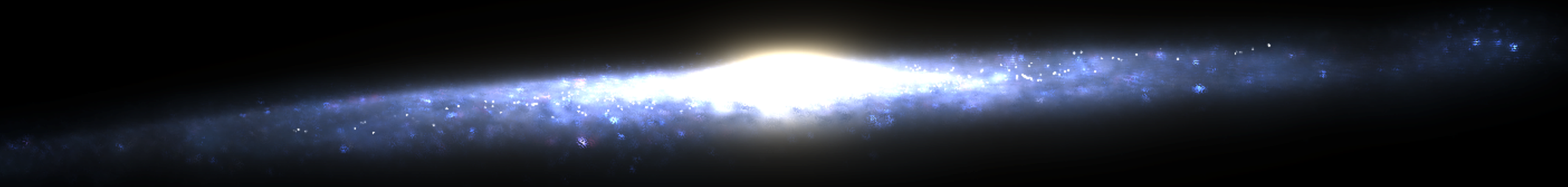
Direct Volume Rendering
Part I: Foundations, Transfer functions, Lighting models

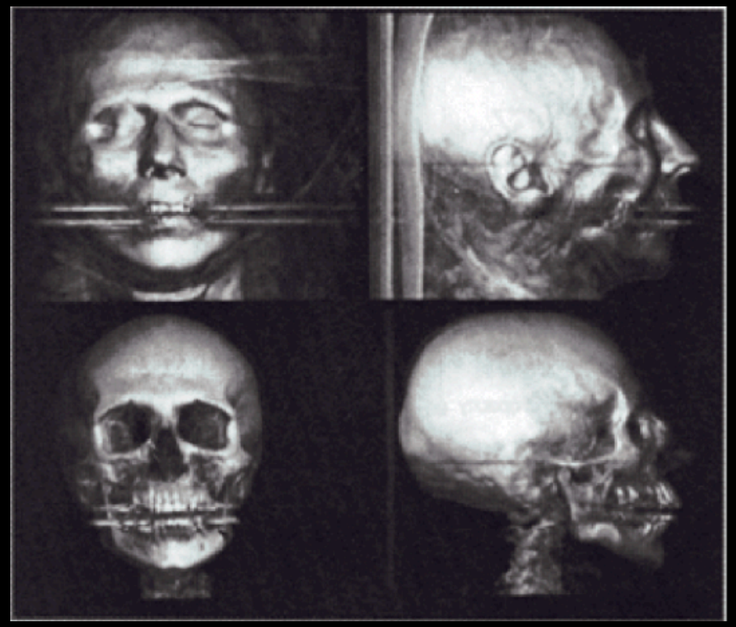


Alexander Bock
Moore-Sloan Postdoctoral Fellow
Center for Data Science
New York University

- Slides:
  - https://github.com/alexanderbock/alexanderbock.github.io/blob/master/lectures/2018/ds_ga_3001_017_volumerendering.pdf
  - alexanderbock.eu/lectures/2018/ds_ga_3001_017_volumerendering/index.html

Direct Volume Rendering

Direct Volume Rendering

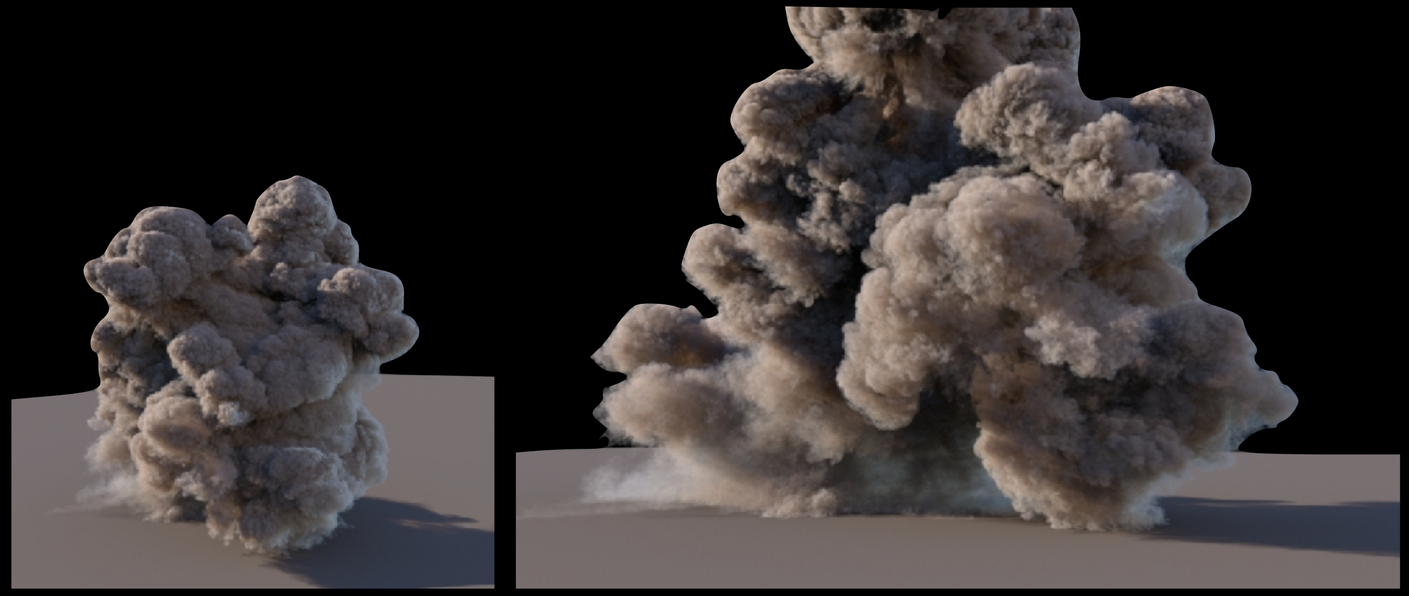


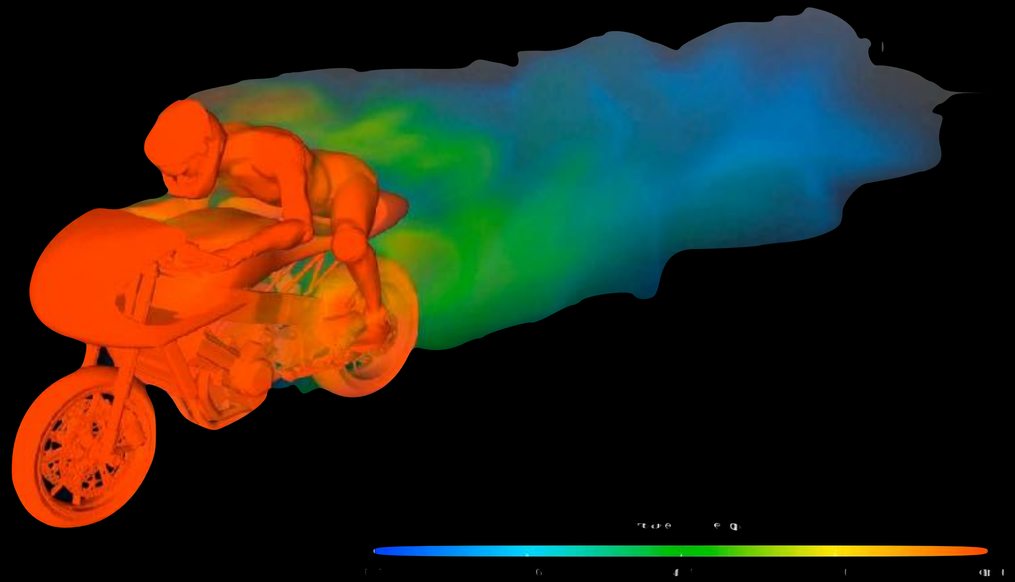
Levoy, 1988

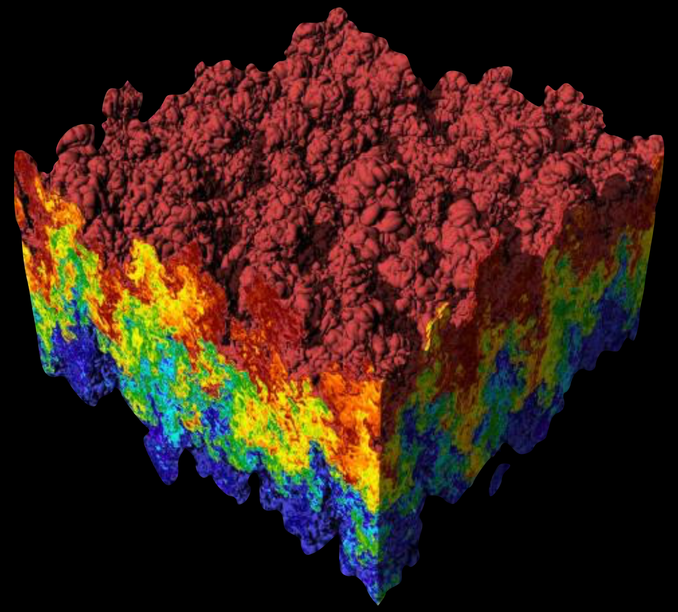
Kronander, 2012

Jönsson, 2012

Direct Volume Rendering







Direct Volume Rendering

- **Direct**
- No explicit geometric representation is generated

- **Volume**
- Extension of 2D image into 3D
- For today: Regular grid structure, but other methods exist (AMR, spherical, ...)

- **Rendering**
- ...

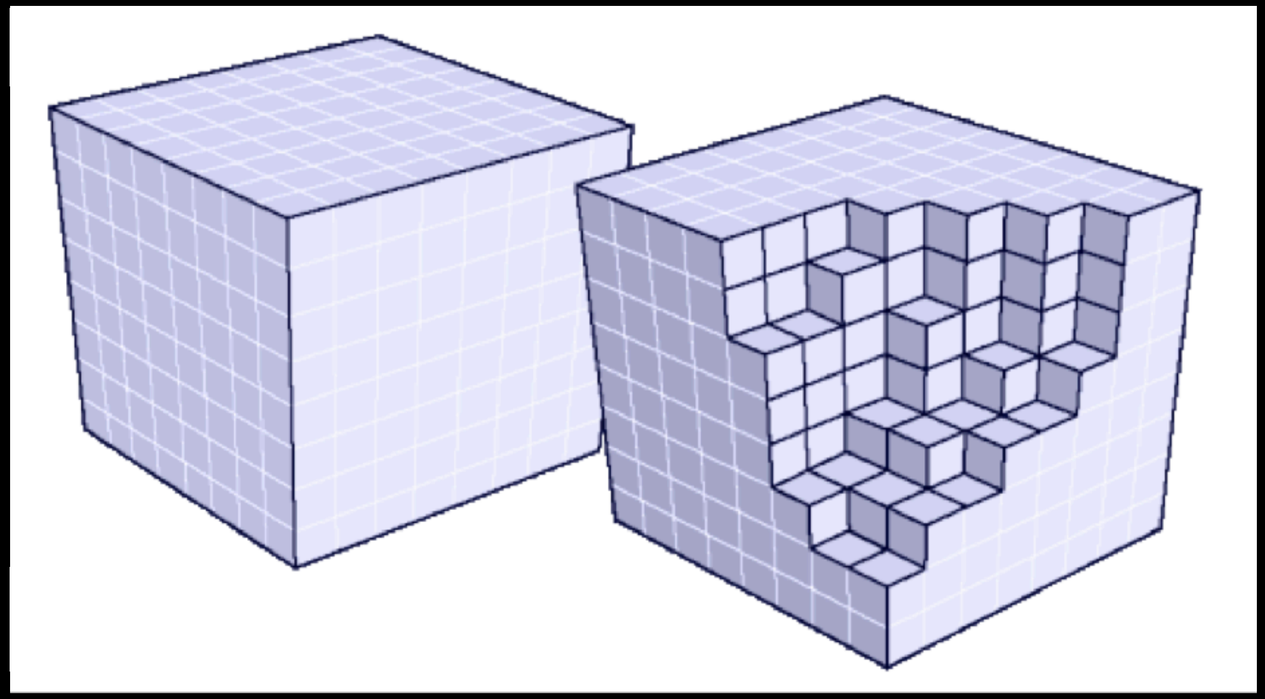


Image © Weiskopf, Machiraju, Möller

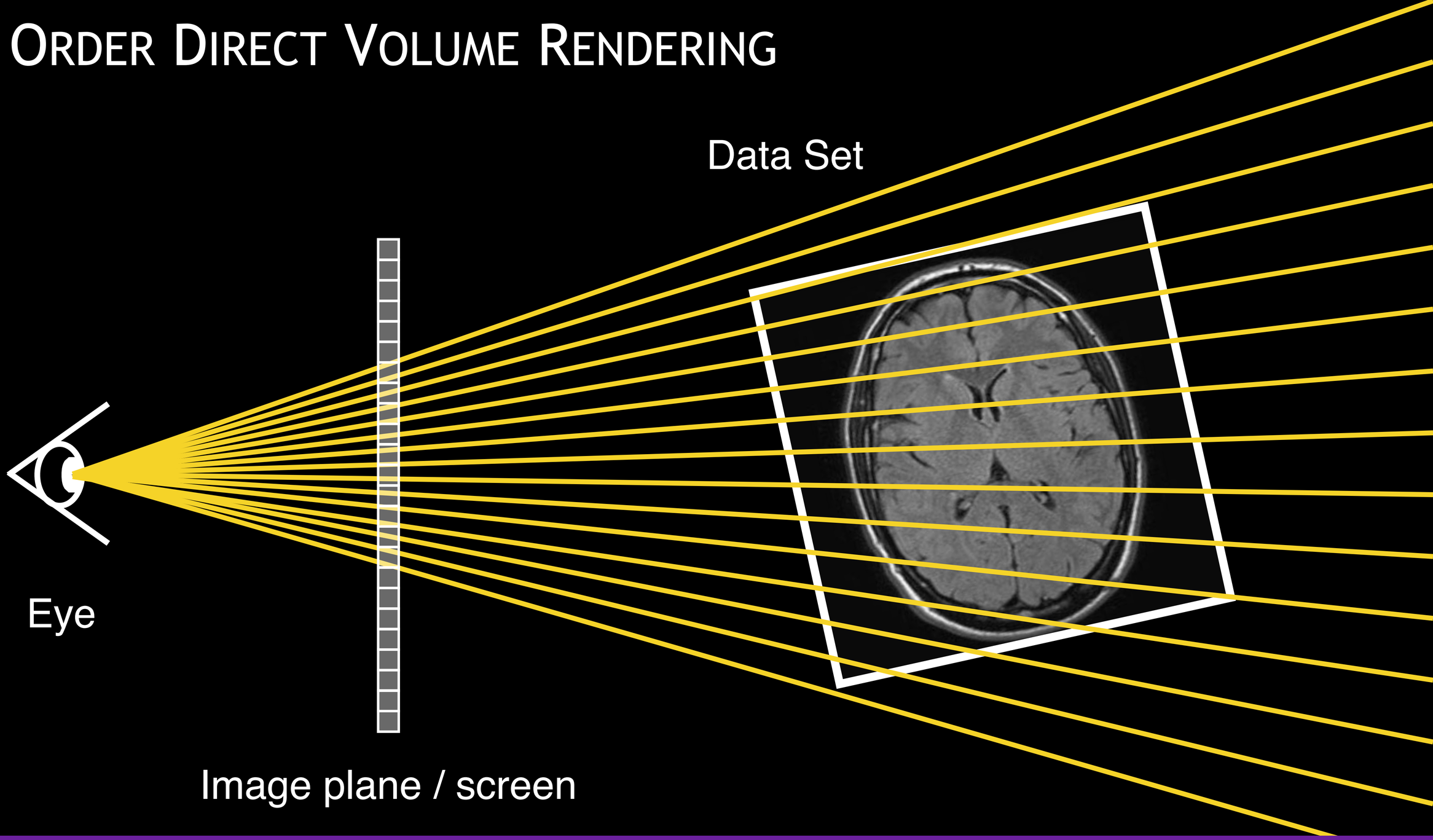
Direct Volume Rendering

- Image-order techniques
  - Perform operations on pixels to produce the image
  - -> Majority of today's implementations

- Object-order techniques
  - Perform operations on the elements of the scene
  - -> What we have been doing so far

Image © Weiskopf, Machiraju, Möller

Image Order Direct Volume Rendering



Emission Absorption Model

- Assumption
  Volume consists of small particles which are
  - opaque
  - non-reflecting
  - light emitting
  - light absorbing
  - the only light sources in the scene

Rendering Integral

$$I(\vec{x_c}) = I_0\left(\vec{x_0}\right) T\left(\vec{x_0}, \vec{x_c}\right) + \int_{\vec{x_c}}^{\vec{x_0}} T(\vec{x_c}, \vec{x}) \sigma_\alpha(\vec{x}) I_c(\vec{x}) \mathrm{d}\vec{x}$$

- No closed form solution of the integral in general
- Numerical approach to compute volume rendering integral: Riemann Sum

$$T(\vec{a}, \vec{b}) = \exp\left(-\int_{\vec{a}}^{\vec{b}} \tau(\vec{x}) \mathrm{d}\vec{x}\right)$$

I: Intensity

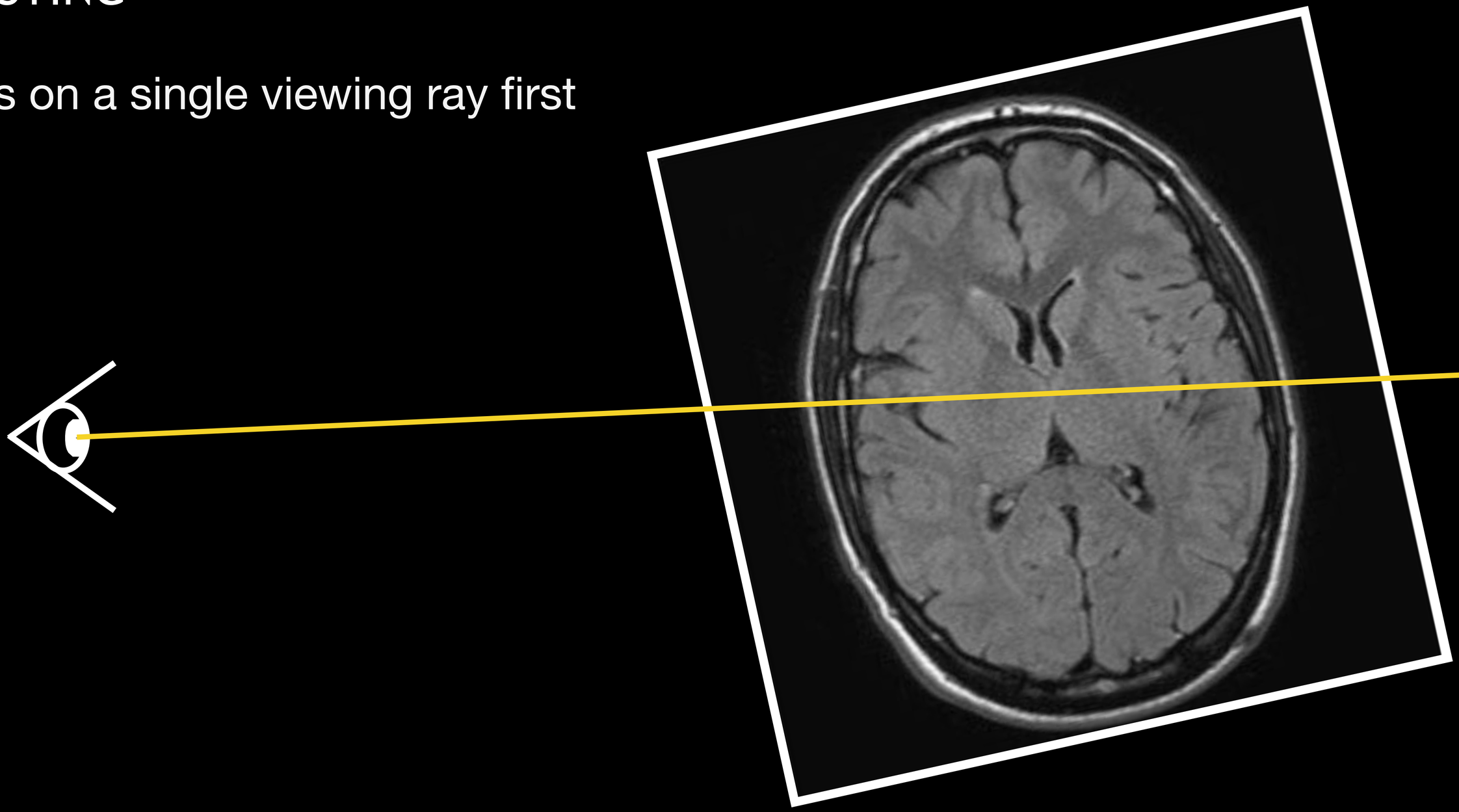
τ: Absorption factor

σ: Emission / Absorption

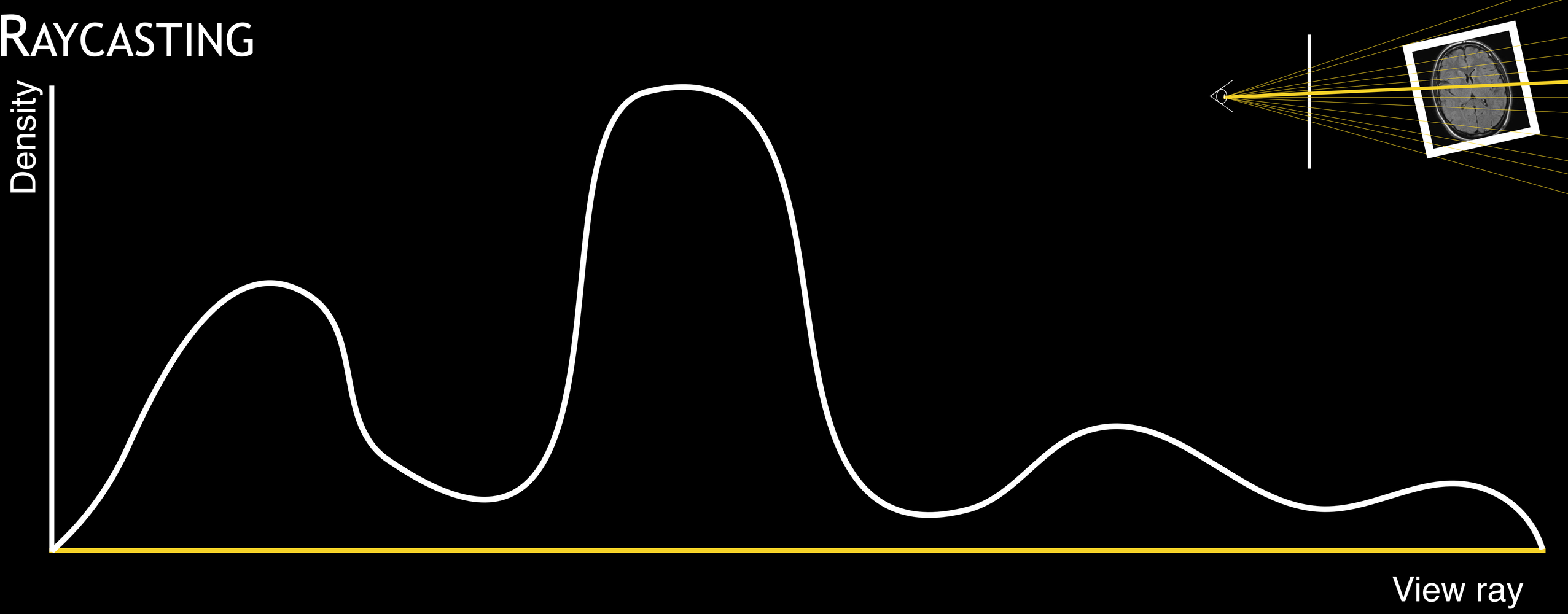
Max, Optical Models for Direct Volume Rendering, 1995

Raycasting

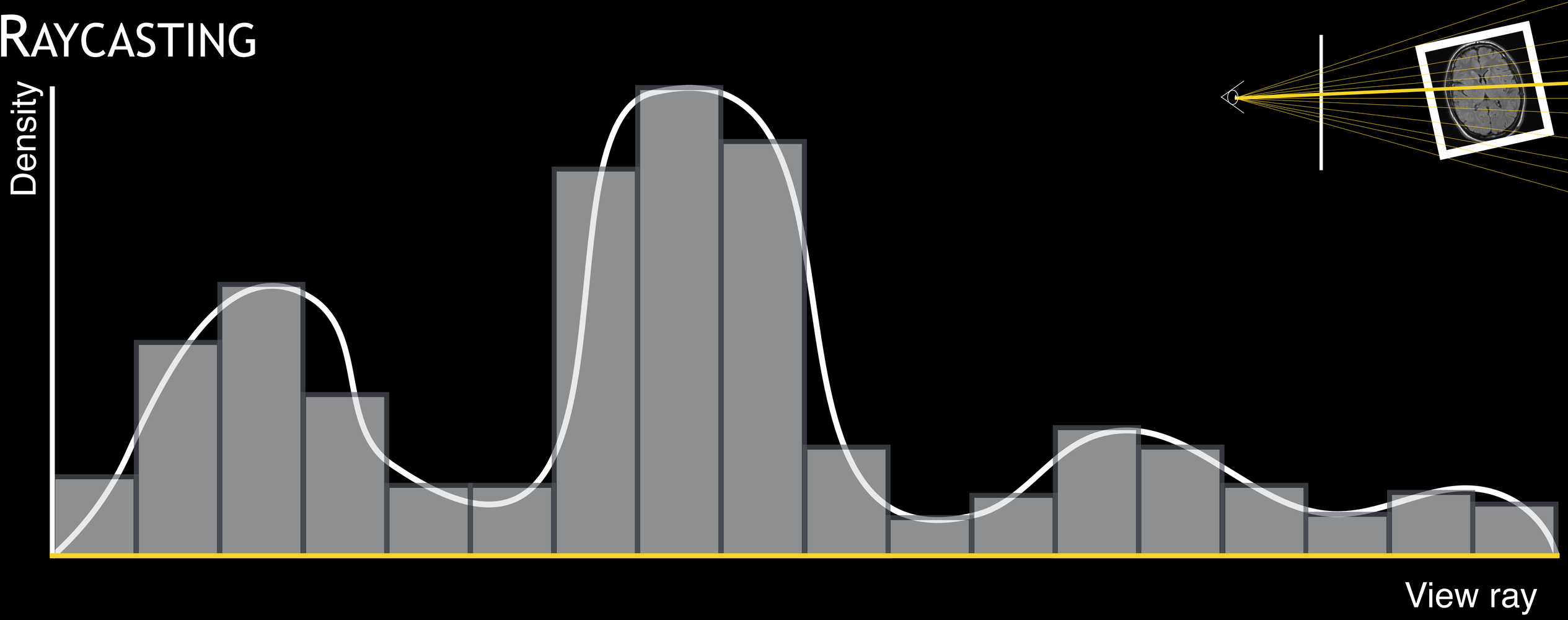
- Focus on a single viewing ray first



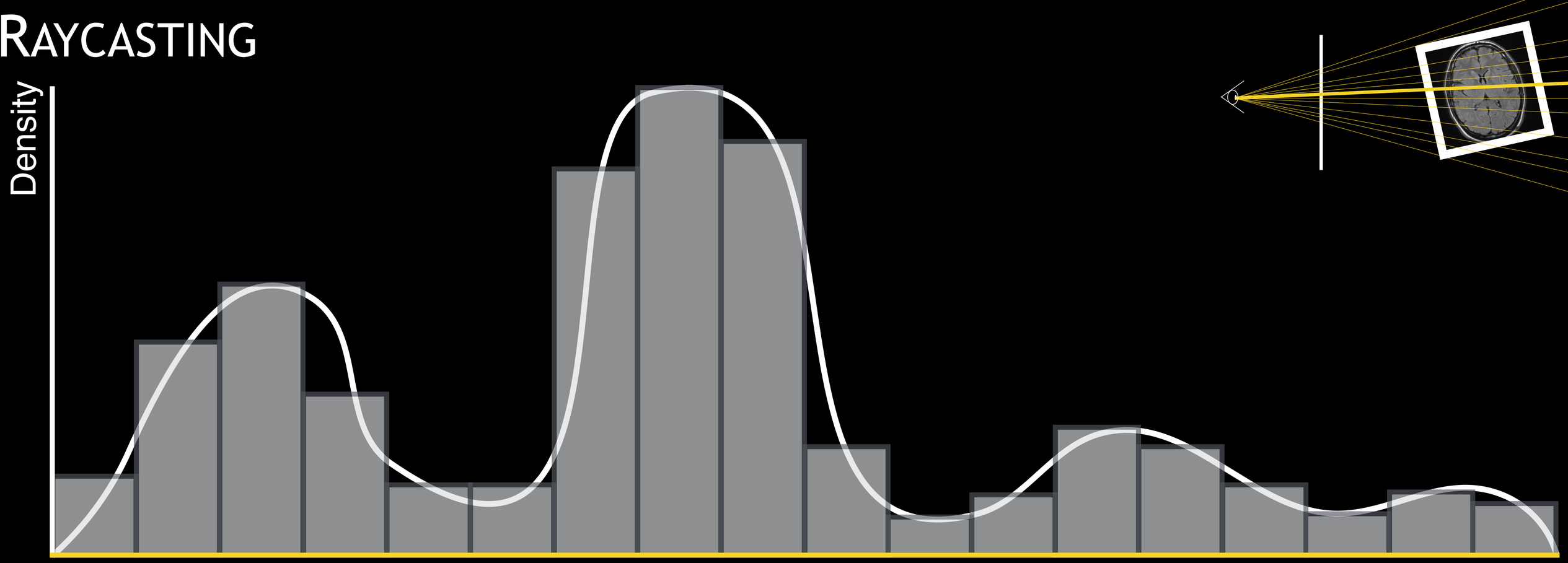
Raycasting



Raycasting



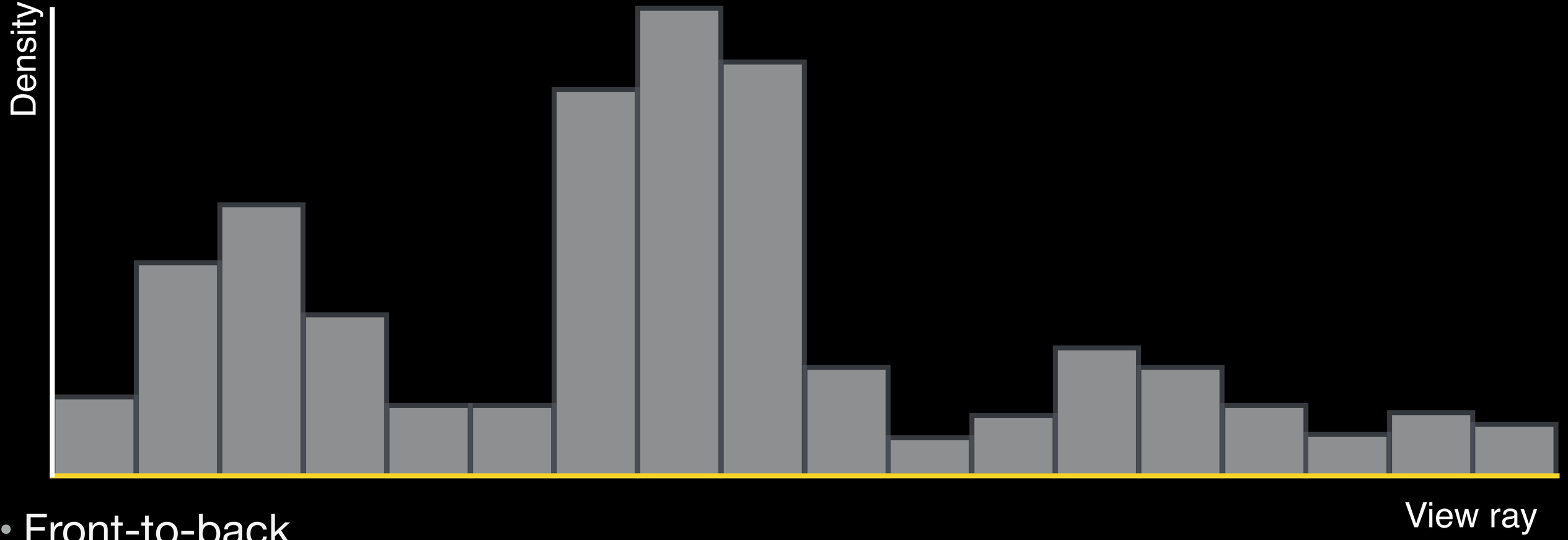
Raycasting



- Nyquist-Shannon sampling theorem
  - Required sampling frequency: 2 * highest frequency to completely reconstruct a signal
  - Problem: Frequency of the dataset is, in general, unknown -> user parameter

Compositing

Compositing



- Front-to-back
- Maximum intensity projection
- First Hit

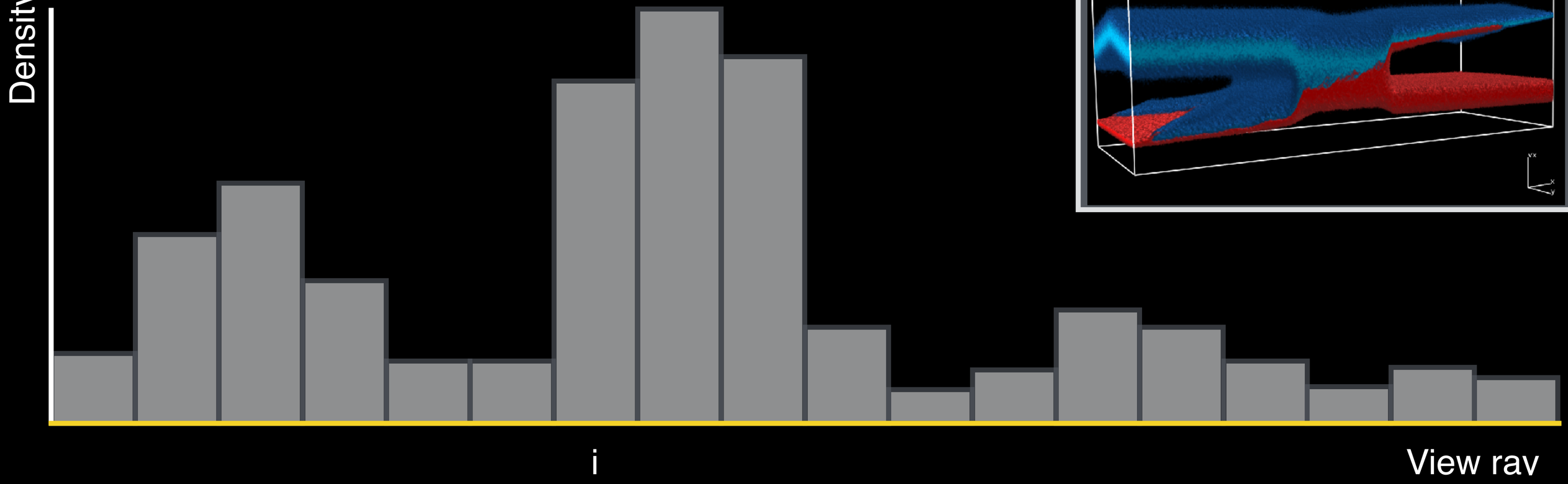
C: Color

A: Transparency/Alpha

Prime: Accumulated

Non-prime: Current value

Front-to-Back Compositing



- See OpenGL glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
- $C'_i = (1 - A'_{i-1})C_i + C'_{i-1}$
- $A'_i = (1 - A'_{i-1}) * A_i + A'_{i-1}$
- Stop when full opacity has been reached (or *Early Ray Termination*)

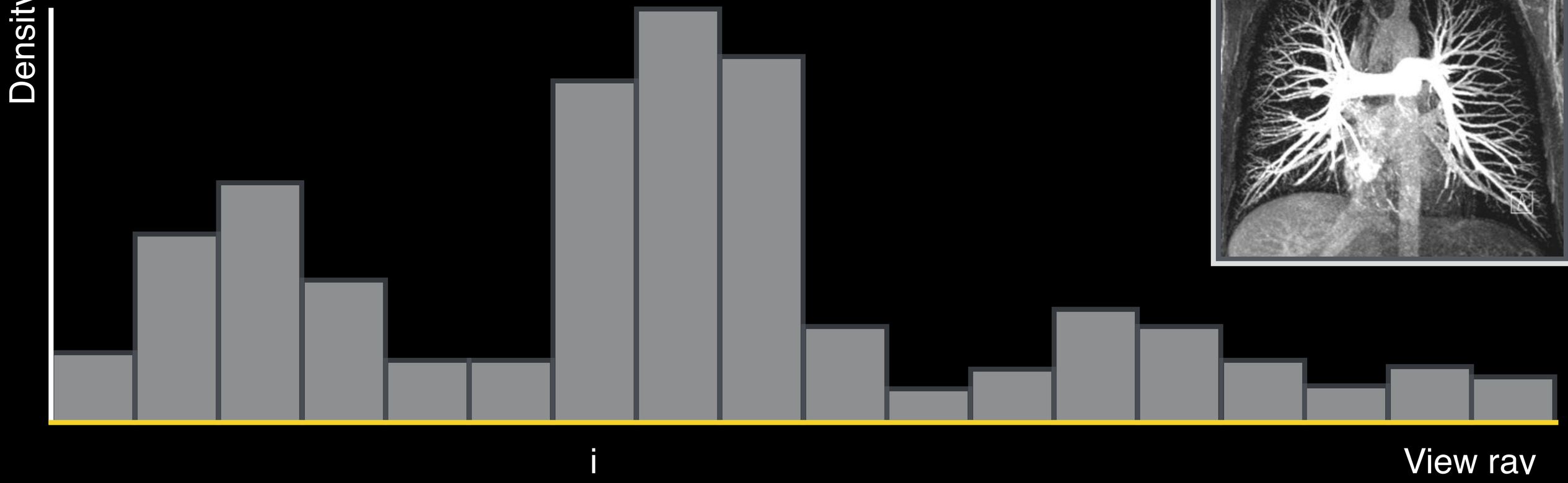
C: Color

A: Transparency/Alpha

Prime: Accumulated

Non-prime: Current value

Maximum Intensity Compositing



- Use the maximum intensity of all samples along the ray
- $C'_i = \max_j(C_i, C'_{i-1})$
- $A'_i = A_j$

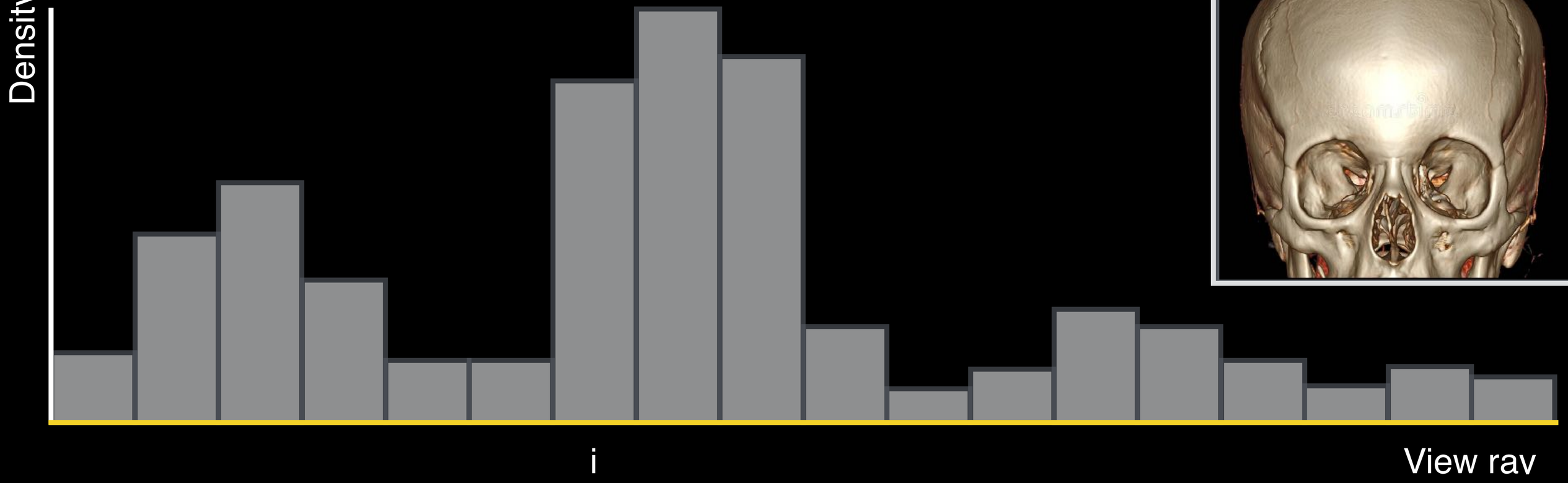
C: Color

A: Transparency/Alpha

Prime: Accumulated

Non-prime: Current value

First Hit Compositing



- Use the first value that is above a given threshold
- C'_i = first C_i above threshold := j
- $A'_i = A_j$
- -> Isosurface rendering

C: Color

A: Transparency/Alpha

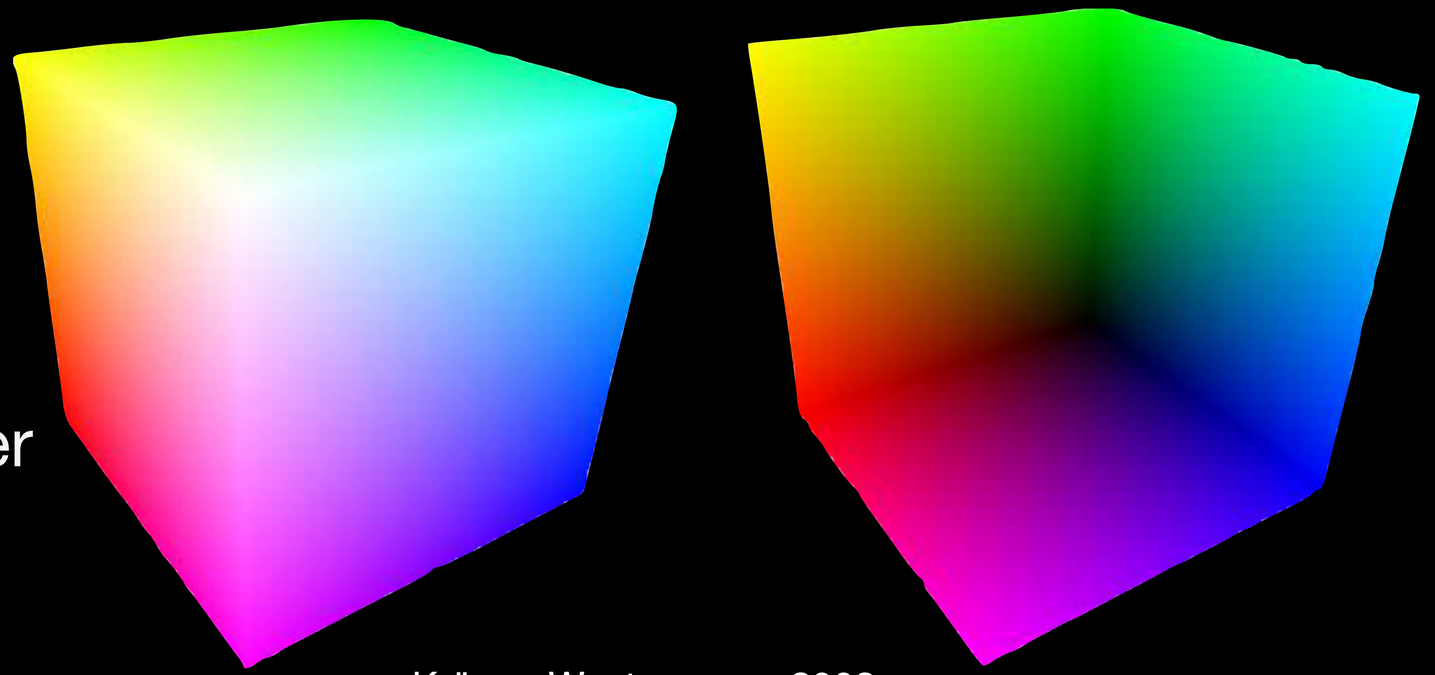
Prime: Accumulated

Non-prime: Current value

Techniques

Krüger-Westermann Proxy Geometry

- Render *Proxy Geometry* twice using the position as color [0, 1]
- First pass
  - Render front front side of the proxy geometry into a frame buffer
- Second pass
  - Render back side of the proxy geometry using the same camera parameters
- Third pass
  - Compute the difference between back side and front side color -> direction vector
  - Loop over the ray and perform compositing



Krüger, Westermann, 2003
Acceleration Techniques for GPU-based Volume Rendering

Raymarching

```
vec4 traverseRay(vec3 first, vec3 last) {
  vec4 result = vec4(0.0);
  vec3 direction = normalize(last - first);
  float stepIncr = length(last - first) / numSteps;          <- uniform float numSteps
  float t = 0.0;
  for (int i = 0; i < numSteps; ++i) {
    vec3 sampleCoord = first + t * direction;
    float intensity = texture(volume, sampleCoord).a;        <- uniform sampler3D volume
    vec4 color = classify(transferFunction, intensity);      <- uniform sampler1D transferFunction
    if (color.a > 0.0) {
      result.rgb = result.rgb + (1.0 - result.a) * color.a * color.rgb;
      result.a   = result.a + (1.0 - result.a) * color.a;
    }
    t += stepIncr;
  }
  return result;
}
```

Demo

WebGL

Transfer Functions

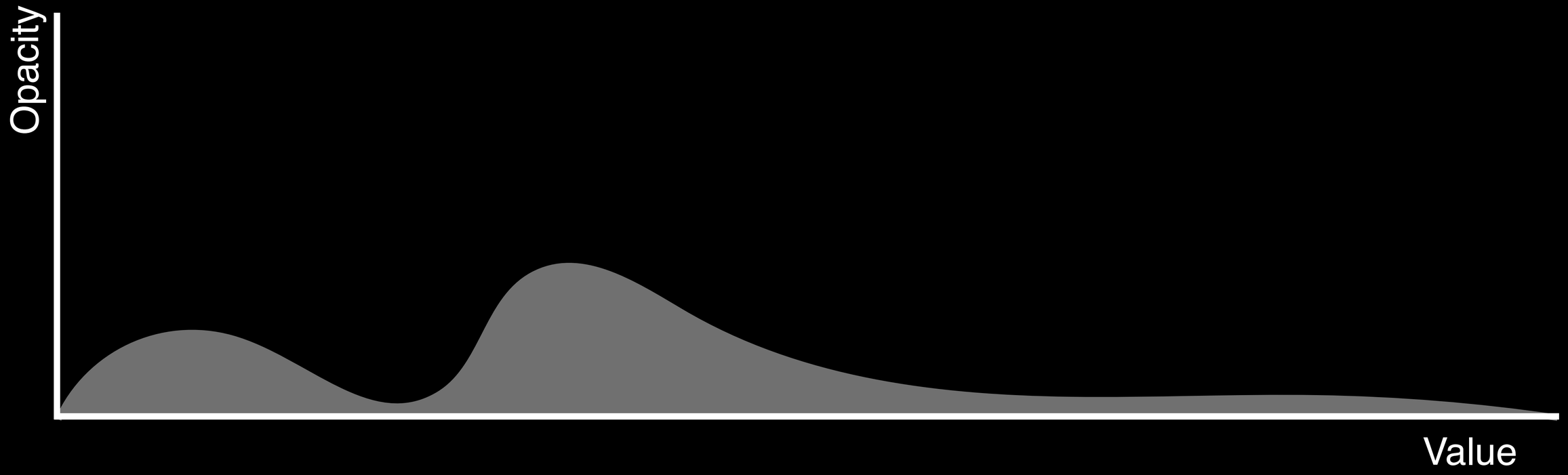
- Most volumes have a single quantity (density, x-ray absorption, # positron emissions, ...)
- We want to create colored images

- Transfer function (or lookup table (LUT)): $f : R^i \rightarrow R^3$ (=RGB)
- Majority of cases: $i = 1$

- In OpenGL, usually implemented with a sampler1D that is texture sampled using the quantity as parameter (requires normalized parameters)

Transfer Functions

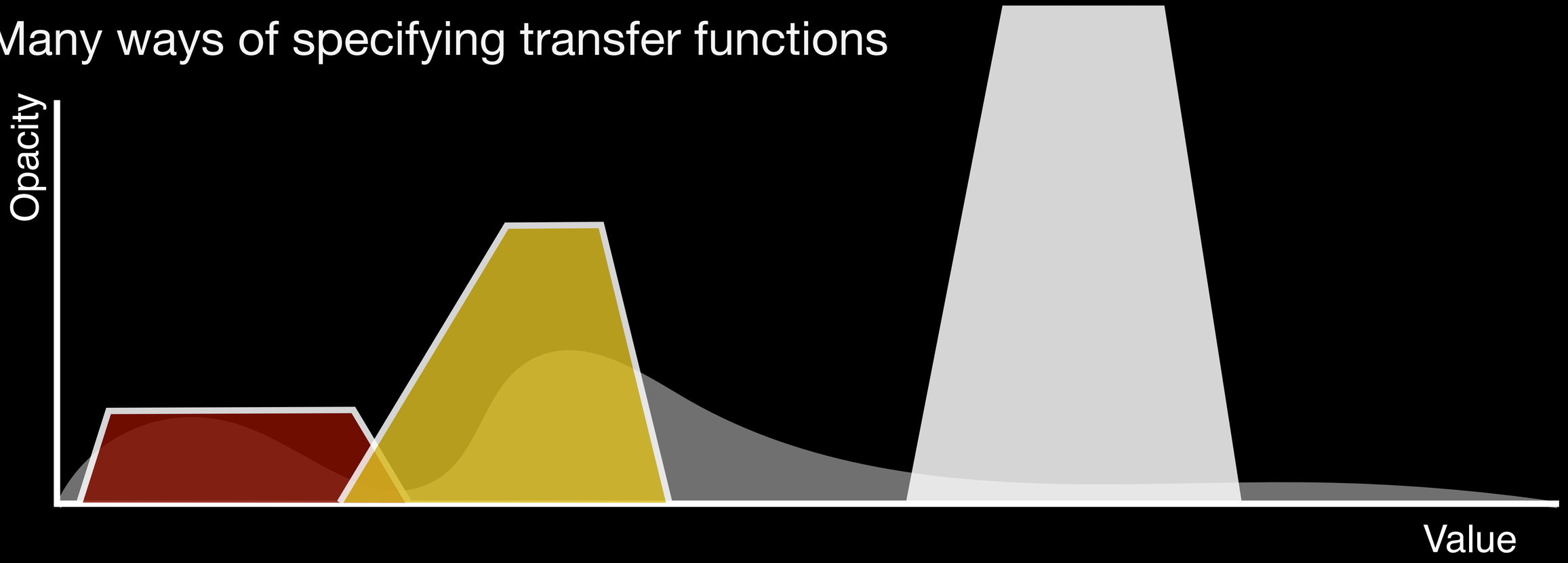
- Many ways of specifying transfer functions



- Background in this case: Histogram over the volume

Transfer Functions

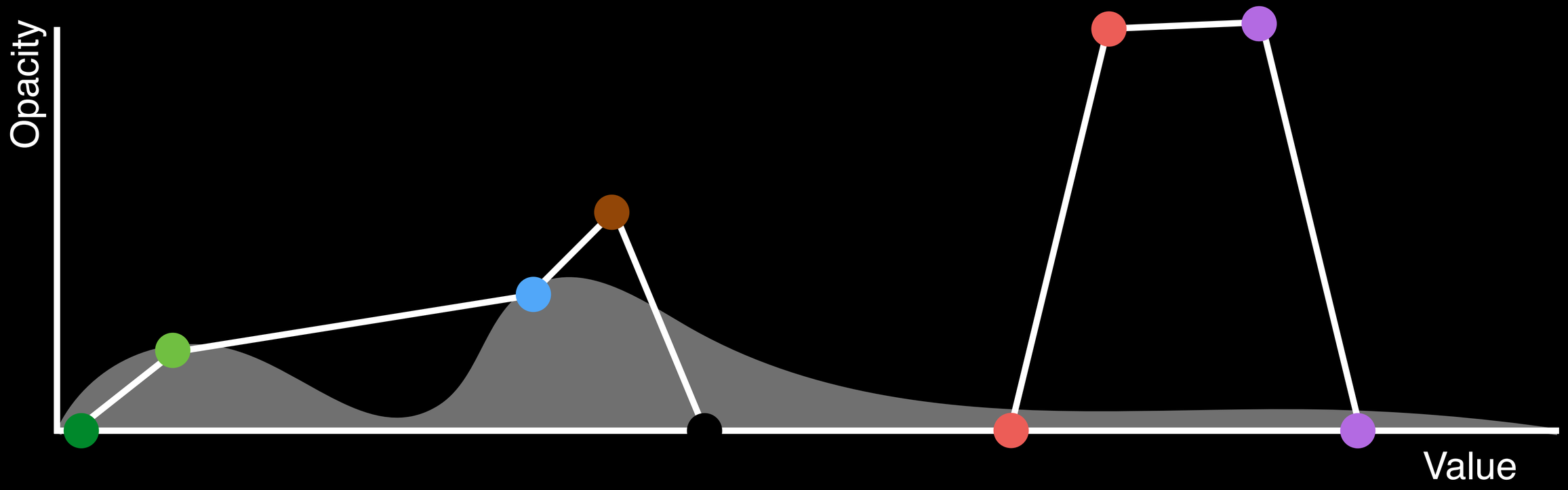
- Many ways of specifying transfer functions



- Widget based
- If a value is covered by multiple widgets, their colors are blended
- More intuitive mapping between visualized objects and colors

Transfer Functions

- Many ways of specifying transfer functions



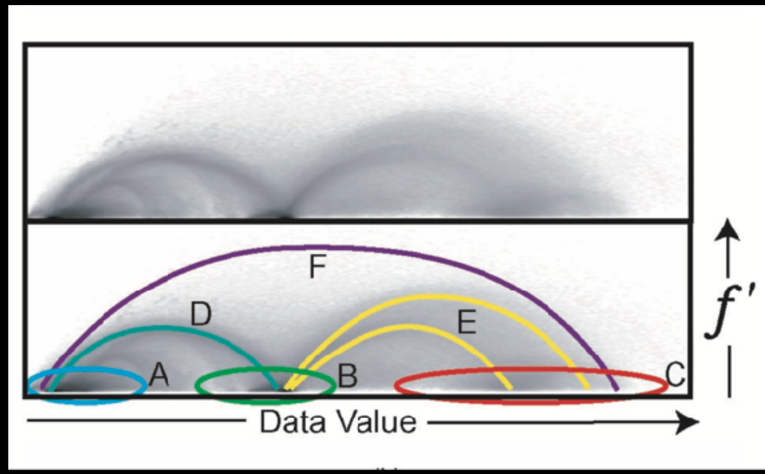
- Node based, colors interpreted between nodes
- More holistic approach

Demo

www.inviwo.org

Multidimensional Transfer Functions

- Not often used
- Examples
  - f vs ||∇f||
  - f vs ||∇f|| vs (1 / ||∇f||²) ∗ (∇f)T ∗ **H**f ∗ ∇f
  - Principle curvature



Kniss, Kindlmann, Hansen, 2002
Multidimensional Transfer Functions
for Interactive Volume Rendering

Lighting models

- Local lighting models similar to OpenGL lecture (Blinn-Phong)
- Problem: Since it is **direct** volume rendering, no explicit surface exists, therefore no normal
- -> replacement: gradient ∇f as replacement for the normal in lighting calculations
- Approximations of ∇f using finite differences
  - Forward difference, backward difference, *central difference*
  - Central difference:
    - f'(x,y,z) = f(x+h, y, z) - f(x-h, y, z),
      f(x, y+h, z) - f(x, y-h, z),
      f(x, y, z+h) - f(x, y, z-h)

Global Illumination

- Local lighting models cannot represent shadows, caustics, refractions, and other global effects
- *Global Illumination* techniques such as *Photon Mapping, Metropolis light transport, Radiosity,* and *raytracing* provide higher quality shadows, but are slower to compute

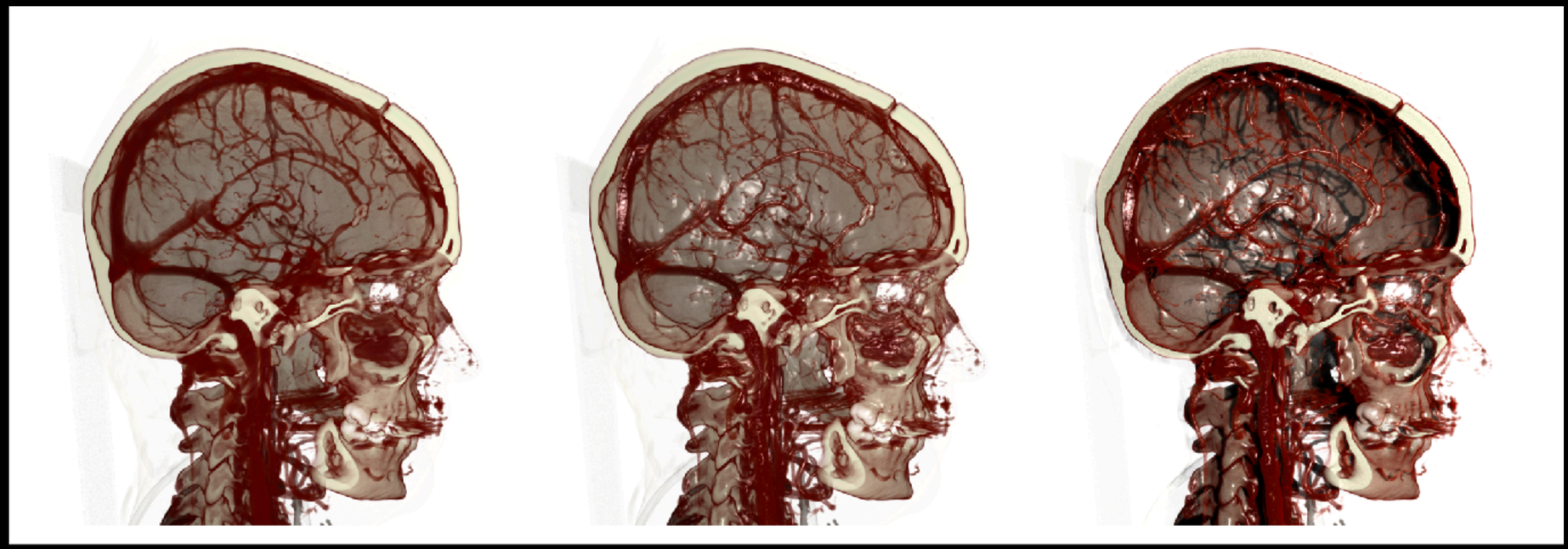


Image © Daniel Jönsson

Global Illumination

- For each sample evaluate the occlusion towards the light source
- In the case of *raytracing*, this is done through secondary rays that sample the volume analogous to the primary ray

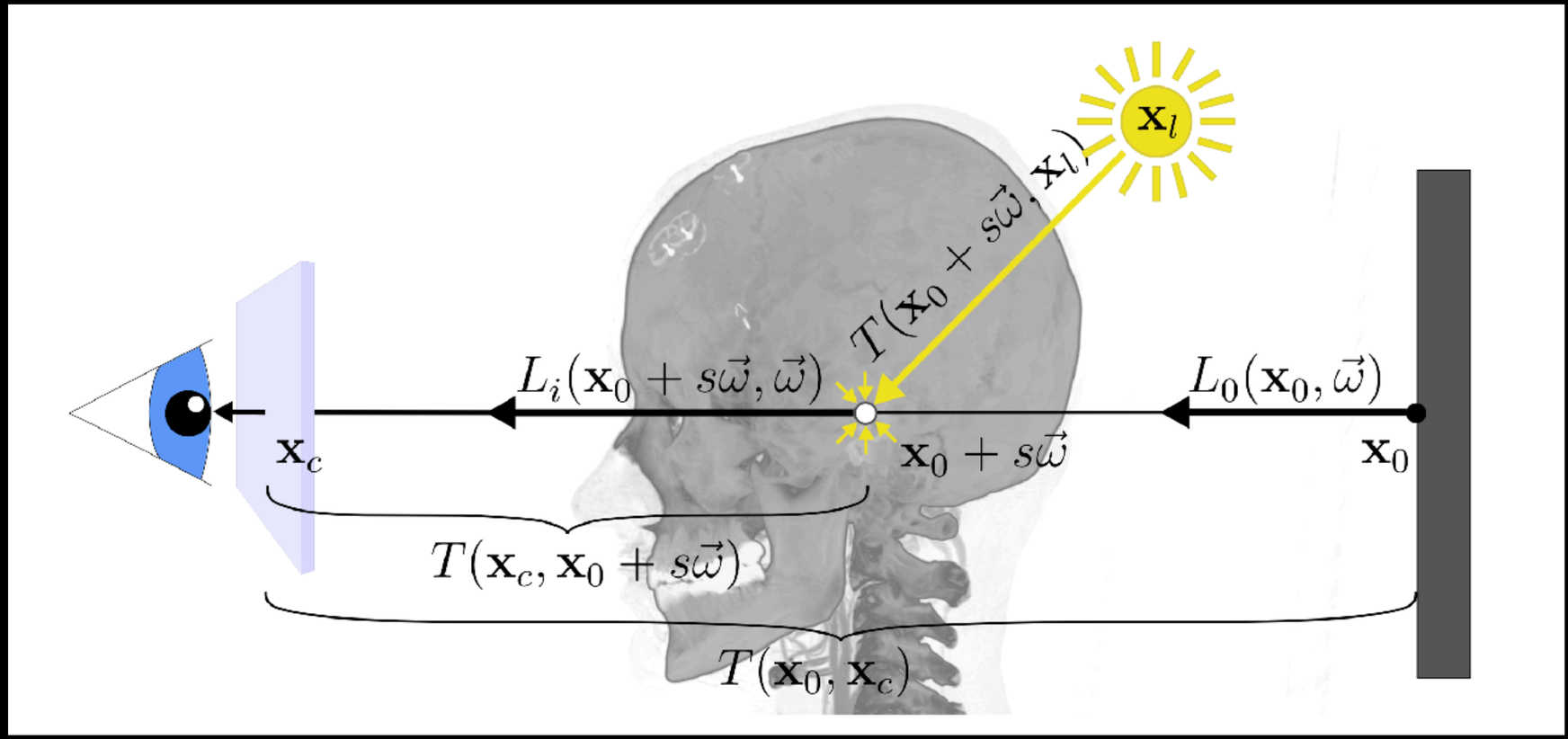
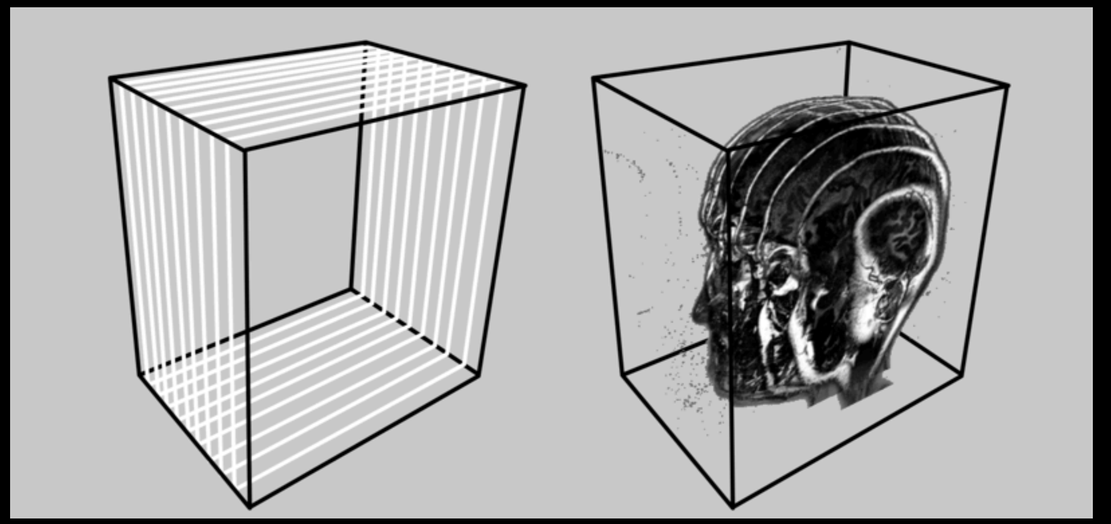


Image © Daniel Jönsson

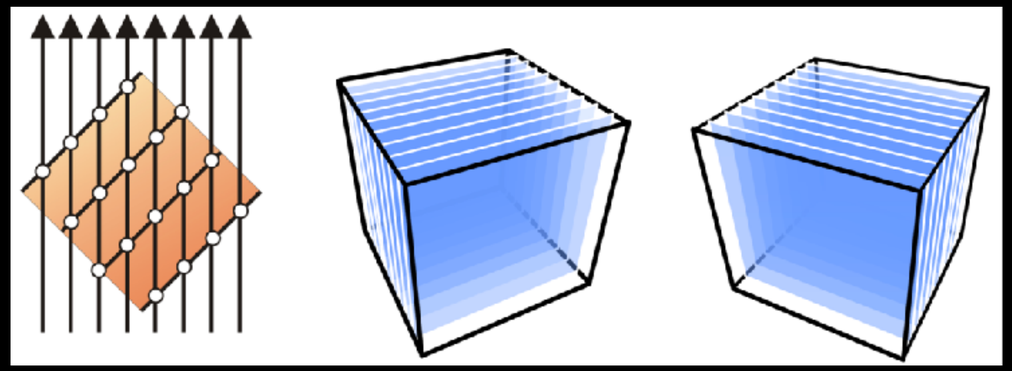
Object-Order Methods

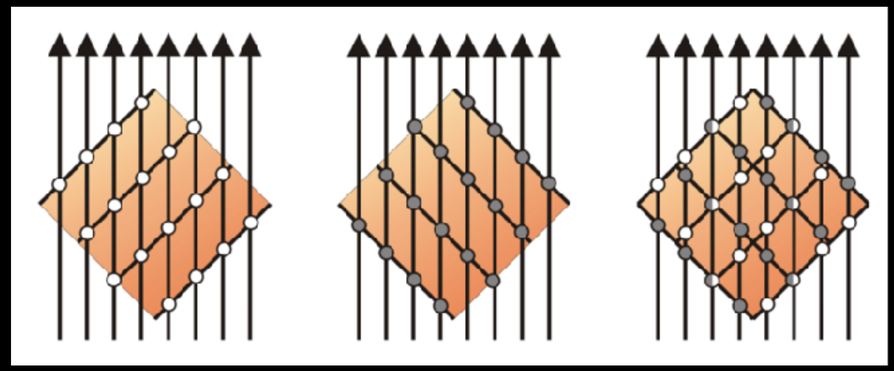
Texture Slicing

- Representing volume as stacks of 2D textures that are composited



- Option 1: Three stacks of 2D textures along primary axes
- Problematic if view direction changes past 45° as sampling points change drastically
- Does not require any complex shader programs; no need for 3D texture support



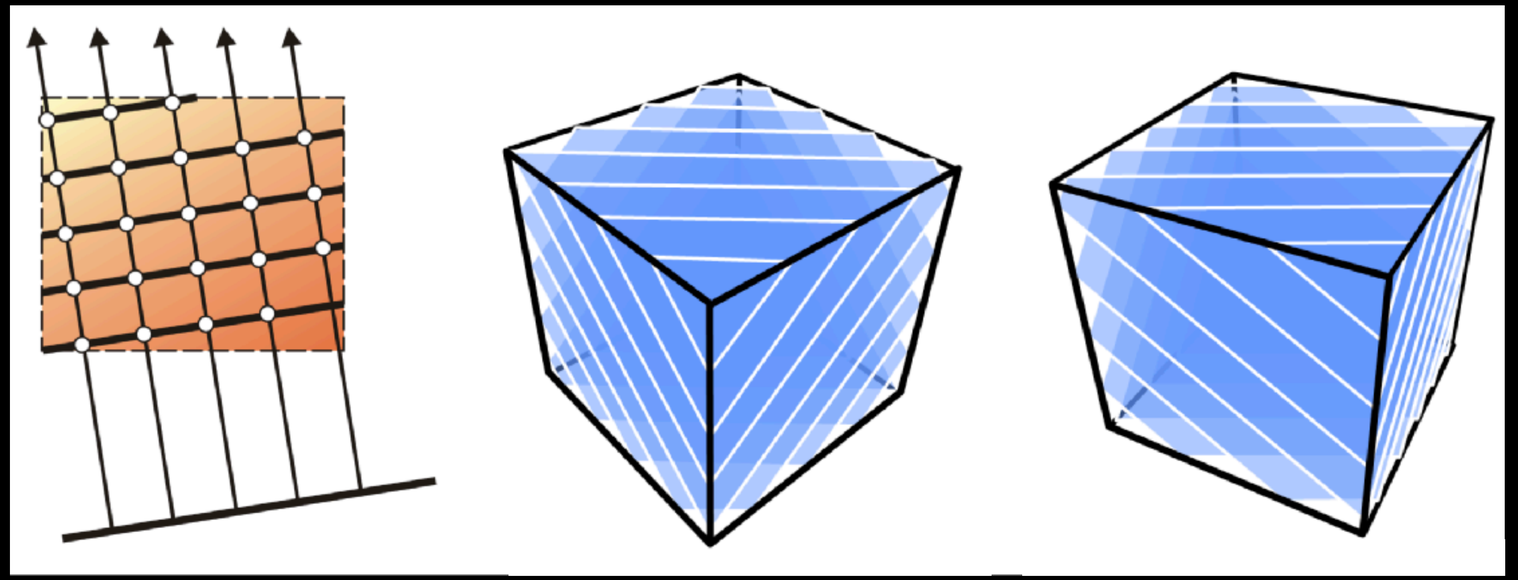


Images © Weiskopf, Machiraju, Möller

Texture Slicing

- Option 2: Single stack of 2D textures along view direction

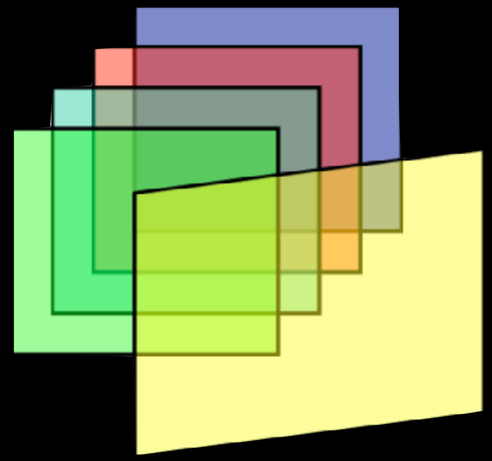
- Trilinear interpolation slower than bilinear interpolation
- Sampling distance independent of viewing angle



Images © Weiskopf, Machiraju, Möller

Volume Splatting

- Object-order
- Project each voxel onto the image plane individually
- Each voxel is represented by a 3D kernel
  - Kernel is coverted into 2D footprint on the image plane
  - Size and shape of kernel determines image quality (sharpness, # holes, ...)
- One voxel is splatted onto many pixels
- Voxels are added within sheets
  - Front-to-back compositing each sheet
  - Accumulating final result in a separate buffer



Westover, 1990
Splatting: A Parallel, Feed-Forward Volume Rendering Algorithm