

**Lecture 26:**

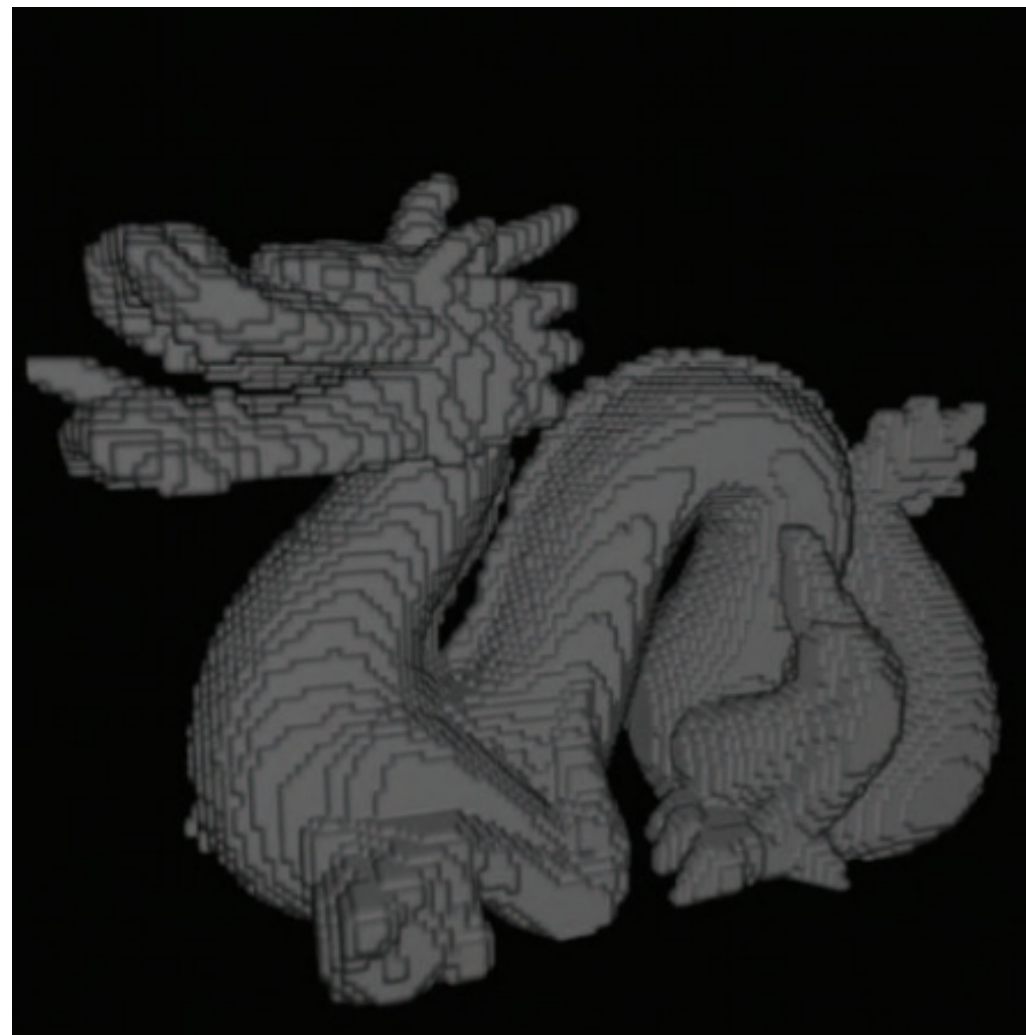
# **Real-Time Voxelization for Global Illumination**

---

**Visual Computing Systems  
CMU 15-869, Fall 2013**

# Voxelization to regular grid

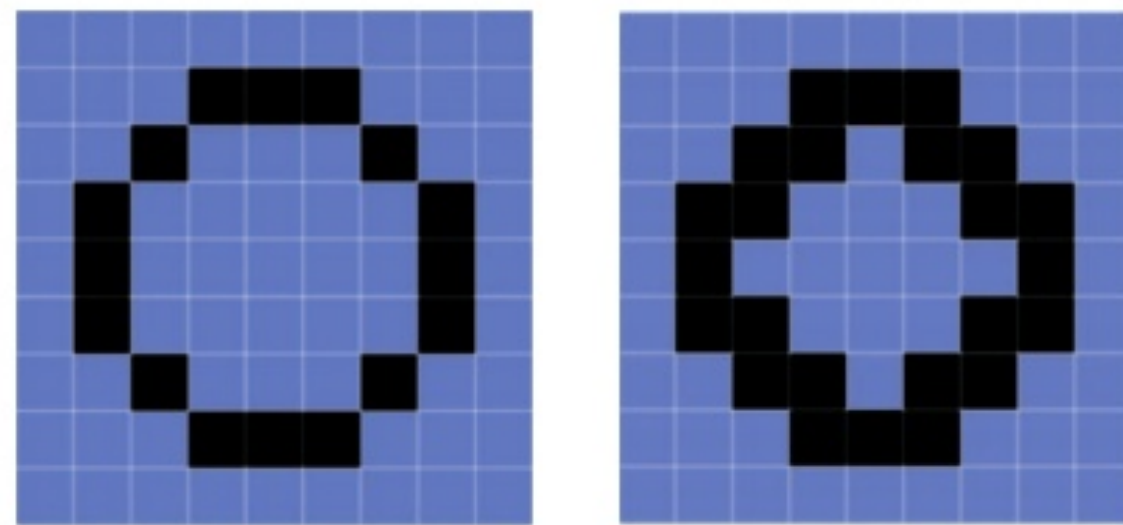
- **Input: scene triangles**
- **Output: surface information at each voxel in 3D grid**
  - **Simple case: voxel stores presence (does voxel contain geometry)**
  - **More complex case: voxel stores shading parameters (e.g., normal, albedo)**



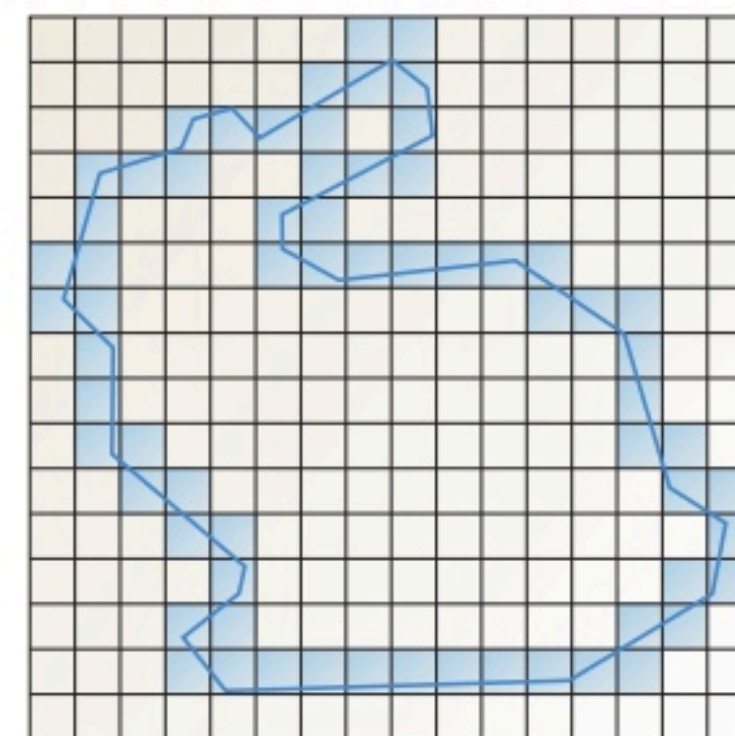
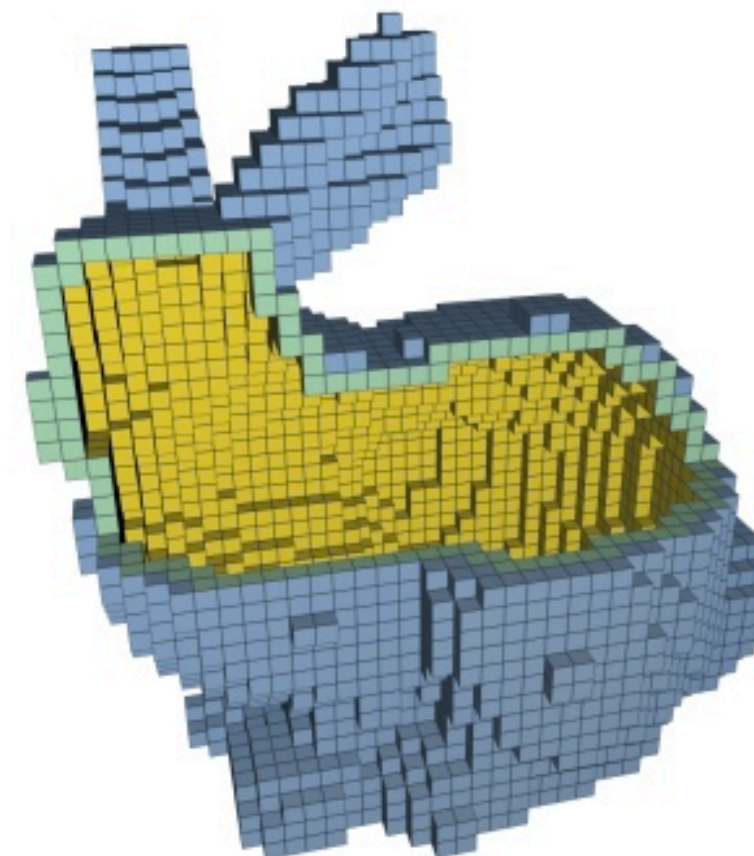
# Voxelization as conservative 3D-rasterization

- **Fragment generation:**

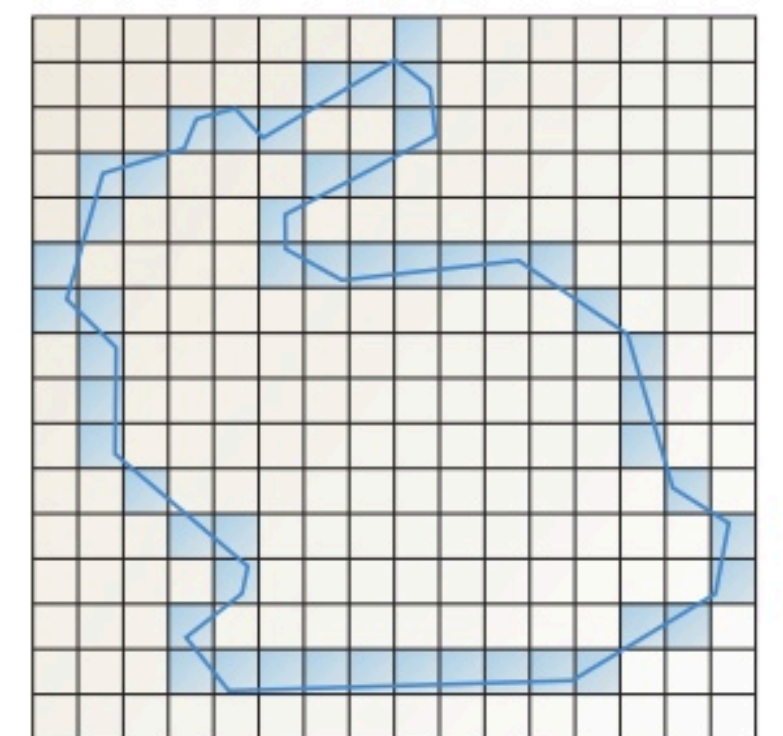
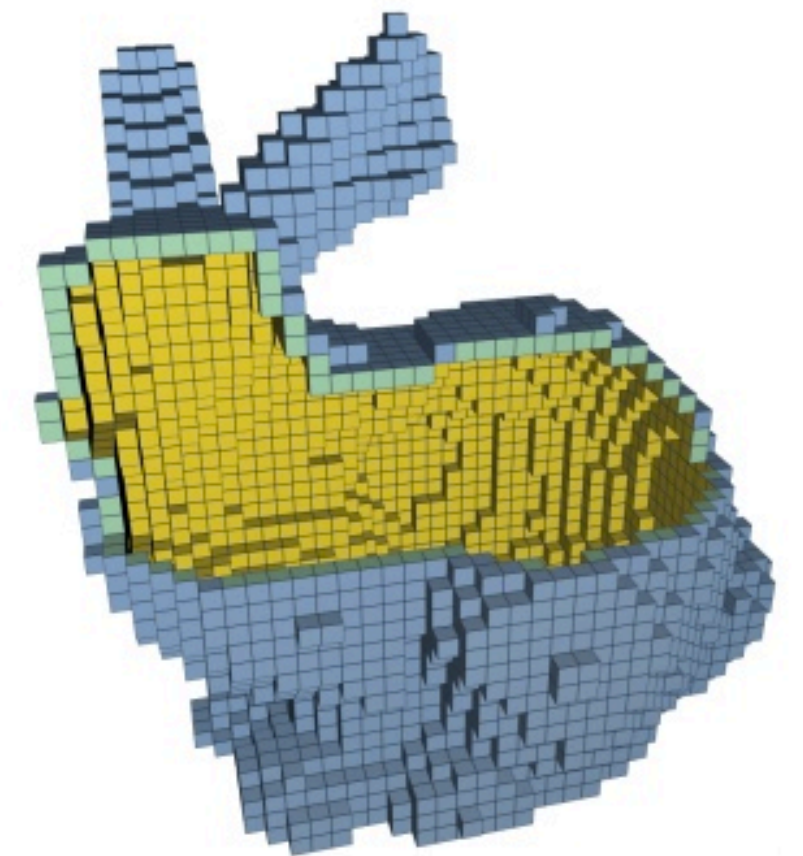
- **2D point-in-projected triangle coverage test → triangle-3D-box intersection test**



**Simpler example:**

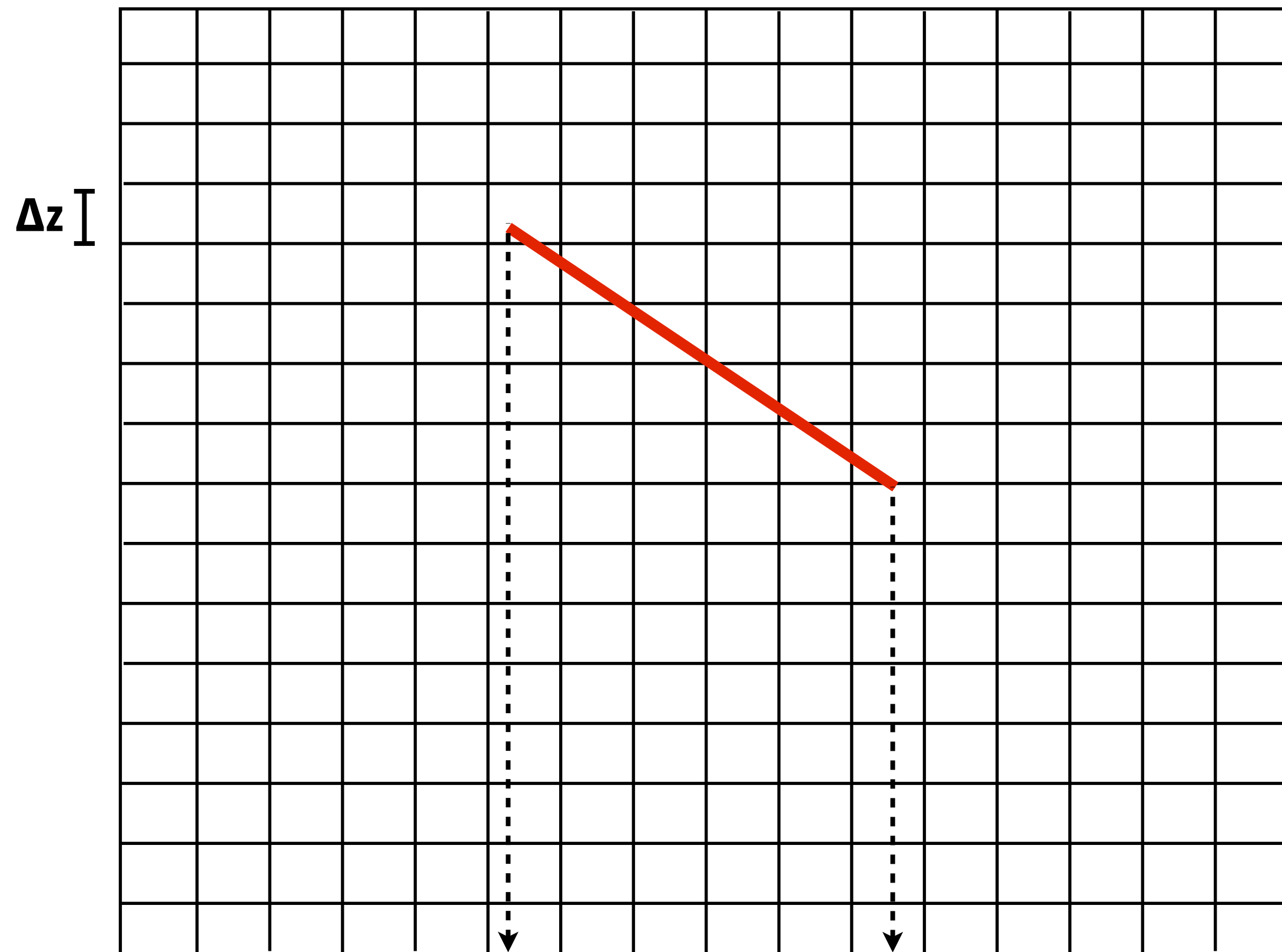


**26-separating  
(fully conservative)**



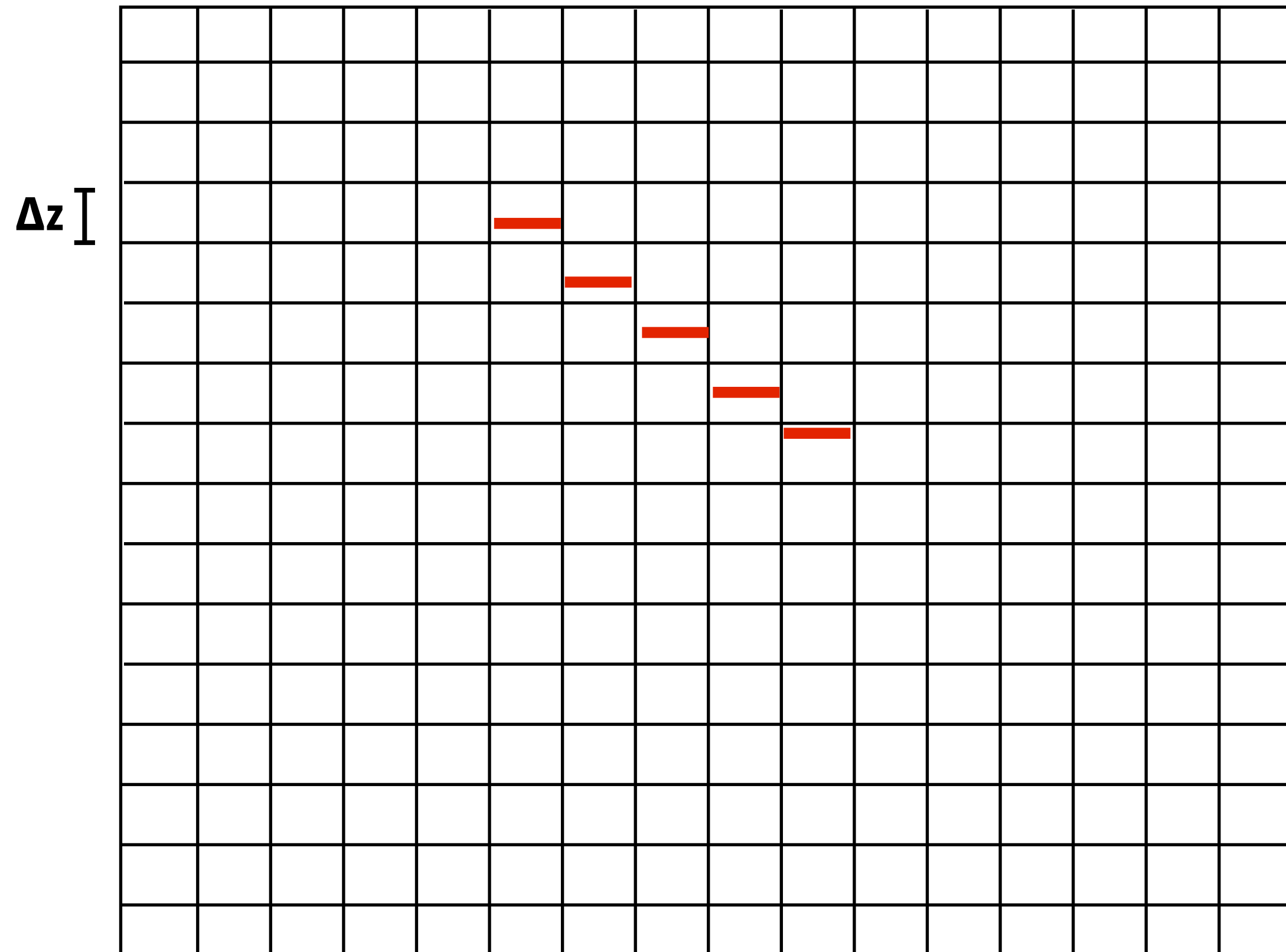
**6-separating  
("thin" voxelization)**

# Basics: voxelization using GPU rasterizer



**Rough algorithm sketch: rasterize scene with orthographic projection, and without Z-cull**

# Resulting fragments

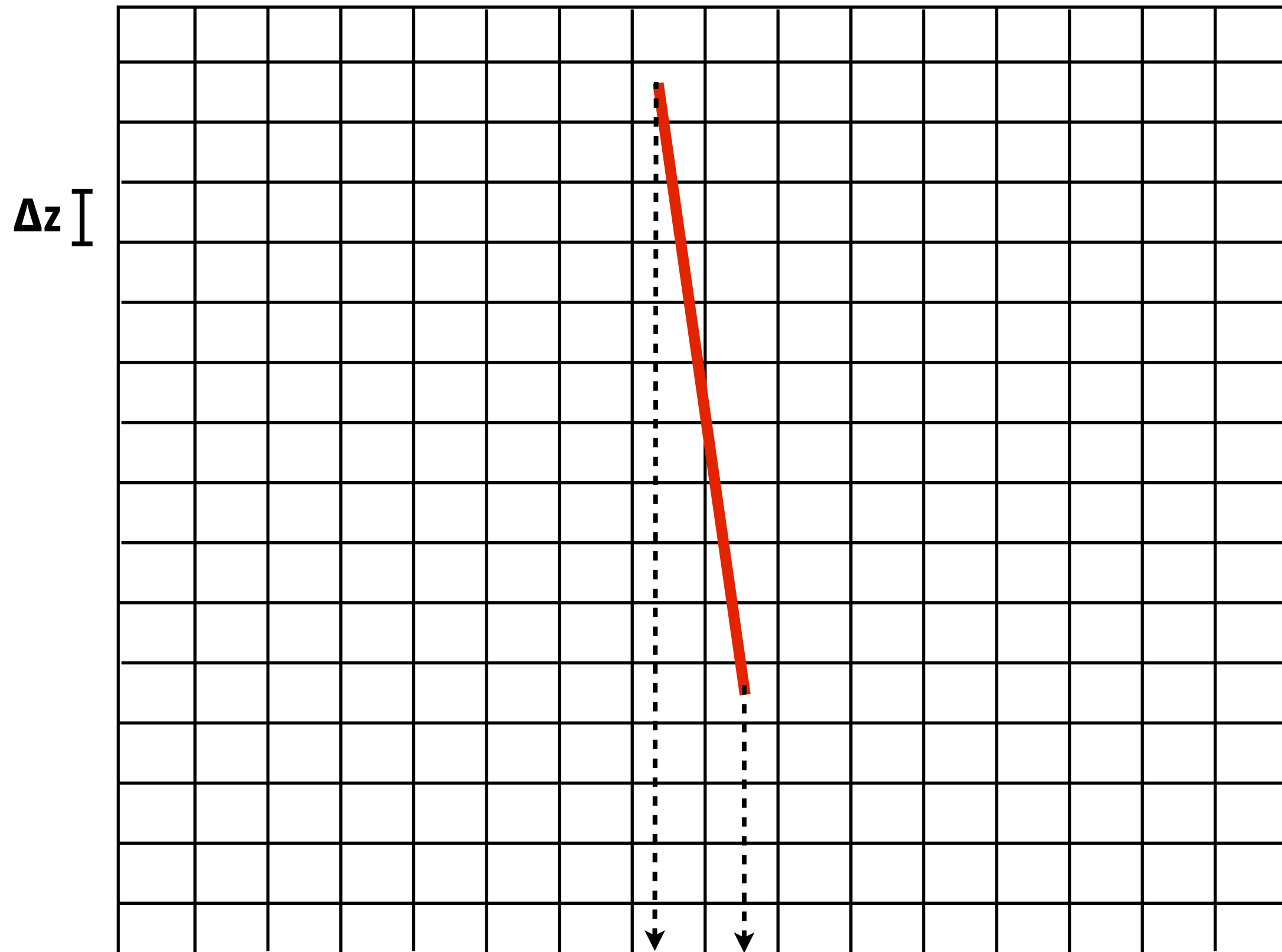


**Rough algorithm sketch: rasterize scene with orthographic projection, and without Z-cull**

**Voxel containing fragment = (frag.x, frag.y, floor(frag.z /  $\Delta z$ ));**

# Voxelization using GPU rasterizer

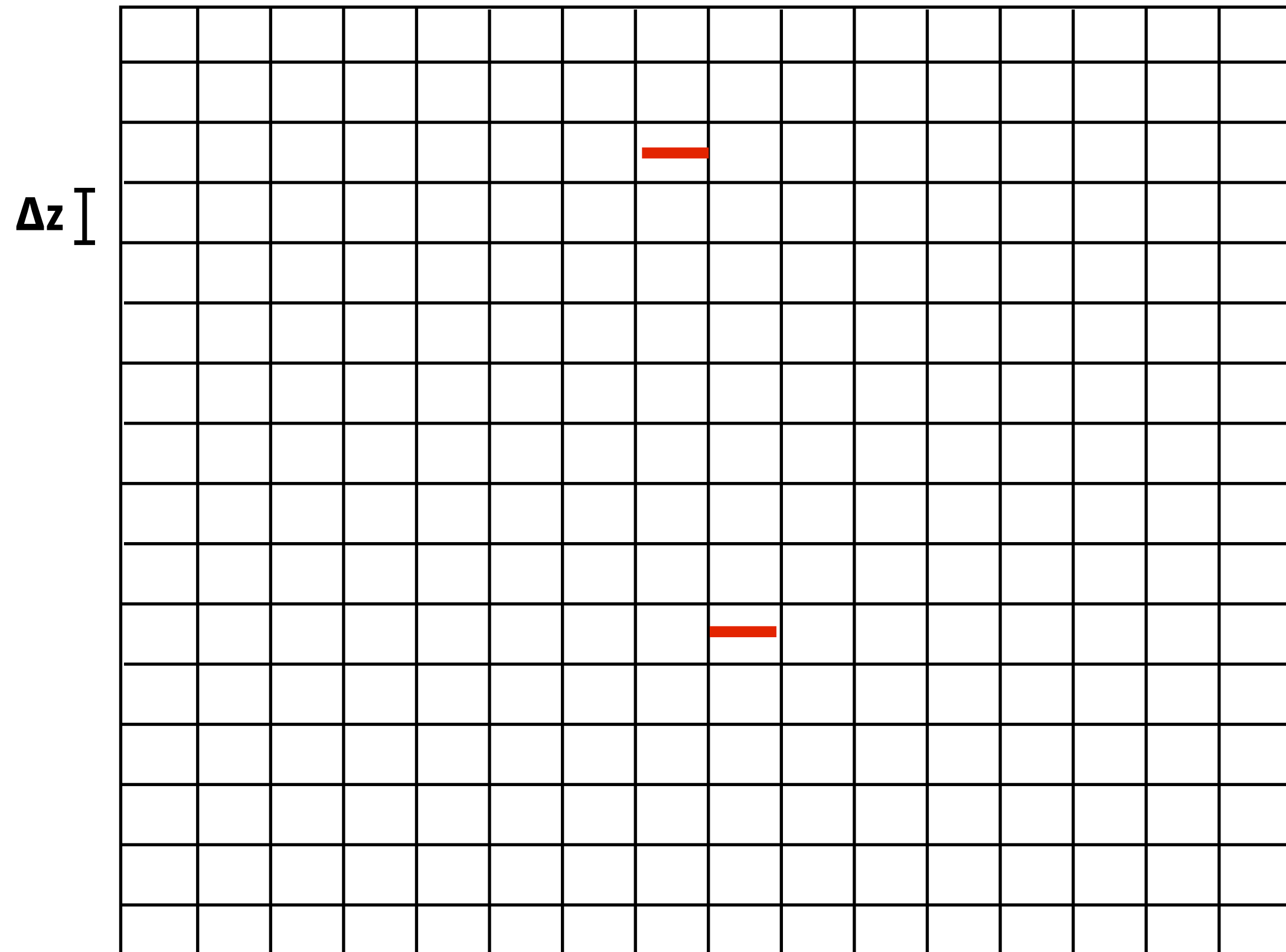
Consider case: triangle normal near-orthogonal from axis of projection



# Resulting fragments

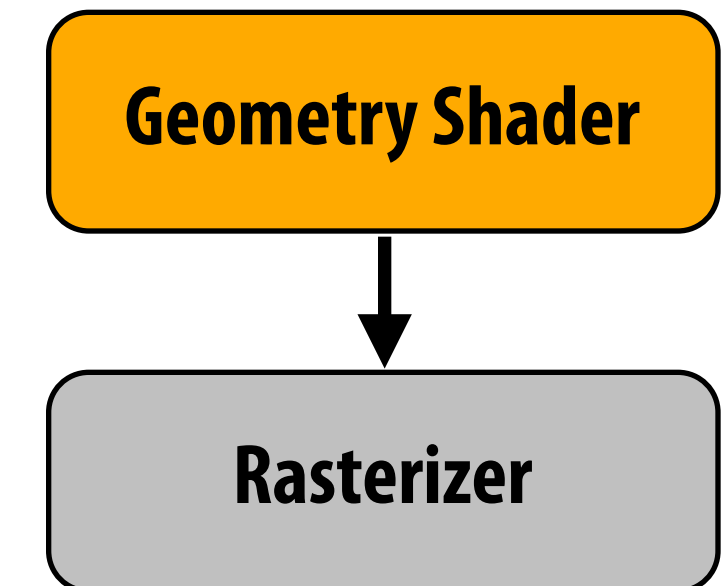
Rasterization generates one fragment per triangle per screen pixel

Problem: discontinuous voxelized representation of surface



# Dominant axis determination

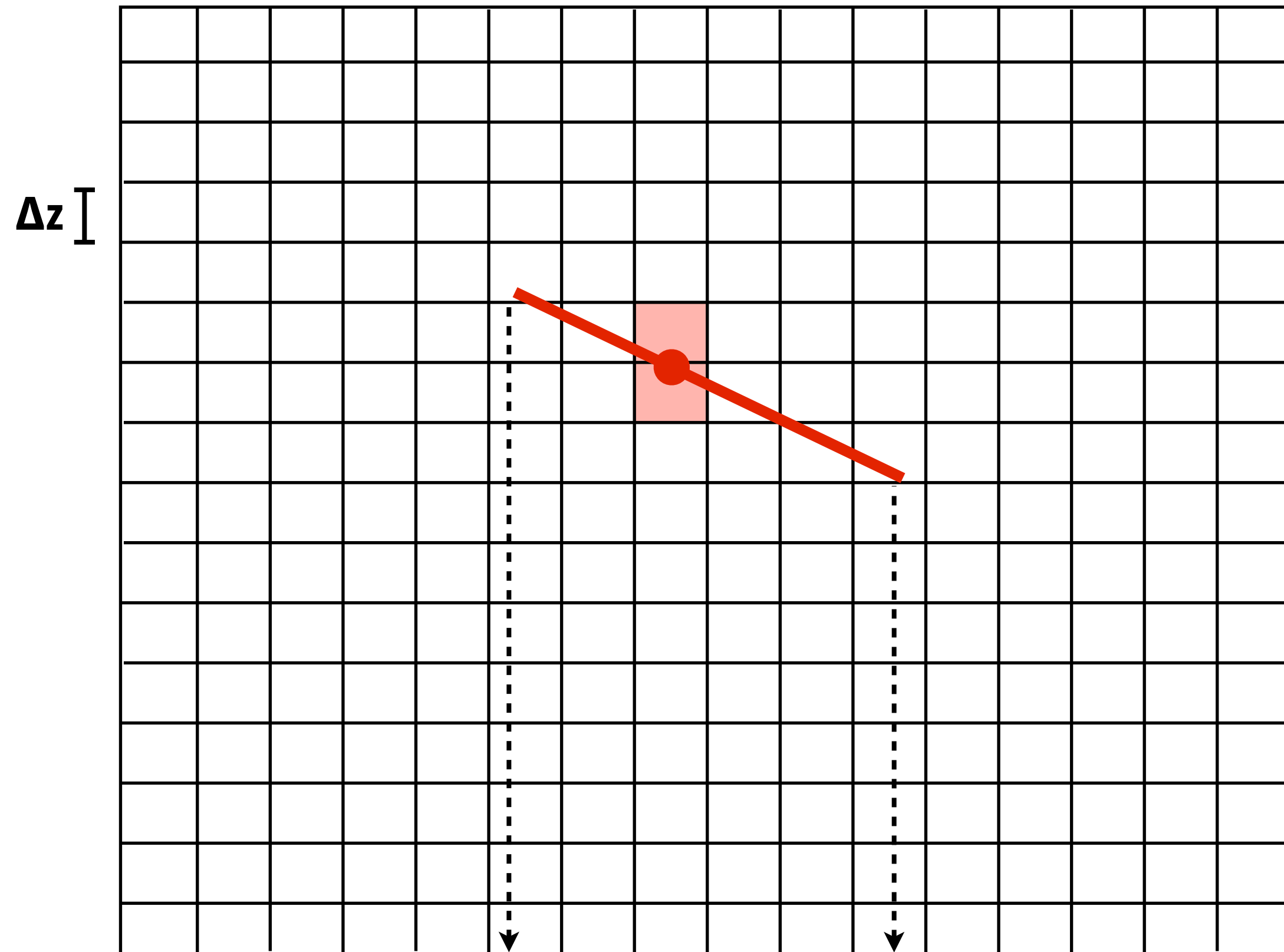
- **Single pass geometry shader implementation:**
  - **Use dot product of triangle's normal and each of three potential axes to choose dominant axis (per triangle operation)**
  - **Project triangle orthogonally along chosen axis (emit projected vertices to rasterizer)**





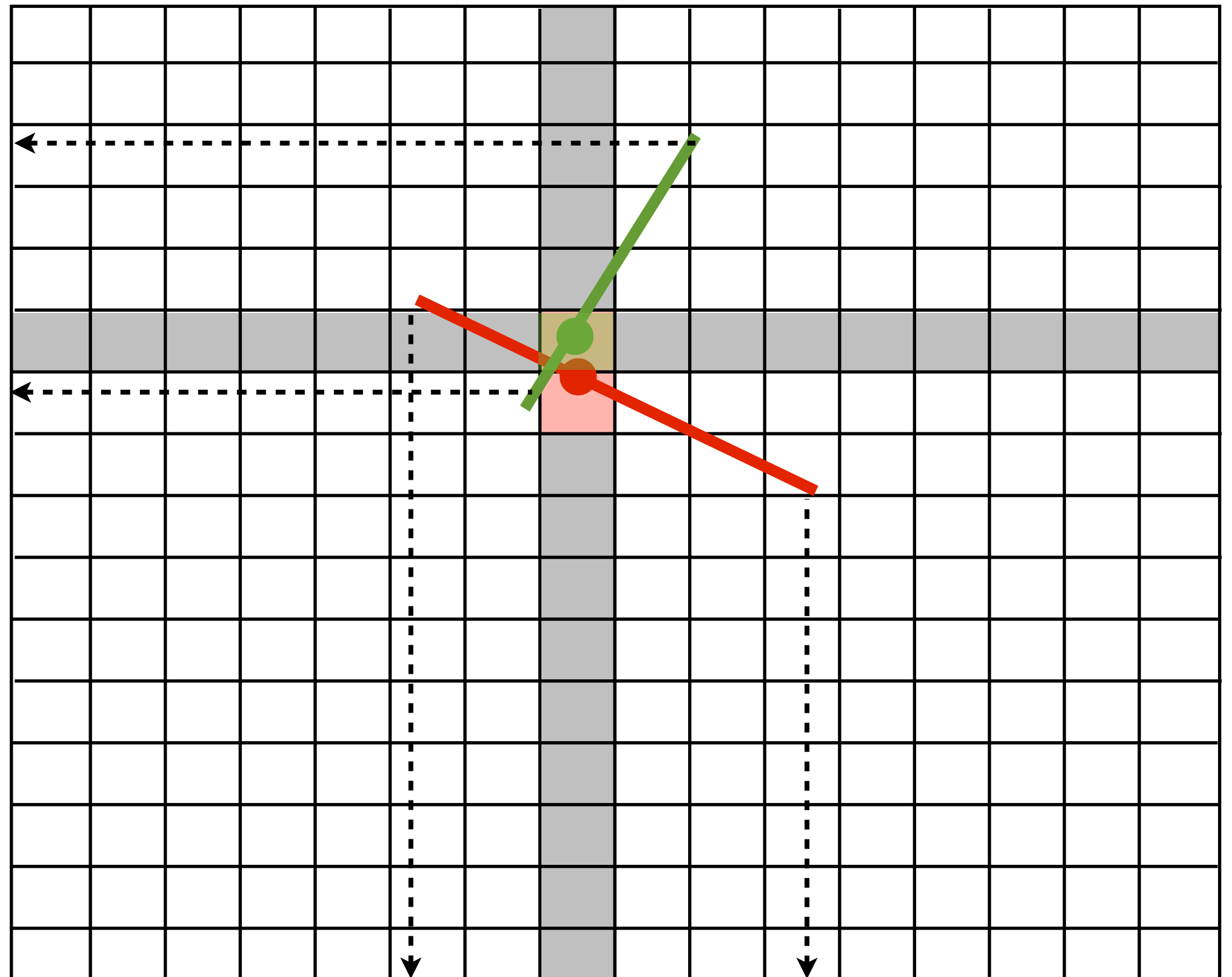
# Fragment may overlap up to multiple voxels

Compute voxel overlap given  $Z(x,y)$  (depth at pixel center),  $dZdx$ ,  $dZdy$



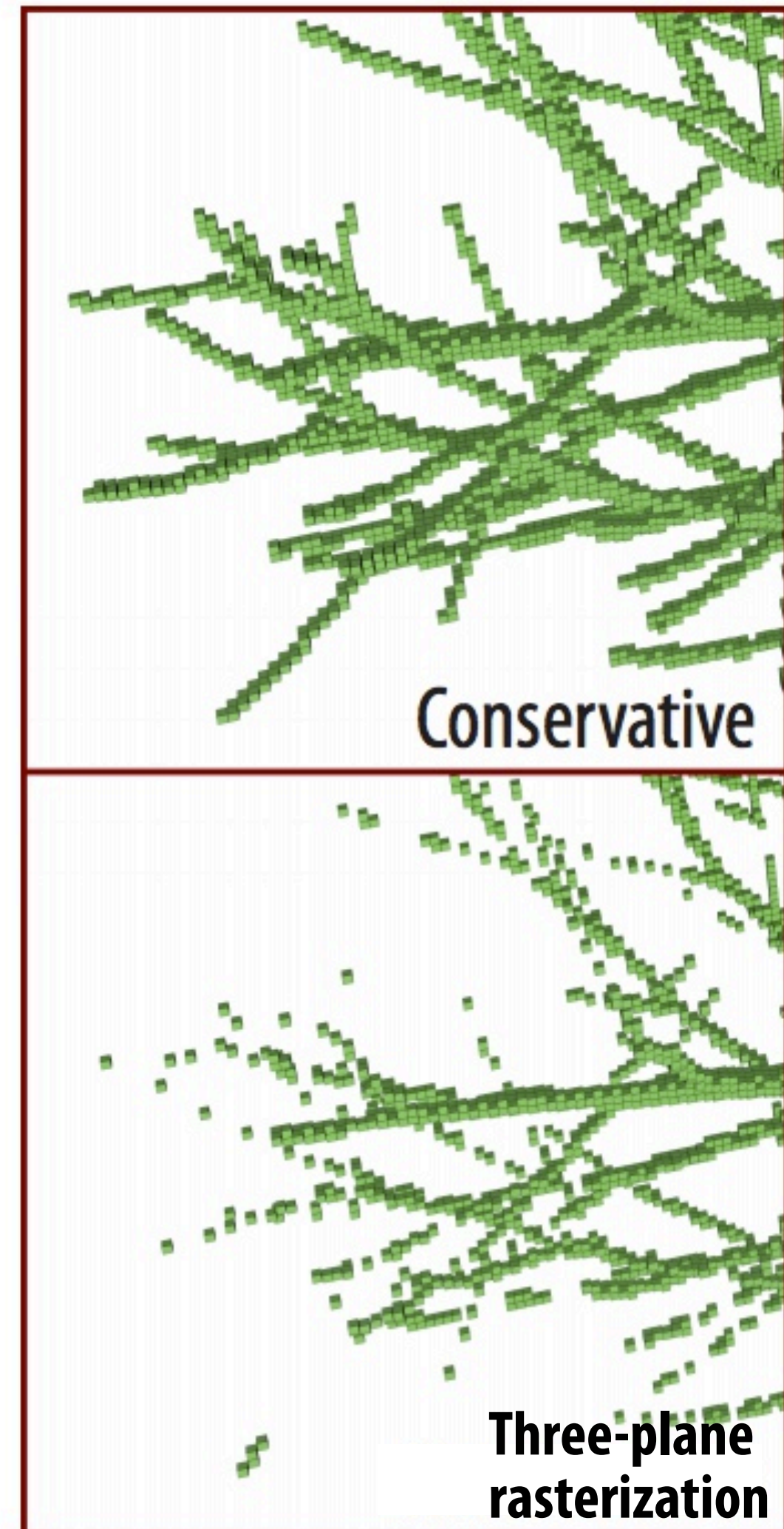
# Writing to 3D voxel data structure

- Fragment must update voxel data structure with surface color, normal, etc. information
  - frag.xy + axis of projection identifies column of voxels
  - fragment depth identifies overlapped voxels
- **Multiple fragments will overlap same voxels**
- Arbitrary memory access:
  - Use new GLSL Shader Model 5 image load/store operations or load/store to “buffer textures”
- Synchronization:
  - Ensure atomicity of updates using `compare_and_swap`
- Many-to-one reduction:
  - Average surface attributes over all fragments contributing to each voxel
  - Occupancy: OR
  - Albedo: average
  - Normal?



# Problem

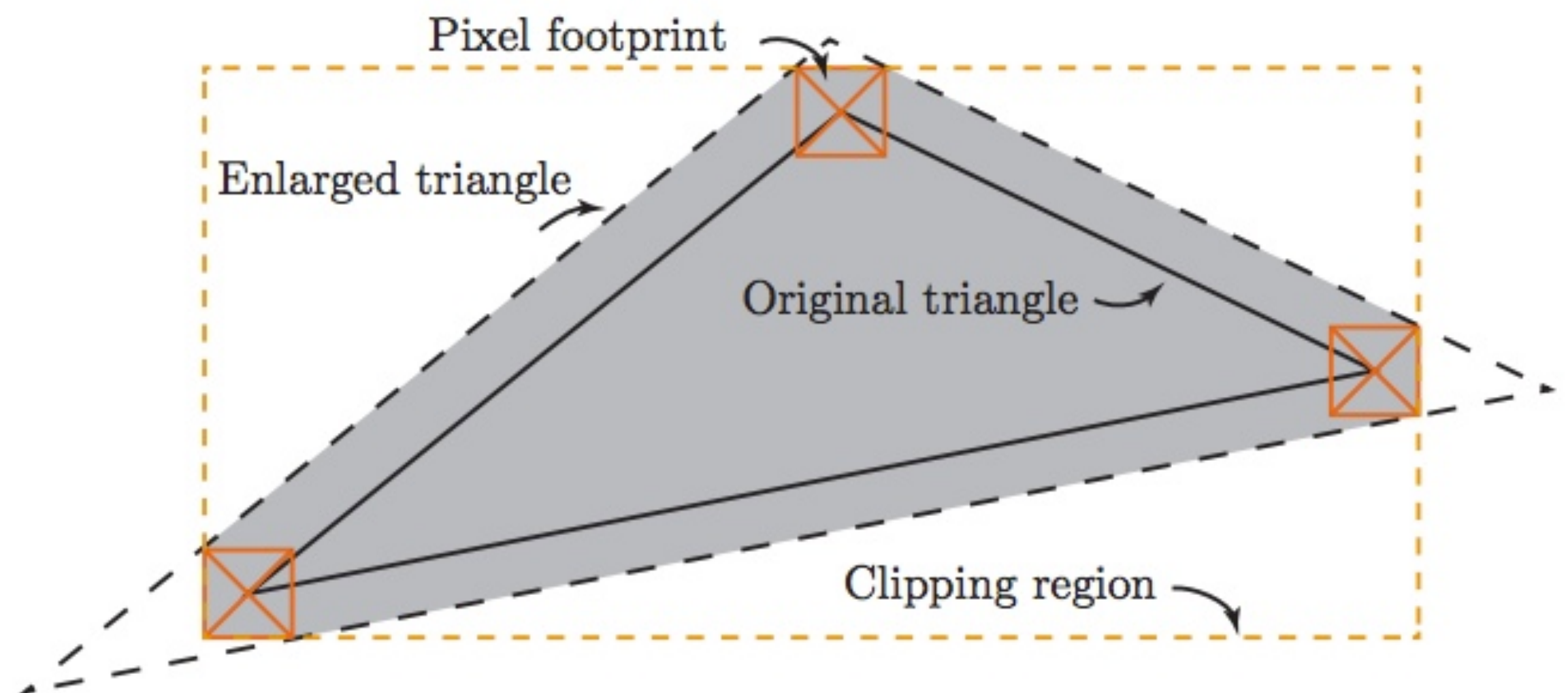
- Rasterizer only point-samples coverage: if projected triangle does not cover pixel center, no fragments will be generated
- Particularly pronounced with thin geometry



[Schwarz and Sidel 2010]

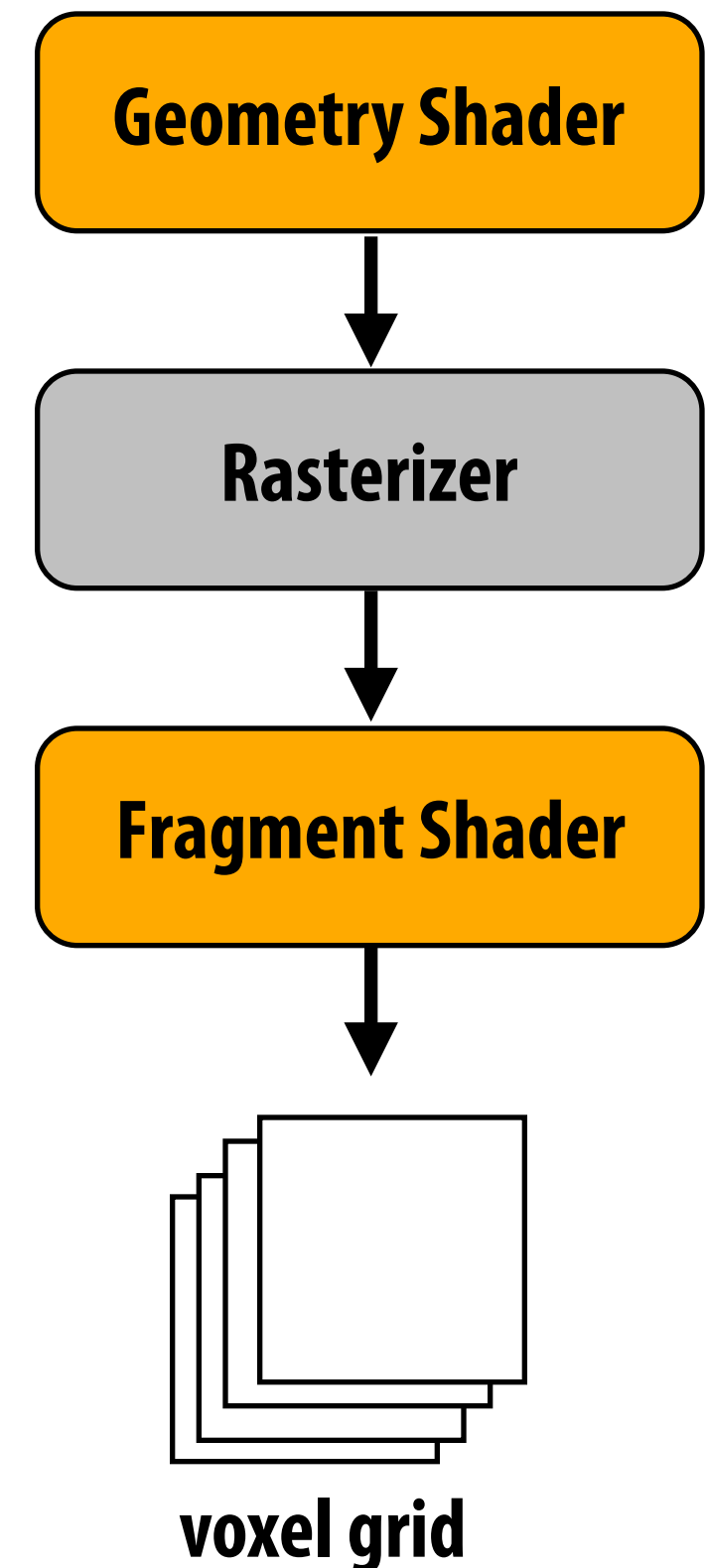
# Conservative rasterization by enlarging triangle

- **Current GPU hardware solution:**
  - **Generate bounding polygon in geometry shader**
  - **Extend bounds by up to one pixel**
  - **Fragment shader “kills” fragments that lie outside clipping region**
- **“Conservative rasterization” to accelerate voxelization is an attractive feature for future hardware rasterizers**



# Basic GPU rasterizer-accelerated voxelization

- **Single-pass geometry shader implementation:**
  - Use dot product of triangle's normal and each of three potential axes to choose dominant axis (per triangle operation)
  - Enlarge triangle to enable conservative voxelization
  - Project triangle along chosen axis (emit projected vertices to rasterizer)
- **Rasterizer generates fragments (performs part of triangle-voxel overlap computation)**
- **Fragment shader computes covered voxels and samples surface attributes**
- **Fragment shader updates 3D voxel grid data structure in parallel using **global synchronization constructs****
  - Note: fixed-function GL frame-buffer functionality NOT used
- **Alternative: implement voxelization entirely in software (using CUDA or compute shader)**

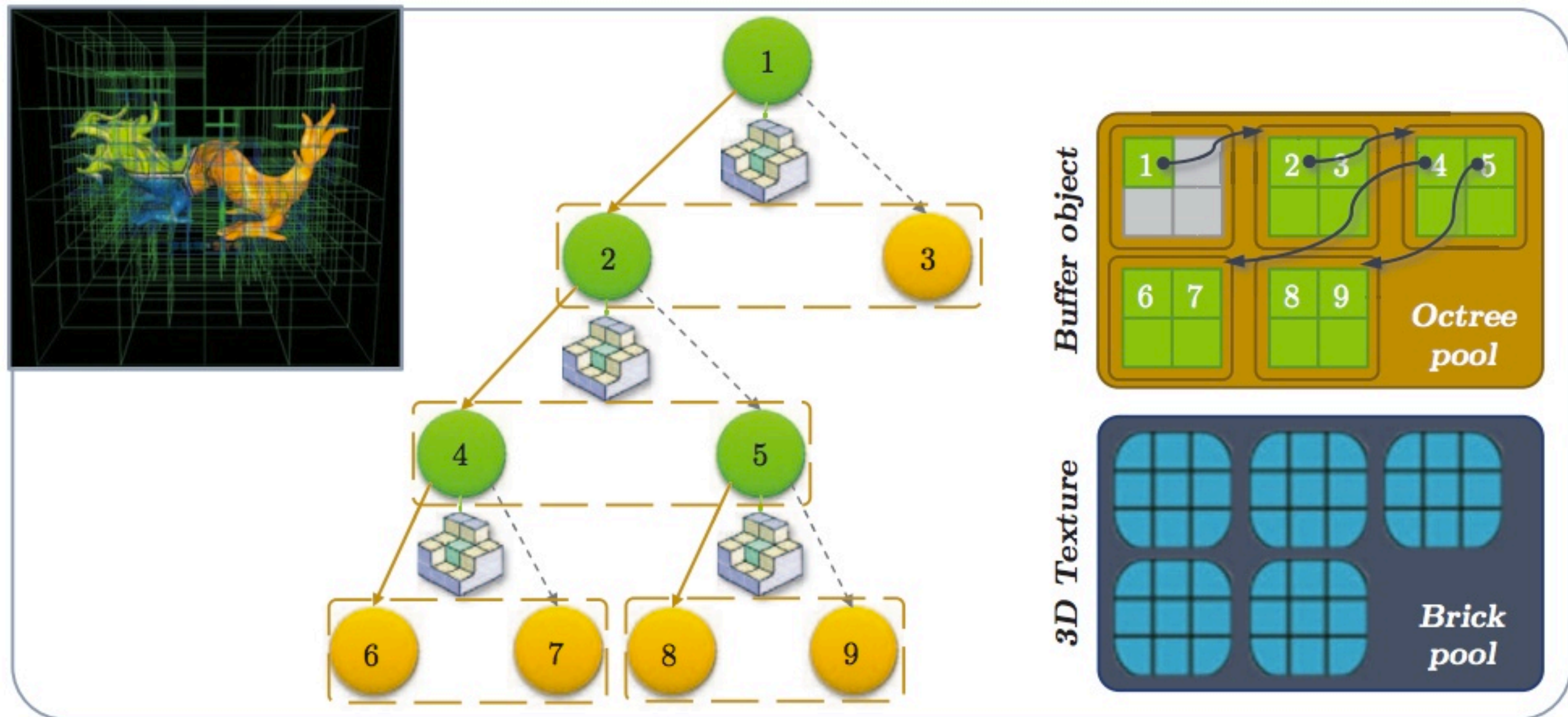


# Problem: storage cost of 3D representation

- **Modest size 512 x 512 x 512 voxel grid, 8 bytes per pixel (albedo + normal)**
  - **1 GB footprint**
- **Solutions for reducing footprint:**
  - **Compress data stored in voxels (e.g., lossy compress using low bit precision... think G-buffer packing techniques for deferred renderer)**
  - **Use sparse representation of voxel grid (only store voxels where surfaces actually exist)**

# Sparse voxel octree (SVO)

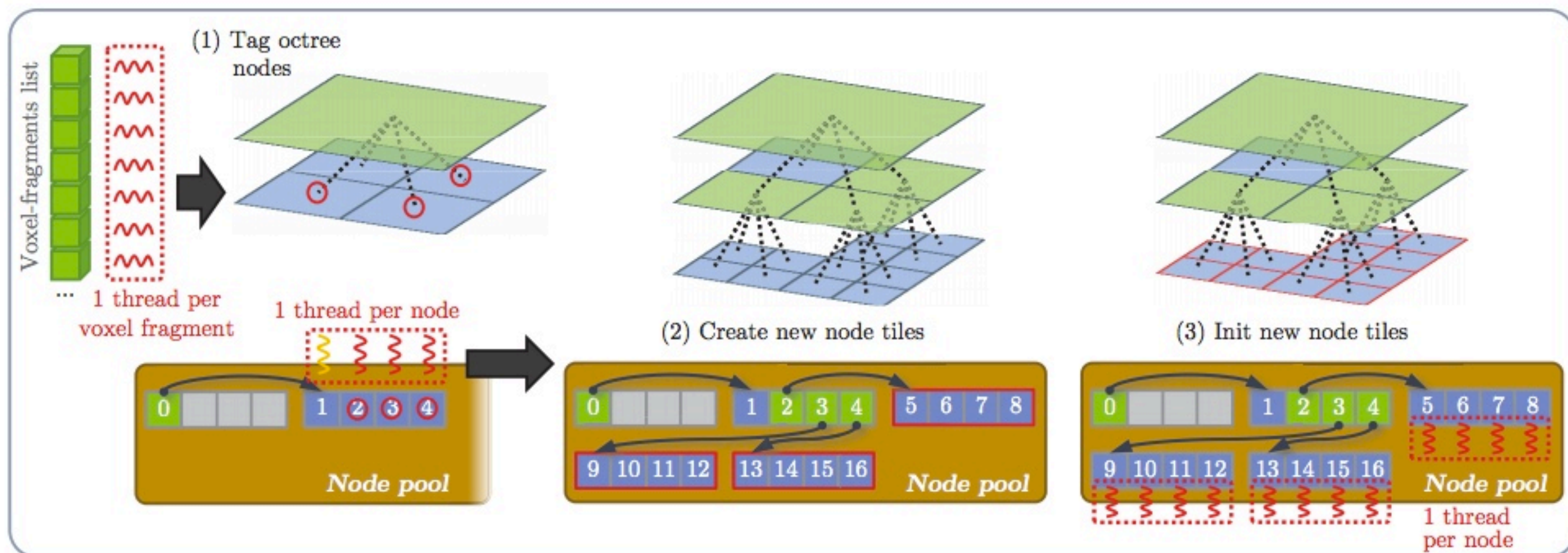
- Compact, multi-resolution voxel data-structure
- Each node corresponds to axis-aligned region in space (root node = the entire scene)
- Each node maintains pre-filtered representation of it's region of space (below: a 3x3 dense voxel grid called a "brick")
- 3D space: each node subdivided uniformly into eight child nodes (child only allocated if it exists)



Simplified illustration: Only two nodes for each interior node shown.

# Constructing a SVO on the GPU

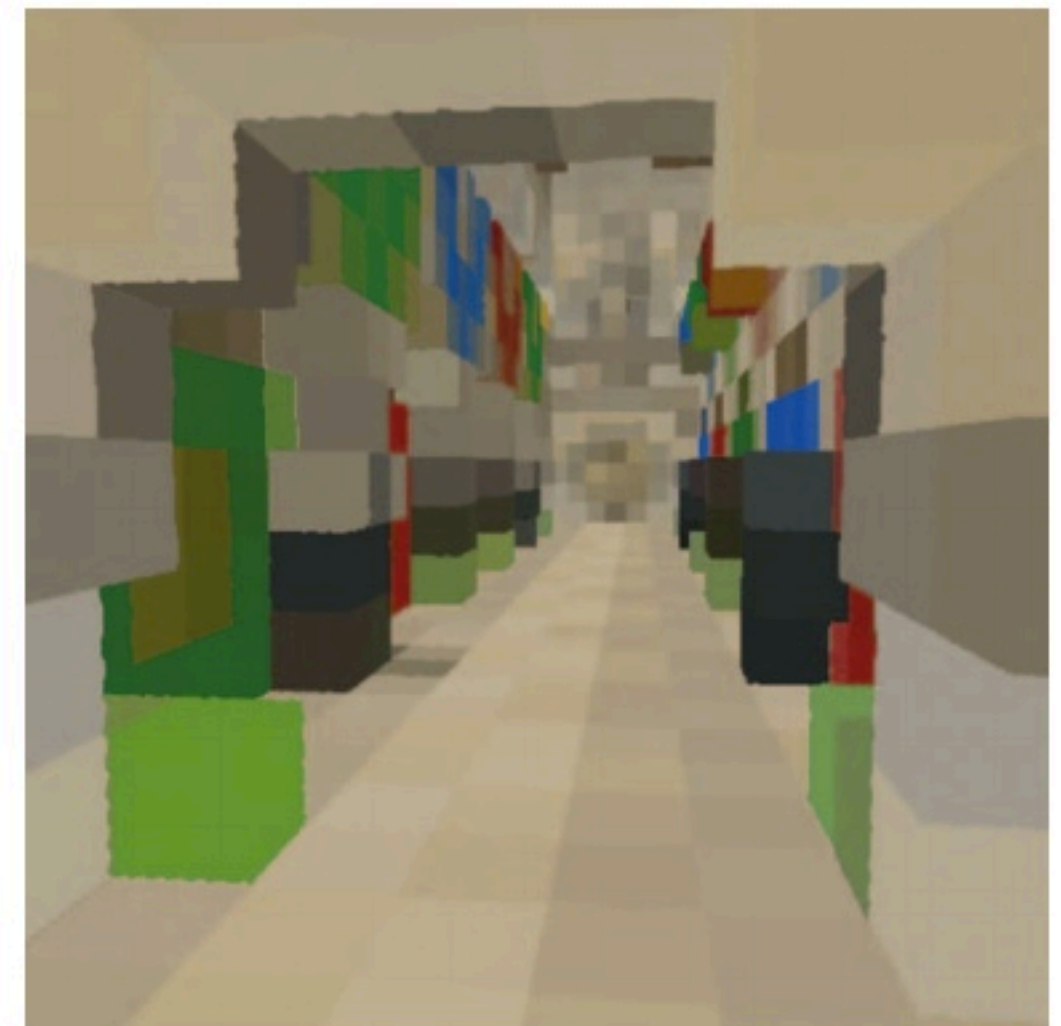
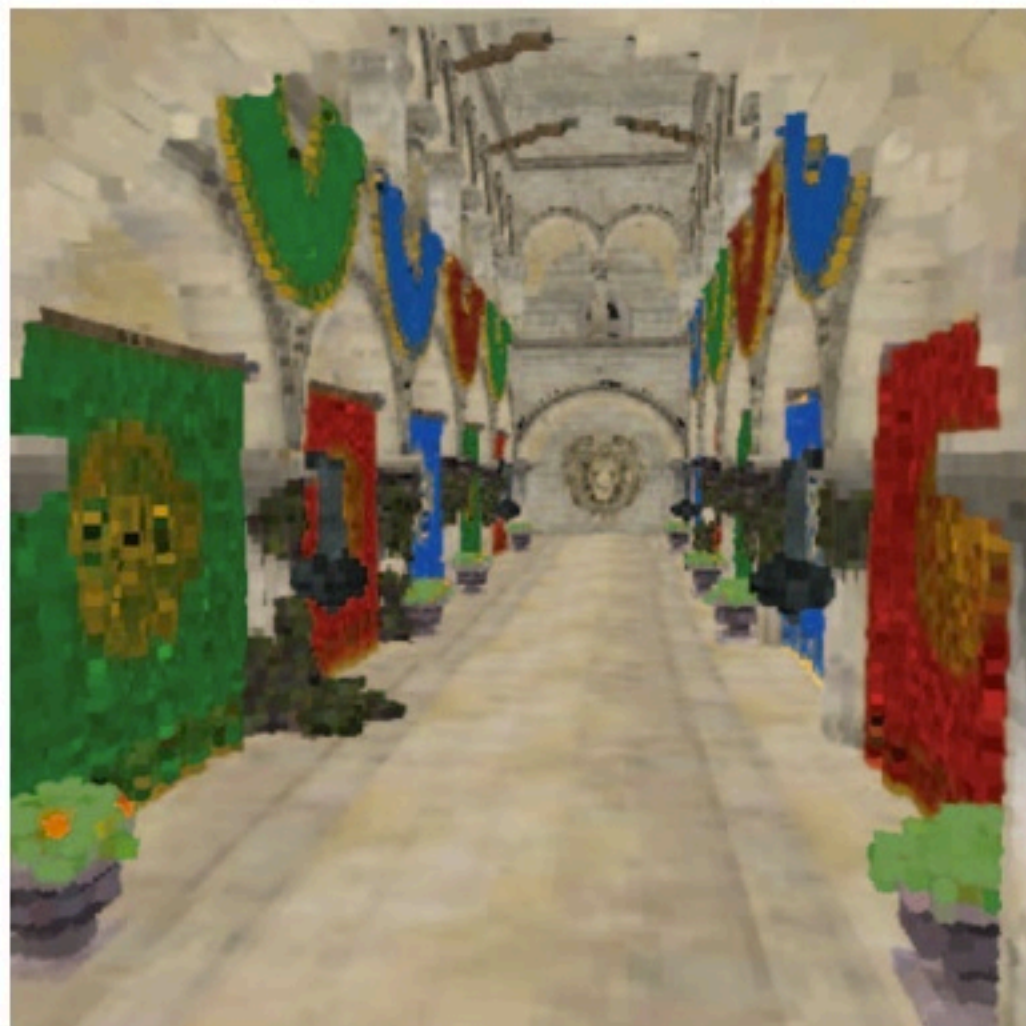
- **Step 1: build list of fragments (voxelize to finest resolution)**
  - Fragment shader appends voxel fragment to list
  - Append synchronization via atomic counter (“next” pointer)
- **Step 2: for each level of octree: from top to bottom**
  - For each fragment in list, mark child cell containing fragment as occupied
  - Allocate occupied child cells
  - Initialize child cells





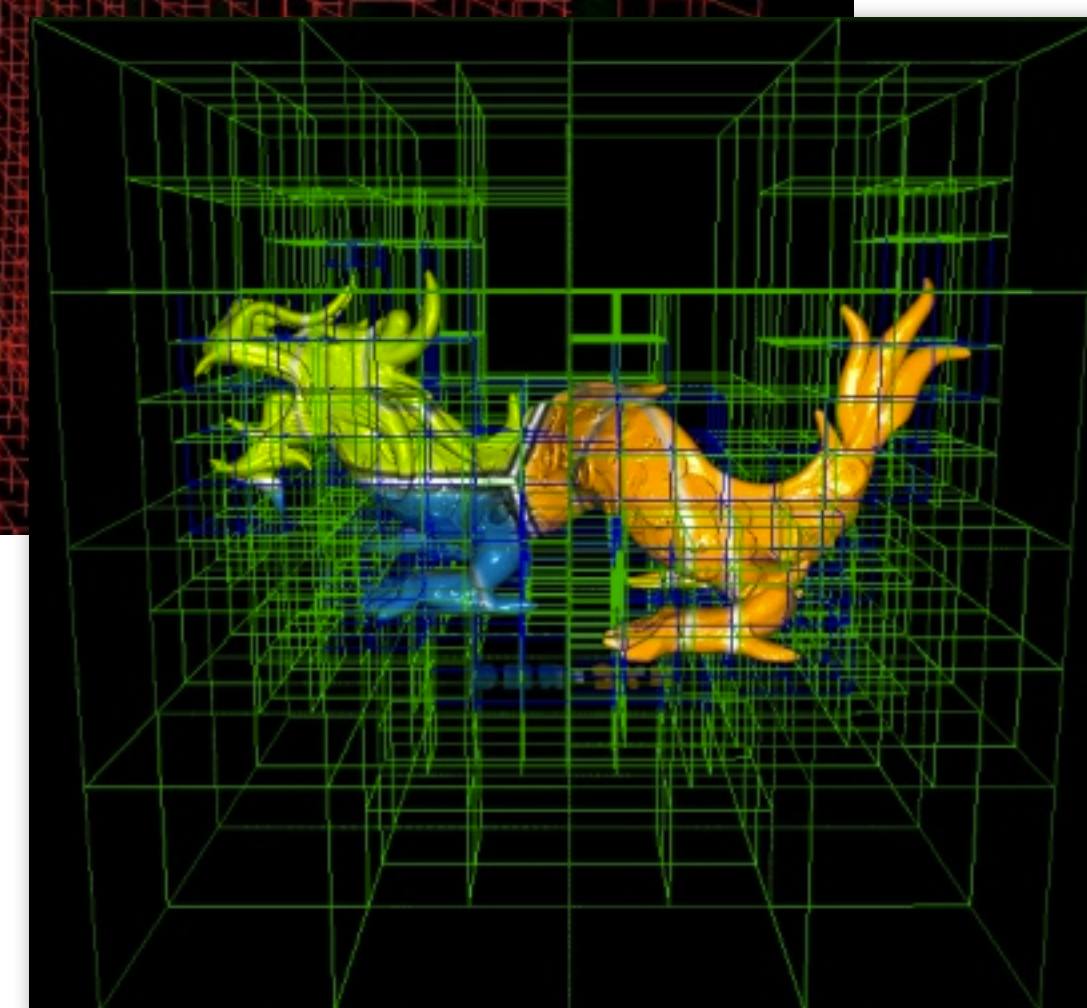
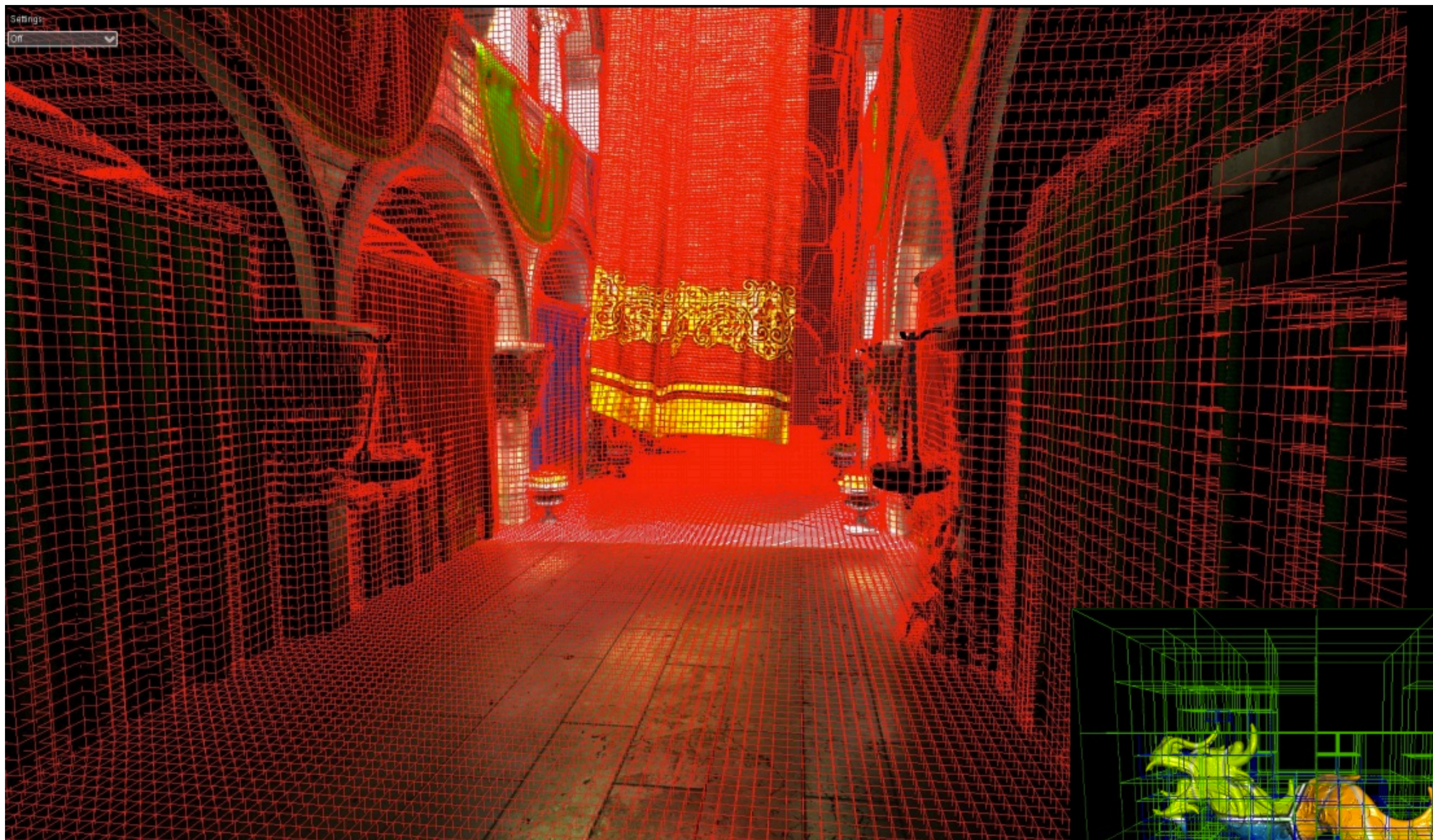
# Constructing a SVO on the GPU

- **Step 3: initialize octree values**
  - **Initialize leaf bricks (copy surface attributes from voxel fragment list)**
  - **For each interior node:**
    - **Resample values from (up to eight) children nodes to construct brick values for current node**
    - **Note: resampling of binary occupancy at leaves yields opacity**



**Visualization of surface albedo stored at three different voxel grid resolutions**

# More examples



# Ray casting through voxel grid

■ Ray:  $p(t) = o + dt$

■ Ray intersects plane perpendicular to x axis at:  $t_x(x) = \left(\frac{1}{d_x}\right)x + \left(\frac{-o_x}{d_x}\right)$

■ Let  $x_0$  be x-aligned plane of box closest to ray origin,  $x_1$  be farthest

■ Range of  $t$  values for which ray is in cube  $(x_0, y_0, z_0), (x_1, y_1, z_1)$ :

-  $t_{\min} = \max(t_x(x_0), t_y(y_0), t_z(z_0))$

-  $t_{\max} = \min(t_x(x_1), t_y(y_1), t_z(z_1))$

■ Advancing to next box at same same level of grid:

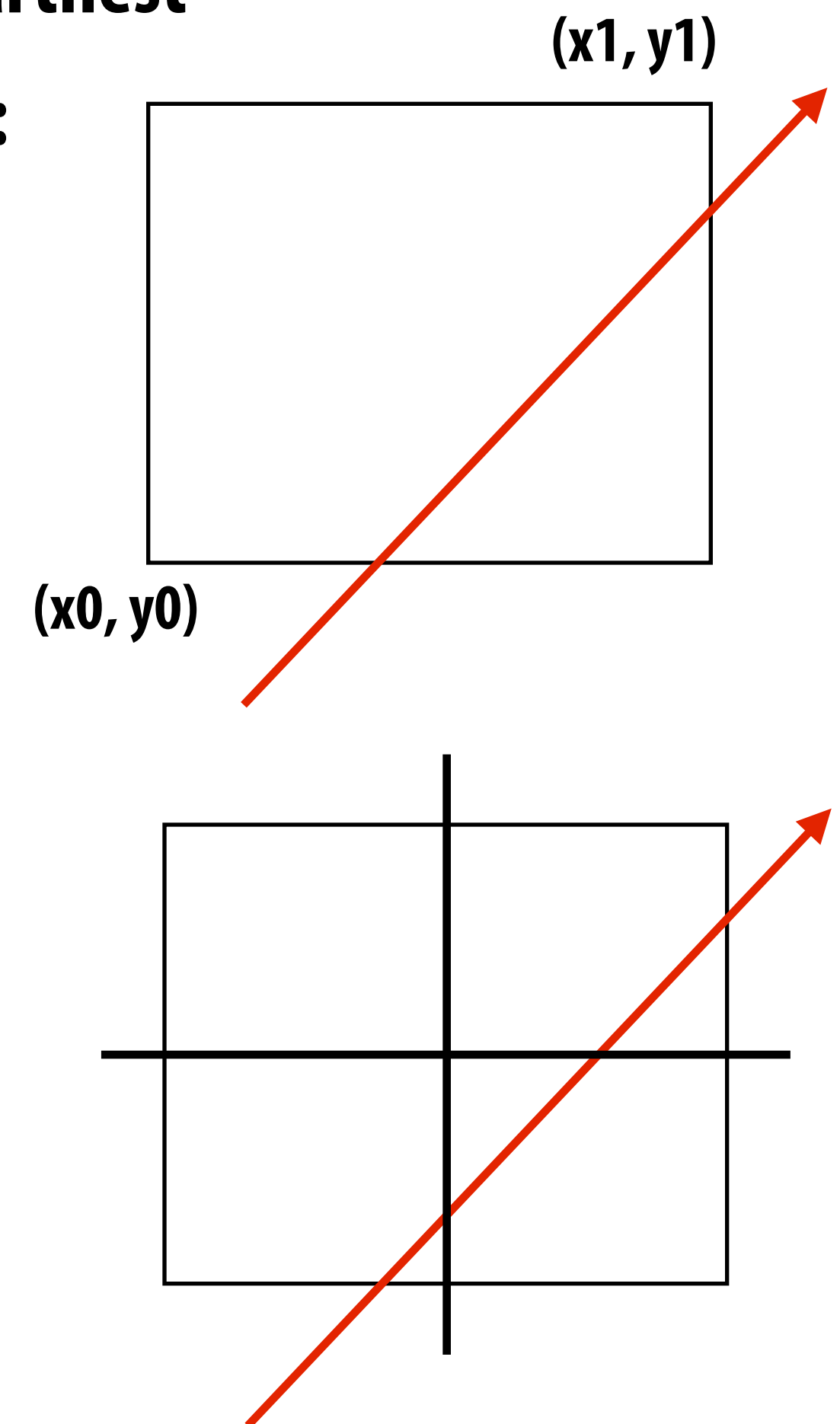
- Compare  $t_x(x_1), t_y(y_1), t_z(z_1)$  with  $t_{\max}$ , advance cell index in all dimensions value is equal

- In octree: just flip bits of index for each dimension!

- Incompatibility between bit flip direction and ray direction indices out of octree node (must pop up a level)

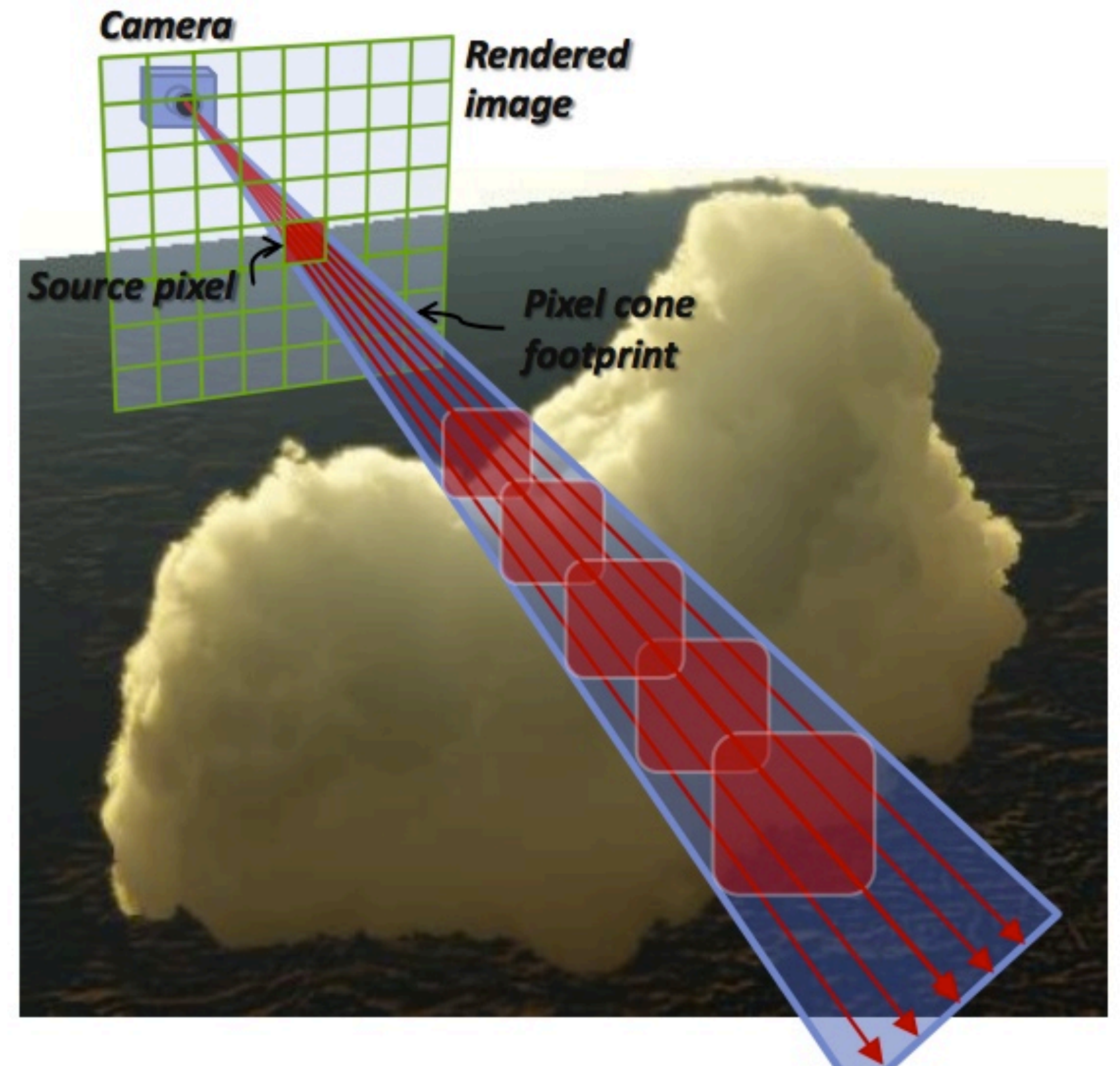
■ Descending to lower levels:

— Determine starting voxel by checking  $t_{\min}$  against node midpoint planes

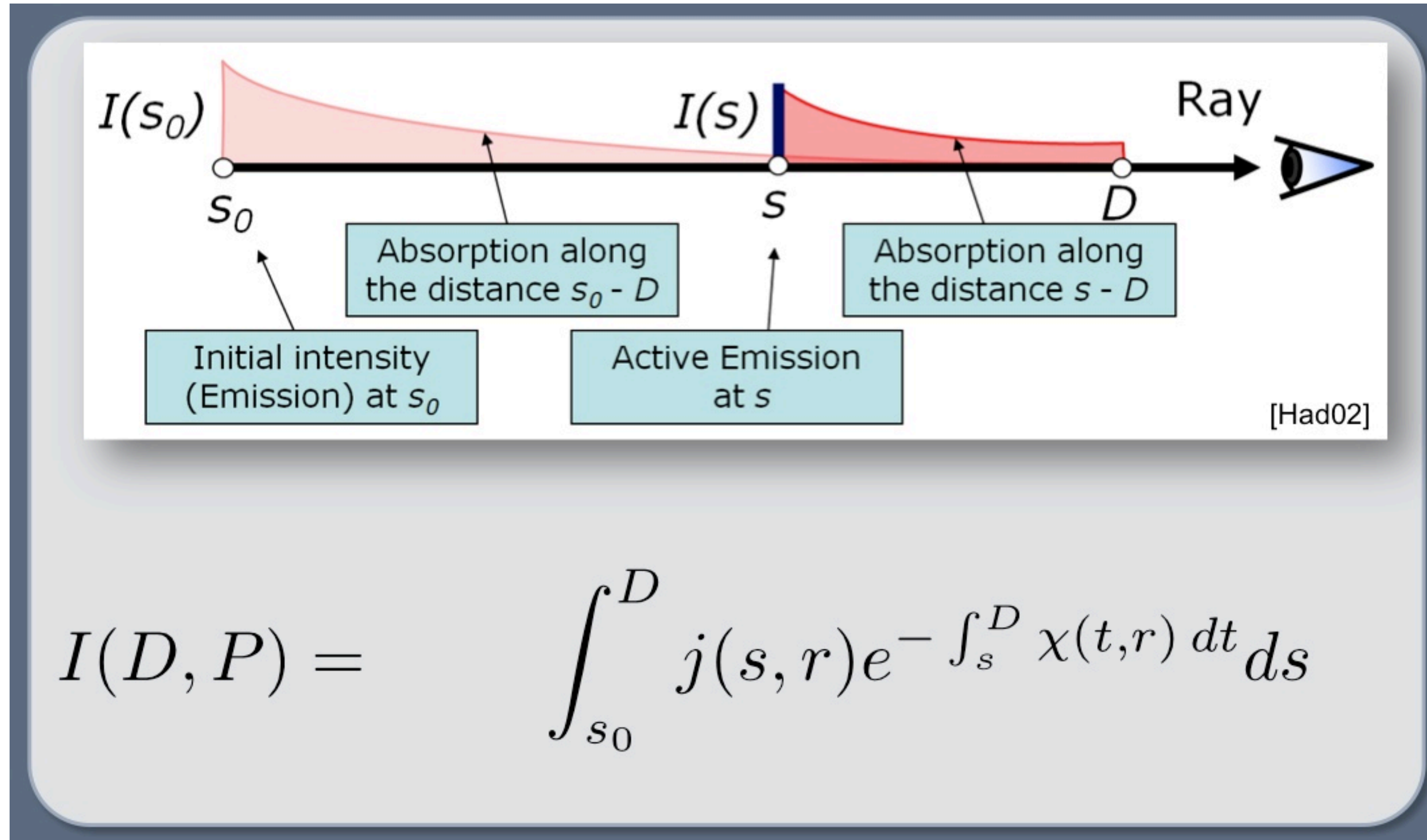


# Voxel cone tracing

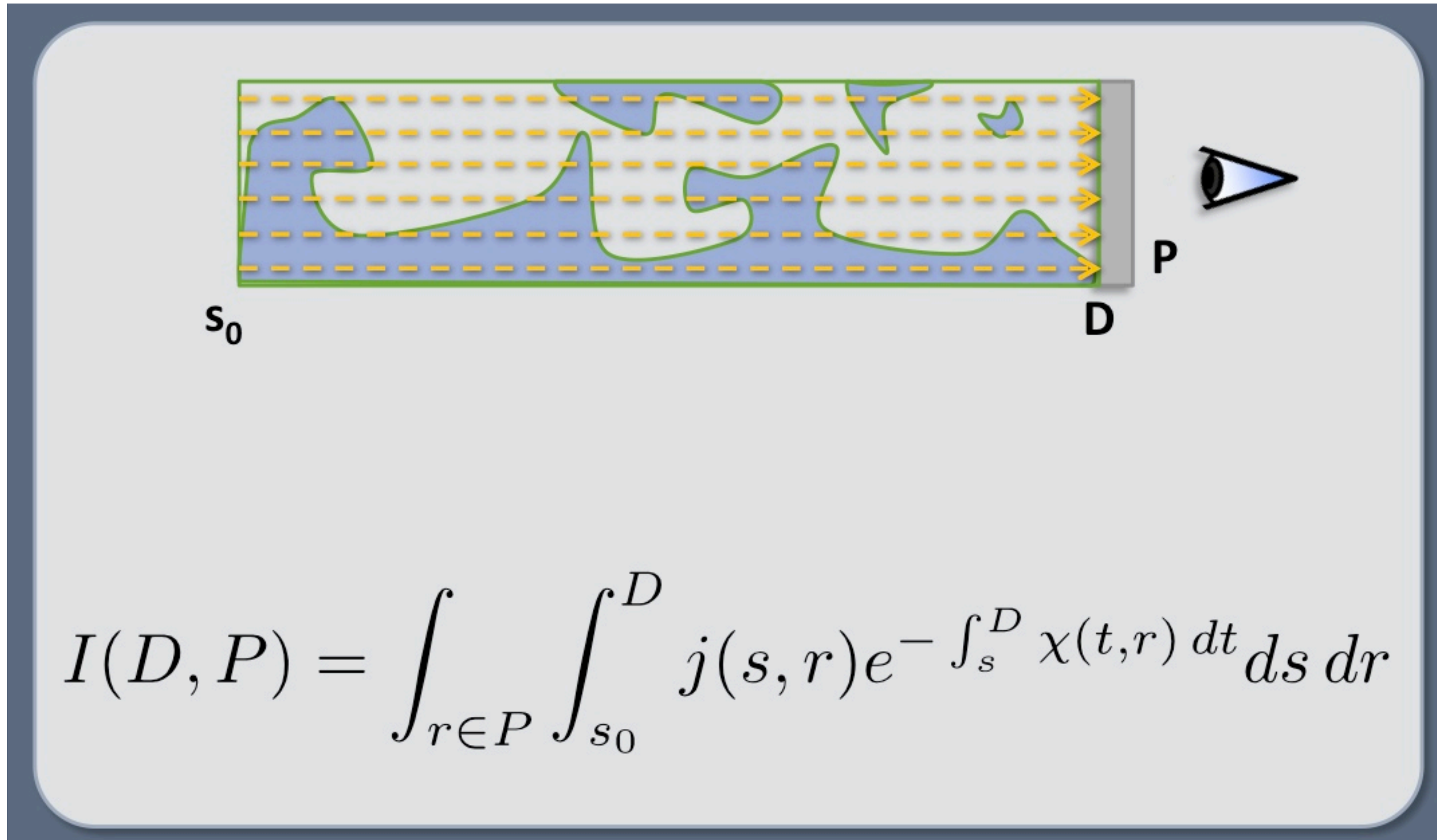
- **Main idea: rather than compute integrals during rendering by stochastic point sampling (requiring tracing of many rays through scene to produce low variance estimates), efficiently estimate integral over “cone” of rays traced through prefiltered voxel representation of geometry**
- **Treat prefiltered geometry as a participating media (density field) rather a definite surface**



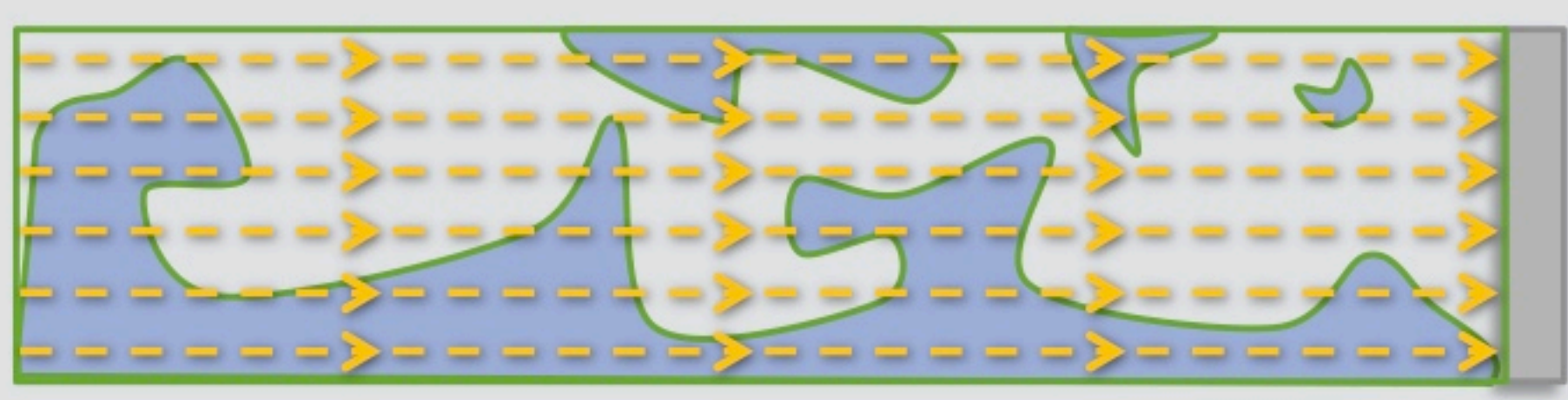
# Review: attenuation along a ray



# Integrate over bundle of rays



# Discretize integrals along rays

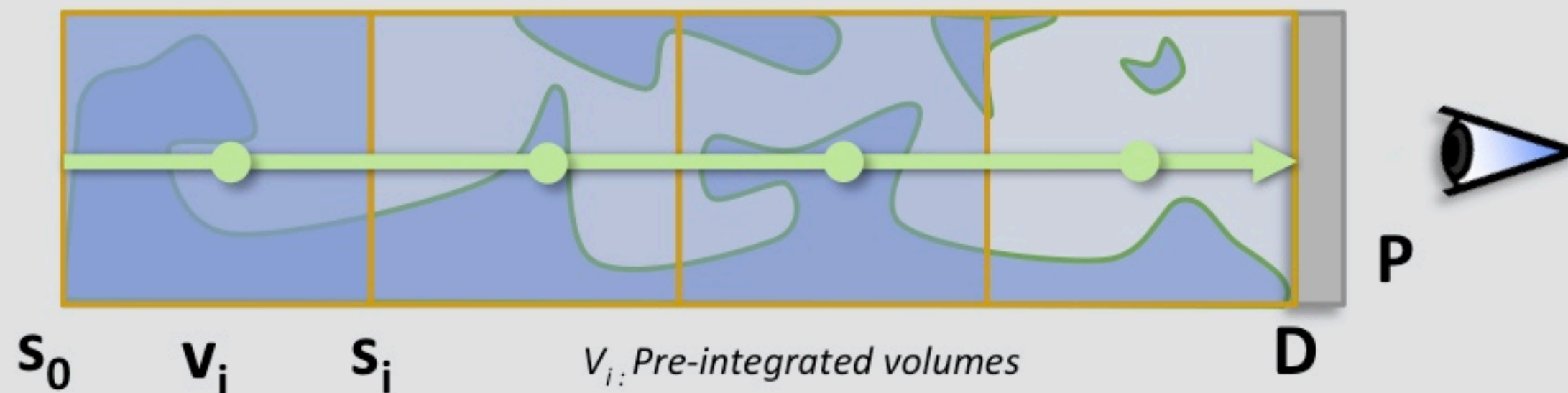


The diagram shows a rectangular volume  $D$  containing a blue irregular shape. A point  $P$  (represented by an eye) is on the right, looking through  $D$ . A series of horizontal dashed yellow arrows represent rays originating from  $P$  and passing through the volume. The volume is discretized into segments  $V_i$  along a ray path. The ray path is marked with points  $s_0, v_i, s_i$  and labeled  $V_i$ : Pre-integrated volumes. The volume  $D$  is labeled at the bottom right.

$$I(D, P) = \int_{r \in P} \sum_{i=0}^n \left( Q(s_i, s_{i+1}, r) \cdot e^{-\sum_{j=i+1}^n \tau(s_j, s_{j+1}, r)} \right) dr$$

# Pre-integrate volumes

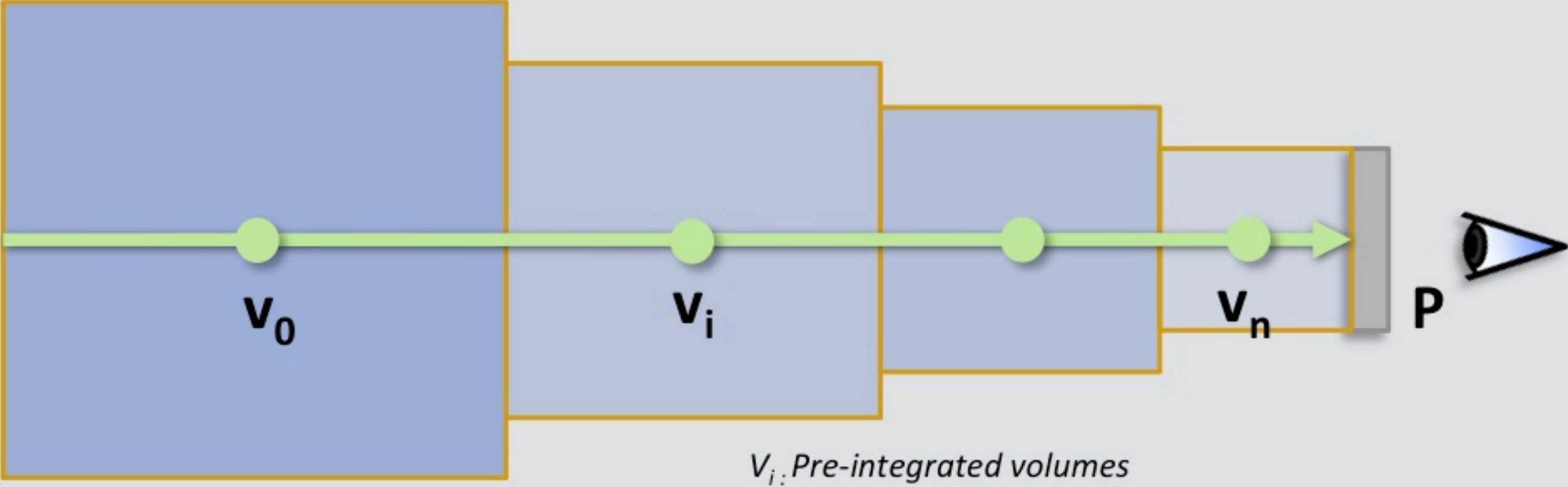
- Move integration over bundle of rays into summation over voxels
- Sum emission over all voxels, modulated by attenuation due to opacity
  - Makes de-correlation assumption (of energy and opacity)



$$I(D, P) \approx \sum_{i=0}^n \left( \left( \int_{r \in P} Q(s_i, s_{i+1}, r) dr \right) \cdot \prod_{j=i+1}^n \left( \int_{r \in P} e^{-\tau(s_j, s_{j+1}, r)} dr \right) \right)$$



# Voxel cone tracing: integration over cone of rays



The diagram illustrates the process of voxel cone tracing. A green ray originates from a point  $V_0$  and passes through a series of overlapping blue rectangular volumes, labeled  $V_i$  and  $V_n$ . These volumes are referred to as "Pre-integrated volumes". The ray terminates at a gray plane labeled  $P$ , which is viewed from the right by an eye icon. The ray path is shown as a green line with circular markers at  $V_0$ ,  $V_i$ , and  $V_n$ .

$$I(D, P) \approx \sum_{i=0}^n \left( \left( \int_{r \in P} Q(s_i, s_{i+1}, r) dr \right) \cdot \prod_{j=i+1}^n \left( \int_{r \in P} e^{-\tau(s_j, s_{j+1}, r)} dr \right) \right)$$

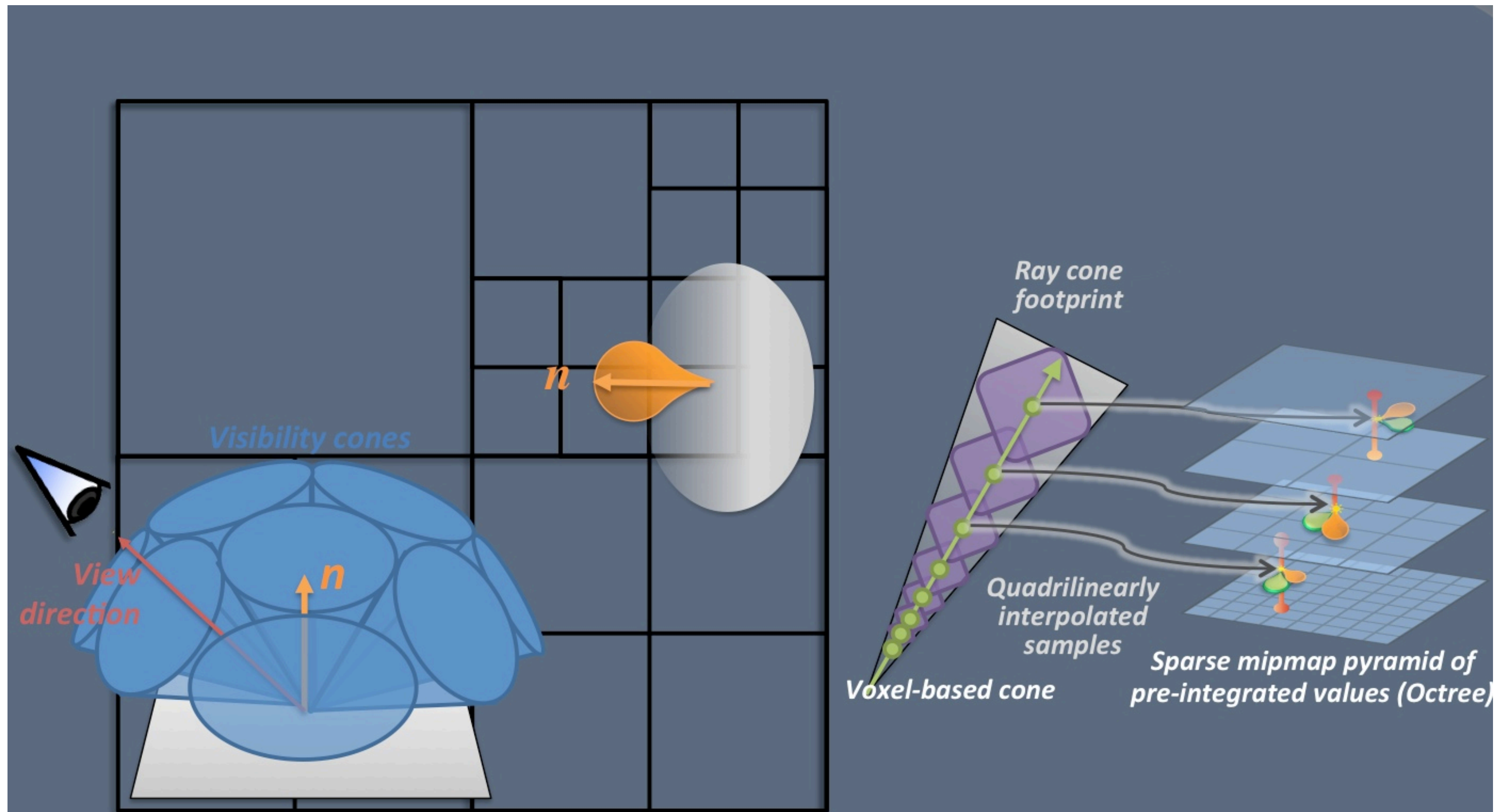
# Ambient Occlusion



Scene model courtesy of Guillermo M. Leal Llaguno

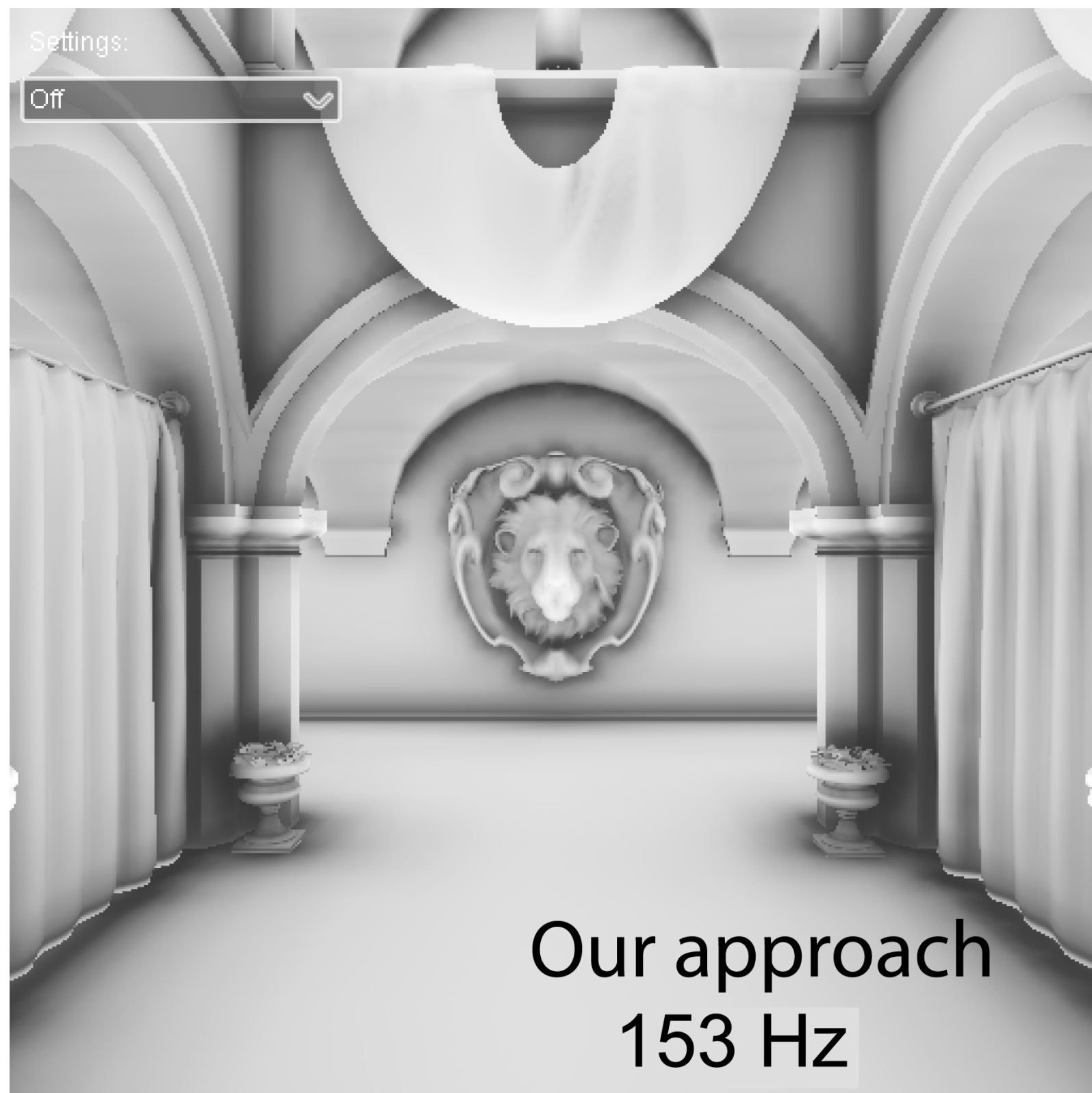
- **Fraction of distant incoming irradiance attenuated by occluders (integration of occlusion over hemisphere)**
- **Coarse proxy for shadowing of ambient indirect illumination**

# Ambient occlusion via voxel cone tracing



- Store prefiltered occlusion value  $\alpha$  at each octree node
- Step cone from surface point through octree:  $\alpha = \alpha + (1 - \alpha) f(r) \alpha_{\text{voxel}}$
- Octree depth and determined from cone width
  - Use deep voxels near starting surface point, higher (larger) voxels farther away

# Comparison with ray traced A0 solution

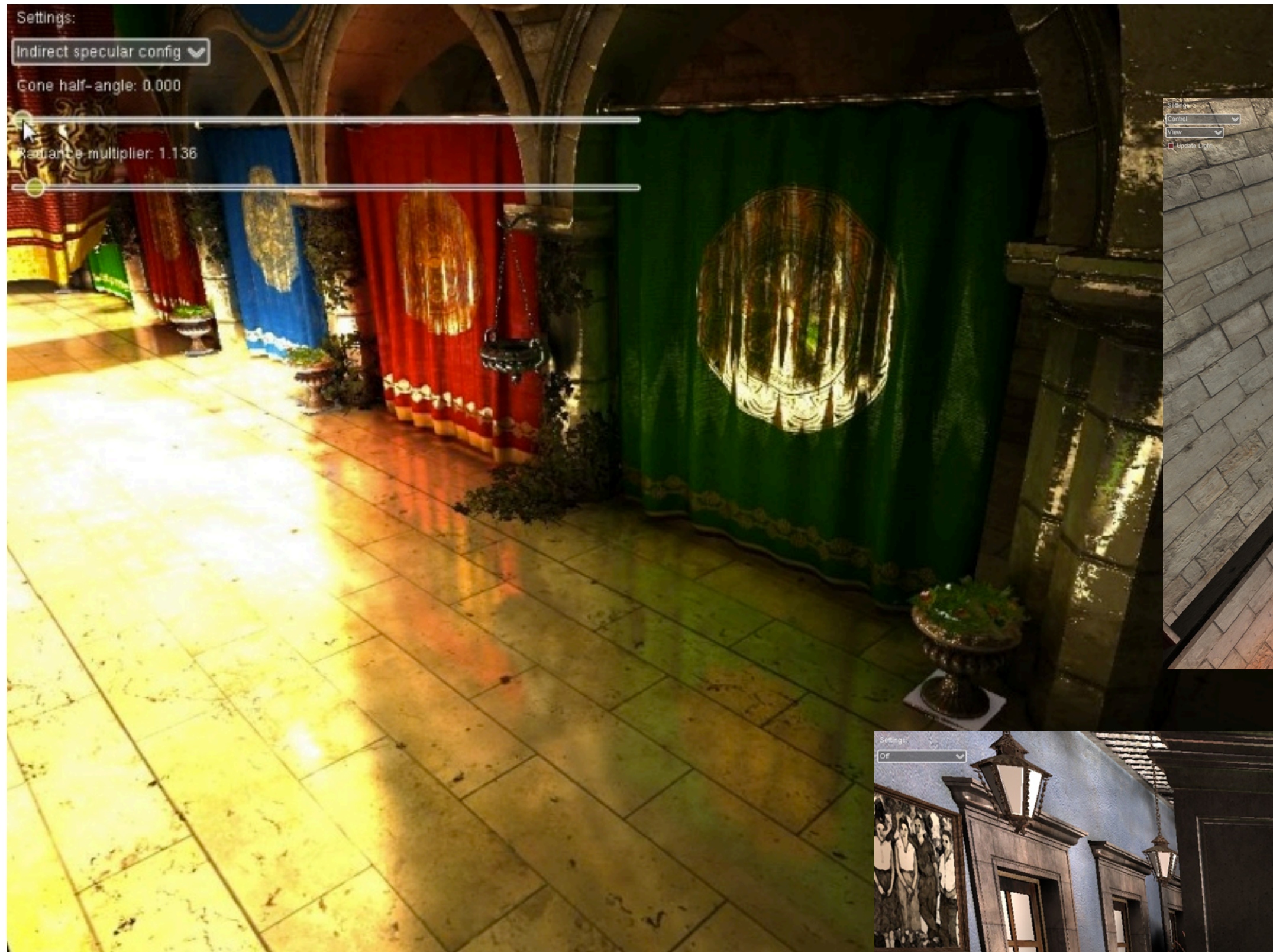


**Voxel cone tracing using three cones**

# Indirect illumination using voxel cone tracing

- **Step 1: compute direct illumination due to light source at every leaf voxel**
  - Render  $W \times H$  “light-view-map” image from light.
  - For every fragment sample, trace through SVO, deposit irradiance in leaf
  - Requires atomic update of data structure (similar to SVO construction)
- **Step 2: filter irradiance values to higher levels of octree**
  - Store Gaussian lobes in voxels: direction + variance
- **Step 3: (Gather phase) In standard forward or deferred rendering use cone tracing to compute indirect illumination at every fragment**
  - $\alpha = \alpha + (1 - \alpha)\alpha_{\text{voxel}}$   
 $c = \alpha c + (1 - \alpha)\alpha_{\text{voxel}}c_{\text{voxel}}$
  - $c_{\text{voxel}}$  is radiance from voxel reflected in direction of gathering surface (cone origin)

# Examples



Scene model courtesy of Guillermo M. Leal Liaguno

# Discussion

- **Increasing use of voxelization techniques in interactive rendering**
- **When are voxelization techniques applicable?**
  - **When the cost of pre-filtering can be amortized!**
    - **Amortize construction/pre-filtering over many queries in a frame (e.g., GI)**
    - **Or over many frames (e.g., static geometry)**
- **Why not just drop triangles all together?**
  - **e.g. Ray cast through voxel representation for primary rays**

# Credits

- **This lecture used figures from:**
  - **Crassin et al. 2011**
  - **Cyril Crassin's BPS 2012 talk**
  - **Crassin and Green's Chapter 22 of OpenGL Insights (Cozzi and Riccio)**