Lecture 17:

Learning Scene Representations

Visual Computing Systems
Stanford CS348K, Spring 2024

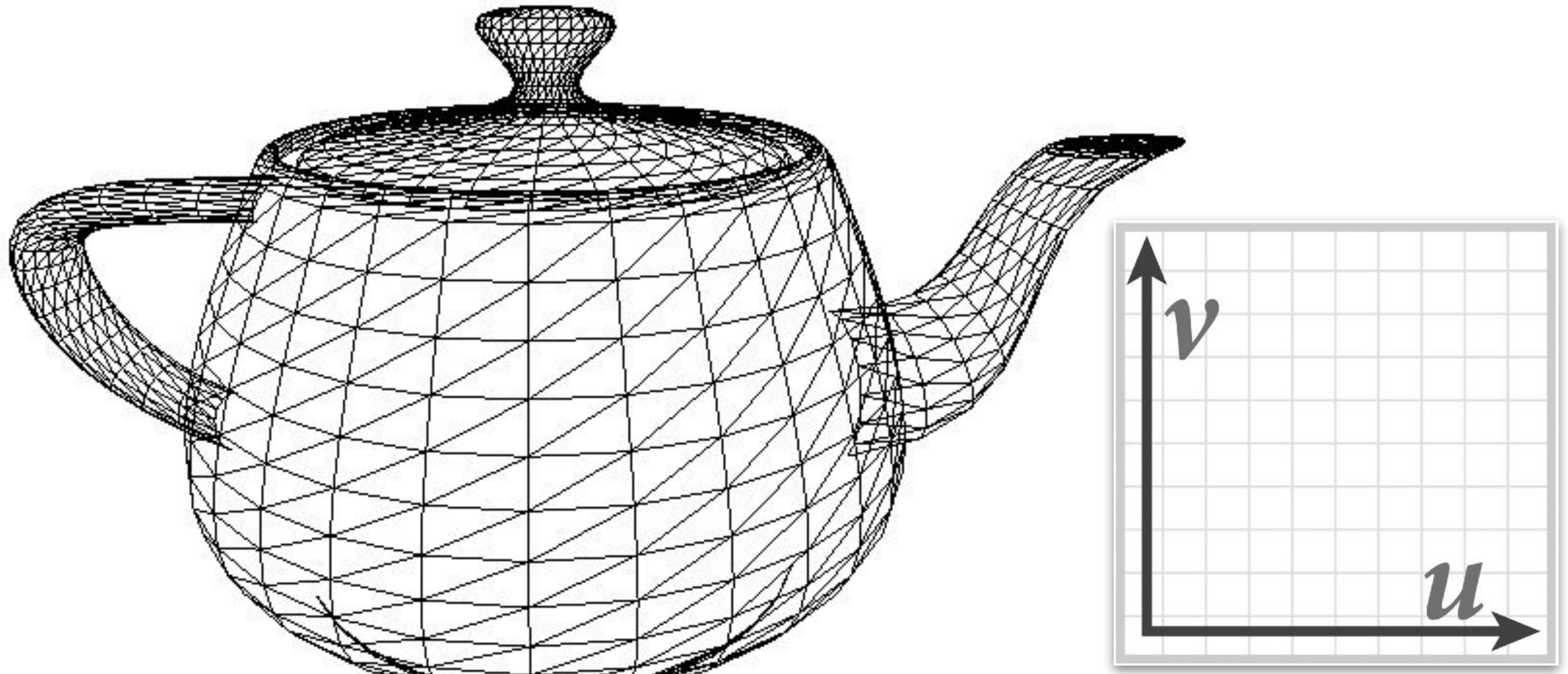
Computer science in a nutshell:
Choose the right representation for the task at hand

Today's task

- Recovering a 3D scene representation from a collection of photos

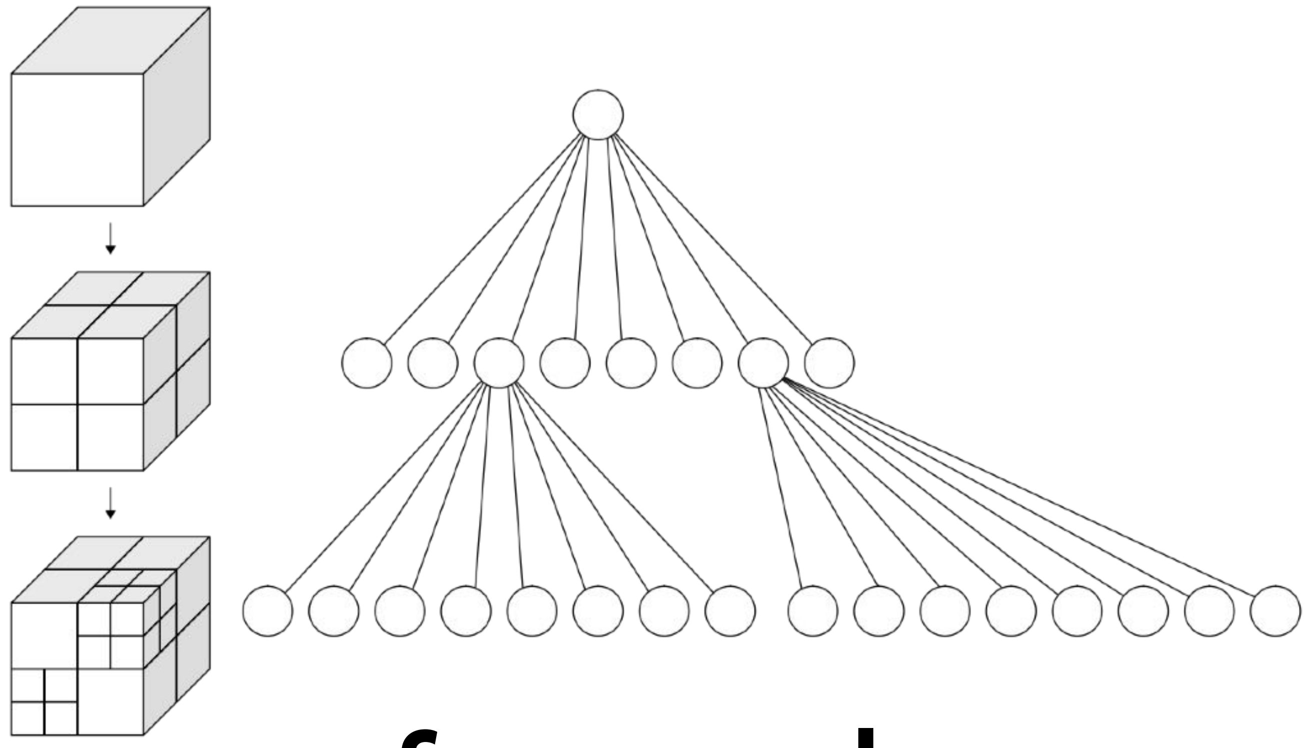
- Why?
  - So we can render them from novel viewpoints
  - So we can perform editing
    - Geometric edits vs. material edits vs. lighting edits
  - To aid interpretation of their contents

Many scene representations

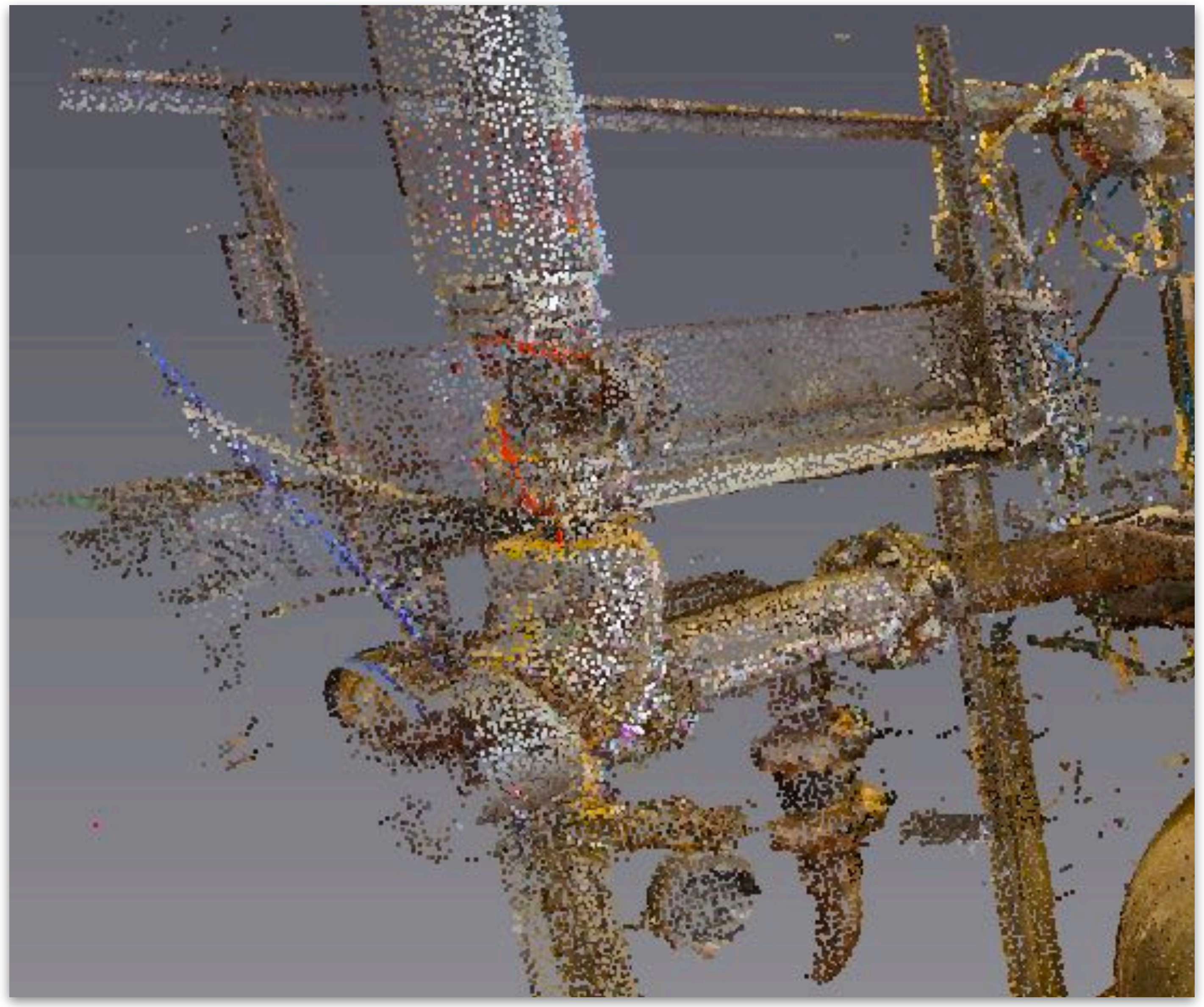


3D triangle mesh + texture map

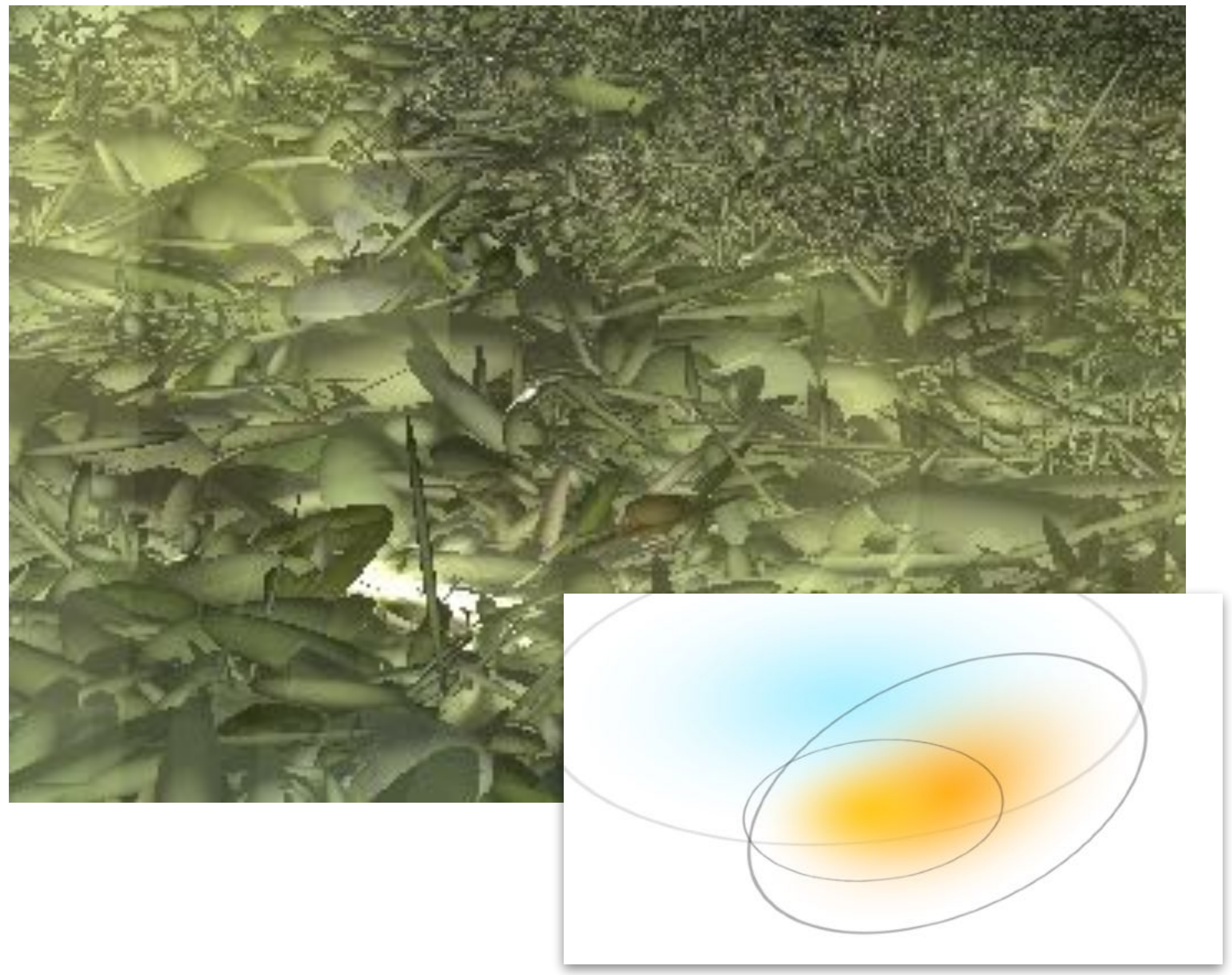
3D volume (voxels)



Sparse voxels

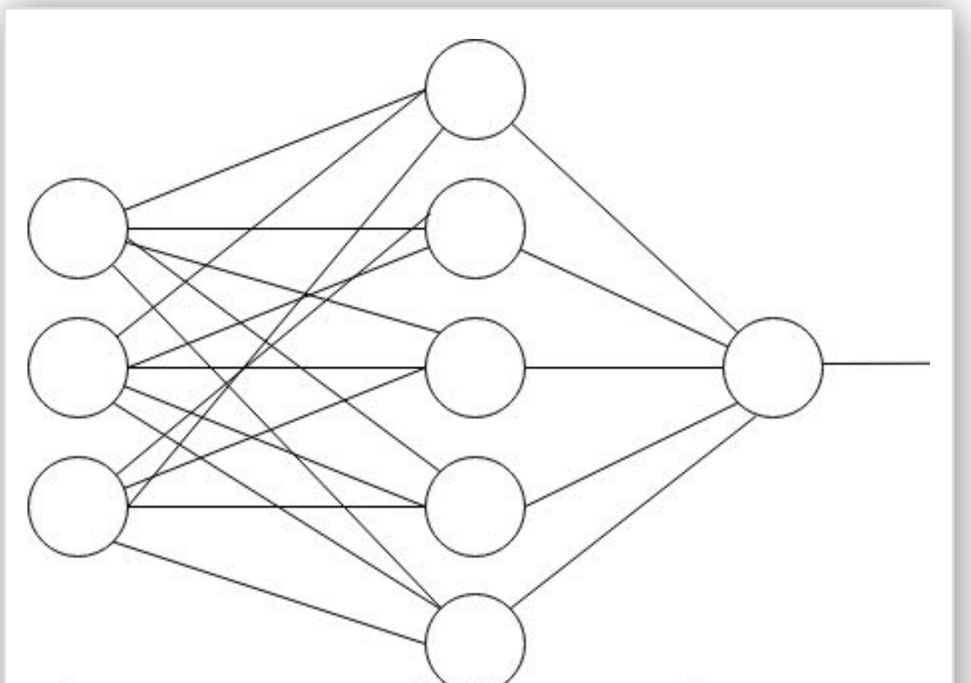


Point cloud (list of points)



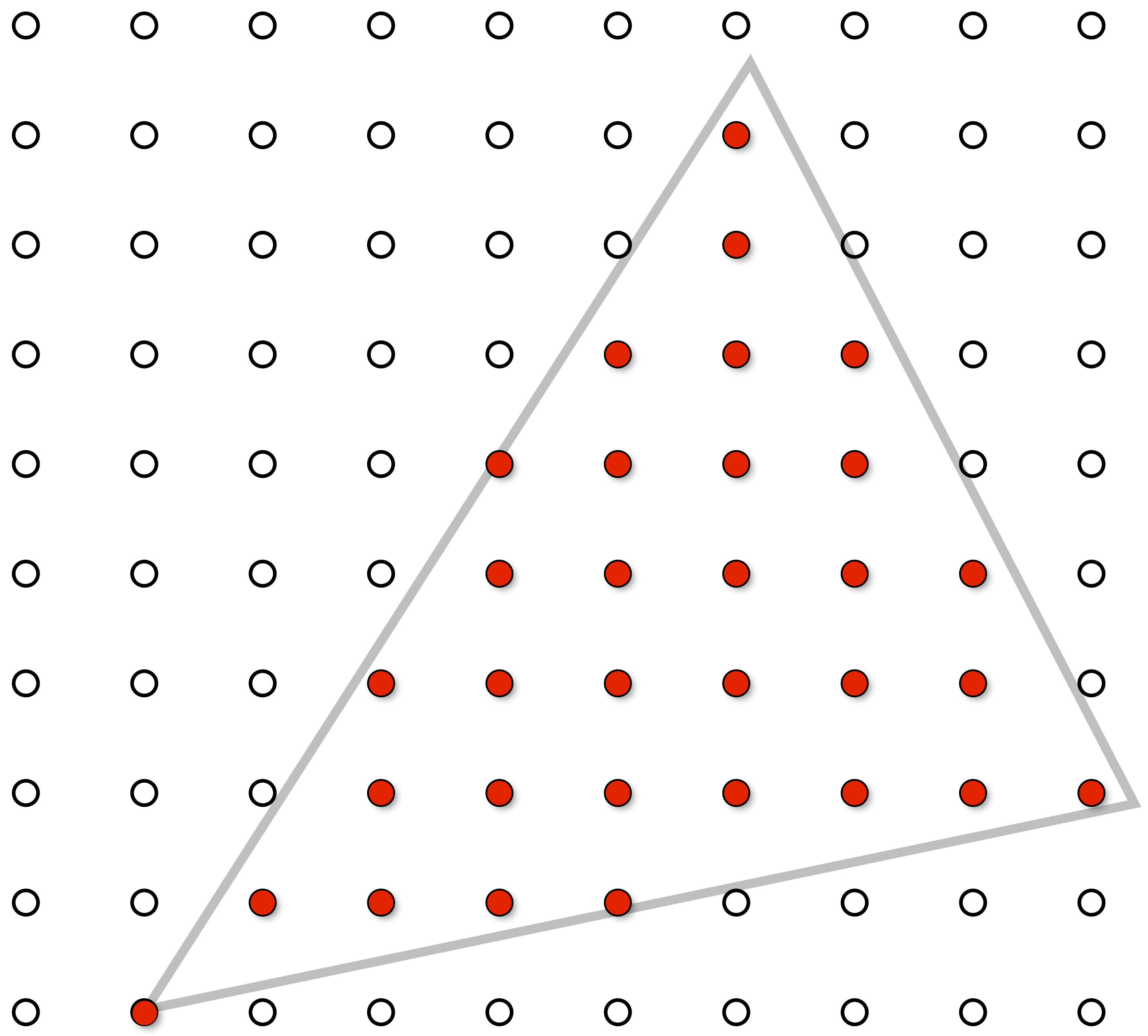
Oriented 3D Gaussians

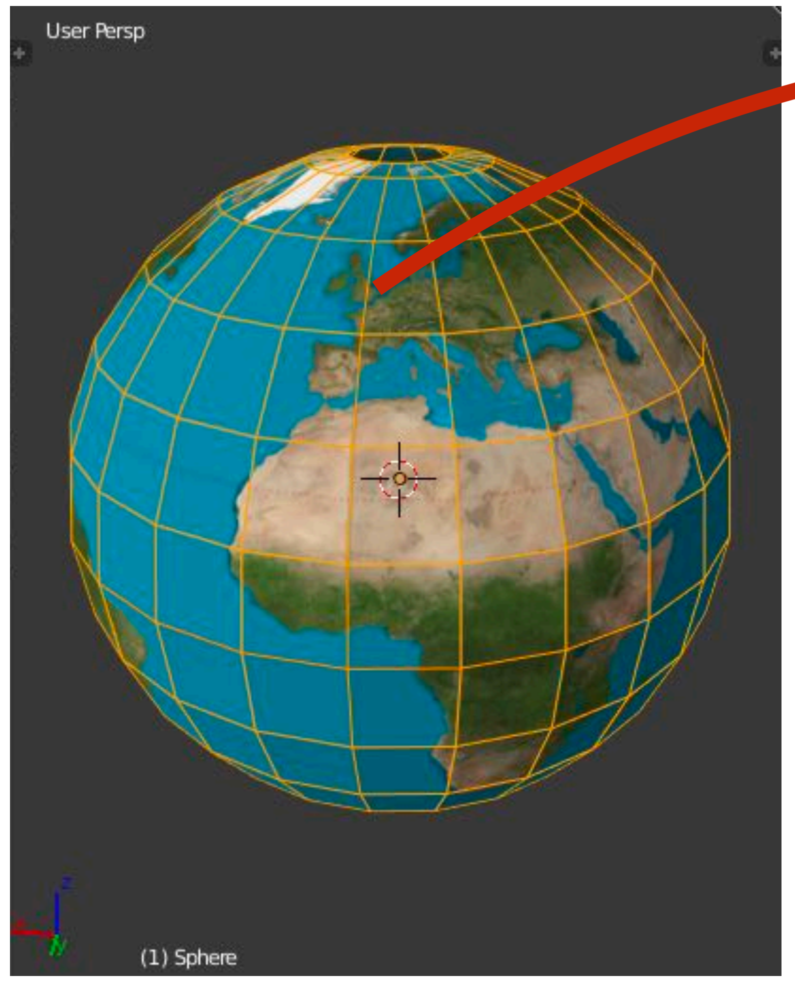
DNN (MLP)

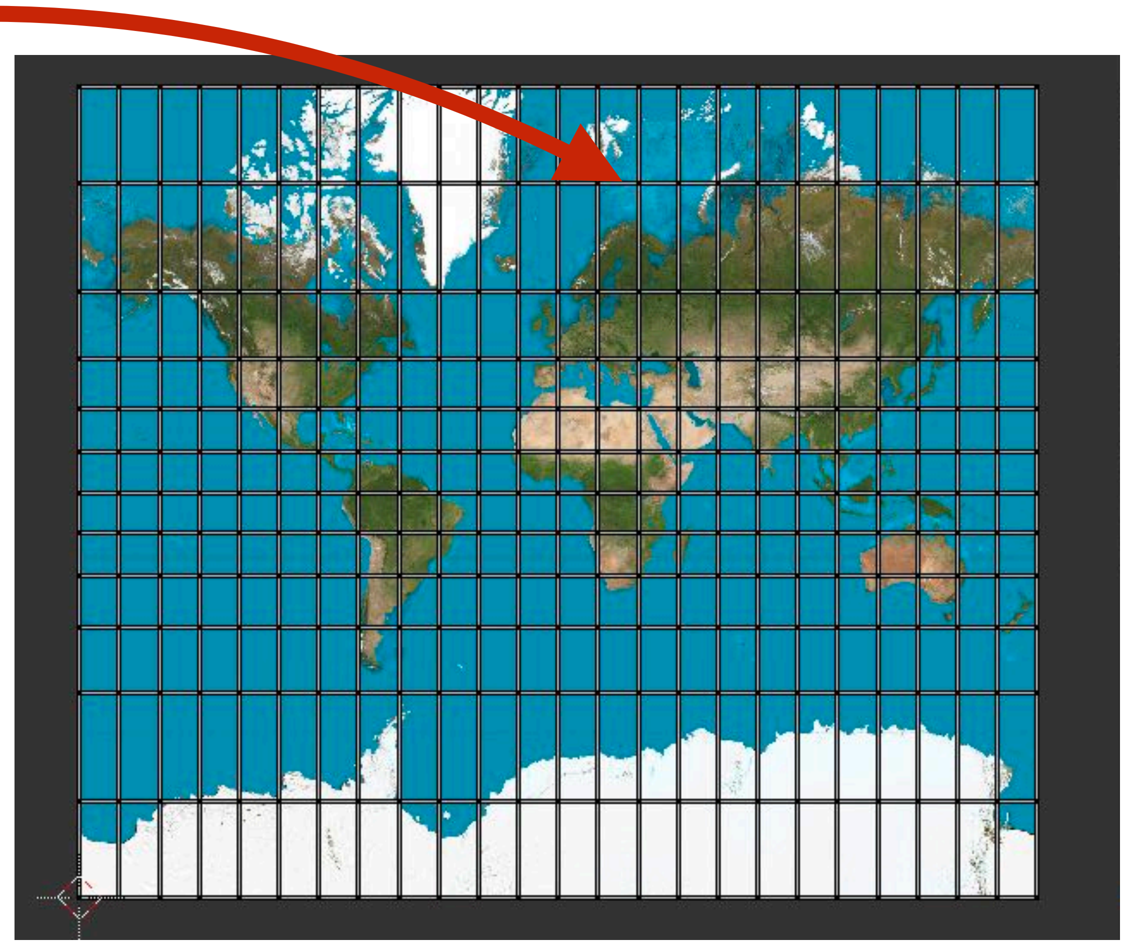


Rendering triangles

Given camera position and 3D position of vertices: (1) project vertices onto screen (2) color pixels within 2D triangle

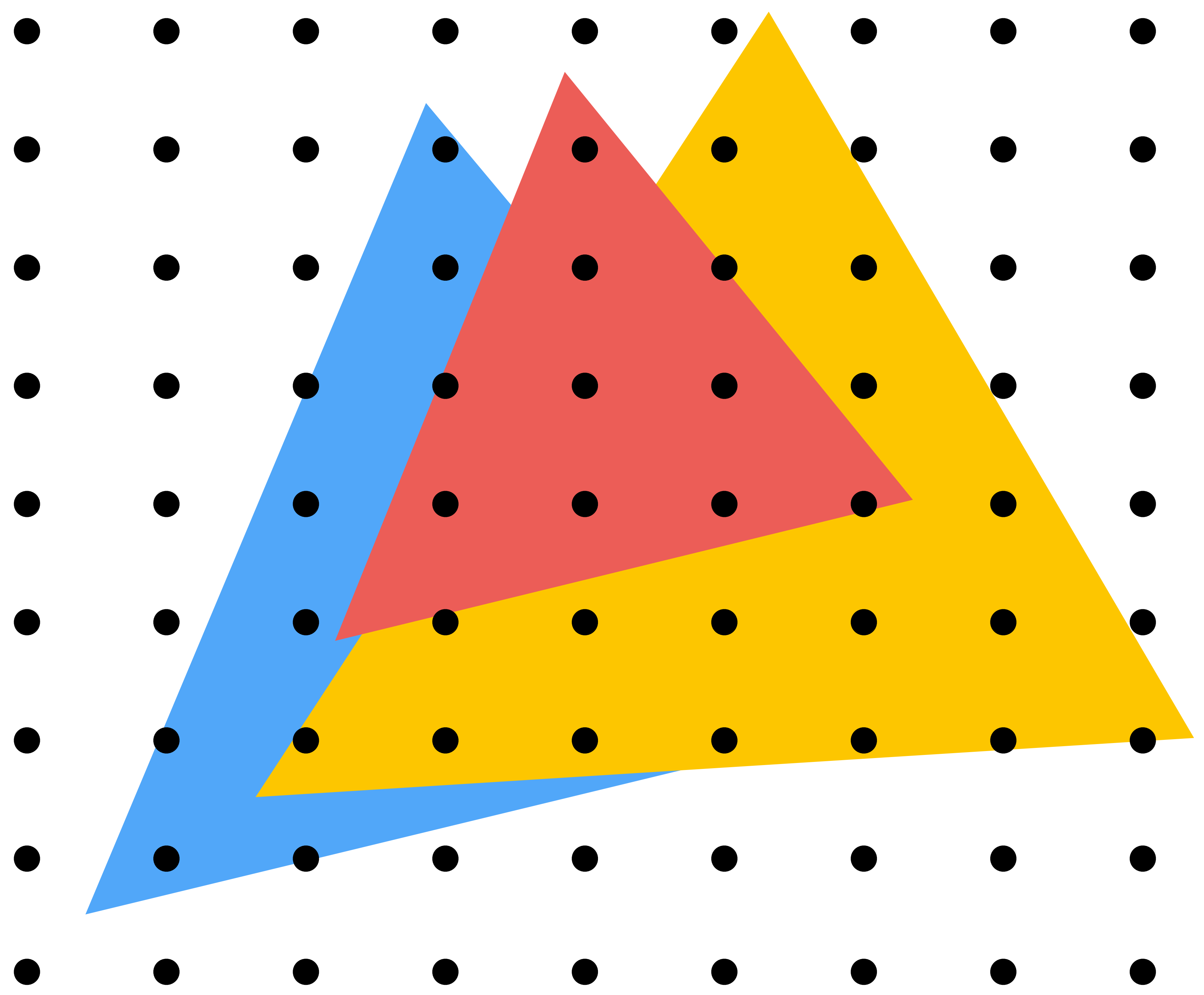






https://blender.stackexchange.com/questions/3315/how-to-get-perfect-uv-sphere-mercator-projection

Example: rendering three opaque triangles



Depth buffer (aka "Z buffer")

Color buffer:

(stores color per sample… e.g., RGB)

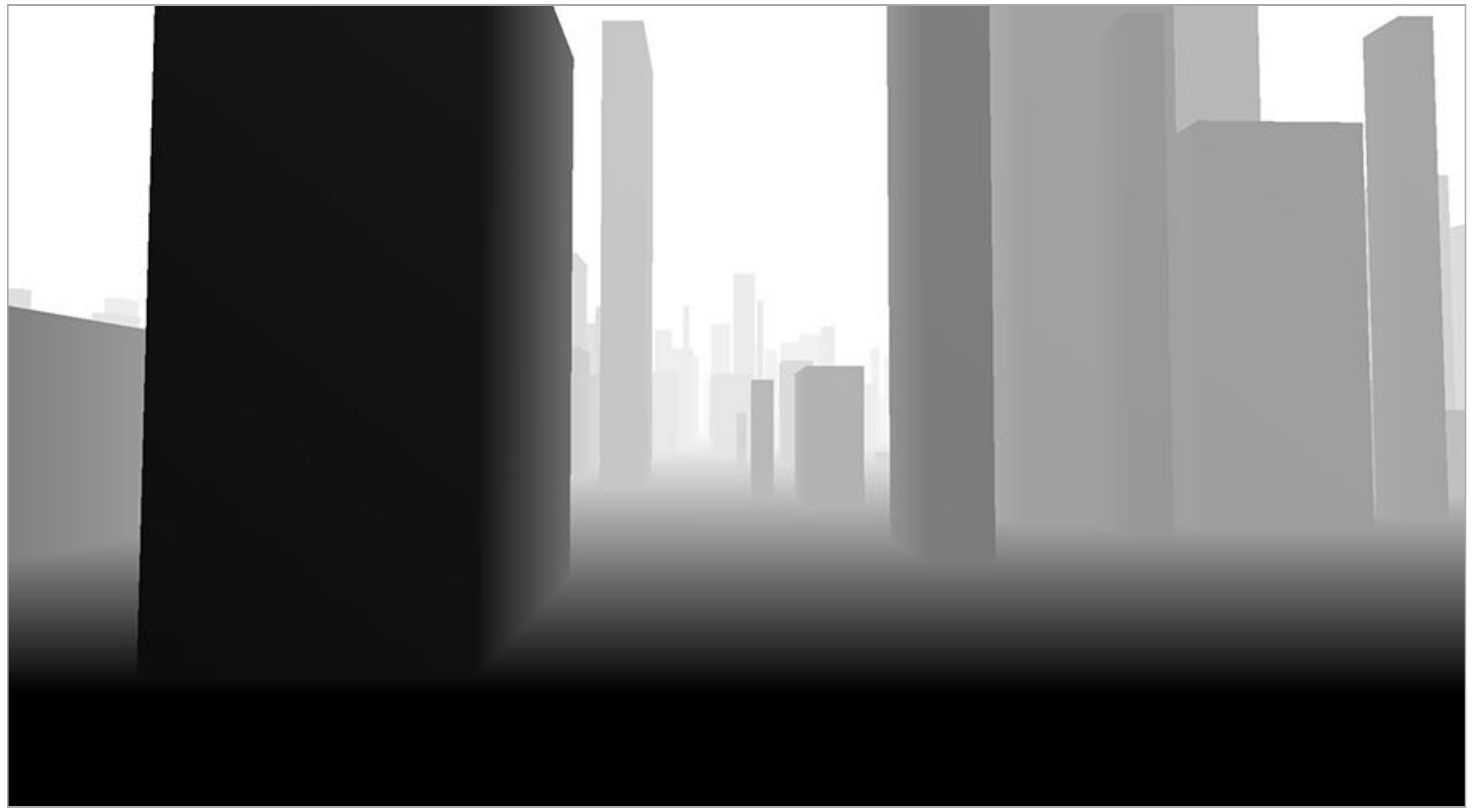
Depth buffer:

(stores depth per sample)

Stores depth of closest surface drawn so far

black = close depth

white = far depth



Occlusion using the depth buffer (opaque surfaces)

```
bool pass_depth_test(d1, d2) {
   return d1 < d2;
}


depth_test(tri_d, tri_color, x, y) {

  if (pass_depth_test(tri_d, depth_buffer[x][y]) {

    // if triangle is closest object seen so far at this
    // sample point. Update depth and color buffers.

    depth_buffer[x][y] = tri_d;   // update depth_buffer
    color[x][y] = tri_color;      // update color buffer
  }
}
```

Basic rasterization algorithm

Sample = 2D point

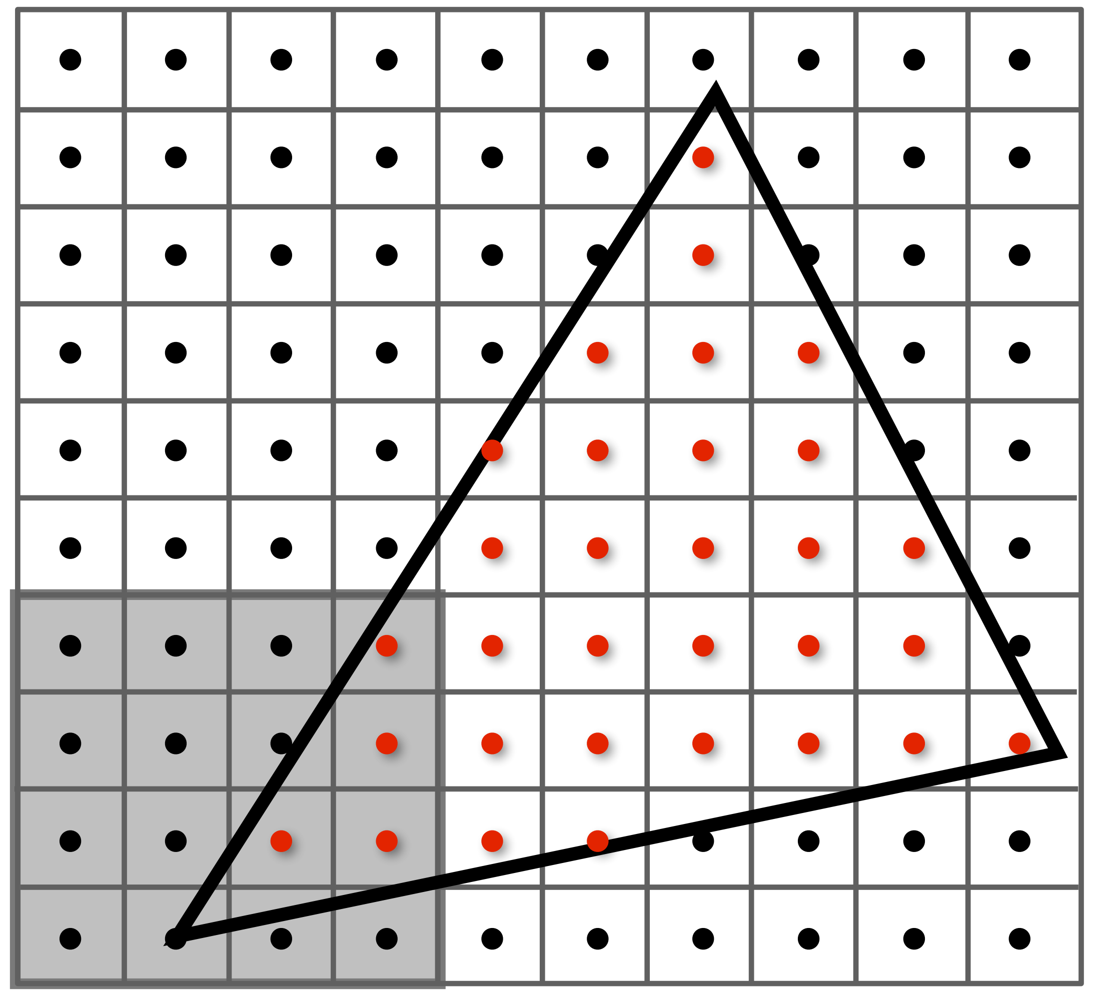
Coverage: 2D triangle/sample tests (does projected triangle cover 2D sample point)

Occlusion: depth buffer

```
initialize z_closest[] to INFINITY               // store closest-surface-so-far for all samples
initialize color[]                               // store scene color for all samples
for each triangle t in scene:                    // loop 1: over triangles
    t_proj = project_triangle(t)
    for each 2D sample s in frame buffer:        // loop 2: over visibility samples
        if (t_proj covers s)
            compute color of triangle at sample
            if (depth of t at s is closer than z_closest[s])
                update z_closest[s] and color[s]
```

"Given a triangle, find the samples it covers"

(finding the samples is relatively easy since they are distributed uniformly on screen)



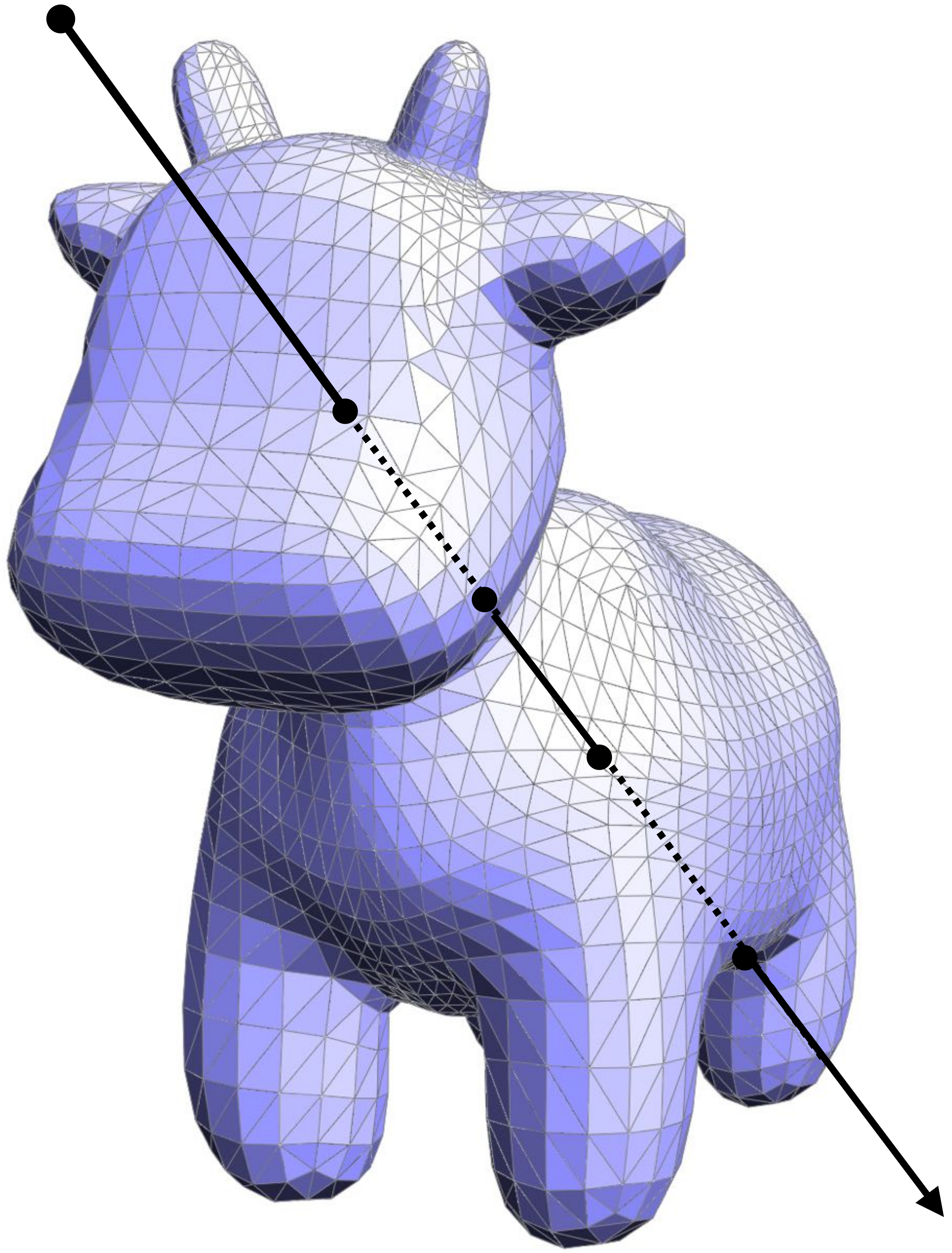
Another way of rendering triangles: ray-scene intersection

Given a scene defined by a set of *N* primitives and a ray *r*, find the closest point of intersection of *r* with the scene

```
t_closest = inf
for each primitive p in scene:
   t = p.intersect(r)
   if t >= 0 && t < t_closest:
      t_closest = t

// closest hit is:
// r.o + t_closest * r.d
```

(Assume p.intersect(r) returns value of *t* corresponding to the point of intersection with ray *r*)



Representing rays

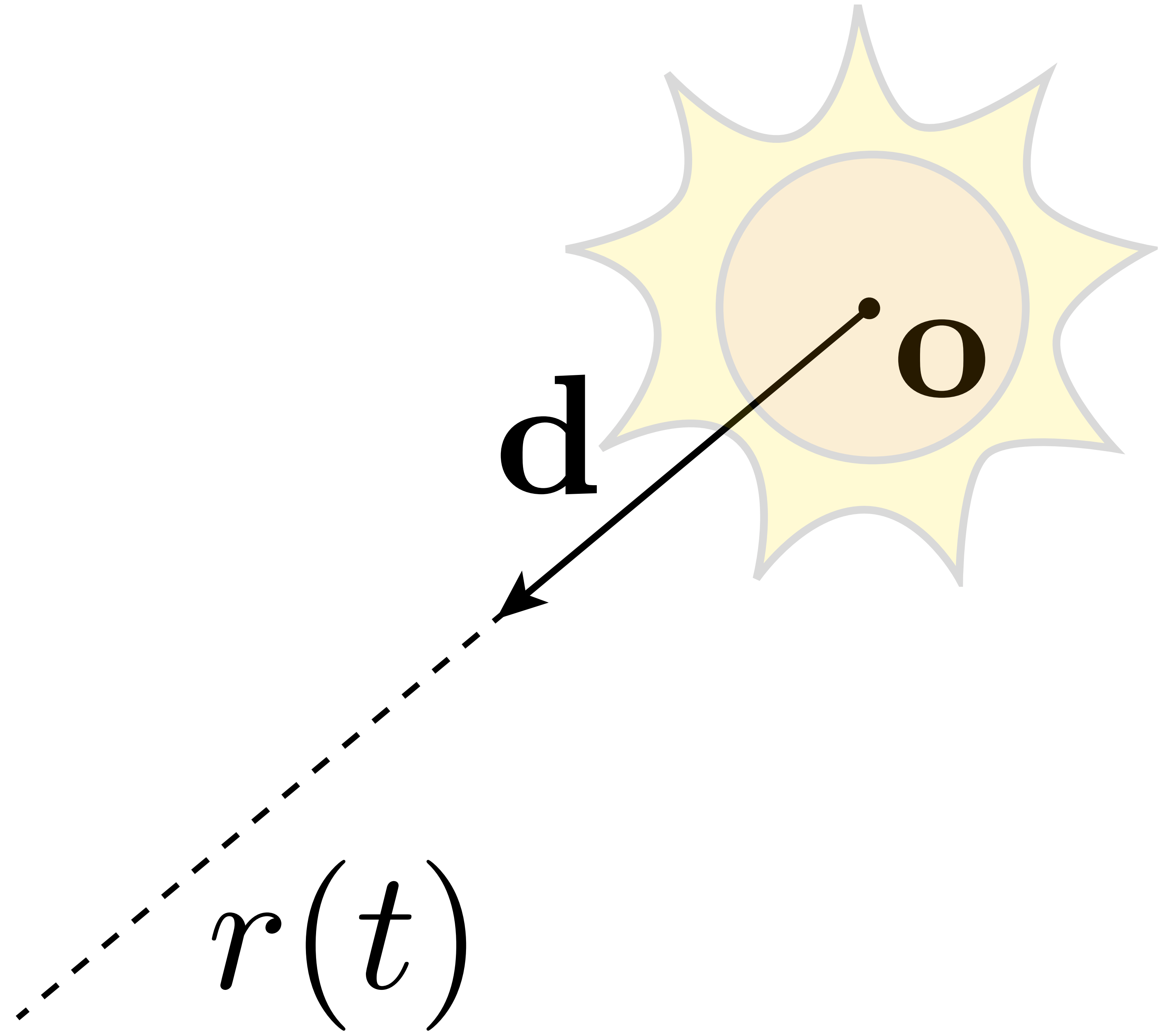
origin

unit direction

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

point along ray

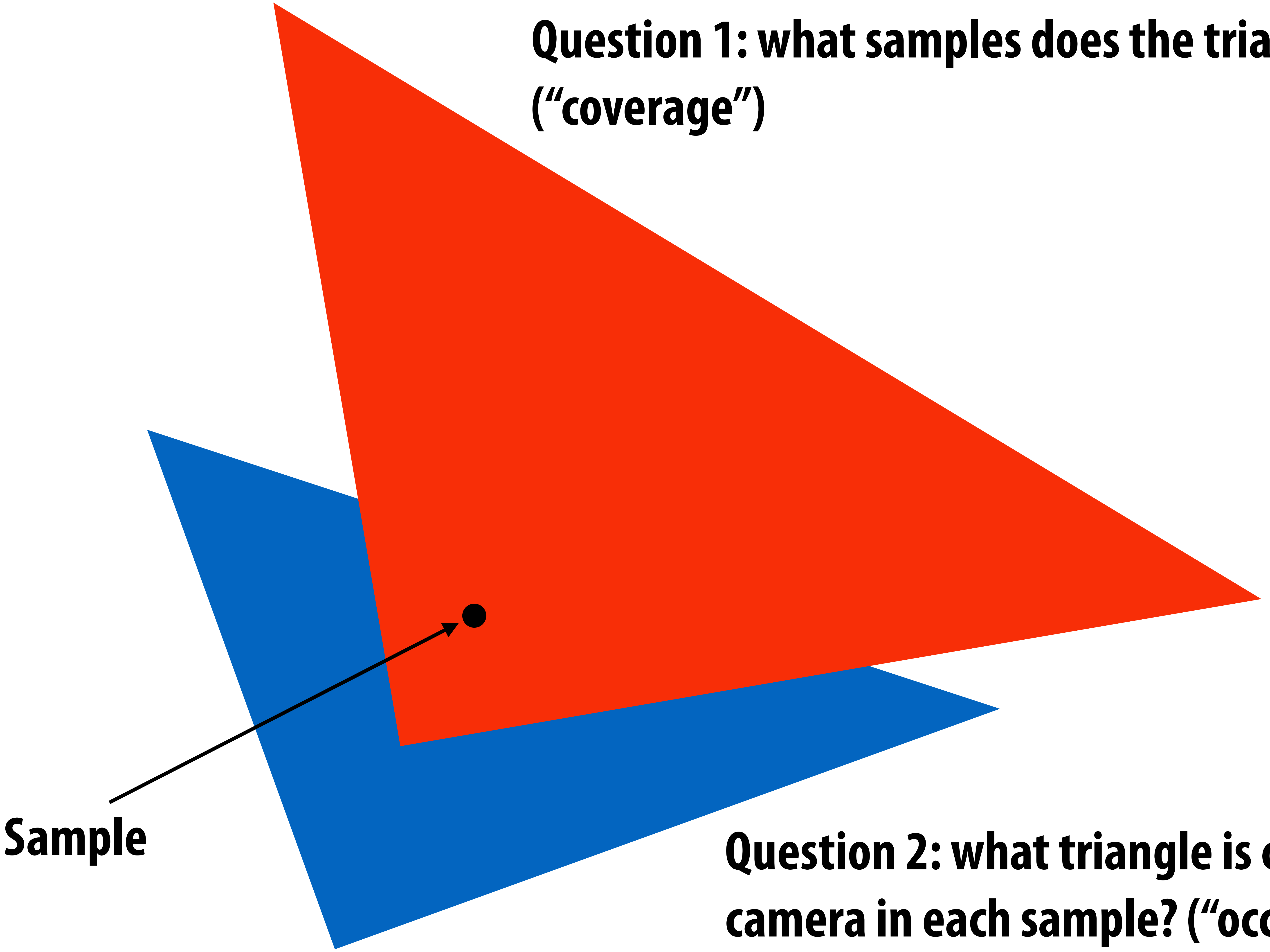
“Distance” or
“time”



Rasterization and ray casting are two algorithms for solving the same problem: determining surface "visibility" from a virtual camera

Recall triangle visibility problem:

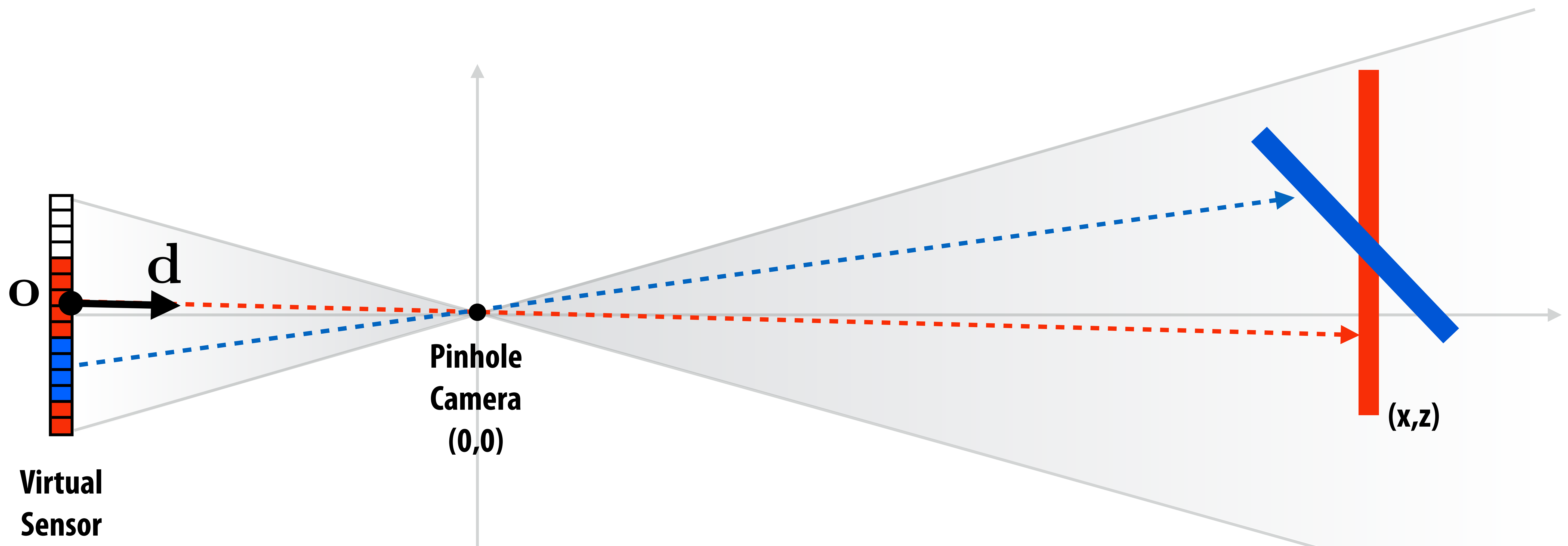
Question 1: what samples does the triangle overlap? ("coverage")



Question 2: what triangle is closest to the camera in each sample? ("occlusion")

The visibility problem (described differently)

- In terms of casting rays from the camera:
  - Is a scene primitive hit by a ray originating from a point on the virtual sensor and traveling through the opening of a pinhole camera? (coverage)
  - What primitive is the first hit along that ray? (occlusion)



Basic ray casting algorithm

Sample = a ray in 3D

Coverage: 3D ray-triangle intersection tests (does ray "hit" triangle)

Occlusion: closest intersection along ray

```
initialize color[]                                   // store scene color for all samples
for each sample s in frame buffer:                   // loop 1: over visibility samples (rays)
    r = ray from s on sensor through pinhole aperture
    r.min_t = INFINITY                               // only store closest-so-far for current ray
    r.tri = NULL;
    for each triangle tri in scene:                   // loop 2: over triangles
        if (intersects(r, tri)) {                     // 3D ray-triangle intersection test
            if (intersection distance along ray is closer than r.min_t)
                update r.min_t and r.tri = tri;
        }
    color[s] = compute surface color of triangle r.tri at hit point
```

Compared to rasterization approach: just a reordering of the loops!

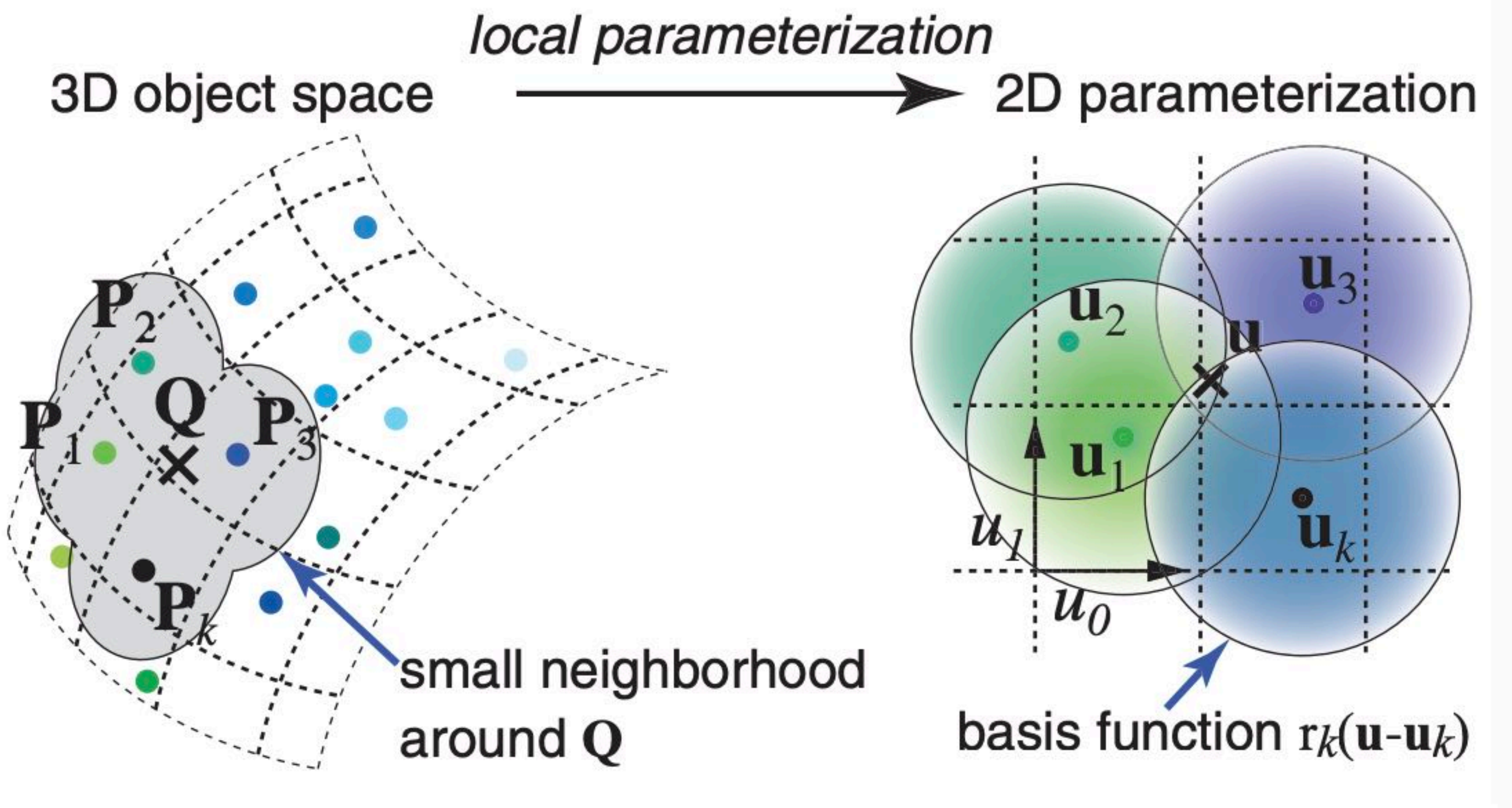
"Given a ray, find the closest triangle it hits."

Rendering 3D points

Given camera position and 3D position of point: (1) project point onto screen (2) color pixel if closest

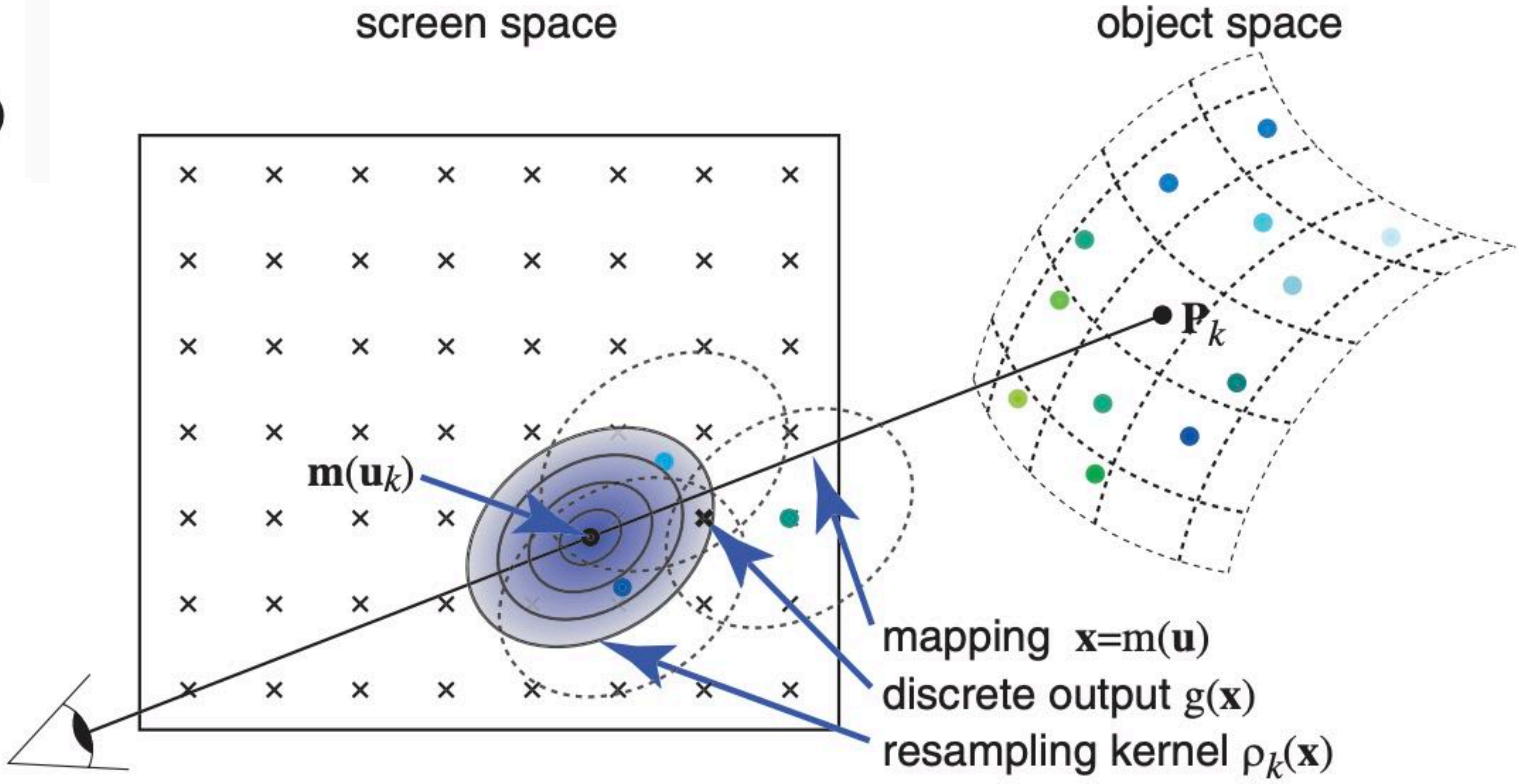
Rendering “splats” / 3D gaussians / surfels

- Treat surface as a collection of “Gaussian blobs” (convolve points with Gaussian filter)



- 3D Gaussians turn into oriented 2D gaussians when projected onto the 2D screen

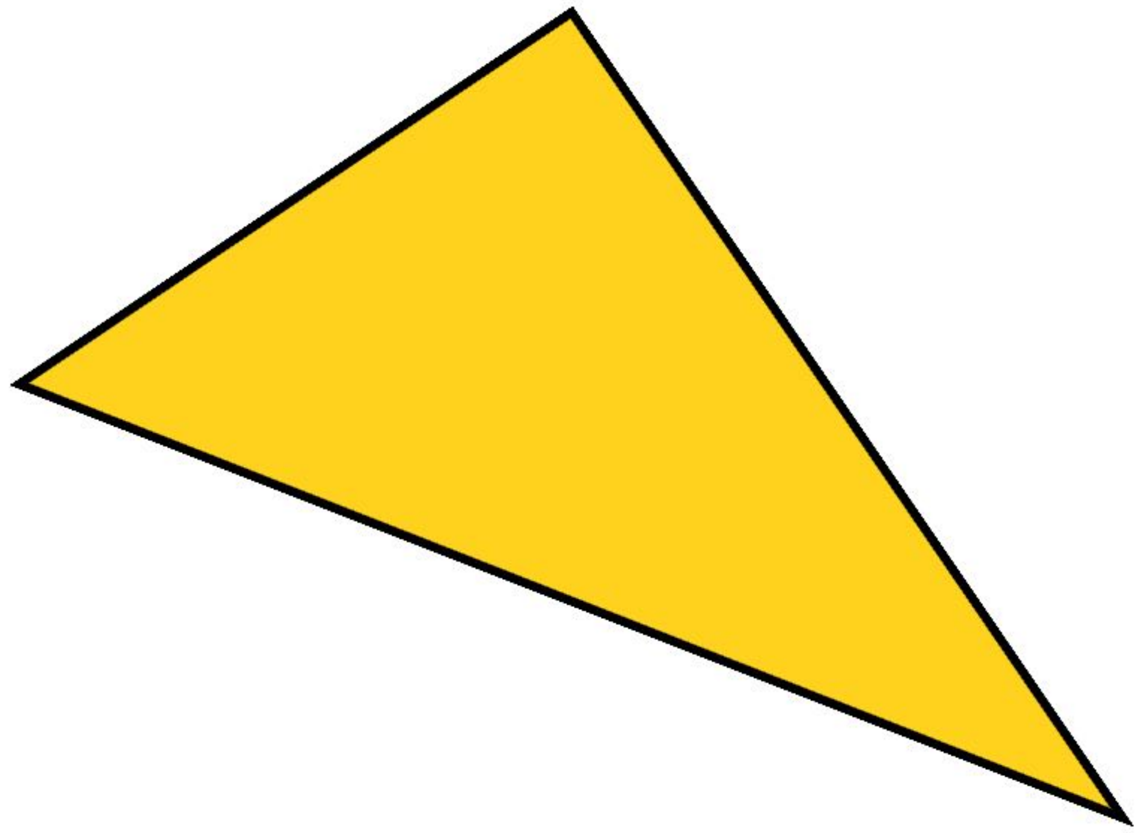
- Can render the blobs back to front (requires alpha compositing)



[Zwicker 2001]

Consider representing a triangle with Gaussians

- Could approximate the triangle with a lot of small 3D gaussians near the triangle's edges
- Not so efficient, eh?



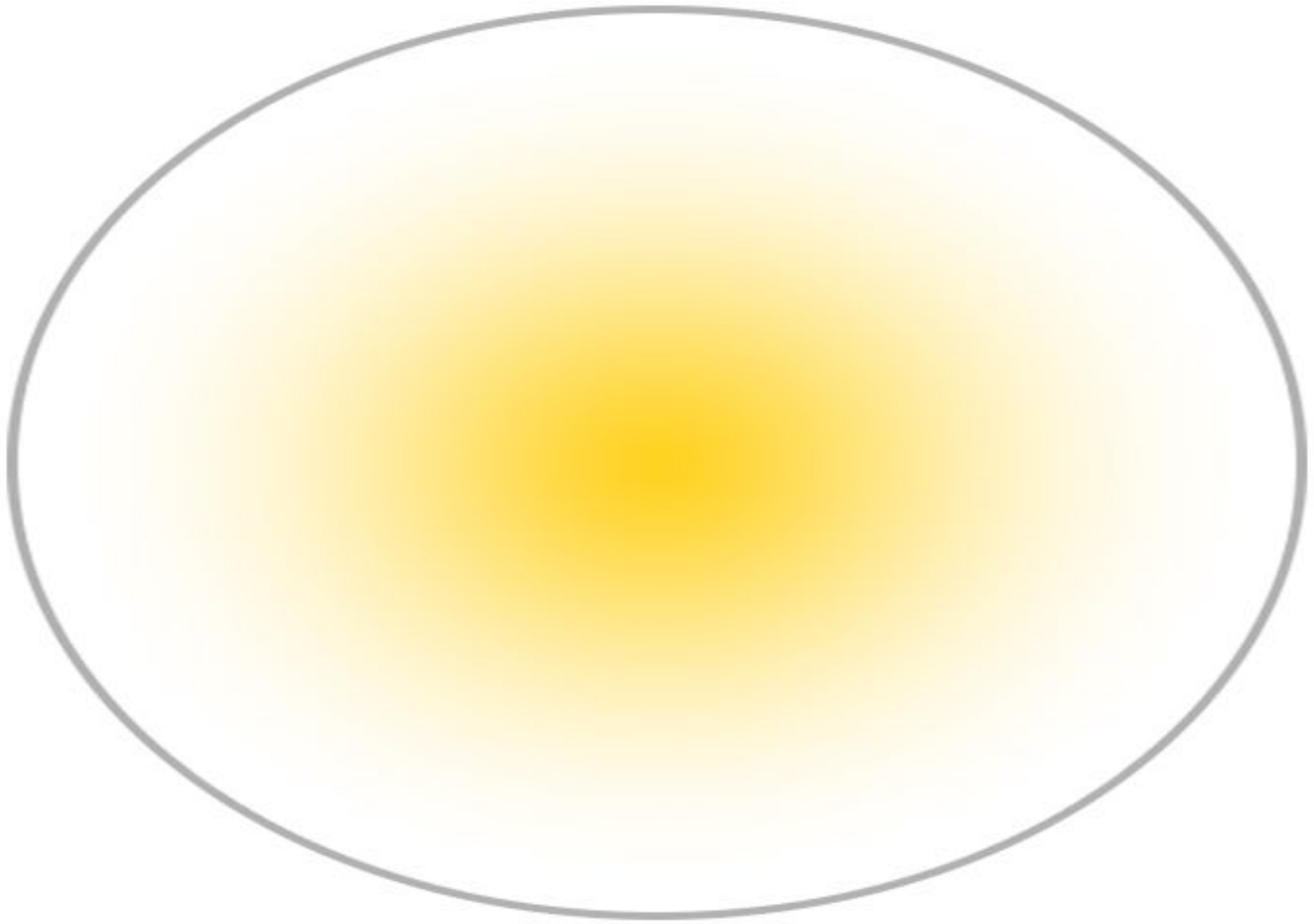


Image Credit: [Dylan Ebert] https://huggingface.co/blog/gaussian-splatting

Another representation: regular 3D grid representation

Consider storage requirements:
1024^3 cells

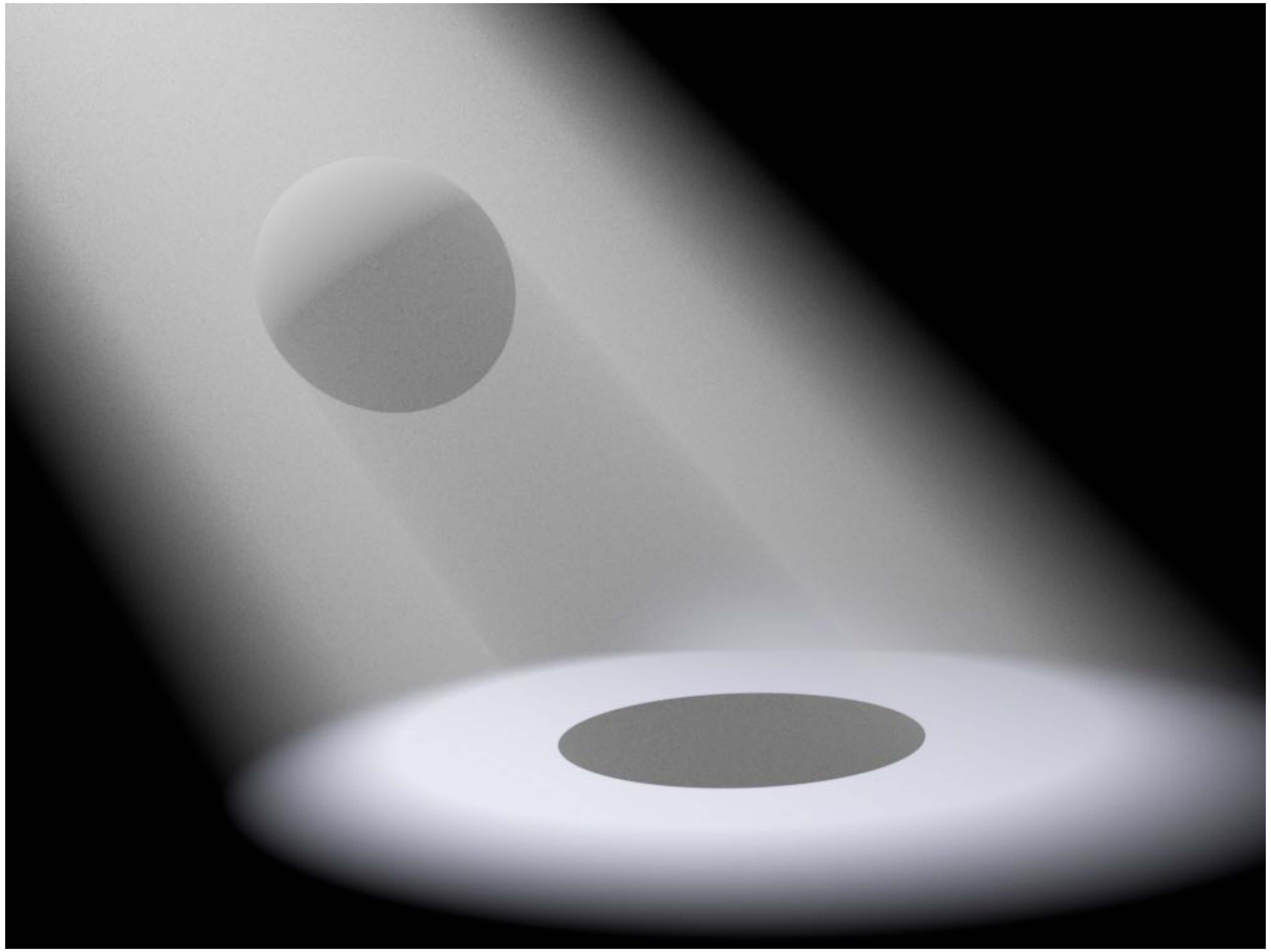
Ignore directional dependency: rgbσ 4 bytes/cell
(~4 GB)

Now consider directional dependency on (ϕ, θ)
… much worse

Typical challenge of dense voxel representations: limited resolution

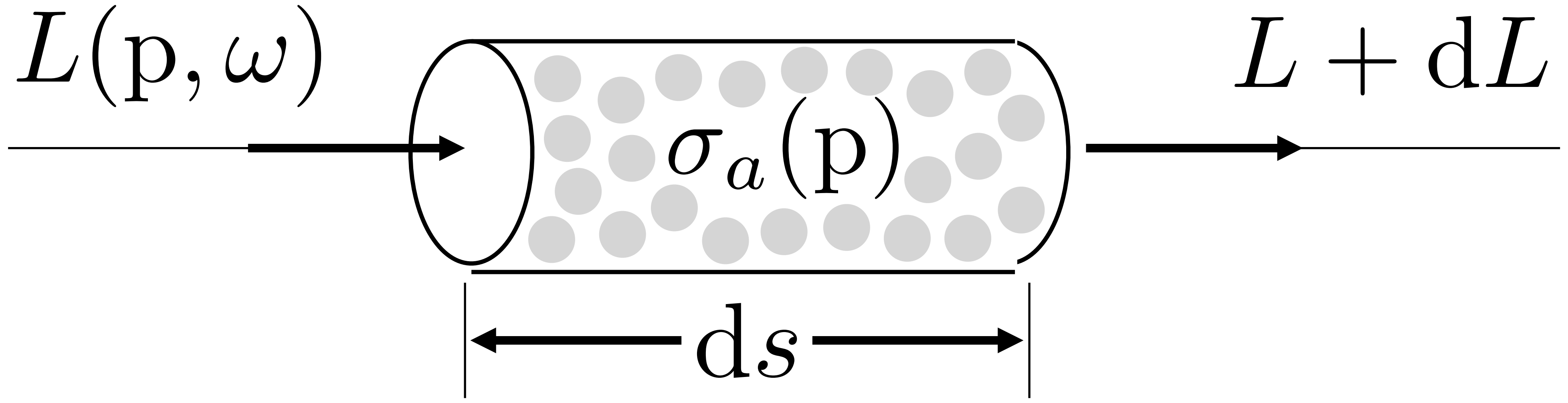
Credit: Voxel Ville NFT (voxelville.io)

Volumetric effects



Another motivation for non-triangle representations like points/gaussians/volumes: hard to accurately estimate surface triangle mesh in complex real-world situations

Absorption in a volume



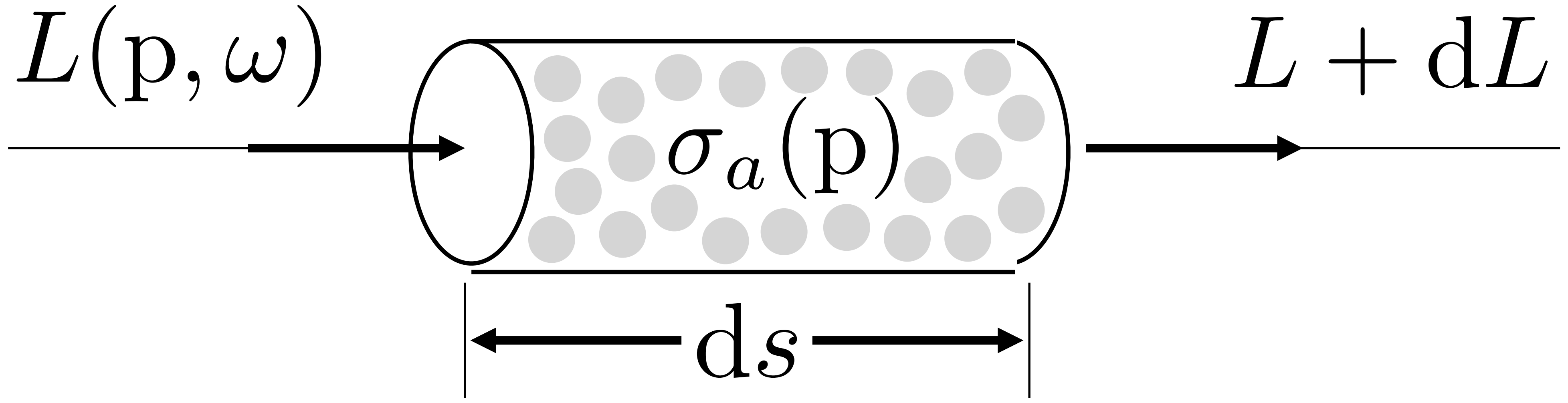
$$\mathrm{p} = (x, y, z)$$
$$\omega = (\phi, \theta)$$

$$\mathrm{d}L(\mathrm{p}, \omega) = -\sigma_a(\mathrm{p})\, L(\mathrm{p}, \omega)\, \mathrm{d}s$$

$$\frac{dL(\mathrm{p}, \omega)}{ds} = -\sigma_a(\mathrm{p}) L(\mathrm{p}, \omega)$$

- $L(\mathrm{p}, \omega)$ **radiance along a ray from *p* in direction ω**
- **Absorption cross section at point in space:** $\sigma_a(\mathrm{p})$
  - **Probability of being absorbed per unit length**
  - **Units: 1/distance**

Absorption in a volume



$$\mathrm{p} = (x, y, z)$$

$$\omega = (\phi, \theta)$$

$$\frac{dL(\mathrm{p}, \omega)}{L(\mathrm{p}, \omega)} = -\sigma_a(\mathrm{p})\mathrm{d}s$$

$$L(\mathrm{p} + s\omega, \omega) = \mathrm{e}^{-\int_0^s \sigma_\mathrm{a}(\mathrm{p}+s'\omega)\,\mathrm{d}s'} L(\mathrm{p}, \omega) = T(s)\, L(\mathrm{p}, \omega)$$

Transmittance: $T(s) = \mathrm{e}^{-\int_0^s \sigma_\mathrm{a}(\mathrm{p}+s'\omega, \omega)\,\mathrm{d}s'}$

Absorption: lower density

Credit: Walt Disney Animation Studios

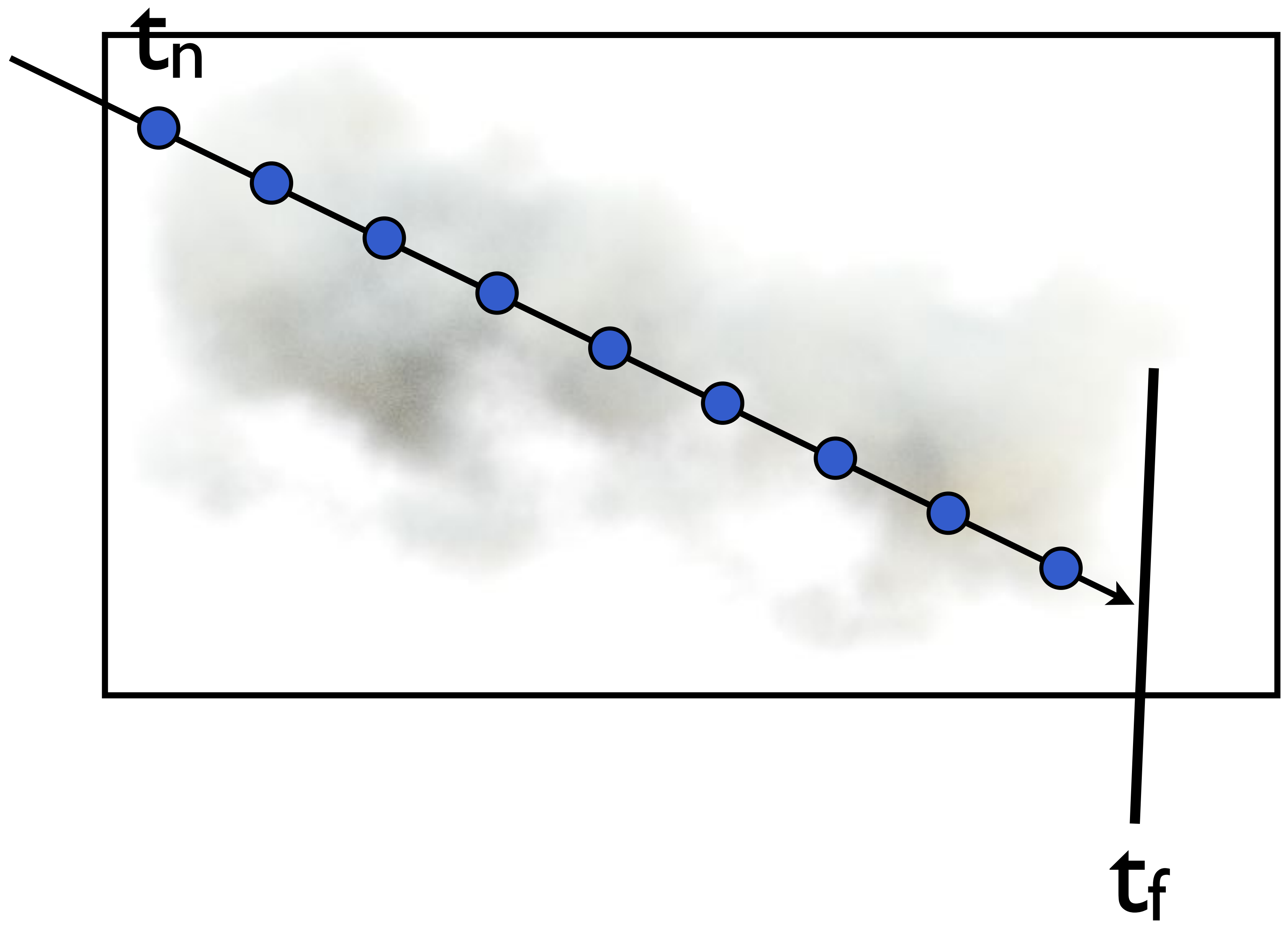
Absorption: higher density

Credit: Walt Disney Animation Studios

Rendering volumes

$$\sigma(\mathrm{p})$$
$$c(\mathrm{p}, \omega)$$

Volume density and color at all points in space.
e.g., Values stored in a 3D grid



$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt\,, \text{ where } T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right)$$

Dense 3D volumes = high storage cost

Consider storage requirements:
1024^3 cells

Ignore directional dependency: rgbσ 4 bytes/cell
(~4 GB)

Now consider directional dependency on (ϕ, θ)
… much worse

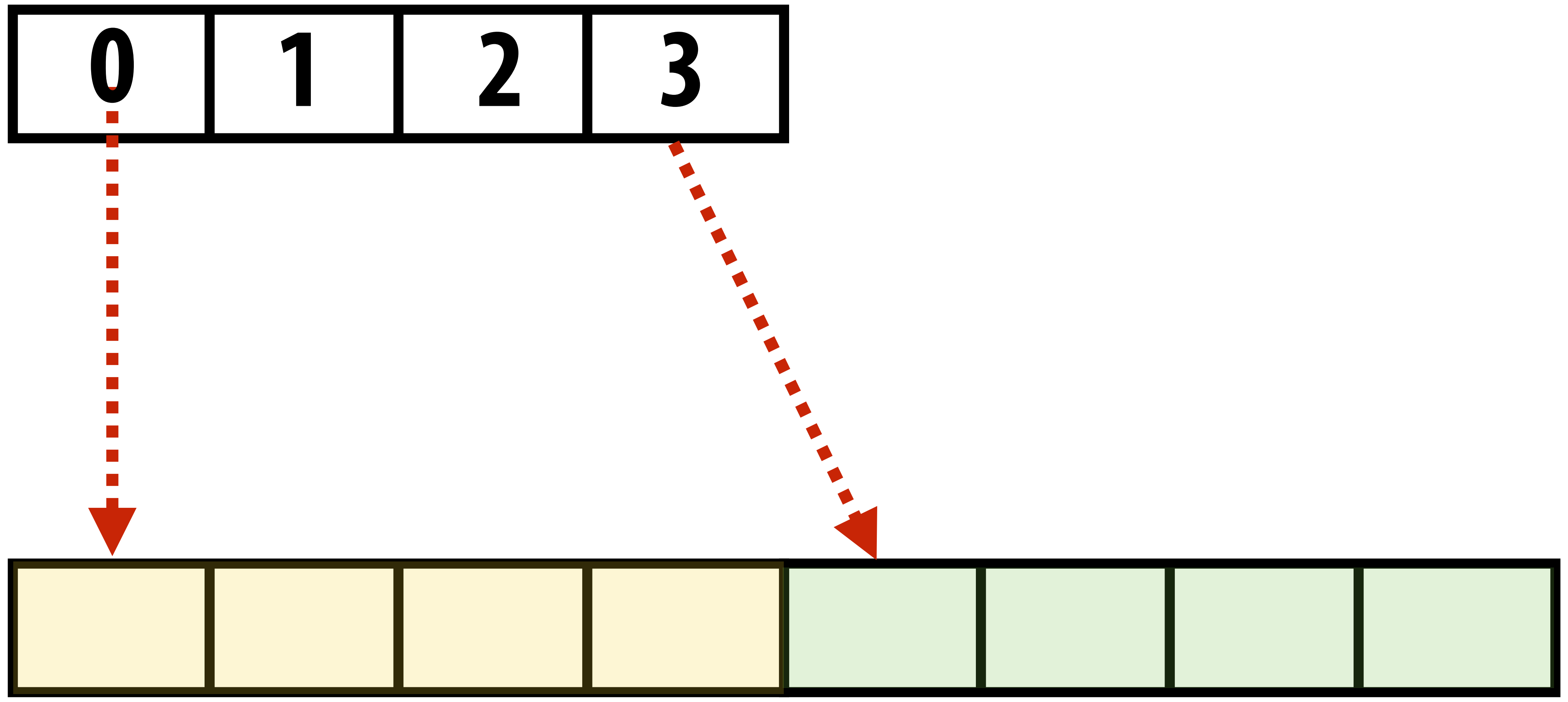
Typical challenge of dense voxel representations: limited resolution

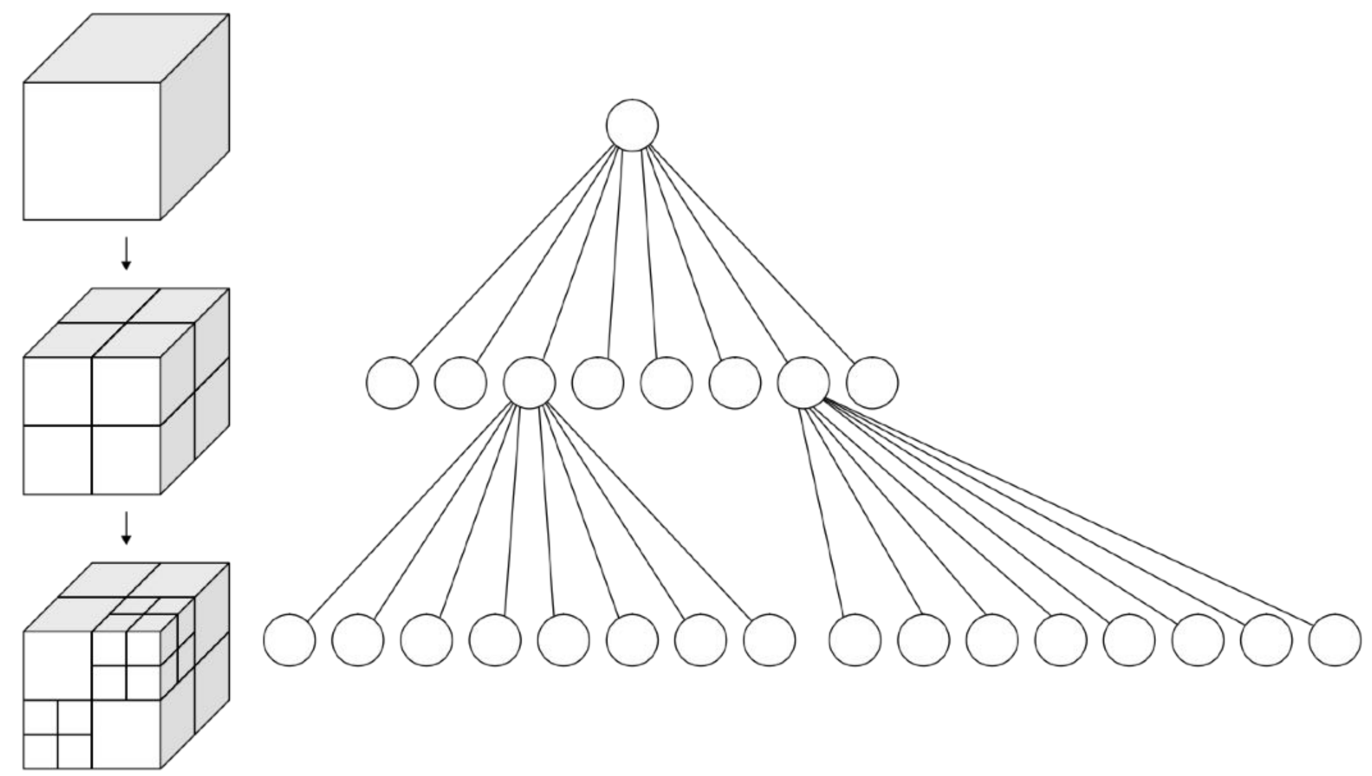
Credit: Voxel Ville NFT (voxelville.io)

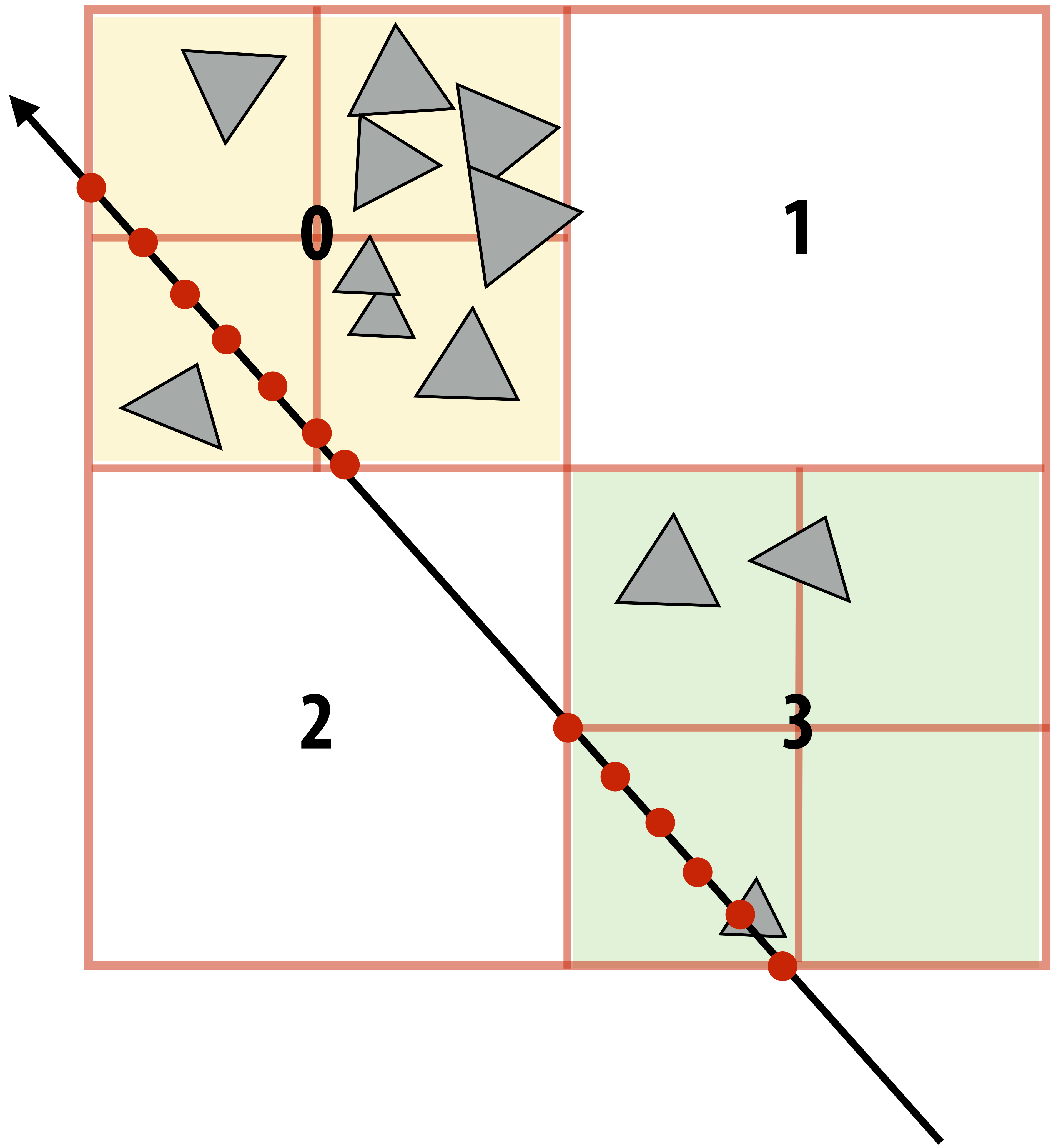
Sparse volumes

Quad-tree: nodes have 4 children (partitions 2D space)

Octree: nodes have 8 children (partitions 3D space)



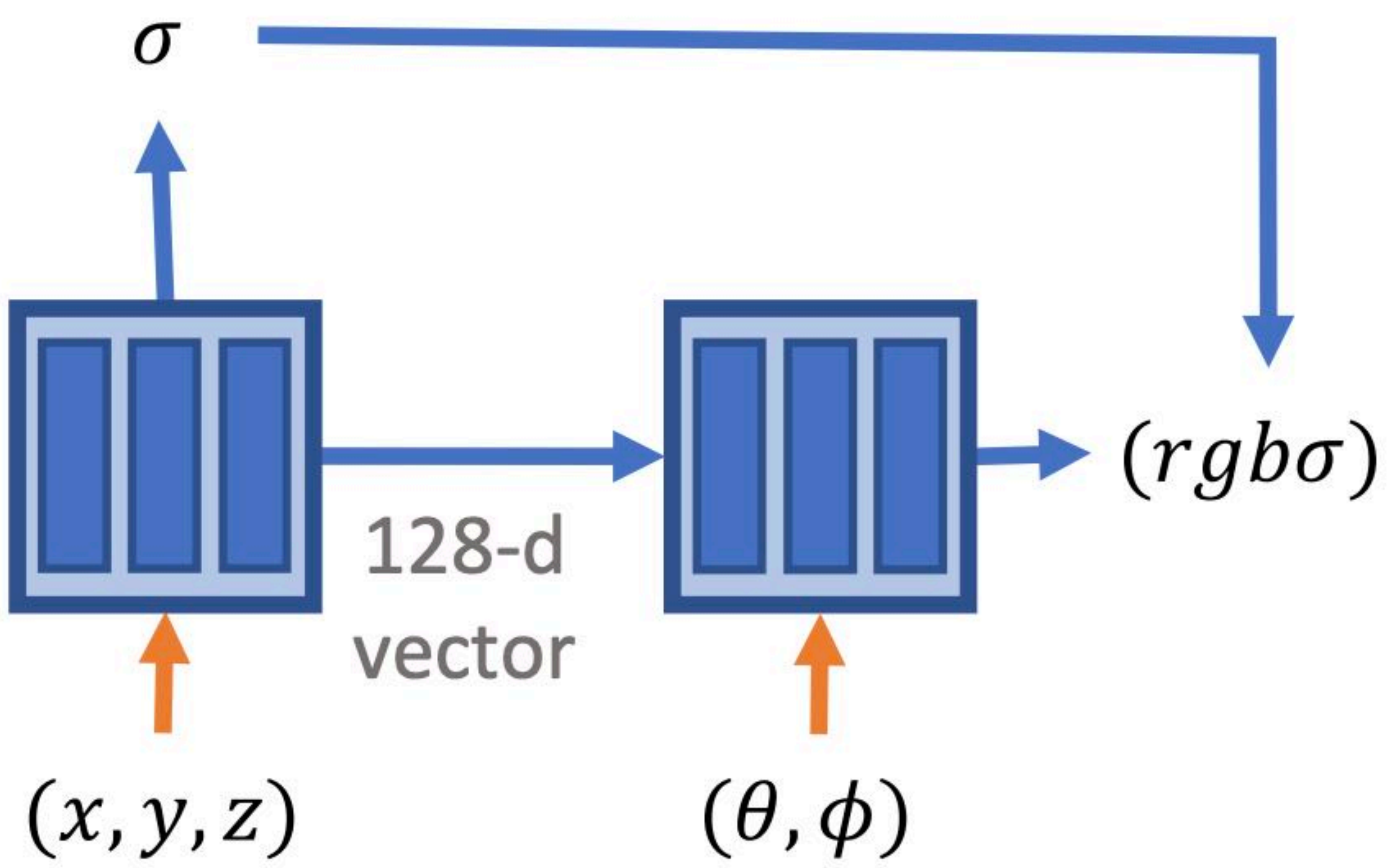




A very compressed volume representation

- Use DNN to compress information in a volume
- Why not just learn an approximation to the continuous function that matches observations from different viewpoints?

$$(\mathrm{p}, \omega) \rightarrow \boxed{F_\theta(\mathrm{p}, \omega)} \rightarrow \begin{matrix} \sigma(\mathrm{p}) \\ c(\mathrm{p}, \omega) \end{matrix}$$



Many different parameterizations

- Mesh vertex positions + texture values
- 3D point positions + colors
- 3D oriented Gaussians + colors
- Dense 3D voxels
- Sparse 3D voxels
- DNN weights
- Many combinations not discussed: sparse 3D grid of DNN weights, hash table of DNN weights, etc…

Reconstruction problem

- Given many views of a scene for which camera position is known, recover the parameters of a scene representation SO THAT rendering the scene representation from that known view generates the captures image!

- Need a differentiable renderer to recover parameters!

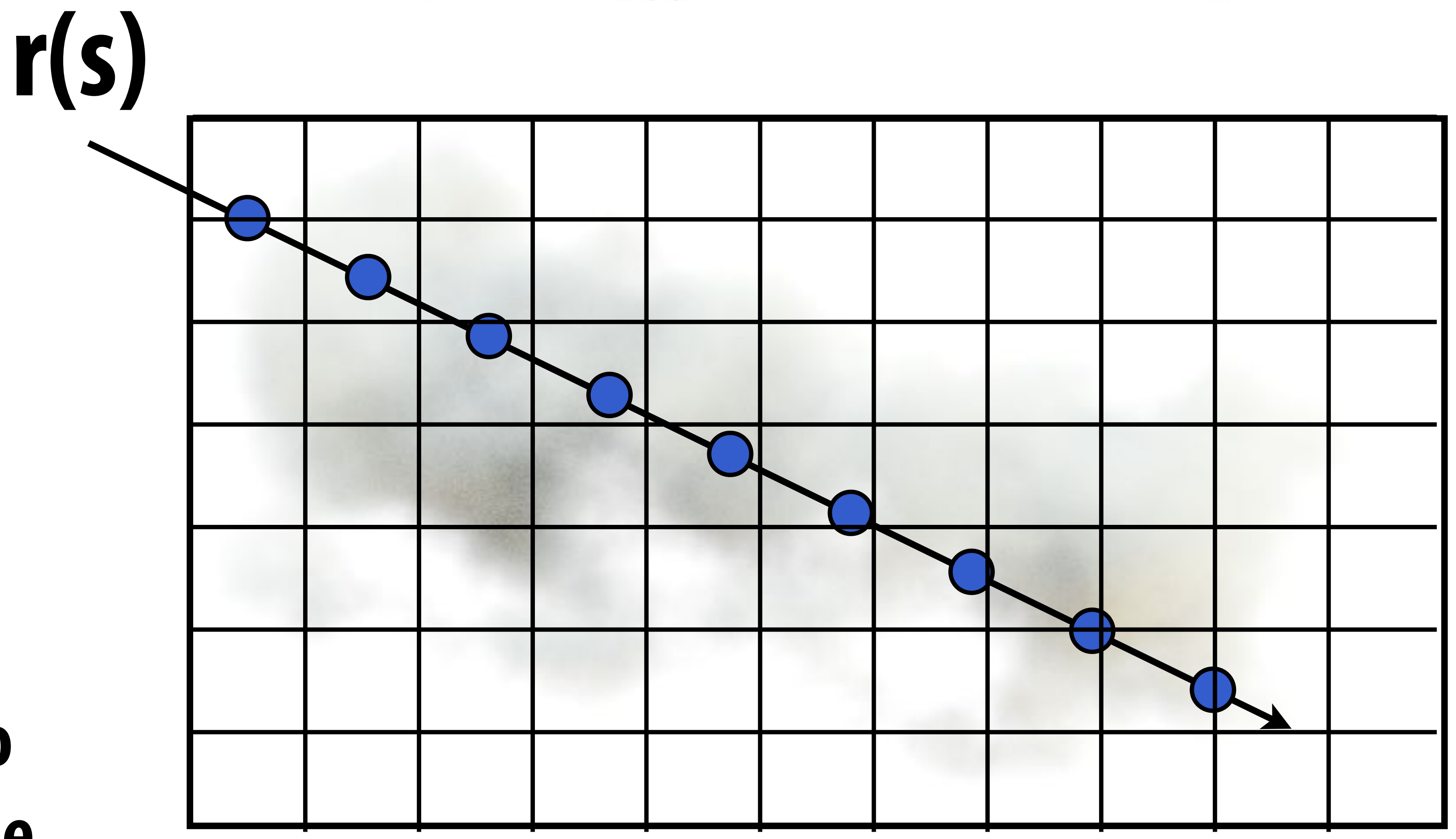
Novel view synthesis problem

Input photos (from a fixed set of views)

Novel views
(camera position different from those in input photos)

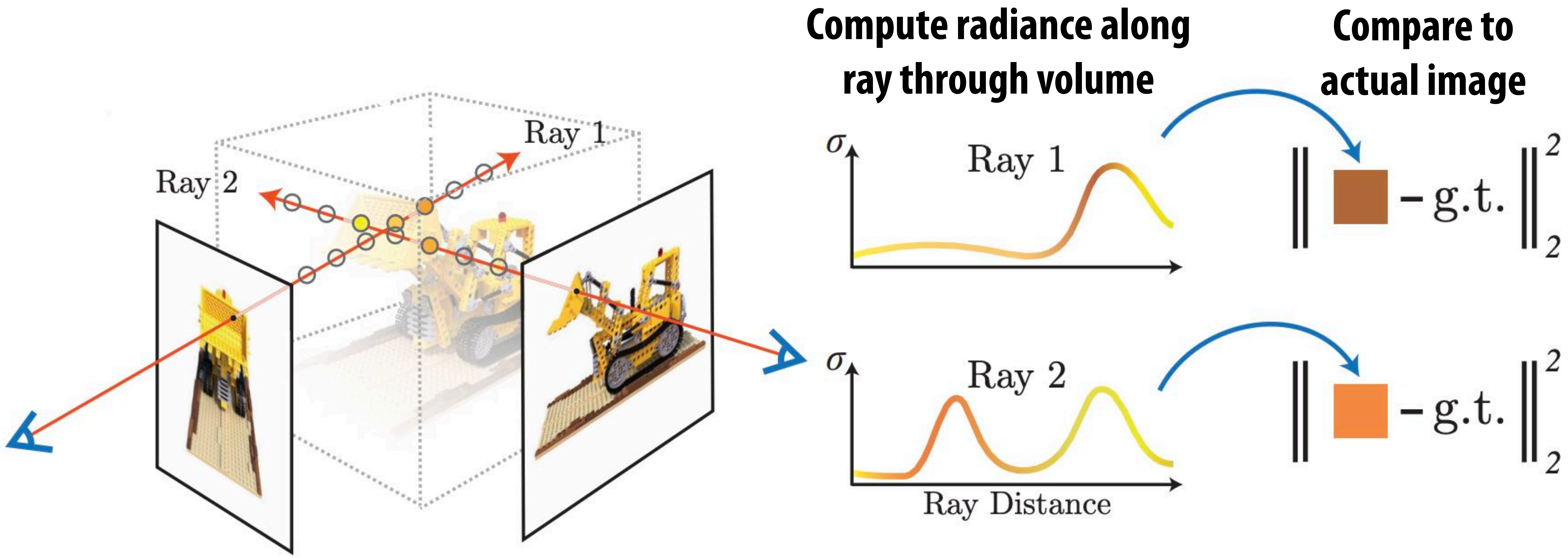
Optimizing volumes

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t),\mathbf{d})dt\,, \text{ where } T(t) = \exp\left(-\int_{t_n}^{t}\sigma(\mathbf{r}(s))ds\right)$$

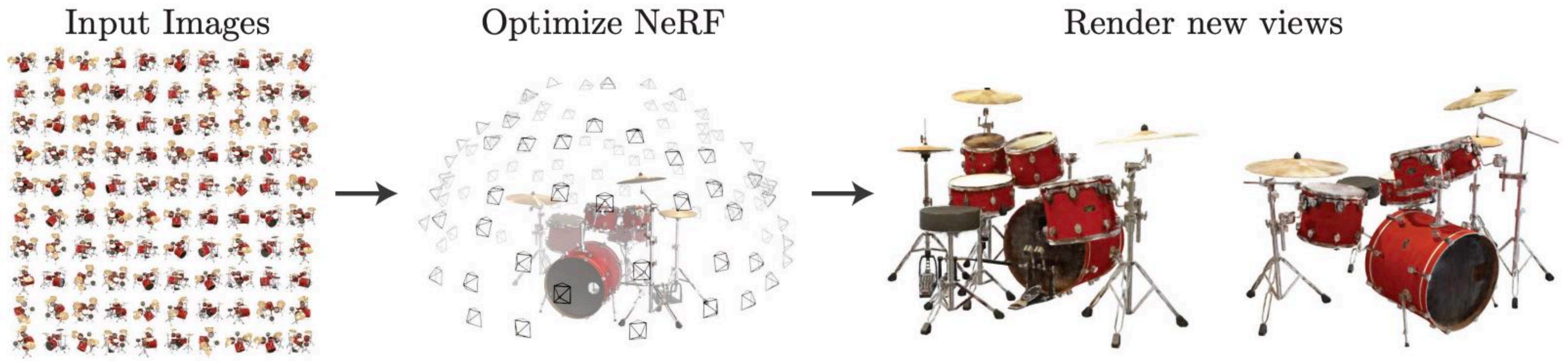


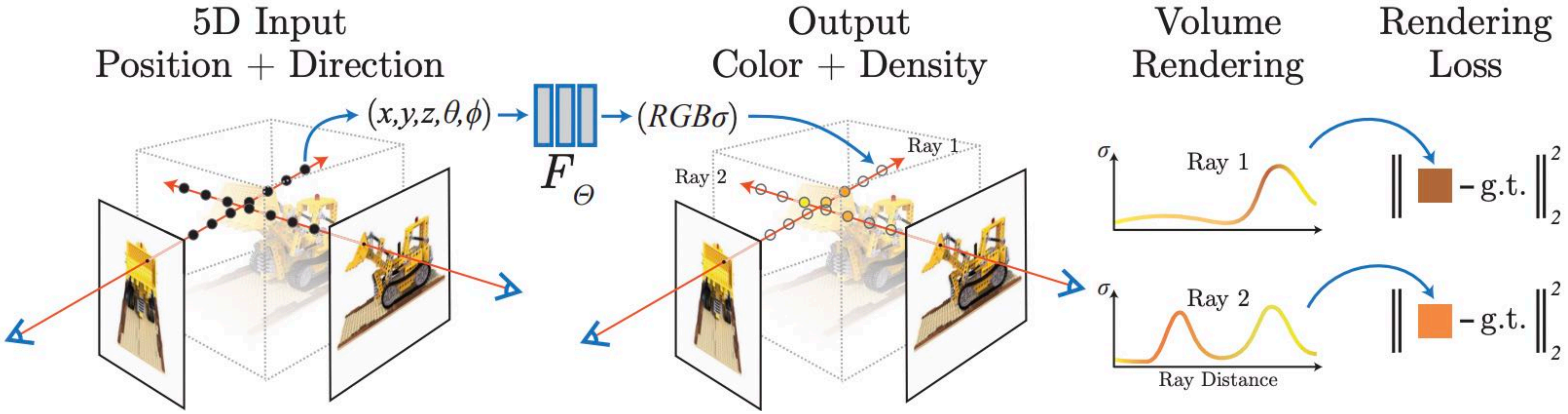
Idea: optimize volume values (opacity and color) so that C(r) matches that of photos.

For many rays…. trace through volume… see if the result matches the photo… use error to update volume opacity/color values



Learning neural radiance fields (NeRF)





Key idea: differentiable volume renderer to compute dC/d(color)d(opacity)

Great visual results!

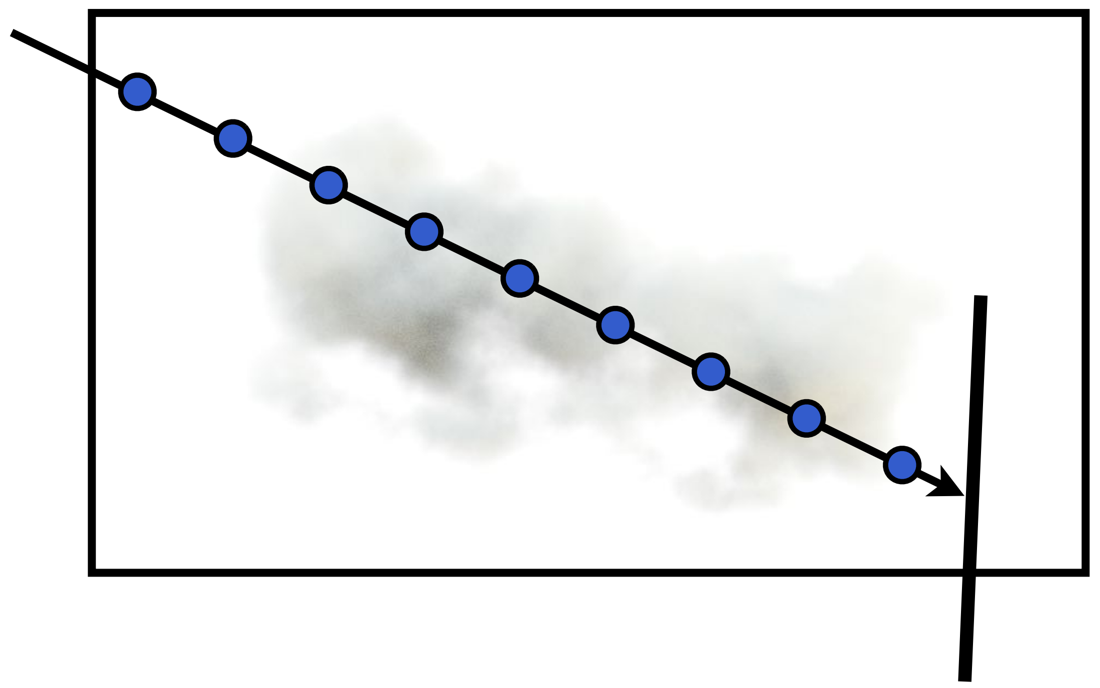
Credit: Mildenhall 2023

What just happened?

- Continuous coordinate-based representation vs regular grid: DNN "learns" how to use its weights to produce high-resolution output where needed… given input data

- Compact representation: trades-off space for expensive rendering
  - Good: a few MBs = effectively very high-resolution dense grid
  - Bad: must evaluate DNN every step during ray marching
    - And the DNN is a "big" MLP (8-layer x 256)

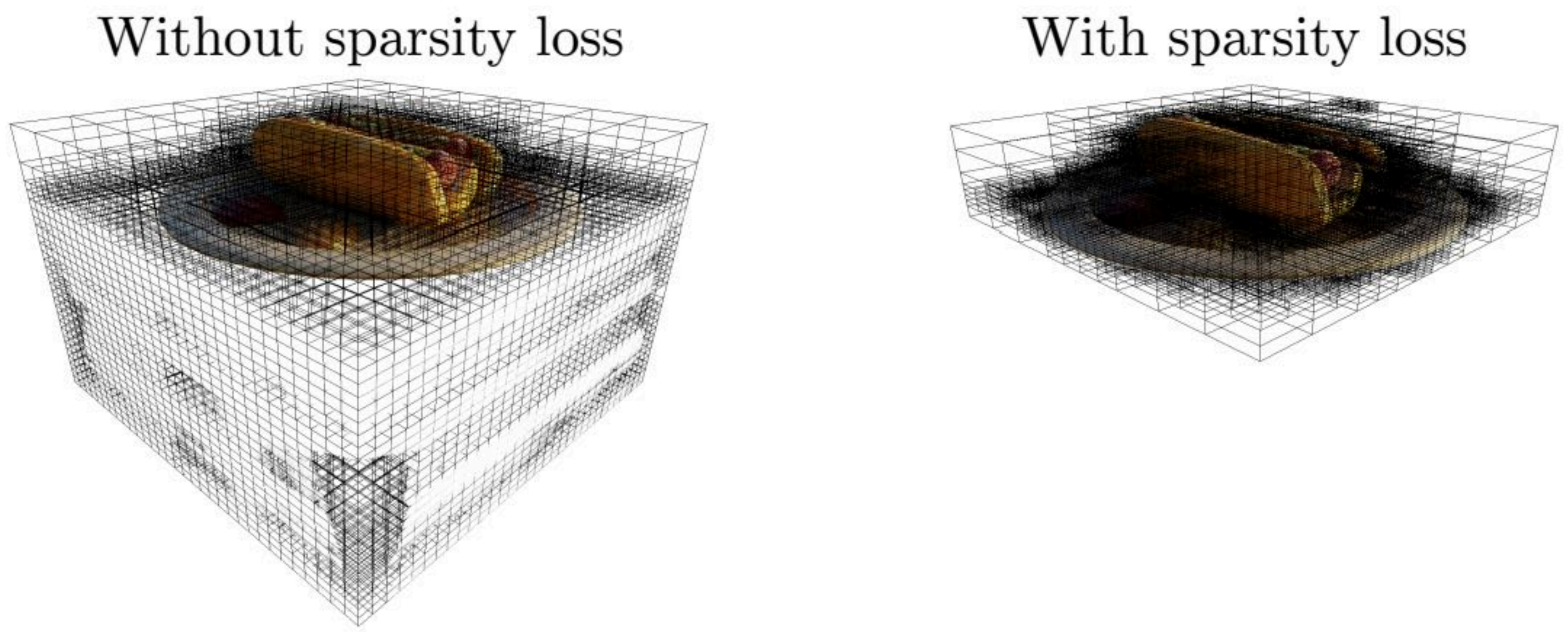
      MLP must do real work to associate weights with 5D locations
  - Bad: must step densely (because we don't know where the surface is)



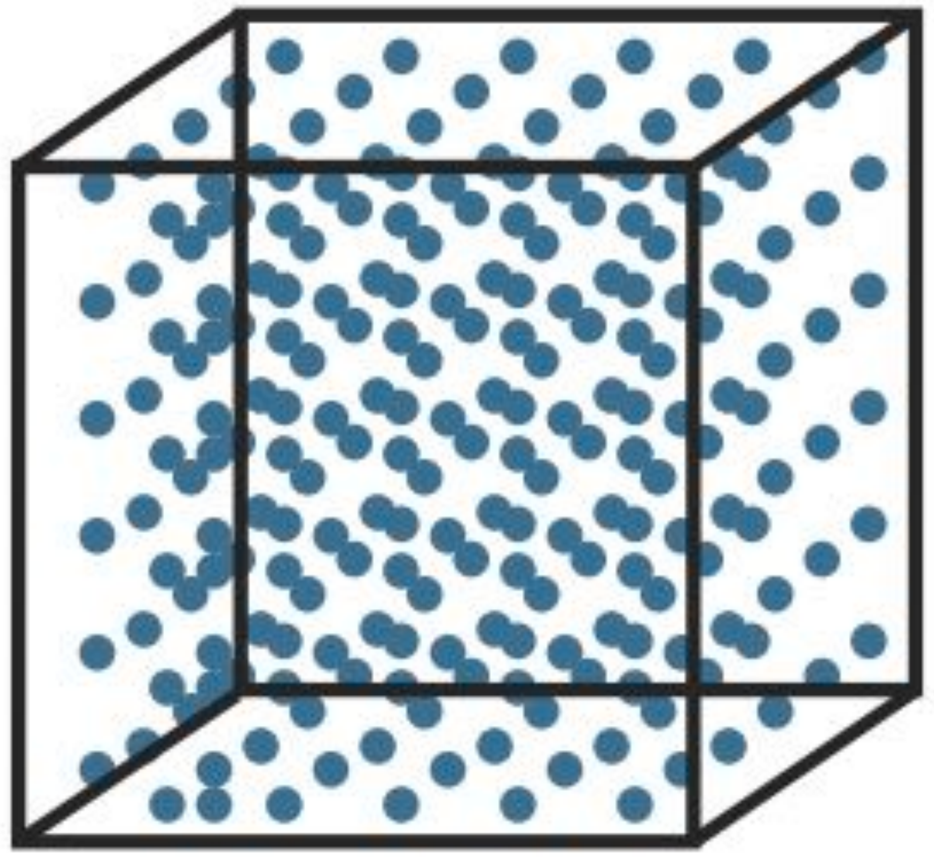
- Compact representation: optimization can learn to interpolate views despite complexity of volume density and radiance function
  - Only prior is the separation into positional σ and directional rgb
  - Training time: hours to a day to learn a good NeRF

Let's just run optimization for a bit…

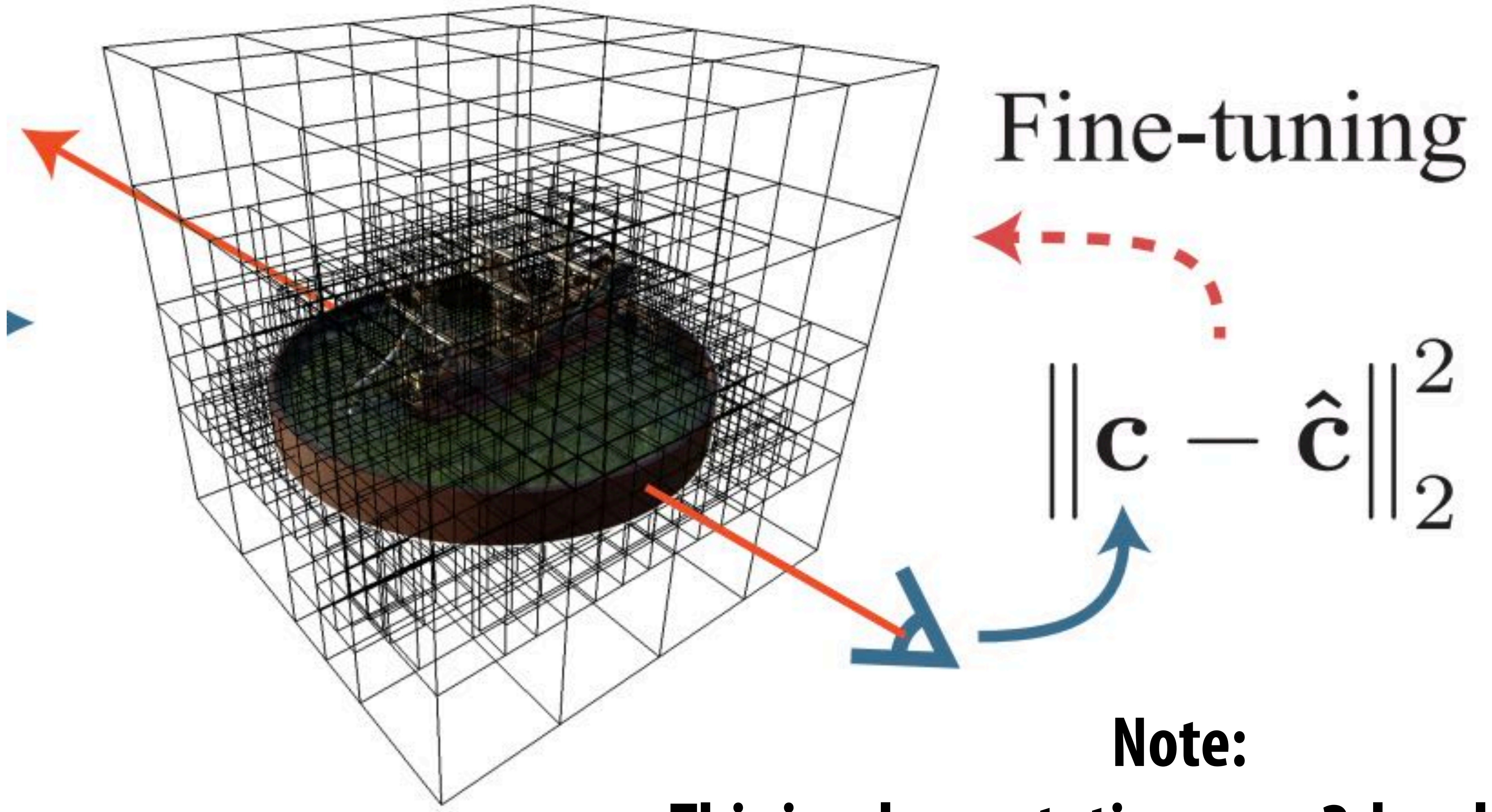
- Optimization will push some opacity values to 0
- DNN tells us where the empty space is!



- Then convert dense opacity grid to an octree representation that's more efficient to render from…
- With the octree structure *fixed*, we can continue to optimize color/density at leaves



Use the initial MLP to densely sample volume
(Find the empty space that's used to build the octree)



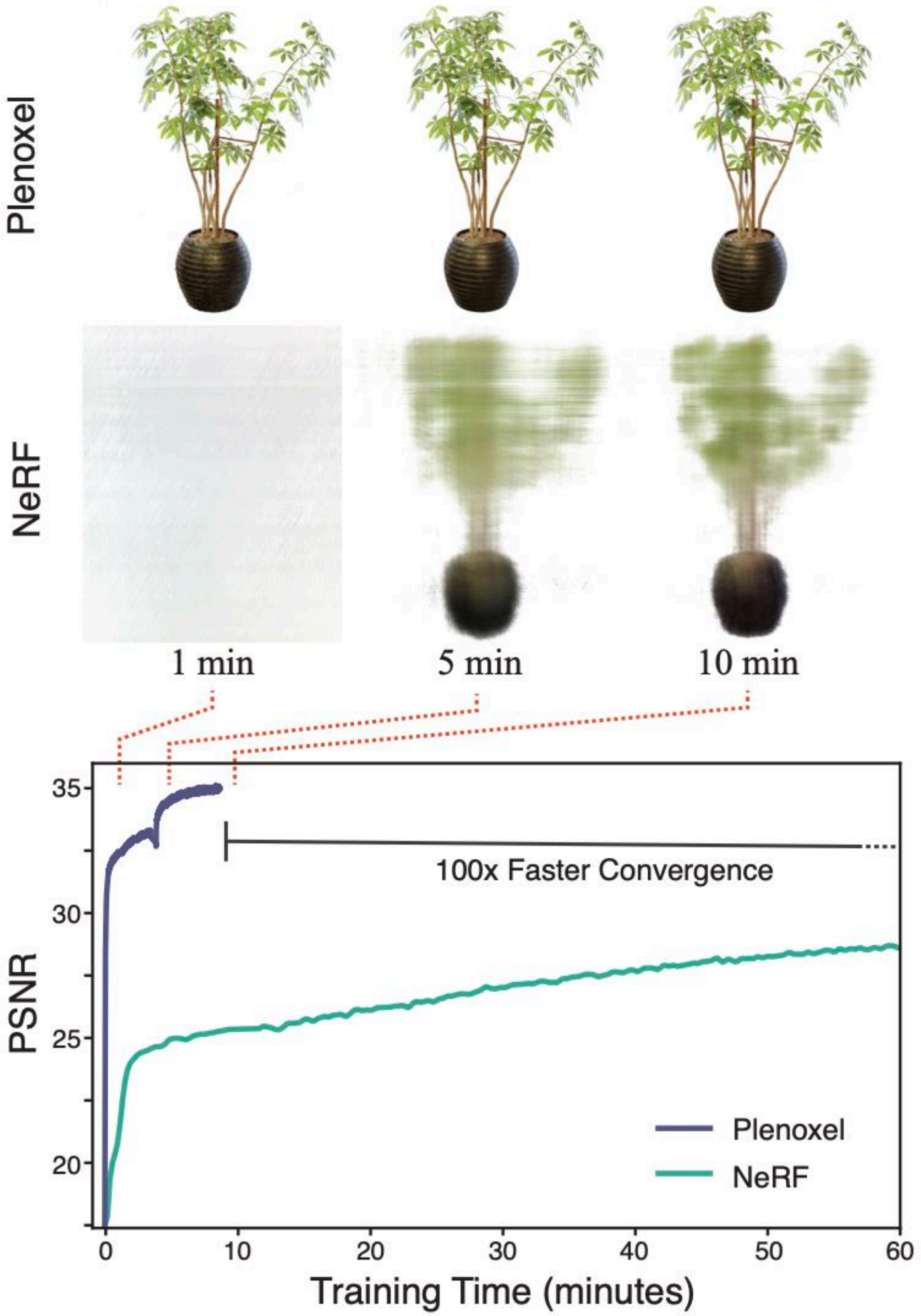
Note:
This implementation uses 2-level octreee

Finally...back to where we began

Plenoxels [CVPR 22]

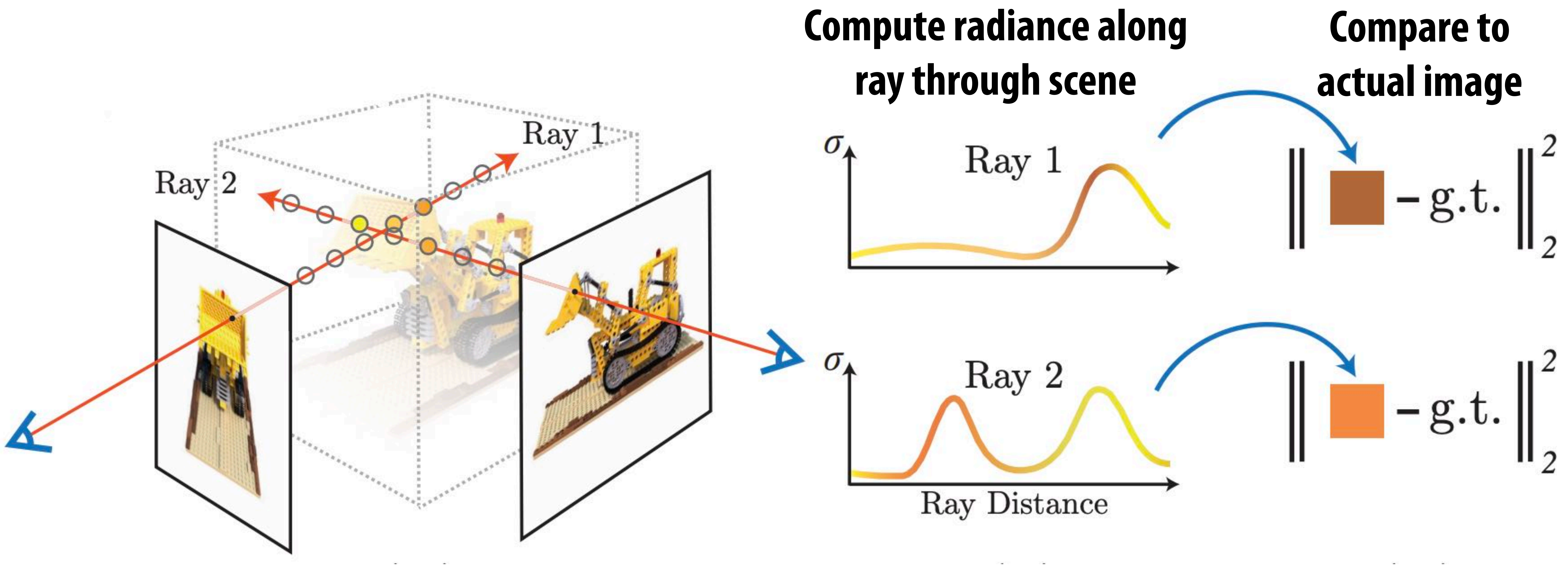
- Start with a dense 3D grid of SH coefficients, optimize those coefficients at low resolution
- Now move to a sparse higher resolution representation (octree)
- Directly optimize for opacities and SH coefficients using differentiable volume rendering
- No neural networks. Just optimizing the octree representation of baked SH lighting

- Takeaway: conventional computer graphics representations are efficient representations to learn/optimize on



Optimization to produce Gaussians, not voxels

- Earlier in lecture: optimization produces color and opacity at each voxel
- Now: same idea, but optimization chooses color, position, and radius of the Gaussians
  - Now: also need to decide on the number of Gaussians (a bit tricker)



Key idea: differentiable Gaussian splatting rendering to compute dC/d(color)d(radius)d(location)

See "3D Gaussian Splatting for Real-Time Radiance Field Rendering" [Kerbl 2023]

Novel View Synthesis

[Kerbl 2023]

But it's hard to accurately estimate depth or geometry

Discussion: what have we learned?

- Key idea 1:“unreasonable effectiveness of large-scale optimization”
  - High-performance optimization can recover parameter values for complex parameterized models
  - Credit: Ren Ng for this perspective

- Key idea 2: Many different scene representations can be reconstructed
  - Differentiable rendering of these representations is the key technology

- There’s a huge “art" to getting optimization to work
  - I doubt I could get these things to successfully optimize without a lot of practice and learning myself!
  - If I was a early career graphics student, I’d want to become very accomplished in the “art” of getting an optimizer to work for me

- Neural representations != preferred representations: neural data compression can be a good thing
  - But techniques like Gaussian splatting, sparse voxels, and Plenoxels are strong evidence that even better compact representations are already present (and don’t require resorting to neural representations)