Friedrich-Alexander-Universität
Technische Fakultät

Neural Radiance Fields - NeRFs

Summer Term 2023

Bernhard Egger

Slide Credit: Marc Stamminger, NeuGIeR -Neural Graphics and Inverse Rendering

Neural Radiance Fields

- Mildenhall et al.: „NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis", ECCV 2020
  - > 3.309 citations (as of June 2022)
  - thousands of successor papers

NeRF

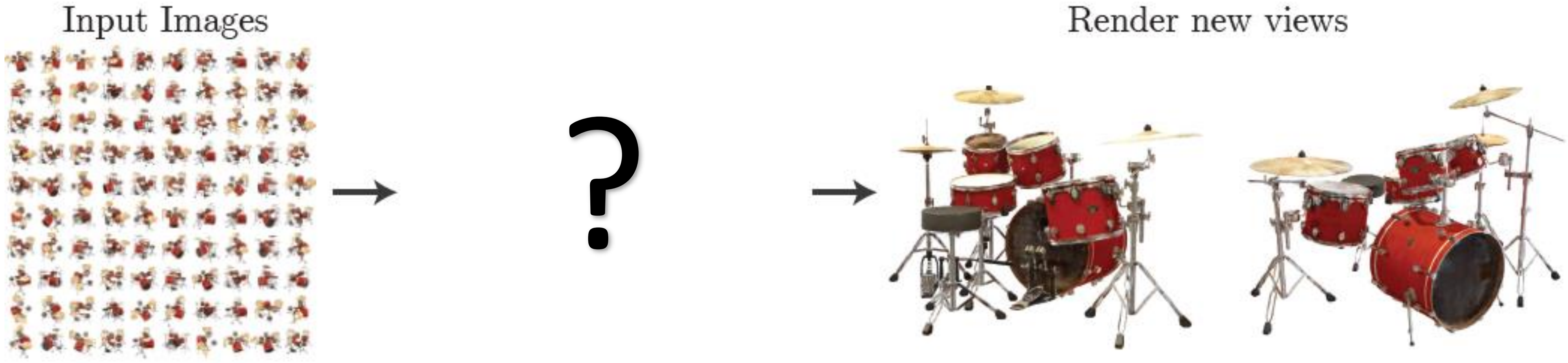
Representing Scenes as Neural Radiance Fields for View Synthesis

ECCV 2020 Oral - Best Paper Honorable Mention

Ben Mildenhall* UC Berkeley
Pratul P. Srinivasan* UC Berkeley
Matthew Tancik* UC Berkeley
Jonathan T. Barron Google Research
Ravi Ramamoorthi UC San Diego
Ren Ng UC Berkeley

*Denotes Equal Contribution

Novel View Synthesis



Today

- We will motivate and learn about
  - Novel View Synthesis
  - Volume Rendering
- You already know quite a bit about artificial neural networks
- And will combine all this to introduce **Neural Radiance Fields**

Novel View Synthesis / Image-based Rendering

Remember: Computer Graphics

- Given a 3D model of a scene (triangle mesh)
- Project to 2D screen, rasterize, apply lighting and shading to determine pixel colors
- Textures can be applied to get visually richer objects
- Burden to generate 3D models and textures → costly, tedious, annoying
- In particular annoying, if we have to re-model **a real-world object or scene**

3D-Reconstruction

- Classical Computer Vision task:
  Generate a 3D geometric model plus texture from a set of input images
  → lectures „Computer Vision“ or
  „Computational Photography & Capture“
  (both in summer term)

…

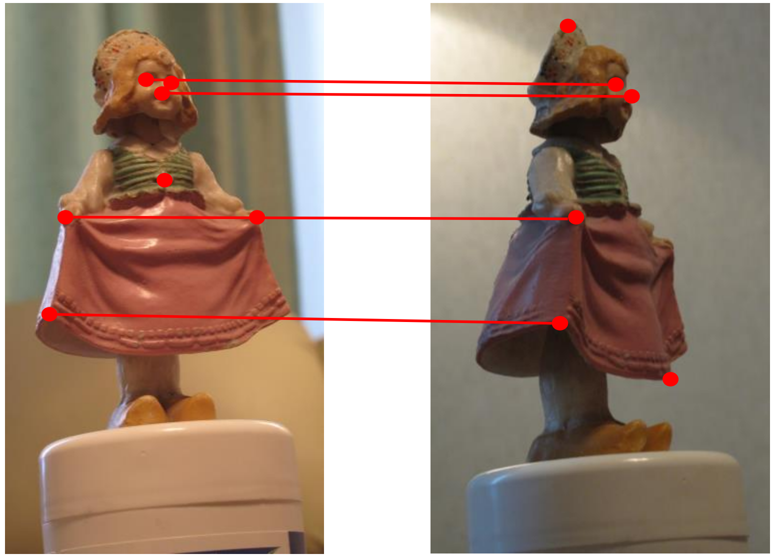
doll images: agisoft metashape, www.agisoft.com

3D-Reconstruction

- Steps:
  - detect **feature points** in images

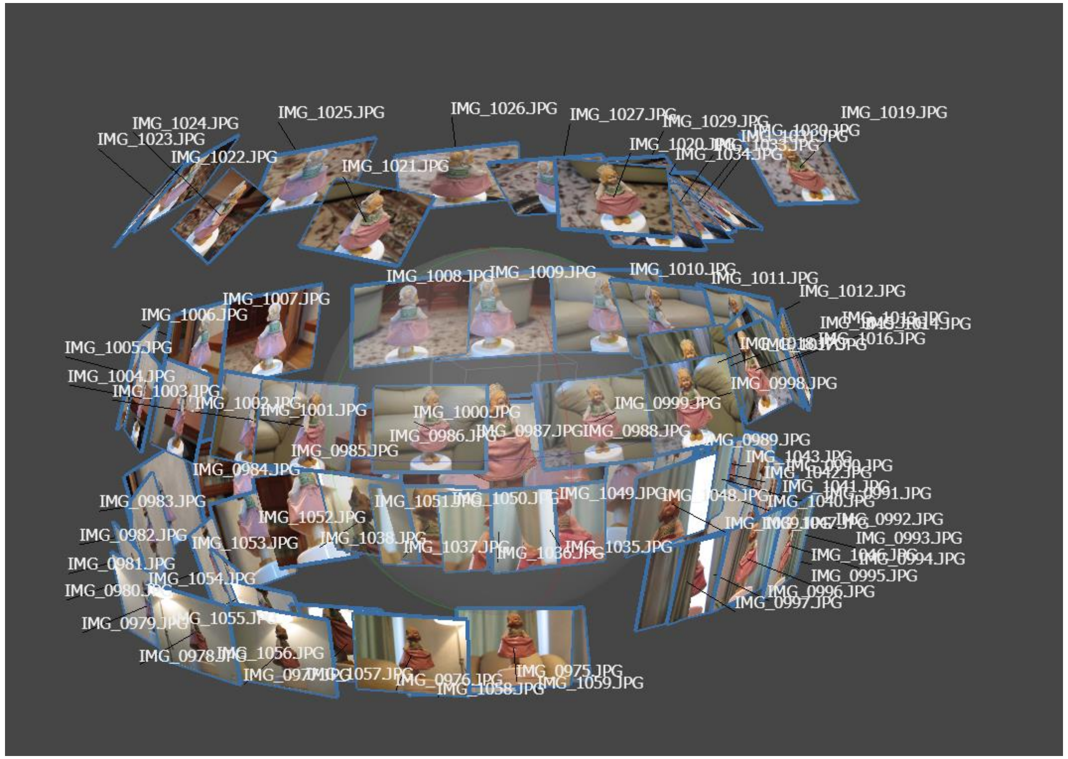
3D-Reconstruction

- Steps:
  - detect **feature points** in images
  - **match feature points** over multiple images



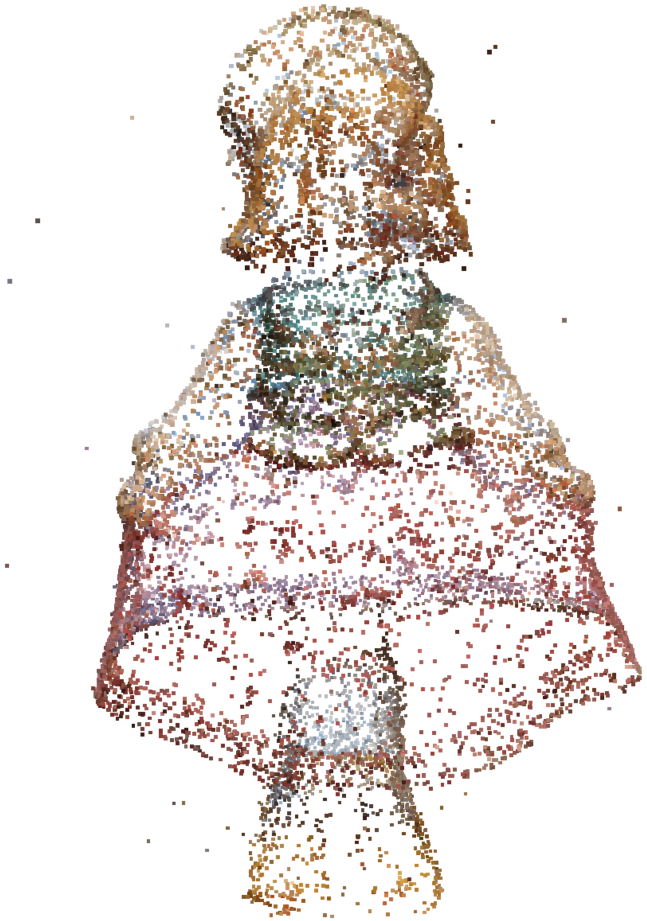
3D-Reconstruction

- Steps:
  - detect **feature points** in images
  - **match feature points** over multiple images
  - determine camera poses of input images



3D-Reconstruction

- Steps:
  - detect **feature points** in images
  - **match feature points** over multiple images
  - determine camera poses of input images
  - project matching feature points to 3D → **sparse 3D point cloud**

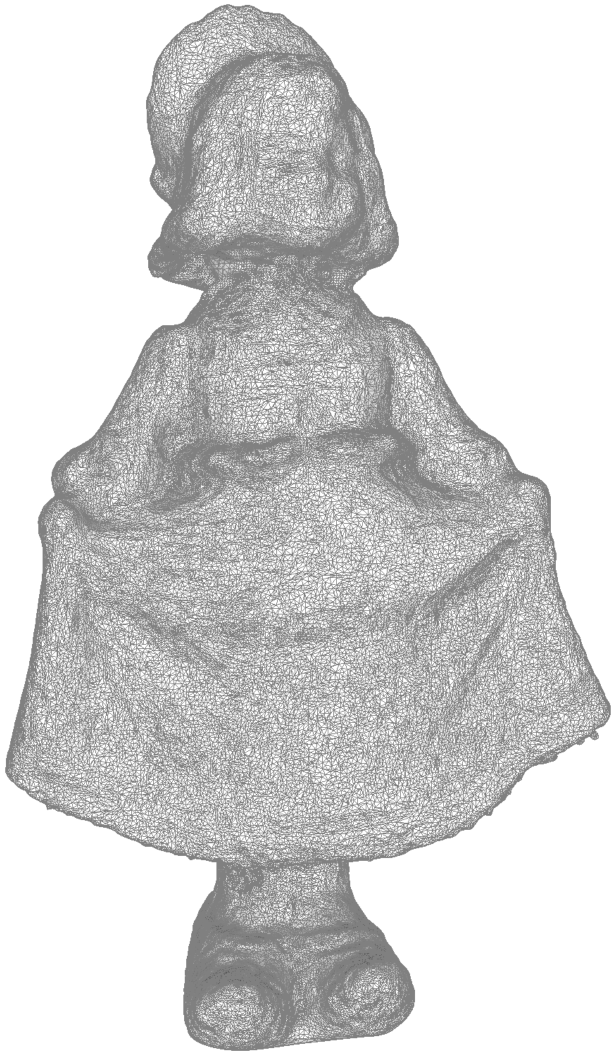


3D-Reconstruction

- Steps:
  - detect **feature points** in images
  - **match feature points** over multiple images
  - determine camera poses of input images
  - project matching feature points to 3D → **sparse 3D point cloud**
  - densify point cloud → **dense point cloud**

3D-Reconstruction

- Steps:
  - detect **feature points** in images
  - **match feature points** over multiple images
  - determine camera poses of input images
  - project matching feature points to 3D → **sparse 3D point cloud**
  - densify point cloud → **dense point cloud**
  - convert point cloud to triangle mesh → **3D triangle mesh**



3D-Reconstruction

- Steps:
  - detect **feature points** in images
  - **match feature points** over multiple images
  - determine camera poses of input images
  - project matching feature points to 3D → **sparse 3D point cloud**
  - densify point cloud → **dense point cloud**
  - convert point cloud to triangle mesh → **3D triangle mesh**
  - compute texture for triangle mesh

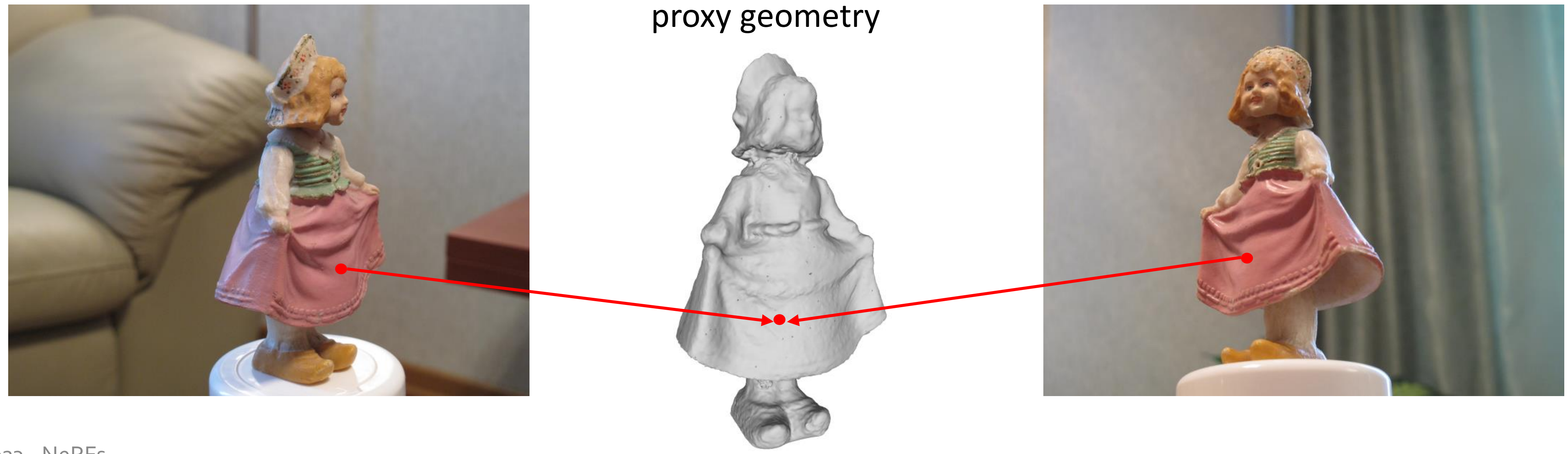
→ **Quality moderate, no specular effects**

Image-based Rendering

- Most often used variant:
  - Given a set of photographs
  - Determine camera poses
  - Reconstruct (rough) 3D model of the object = **proxy geometry**
  - At render time:
    - render **proxy geometry**
    - shade pixels using colors from photographs

Image-based Rendering

- for each pixel, look up color in input images, which contain this point
- blend colors
- → improved quality, view-dependent effects
- → artifacts, if proxy geometry / poses not perfect



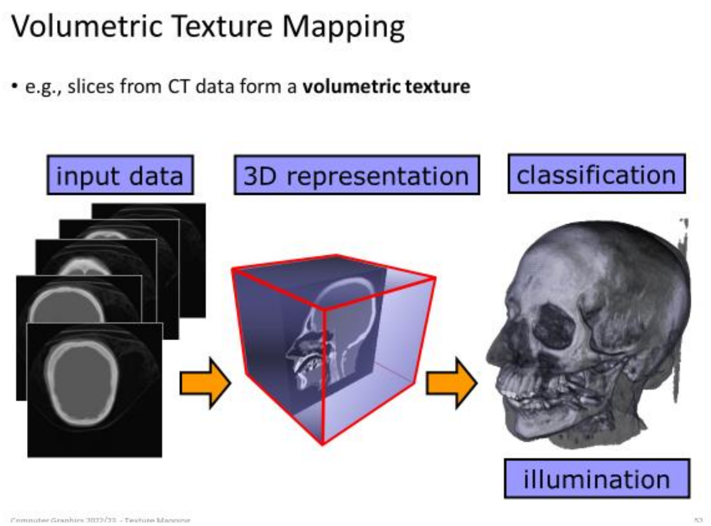
Novel View Synthesis

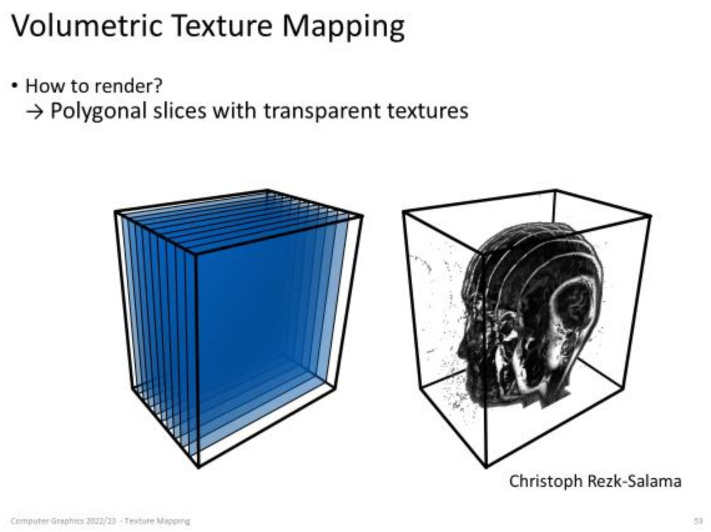
- Image-based rendering is an algorithm for **„Novel View Synthesis“**
- **General problem:**
  - **Given a set of input images of an object**
  - **Generate novel views of the object**

Volume Rendering

And now for something completely different…

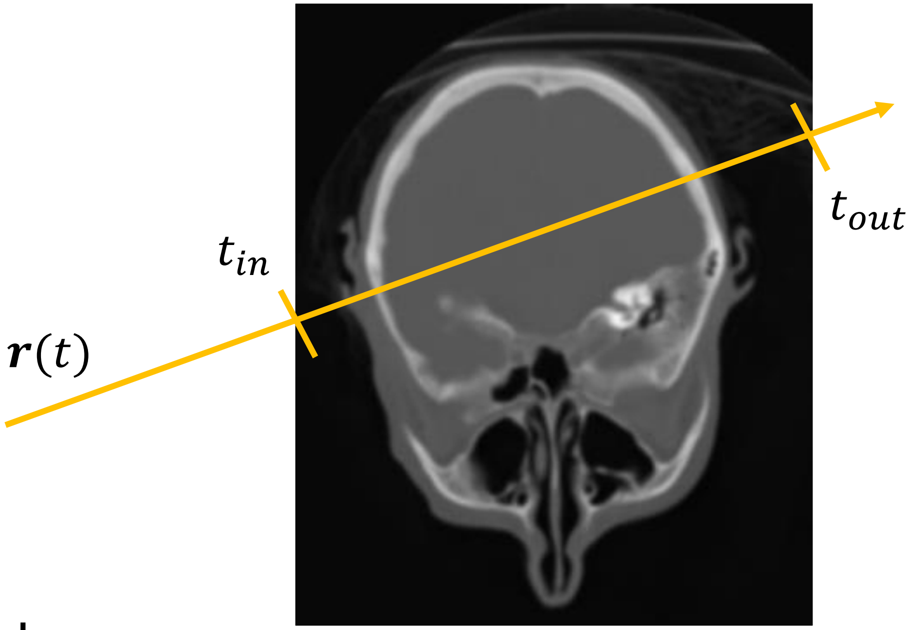
- Remember from CG:
  Volumetric Texture Mapping
- Given a 3D volumetric texture of density values, generate an image of this volume
- Can be rendered by rendering semi-transparent slices of the volume using alpha-blending
- Today, more often **ray casting** is used





Volume Rendering

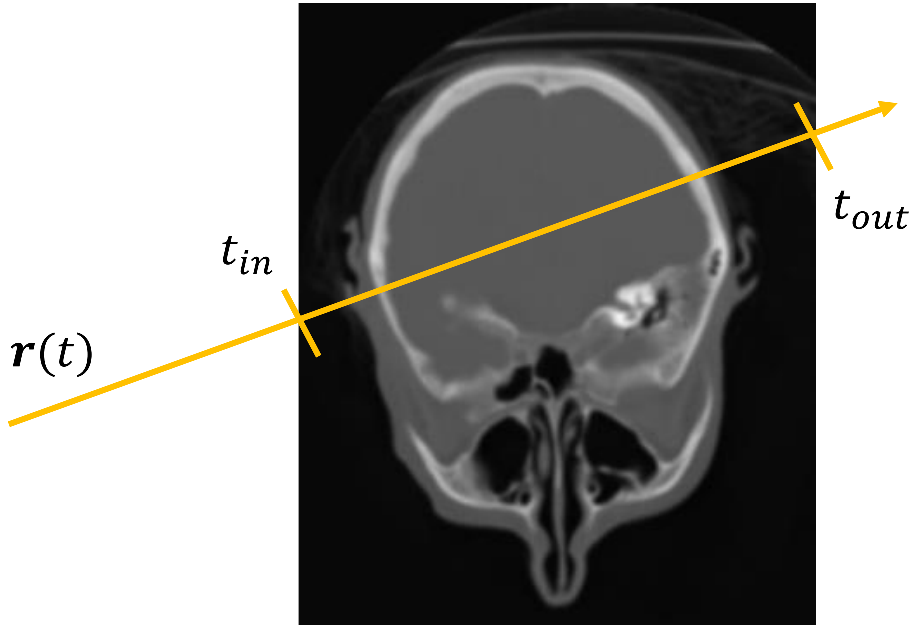
- More formally using **ray casting**
- Given a volumetric density field $\sigma(\boldsymbol{x})$, where ρ is the optical density at position $\boldsymbol{x} \in \boldsymbol{R}^3$
- For each image pixel, we cast a ray $\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}$ through the volume and determine its color
- To determine color, we need a volumetric lighting model, e.g.:



$$\boldsymbol{C}(\boldsymbol{r}) = \int_{t \in [t_{in}, t_{out}]} T(t)\, \sigma(\boldsymbol{r}(t))\, \boldsymbol{c}(\boldsymbol{r}(t), \boldsymbol{d})\, dt$$

???

$$\boxed{\boldsymbol{C}(\boldsymbol{r})} = \int_{t\in[t_{in},t_{out}]} T(t)\,\sigma(\boldsymbol{r}(t))\,\boldsymbol{c}(\boldsymbol{r}(t),\boldsymbol{d})\,dt$$

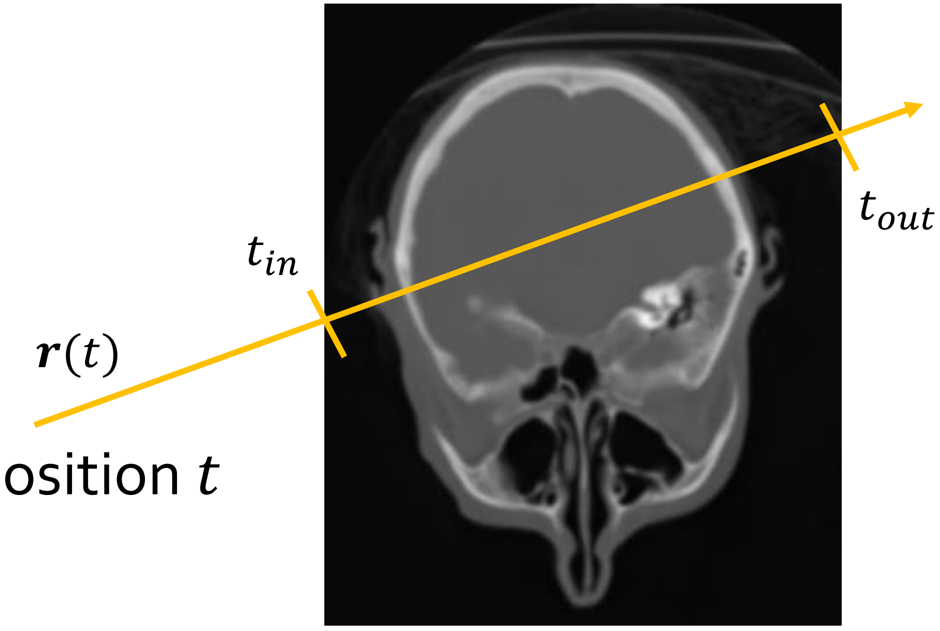


- do simple ray casting
- use formula above to compute ray colors:

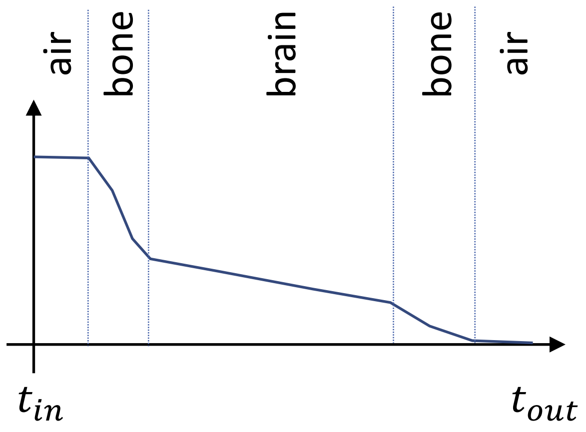
```
for y = 0; y < height; y++
    for x = 0; x < width; x++
        r = generate eye ray through (x,y)
        C = approximate integral above for r
        setPixel(x,y,C)
```

???

$$\boldsymbol{C}(\boldsymbol{r}) = \int_{t\in[t_{in},t_{out}]} T(t)\, \sigma(\boldsymbol{r}(t))\, \boldsymbol{c}(\boldsymbol{r}(t),\boldsymbol{d})\, dt$$

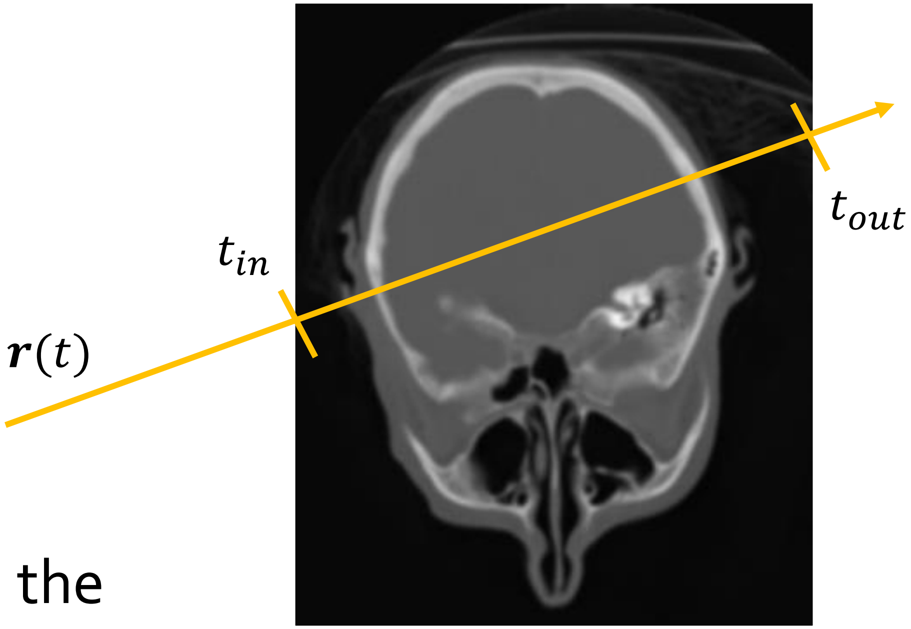


- $T(t)$: accumulated transmittance along the ray up to position t
  - If a photon starts at $r(t)$ travelling towards $r(0)$, how likely is it that it gets through and is not absorbed?
  - (Can be computed using Beer's law)

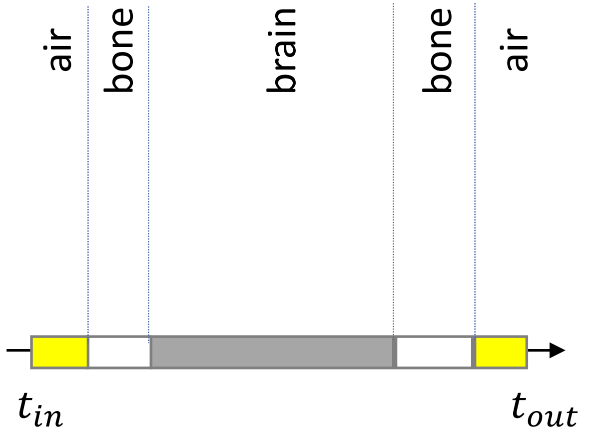


???

$$\boldsymbol{C}(\boldsymbol{r}) = \int_{t \in [t_{in}, t_{out}]} T(t)\, \sigma(\boldsymbol{r}(t))\, \boldsymbol{c}(\boldsymbol{r}(t), \boldsymbol{d})\, dt$$

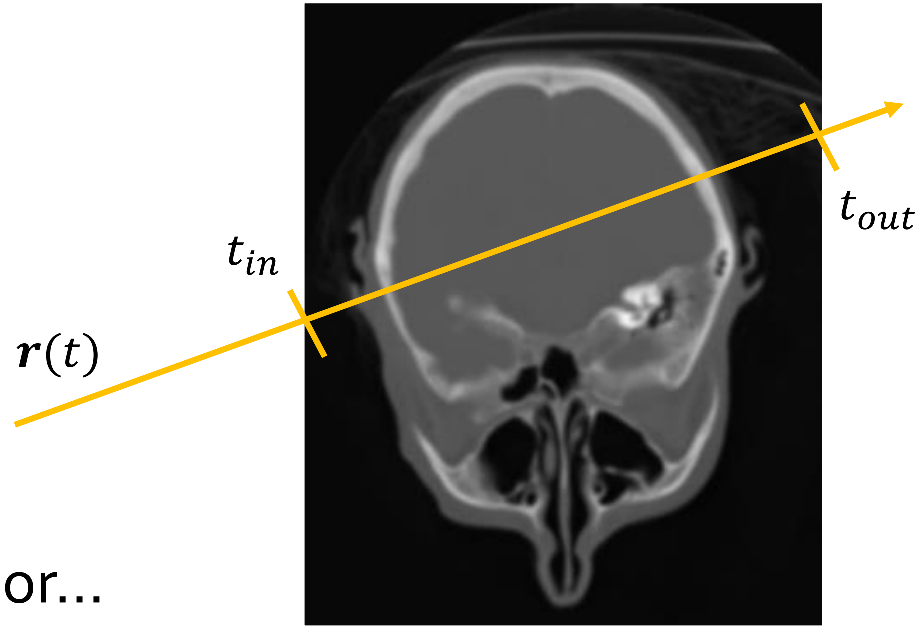


- as "color" of a point in the volume we use $\boldsymbol{c}(\boldsymbol{r}(t), \boldsymbol{d})$
- c varies in the volume, most often it is determined from the input density $\sigma(\boldsymbol{x})$ by a **transfer function**
  - e.g. in medical visualization:
    - large $\sigma \rightarrow$ bone $\rightarrow$ color is white
    - medium $\sigma \rightarrow$ blood $\rightarrow$ red
- in this lighting model, color varies with the view direction $\boldsymbol{d}$ $\rightarrow$ uncommon in medical visualization

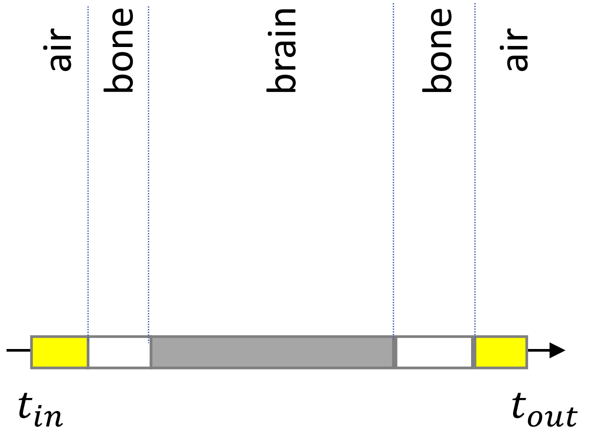


???

$$\boldsymbol{C}(\boldsymbol{r}) = \int_{t \in [t_{in}, t_{out}]} T(t)\, \sigma(\boldsymbol{r}(t))\, \boldsymbol{c}(\boldsymbol{r}(t), \boldsymbol{d})\, dt$$



- but what is the color of air?
- air has zero density, so we cannot see it, so it has no color…
- we thus multiply the color at $\boldsymbol{r}(\boldsymbol{t})$ by its density $\sigma(\boldsymbol{r}(\boldsymbol{t}))$
  → the color of air is irrelevant
- separates density and color, allows for better handling



How to compute

- Numerical integration: we step along the ray using step width δ:

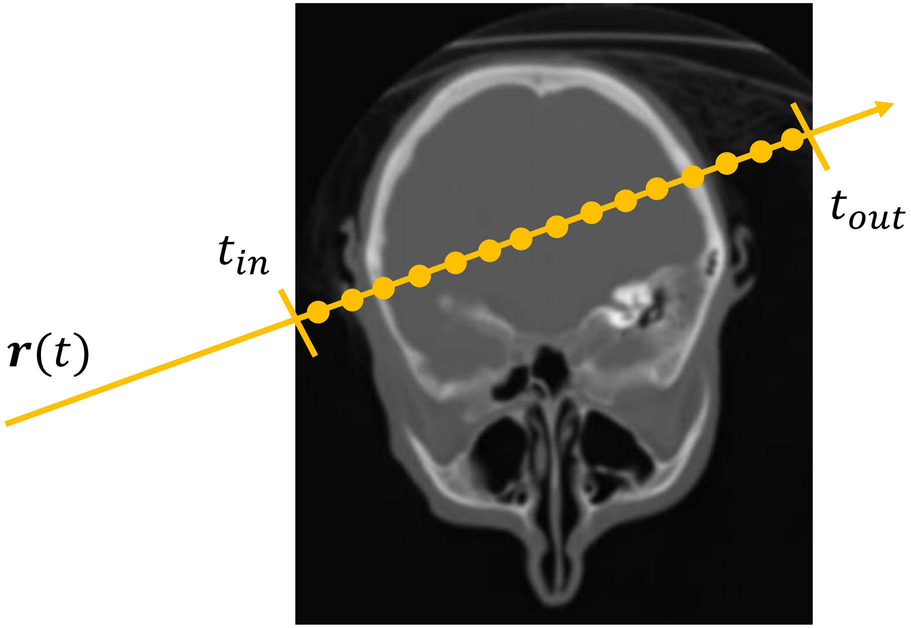
$$t_i = t_{in} + i\delta$$

- We accumulate transmittance while stepping forward:

$$T_0 = 1$$
$$T_{i+1} = T_i \exp\left(-\sigma\left(\boldsymbol{r}(t_i)\right)\delta\right)$$

- We accumulate the final color:

$$C_0 = (0,0,0)$$
$$C_{i+1} = C_i + T_i\,\sigma\left(\boldsymbol{r}(t_i)\right)\boldsymbol{c}(\boldsymbol{r}(t_i), -d)\,\delta$$



CT reconstruction

- With a lighting model like the above, we can simulate x-ray images
- x-rays are like light, but with higher frequency, so they penetrate bodies
- CT-reconstruction (CT = computed tomography)
  - given a number of such x-ray images from various directions, reconstruct a volumetric density function $\sigma(\boldsymbol{x})$, such that volumetric renderings are equivalent to the x-ray images
    $\rightarrow$ in this context, color $\boldsymbol{c}(\boldsymbol{r}(t), \boldsymbol{d})$ is assumed to be constant white
  - more formally: find a volumetric function $\sigma(\boldsymbol{x})$ that minimizes the difference between the observed images I^i and the volume rendered images $\tilde{I}^i$
  - Typical CT-reconstruction: represent volume as a voxel grid, e.g. with 256^3 voxels, each storing a density
  - highly (or infinite)-dimensional optimization problem, but solvable

Radiance Fields

- We can apply all this for novel view synthesis → **Radiance Fields**
- A **radiance field** is
  - a function $\sigma(\boldsymbol{x})$ representing density at $\boldsymbol{x}$ within the bounding box of an object
  - a function $\boldsymbol{c}(\boldsymbol{r}(t), \boldsymbol{d})$ representing the color at $\boldsymbol{x}$ when viewed from direction $\boldsymbol{d}$
- Idea:
  - Given a set of input images with known poses
  - We then determine a radiance field that under volume rendering generates the same images
    $\longrightarrow$ complex optimization like for CT reconstruction
  - We can then render this radiance field from arbitrary views
    $\longrightarrow$ novel view synthesis

Radiance Fields

- How can we represent a radiance field?
- Option 1: Voxelization (like in CT reconstruction)
  - build a voxel grid, e.g. 256^3 voxels
  - in each cell
    - store a density value σ
    - store a representation of the view-dependent color $\boldsymbol{c}(\boldsymbol{d})$
  - high memory costs
  - unclear how to store view-dependent color function
  - results vary with resolution

Radiance Fields

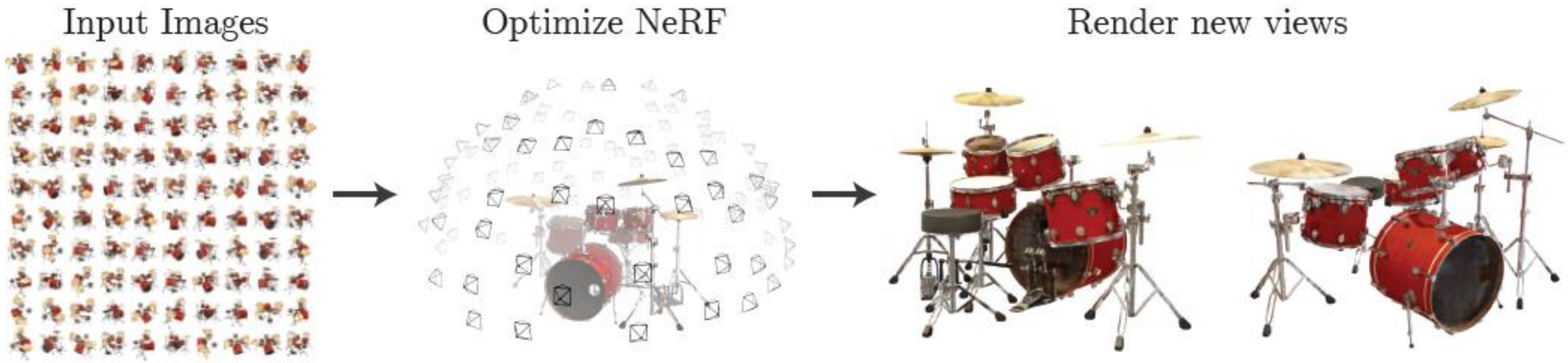
- How can we represent a radiance field?
- Option 2: **Neural** radiance fields
  - store and optimize radiance field as a **neural** function
    → more precisely, as a multi-layer perceptron (**MLP**)

NeRFs
Neural Radiance Fields – NeRFs

Novel View Synthesis by combining
Volume Rendering and
Deep Neural Networks

Neural Radiance Fields

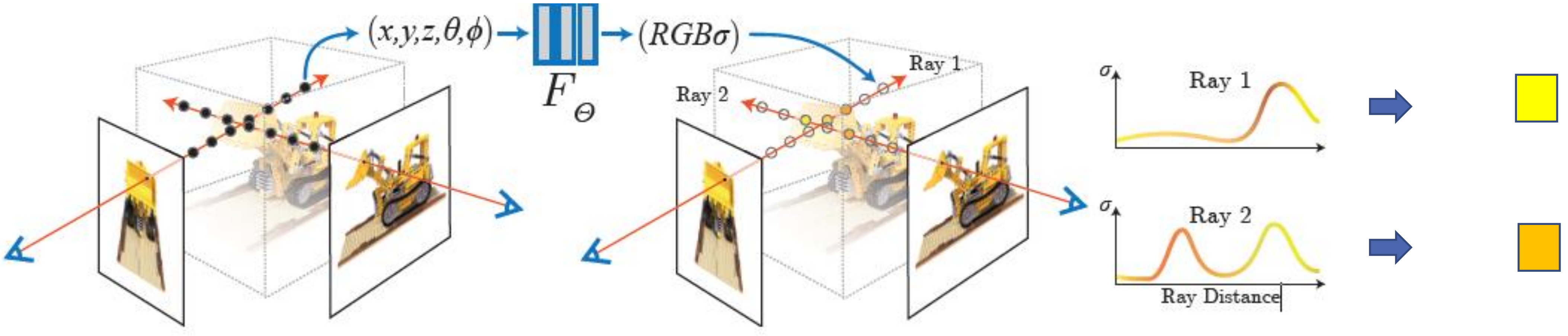
- Represent radiance field as an MLP
  - with five inputs: $(x, y, z) \in \boldsymbol{R}^3$, direction described in polar coordinates $(\theta, \phi) \in \boldsymbol{R}^2$
  - and four outputs: $\boldsymbol{c}(x, y, z, \theta, \phi) \in \boldsymbol{R}^3$ and $\sigma(x, y, z) \in \boldsymbol{R}^1$
- Optimize neural radiance field (NeRFs) so it reproduces input images
  $\longrightarrow$ costly optimization, slow
- Render novel views from NeRF by volume rendering
  $\longrightarrow$ rather fast



NeRF Rendering

- For each ray, sample the NeRF along the ray (with some predefined stepsize δ) $\longrightarrow$ densities σ_i, colors $\boldsymbol{c}_i$

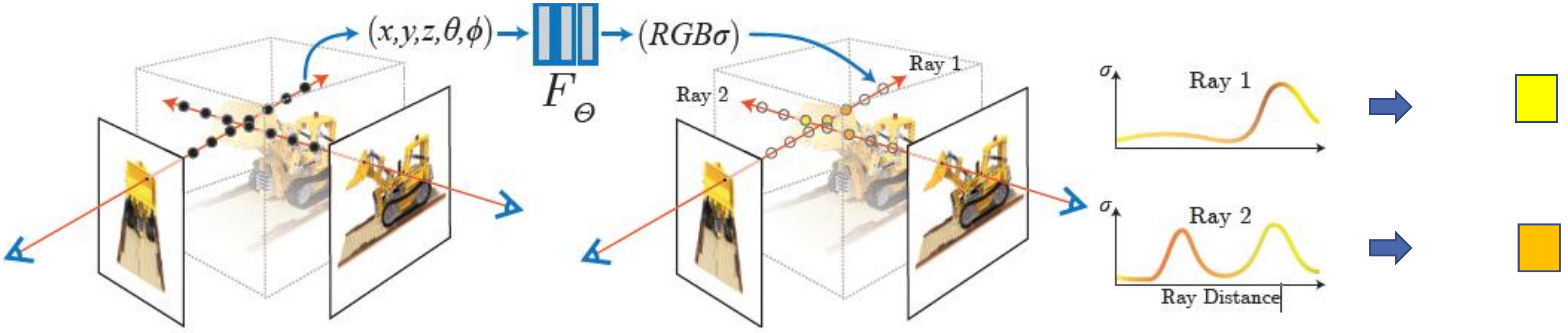
generate samples along rays | evaluate NeRF at sample positions | blend colors



NeRF Rendering

- Blend the sampled values like in Section "Volume Rendering":
  $\boldsymbol{c}(\boldsymbol{r}) = \sum_i T_i \left(1 - \exp(-\boldsymbol{\sigma_i}\delta)\right) \boldsymbol{c}_i$ with $T_i = \exp\left(-\delta \sum_{j=1\ldots i-1} \sigma_i\right)$
- can be derived w.r.t. σ_i and $\boldsymbol{c}_i$

generate samples along rays | evaluate NeRF at sample positions | blend colors



NeRF Rendering – Functional View

- Given a ray $(\boldsymbol{o}, \boldsymbol{d})$
- Generate samples: $S(\boldsymbol{o}, \boldsymbol{d}) = \left(\boldsymbol{o} + (t_{in} + \delta i)\boldsymbol{d}\right)_i = (\boldsymbol{x}_i)_i = \boldsymbol{X}$
- Evaluate NeRF: $F(\boldsymbol{X}) = (F(\boldsymbol{\theta}; x_i))_i = \boldsymbol{F}$ // evaluate NeRF for vector of samples
  $\longrightarrow$ network is a 9-layer / 256 channels per layer, ReLU, fully connected (details later)
- Blend values: $B(F) = \cdots = \boldsymbol{c} \in \boldsymbol{R}^3$ // volume rendering
- Altogether: $\boldsymbol{c}(\boldsymbol{\theta}; \boldsymbol{o}, \boldsymbol{d}) = B\left(F\left(\boldsymbol{\theta};\ S(\boldsymbol{o}, \boldsymbol{d})\right)\right)$

- **Input of the pipeline is a ray $\boldsymbol{x} = (\boldsymbol{o}, \boldsymbol{d})$, output is its color $\boldsymbol{y} = \boldsymbol{c}$**

NeRF Optimization

- More general:
  - Each pixel i of each input image corresponds to one input ray (o_i, d_i), its color c_i is the desired output
- Training data:
  - Each i iterates over all pixels of all images
  - input $x_i = (o_i, d_i)$ of pixel i // for this, we need the camera poses
  - y_i = color of pixel i // the color of the ray in the input image → goal of optimization
- Training process:
  - optimize θ, such that $\sum_i L\left(B\left(F(\boldsymbol{\theta};\ S(x_i))\right), y_i\right)$ is minimized

NeRF Optimization

– Stochastic Gradient Descent Optimization of NeRFs:

```
Initialize θ with small random values
For a bunch of rays (o_i, d_i) with target color c_i:
    compute G = ∂L(B(F(θ; S(o_i,d_i))),c_i) / ∂θ
    update θ ← θ − λG    // λ is learning rate
Repeat upper loop until convergence
```

Neural Radiance Fields

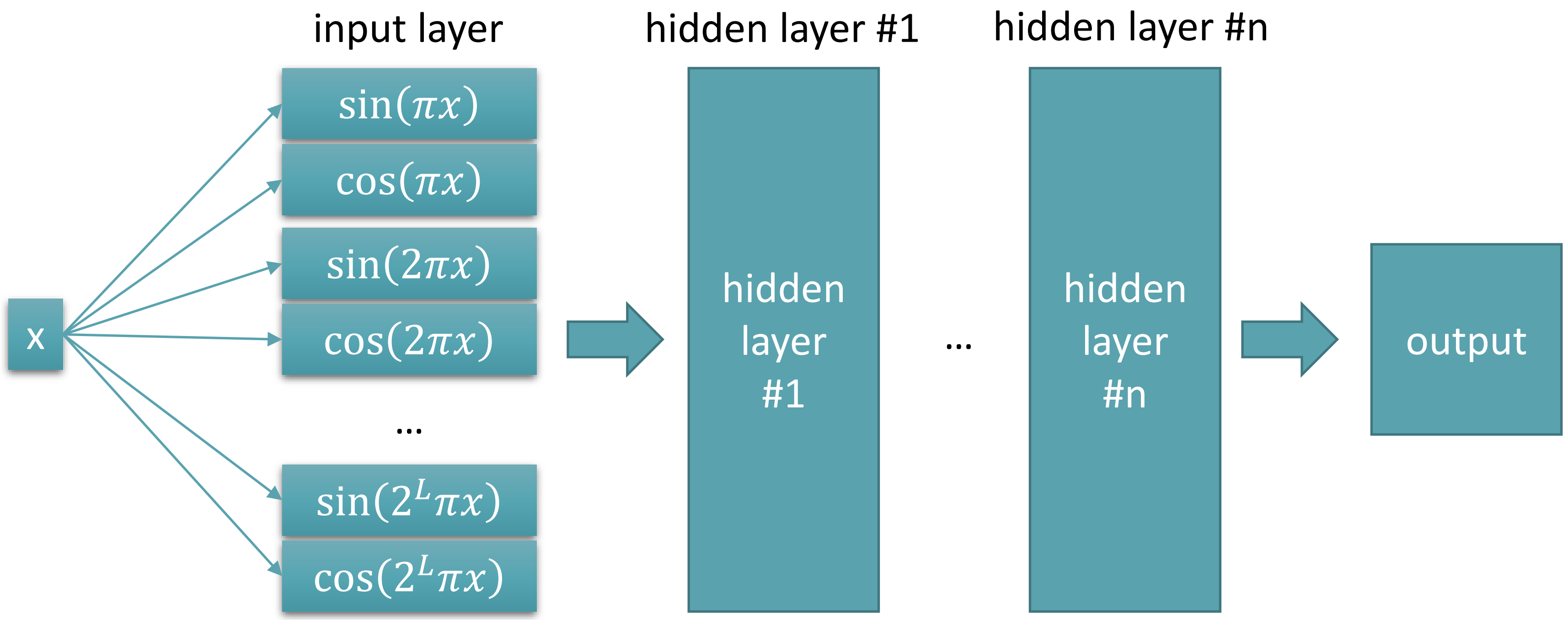
NeRFs – View Dependence

- In general, we can drop the view dependence of the color, which simplifies the NeRF significantly. However, view-dependent effects such as glossy reflections get lost then.

Ground Truth Complete Model No View Dependence

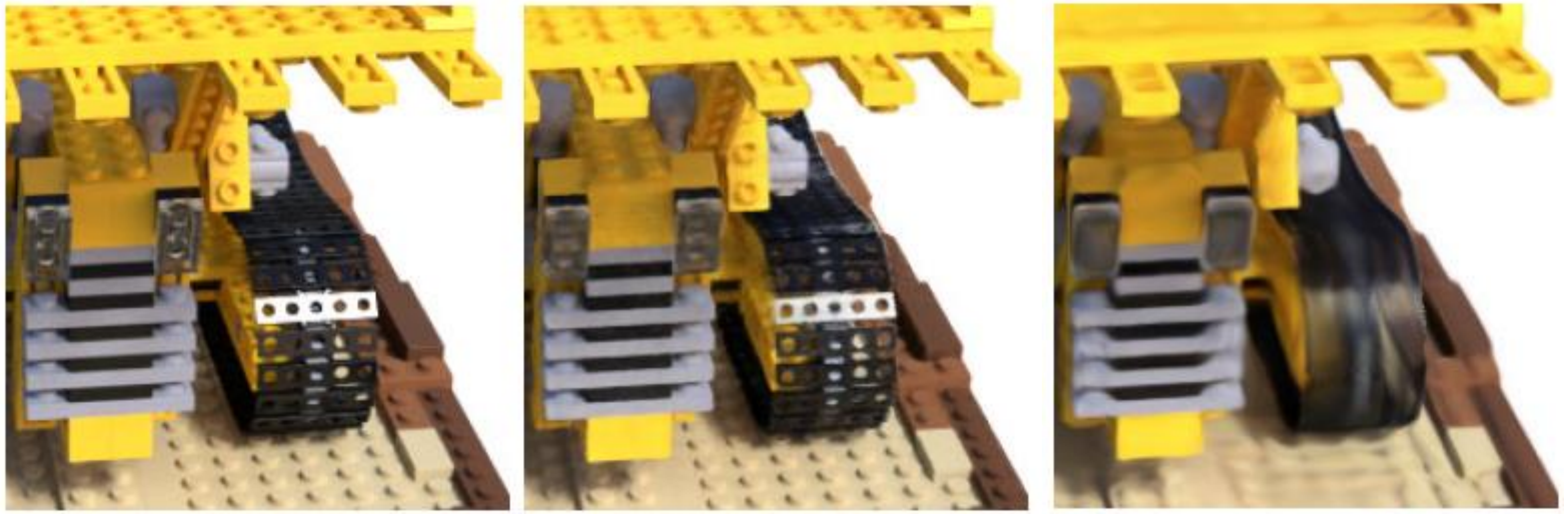
NeRFs – Positional Encoding

- Normal MLPs tend to blur the represented function
- This improves significantly, if we input the coordinates encoded with different frequencies.



NeRFs – Positional Encoding

- This idea is called „Positional Encoding“

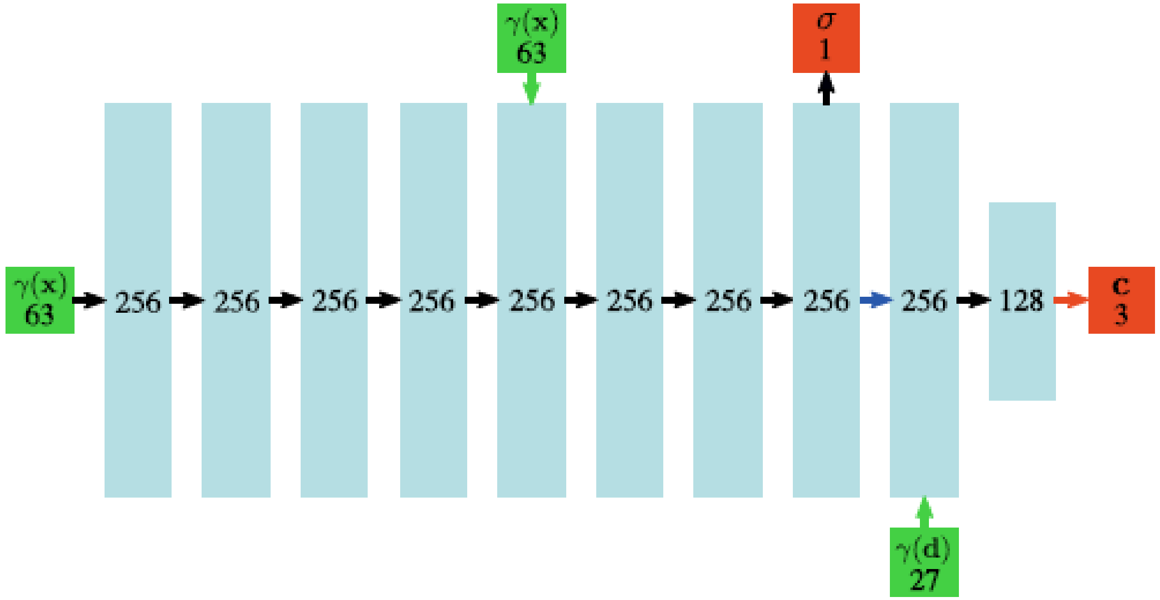


Ground Truth Complete Model No Positional Encoding

- see also later paper: Mildenhall et al.: „Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains“, NeurIPS 2020

NeRFs – Positional Encoding

- Network architecture
  - $\gamma(x)$: positional encoding
  - position is re-injected again into 5th layer
    $\longrightarrow$ *skip* connection
  - direction d is injected very late only
  - 8th layer outputs density
  - last layer uses sigmoid as activation
    and outputs color



NeRFs – Hierarchical Training

- Generally, we consider very many „transparent“ samples that don‘t contribute
  → inefficient
- Better: optimize a coarse and a fine NeRF
- First, sample the coarse NeRF
- Only in regions, where coarse NeRF has higher density,
  sample finer and update fine NeRF

NeRFs – Successors

- There is a huge number of successor papers and tools that improve on certain aspects
  - KiloNeRF
  - Instant Neural Graphics Primitives
  - Plenoxels
  - MobileNeRF
  - NeRFStudio
  - FlowCam
  - Radiance Field Gradient Scaling

Open Challenges

- Realtime training
- Few Input Images / Generalization
- Extract Physical Properties
- Control/Editing