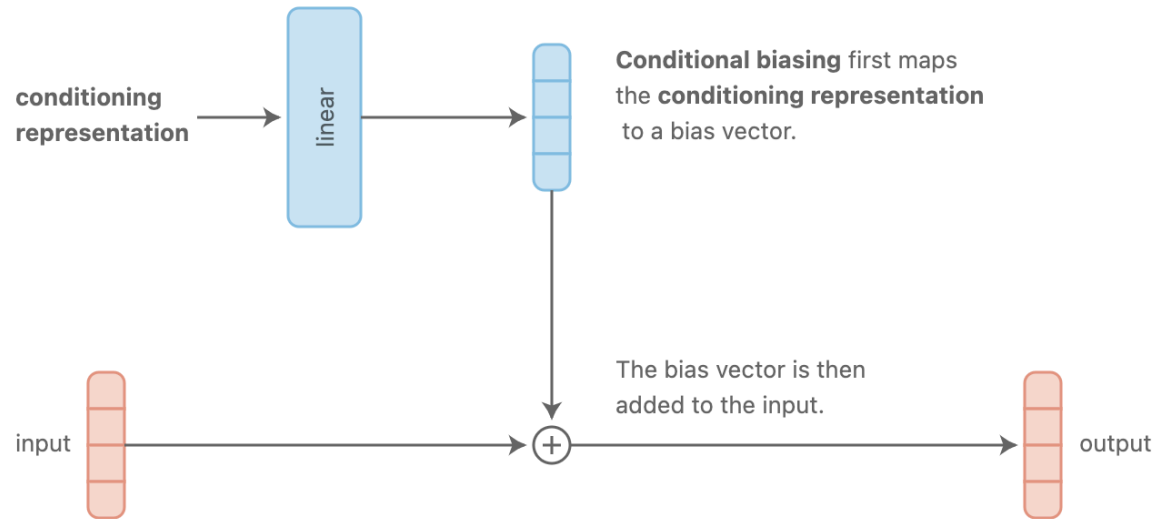
Simplest approach — just concatenate the latent code and the query point

Skip connections for the position help!

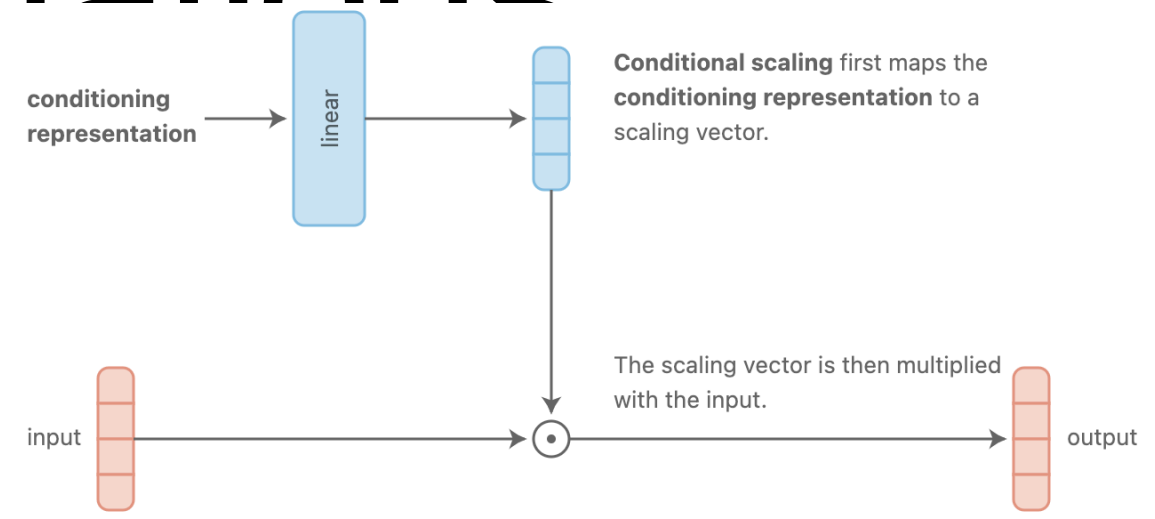
Conditional neural implicits



Conditioning via feature-wise transformations



Additive transformation

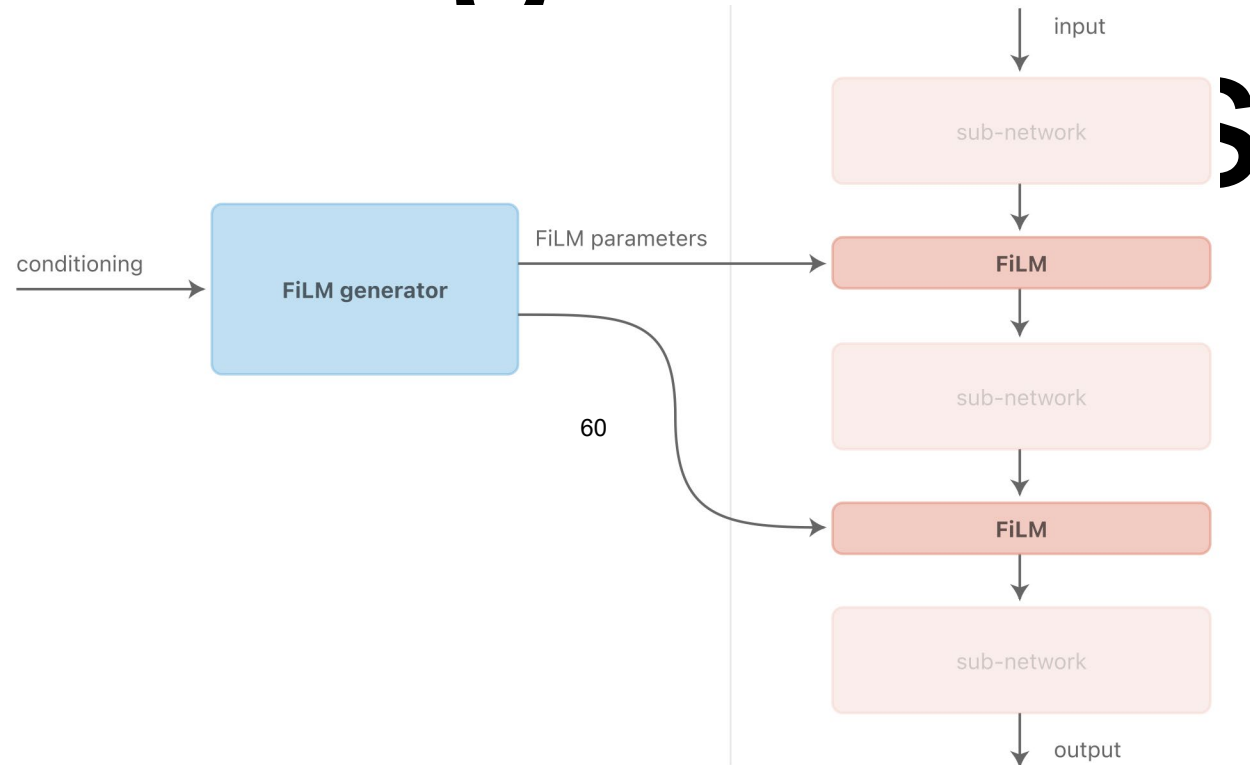


Multiplicative transformation

$$\text{FiLM}(\mathbf{x}) = \gamma(\mathbf{z}) \odot \mathbf{x} + \beta(\mathbf{z})$$

59

Conditioning via feature-wise transformations



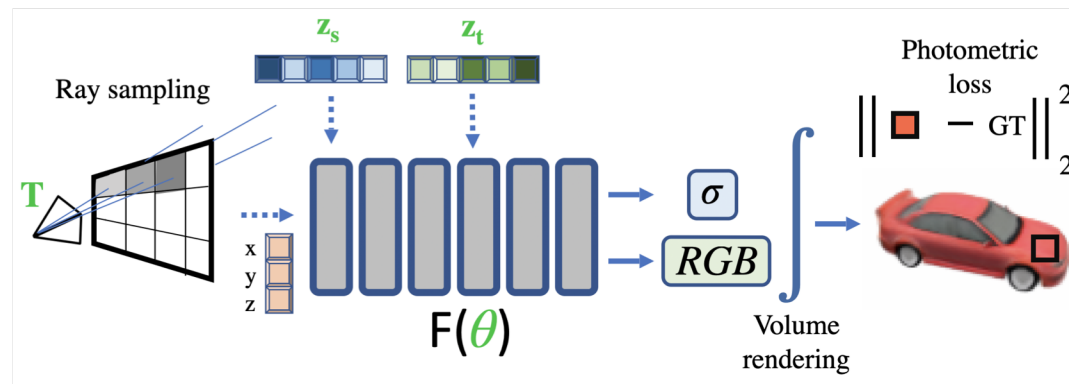
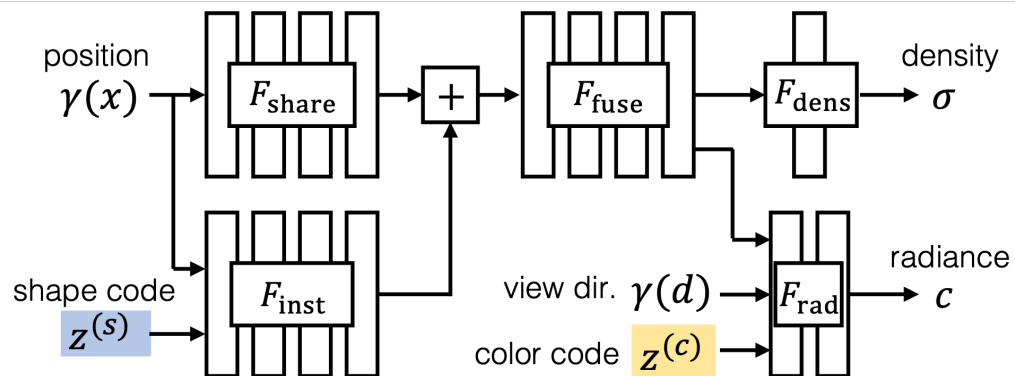
Additive and multiplicative transformations in **multiple layers**

Allows better modeling of conditional detail

Conditional NeRFs

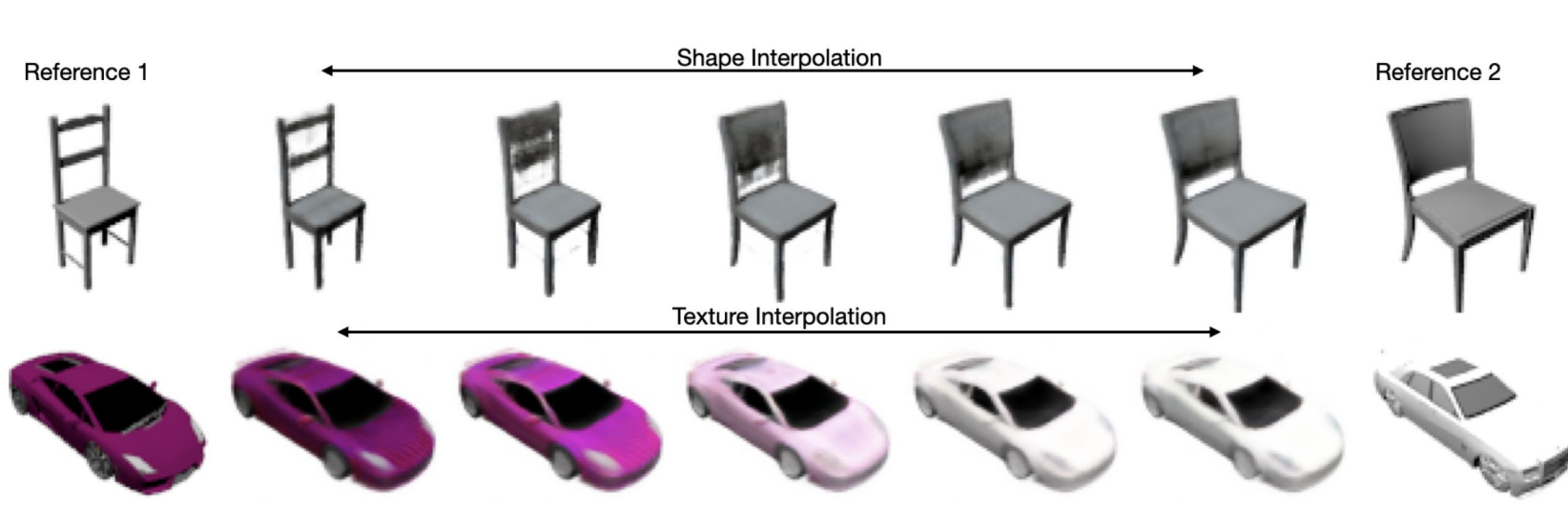
Editing Conditional Radiance Fields. Liu et. al.

CodeNeRF: Disentangled Neural Radiance Fields for Object Categories

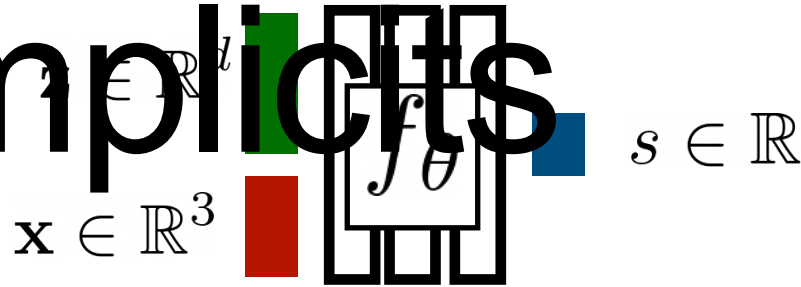


62

Two latent variables—shape and color/texture code



Latent conditioning-based neural implicit



$$f_\theta : \mathbb{R}^3 \times \mathbb{R}^d \rightarrow \mathbb{R}$$

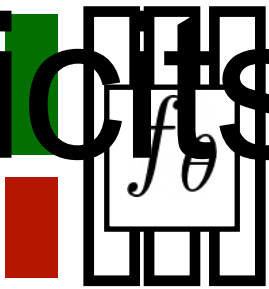
Generalizable implicit field:

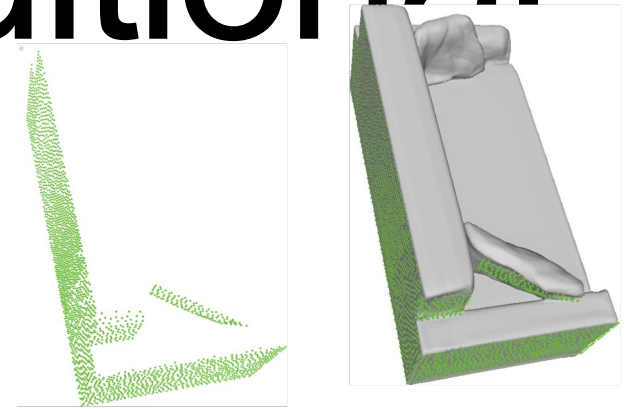
Q: What is a good architecture?
(latent code, position) ->

Q: Given some training shapes, how to learn f ?
(implicit function)

Q: How do we infer shapes for new instances?

Inference with conditional neural implicit

$$\mathbf{x} \in \mathbb{R}^3 \quad \mathbf{z} \in \mathbb{R}^d \quad f_{\theta} \quad s \in \mathbb{R}$$




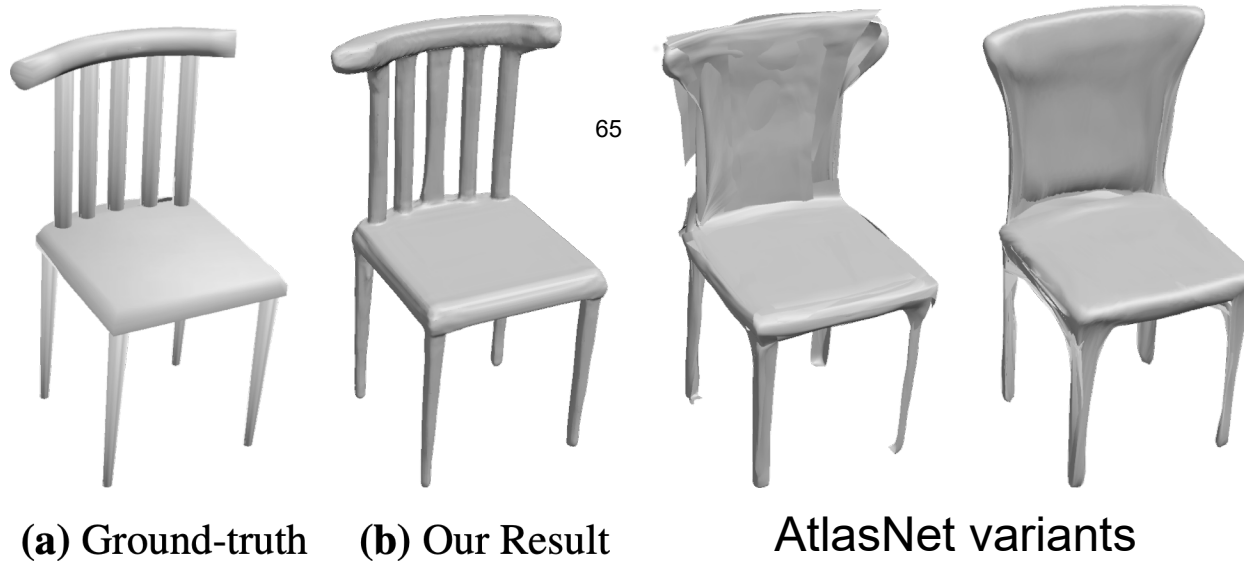
64

$$\hat{\mathbf{z}} = \arg \min_{\mathbf{z}} \sum_{(\mathbf{x}_j, s_j) \in X} \underbrace{\mathcal{L}(f_{\theta}(\mathbf{z}, \mathbf{x}_j), s_j)}_{\text{data term}} + \underbrace{\frac{1}{\sigma^2} \|\mathbf{z}\|_2^2}_{\text{Gaussian prior}}$$

Infer the optimal $\mathbf{z}$ given some observations (points $\mathbf{x}_j$ with known implicit function value s_j)

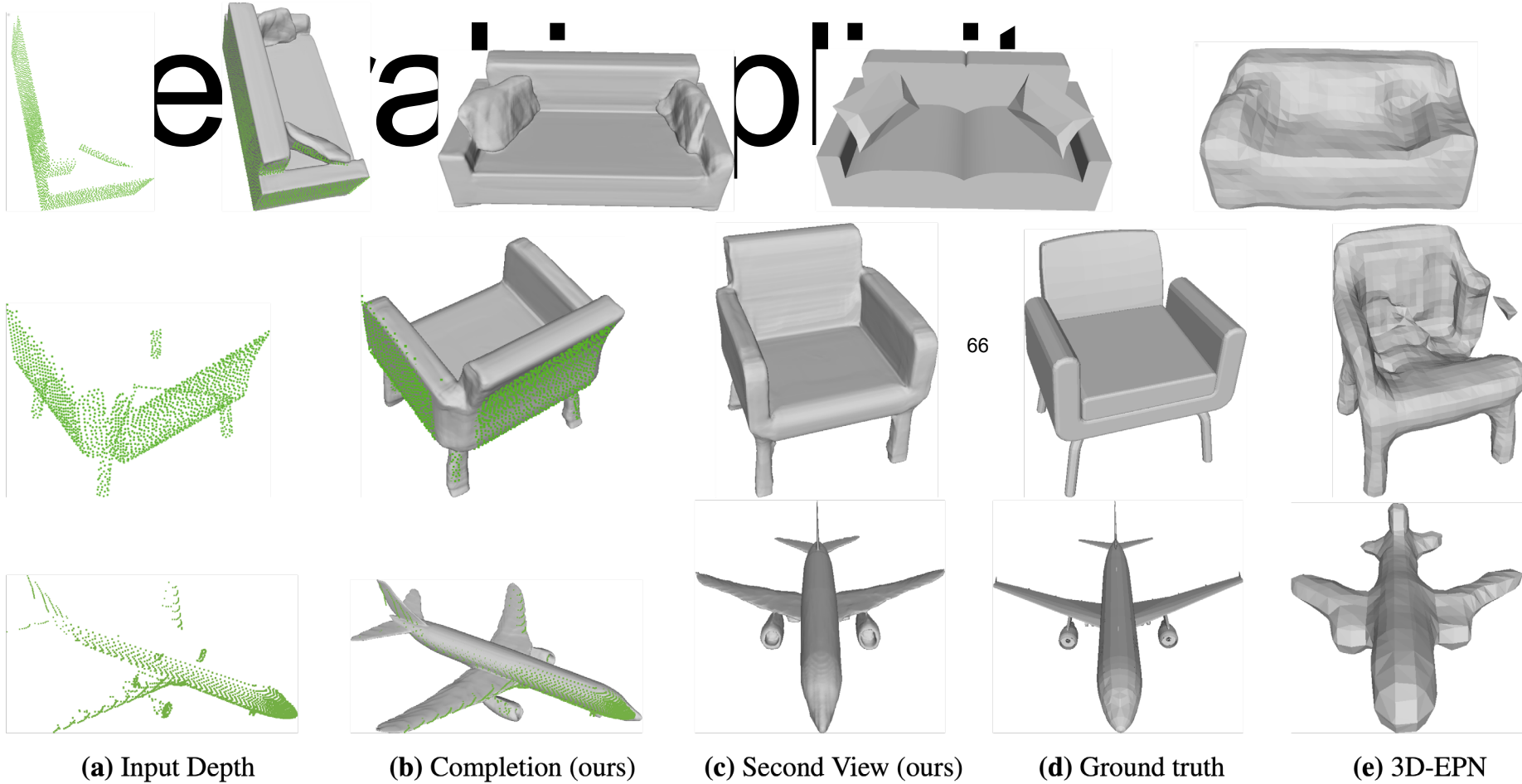
Inference with conditional neural implicit

$$\hat{z} = \arg \min_{z \in \mathcal{Z}} \mathbb{E}_{x_j, s_j \in \mathcal{X}} \mathcal{L}(f_\theta(z, x_j), s_j) + \frac{1}{\sigma^2} \|z\|_2^2$$



Inference using full 3D shape

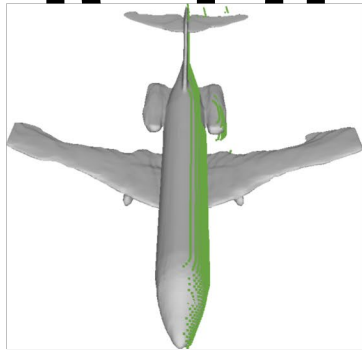
Inference with conditional



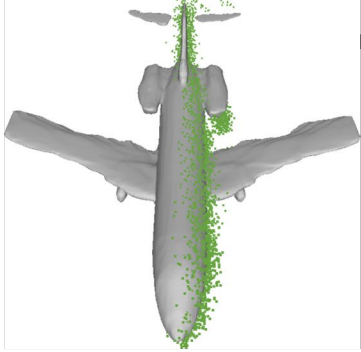
**Q: How to
obtain (x_j, s_j)
pairs from
point
clouds?**

Inference from partial 3D shapes

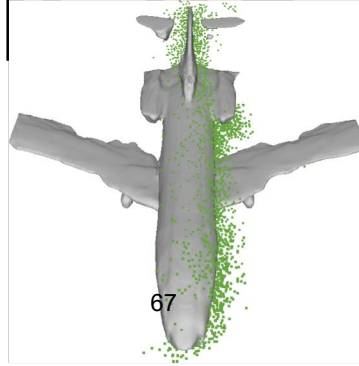
Inference with conditional neural implicit



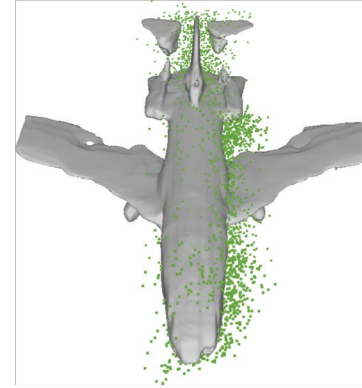
(a) No noise



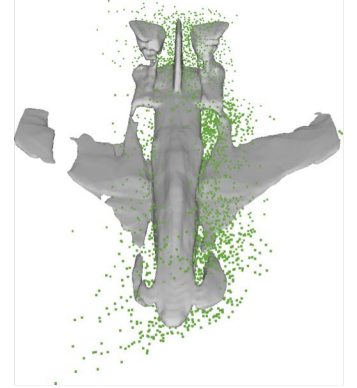
(b) $\alpha = 0.01$



(c) $\alpha = 0.02$



(d) $\alpha = 0.03$

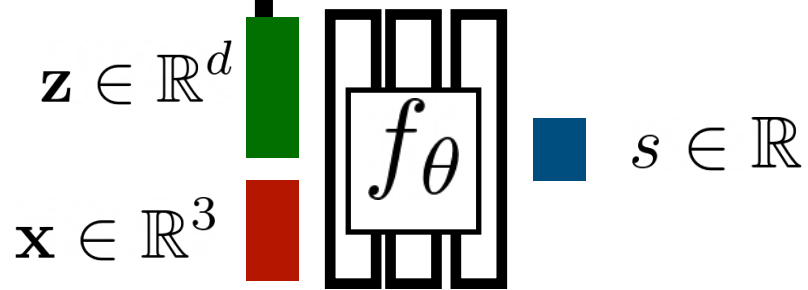


(e) $\alpha = 0.05$

Inference is robust to noise

Allows hallucinating unseen regions

Conditional neural implicit



Training (via auto-
decoding)

$$\arg \min_{\theta, \{\mathbf{z}_i\}_{i=1}^N} \sum_{i=1}^N \left(\sum_{j=1}^K \mathcal{L}(f_\theta(\mathbf{z}_i, \mathbf{x}_j), s_j) + \frac{1}{\sigma^2} \|\mathbf{z}_i\|_2^2 \right)$$

68

Inference:

$$\hat{\mathbf{z}} = \arg \min_{\mathbf{z}} \sum_{(\mathbf{x}_j, s_j) \in X} \mathcal{L}(f_\theta(\mathbf{z}, \mathbf{x}_j), s_j) + \frac{1}{\sigma^2} \|\mathbf{z}\|_2^2$$

Similar objectives at training and inference

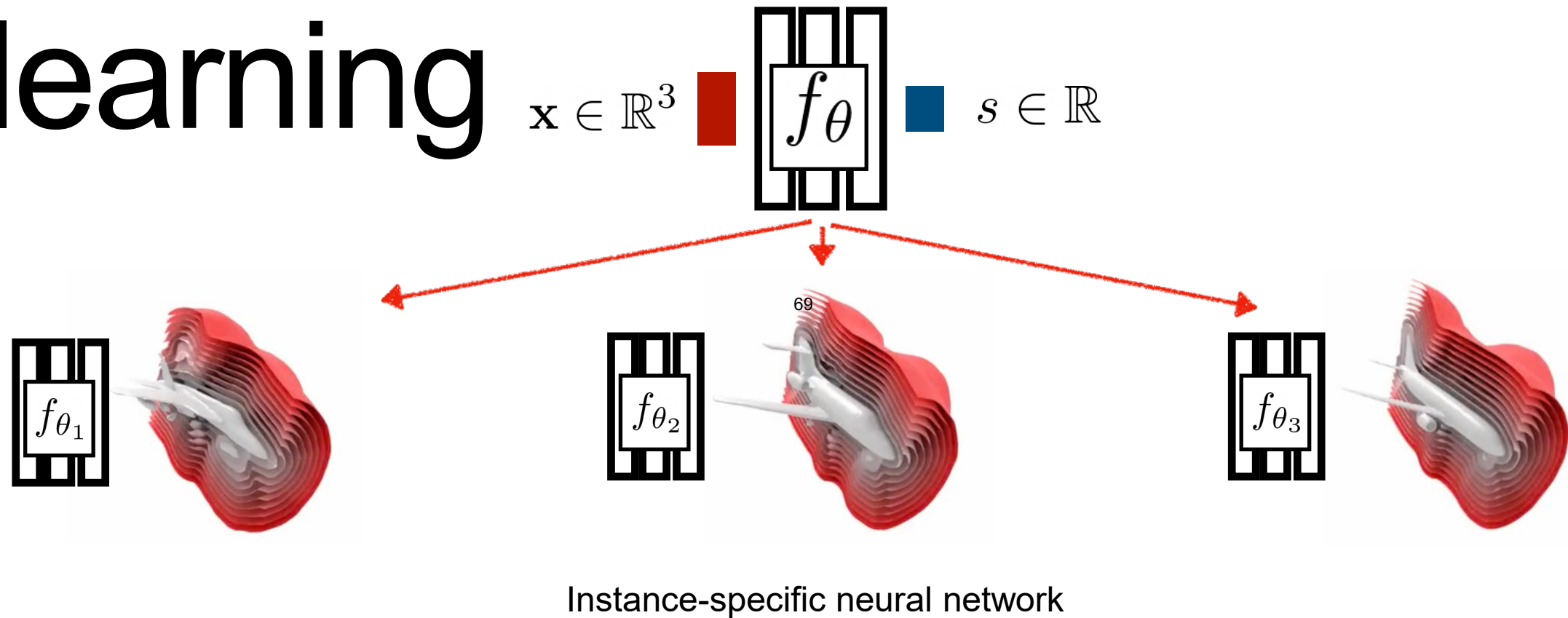
Inference: Only optimize latents

(but a low-dimensional latent vector cannot
arbitrary detail)

Q: Why not also optimize network



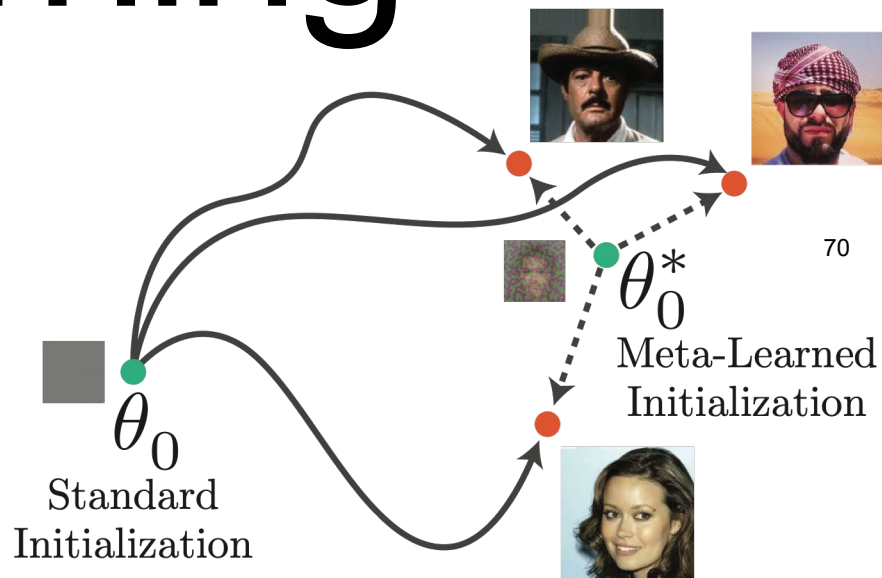
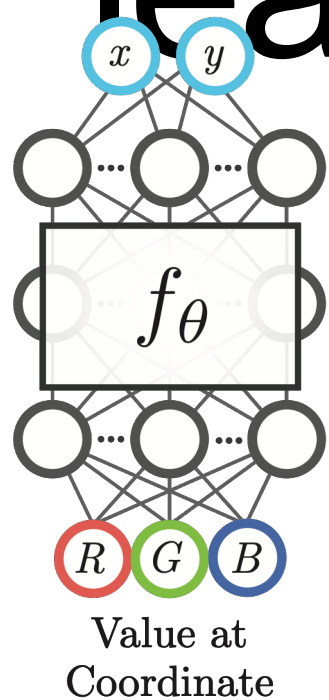
Generalization via meta learning



Key idea: Learn a common ‘meta-network’ s.t. a few gradient steps let it specialize to a new instance

Generalization via meta learning

Input Coordinate



On First-Order Meta-Learning Algorithms

Alex Nichol and Joshua Achiam and John Schulman
OpenAI

{alex, jachiam, joschu}@openai.com

Algorithm 1 Reptile (serial version)

Initialize ϕ , the vector of initial parameters

for iteration = 1, 2, ... **do**

 Sample task τ , corresponding to loss L_τ on weight vectors $\tilde{\phi}$

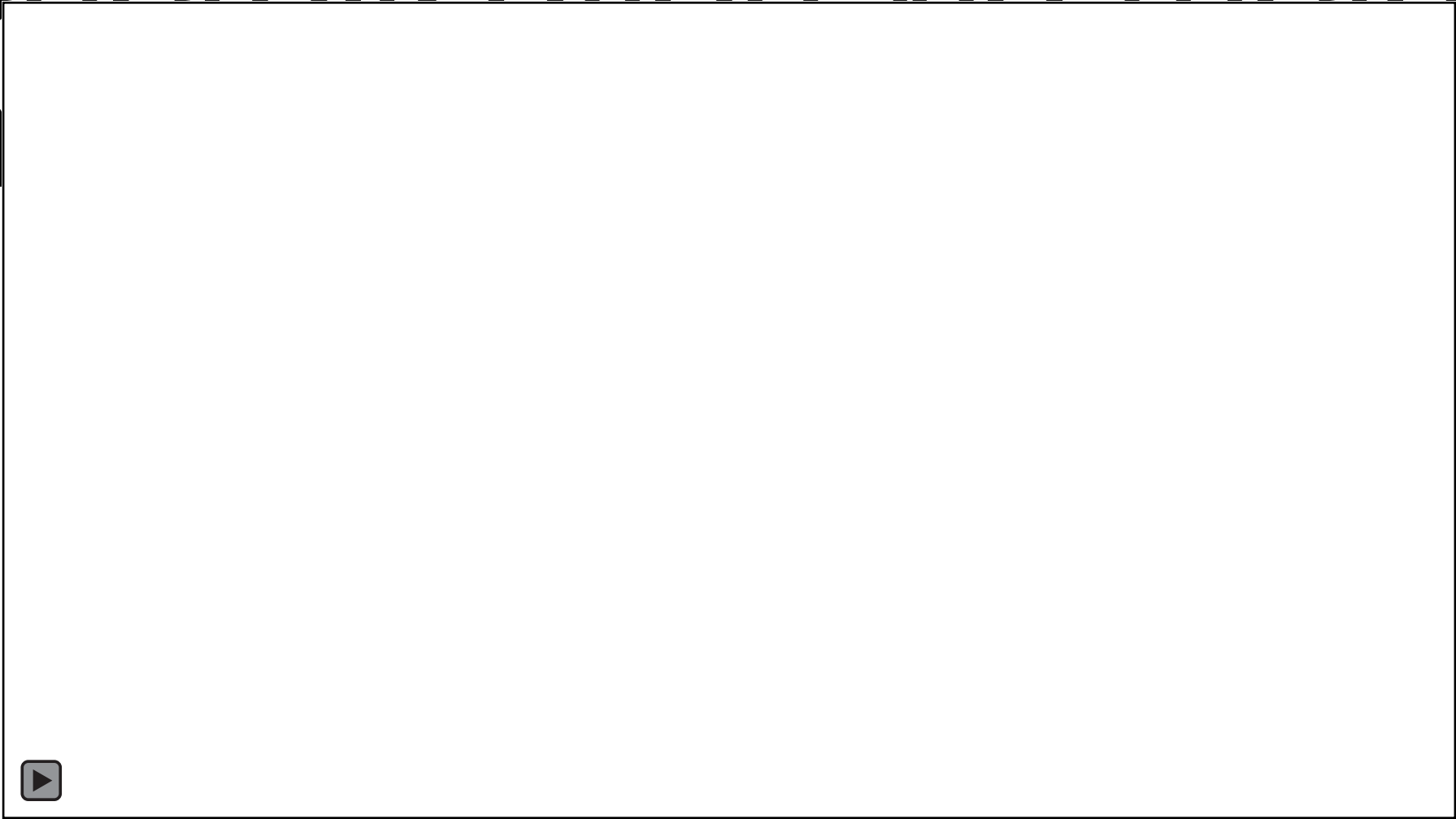
 Compute $\tilde{\phi} = U_\tau^k(\phi)$, denoting k steps of SGD or Adam

 Update $\phi \leftarrow \phi + \epsilon(\tilde{\phi} - \phi)$

end for

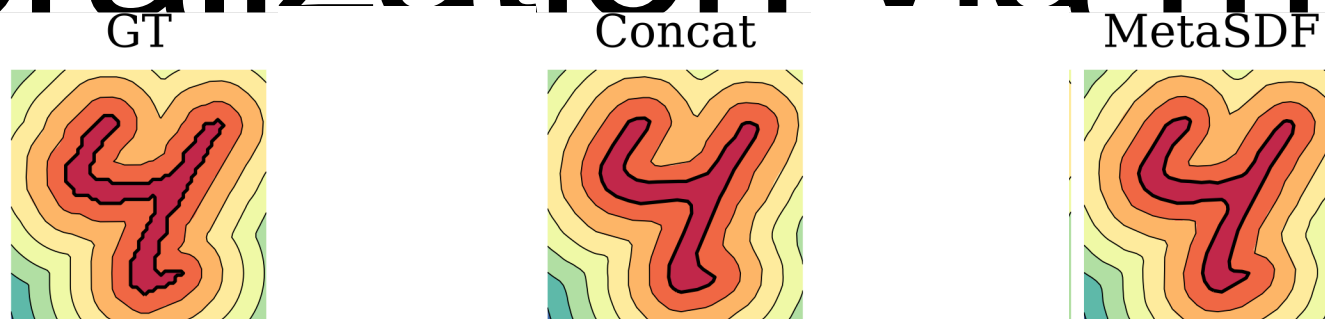
TL;dr: push initialization network's weights
towards instance-specific optimized weights

Generalization via meta learning

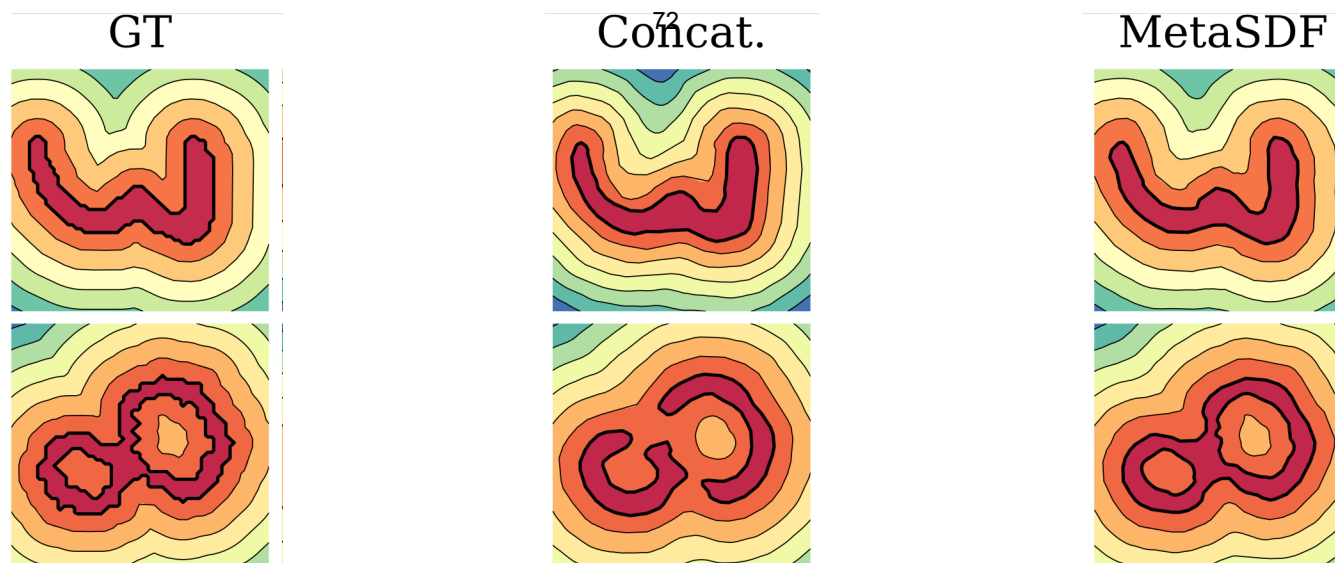


Inference: Similar objective as optimizing conditional neural implicits, but we optimize full network

Generalization via meta learning

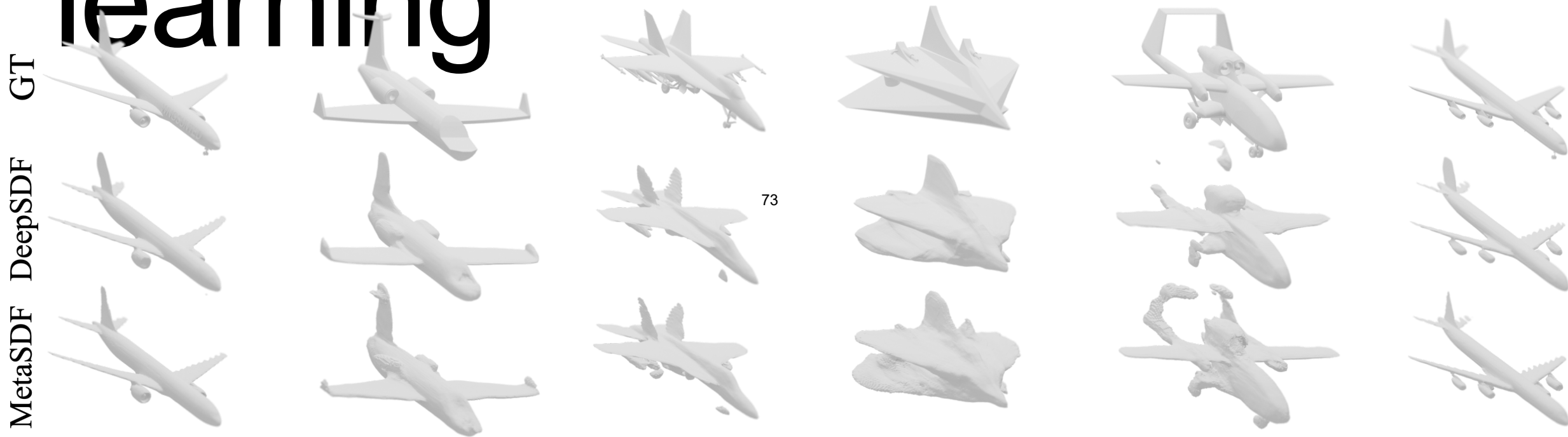


Similar expressivity for in-domain instances



Better at generalization to out-of-domain instances (e.g., rotated digits)

Generalization via meta learning



Similar accuracy, but more efficient optimization as only few SGD steps needed

Meta NeRF: Optimizing

W



Learned Initial

Meta NeRF: Single-view

31



Meta NeRF

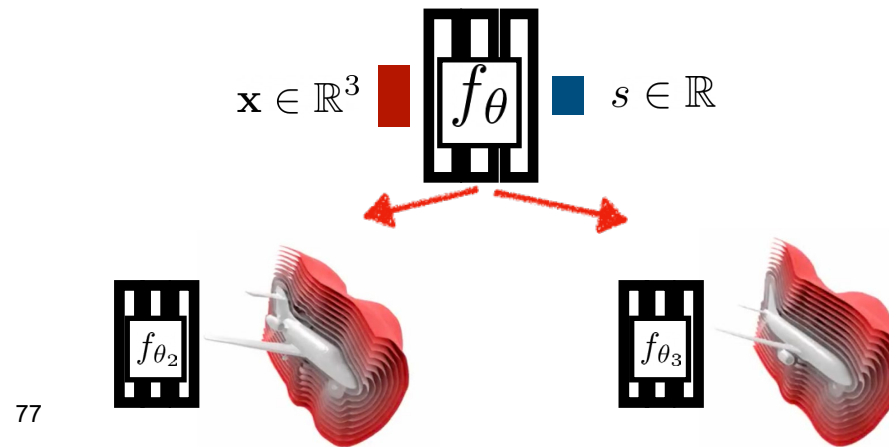
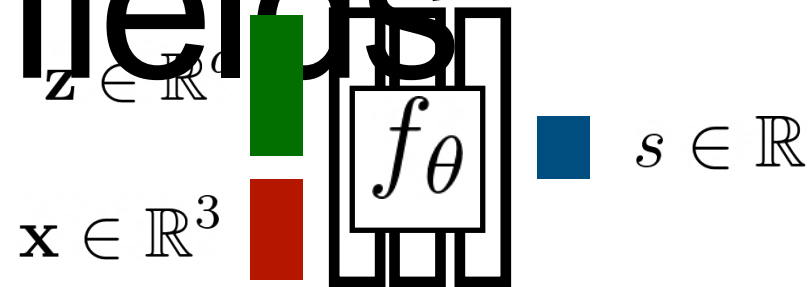


Requires precise camera pose at inference

Per-category meta-network

No view-dependence in appearance (not a 'radiance' field)

Generalizable radiance fields



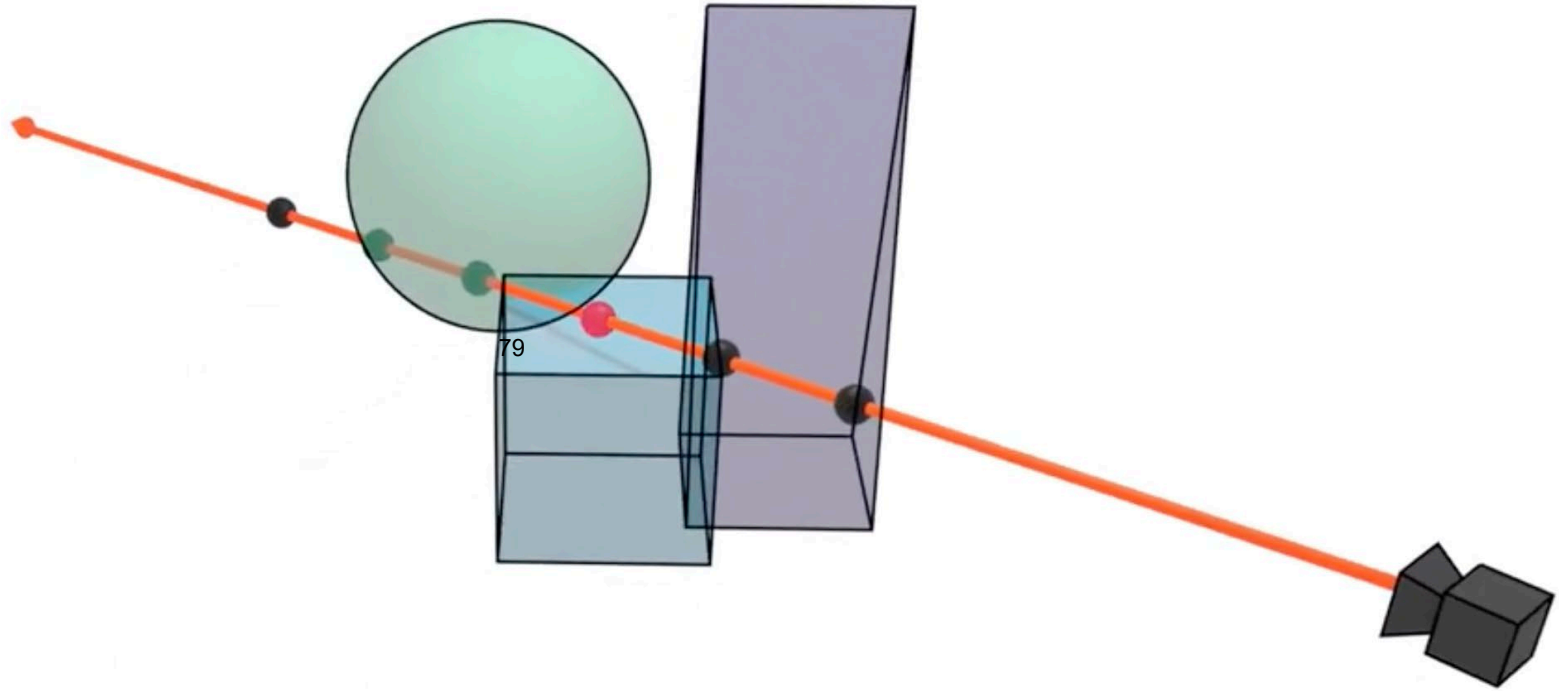
77

Examples of global priors for generalization (e.g., how do chairs and airplanes look?)



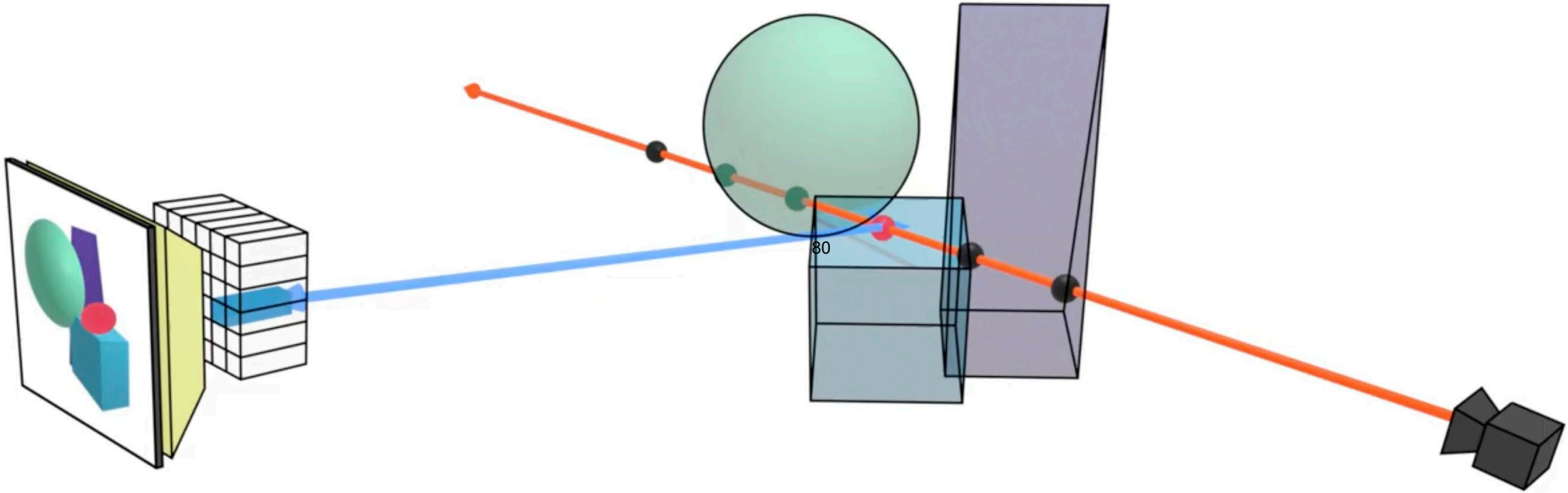
Cannot easily handle large variation (e.g., scenes with multiple objects)

Generalization via pixel-alignnec



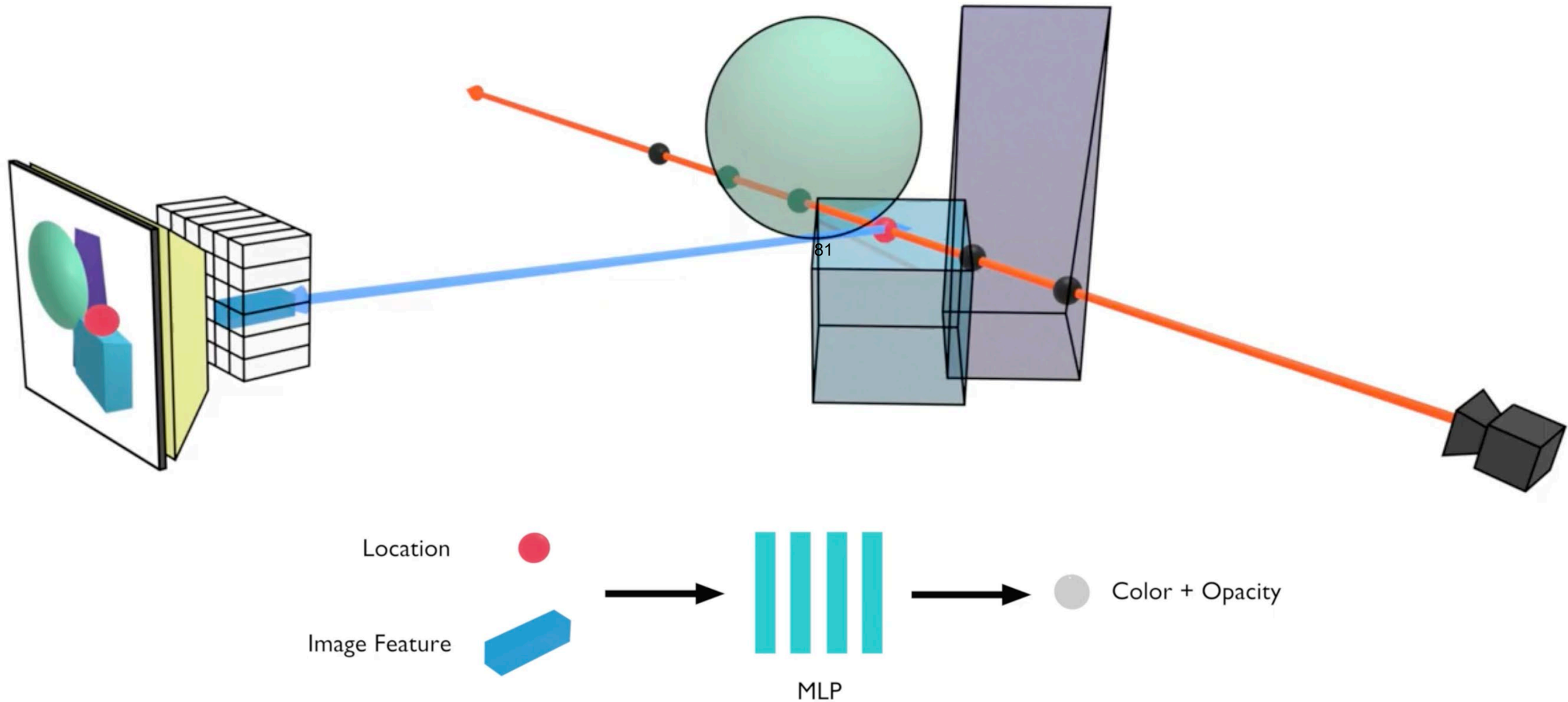
Given a query (unobserved) viewpoint, we need per point density and radiance to render

Generalization via pixel-



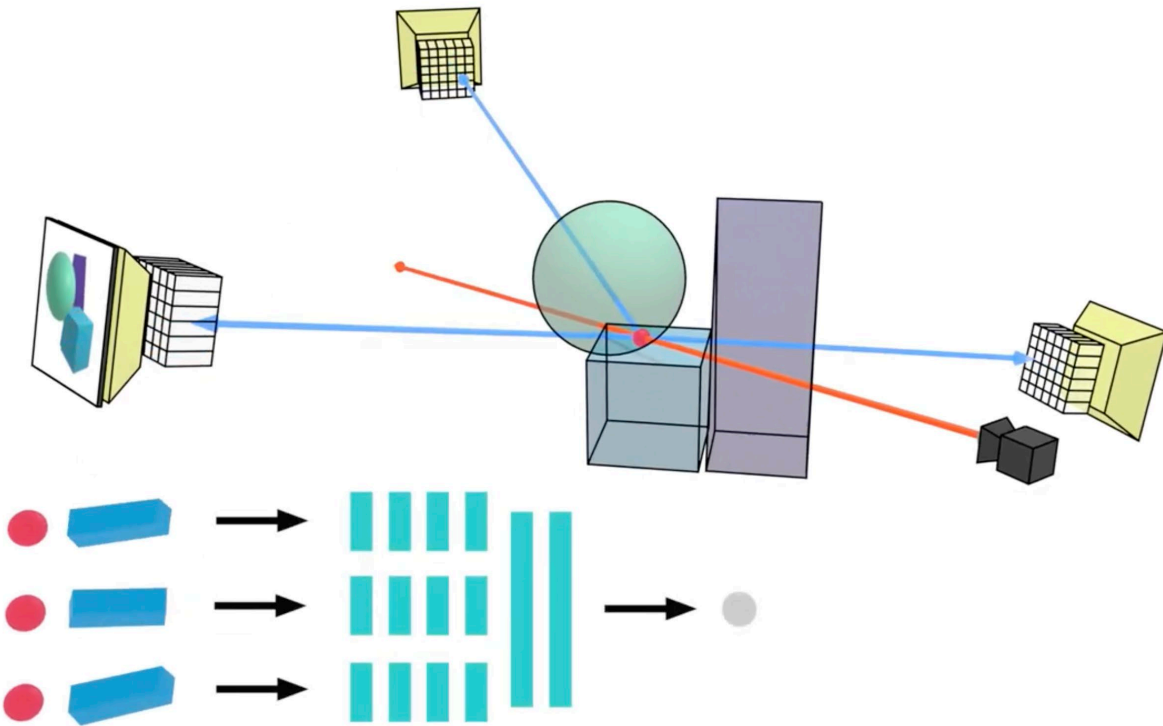
Key idea: Predict using projection onto observed image(s)

Generalization via pixel-



Generalization via pixel-

Training:



Can be extended to multiple input views

82

Inference:



Learn using view-synthesis objectives across many scenes

Rendering **without**
test-time optimization

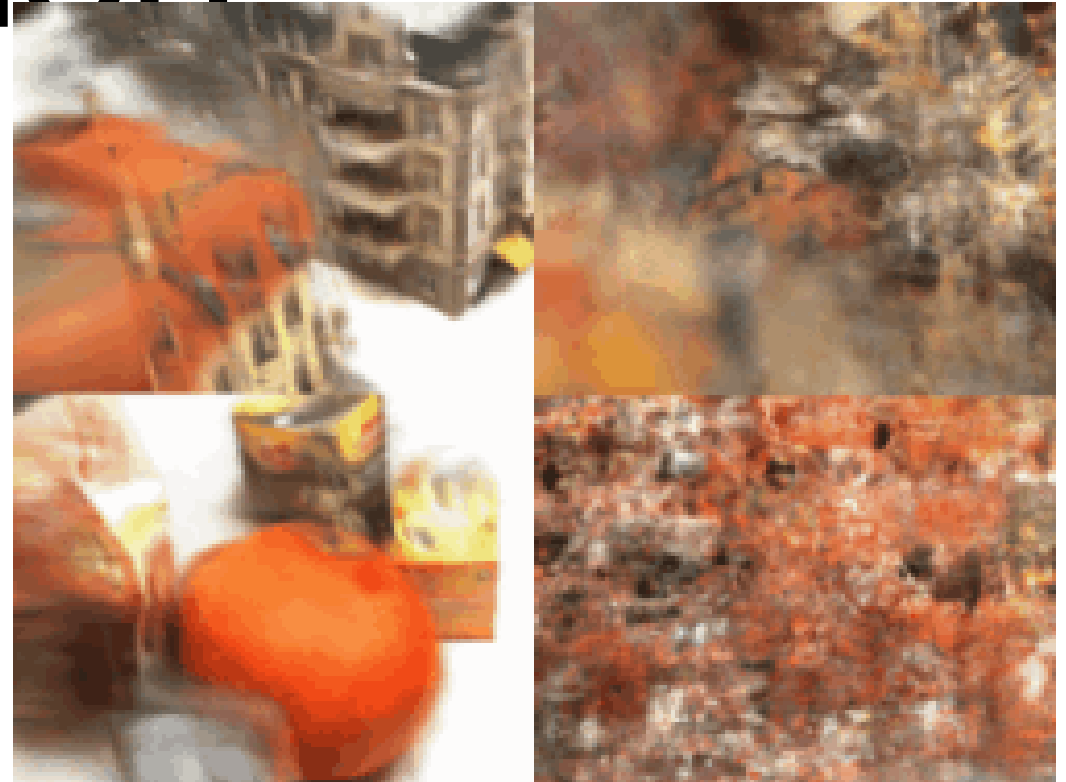
Generalization via pixel-aligned prediction

PixelNeRF

NeRF

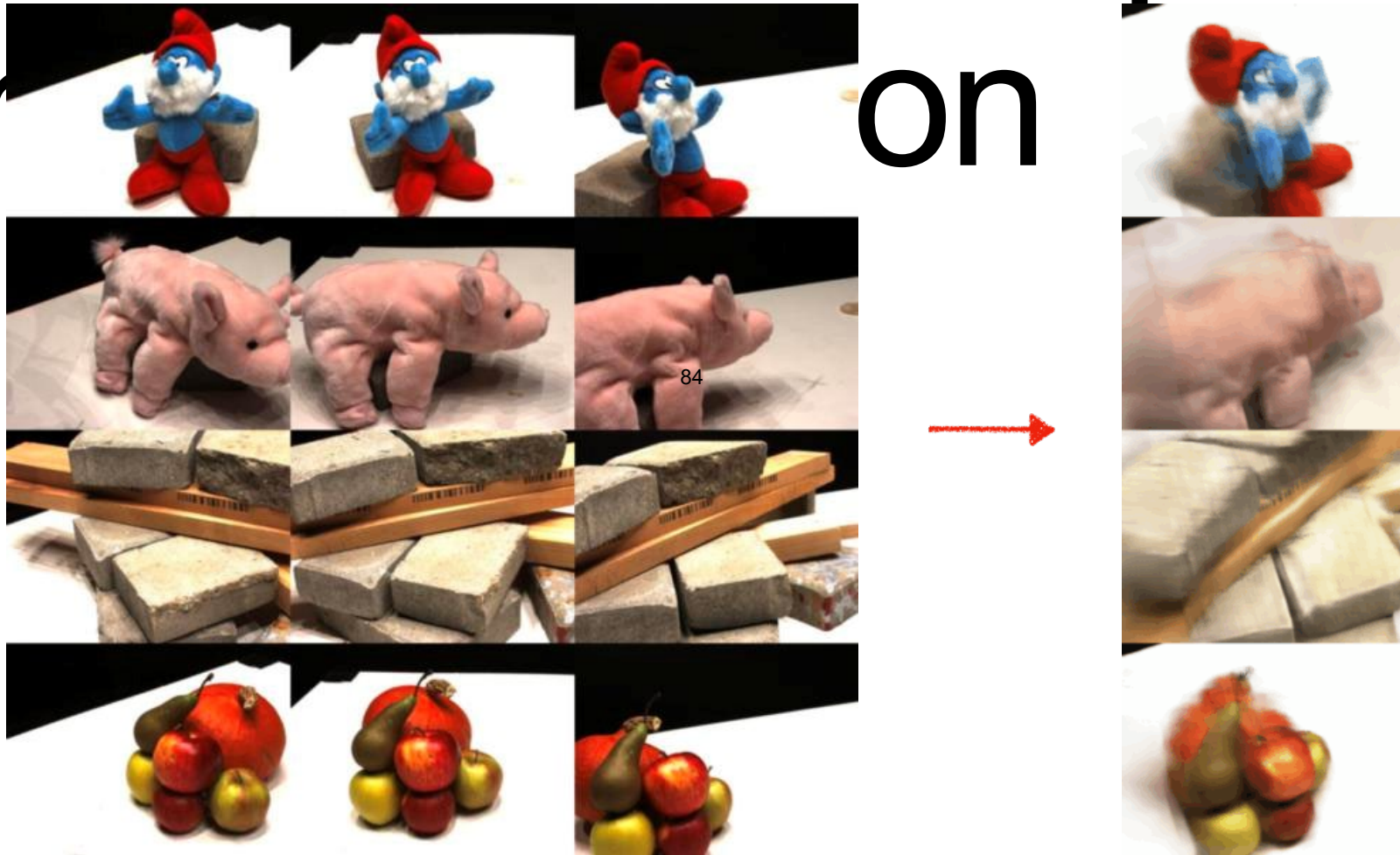


83



Images with precise camera poses

Generalization via pixel- alignment



Missing details in unobserved regions

Generalization via pixel-

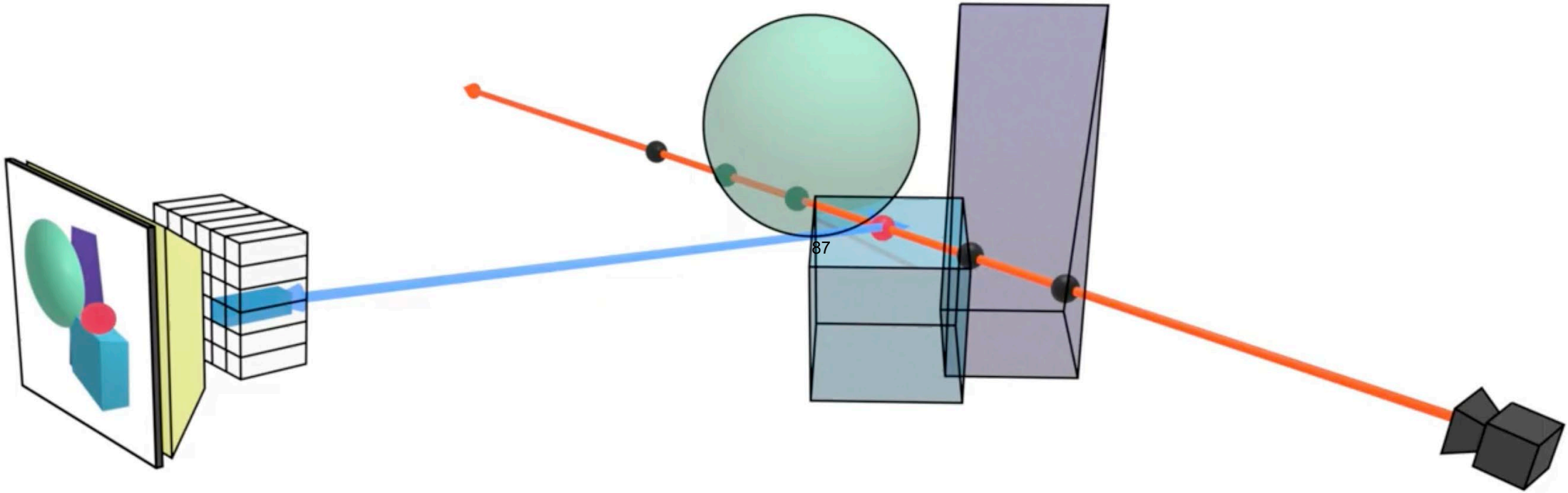


less coherent than global
priors for isolated objects

Generalization via pixel-

Allows generalization for more
complex structures

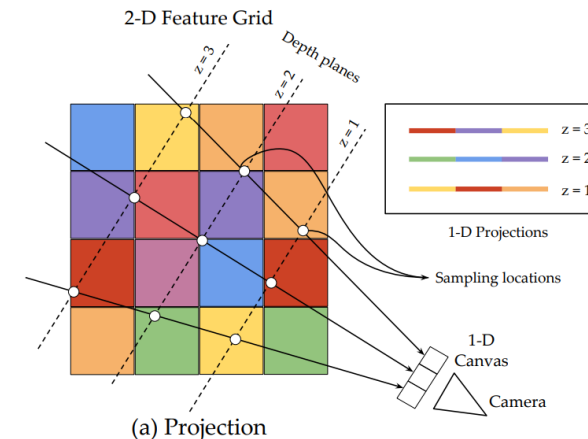
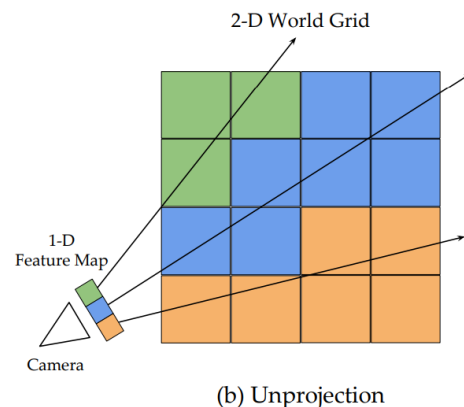
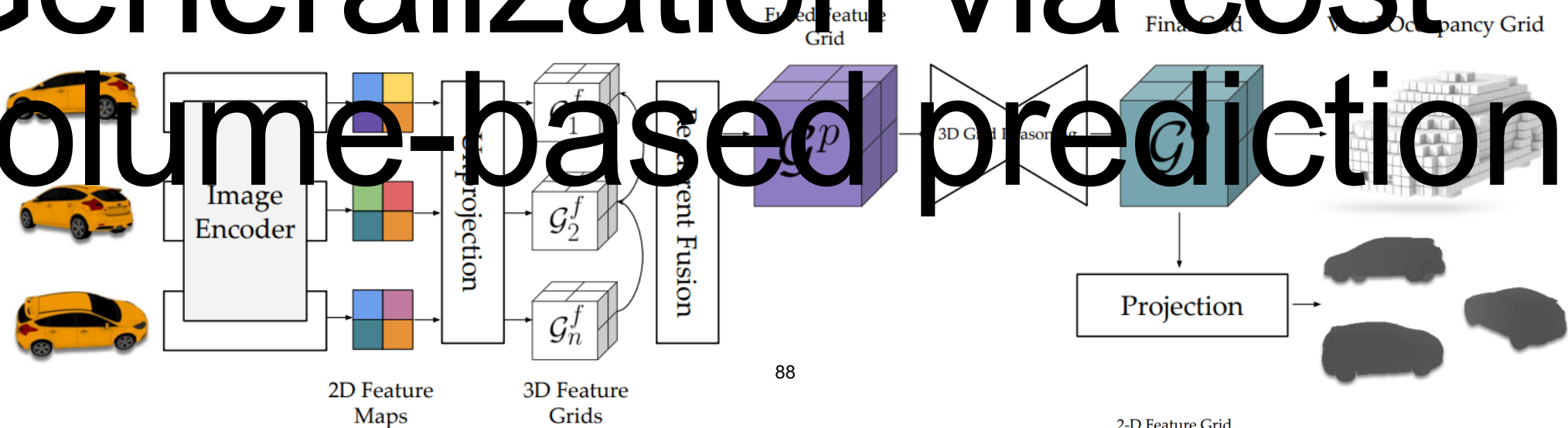
Generalization via pixel-



Key idea: Predict using projection onto observed image(s)

Q: Why does this make sense?

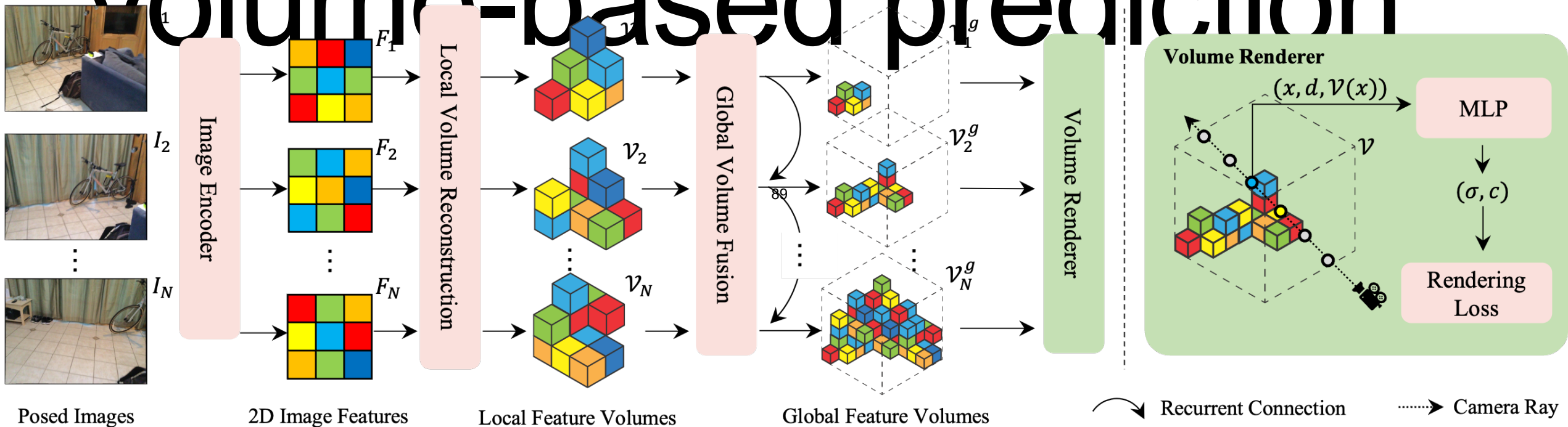
Generalization via cost volume-based prediction



Key idea: Leverage structure of *learned* multi-view stereo cost volume in scene space

- Learning of image feature and fusion features happens at train time on a dataset

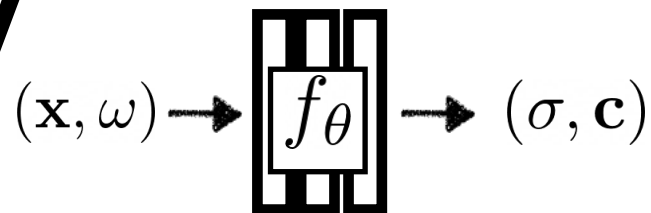
Generalization via cost volume-based prediction



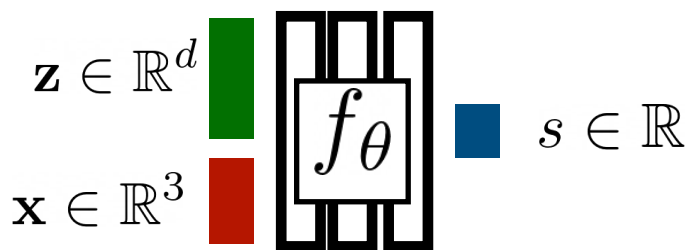
Key idea: Ray trace cost volume and use MLP to predict density and color for rendering

- For instance-specific inference, train only MLP, or use pre-trained MLP.

Summary

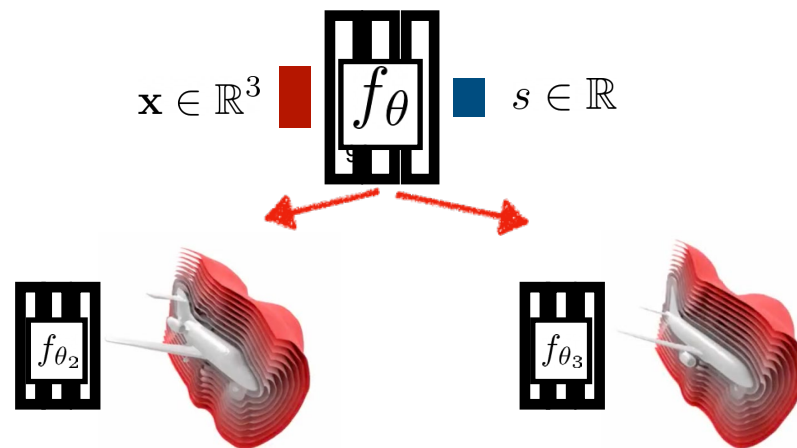


One neural network **per** scene (or object)



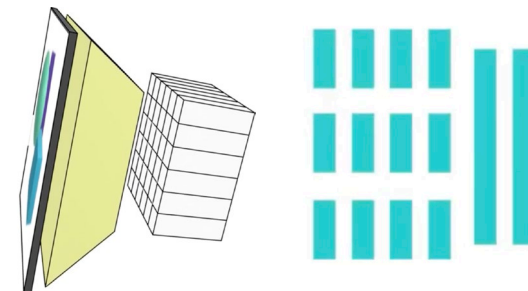
Latent variable conditional model

Inference: Optimize instance-specific latent



Learned initialization for per-scene network

Inference: Optimize entire NN



Learned encoder and predictor

Inference: Feed-forward