# Neural Radiance Field (NERF)

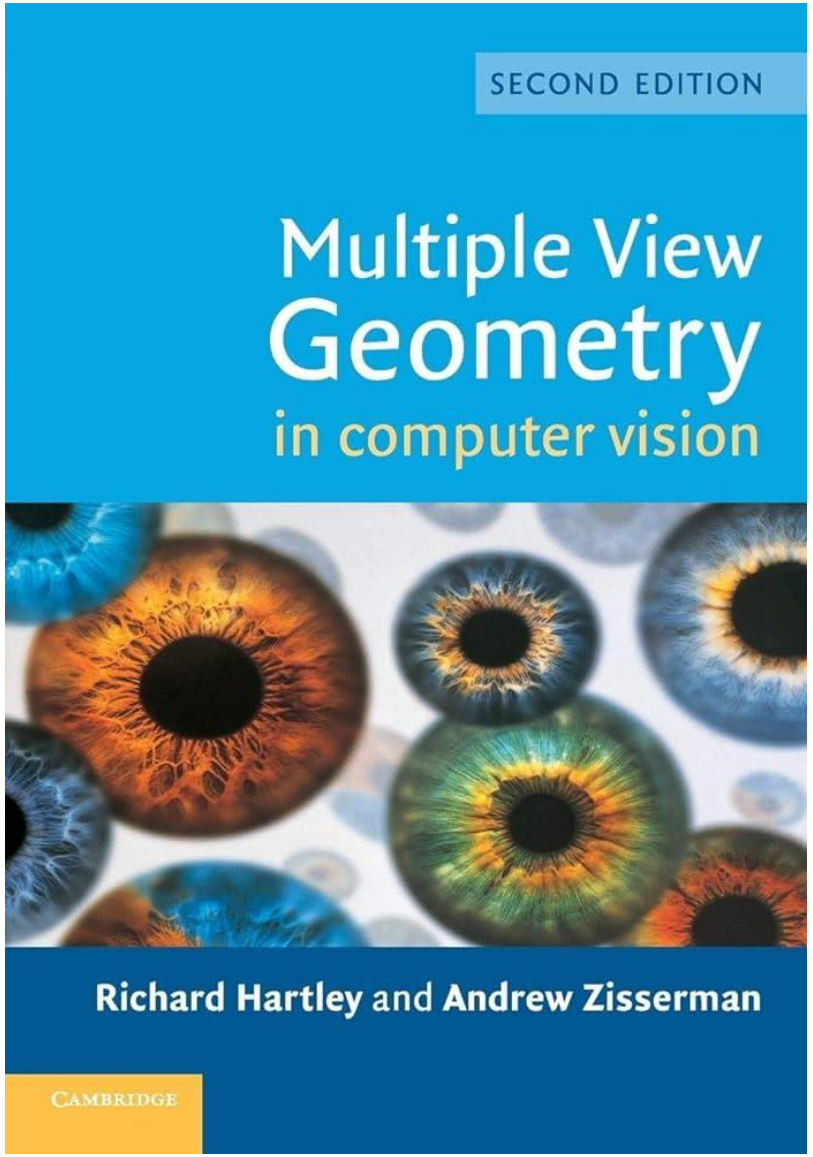
**Chul Min Yeum**
Assistant Professor
Civil and Environmental Engineering
University of Waterloo, Canada

**CIVE 497 – CIVE 700: Smart Structure Technology**

**Last updated: 2024-04-12**

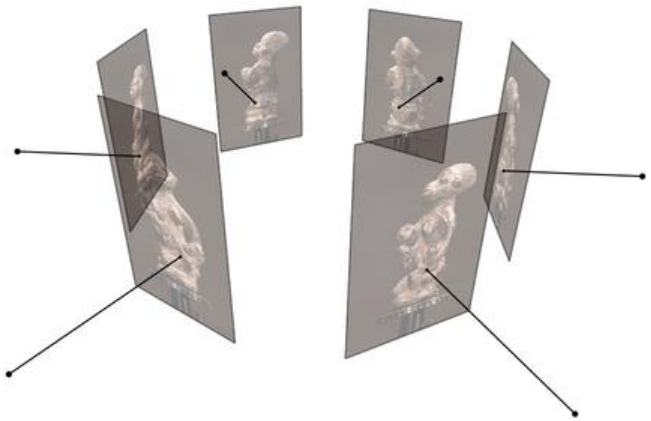
UNIVERSITY OF WATERLOO
FACULTY OF ENGINEERING

# Multi-View Stereo



# Multi-view Stereo (Lots of calibrated images)

- Input: calibrated images from several viewpoints (known camera: intrinsics and extrinsics)
- Output: 3D Model



Figures by Carlos Hernandez

Slide credit: Noah Snavely

In general, conducted in a controlled environment with multi-camera setup that are all calibrated

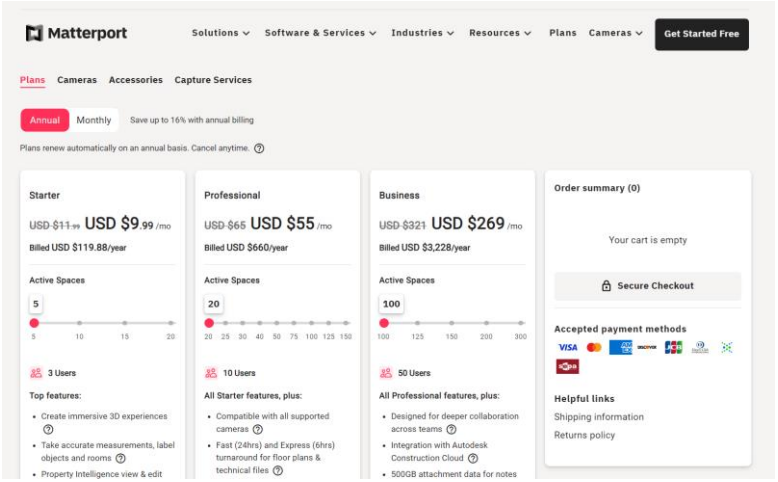
# 3D Reconstruction for Sculptures


Slide credit: Noah Snavely

# What if We Want Solid Models?


Slide credit: Noah Snavely

# Matterport

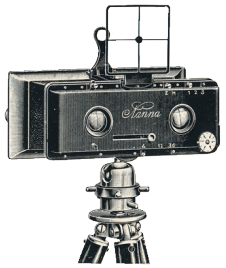

Matterport
Solutions
Software & Services
Industries
Resources
Plans
Cameras
Get Started Free
Plans
Cameras
Accessories
Capture Services
Annual
Monthly
Save up to 16% with annual billing
Plans renew automatically on an annual basis. Cancel anytime.
Starter
USD $11.99 USD $9.99 /mo
Billed USD $119.88/year
Active Spaces
5
3 Users
Top features:
Create immersive 3D experiences
Take accurate measurements, label objects and rooms
Property Intelligence view & edit
Professional
USD $65 USD $55 /mo
Billed USD $660/year
Active Spaces
20
10 Users
All Starter features, plus:
Compatible with all supported cameras
Fast (24hrs) and Express (6hrs) turnaround for floor plans & technical files
Business
USD $321 USD $269 /mo
Billed USD $3,228/year
Active Spaces
100
50 Users
All Professional features, plus:
Designed for deeper collaboration across teams
Integration with Autodesk Construction Cloud
500GB attachment data for notes
Order summary (0)
Your cart is empty
Secure Checkout
Accepted payment methods
Helpful links
Shipping information
Returns policy


matterport


Matterport™

# Multi-view Stereo

Problem formulation: given several images of the same object or scene, compute a representation of its 3D shape



Binocular Stereo

Multi-view stereo

Slide credit: Noah Snavely

# Examples: Panoptic Studio

## Panoptic Studio: A Massively Multiview System for Social Interaction Capture

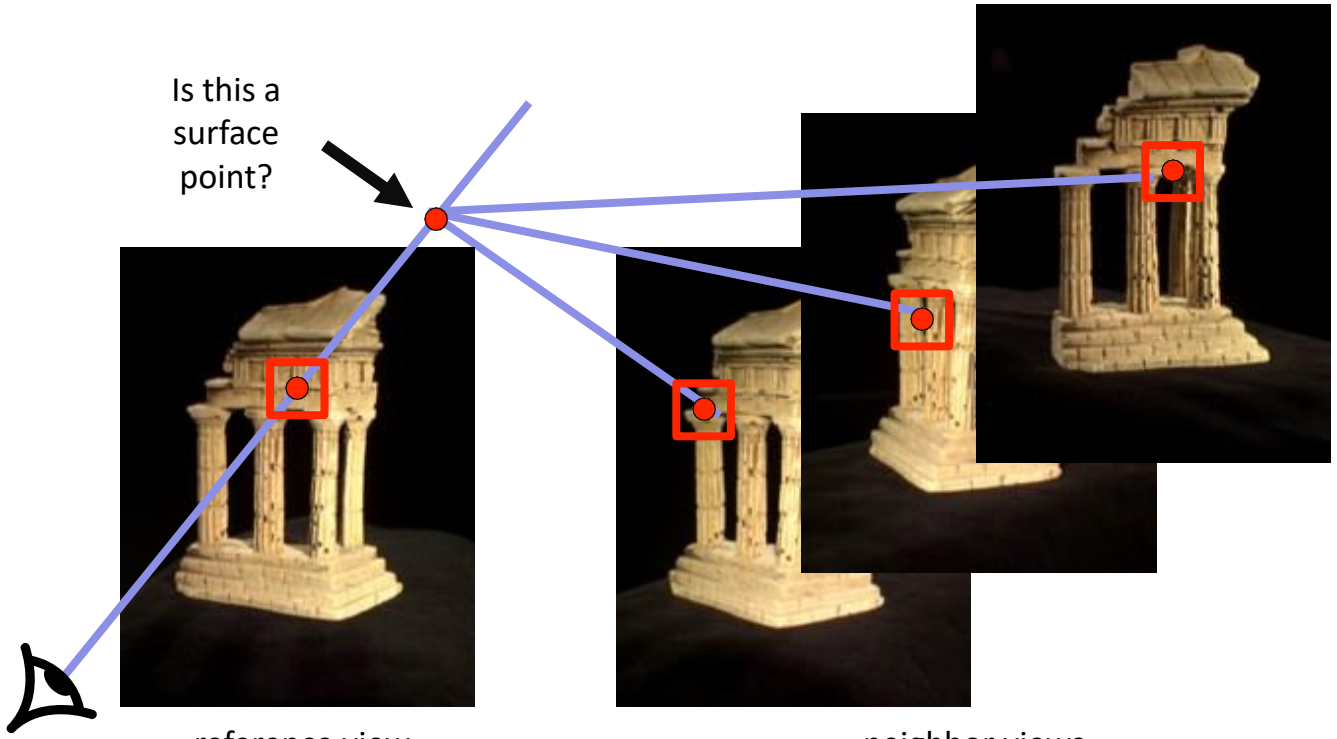
Hanbyul Joo, Tomas Simon, Xulong Li, Hao Liu, Lei Tan, Lin Gui,
Sean Banerjee, Timothy Godisart, Bart Nabbe, Iain Matthews,
Takeo Kanade, Shohei Nobuhara, and Yaser Sheikh

http://domedb.perception.cs.cmu.edu

The Robotics Institute
Carnegie Mellon University

http://domedb.perception.cs.cmu.edu/

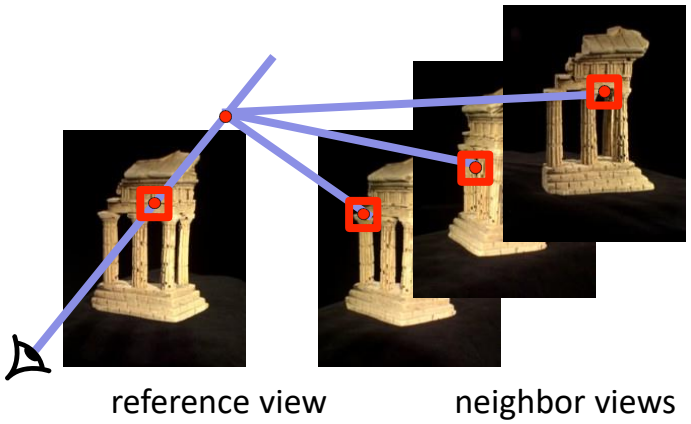
# Multi-view Stereo: Basic idea

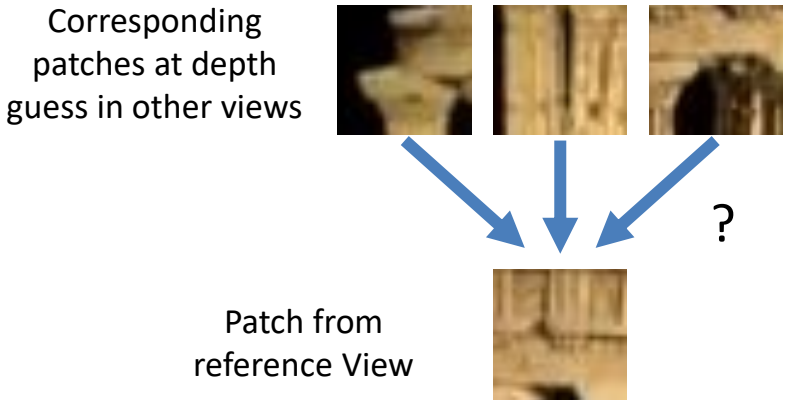



Source: Y. Furukawa

# Multi-view Stereo: Basic idea

Evaluate the likelihood of geometry at a particular depth for a particular reference patch:

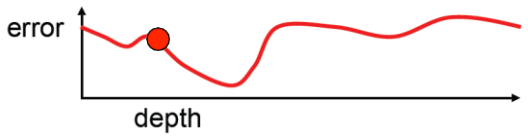



Source: Y. Furukawa

# Multi-view Stereo: Basic idea

Photometric error across different depths





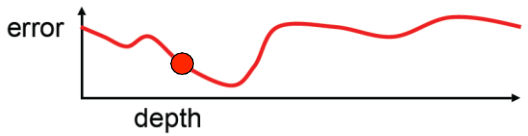

reference view

neighbor views

Source: Y. Furukawa

# Multi-view Stereo: Basic idea

Photometric error across different depths





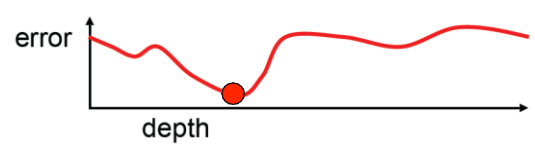

reference view

neighbor views

Source: Y. Furukawa

# Multi-view Stereo: Basic idea

Photometric error across different depths






reference view

neighbor views

In this manner, solve for a depth map over the whole reference view

# Multi-view Stereo: Advantages over Two View

- Can match windows using more than 1 other image, giving a **stronger match signal**
- If you have lots of potential images, can **choose the best subset** of images to match per reference image
- Can reconstruct a depth map for each reference frame, and the merge into a **complete 3D model**

Source: Y. Furukawa

# Tools for SfM

## Commercial Software

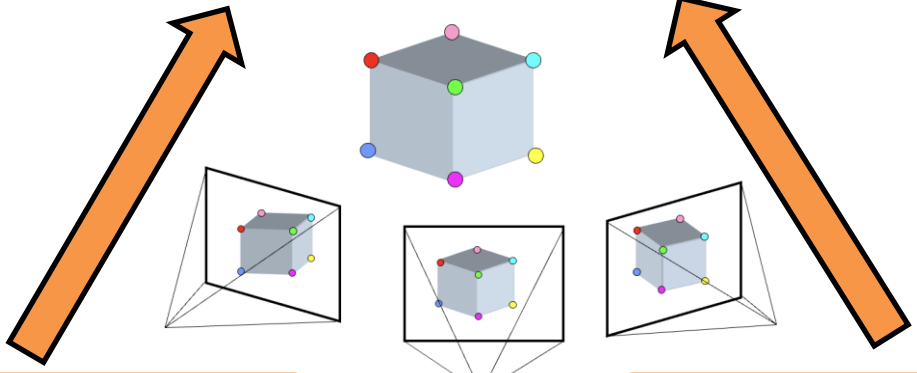
- **Agisoft**
- **Pix4D**
- RealityCapture
- ReCap Pro
- ContextCapture
- 3DF Zephyr
- Coorelator3D
- Trimble Inpho
- Matterport
- Drone Deploy
- **PhotoModeler**
- SURE
- One3D

## Open Source Software

- OpenDroneMap
- **VisualSFM**
- **OpenMVG**
- Bundler
- **COLMAP**
- Regard3D
- Satellite Stereo Pipeline (S2P)
- Danesfield
- Automated 3D Model Pipeline
- OpenSFM
- Clustering Views for Multi-view Stereo (CMVS)
- Multi-View Environment (MVE)

https://medium.com/@krinadhimar/photogrammetry-commercial-and-open-source-tools-and-software-399c456f682d

# Multi-View Stereo



3D Points
(Structure)

Correspondences

Camera
(Motion)

**Observation**

$$\begin{matrix} \tilde{\mathbf{x}}_1^1 & \tilde{\mathbf{x}}_1^2 & \\ \tilde{\mathbf{x}}_2^1 & \tilde{\mathbf{x}}_2^2 & \tilde{\mathbf{x}}_2^3 \\ \tilde{\mathbf{x}}_3^1 & & \tilde{\mathbf{x}}_3^3 \end{matrix}$$

**Features matching**

=

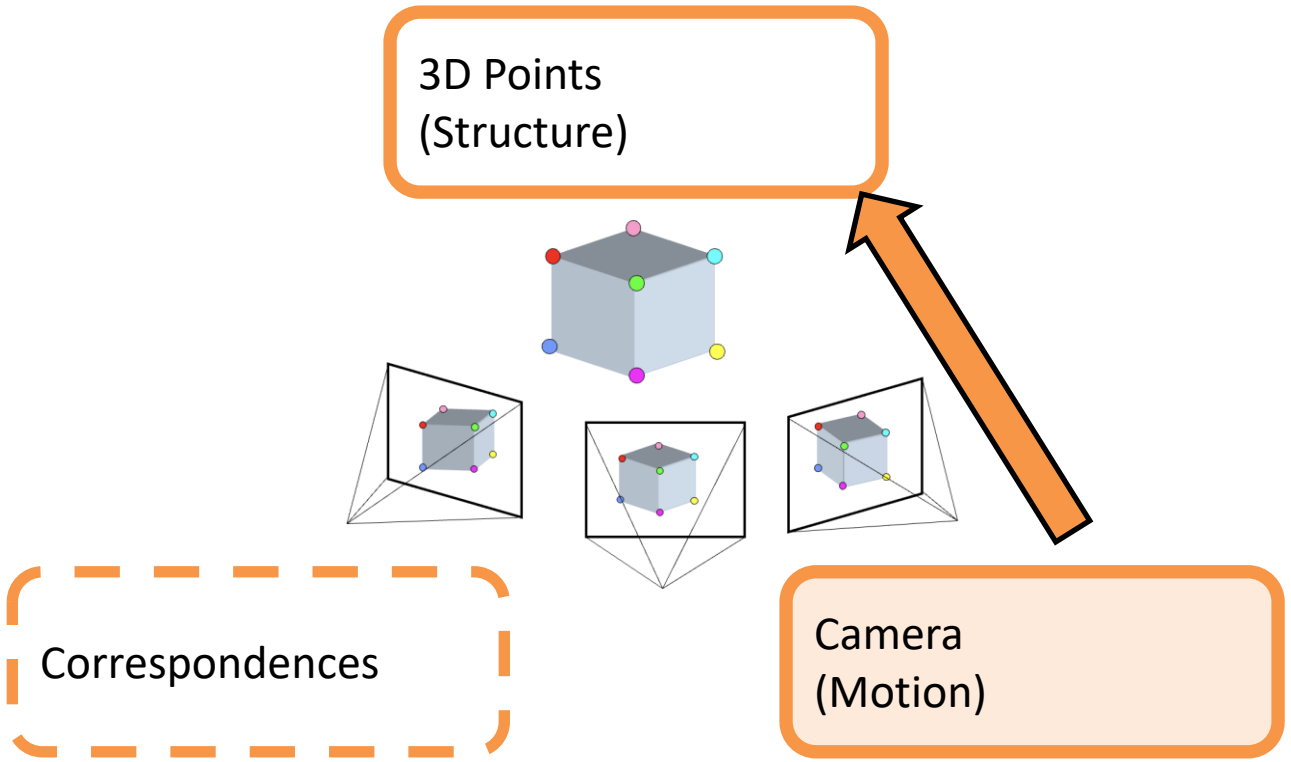
**Re-projection**

| | | | |
|---|---|---|---|
| $\mathbf{x}_1^1 = \mathbf{P}_1\mathbf{X}^1$ | | ⋯ | $\mathbf{x}_1^M = \mathbf{P}_1\mathbf{X}^M$ |
| | $\mathbf{x}_2^2 = \mathbf{P}_2\mathbf{X}^2$ | ⋯ | $\mathbf{x}_2^M = \mathbf{P}_2\mathbf{X}^M$ |
| ⋮ | ⋮ | ⋱ | ⋮ |
| $\mathbf{x}_N^1 = \mathbf{P}_N\mathbf{X}^1$ | $\mathbf{x}_N^2 = \mathbf{P}_N\mathbf{X}^2$ | ⋯ | $\mathbf{x}_N^M = \mathbf{P}_N\mathbf{X}^M$ |

$$\min \sum_i \sum_j \left( \tilde{\mathbf{x}}_i^j - \mathbf{P}_i \mathbf{X}^j \right)^2$$

**Optimization problem**

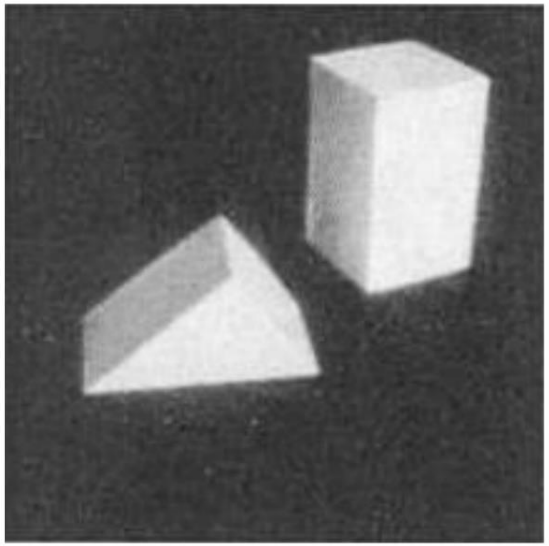
# Volumetric “Neural” Rendering



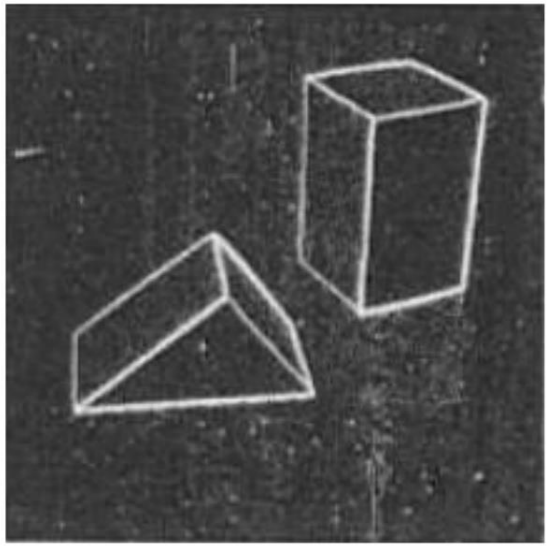

Does not use explicit correspondences,
relies on reconstruction loss (Analysis-by-Synthesis)

# Analysis-by-Synthesis

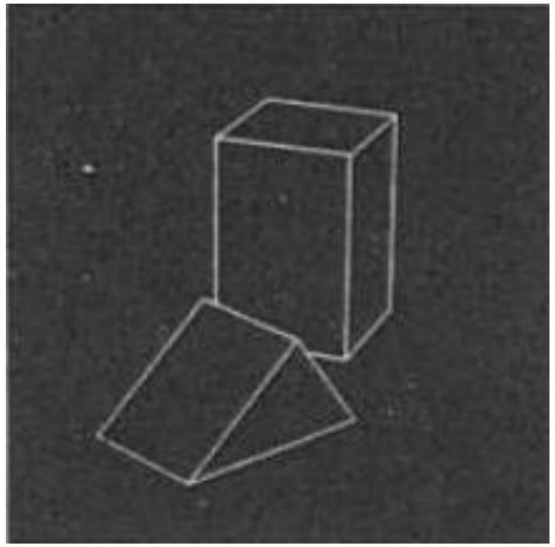
Larry Roberts
"Father of Computer Vision"



Input image



2x2 gradient operator

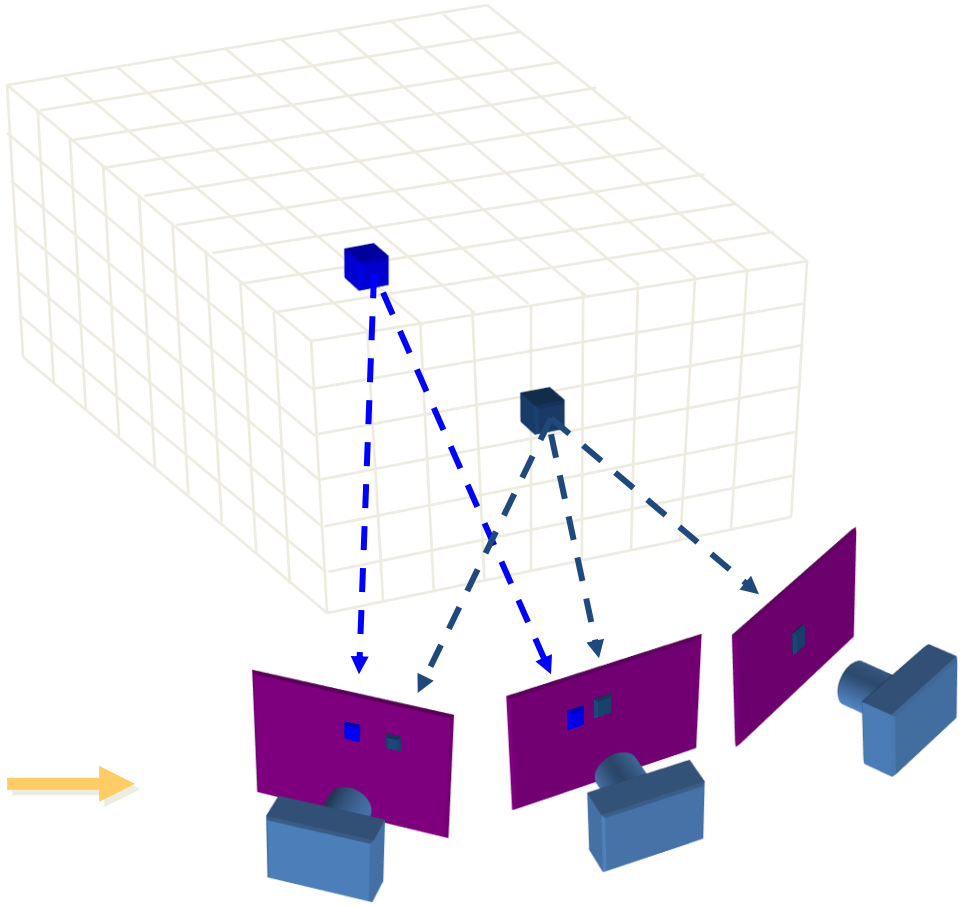


computed 3D model
rendered from new viewpoint

- History goes way back to the **first** Computer Vision paper!
  Roberts: Machine Perception of Three-Dimensional Solids, MIT, 1963

# Volumetric Stereo

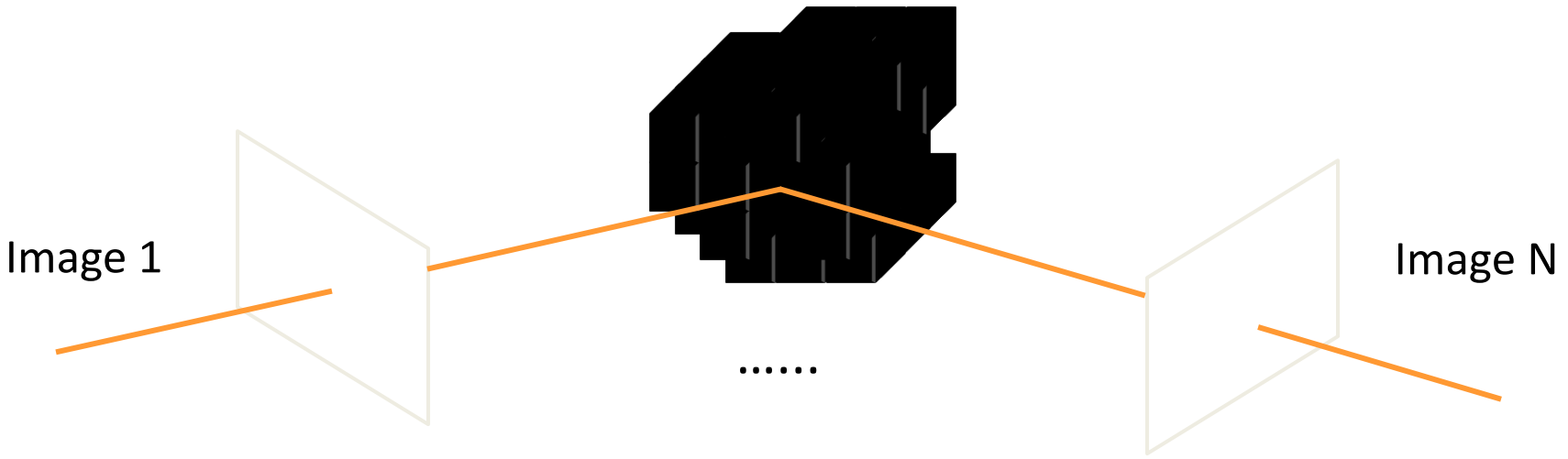
Discretized
Scene Volume



Input Images
(Calibrated)

Goal: Assign RGB values to voxels in V
*photo-consistent* with images

# Space Carving



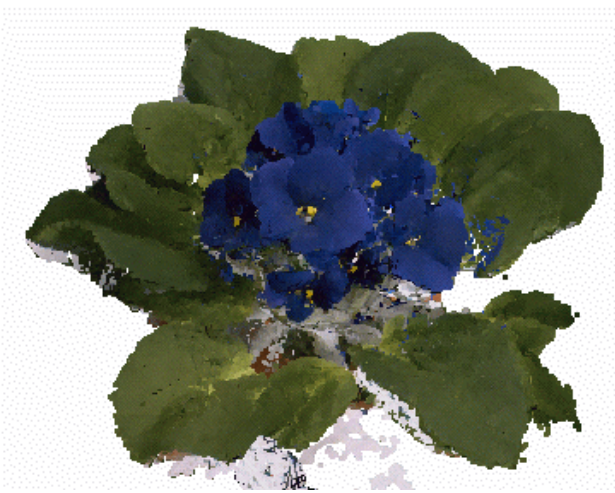

- Space Carving Algorithm
  - Initialize to a volume V containing the true scene
  - Choose a voxel on the outside of the volume
  - Project to visible input images
  - Carve if not photo-consistent
  - Repeat until convergence

K. N. Kutulakos and S. M. Seitz, A Theory of Shape by Space Carving, *ICCV* 1999

# Space Carving Result

Input Image (1 of 45)

Reconstruction



Reconstruction

Reconstruction

Source: S. Seitz

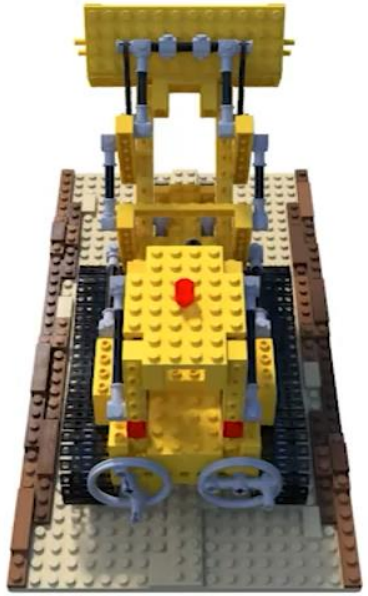
# Space Carving Result

Input Image
(1 of 100)

Reconstruction

Source: S. Seitz

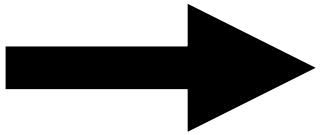
# Neural Radiance Fields



Original NeRF paper: 6400+ citations in 4 years

# Rendering in Computer Graphics

3D
Scene
Representation



Rendering

Adapted from material from Pratul Srinivasan

# Rendering in Computer Graphics

Inverse Rendering

3D
Scene
Representation

Rendering

Adapted from material from Pratul Srinivasan

# Neural Radiance Fields (NeRF) as An Approach to Inverse Rendering

Inverse Rendering

Neural Radiance Field

Rendering

Adapted from material from Pratul Srinivasan

# Deep Learning for 3D Reconstruction

- Previously: we reconstruct geometry by running stereo or multi-view stereo on a set of images
  - "Classical" approach

- How can we leverage powerful tools of deep learning?
  - Deep neural networks
  - GPU-accelerated stochastic gradient descent

# NeRF and Related Methods – Key Ideas

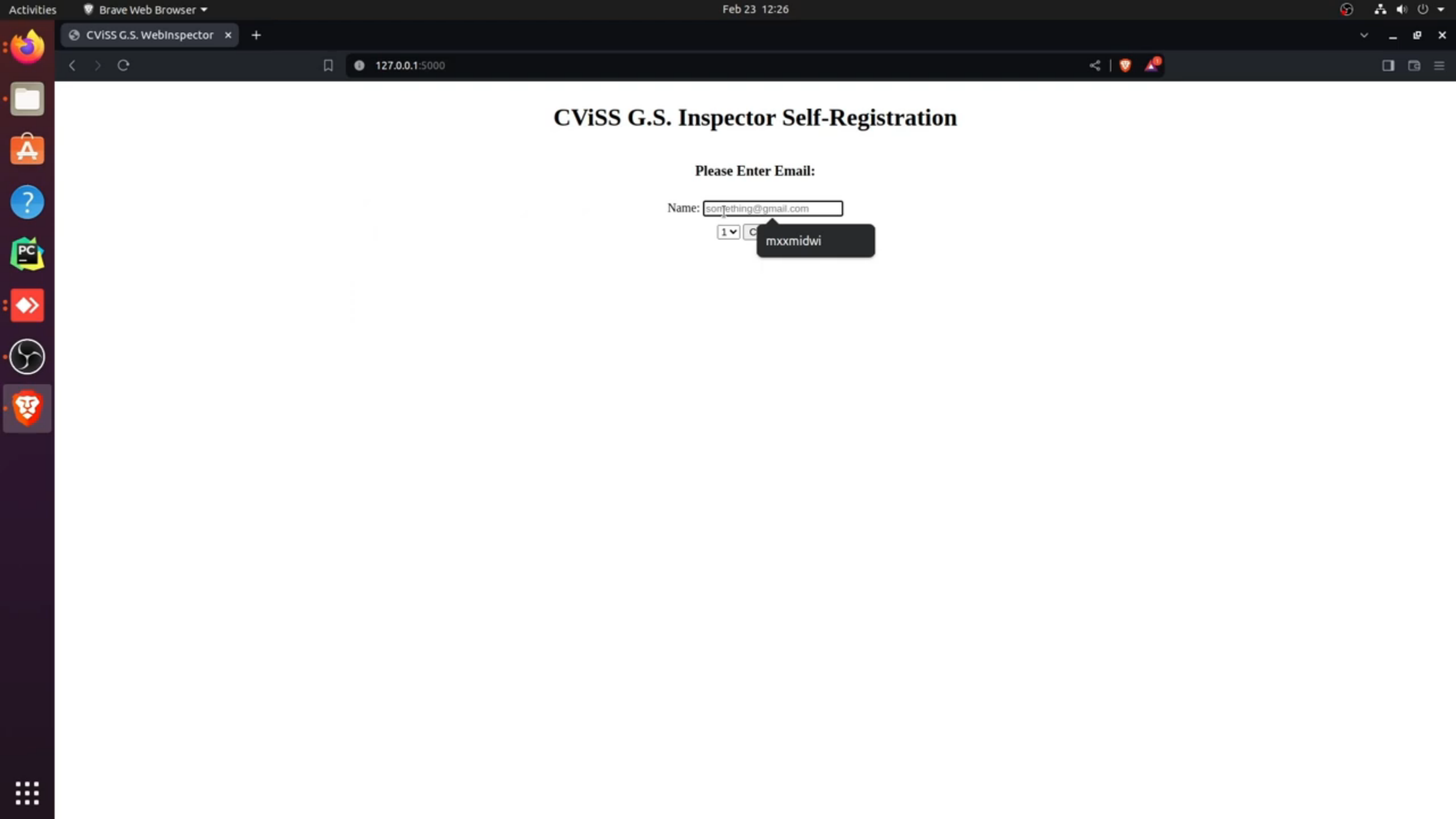
- We need to create a loss function and a scene representation that we can optimize using gradient descent to reconstruct the scene
- **Differentiable rendering**

# Handling Appearance Changes

Nerf-W [Martin-Brualla et al. CVPR 2021]

# Real-time Rendering


Video from PlenOctrees [Yu et al. CVPR 2021]


Activities
Brave Web Browser
Feb 23 12:26
CViSS G.S. WebInspector
127.0.0.1:5000
CViSS G.S. Inspector Self-Registration
Please Enter Email:
Name:
something@gmail.com
1
C
mxxmidwi

# City-Scale NeRFs

BlockNeRF [Tancik et al. CVPR 2022]

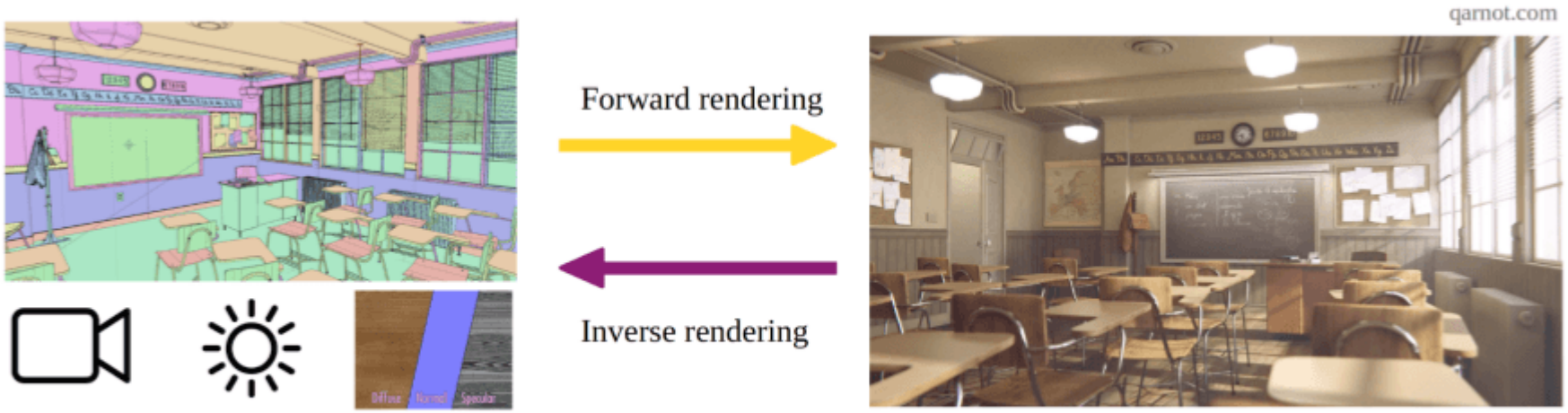
# Querying with Language

[Kerr* and Kim* et al. LERF, ICCV 2023]

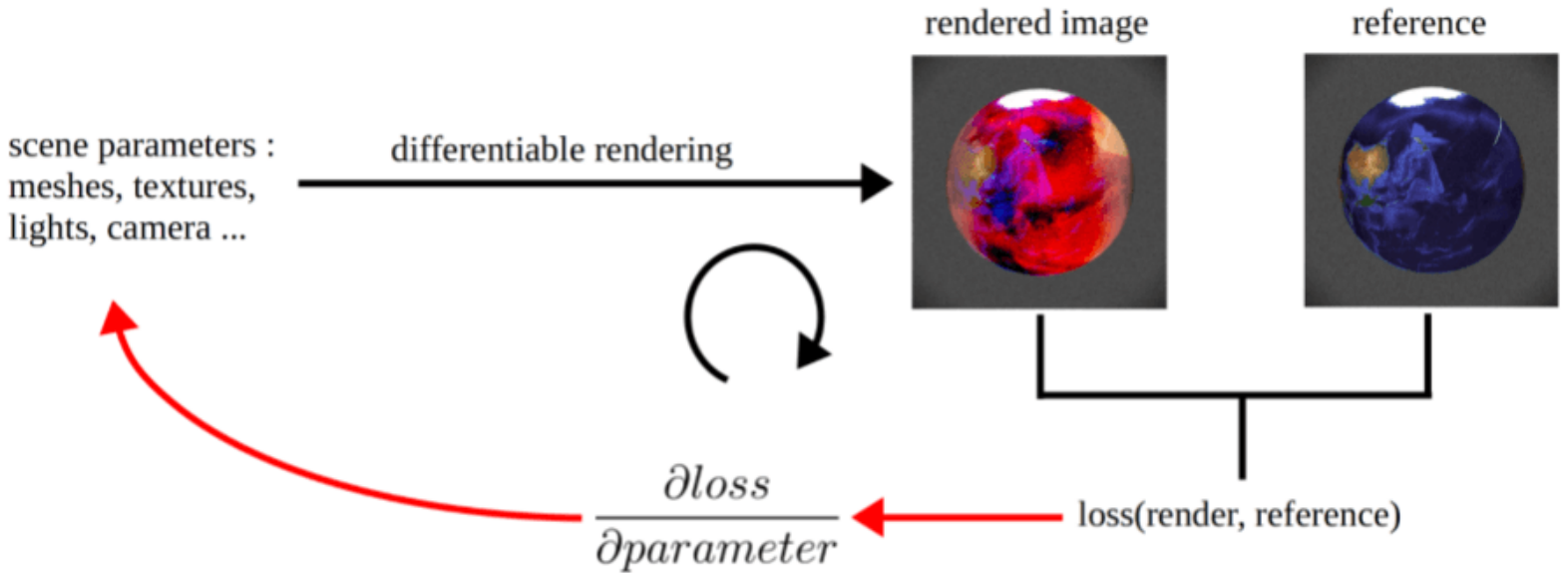
# Differentiable Rendering

What is **Differentiable Rendering**?

3D rendering can be defined as a function that takes a 3D scene as an input and outputs a 2D image. The goal of differentiable rendering is to provide a differentiable rendering function, that is to say to compute the derivatives of that function with regard to different scene parameters.



# Differentiable Rendering (Continuous)




Once a renderer is differentiable, it can be integrated in optimization or neural network pipelines. These pipelines can then be used to solve inverse graphics problems such as 3D reconstruction from 2D images or light transport optimization tasks.

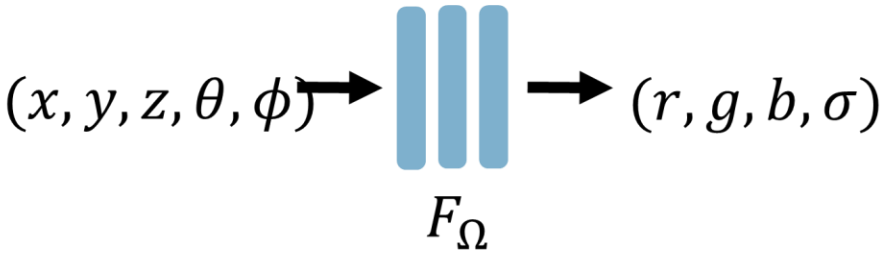
Many forward rendering algorithms (as opposed to inverse rendering algorithms) have not been designed with differentiation in mind, phenomena such as occlusion introduce many discontinuities and in rasterization algorithms almost every step is non-differentiable. While derivatives might be easy to obtain with regard to parameters such as color or glossiness, differentiation with regard to geometric parameters such as vertex positions or object orientation often requires changing the way the image is computed. Designing powerful and efficient differentiable rendering methods is an active area of computer graphics research.

# Problem Statement of NeRF

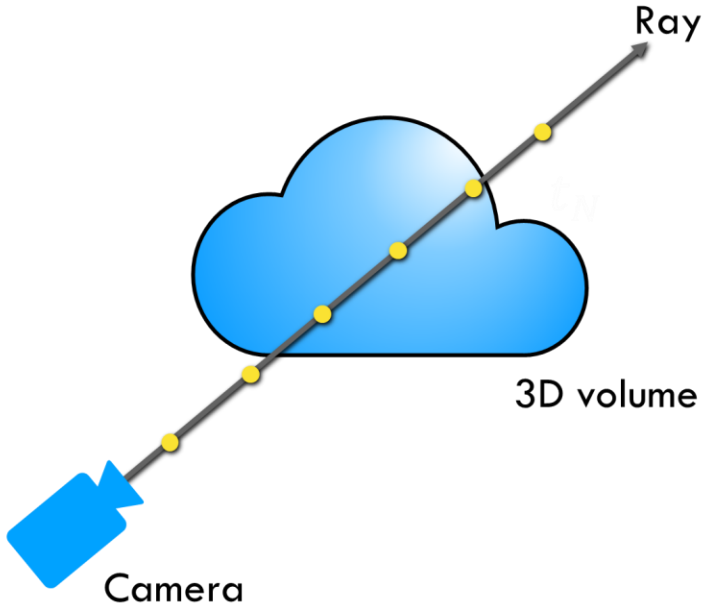
Input:
A set of calibrated Images

Output:
A 3D scene representation that renders novel views

# Three Key Components

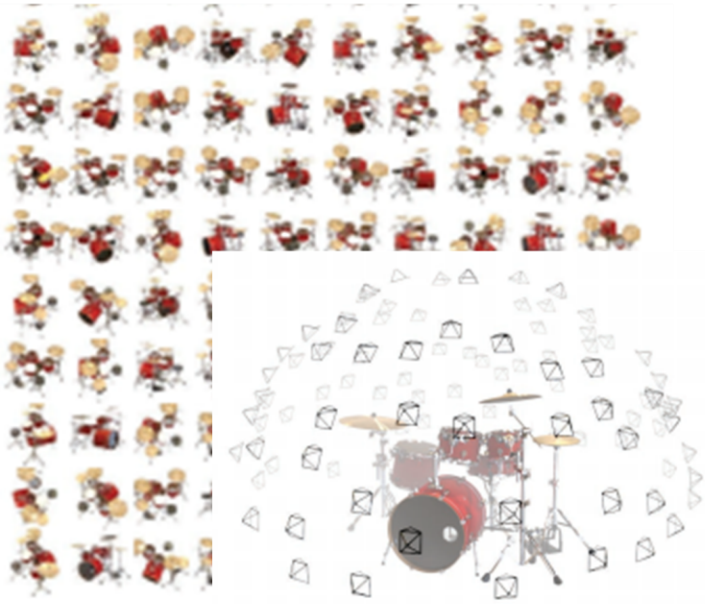



Neural Volumetric 3D Scene Representation




Differentiable Volumetric Rendering Function

Objective: Synthesize all training views



Optimization via Analysis-by-Synthesis

# What is the Problem that is being Solved?

# Plenoptic Function

Figure by Leonard McMillan

Q: What is the set of all things that we can ever see?

A: The Plenoptic Function (Adelson & Bergen ‘91)

Slide credit:
Alyosha Efros

# Grayscale Snapshot

$$P(\theta,\phi)$$

- is intensity of light
- Seen from a single position (viewpoint)
- At a single time
- Averaged over the wavelengths of the visible spectrum

# Color Snapshot

$$P(\theta,\phi,\lambda)$$

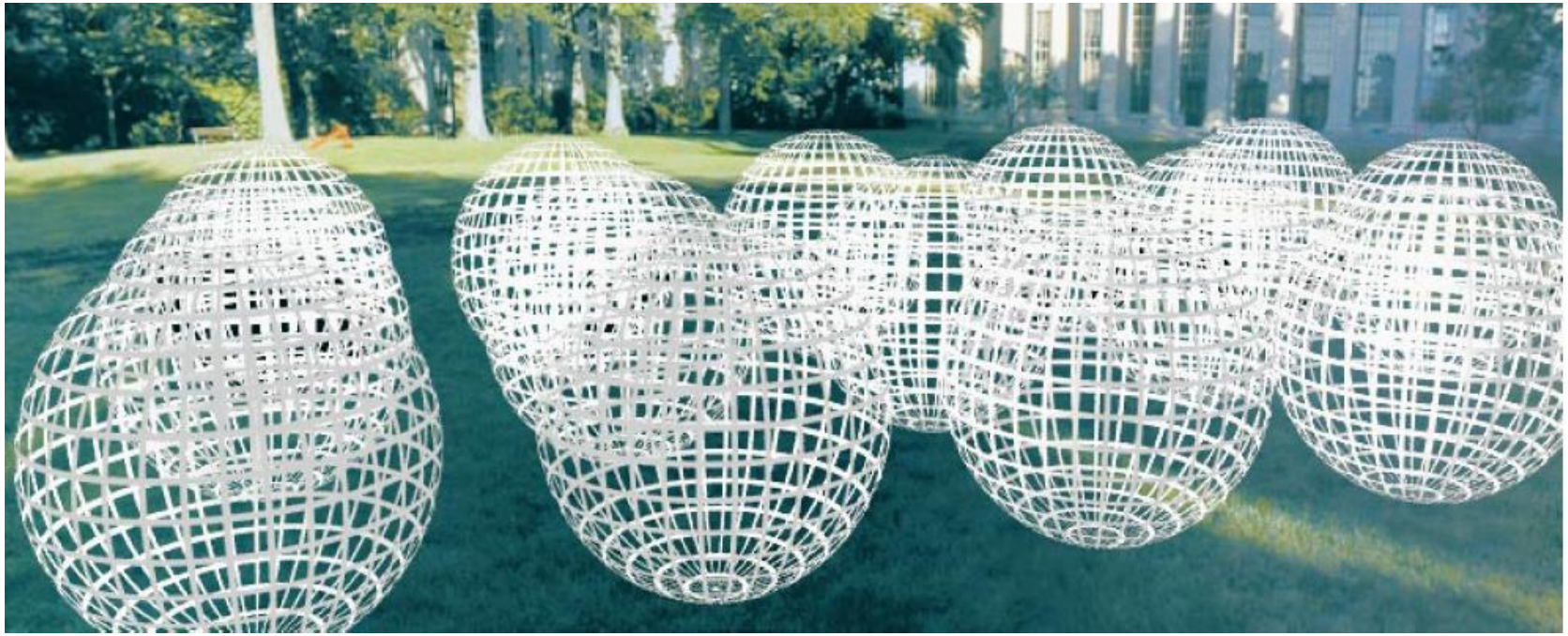
- is intensity of light
- Seen from a single position (viewpoint)
- At a single time
- As a function of wavelength

# A Movie

$$P(\theta,\phi,\lambda,t)$$

- is intensity of light
- Seen from a single position (viewpoint)
- Over time
- As a function of wavelength

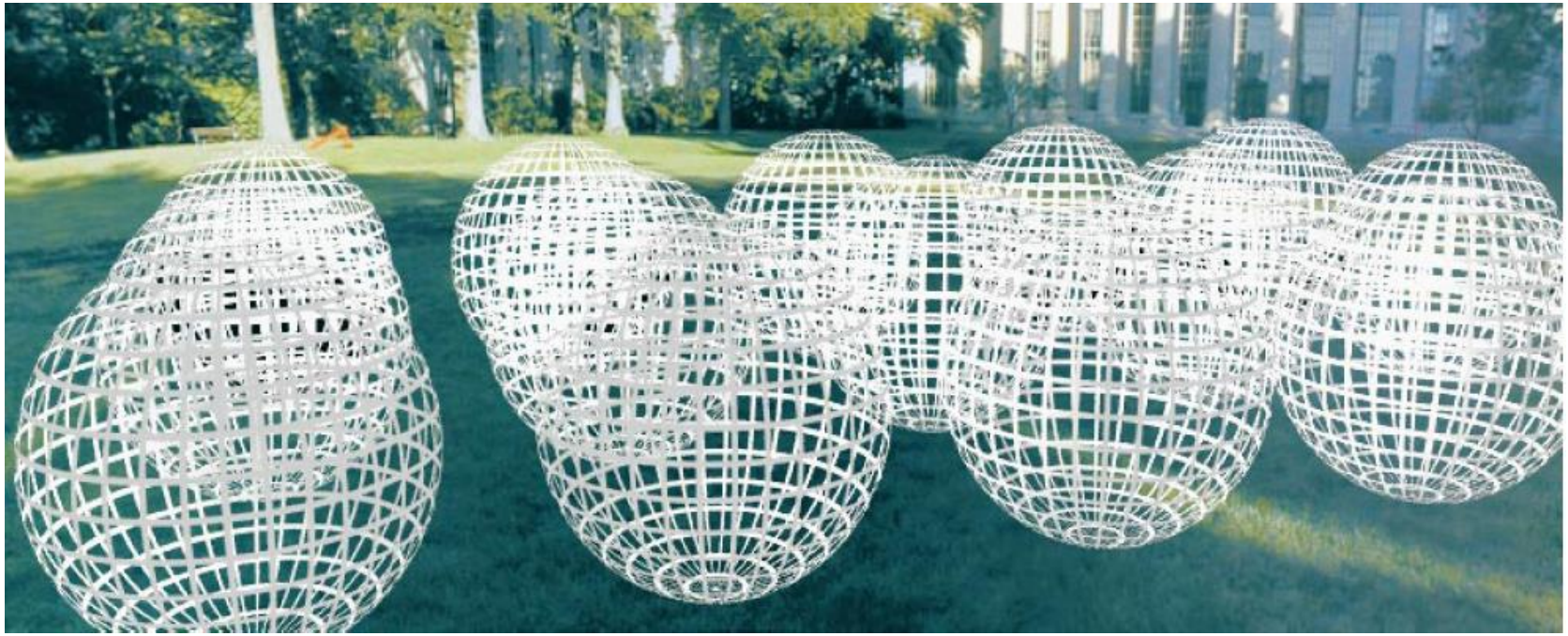
# The Plenoptic Function



$$P(\theta,\phi,\lambda,t,V_X,V_Y,V_Z)$$

- is intensity of light
- Seen from ANY position and direction
- Over time
- As a function of wavelength

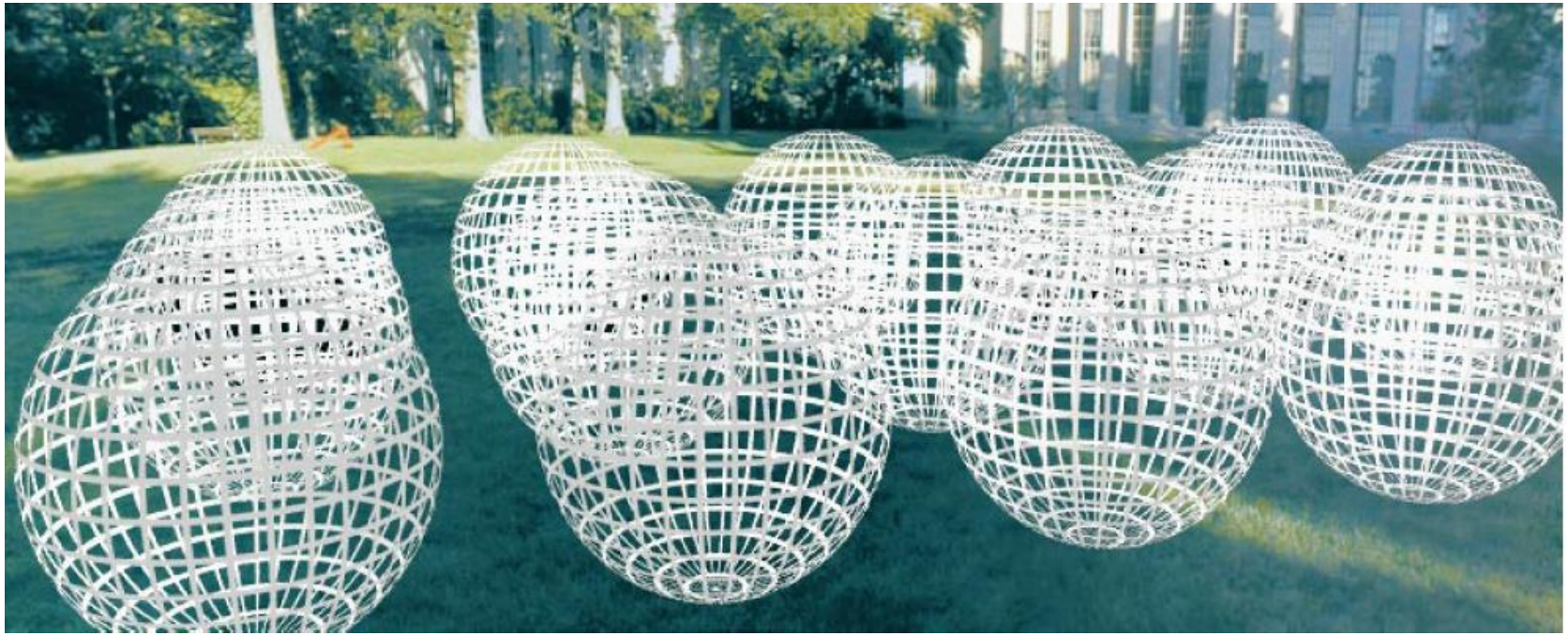
# A Holographic Movie



$$P(\theta,\phi,\lambda,t,V_X,V_Y,V_Z)$$

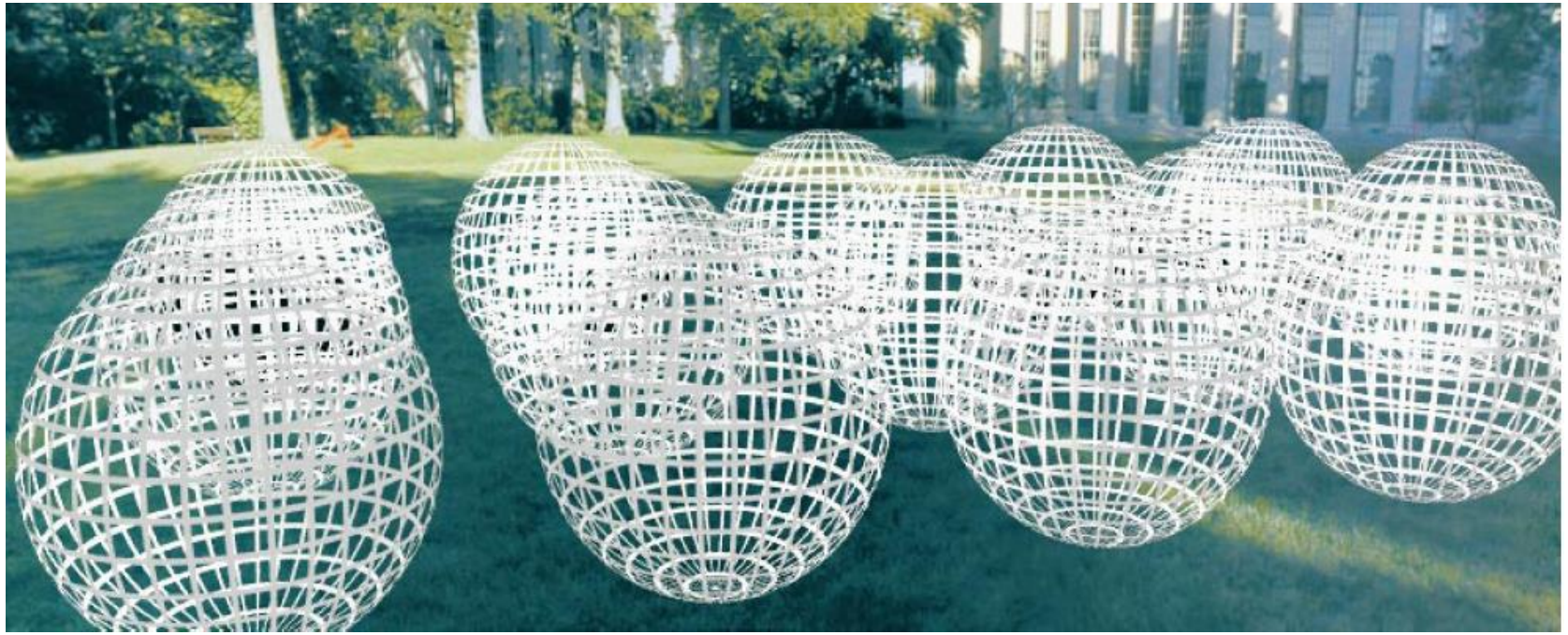
- 7D function, that can reconstruct every position & direction, at every moment, at every wavelength = it recreates the entirety of our visual reality!

# Goal: Plenoptic Function from a Set of Images



- Objective: Recreate the visual reality
- All about recovering photorealistic pixels, not about recording 3D points or surfaces
  —Image Based Rendering *aka Novel View Synthesis*
- It is a conceptual device (Adelson & Bergen do not discuss how to solve this)

# Plenoptic Function



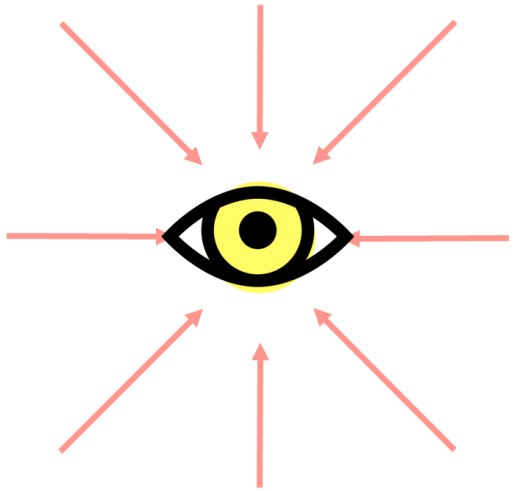
7D function:
2 – direction
1 – wavelength
1 – time
3 – location

$$P(\theta,\phi,\lambda,t,V_X,V_Y,V_Z) \longrightarrow P(\theta,\phi,V_X,V_Y,V_Z)$$

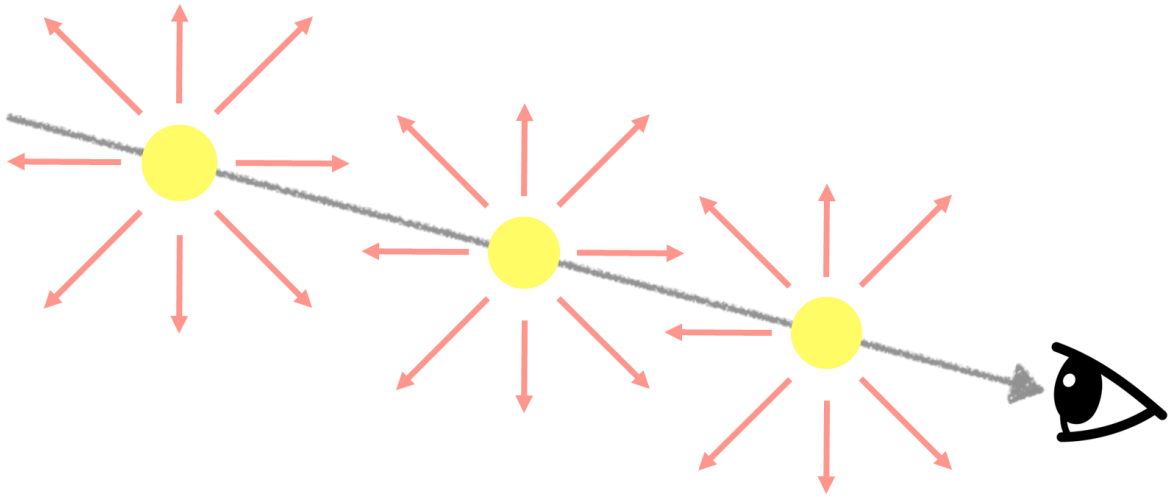
Let's simplify:

1. Remove the time
2. Remove the wavelength & let the function output RGB colors

# A Subtle difference



**Plenoptic Function**



**NeRF**

So NeRF requires the integration along the viewing ray to compute the Plenoptic Function

Bottom line: it models the full (5D) plenoptic function!

# Summary of the NeRF Training Process

## Step 1: Marching the Camera Rays Through the Scene

https://dtransposed.github.io/blog/2022/08/06/NeRF/

**Input:** A set of camera poses $\{x_c, y_c, z_c, \gamma_c, \theta_c\}_n$
**Output:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$

## Step 2: Collecting Query Points

**Input:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$
**Output:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$

## Step 3: Projecting Query Points to High-Dimensional Space (Positional Encoding)

**Input:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$
**Output:** A set of query points embedded into (d)-dimensional space $\{x_1, x_2, ..., x_d\}_{n \times m \times H \times W}$

## Step 4: Neural Network Inference and Volume Rendering

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$
**Output:** Volume density for every query point $\{\sigma\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$ and $\{v_d\}_{n \times m \times H \times W}$
**Output:** RGB colour for every query point $\{\mathrm{RGB}\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) + their volume profile + RGB value $\{x_1, x_2, \ldots, x_d, \mathrm{RGB}, \sigma\}_{n \times m \times H \times W}$
**Output:** A set of rendered images (one per pose) $\{H, W\}_n$

**Input:** A set of rendered images (one per pose) $\{H, W\}_n$ and a set of ground truth images (one per pose) $\{H, W\}_n^{\mathrm{gt}}$
**Output:** L2 loss between the inputs, a single scalar $\{\ell\}_n$

# Summary of the NeRF Training Process

## Step 1: Marching the Camera Rays Through the Scene

**Input:** A set of camera poses $\{x_c, y_c, z_c, \gamma_c, \theta_c\}_n$
**Output:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$

## Step 2: Collecting Query Points

**Input:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$
**Output:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$

## Step 3: Projecting Query Points to High-Dimensional Space (Positional Encoding)

**Input:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$
**Output:** A set of query points embedded into (d)-dimensional space $\{x_1, x_2, ..., x_d\}_{n \times m \times H \times W}$

## Step 4: Neural Network Inference and Volume Rendering

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$
**Output:** Volume density for every query point $\{\sigma\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$ and $\{v_d\}_{n \times m \times H \times W}$
**Output:** RGB colour for every query point $\{\mathrm{RGB}\}_{n \times m \times H \times W}$

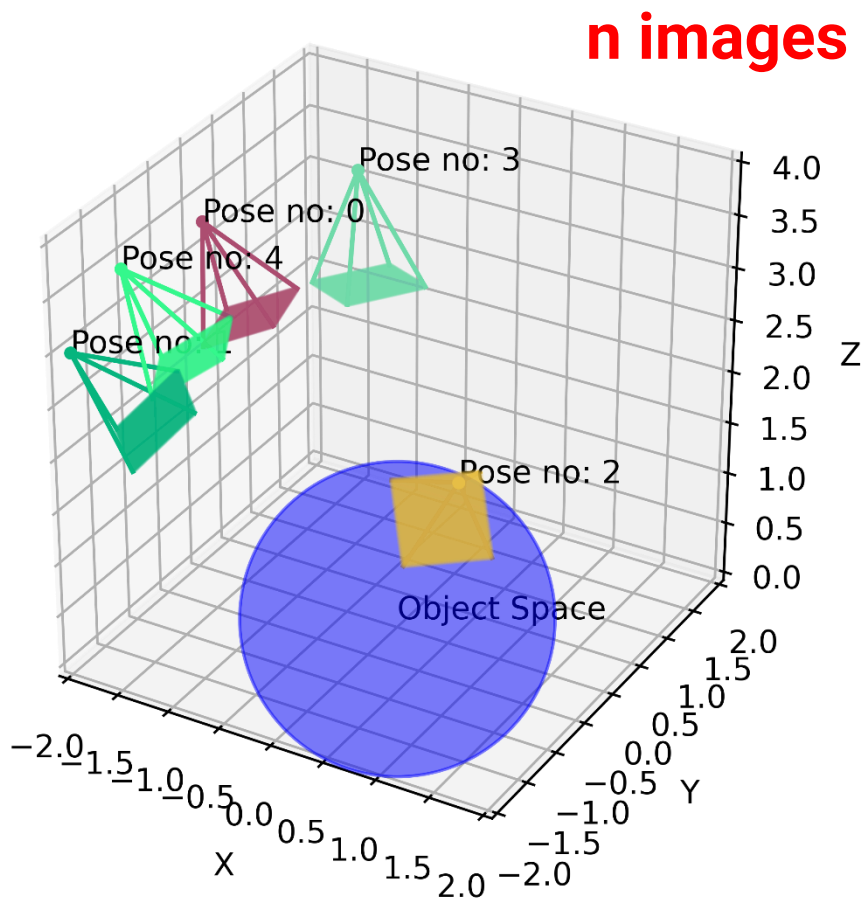
**Input:** A set of 3D query points (after positional encoding) + their volume profile + RGB value $\{x_1, x_2, \ldots, x_d, \mathrm{RGB}, \sigma\}_{n \times m \times H \times W}$
**Output:** A set of rendered images (one per pose) $\{H, W\}_n$

**Input:** A set of rendered images (one per pose) $\{H, W\}_n$ and a set of ground truth images (one per pose) $\{H, W\}_n^{\mathrm{gt}}$
**Output:** L2 loss between the inputs, a single scalar $\{\ell\}_n$

## Step 1: Marching the Camera Rays Through the Scene

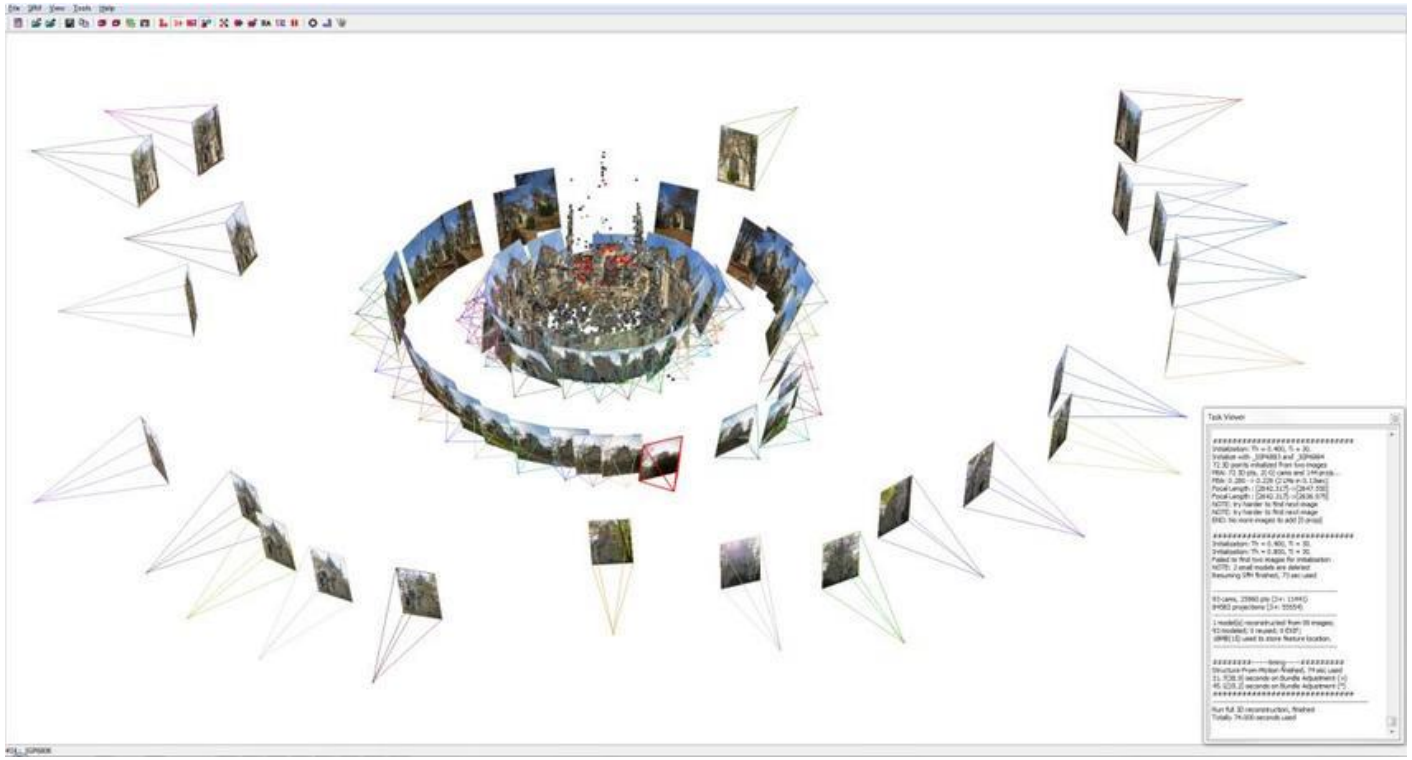
**Input:** A set of camera poses $\{x_c, y_c, z_c, \gamma_c, \theta_c\}_n$

**Output:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$



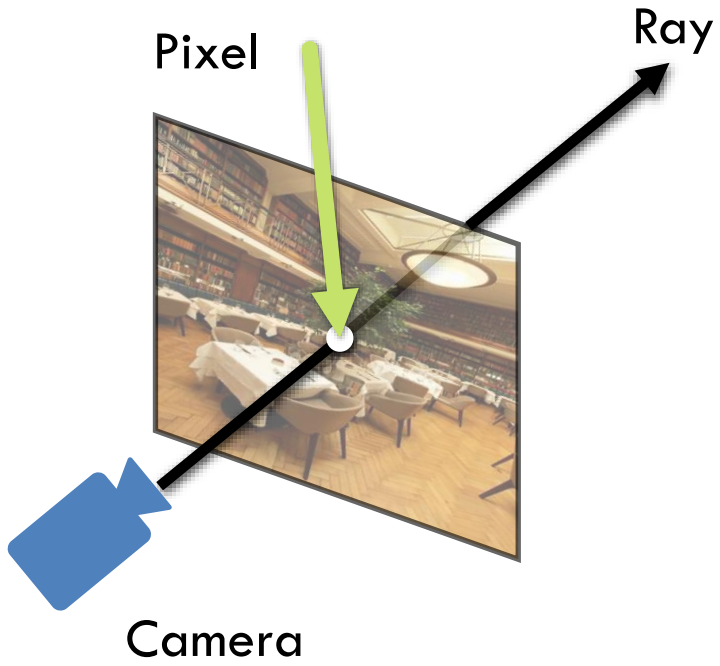

Every ray is described by two vectors:

- $\mathbf{v_o}$, a vector that specifies the origin of the ray. Note that $\mathbf{v_o} = (x_o, y_o, z_o) = (x_c, y_c, z_c)$
- $\mathbf{v_d}$, a normalized vector that specifies the direction of the ray.

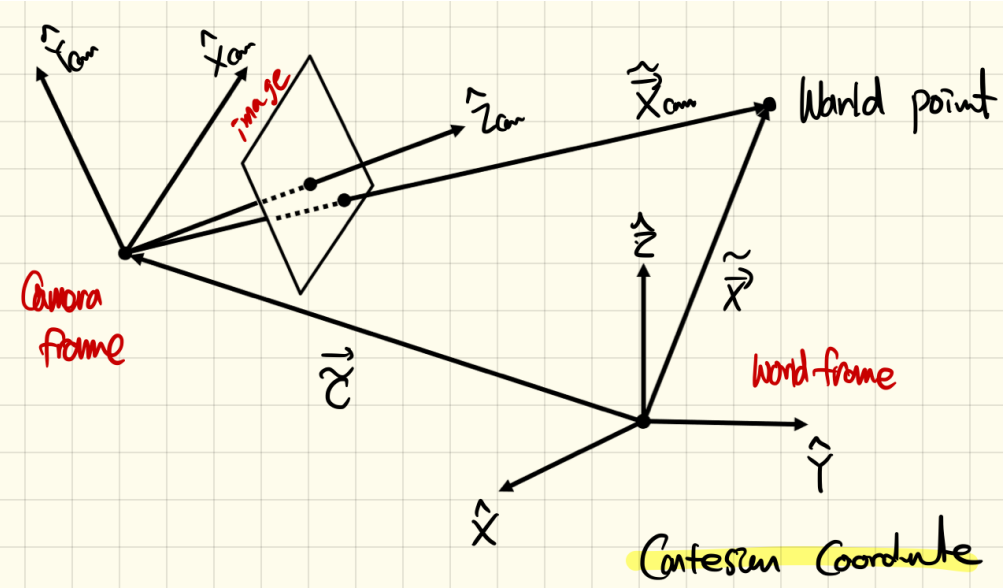


# Cameras and Rays

- We need the mathematical mapping from (*camera*, *pixel*) → *ray*
- Then can abstract underlying problem as learning the function *ray* → *color* (the "plenoptic function")



# Camera Model : Full Configuration (but, in the Cylindrical Coordinate)



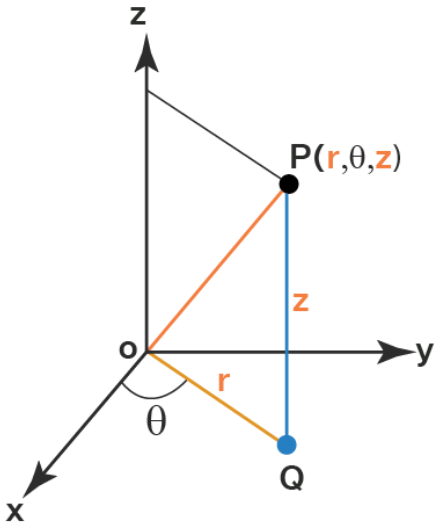

Every ray is described by two vectors:

- $\mathbf{v_o}$, a vector that specifies the origin of the ray. Note that $\mathbf{v_o} = (x_o, y_o, z_o) = (x_c, y_c, z_c)$
- $\mathbf{v_d}$, a normalized vector that specifies the direction of the ray.

$\vec{X}_{cam} = R\vec{X}$ World frame → Camera frame. Rotation

$\vec{X}_{cam} = \vec{X} - \vec{C}$ World frame → Camera frame Translation.

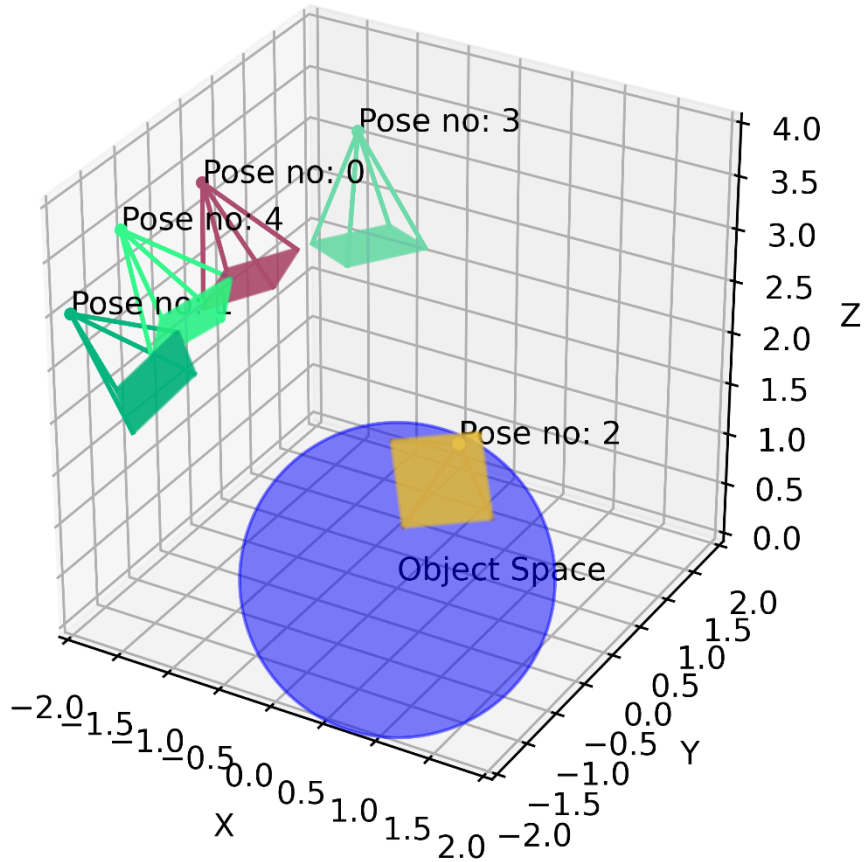
We move the origin first and rotate the axes.

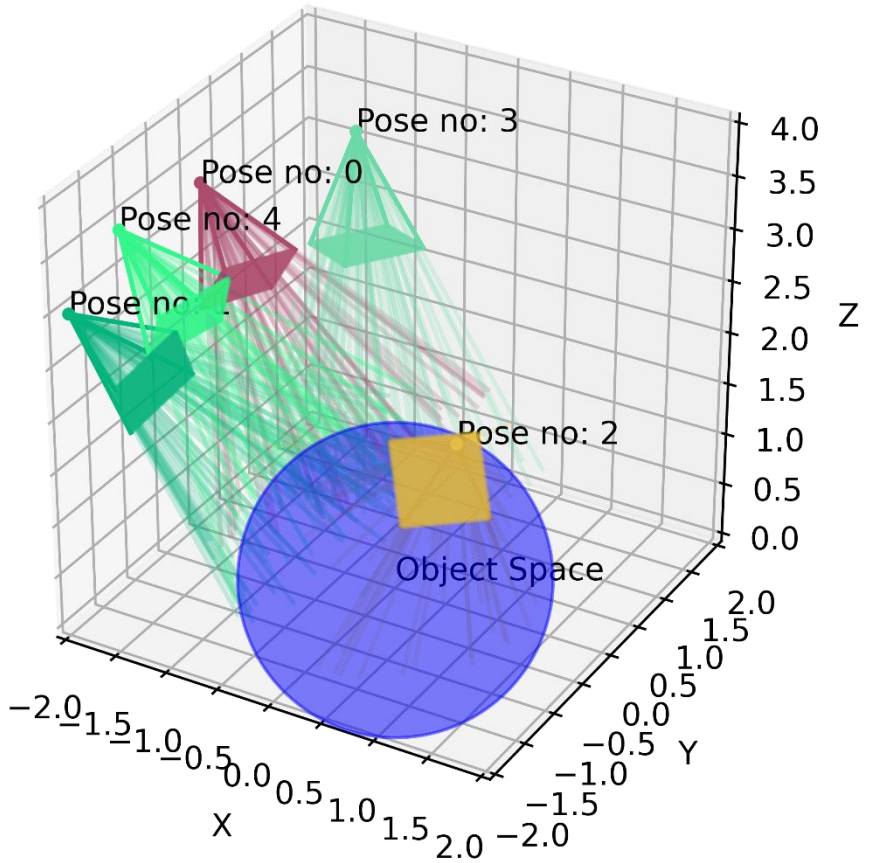


## Step 1: Marching the Camera Rays Through the Scene

**Input:** A set of camera poses $\{x_c, y_c, z_c, \gamma_c, \theta_c\}_n$

**Output:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$



# Summary of the NeRF Training Process

## Step 1: Marching the Camera Rays Through the Scene

**Input:** A set of camera poses $\{x_c, y_c, z_c, \gamma_c, \theta_c\}_n$
**Output:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$

## Step 2: Collecting Query Points

**Input:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$
**Output:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$

## Step 3: Projecting Query Points to High-Dimensional Space (Positional Encoding)

**Input:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$
**Output:** A set of query points embedded into (d)-dimensional space $\{x_1, x_2, ..., x_d\}_{n \times m \times H \times W}$

## Step 4: Neural Network Inference and Volume Rendering

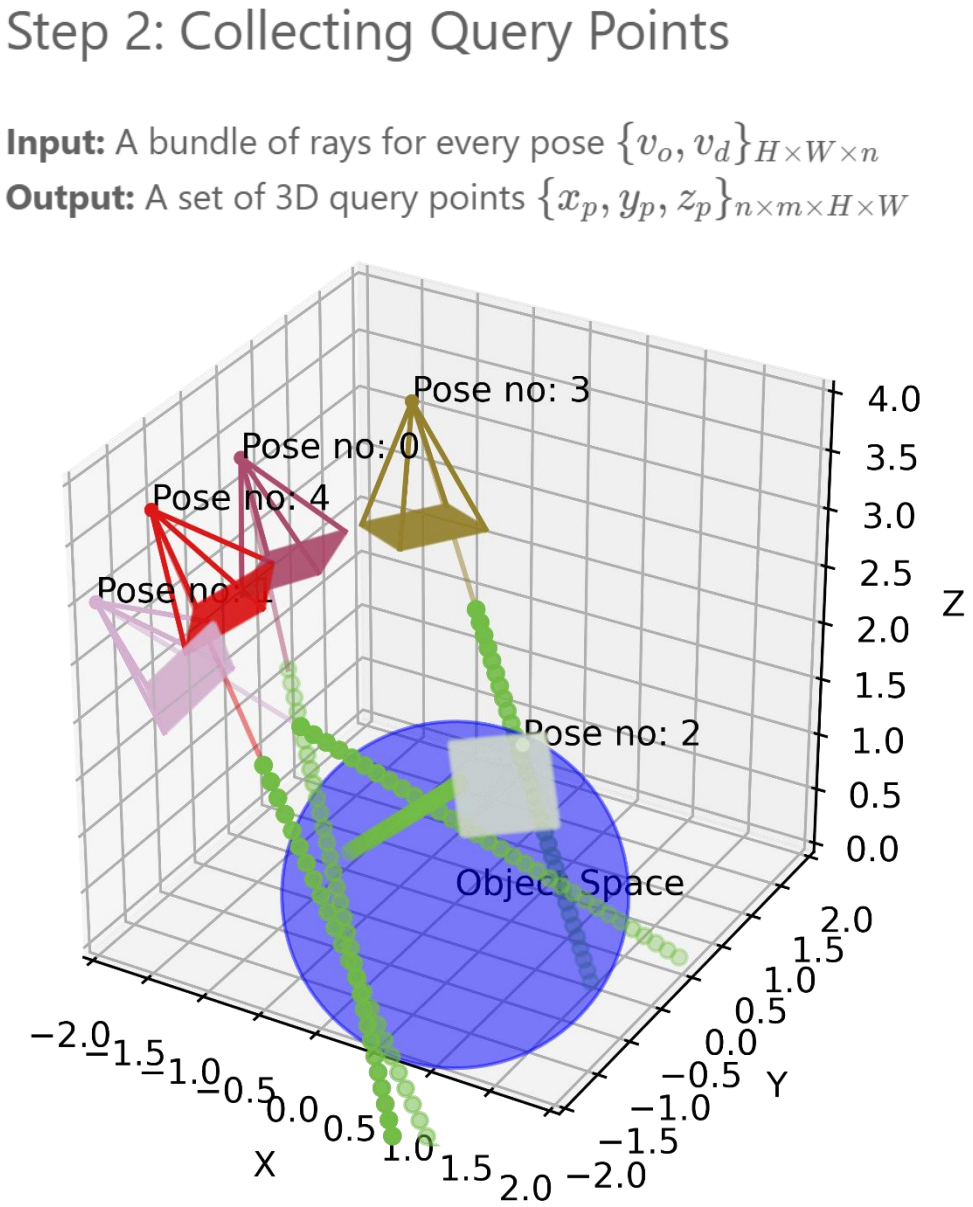
**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$
**Output:** Volume density for every query point $\{\sigma\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$ and $\{v_d\}_{n \times m \times H \times W}$
**Output:** RGB colour for every query point $\{\mathrm{RGB}\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) + their volume profile + RGB value $\{x_1, x_2, \ldots, x_d, \mathrm{RGB}, \sigma\}_{n \times m \times H \times W}$
**Output:** A set of rendered images (one per pose) $\{H, W\}_n$

**Input:** A set of rendered images (one per pose) $\{H, W\}_n$ and a set of ground truth images (one per pose) $\{H, W\}_n^{\mathrm{gt}}$
**Output:** L2 loss between the inputs, a single scalar $\{\ell\}_n$

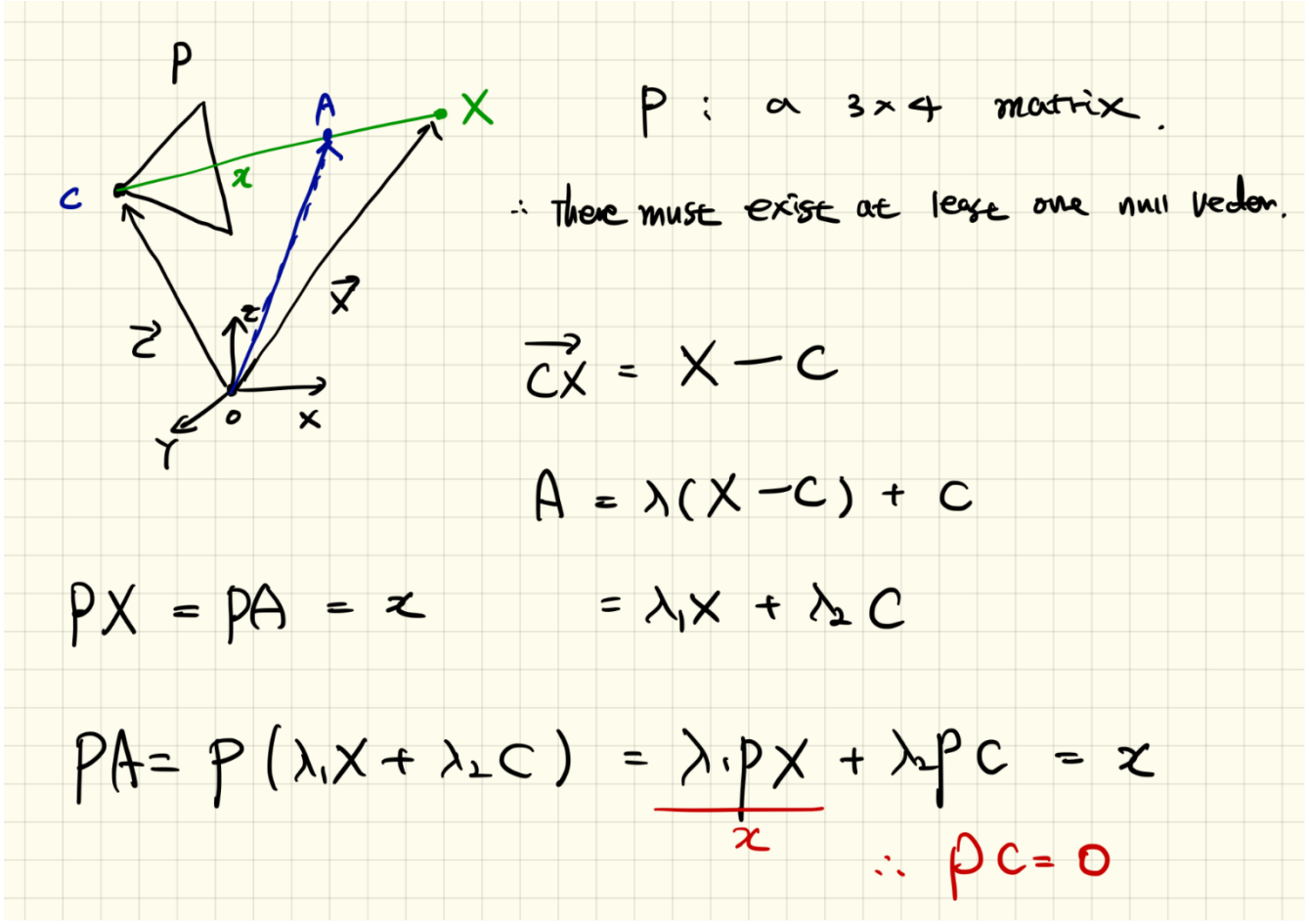
# Step 2: Collecting Query Points

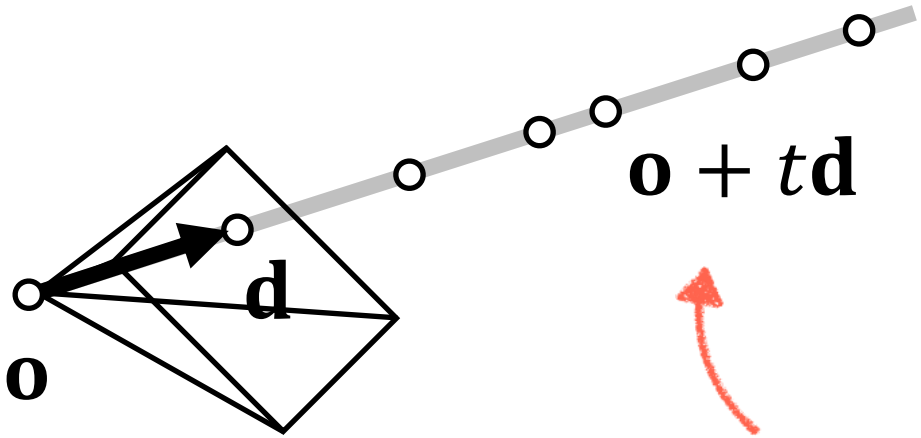



**m points in each ray**

In computer graphics, the 3D scene is very often modeled as a set of tiny, discrete region “cubes” called voxels. When a ray “flies” through the scene, it will pass through a large number of points in space. Most of those points represent “emptiness”, however, some may land on the object volume itself. The latter points are very valuable to us - they give us some knowledge about the scene. To render an image we will query the trained neural network, whether a point, some tiny piece of scene volume, belongs to the object or not, and more importantly, which visual properties it has.

# Calculating Points Along a Ray



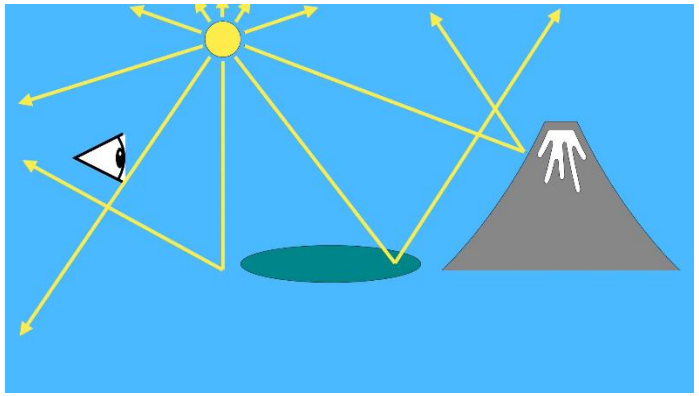




Scalar $t$ controls distance along the ray

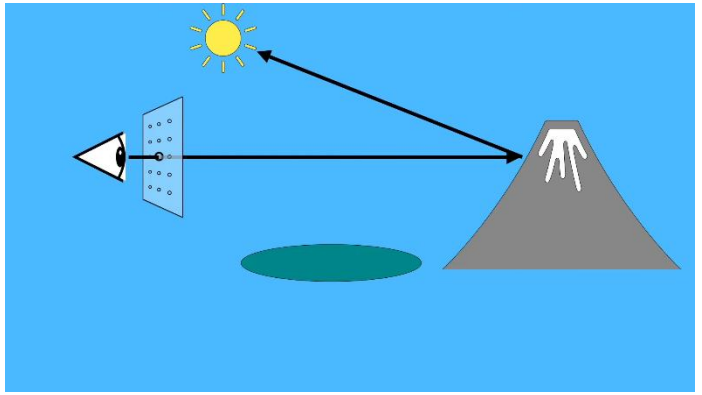
Parametric equation $P = \mathbf{v_o} + t * \mathbf{v_d}$ defines any point on the ray. So to do the "ray marching", we make the $t$ parameter larger (thus extending our rays) until a ray reaches some interesting location in the object space.

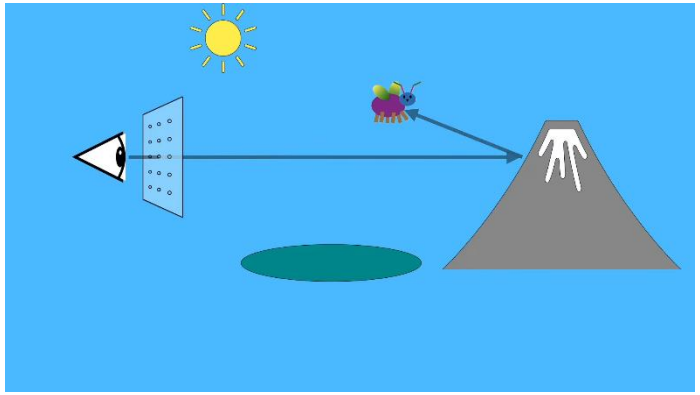
# Ray Tracing

Ray tracing is a rendering technique that can realistically simulate the lighting of a scene and its objects by rendering physically accurate reflections, refractions, shadows, and indirect lighting. Ray tracing generates computer graphics images by tracing the path of light from the view camera (which determines your view into the scene), through the 2D viewing plane (pixel plane), out into the 3D scene, and back to the light sources.



Real Rays



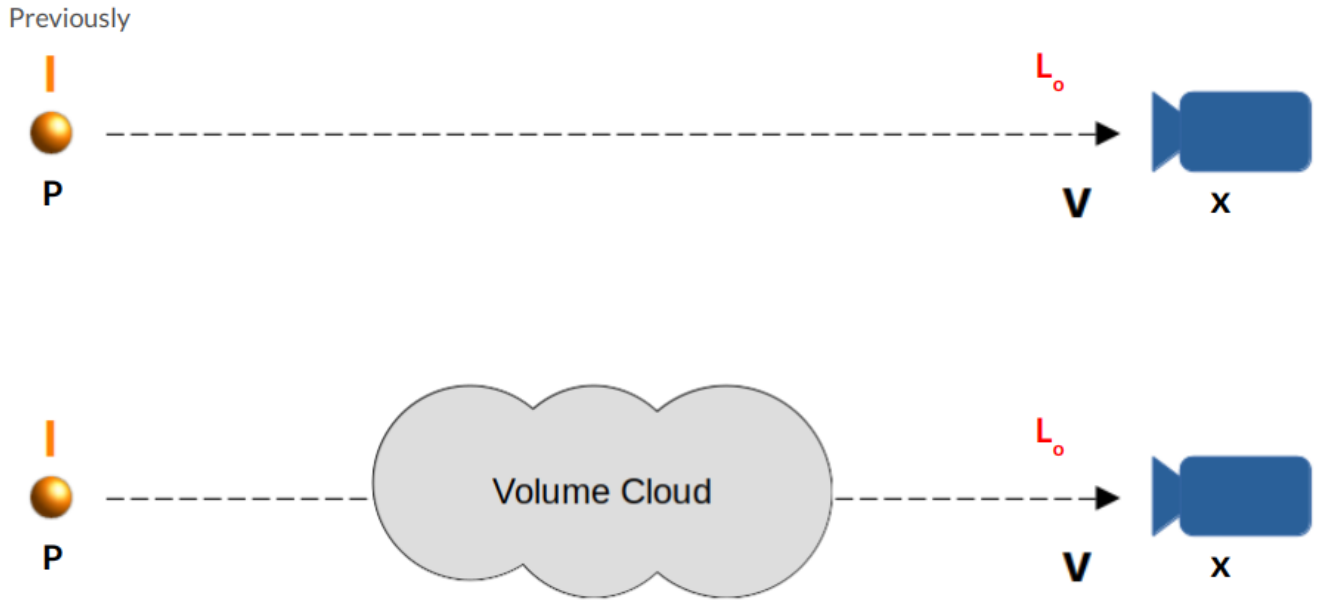


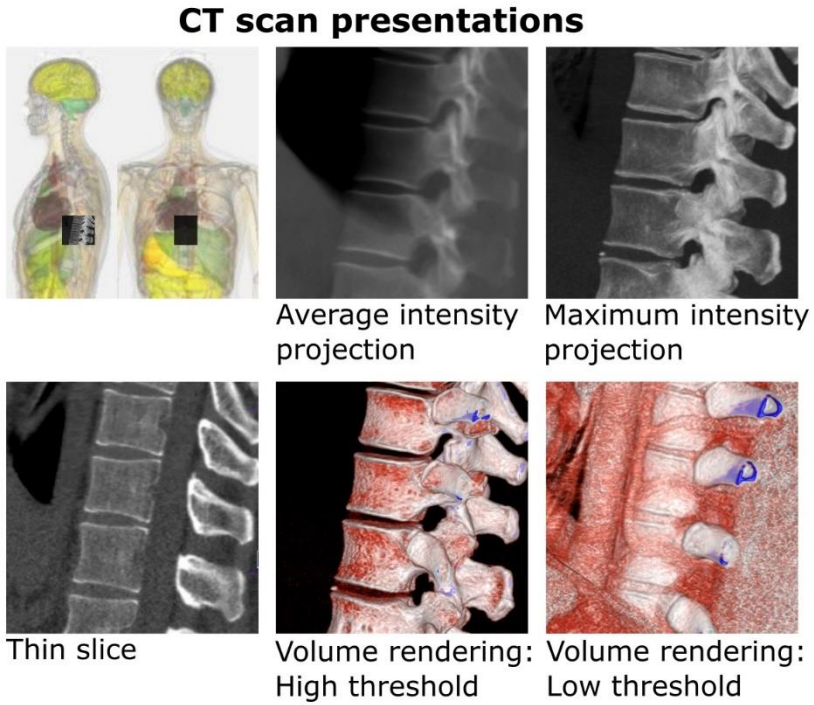
Ray Tracings

https://www.youtube.com/watch?v=H5TB2l7zq6s

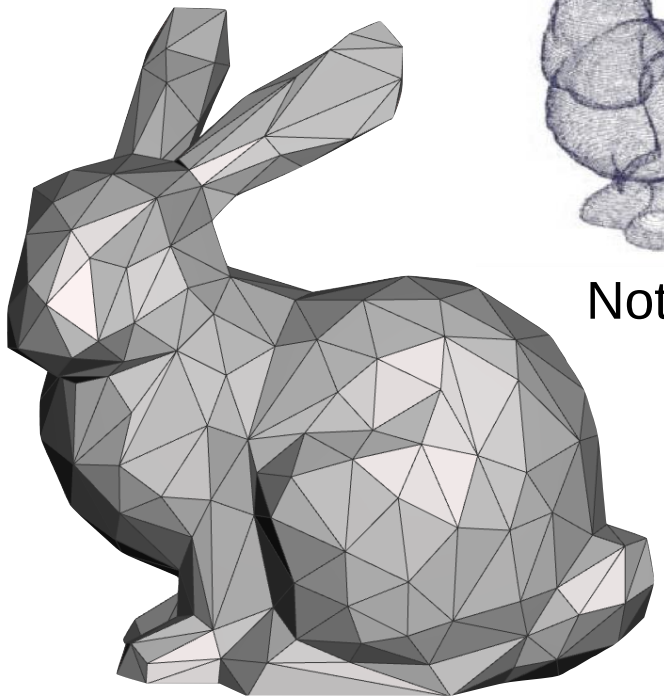
# Volume Rendering

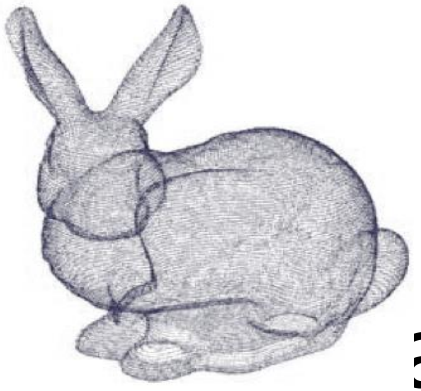
Volume rendering is a set of techniques used to display a 2D projection of a 3D discretely sampled data set. To render a 2D projection of the 3D data set, one first needs to define a camera in space relative to the volume. Also, one needs to define the opacity and color of every voxel. This is usually defined using an **RGBA** (for red, green, blue, alpha) transfer function that defines the **RGBA** value for every possible voxel value.



# Volume Rendering





a Mesh

Not a point cloud either

It is volumetric

It’s *continuous* voxels

made of shiny transparent cubes

# Volume Rendering Aspect of NeRF

## NeRF: A Volume Rendering Perspective

YU Yue (Original) September 12, 2022 About 32 min

### Overview

NeRF **implicitly** represents a 3D scene with a multi-layer perceptron (MLP) $F : (\boldsymbol{x}, \boldsymbol{d}) \to (\boldsymbol{c}, \sigma)$ for some position $\boldsymbol{x} \in \mathbb{R}^3$, view direction $\boldsymbol{d} \in [0, \pi) \times [0, 2\pi)$, color $\boldsymbol{c}$, and "opacity" $\sigma$. Rendered results are spectacular.

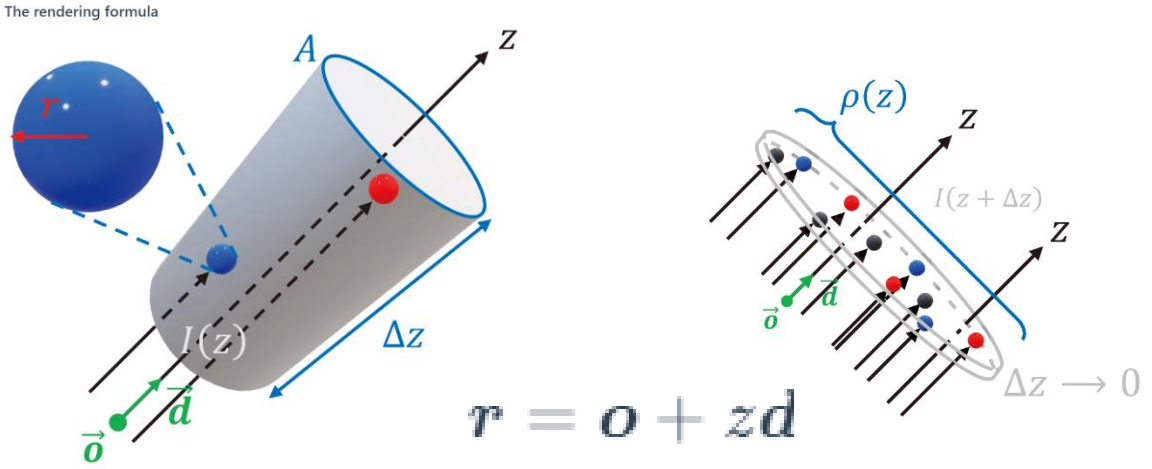

There have been a number of articles introducing NeRF since its publication in 2020. While most posts mention general methods, few of them elaborate on *why* the volume rendering procedure

$$\mathbf{C}(\boldsymbol{r}) = \mathbf{C}(z; \boldsymbol{o}, \boldsymbol{d}) = \int_{z_n}^{z_f} T(z)\sigma\left(\boldsymbol{r}(z)\right)\boldsymbol{c}\left(\boldsymbol{r}(z), \boldsymbol{d}\right)\, dz,\ T(z) = \exp\left(-\int_{z_n}^{z}\sigma\left(\boldsymbol{r}(s)\right)\, ds\right)$$

On This Page



The rendering formula




There have been a number of articles introducing NeRF since its publication in 2020. While most posts mention general methods, few of them elaborate on *why* the volume rendering procedure

$$\mathbf{C}(\boldsymbol{r}) = \mathbf{C}(z; \boldsymbol{o}, \boldsymbol{d}) = \int_{z_n}^{z_f} T(z)\sigma\left(\boldsymbol{r}(z)\right)\boldsymbol{c}\left(\boldsymbol{r}(z), \boldsymbol{d}\right)\, dz,\ T(z) = \exp\left(-\int_{z_n}^{z}\sigma\left(\boldsymbol{r}(s)\right)\, ds\right)$$

for a ray $\boldsymbol{r} = \boldsymbol{o} + z\boldsymbol{d}$ works and *how* the equation is reduced to

$$\hat{\mathbf{C}}(\boldsymbol{r}) = \hat{\mathbf{C}}(z_1, z_2, \ldots, z_N; \boldsymbol{o}, \boldsymbol{d}) = \sum_{i=1}^{N} T_i\left(1 - e^{-\sigma_i\delta_i}\right)\boldsymbol{c}_i,\ T_i = \exp\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right)$$

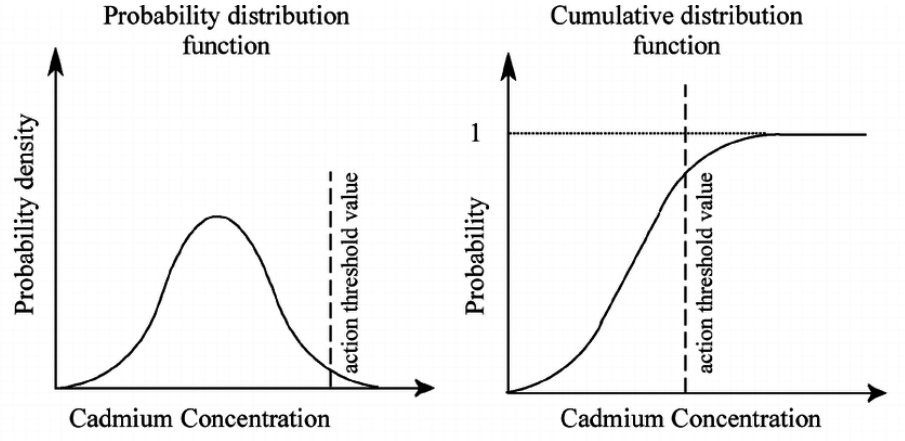
via numerical quadrature, let alone exploring its implementation via ***Monte Carlo* method**.

https://yconquesty.github.io/blog/ml/nerf/nerf_rendering.html#numerical-quadrature

# Volume Rendering Aspect of NeRF (Continue)

$$I(z) = I(z_0)\underbrace{e^{\int_{z_0}^{z} -\sigma(s)\,ds}}_{T(z)}$$

Define accumulated *transmittance* $T(z) := e^{\int_{z_0}^{z} -\sigma(s)\,ds}$, then $I(z) = I(z_0)T(z)$ means the **remainning** intensity after the rays travels from $z_0$ to $z$. $T(z)$ can also be viewed as the *cumulative density function* (CDF) that a ray does **not** hit any particles from $z_0$ to $z$. But **no** color will be observed if a ray passes empty space; radiance is "emitted" only when there is **contact** between rays and particles. Define




empty space; radiance is "emitted" only when there is **contact** between rays and particles. Define

$$H(z) := 1 - T(z)$$

as the CDF that a ray hits particles from $z_0$ to $z$, then its *probability density function* (PDF) is

$$p_{\text{hit}}(z) = -\underbrace{e^{\int_{z_0}^{z} -\sigma(s)\,ds}}_{T(z)}\sigma(z)$$

Let a *random variable* a random variable $\mathbf{R}$ denote the emitted radiance, then

$$\begin{aligned} p_{\mathbf{R}}(\boldsymbol{r}) &= p_{\mathbf{R}}(z; \boldsymbol{o}, \boldsymbol{d}) \\ &:= P[\mathbf{R} = \boldsymbol{c}(z)] \\ &= p_{\text{hit}}(z) \end{aligned}$$

Hence, the color of a pixel is the *expectation* of emitted radiance:

$$\begin{aligned} \mathbf{C}(\boldsymbol{r}) = \mathbf{C}(z; \boldsymbol{o}, \boldsymbol{d}) &= \mathbb{E}(\mathbf{R}) \\ &= \int_0^\infty \mathbf{R}\, p_{\mathbf{R}}\, dz \\ &= \int_0^\infty \boldsymbol{c}\, p_{\text{hit}}\, dz \\ &= \int_0^\infty T(z)\sigma(z)\boldsymbol{c}(z)\, dz \end{aligned}$$

**Integration bounds**

In practice, $\boldsymbol{c}$, obtained from MLP query, is a function of both position $z$ (or coordinate $\boldsymbol{x}$) and view direction $\boldsymbol{d}$. Also different are the integration bounds. A computer does not support an infinite range; the lower and upper bounds of integration are $z_{\text{near}} = \texttt{near}$ and $z_{\text{far}} = \texttt{far}$ within the range of floating point representation:

$$\mathbf{C}(\boldsymbol{r}) = \int_{\texttt{near}}^{\texttt{far}} T(z)\sigma(z)\boldsymbol{c}(z)\, dz$$

In NeRF, `near` = `0.` and `far` = `1.` for scaled **bounded** scenes and front facing scenes after conversion to normalized device coordinates (NDC).

https://yconquesty.github.io/blog/ml/nerf/nerf_rendering.html#numerical-quadrature

# Volume Rendering Aspect of NeRF (Continue)

## Numerical quadrature

We took **a step from discrete to continuous** to derive the rendering integral. Nevertheless, integration on a continuous domain is not supported by computers. An alternative is numerical quadrature. Sample $\texttt{near} < z_1 < z_2 < \cdots < z_N < \texttt{far}$ along a ray, and define differences between adjacent samples as

$$\delta_i := z_{i+1} - z_i \ \forall i \in \{1, \ldots, N-1\}$$

then the **transmittance** is approximated by

$$\begin{aligned} T_i &:= T(z_i) \\ &\approx e^{-\sum_{j=1}^{i-1} \sigma_j \delta_j} \end{aligned}$$

$$I(z) = I(z_0)\, \underbrace{e^{\int_{z_0}^{z} -\sigma(s)\, ds}}_{T(z)}$$

where $T_1 = 1$ and $\sigma_j = \sigma(z_j; \boldsymbol{o}, \boldsymbol{d})$. Meanwhile, differentiation in $p_{\text{hit}}(z)$ is also substituted by **discrete difference**. That is,

$$\begin{aligned} p_i := p_{\text{hit}}(z_i) &= \left.\frac{dH}{dz}\right|_{z_i} \\ &\approx H(z_{i+1}) - H(z_i) \\ &= 1 - T(z_{i+1}) - (1 - T(z_i)) \\ &= T(z_i) - T(z_{i+1}) \\ &= T_i \left(1 - e^{-\sigma_i \delta_i}\right) \end{aligned}$$

$$\int_a^b f(x)dx = \lim_{n\to\infty} \sum_{i=1}^{n} f(x_i^*)\Delta x$$

$$a = x_0 < x_1 < x_2 < \cdots < x_n = b$$

$$\begin{aligned} T(z_i) - T(z_{i+1}) &= T(z_i)\left(1 - \frac{T(z_{i+1})}{T(z_i)}\right) \\ &= T(z_i)\left(1 - \frac{e^{-\sum_{j=1}^{i} \sigma_j \delta_j}}{e^{-\sum_{j=1}^{i-1} \sigma_j \delta_j}}\right) \\ &= T(z_i)\left(1 - e^{-\sum_{j=1}^{i} \sigma_j \delta_j + \sum_{j=1}^{i-1} \sigma_j \delta_j}\right) \\ &= T_i \left(1 - e^{-\sigma_i \delta_i}\right) \end{aligned}$$

Hence, the discretized emmitted radiance is

$$\begin{aligned} \hat{\mathbf{C}}(\boldsymbol{r}) = \hat{\mathbf{C}}(z; \boldsymbol{o}, \boldsymbol{d}) &= \mathbb{E}(\mathbf{R}) \\ &= \int_{\texttt{near}}^{\texttt{far}} \mathbf{R}\, p_{\mathbf{R}}\, dz \\ &\approx \sum_{i=1}^{N} \boldsymbol{c}_i T_i \left(1 - e^{-\sigma_i \delta_i}\right) \end{aligned}$$

where $\boldsymbol{c}_i := \boldsymbol{c}(z_i; \boldsymbol{o}, \boldsymbol{d})$ is the output RGB upon MLP query at $\{z_i \mid \boldsymbol{o}, \boldsymbol{d}\}$.

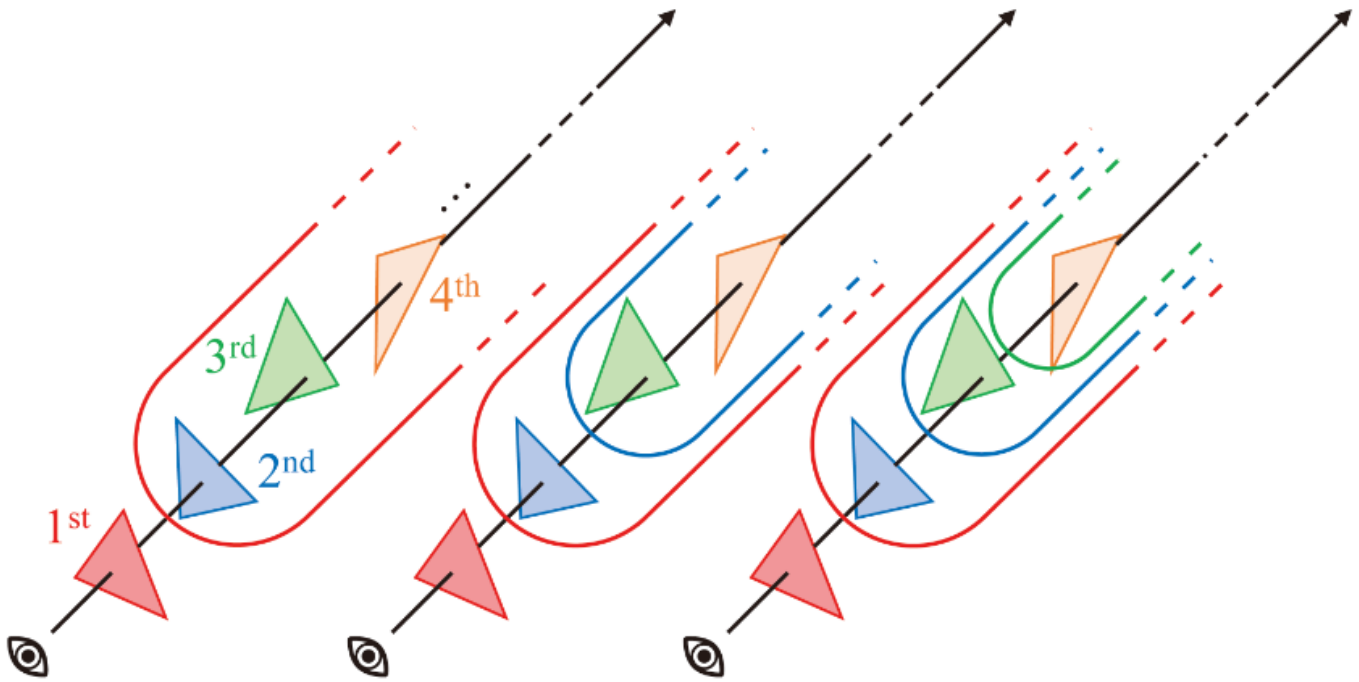
Note that if we denote $\alpha_i := 1 - e^{-\sigma_i \delta_i}$, then $\hat{\mathbf{C}}(\boldsymbol{r}) = \sum_{i=1}^{N} \alpha_i T_i \boldsymbol{c}_i$ resembles classical *alpha compositing*.

https://yconquesty.github.io/blog/ml/nerf/nerf_rendering.html#numerical-quadrature

# Comparison with Alpha Compositing

In *alpha compositing*, a parameter $\alpha$ determines the extent to which each object contributes to what is displayed in a pixel. Let $\alpha$ denote the opacity (or *pixel coverage*) of the foreground object, then a pixel showing foreground color $\boldsymbol{c}_f$ over background color $\boldsymbol{c}_b$ is composited as

$$\boldsymbol{c} = \alpha \boldsymbol{c}_f + (1 - \alpha)\boldsymbol{c}_b$$




Alpha compositing in NeRF

There are $N$ samples along each ray in NeRF. Consider the first $N-1$ samples as occluding foreground objects with opacity $\alpha_i$ and color $\boldsymbol{c}_i$, and the last sample as background, then the blended pixel value is

$$\begin{aligned}\tilde{\mathbf{C}} &= \alpha_1 \boldsymbol{c}_1 + (1-\alpha_1)\alpha_2 \boldsymbol{c}_2 \\ &\quad + (1-\alpha_1)(1-\alpha_2)\alpha_3 \boldsymbol{c}_3 \\ &\quad + (1-\alpha_1)(1-\alpha_2)(1-\alpha_3)\alpha_4 \boldsymbol{c}_4 \\ &\quad \cdots \\ &\quad + (1-\alpha_1)(1-\alpha_2)(1-\alpha_3)\cdots(1-\alpha_{N-1})\boldsymbol{c}_N \\ &= \sum_{i=1}^{N} \left( \alpha_i \boldsymbol{c}_i \prod_{j=1}^{i-1} (1-\alpha_j) \right)\end{aligned}$$

where $\alpha_0 = 0$. Recall $\hat{\mathbf{C}} = \sum_{i=1}^{N} \alpha_i T_i \boldsymbol{c}_i$, then it remains to show that $\prod_{j=1}^{i-1}(1-\alpha_j) = T_i$:

$$\begin{aligned}\prod_{j=1}^{i-1}(1-\alpha_j) &= \prod_{j=1}^{i-1} e^{-\sigma_j \delta_j} \\ &= \exp\left( \sum_{j=1}^{i-1} -\sigma_j \delta_j \right) \\ &= T_i\end{aligned}$$

concluding the proof. This manifests the elegancy of differentiable volume rendering.

https://yconquesty.github.io/blog/ml/nerf/nerf_rendering.html#numerical-quadrature

# Differentiable Rendering

Given the above renderer, a coarse training pipeline is

$$(\boldsymbol{x}, \boldsymbol{d}) \xrightarrow{\text{MLP}} (\boldsymbol{c}, \sigma) \xrightarrow{\text{discrete rendering}} \left.\begin{matrix} \text{ground truth } \mathbf{C} \\ \text{prediction } \hat{\mathbf{C}} \end{matrix}\right\} \xrightarrow{\text{MSE}} \mathcal{L} = \|\hat{\mathbf{C}} - \mathbf{C}\|_2^2$$

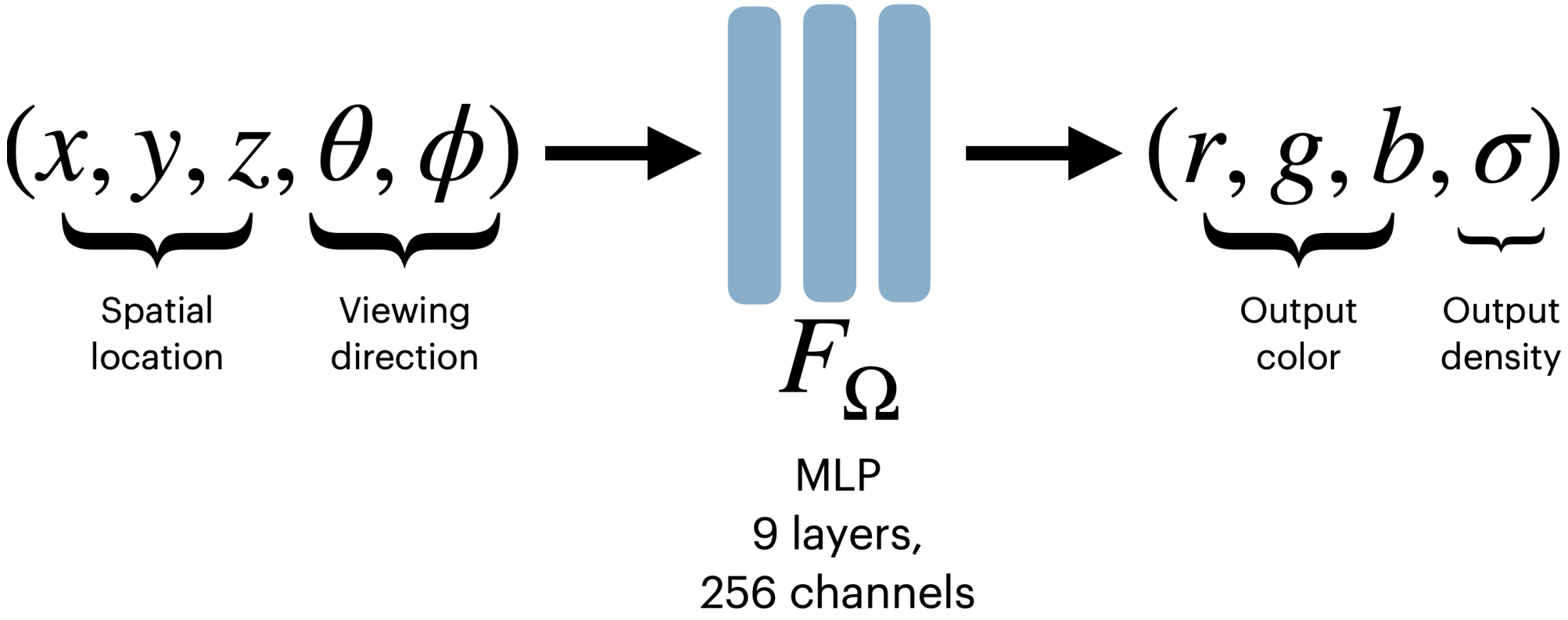
If the discrete renderer is differentiable, then we can train the *end-to-end* model through gradient descent. No suprise, given a (sorted) sequence of random samples $\boldsymbol{t} = \{t_1, t_2, \ldots, t_N\}$, the derivatives are

$$\begin{aligned}
\left.\frac{d\hat{\mathbf{C}}}{d\boldsymbol{c}_i}\right|_{\boldsymbol{t}} &= T_i \left(1 - e^{-\sigma_i \delta_i}\right) \\
\left.\frac{d\hat{\mathbf{C}}}{d\sigma_i}\right|_{\boldsymbol{t}} &= \boldsymbol{c}_i \left(\frac{dT_i}{d\sigma_i}\left(1 - e^{-\sigma_i \delta_i}\right) + T_i \frac{d}{d\sigma_i}\left(1 - e^{-\sigma_i \delta_i}\right)\right) \\
&= \mathbf{c}_i \left(\left(1 - e^{-\sigma_i \delta_i}\right) \exp\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right) \underbrace{\frac{d}{d\sigma_i}\left(-\sum_{j=1}^{i-1} \sigma_j \delta_j\right)}_{0} + T_i \left(-e^{-\sigma_i \delta_i}\right) \frac{d(-\sigma_i \delta_i)}{d\sigma_i}\right) \\
&= \delta_i T_i \boldsymbol{c}_i e^{-\sigma_i \delta_i}
\end{aligned}$$

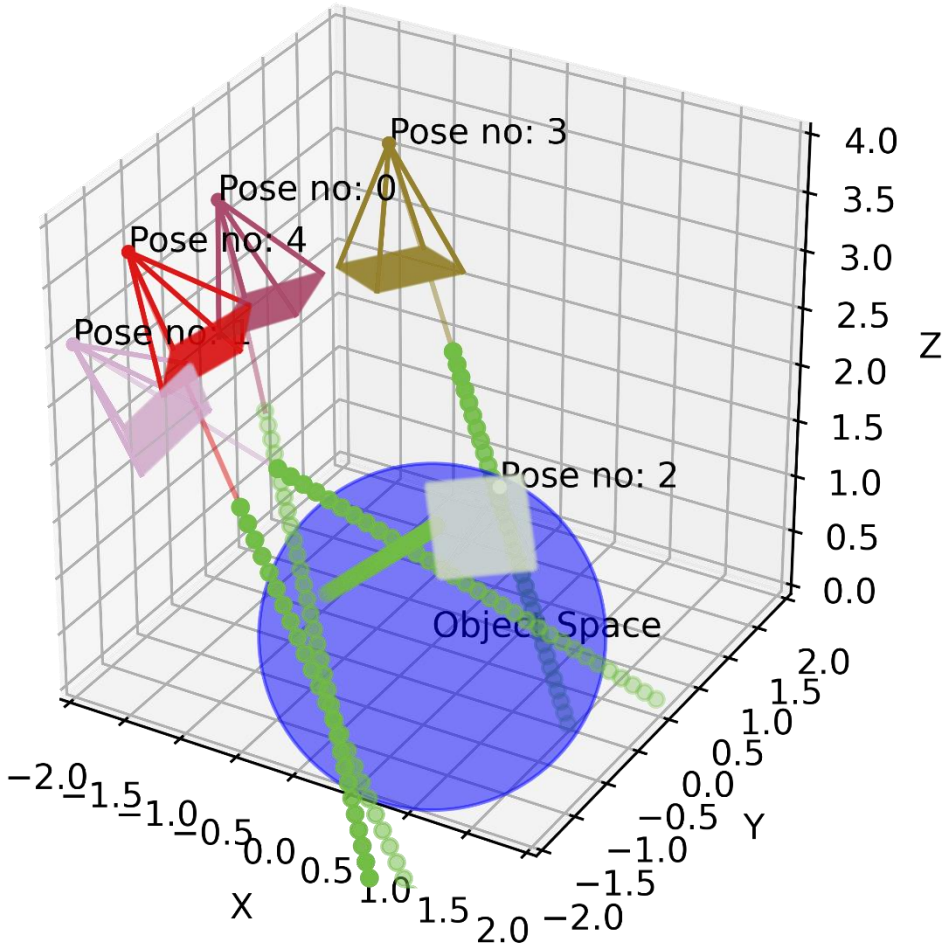
Once the renderer is differentiable, weights and biases in an MLP can be updated via the chain rule.

https://yconquesty.github.io/blog/ml/nerf/nerf_rendering.html#numerical-quadrature

# Neural Radiance Fields



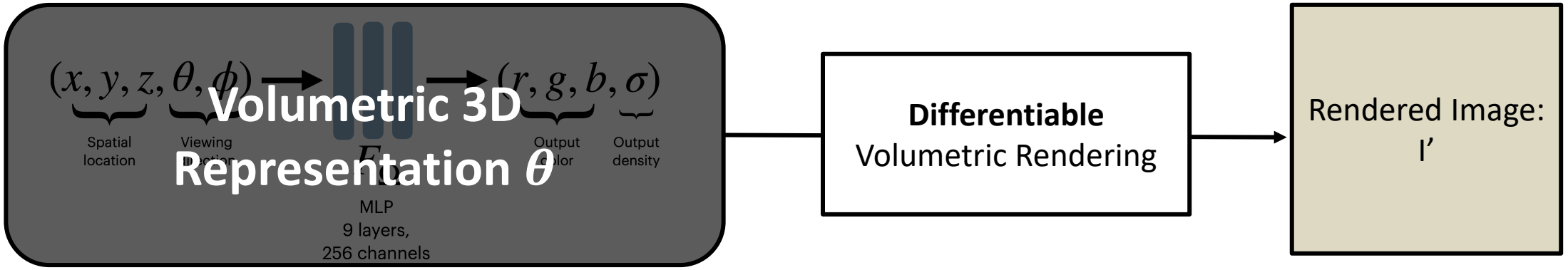
# Neural Radiance Fields




$(\underbrace{x, y, z}_{\text{Spatial location}}, \underbrace{\theta, \phi}_{\text{Viewing direction}}) \rightarrow$ MLP $F_\Omega$ (9 layers, 256 channels) $\rightarrow (\underbrace{r, g, b}_{\text{Output color}}, \underbrace{\sigma}_{\text{Output density}})$

# Neural Radiance Fields

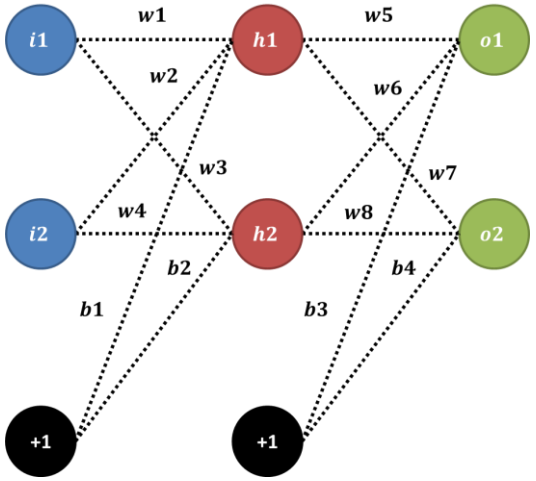
How an image is made ("Inference")



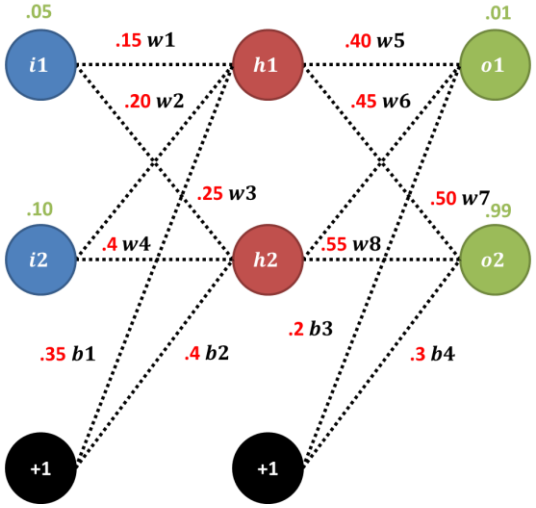

"Training" Objective (aka Analysis-by-Synthesis):

$$\min_{\theta} \| \text{Rendered Image: I'} - \text{Observed Image: I} \|_2$$

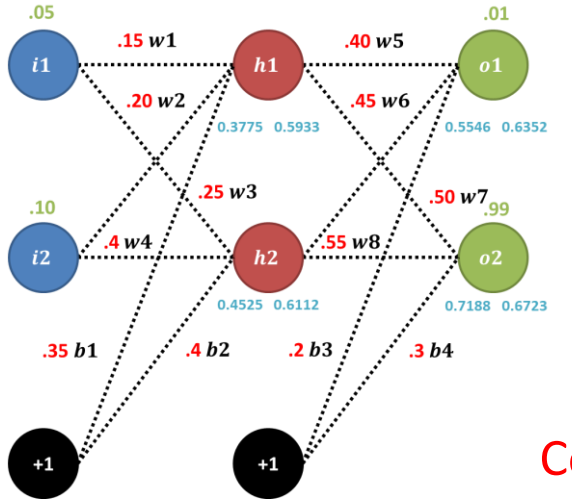
# Summary of Neural Network




**S1. Design Neural Network**

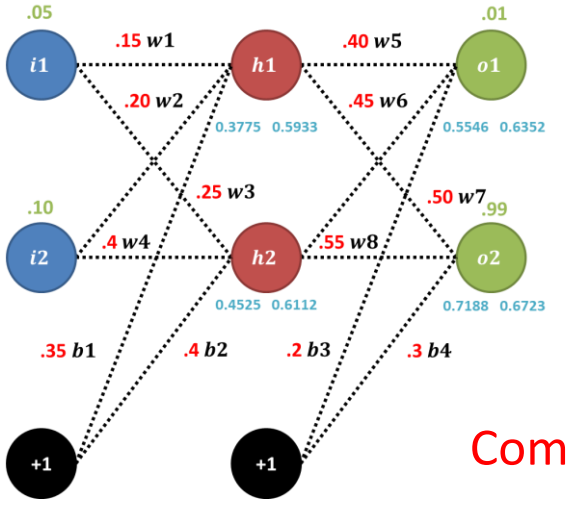



**S2. Initialization of NN**




**S3. Forward Propagation**

Compute $J(\theta)$




**S4. Backward Propagation**

Compute $\frac{\partial}{\partial \theta_j} J(\theta)$

$$\theta_{j+1} \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

**S5. Update NN**

$$\theta_{j+1} \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

**Batch (Vanilla) Gradient Descent**

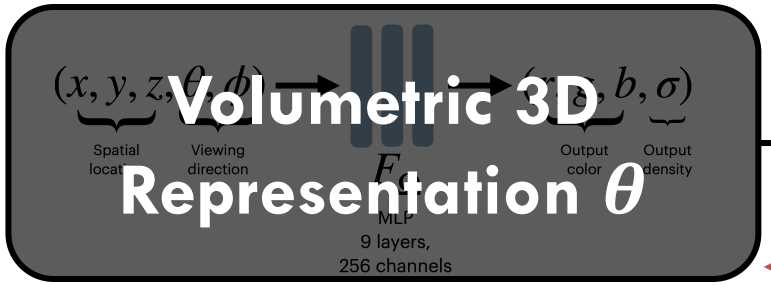
$$\theta_{j+1} \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(x^i, y^j; \theta)$$

**Stochastic Gradient Descent**

$$\theta_{j+1} \leftarrow \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(x^{\{i:i+b\}}, y^{\{i:i+b\}}; \theta)$$

**Mini-batch Gradient Descent**

# Neural Radiance Fields




**Volumetric 3D Representation $\theta$**

**Differentiable** Volumetric Rendering

Rendered Image: I'

Loss fn

Observed Image: I

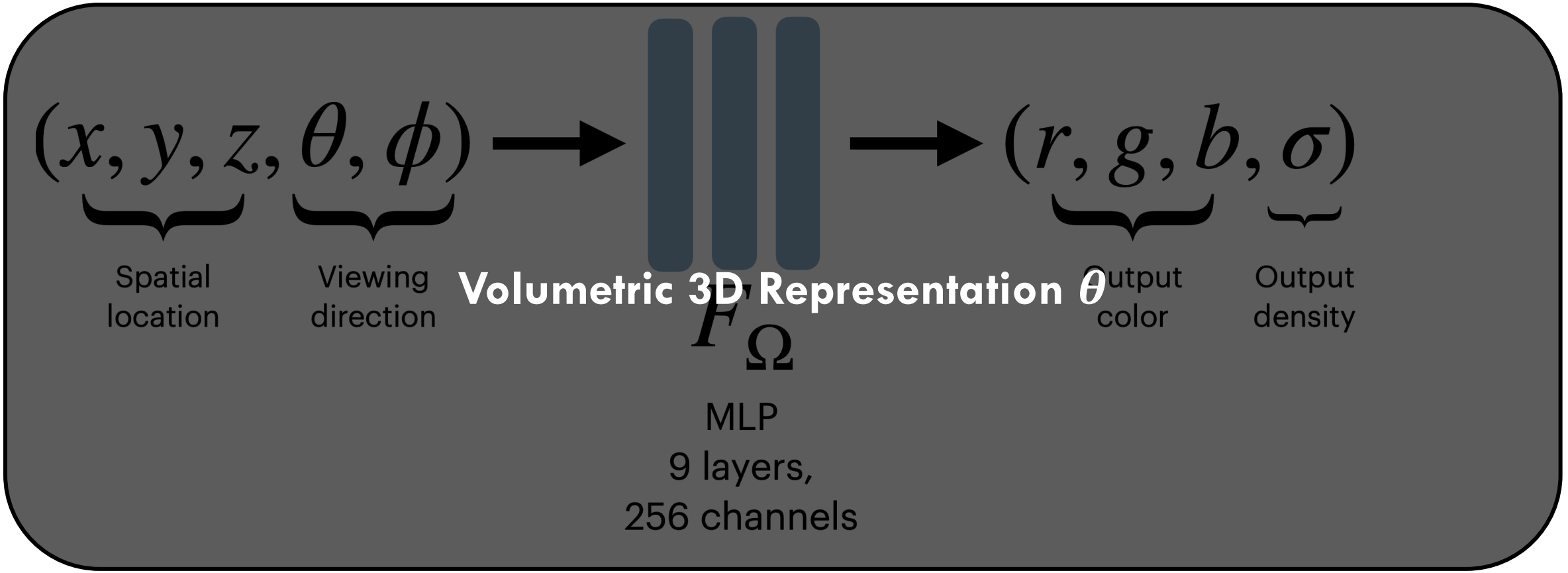
"How to change $\boldsymbol{\theta}$ to change I'"" $\frac{\partial \mathrm{I}'}{\partial \theta}$

"How to change I' to reduce Loss" $\frac{\partial L}{\partial \mathrm{I}'}$

Multiplied together

"How to change $\boldsymbol{\theta}$ to reduce Loss" $\frac{\partial L}{\partial \theta}$

# Neural Radiance Fields



# Summary of the NeRF Training Process

## Step 1: Marching the Camera Rays Through the Scene

**Input:** A set of camera poses $\{x_c, y_c, z_c, \gamma_c, \theta_c\}_n$
**Output:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$

## Step 2: Collecting Query Points

**Input:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$
**Output:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$

## Step 3: Projecting Query Points to High-Dimensional Space (Positional Encoding)

**Input:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$
**Output:** A set of query points embedded into (d)-dimensional space $\{x_1, x_2, ..., x_d\}_{n \times m \times H \times W}$

## Step 4: Neural Network Inference and Volume Rendering

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$
**Output:** Volume density for every query point $\{\sigma\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$ and $\{v_d\}_{n \times m \times H \times W}$
**Output:** RGB colour for every query point $\{\mathrm{RGB}\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) + their volume profile + RGB value $\{x_1, x_2, \ldots, x_d, \mathrm{RGB}, \sigma\}_{n \times m \times H \times W}$
**Output:** A set of rendered images (one per pose) $\{H, W\}_n$

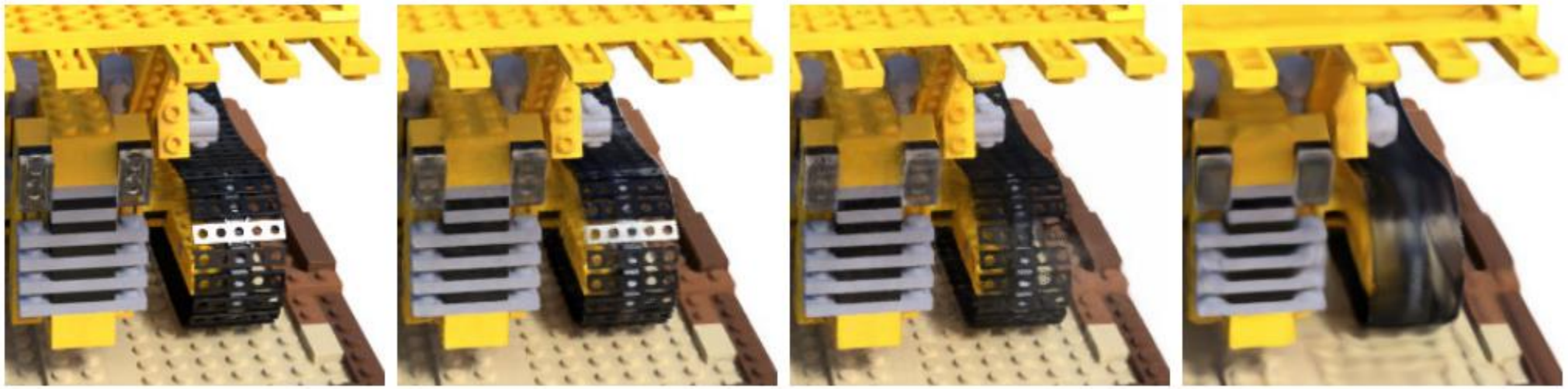
**Input:** A set of rendered images (one per pose) $\{H, W\}_n$ and a set of ground truth images (one per pose) $\{H, W\}_n^{\mathrm{gt}}$
**Output:** L2 loss between the inputs, a single scalar $\{\ell\}_n$

# Step 3: Projecting Query Points to High-Dimensional Space (Positional Encoding)

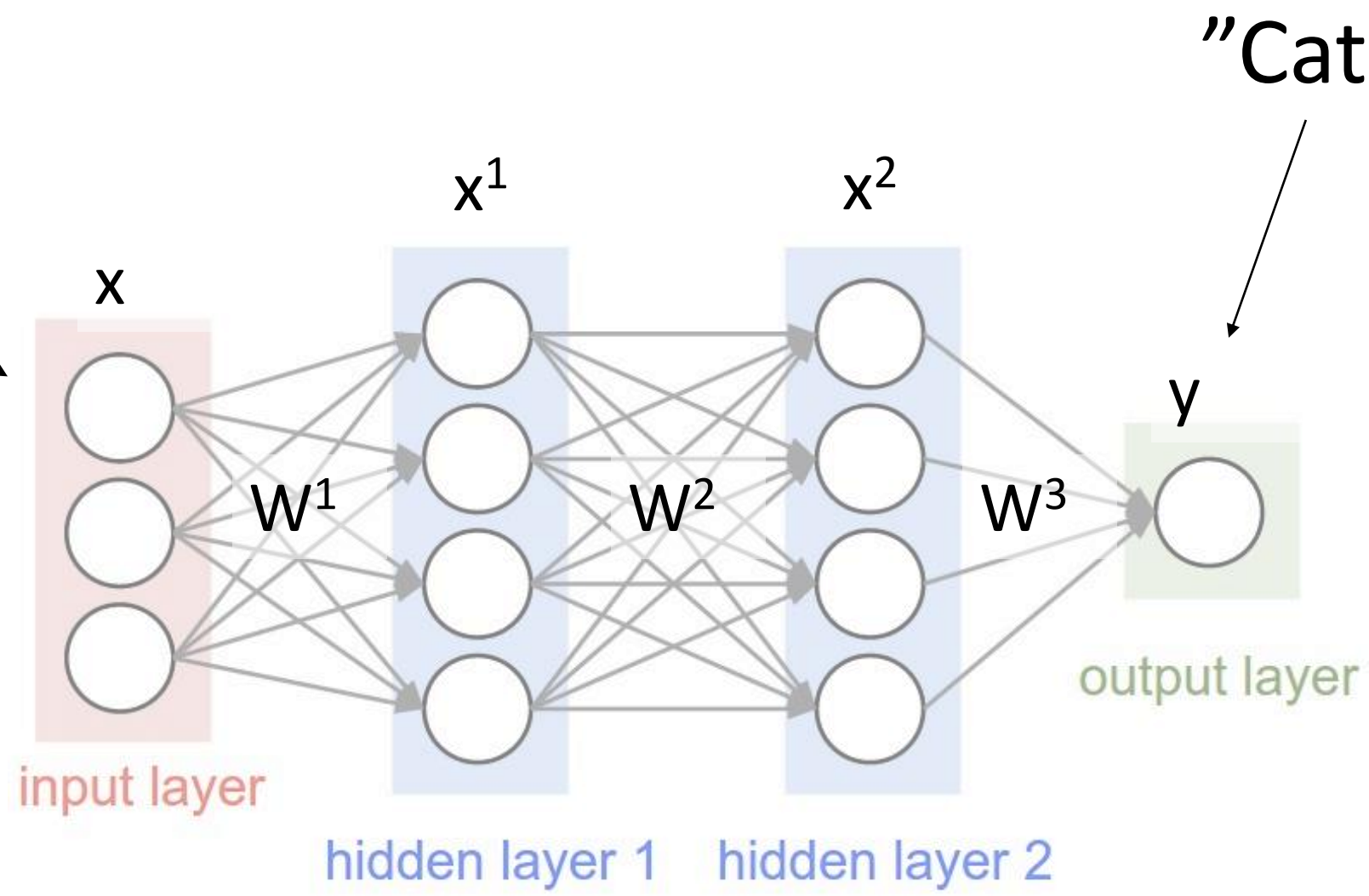
Step 3: Projecting Query Points to High-Dimensional Space (Positional Encoding)

**Input:** A set of 3D query points $\{x_p, y_p, z_p\}_{n\times m\times H\times W}$

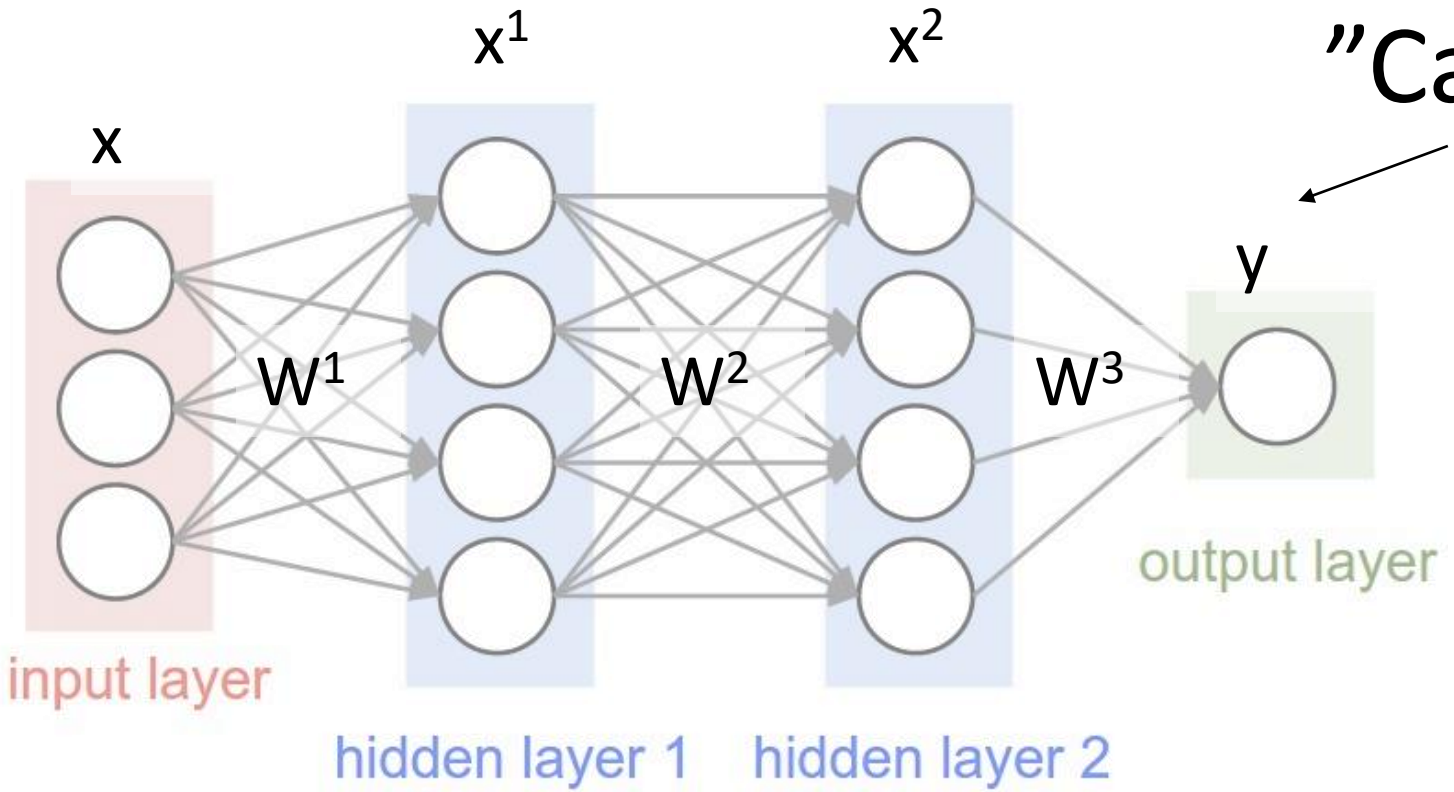
**Output:** A set of query points embedded into (d)-dimensional space $\{x_1, x_2, ..., x_d\}_{n\times m\times H\times W}$



# ML Recap: Multi-layer Perceptrons / Fully-Connected Layer



# ML Recap: Multi-layer Perceptrons / Fully-Connected Layer




In each layer:

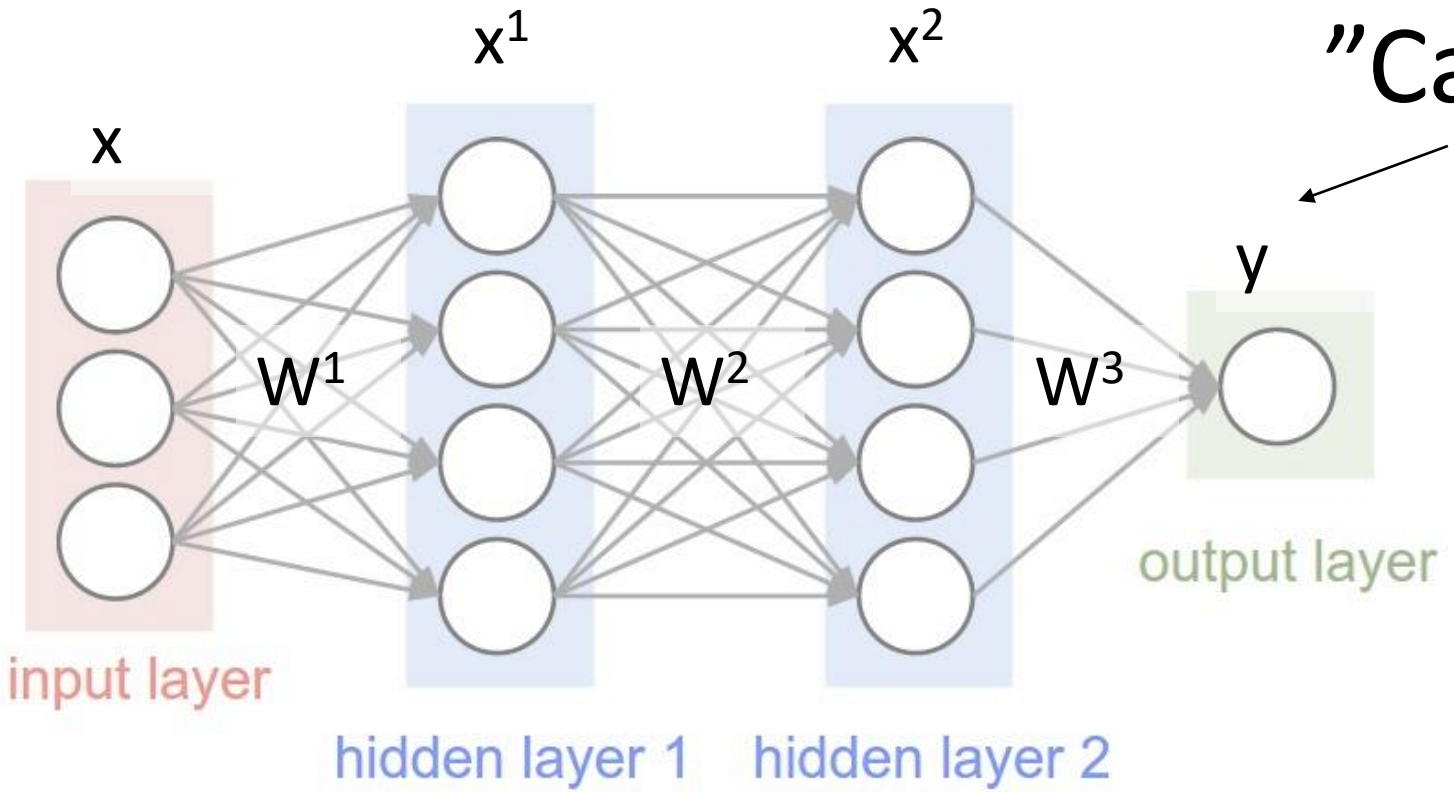
1. Linear Transform $z = W^l x^{l-1} + b$
2. Apply Non-Linearity $x^l = f(z)$

Usually

$$f = RELU(z) = \max(0, z)$$

what happens if f is identity?

# ML Recap: Multi-layer Perceptrons / Fully-Connected Layer




In each layer:

1. Linear Transform $z = W^l x^{l-1} + b$
2. Apply Non-Linearity $x^l = f(z)$

Usually

$$f = RELU(z) = \max(0, z)$$

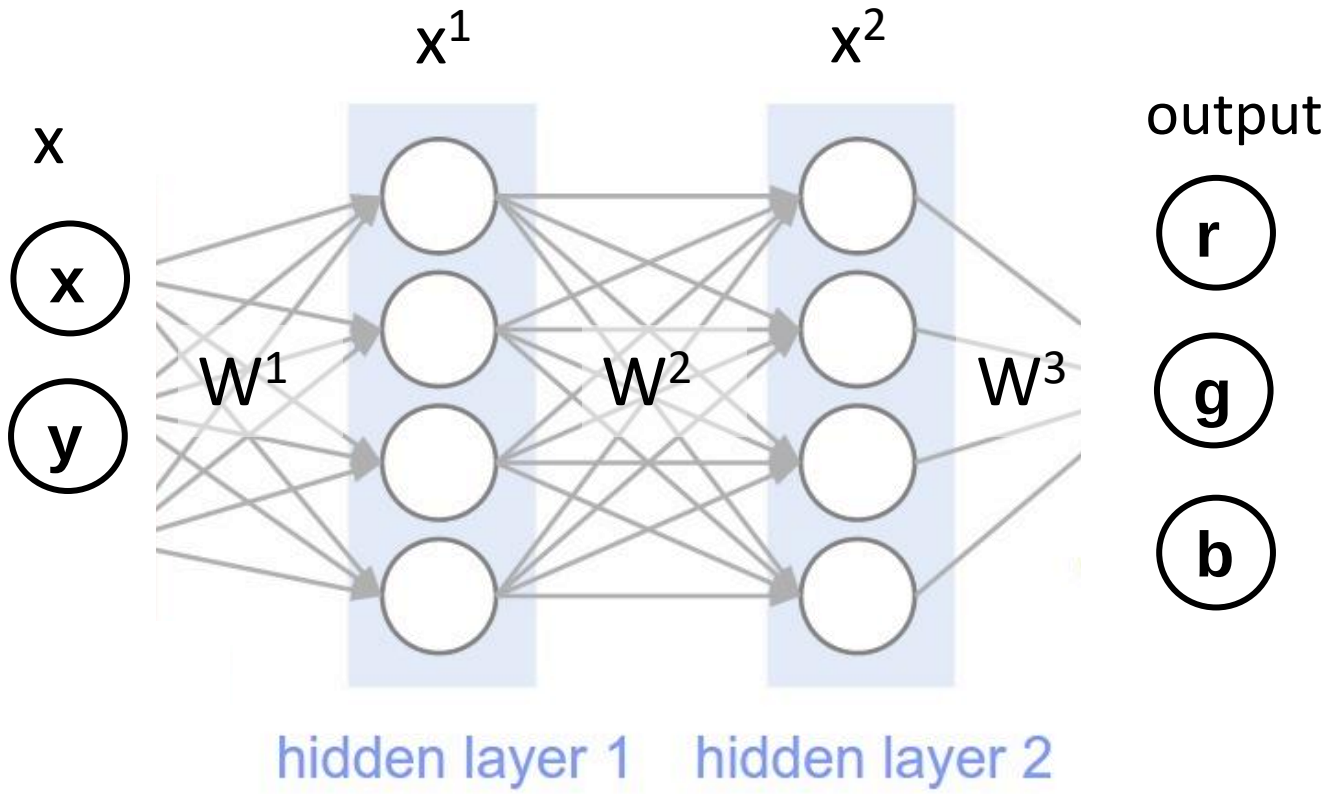
**What are the learnable parameters?**

# Storing 2D image data


Usually, we store an image as a
2D grid of RGB color values

Adapted from material from Pratul Srinivasan

# Storing 2D image data




In each layer:

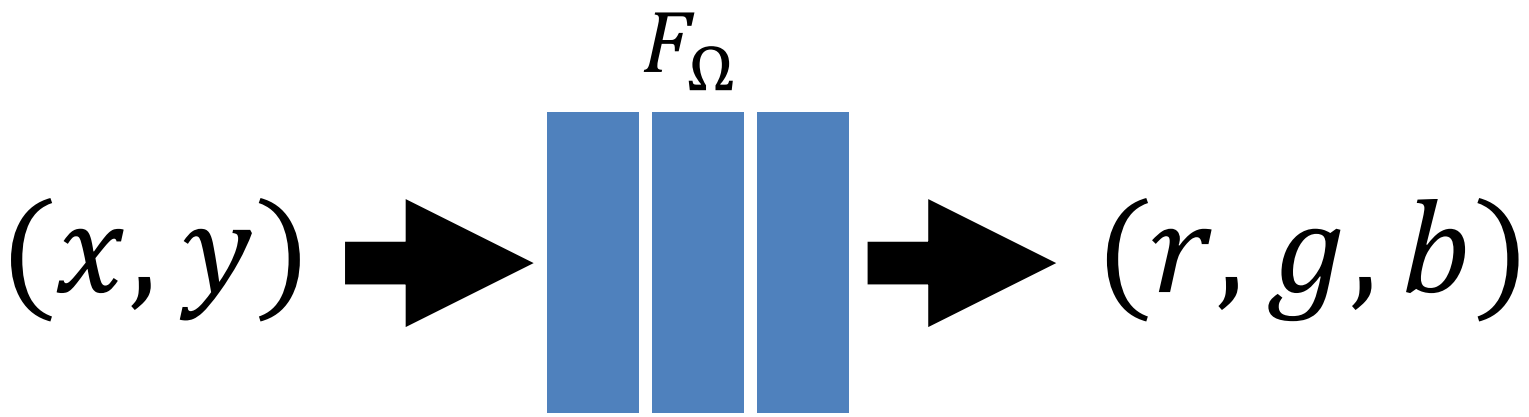
1. Linear Transform $z = W^l x^{l-1} + b$
2. Apply Non-Linearity $x^l = f(z)$

Usually

$$f = RELU(z) = \max(0, z)$$

**What are the learnable parameters?**

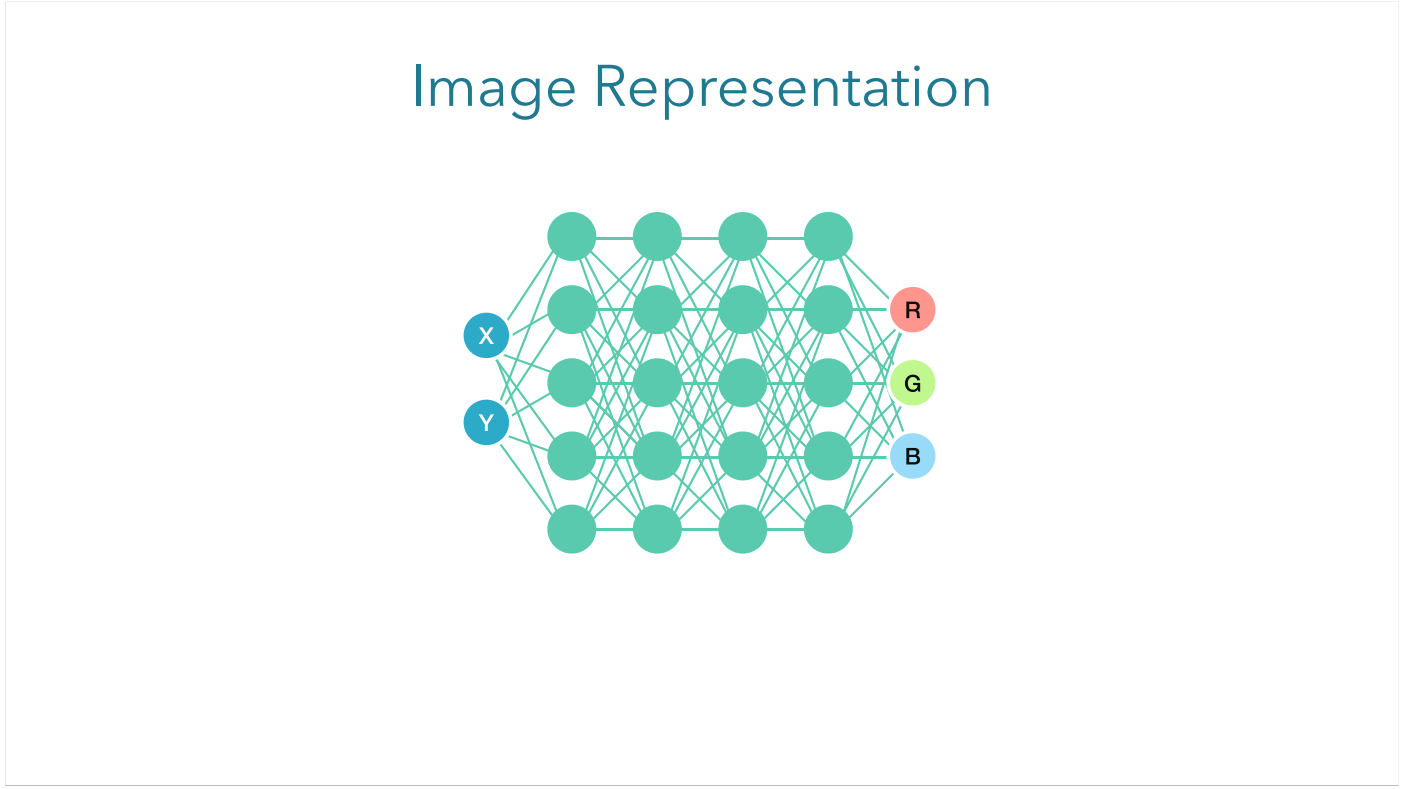
# Toy problem: Storing 2D image data




What if we train a simple fully-connected network (MLP) to do this instead?

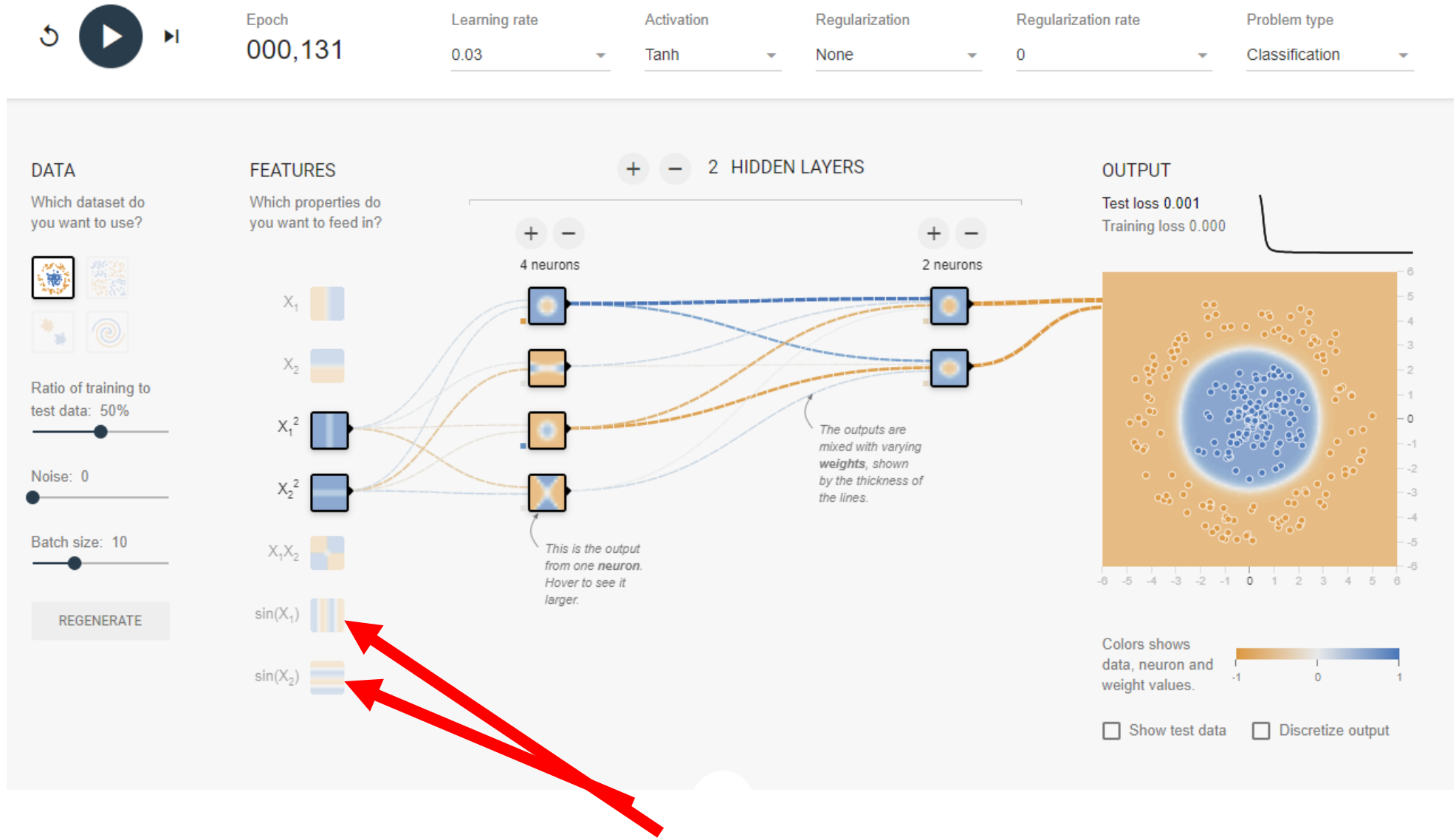
Adapted from material from Pratul Srinivasan

# what happens if you naively optimize this network




MLP output

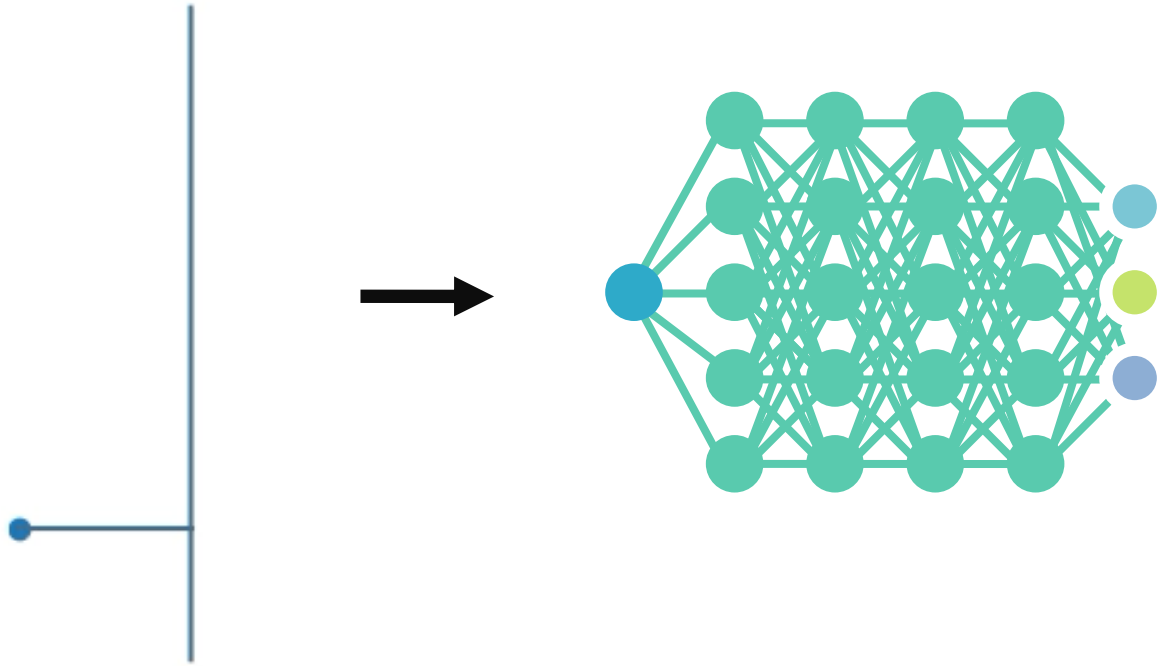
Supervision image




Recall “squared” encoding in TensorFlow Playground

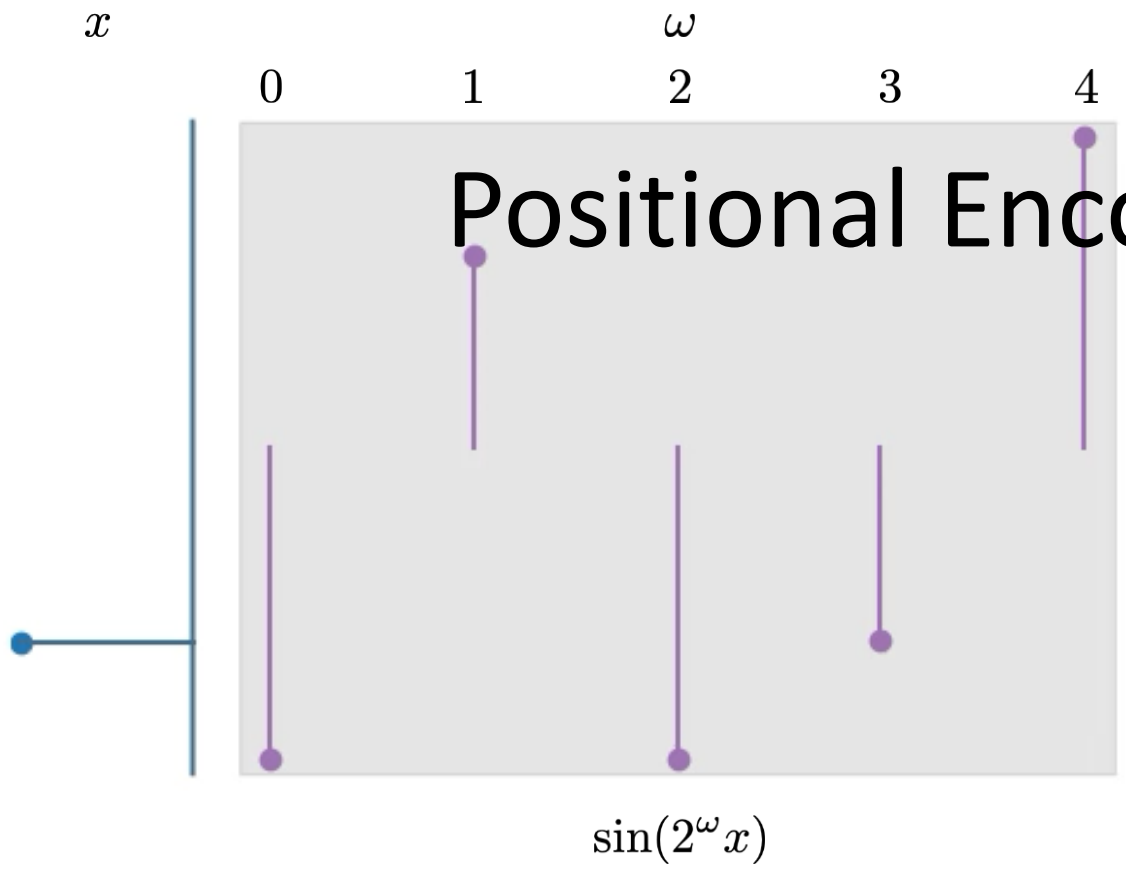
## Standard input

$x$

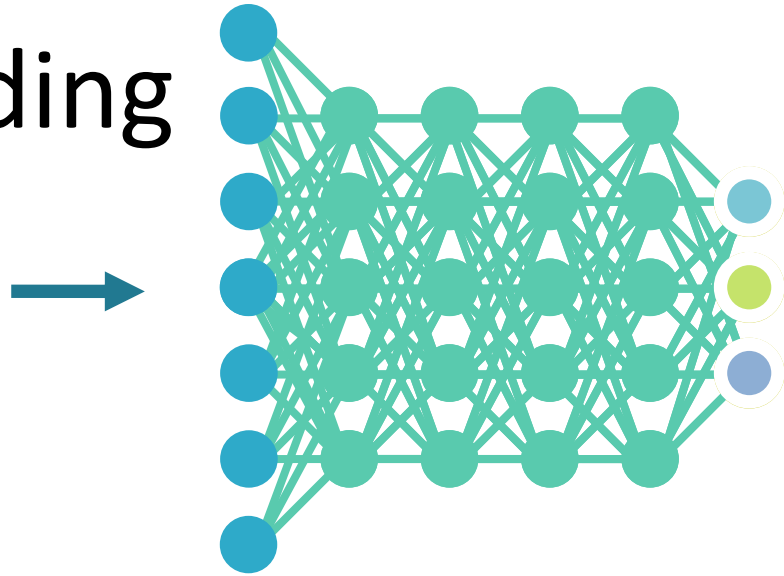


Standard input

Positionally Encoded input




Positional Encoding



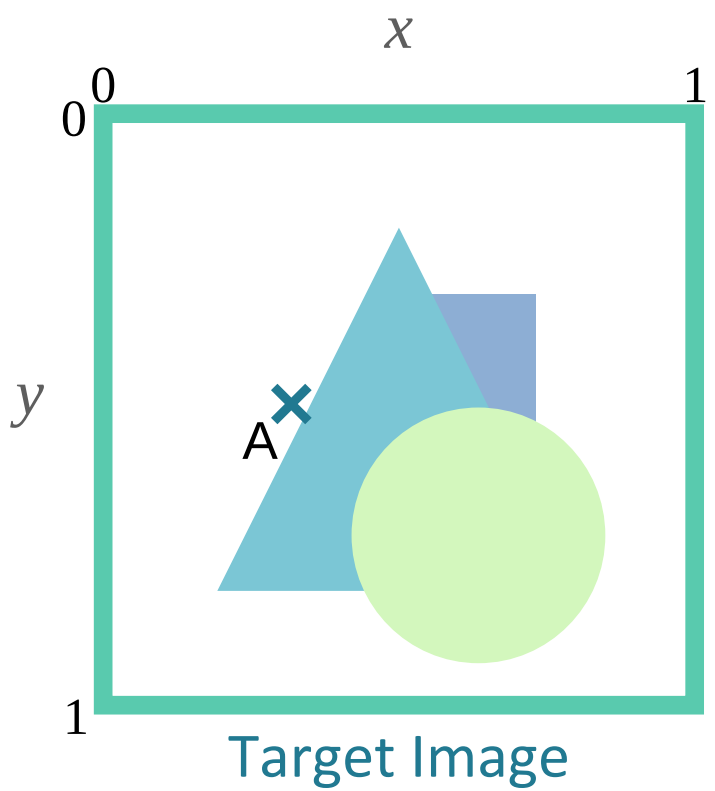
Fourier Features $\gamma(p) = \left(\sin\left(2^0\pi p\right), \cos\left(2^0\pi p\right), \cdots, \sin\left(2^{L-1}\pi p\right), \cos\left(2^{L-1}\pi p\right)\right)$

# Why Does Positional Encoding Help?

Target Image

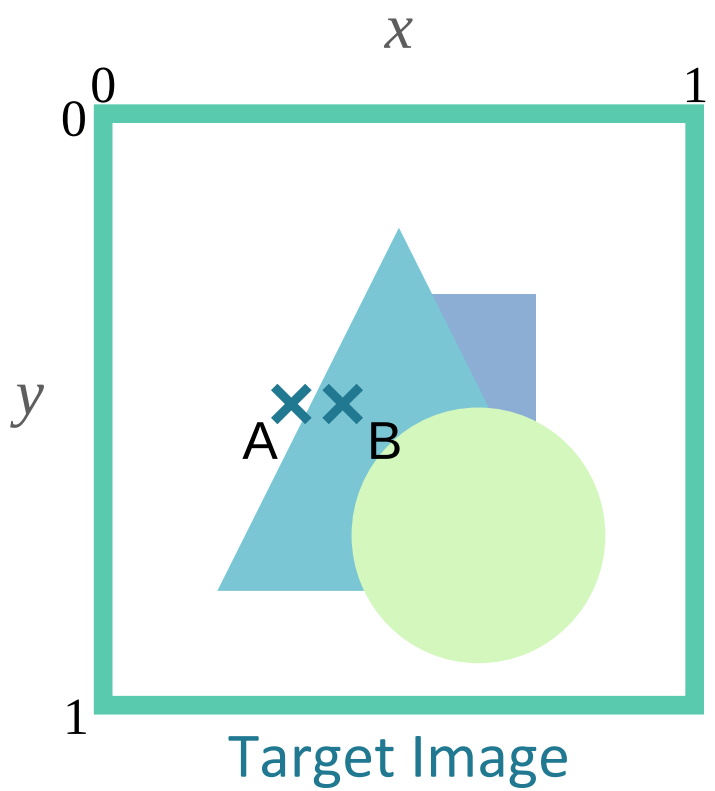
# Why Does Positional Encoding Help?

| | Input | | Target |
|---|---|---|---|
| | $x$ | $y$ | |
| A | .36 | .5 | |

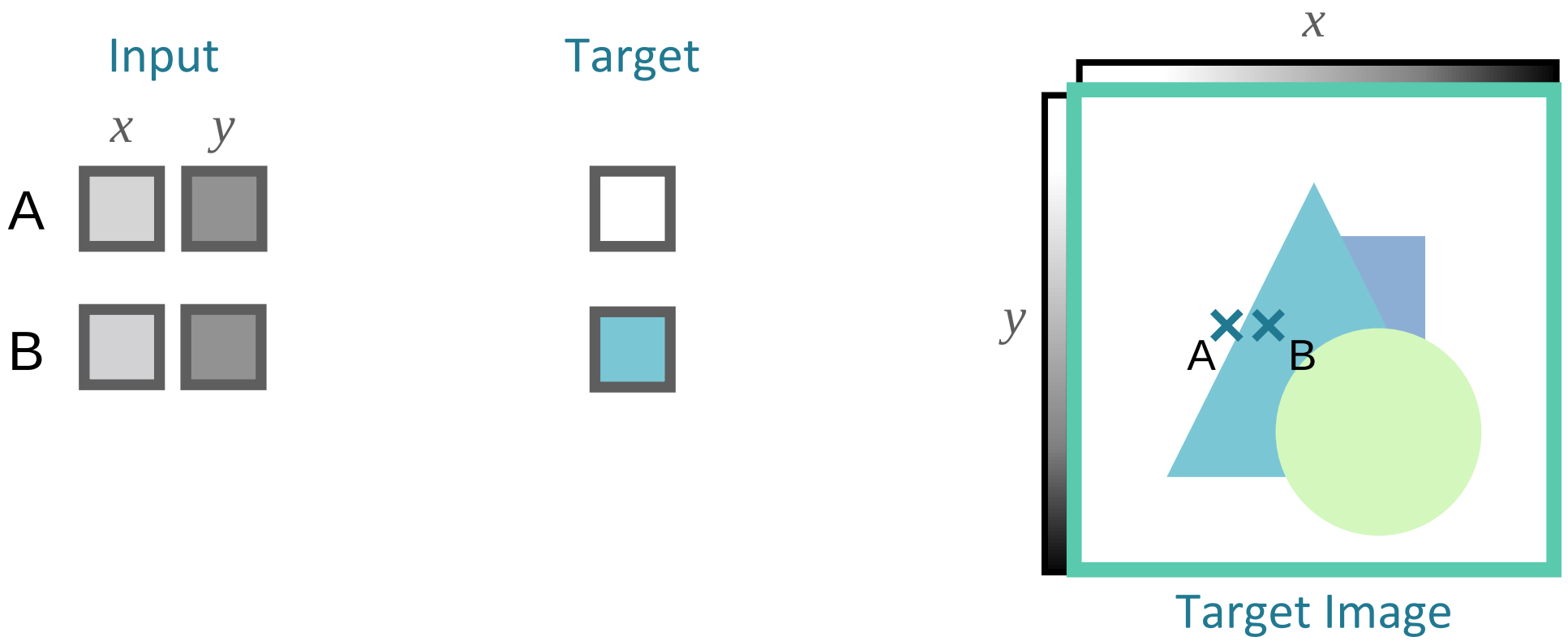


# Why Does Positional Encoding Help?

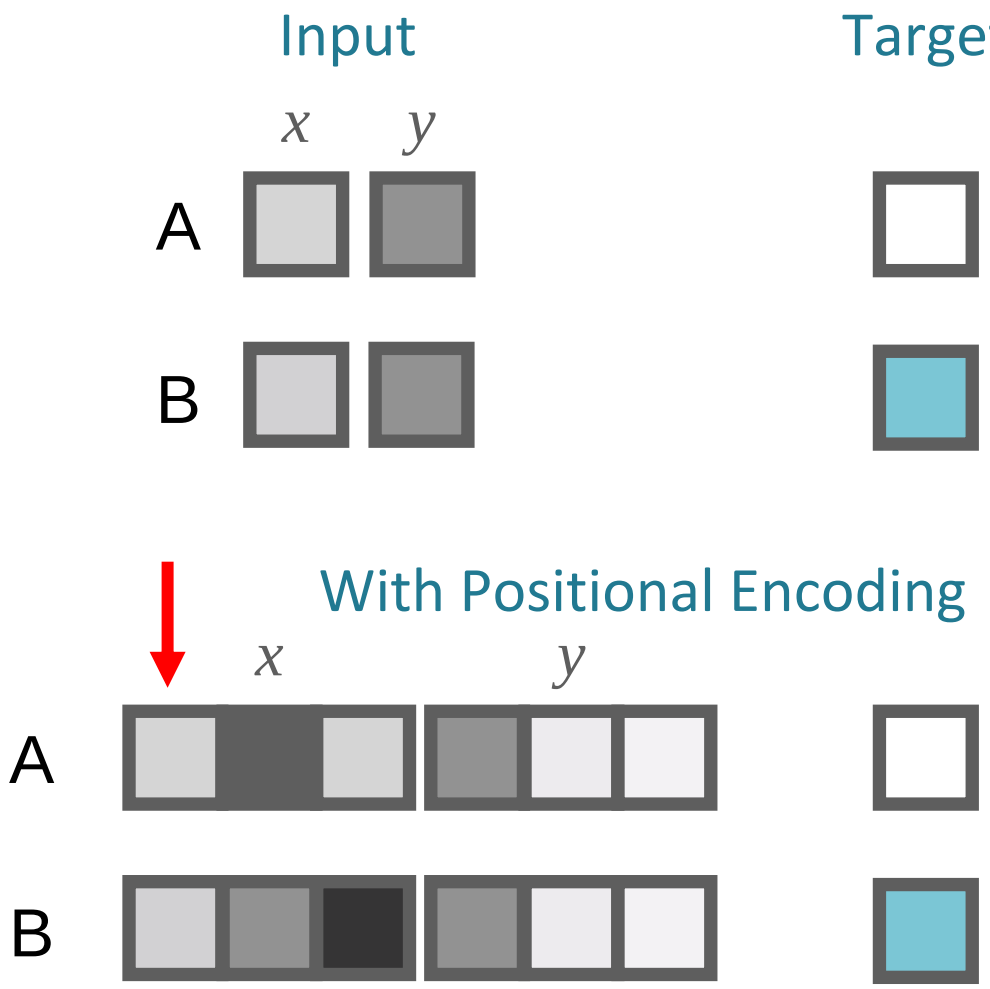
| | Input | | Target |
|---|---|---|---|
| | $x$ | $y$ | |
| A | .36 | .5 | |
| B | .38 | .5 | |

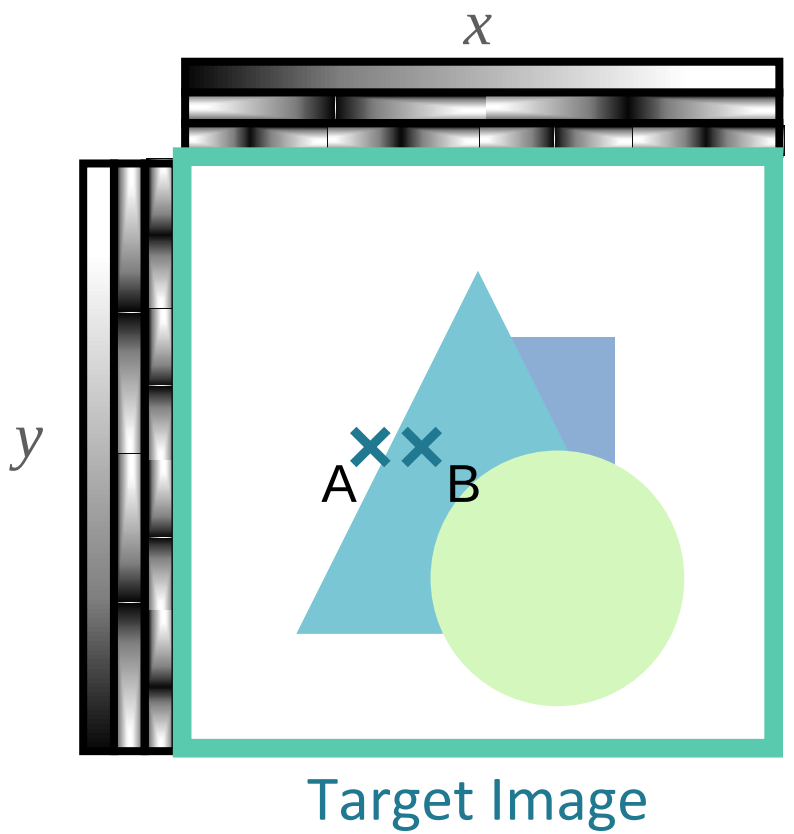


# Why Does Positional Encoding Help?



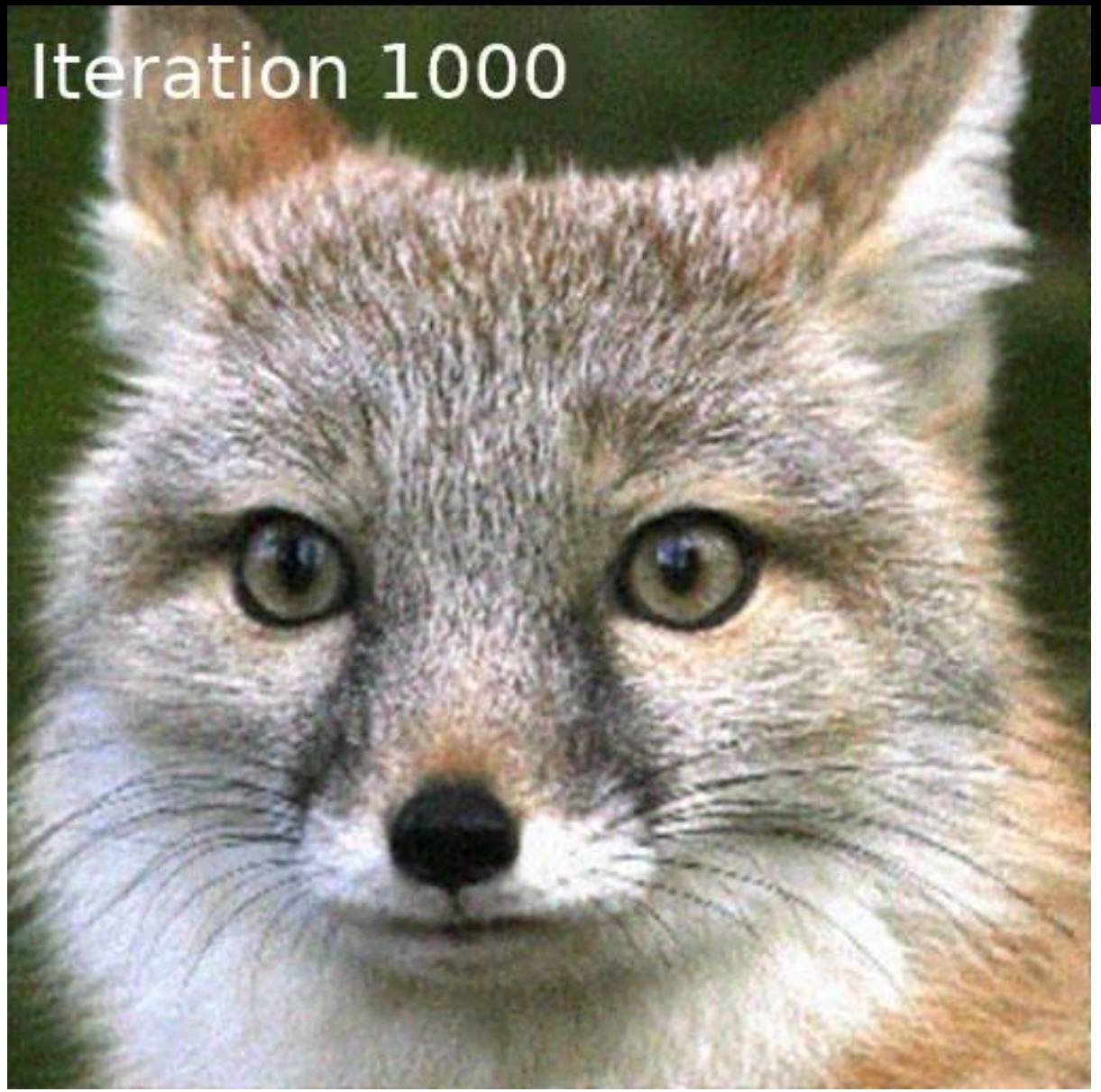
# Why Does Positional Encoding Help?




Target Image


Standard MLP




MLP with Fourier features

# Summary of the NeRF Training Process

## Step 1: Marching the Camera Rays Through the Scene

**Input:** A set of camera poses $\{x_c, y_c, z_c, \gamma_c, \theta_c\}_n$
**Output:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$

## Step 2: Collecting Query Points

**Input:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$
**Output:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$

## Step 3: Projecting Query Points to High-Dimensional Space (Positional Encoding)

**Input:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$
**Output:** A set of query points embedded into (d)-dimensional space $\{x_1, x_2, ..., x_d\}_{n \times m \times H \times W}$

## Step 4: Neural Network Inference and Volume Rendering

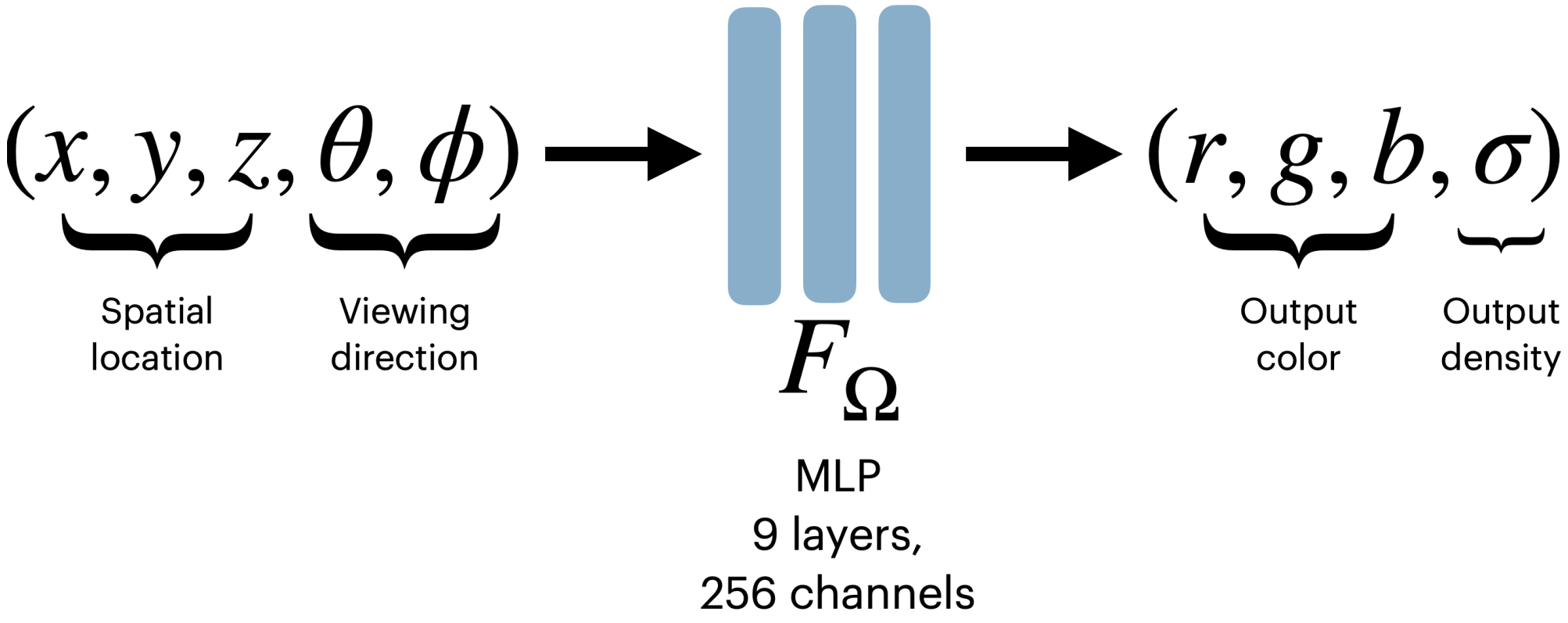
**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$
**Output:** Volume density for every query point $\{\sigma\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$ and $\{v_d\}_{n \times m \times H \times W}$
**Output:** RGB colour for every query point $\{\mathrm{RGB}\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) + their volume profile + RGB value $\{x_1, x_2, \ldots, x_d, \mathrm{RGB}, \sigma\}_{n \times m \times H \times W}$
**Output:** A set of rendered images (one per pose) $\{H, W\}_n$

**Input:** A set of rendered images (one per pose) $\{H, W\}_n$ and a set of ground truth images (one per pose) $\{H, W\}_n^{\mathrm{gt}}$
**Output:** L2 loss between the inputs, a single scalar $\{\ell\}_n$

# Neural Network Inference and Volume Rendering



# Changing Colors! But Not Volume Density

## NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis


Ben Mildenhall*

UC Berkeley

Pratul Srinivasan*

UC Berkeley

Google

Matt Tancik*

UC Berkeley

Jon Barron

Google Research

Google

Ravi Ramamoorthi

UC San Diego

UC San Diego

Ren Ng

UC Berkeley


1

# Neural Network Inference and Volume Rendering

direction as a 3D Cartesian unit vector $\mathbf{d}$. We approximate this continuous 5D scene representation with an MLP network $F_\Theta : (\mathbf{x}, \mathbf{d}) \rightarrow (\mathbf{c}, \sigma)$ and optimize its weights $\Theta$ to map from each input 5D coordinate to its corresponding volume density and directional emitted color.

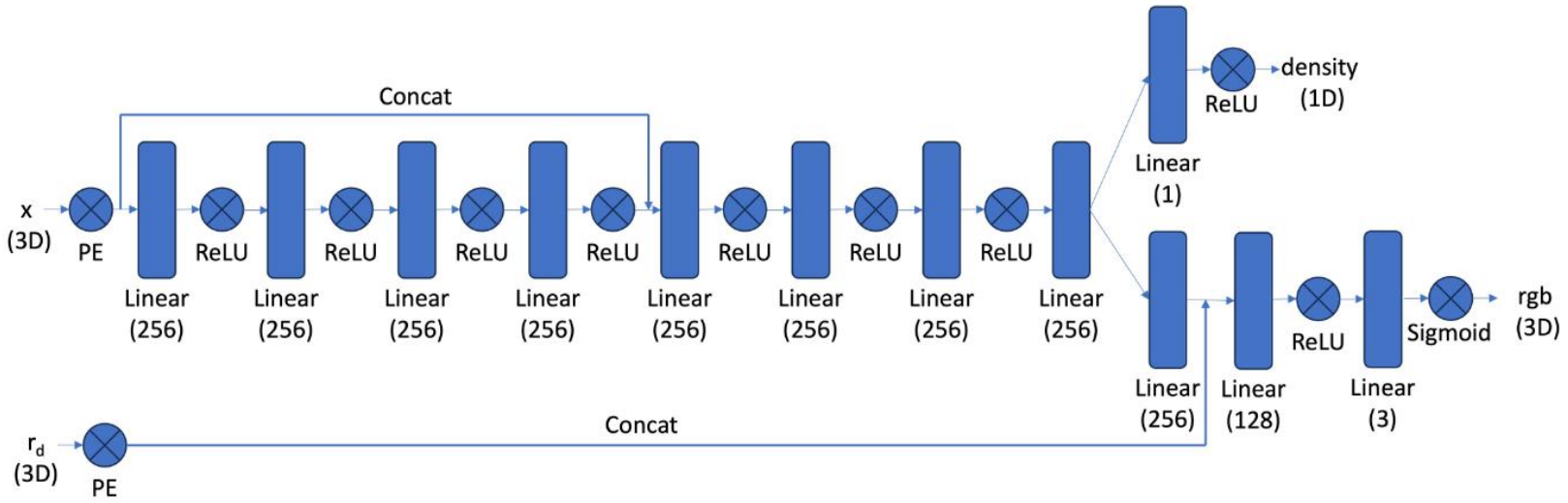
We encourage the representation to be multiview consistent by restricting the network to predict the volume density $\sigma$ as a function of only the location $\mathbf{x}$, while allowing the RGB color $\mathbf{c}$ to be predicted as a function of both location and viewing direction. To accomplish this, the MLP $F_\Theta$ first processes the input 3D coordinate $\mathbf{x}$ with 8 fully-connected layers (using ReLU activations and 256 channels per layer), and outputs $\sigma$ and a 256-dimensional feature vector. This feature vector is then concatenated with the camera ray's viewing direction and passed to one additional fully-connected layer (using a ReLU activation and 128 channels) that output the view-dependent RGB color.

See Fig. 3 for an example of how our method uses the input viewing direction to represent non-Lambertian effects. As shown in Fig. 4, a model trained without view dependence (only $\mathbf{x}$ as input) has difficulty representing specularities.

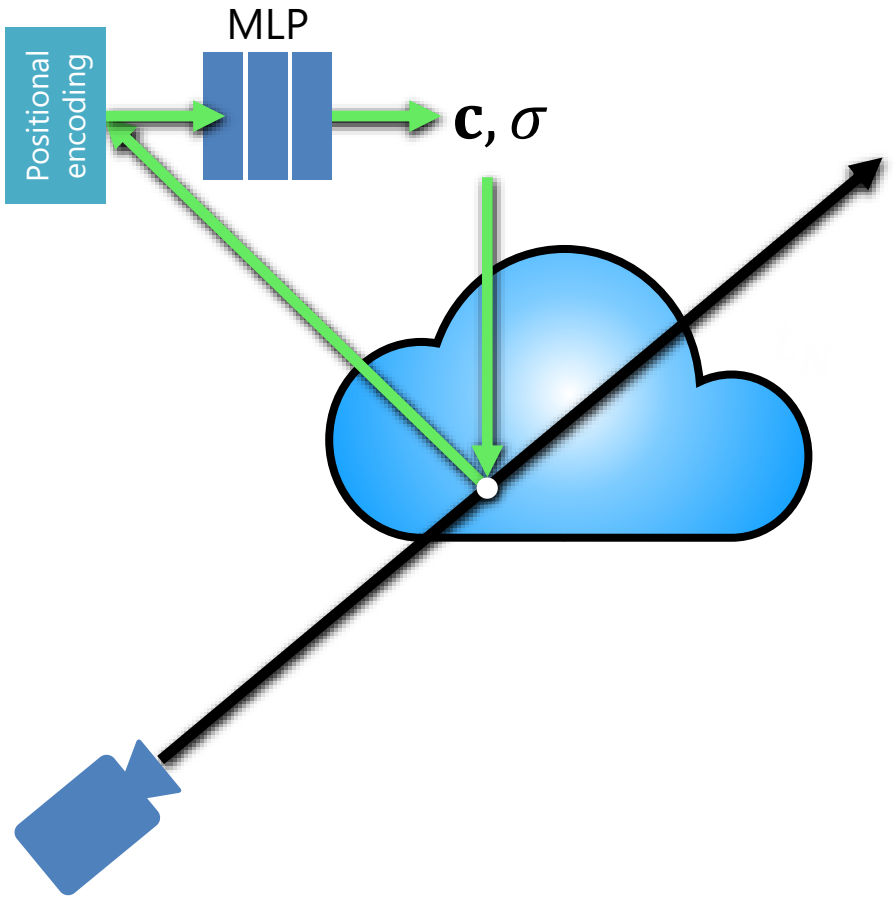
## Step 4: Neural Network Inference and Volume Rendering

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$
**Output:** Volume density for every query point $\{\sigma\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$ and $\{v_d\}_{n \times m \times H \times W}$
**Output:** RGB colour for every query point $\{\mathrm{RGB}\}_{n \times m \times H \times W}$

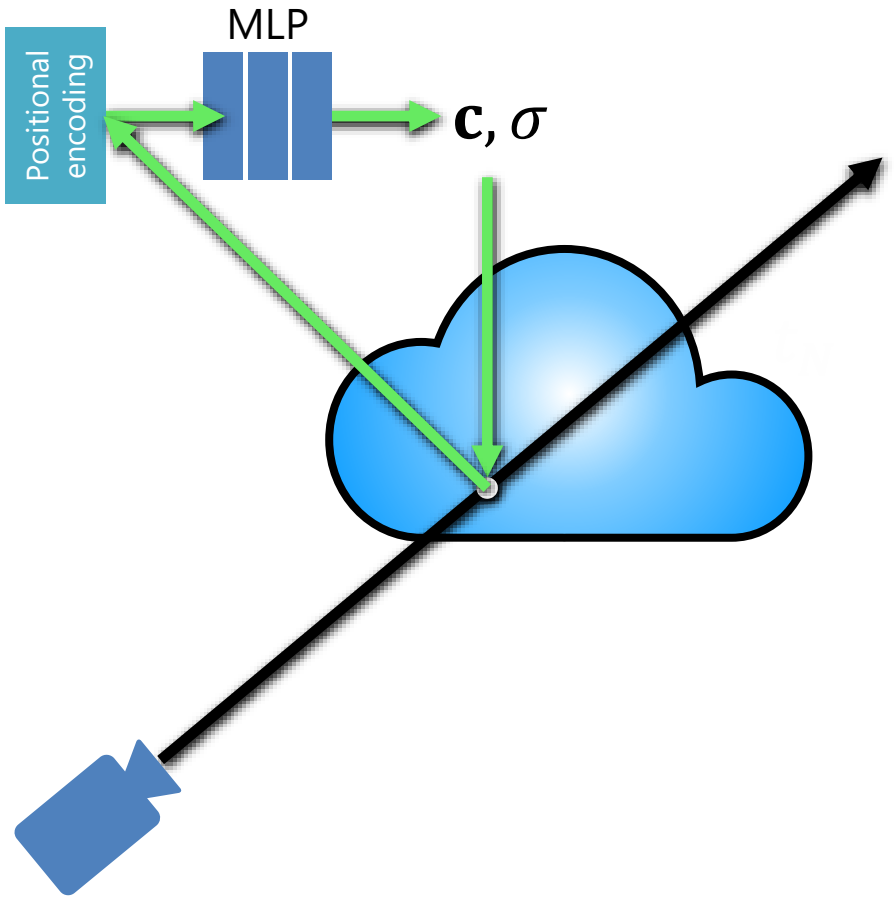


# How do We Store the Values of $\mathbf{c},\sigma$ at Each Point in Space?



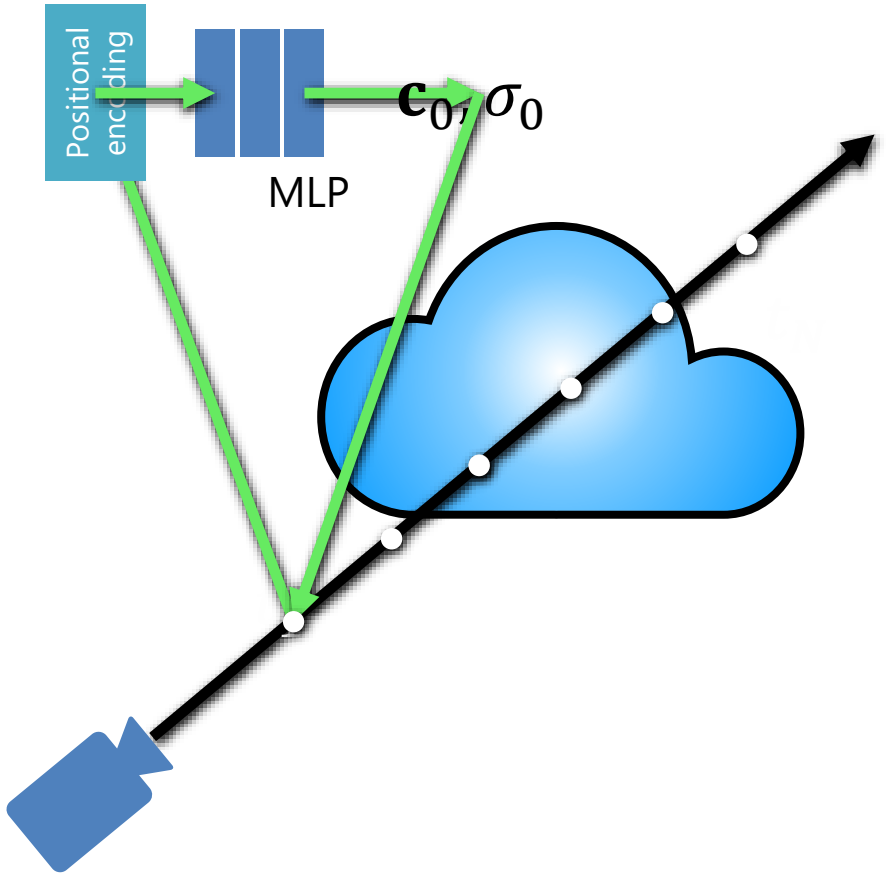

Adapted from material from Pratul Srinivasan

# How do We Store the Values of $\mathbf{c}$,$\sigma$ at Each Point in Space?



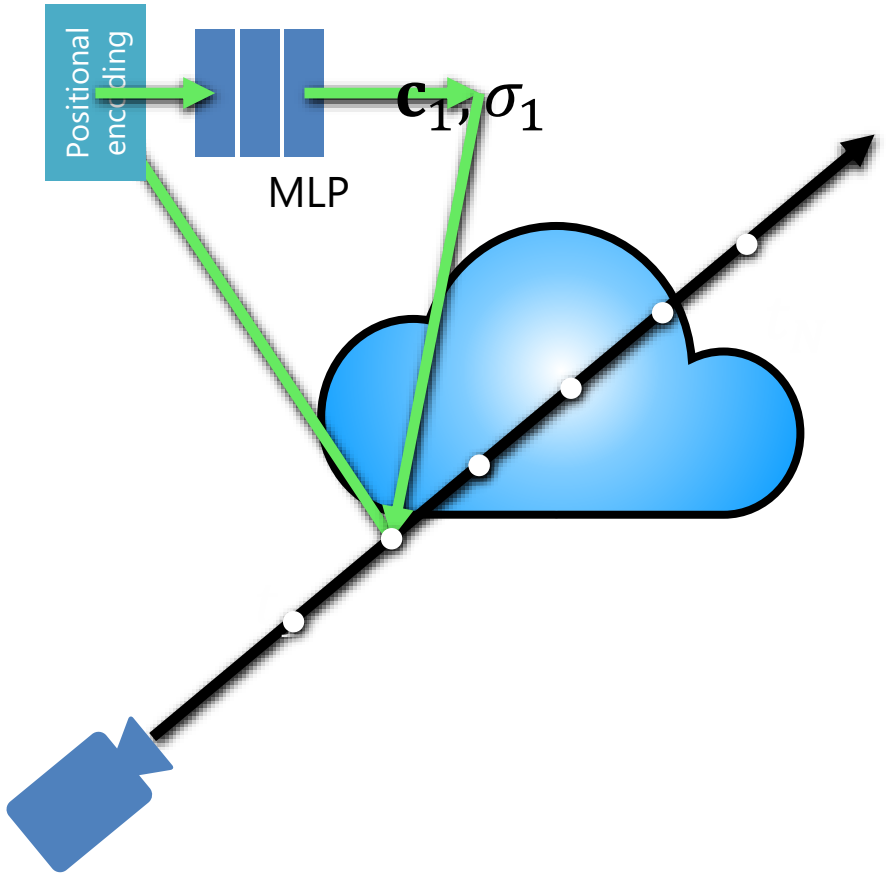

Adapted from material from Pratul Srinivasan

# How do We Store the Values of $\mathbf{c},\sigma$ at Each Point in Space?



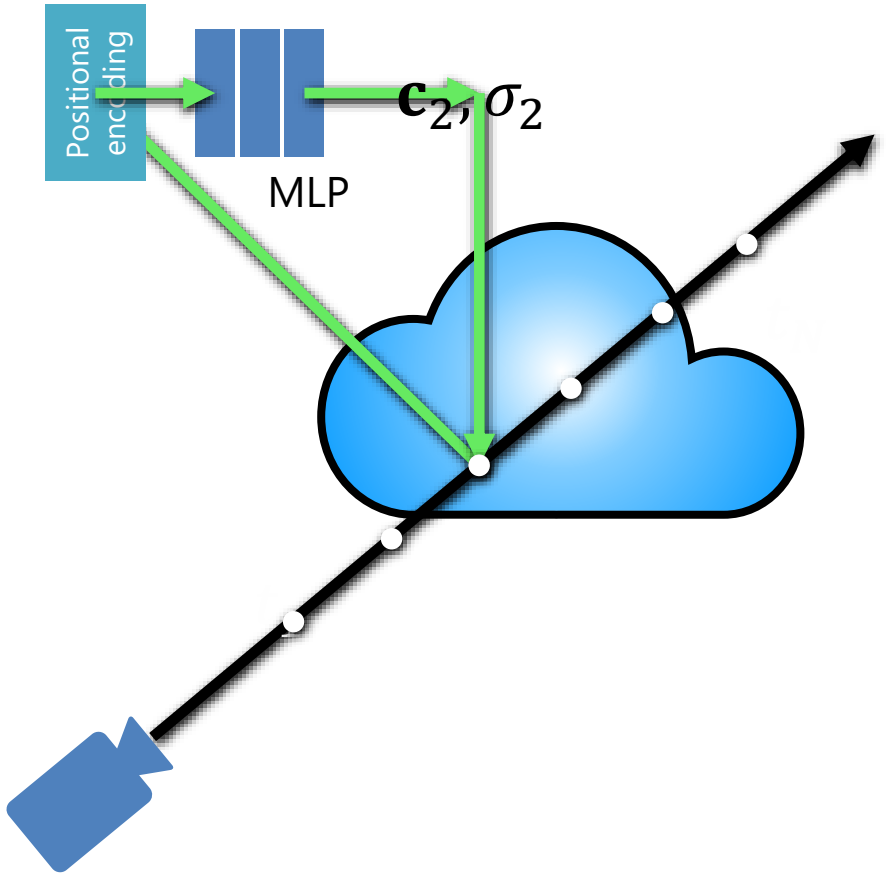

Adapted from material from Pratul Srinivasan

# How do We Store the Values of $\mathbf{c},\sigma$ at Each Point in Space?



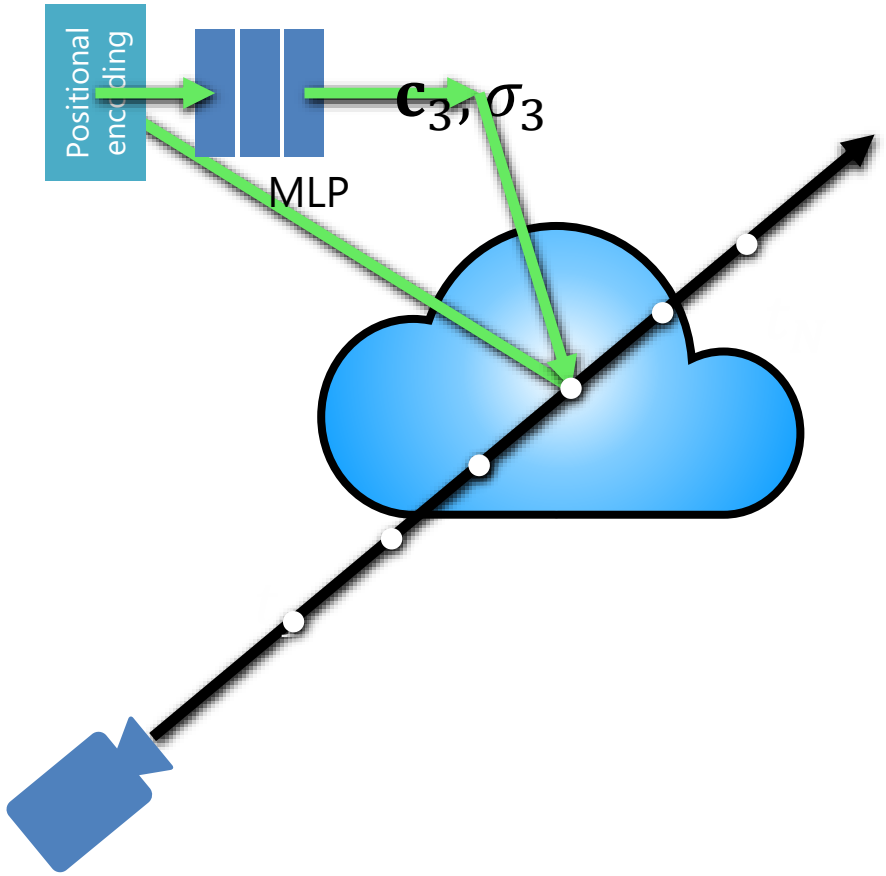

Adapted from material from Pratul Srinivasan

# How do We Store the Values of $\mathbf{c},\sigma$ at Each Point in Space?



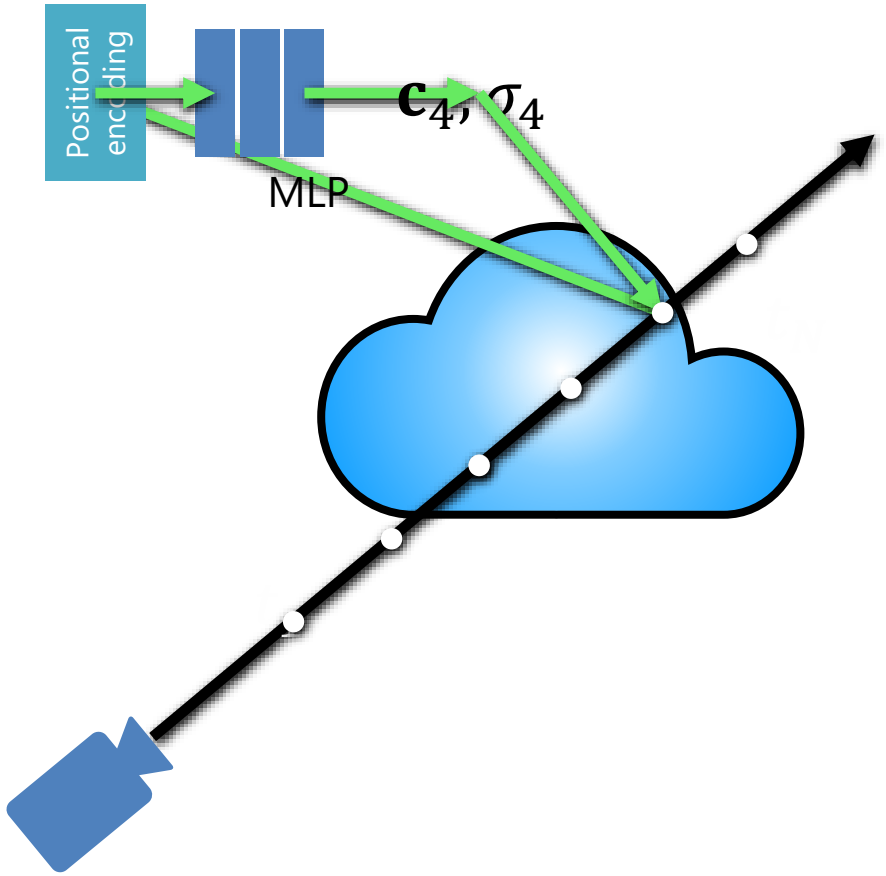

Adapted from material from Pratul Srinivasan

# How do We Store the Values of $\mathbf{c},\sigma$ at Each Point in Space?



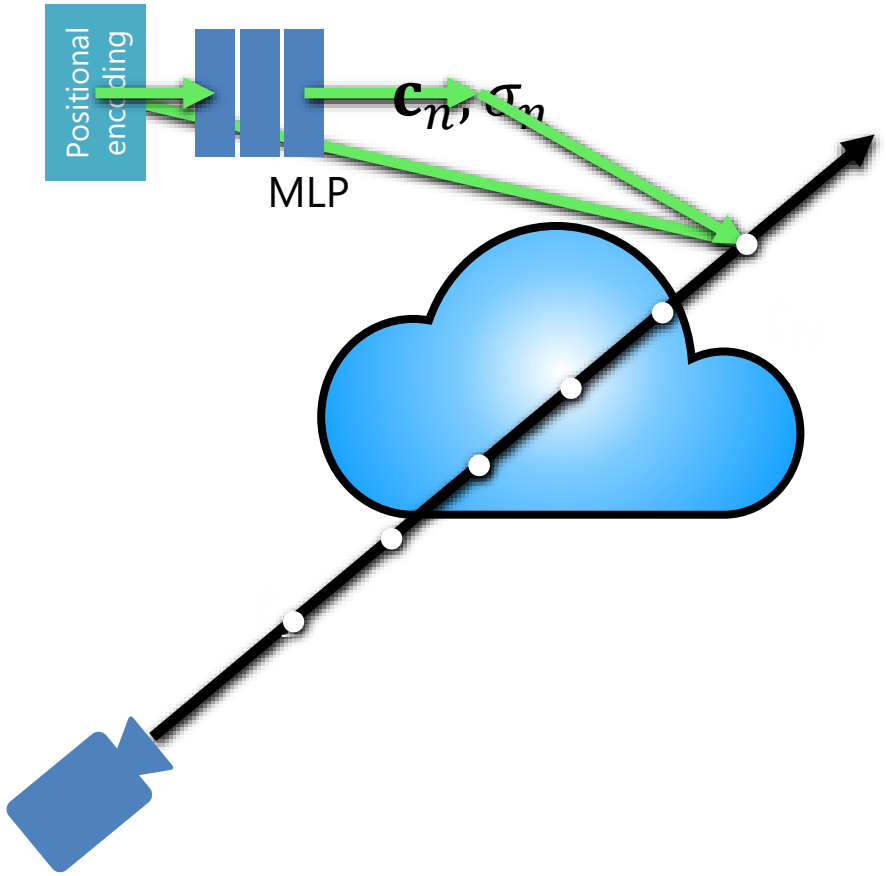

Adapted from material from Pratul Srinivasan

# How do We Store the Values of $\mathbf{c}$,$\sigma$ at Each Point in Space?



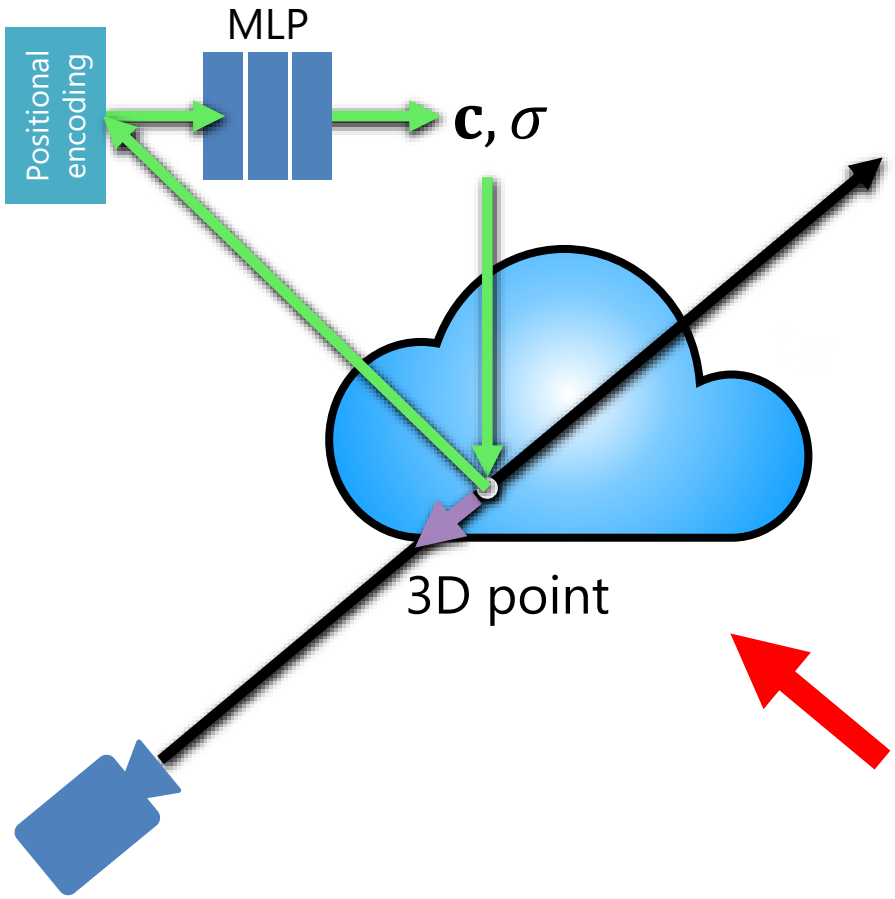

Adapted from material from Pratul Srinivasan

# How do We Store the Values of $\mathbf{c},\sigma$ at Each Point in Space?




Adapted from material from Pratul Srinivasan

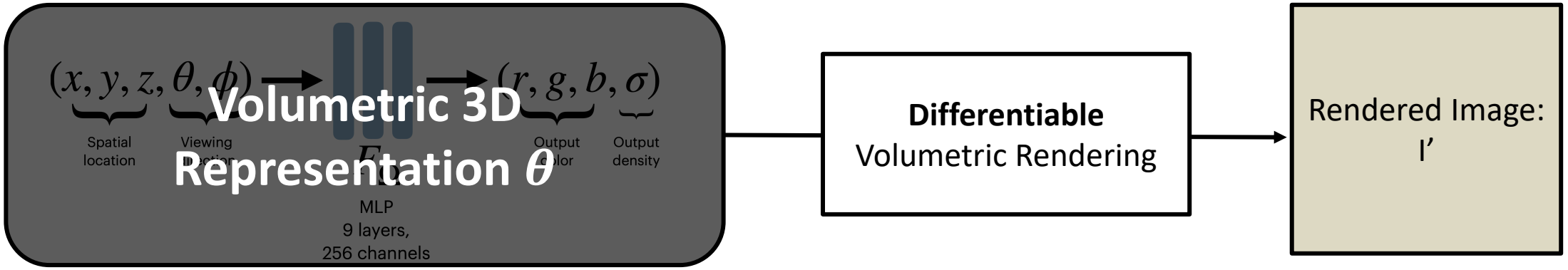
# How do We Store the Values of $\mathbf{c},\sigma$ at Each Point in Space?




Include the ray direction in the input to the MLP → allows for capturing and rendering view-dependent effects (e.g., shiny surfaces)

# Neural Radiance Fields

How an image is made ("Inference")




"Training" Objective (aka Analysis-by-Synthesis):

$$\min_{\theta} \| \text{Rendered Image: I'} - \text{Observed Image: I} \|_2$$

# Summary of the NeRF Training Process

## Step 1: Marching the Camera Rays Through the Scene

**Input:** A set of camera poses $\{x_c, y_c, z_c, \gamma_c, \theta_c\}_n$
**Output:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$

## Step 2: Collecting Query Points

**Input:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$
**Output:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$

## Step 3: Projecting Query Points to High-Dimensional Space (Positional Encoding)

**Input:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$
**Output:** A set of query points embedded into (d)-dimensional space $\{x_1, x_2, ..., x_d\}_{n \times m \times H \times W}$

## Step 4: Neural Network Inference and Volume Rendering

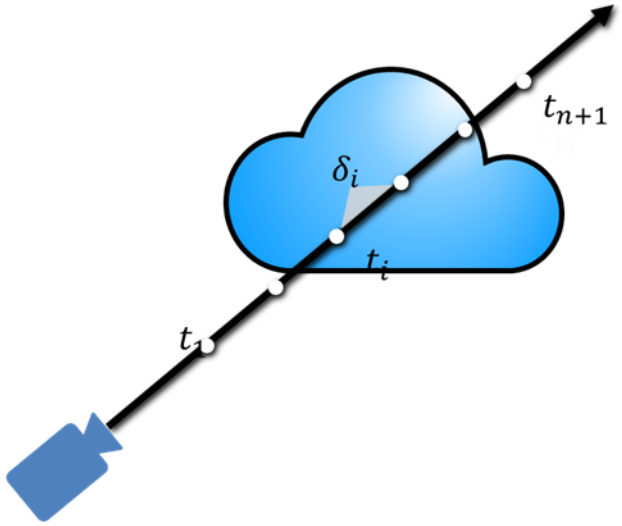
**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$
**Output:** Volume density for every query point $\{\sigma\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$ and $\{v_d\}_{n \times m \times H \times W}$
**Output:** RGB colour for every query point $\{\mathrm{RGB}\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) + their volume profile + RGB value $\{x_1, x_2, \ldots, x_d, \mathrm{RGB}, \sigma\}_{n \times m \times H \times W}$
**Output:** A set of rendered images (one per pose) $\{H, W\}_n$

**Input:** A set of rendered images (one per pose) $\{H, W\}_n$ and a set of ground truth images (one per pose) $\{H, W\}_n^{\mathrm{gt}}$
**Output:** L2 loss between the inputs, a single scalar $\{\ell\}_n$

# Compute the Loss



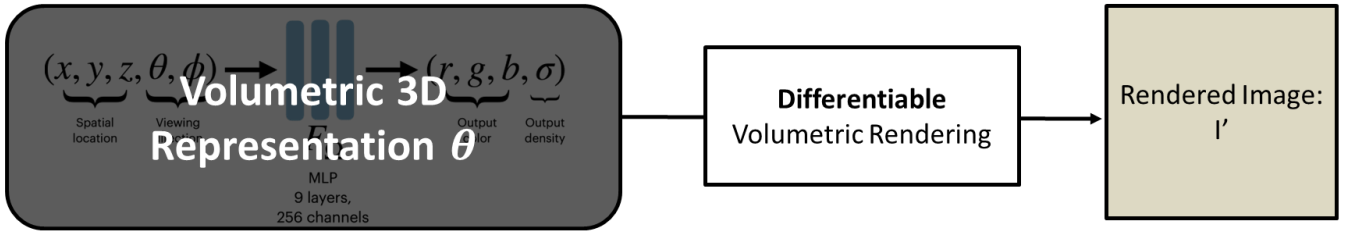

Hence, the discretized emmitted radiance is

$$\begin{aligned}\hat{\mathbf{C}}(\boldsymbol{r}) = \hat{\mathbf{C}}(z; \boldsymbol{o}, \boldsymbol{d}) &= \mathbb{E}(\mathbf{R}) \\ &= \int_{\texttt{near}}^{\texttt{far}} \mathbf{R}\, p_{\mathbf{R}}\, dz \\ &\approx \sum_{i=1}^{N} \boldsymbol{c}_i T_i \left(1 - e^{-\sigma_i \delta_i}\right)\end{aligned}$$

where $\boldsymbol{c}_i := \boldsymbol{c}(z_i; \boldsymbol{o}, \boldsymbol{d})$ is the output RGB upon MLP query at $\{z_i \mid \boldsymbol{o}, \boldsymbol{d}\}$.

Note that if we denote $\alpha_i := 1 - e^{-\sigma_i \delta_i}$, then $\hat{\mathbf{C}}(\boldsymbol{r}) = \sum_{i=1}^{N} \alpha_i T_i \boldsymbol{c}_i$ resembles classical *alpha compositing*.

How an image is made ("Inference")




"Training" Objective (aka Analysis-by-Synthesis):

$$\min_{\theta} \left\| \boxed{\text{Rendered Image: I'}} - \boxed{\text{Observed Image: I}} \right\|_2$$

# Compute the Gradient

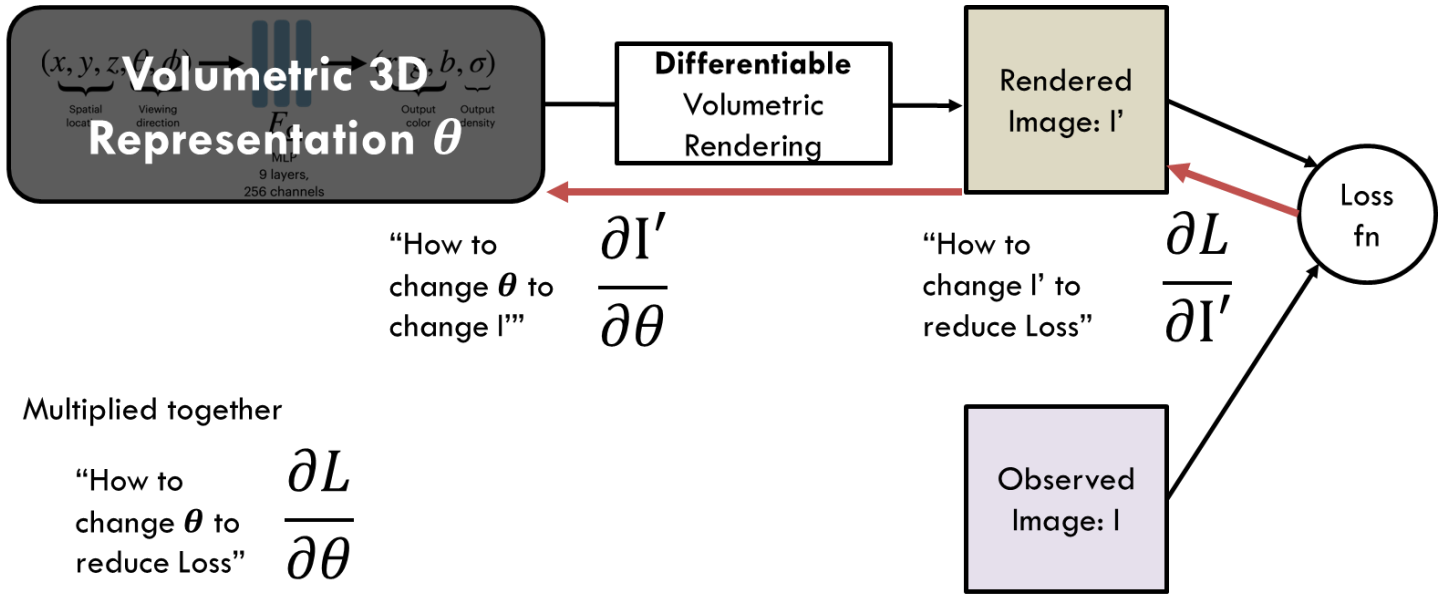
Given the above renderer, a coarse training pipeline is

$$(\boldsymbol{x}, \boldsymbol{d}) \xrightarrow{\text{MLP}} (\boldsymbol{c}, \sigma) \xrightarrow{\text{discrete rendering}} \left.\begin{matrix} \text{ground truth } \mathbf{C} \\ \text{prediction } \hat{\mathbf{C}} \end{matrix}\right\} \xrightarrow{\text{MSE}} \mathcal{L} = \|\hat{\mathbf{C}} - \mathbf{C}\|_2^2$$

If the discrete renderer is differentiable, then we can train the *end-to-end* model through gradient descent. No suprise, given a (sorted) sequence of random samples $\boldsymbol{t} = \{t_1, t_2, \ldots, t_N\}$, the derivatives are

$$\begin{aligned}
\left.\frac{d\hat{\mathbf{C}}}{d\boldsymbol{c}_i}\right|_{\boldsymbol{t}} &= T_i \left(1 - e^{-\sigma_i \delta_i}\right) \\
\left.\frac{d\hat{\mathbf{C}}}{d\sigma_i}\right|_{\boldsymbol{t}} &= \boldsymbol{c}_i \left(\frac{dT_i}{d\sigma_i}\left(1 - e^{-\sigma_i\delta_i}\right) + T_i \frac{d}{d\sigma_i}\left(1 - e^{-\sigma_i\delta_i}\right)\right) \\
&= \mathbf{c}_i \left(\left(1 - e^{-\sigma_i\delta_i}\right) \exp\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right) \underbrace{\frac{d}{d\sigma_i}\left(-\sum_{j=1}^{i-1}\sigma_j\delta_j\right)}_{0} + T_i\left(-e^{-\sigma_i\delta_i}\right)\frac{d(-\sigma_i\delta_i)}{d\sigma_i}\right) \\
&= \delta_i T_i \boldsymbol{c}_i e^{-\sigma_i\delta_i}
\end{aligned}$$

Once the renderer is differentiable, weights and biases in an MLP can be updated via the chain rule.



# Summary of the NeRF Training Process

## Step 1: Marching the Camera Rays Through the Scene

**Input:** A set of camera poses $\{x_c, y_c, z_c, \gamma_c, \theta_c\}_n$
**Output:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$

## Step 2: Collecting Query Points

**Input:** A bundle of rays for every pose $\{v_o, v_d\}_{H \times W \times n}$
**Output:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$

## Step 3: Projecting Query Points to High-Dimensional Space (Positional Encoding)

**Input:** A set of 3D query points $\{x_p, y_p, z_p\}_{n \times m \times H \times W}$
**Output:** A set of query points embedded into (d)-dimensional space $\{x_1, x_2, ..., x_d\}_{n \times m \times H \times W}$

## Step 4: Neural Network Inference and Volume Rendering

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$
**Output:** Volume density for every query point $\{\sigma\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) $\{x_1, x_2, \ldots, x_d\}_{n \times m \times H \times W}$ and $\{v_d\}_{n \times m \times H \times W}$
**Output:** RGB colour for every query point $\{\mathrm{RGB}\}_{n \times m \times H \times W}$

**Input:** A set of 3D query points (after positional encoding) + their volume profile + RGB value $\{x_1, x_2, \ldots, x_d, \mathrm{RGB}, \sigma\}_{n \times m \times H \times W}$
**Output:** A set of rendered images (one per pose) $\{H, W\}_n$

**Input:** A set of rendered images (one per pose) $\{H, W\}_n$ and a set of ground truth images (one per pose) $\{H, W\}_n^{\mathrm{gt}}$
**Output:** L2 loss between the inputs, a single scalar $\{\ell\}_n$



Adapted from material from Pratul Srinivasan

Adapted from material from Pratul Srinivasan

Adapted from material from Pratul Srinivasan

# Final Summary

- Represent the scene as volumetric transparent (“fog”) color compositions
- Store the fog color and density at each point as an MLP mapping 3D position (x, y, z) to color c and density σ
- Render image by shooting a ray through the fog for each pixel
- Optimize MLP parameters by rendering to a set of known viewpoints and comparing to ground truth images
- Produces geometry/reflectance estimates that are good for interpolating views and robust to non-Lambertian surfaces
- Many follow-on works for efficient learning/storing/rendering, extending applicable settings, and manipulations

# Reference


1. **Deep Dive into NeRF (Neural Radiance Fields)** - https://dtransposed.github.io/blog/2022/08/06/NeRF/
2. **CS180: Intro to Computer Vision and Computational Photography Project 5** - https://inst.eecs.berkeley.edu/~cs180/fa23/hw/proj5/
3. **CS180/280A Intro to Computer Vision and Computational Photography** - https://inst.eecs.berkeley.edu/~cs180/fa23/
4. **CSC 2547** - https://www.pair.toronto.edu/csc2547-w21/
5. **An Overview of Differentiable Rendering** - https://blog.qarnot.com/article/an-overview-of-differentiable-rendering
6. **NeRF: A Volume Rendering Perspective** - https://yconquesty.github.io/blog/ml/nerf/nerf_rendering.html#overview
7. **Volume Rendering for Developers: Summary and Equations** - https://www.scratchapixel.com/lessons/3d-basic-rendering/volume-rendering-for-developers/volume-rendering-summary-equations.html (note: this link is not currently accessible)
8. **Volume Rendering** - https://en.wikipedia.org/wiki/Volume_rendering