Towards Path Tracing

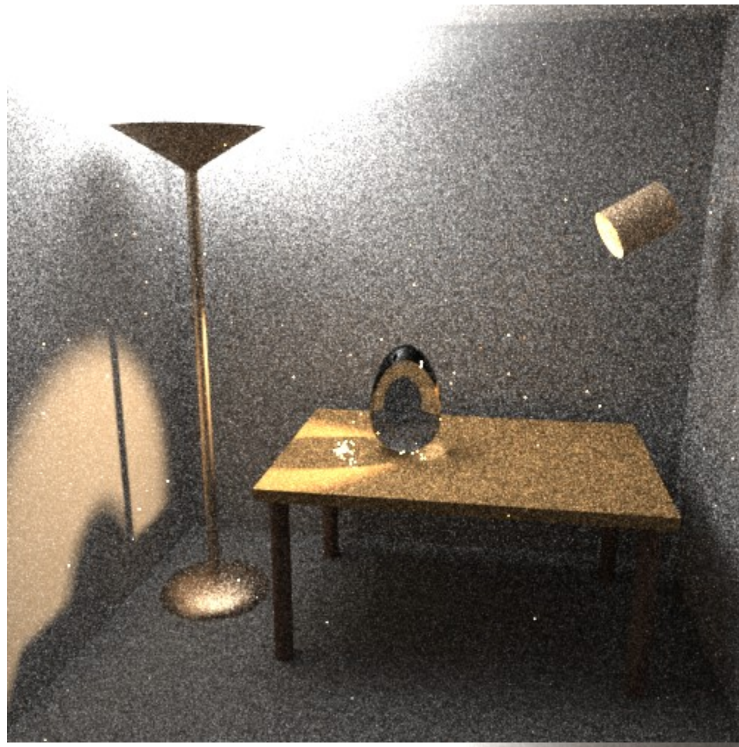
Pixel Filtering and Multidimensional Monte Carlo Integration

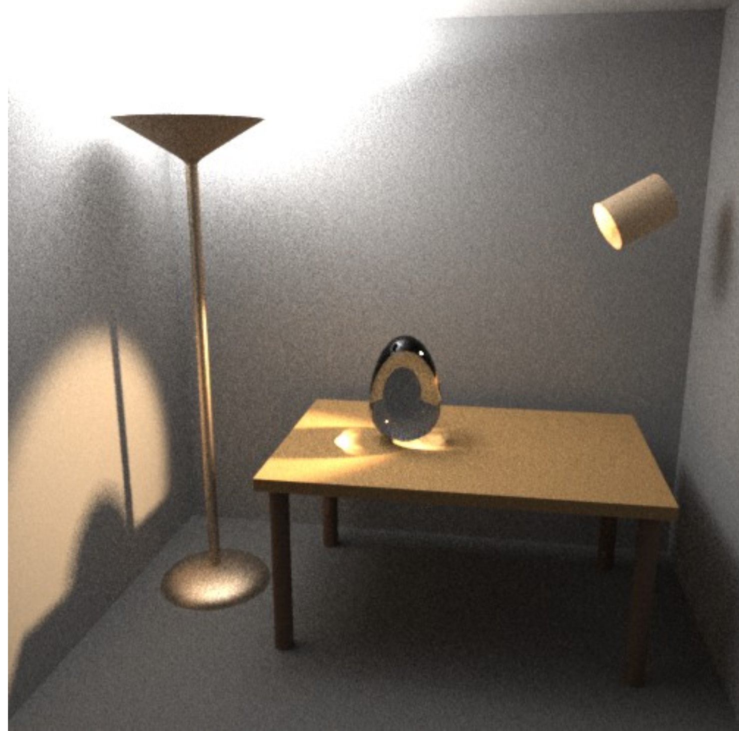
Aalto CS-E5520 Spring 2023
Jaakko Lehtinen

Bridage 2 real-time path tracer

Today

- Path Tracing
  - Intro: nested vs. multidimensional integrals and pixel filtering
  - Recursive sampling of rendering equation using Monte Carlo
  - Direct light sampling
- Bells and whistles





NVIDIA Marbles at Night

Monte Carlo Integration

$$\int_S f(x)\,\mathrm{d}x \;= E\{\frac{f(x)}{p(x)}\}_p$$

- Distribute samples in integration domain S according to probability density function p(x)
- Then integral equals the expected value of f(x)/p(x)

Let’s Go Back to Pixel Filtering

- Remember antialiasing theory from C3100?
- To reduce aliasing, we should ideally...?

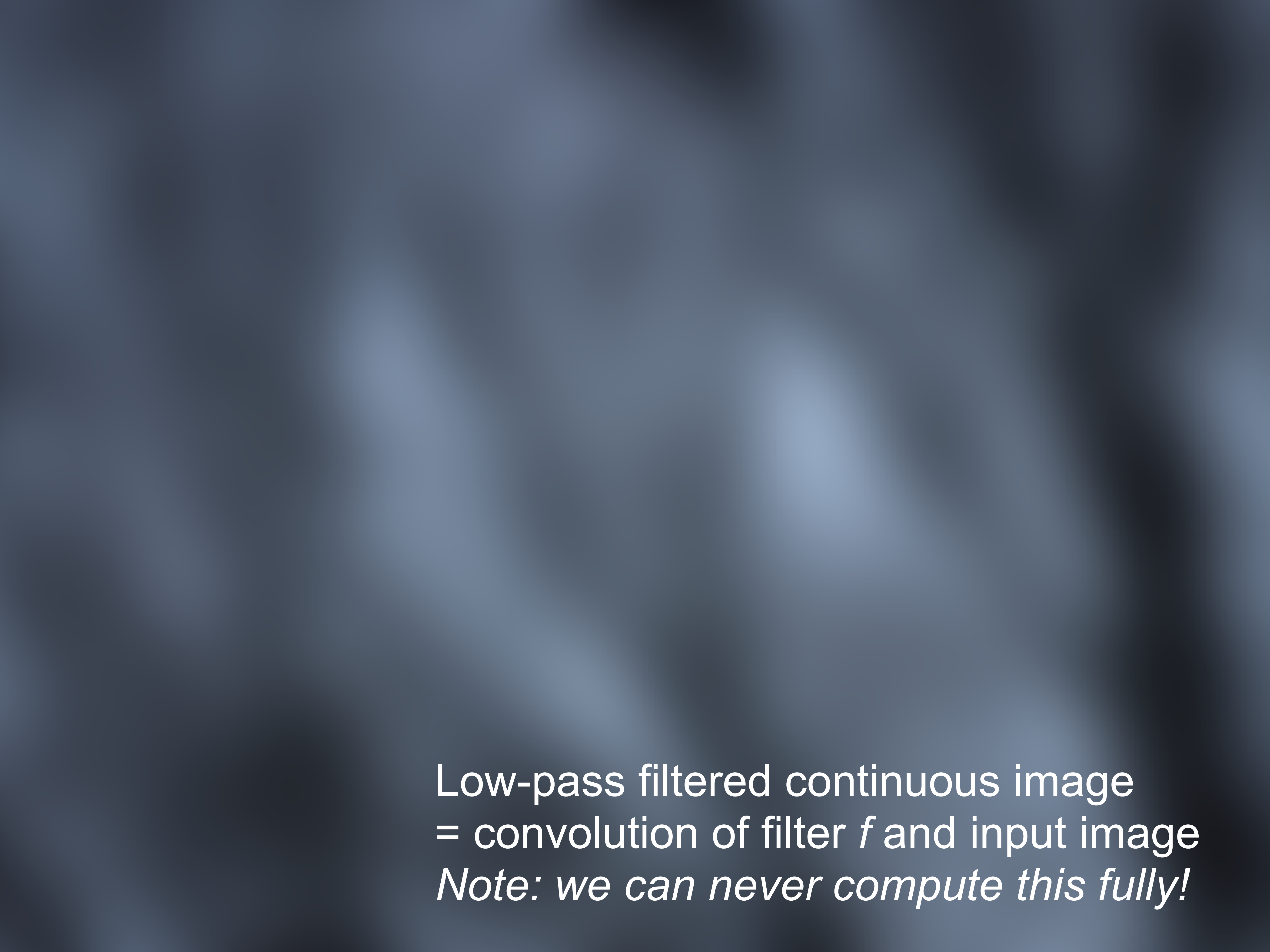
Let’s Go Back to Pixel Filtering

- Remember antialiasing theory from C3100?
- To reduce aliasing, we should ideally…
  1. Low-pass filter the radiance on the image plane before sampling (convolve continuous radiance function + prefilter)
  2. Then sample the low-pass filtered radiance at pixel centers

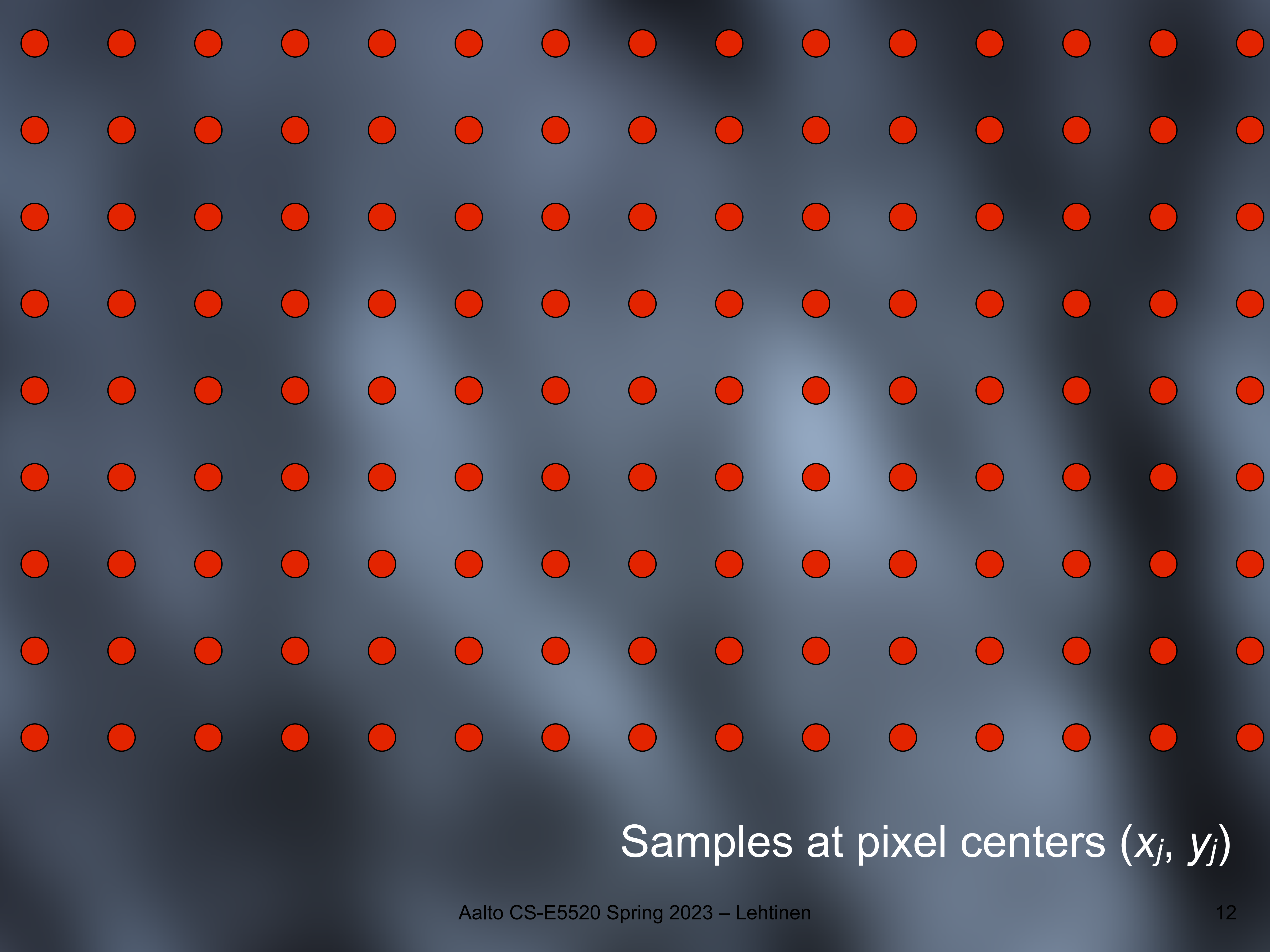
- But we found this was impossible so we turned to supersampling (average many samples in pixel)
  - There is a “proper” way to look at that as well, and here it is..
- (And separate tricks for textures)
  - MIP-maps

Filter $f(x-x_j, y-y_j)$ centered at pixel (x_j, y_j)

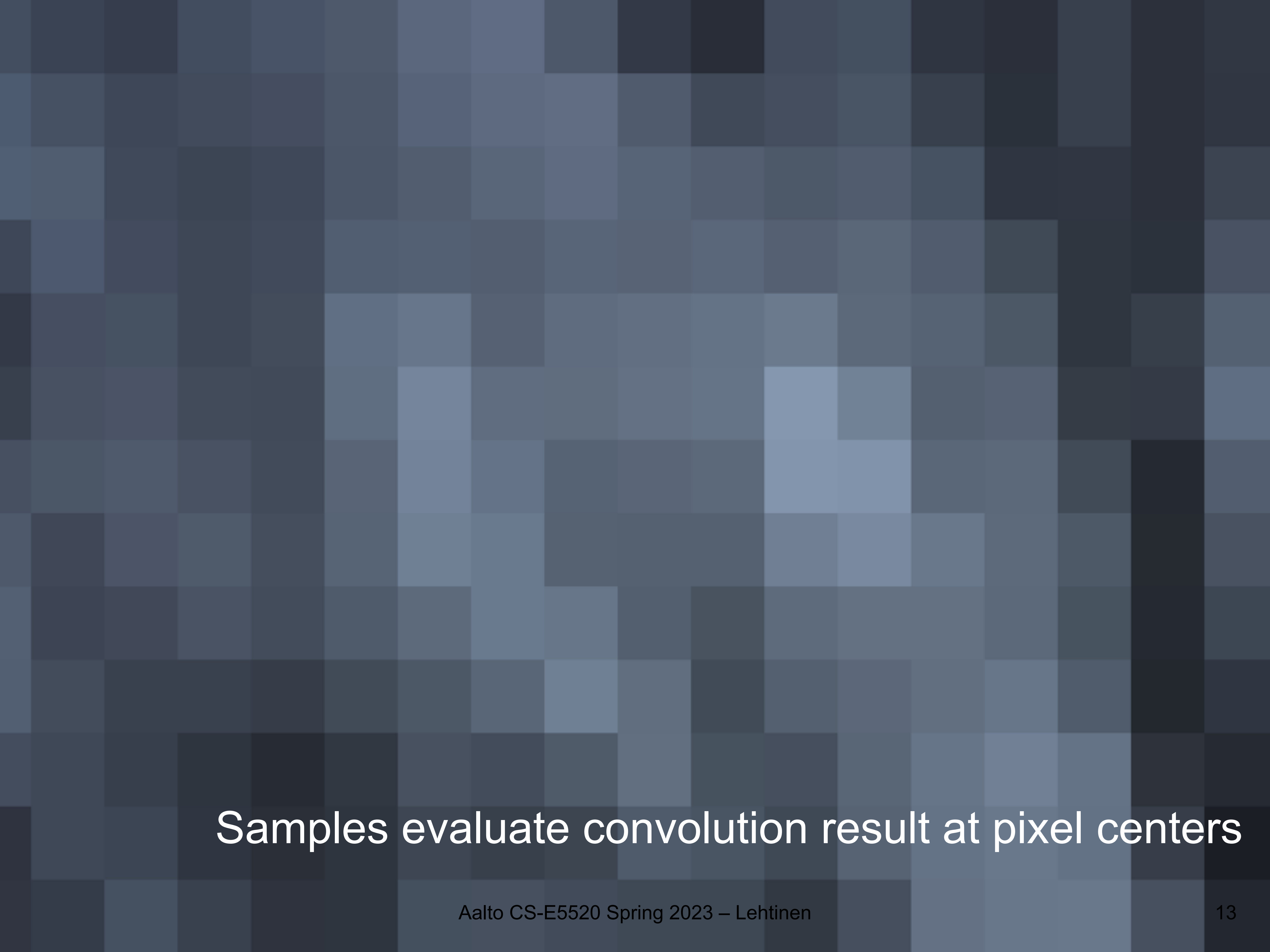
Filter $f(x-x_j, y-y_j)$ centered at pixel (x_j, y_j) times the underlying signal



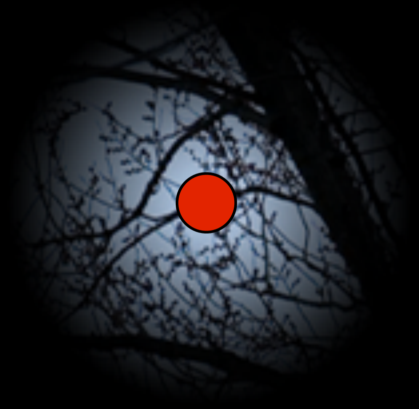
Low-pass filtered continuous image
= convolution of filter *f* and input image
Note: we can never compute this fully!



Samples at pixel centers (x_j, y_j)



Samples evaluate convolution result at pixel centers

$\int$  $\mathrm{d}x\mathrm{d}y$

i.e., value for pixel at (x_j,y_j) is the integral of the filter times the underlying signal

Pixel Filtering

- Prefilter convolution and sampling can be combined:

$$I_j = \int_{\text{screen}} f(x - x_j, y - y_j)\, L(x, y)\, \mathrm{d}x\, \mathrm{d}y$$

- This just means evaluating the result of the continuous convolution at the discrete set of pixel centers
- I_j is the (discrete) intensity/radiance value of *j*th pixel
- Here x_j, y_j are the center of pixel j, f is the *pixel filter*
  - *Yes, it's just a weighted average*

Filter Normalization

$$I_j = \int_{\text{screen}} f(x - x_j, y - y_j)\, L(x, y)\, \mathrm{d}x\, \mathrm{d}y$$

- This only makes sense if filter *f* integrates to 1. *Why?*

Filter Normalization

$$I_j = \int_{\text{screen}} f(x - x_j, y - y_j)\, L(x, y)\, \mathrm{d}x\, \mathrm{d}y$$

- This only makes sense if filter *f* integrates to 1. *Why?*
- Think of a constant radiance function L(x,y) = *C*
  - If *f* integrates to something else than 1, the pixel value will be different from *C* – clearly nonsensical
- So, need to ensure $\int f(x, y)\, \mathrm{d}x\, \mathrm{d}y = 1$

Filter Normalization

- **Important: we do not normalize filters analytically**, but do it numerically instead:

$$I_j = \frac{\int_{\text{screen}} f(x - x_j, y - y_j)\, L(x, y)\, \mathrm{d}x\, \mathrm{d}y}{\int_{\text{screen}} f(x - x_j, y - y_j)\, \mathrm{d}x\, \mathrm{d}y}$$

- Intuitive: when we evaluate the above using MC, we sum the "filter weights" from each sample and divide by the sum in the end
  - Note that 1/N cancels out as it's both above and below
- Why do we do this?

Filter Normalization

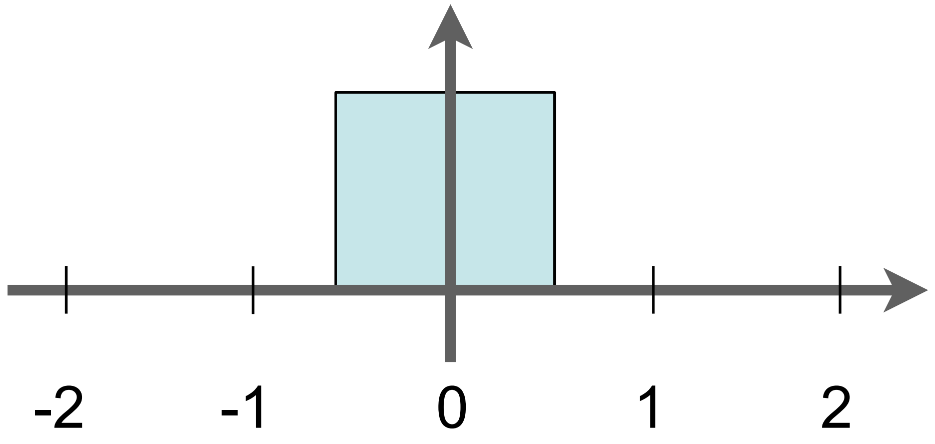
- **Important: we do not normalize filters analytically**, but do it numerically instead:

$$I_j = \frac{\int_{\text{screen}} f(x - x_j, y - y_j)\, L(x, y)\, \mathrm{d}x\, \mathrm{d}y}{\int_{\text{screen}} f(x - x_j, y - y_j)\, \mathrm{d}x\, \mathrm{d}y}$$

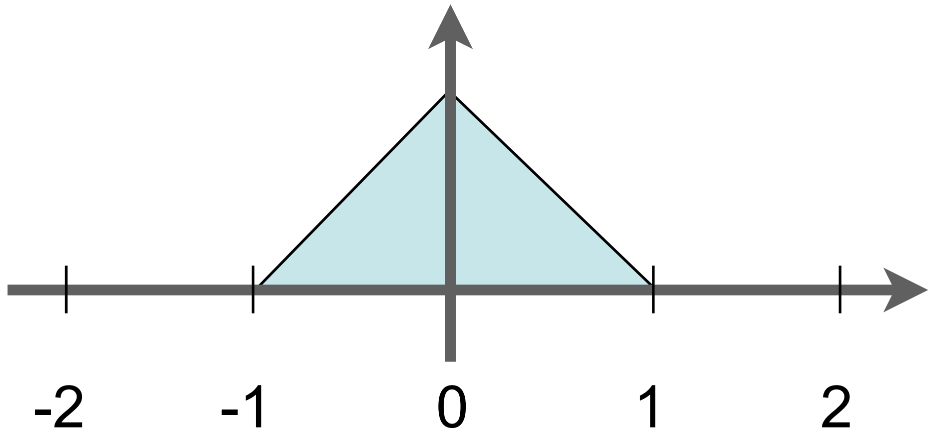
- Why do we do this?
- Again, think of a constant radiance L = *const*.
  - What happens if you use separate random numbers for evaluating the nominator and denominator?

Common Pixel Filters, 1D profiles

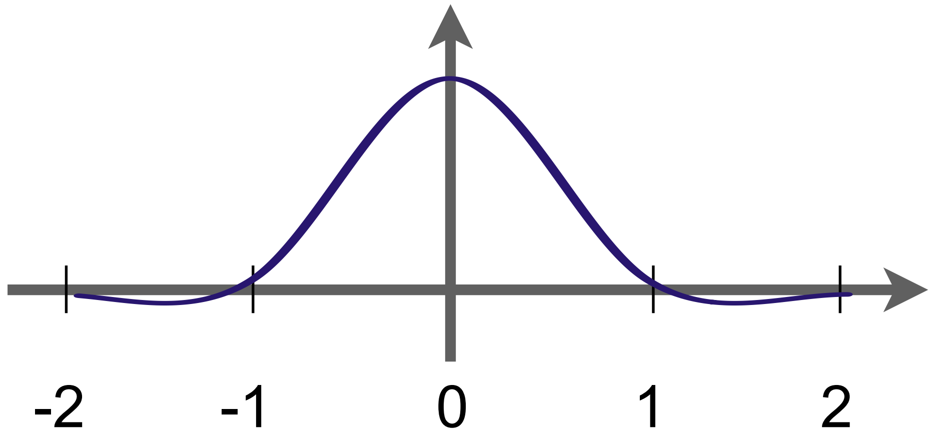
$$f_{\text{box}}(x) = \begin{cases} 1, & -0.5 \le x \le 0.5 \\ 0, & \text{otherwise} \end{cases}$$



$$f_{\text{tent}}(x) = \begin{cases} x+1, & -1 \le x \le 0 \\ 1-x, & 0 \le x \le 1 \\ 0, & \text{otherwise} \end{cases}$$



$$f_{\text{M-N}}(x) = \frac{1}{6} \begin{cases} 7|x|^3 - 12|x|^2 + \frac{16}{3} & |x| < 1 \\ -\frac{7}{3}|x|^3 + 12|x|^2 - 20|x| + \frac{32}{3} & 1 \le |x| \le 2 \\ 0, & \text{otherwise} \end{cases}$$



Mitchell-Netravali filter with A=1/3, B=1/3

Extension to 2D

- "Tensor product" or "separable" filters are constructed from the 1D filters by multiplication

$$f(x,y) = f(x)f(y)$$

- You can also use a non-separable pyramid as a 2D filter, but there seems to be little point
- OK, one more: Gaussian

$$f^{\sigma}_{\text{Gaussian}}(x) = \exp\{-\frac{x^2}{2\sigma^2}\}$$

  – Notes: sigma controls width; not normalized to unit integral!
  – Never drops to zero. We usually cut the filter at 3*sigma or so.

sinc ("perfect" low-pass) is not good

- From Mitchell & Netravali
- "Ringing" = familiar Gibbs phenomenon from basic Fourier series

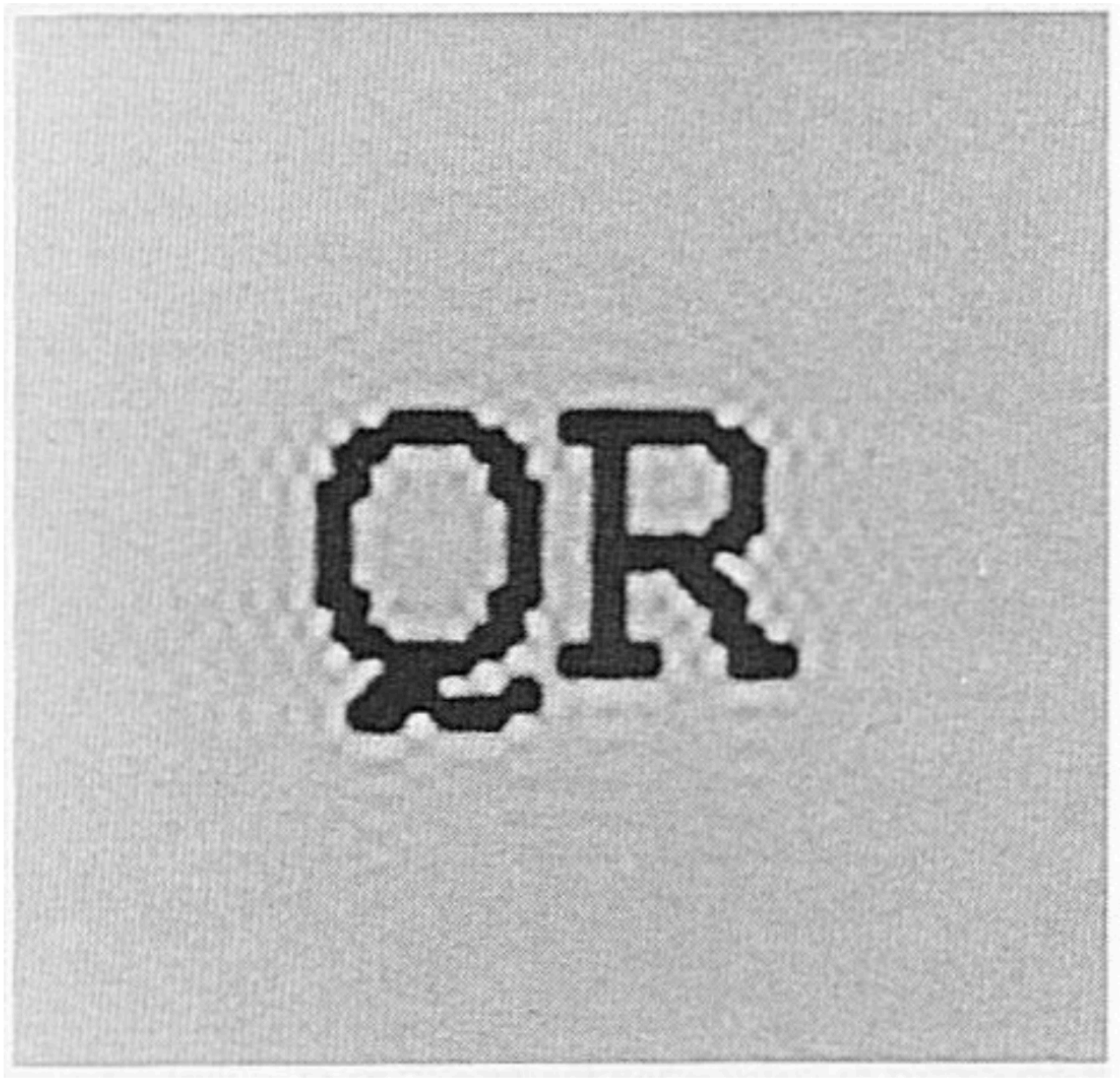


Figure 6. Ringing Caused By Sinc Postfilter

Down to Business: AO

- What if each value of the original image is an integral?
- In assignment 1 you compute, for each primary hit P

$$\int_\Omega V(P,\omega)\cos\theta\,\mathrm{d}\omega$$

  using Monte Carlo integration
  - V is a function that is 1 if the ray of a certain length is unblocked, 0 if it is blocked

Let's Combine Pixel Filter with AO

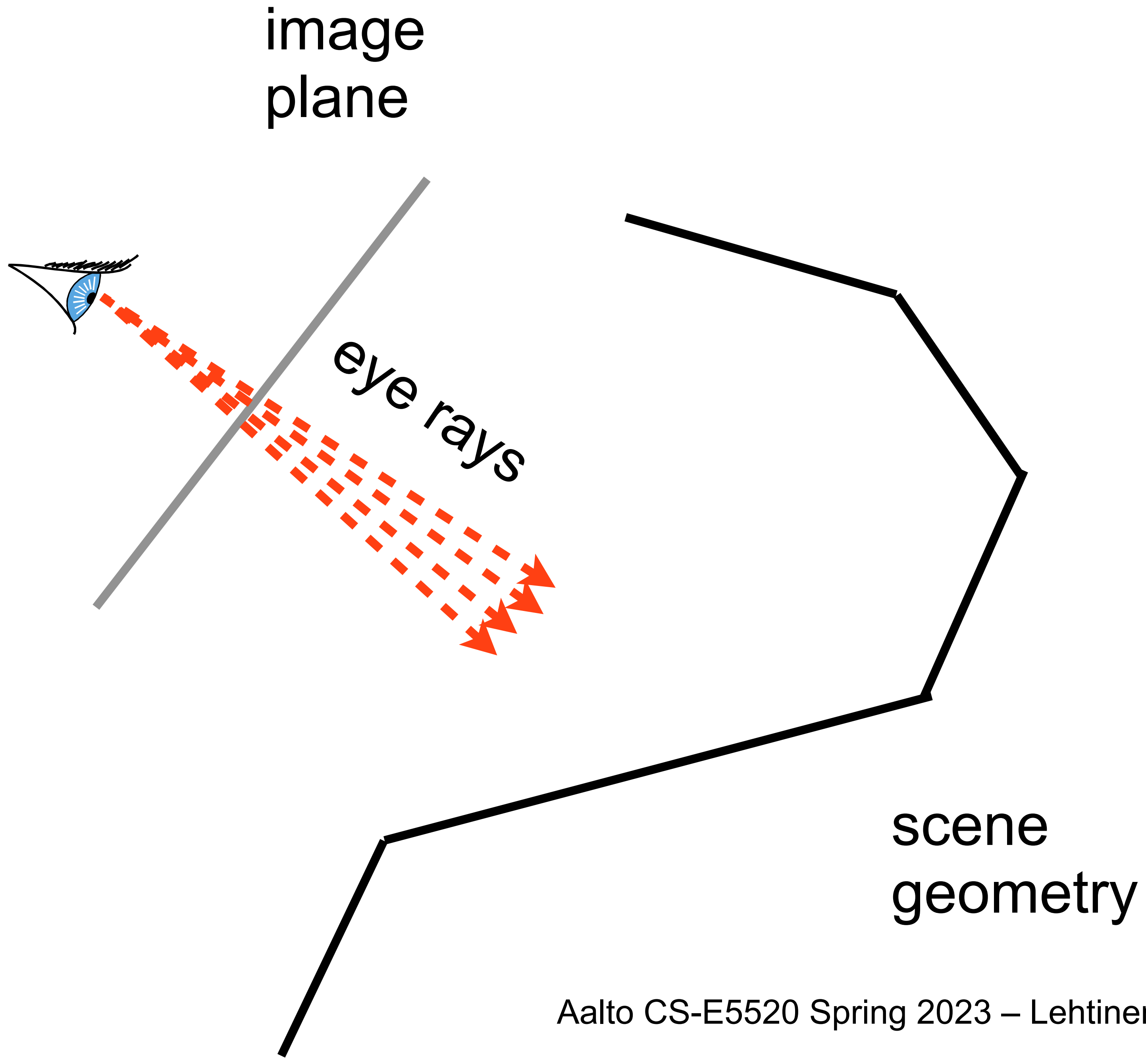
- Each pixel value given by

$$I_j = \int_{\text{screen}} f(x - x_j, y - y_j) \left(\int_\Omega V(P(x,y), \omega) \cos\theta \mathrm{d}\omega \right) \mathrm{d}x \, \mathrm{d}y$$

- (Normalization not shown)
- Two nested 2D integrals
  - Outer one over the screen (2D)
  - Inner one over the hemisphere at the point P hit by ray through image coordinages x,y
    - Again, 2D (hemisphere)

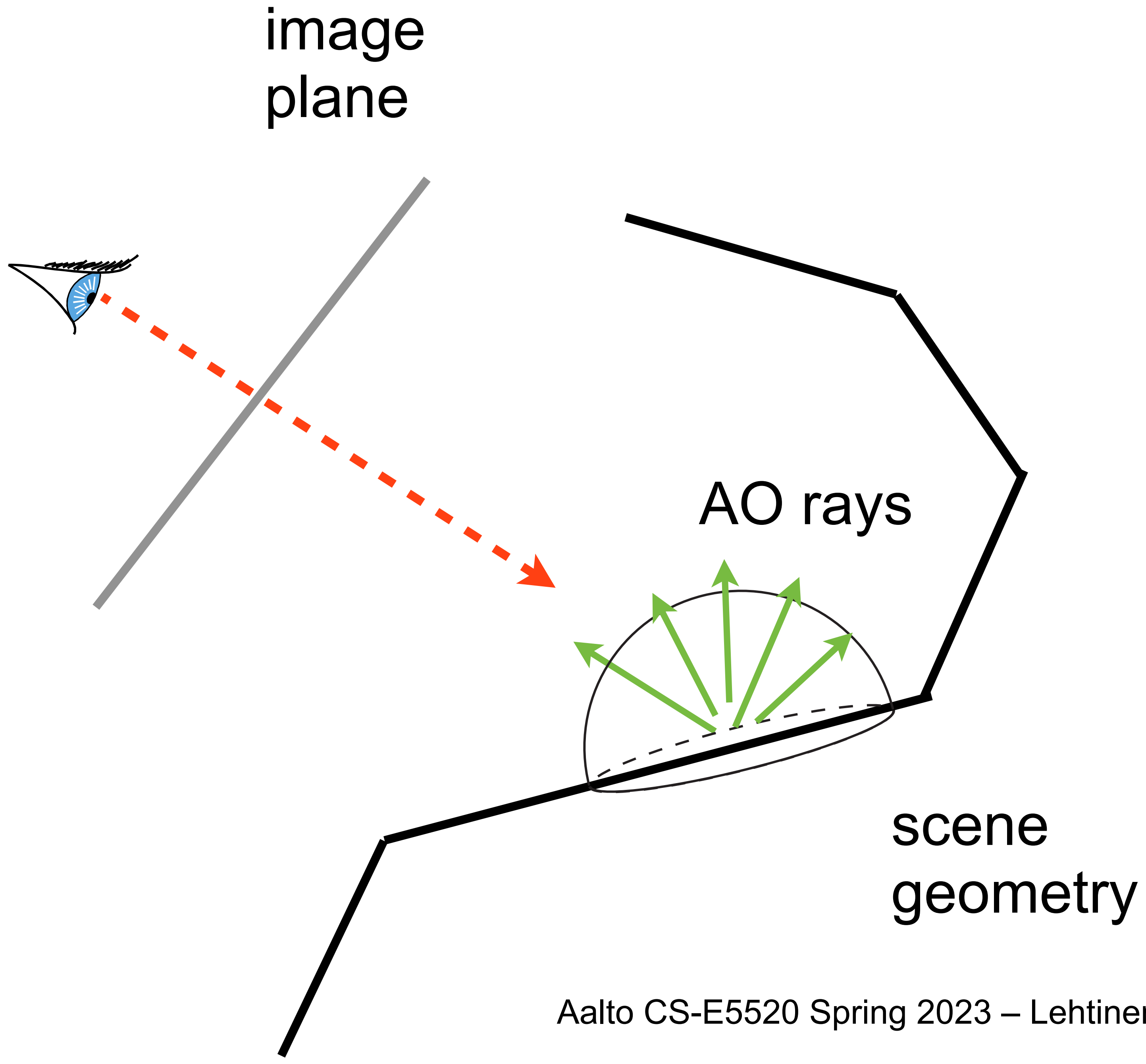
Outer Integral

$$I_j = \int_{\text{screen}} f(x - x_j, y - y_j) \left(\int_\Omega V(P(x,y),\omega) \cos\theta \, \mathrm{d}\omega \right) \mathrm{d}x \, \mathrm{d}y$$



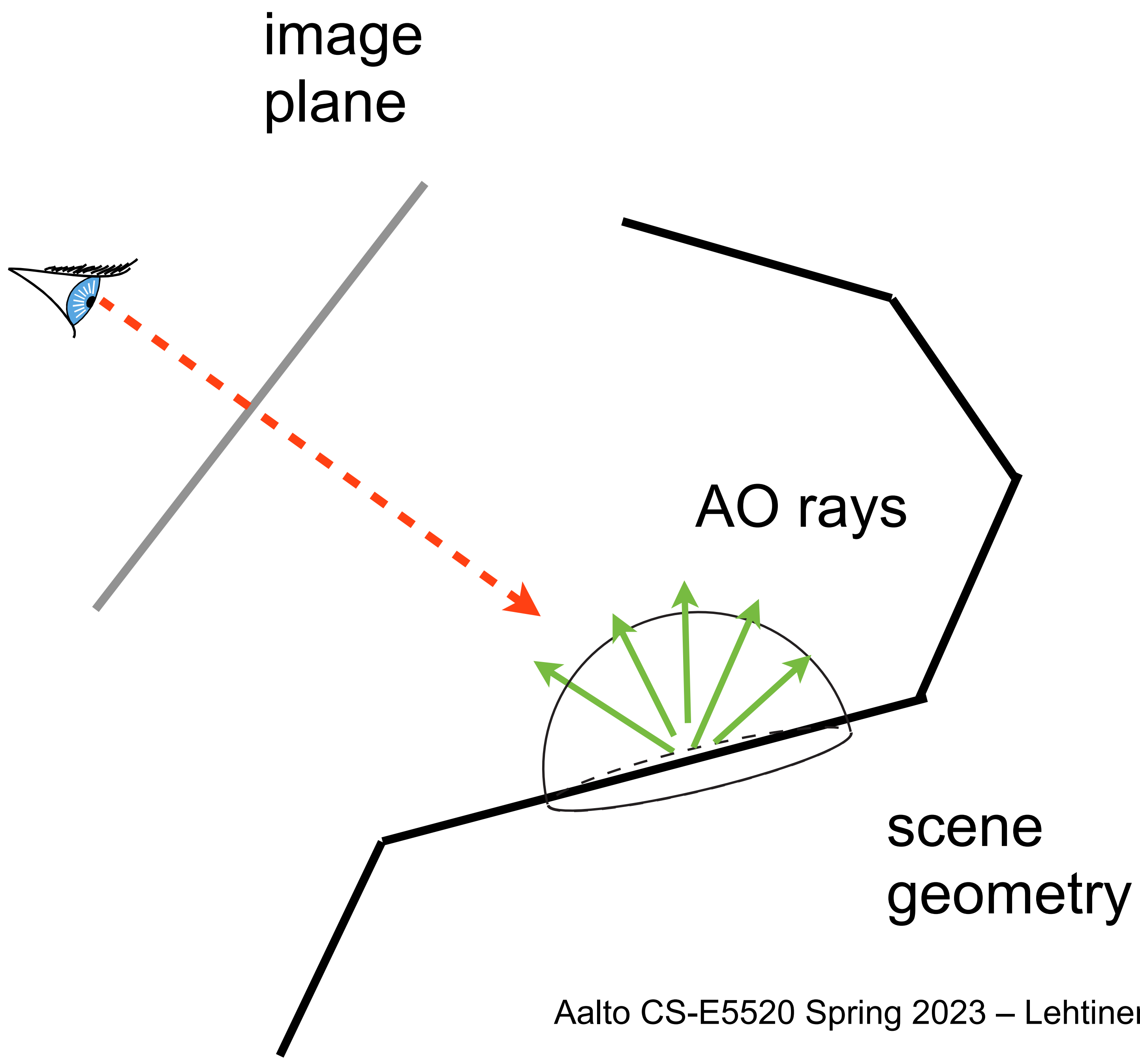
Inner Integral, for each eye ray

$$I_j = \int_{\text{screen}} f(x - x_j, y - y_j) \left(\int_\Omega V(P(x,y),\omega) \cos\theta \mathrm{d}\omega \right) \mathrm{d}x\, \mathrm{d}y$$



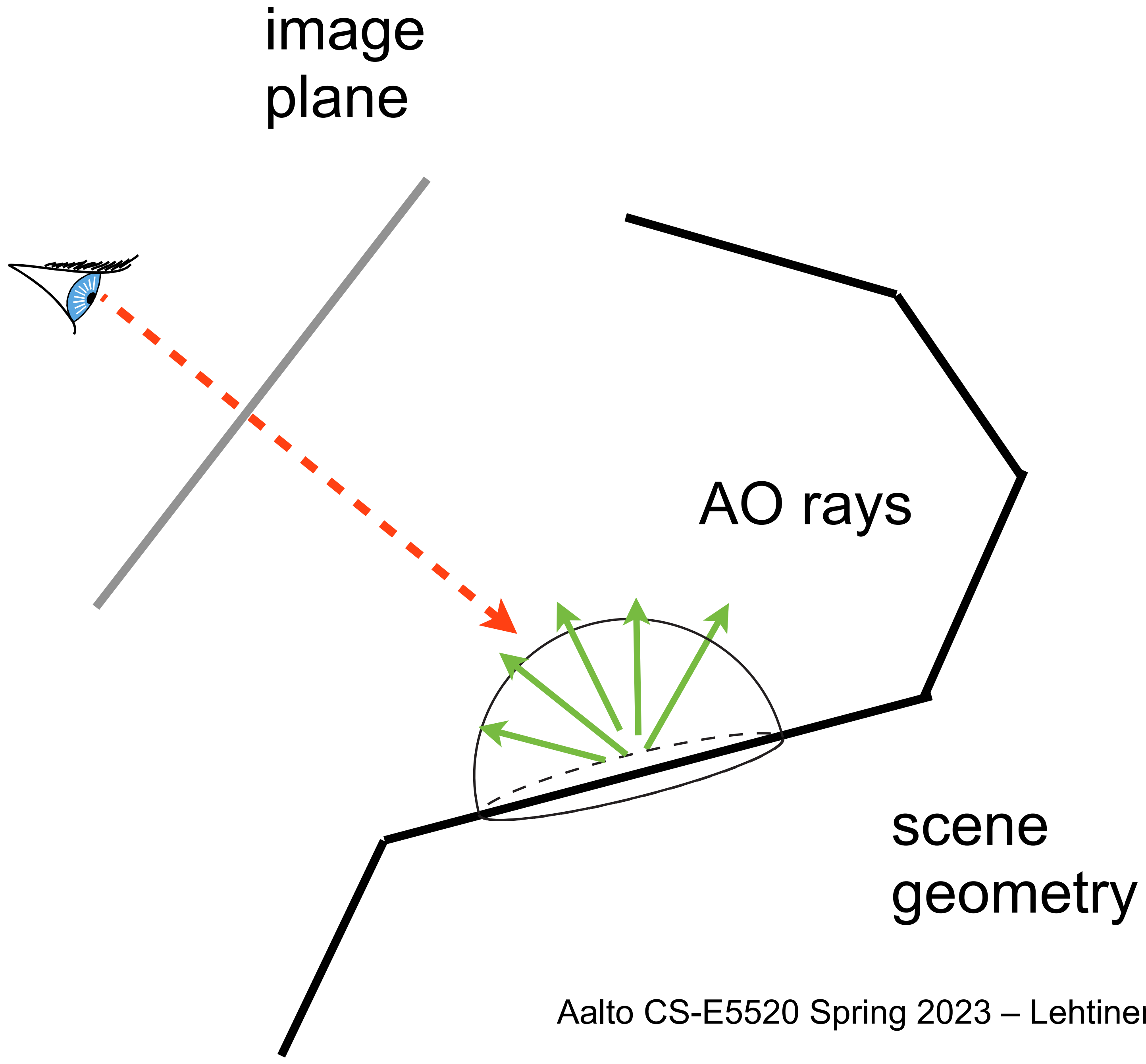
Inner Integral, for each eye ray

$$I_j = \int_{\text{screen}} f(x - x_j, y - y_j) \left(\int_\Omega V(P(x,y),\omega) \cos\theta \, \mathrm{d}\omega \right) \mathrm{d}x \, \mathrm{d}y$$



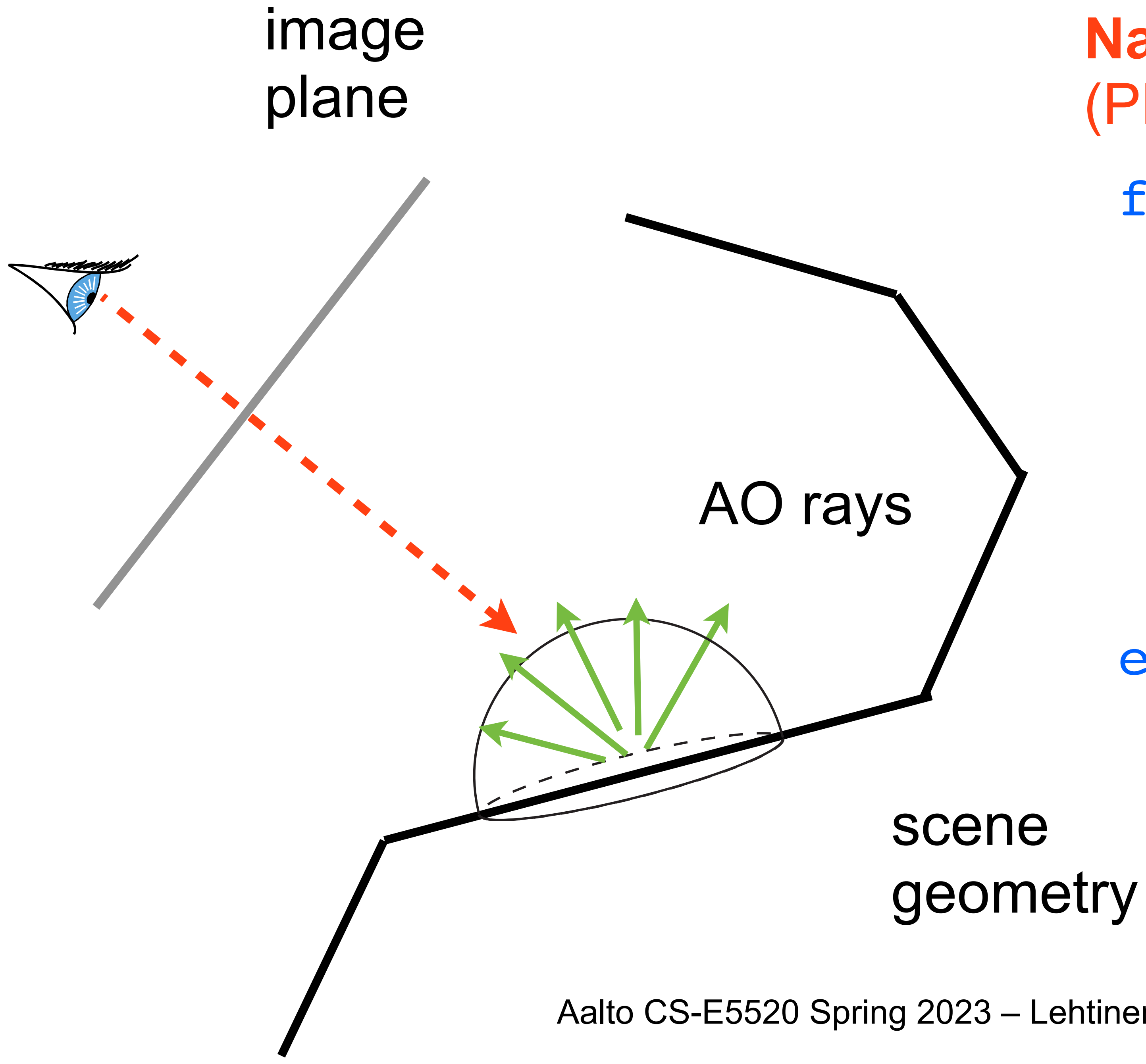
Inner Integral, for each eye ray

$$I_j = \int_{\text{screen}} f(x - x_j, y - y_j) \left(\int_\Omega V(P(x,y),\omega) \cos\theta \, \mathrm{d}\omega \right) \mathrm{d}x \, \mathrm{d}y$$



Inner Integral, for each eye ray

$$I_j = \int_{\text{screen}} f(x - x_j, y - y_j) \left(\int_\Omega V(P(x,y), \omega) \cos\theta \, \mathrm{d}\omega \right) \mathrm{d}x \, \mathrm{d}y$$

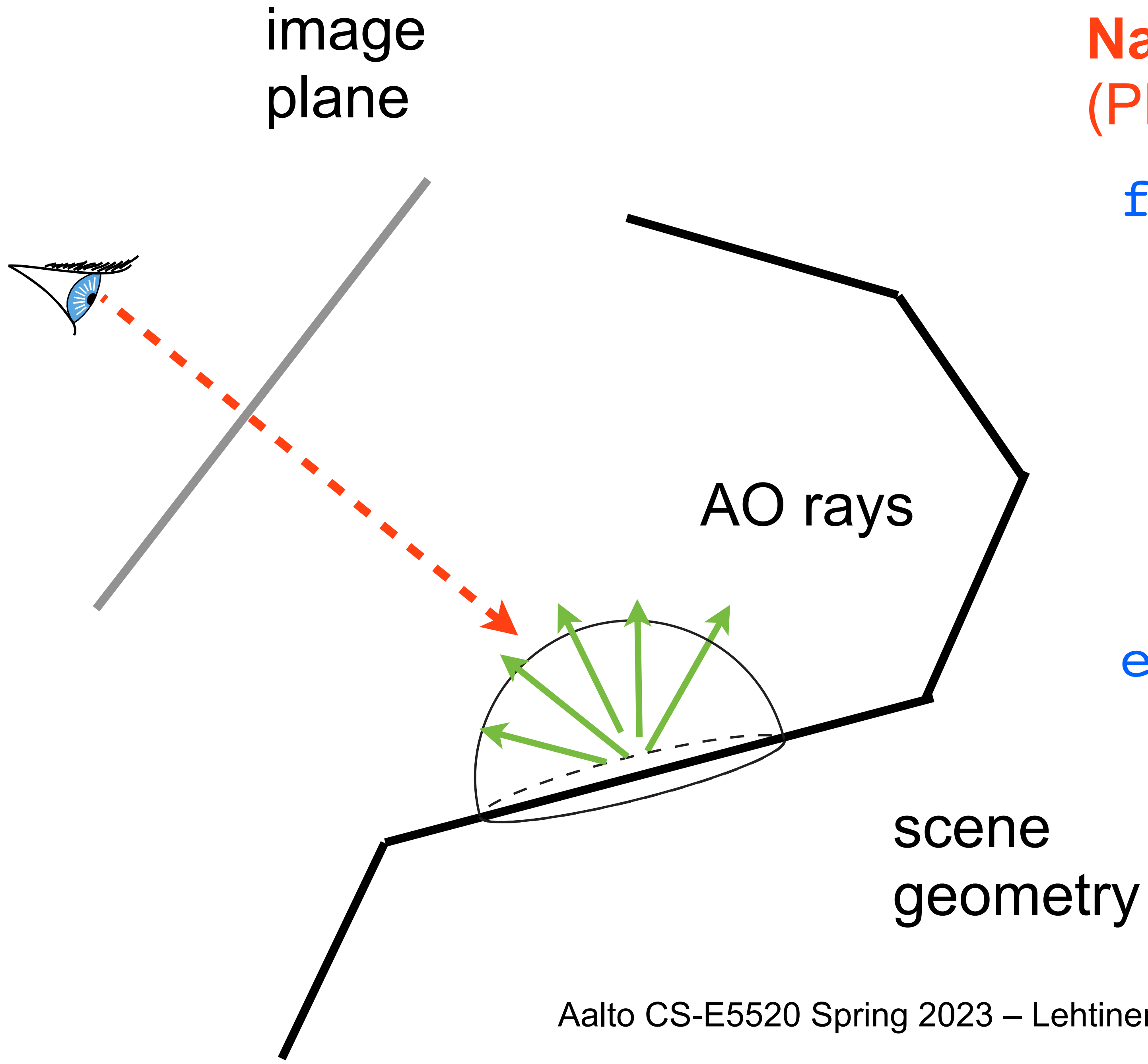


Naive MC implementation:
(PDFs, accumulation not shown)

```
for i=1 to #eyerays
  pick (x,y)
  P=castray(x,y)
  for j=1 to #aorays
    // shoot rays from P
    // etc
  end
end
```

Inner Integral, for each eye ray

$$I_j = \int_{\text{screen}} f(x - x_j, y - y_j) \left(\int_\Omega V(P(x,y),\omega) \cos\theta \mathrm{d}\omega \right) \mathrm{d}x\,\mathrm{d}y$$



Naive MC implementation:
(PDFs, accumulation not shown)

```
for i=1 to #eyerays
  pick (x,y)
  P=castray(x,y)
  for j=1 to #aorays
    // shoot rays from P
    // etc
  end
end
```

Although you do this in assn1, it makes little sense

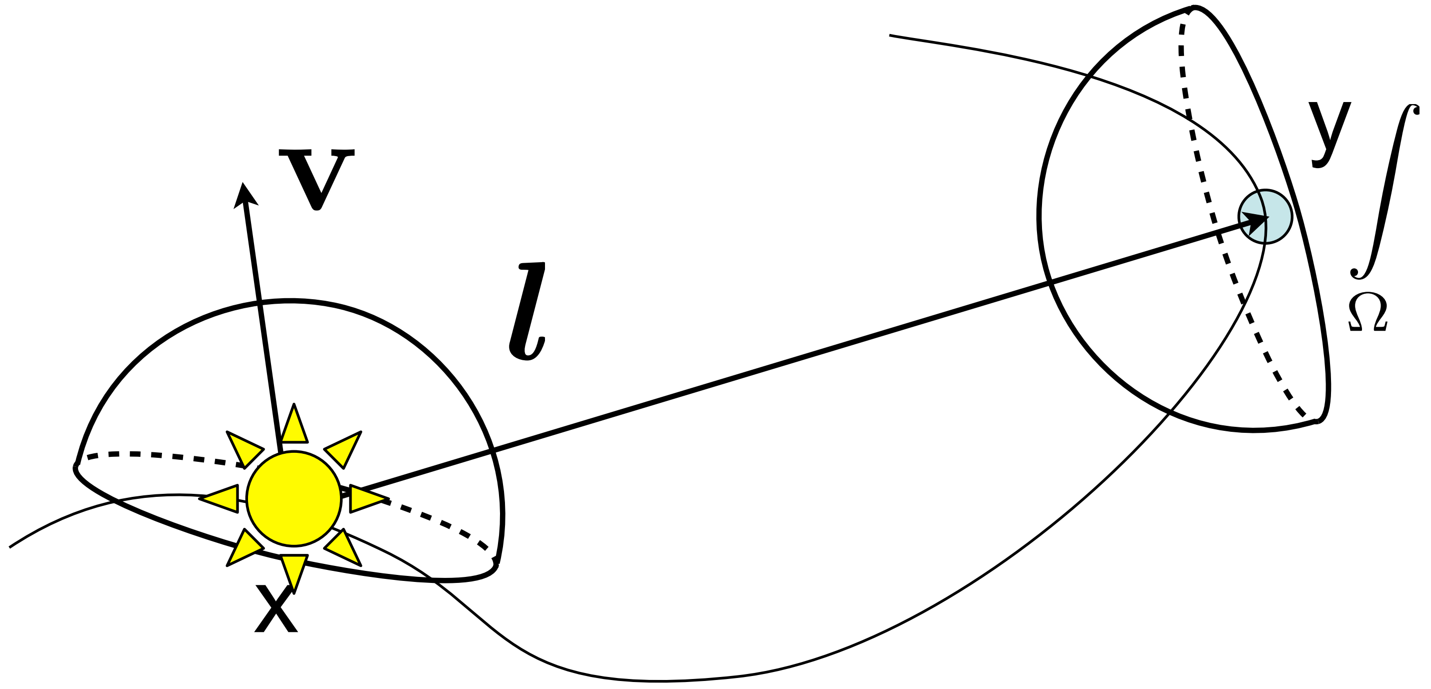
Problems

- Difficult to control number of rays cast in the pixel
  - You have two knobs to tweak
- What if we had even further integrals...?

Recap: Rendering Equation

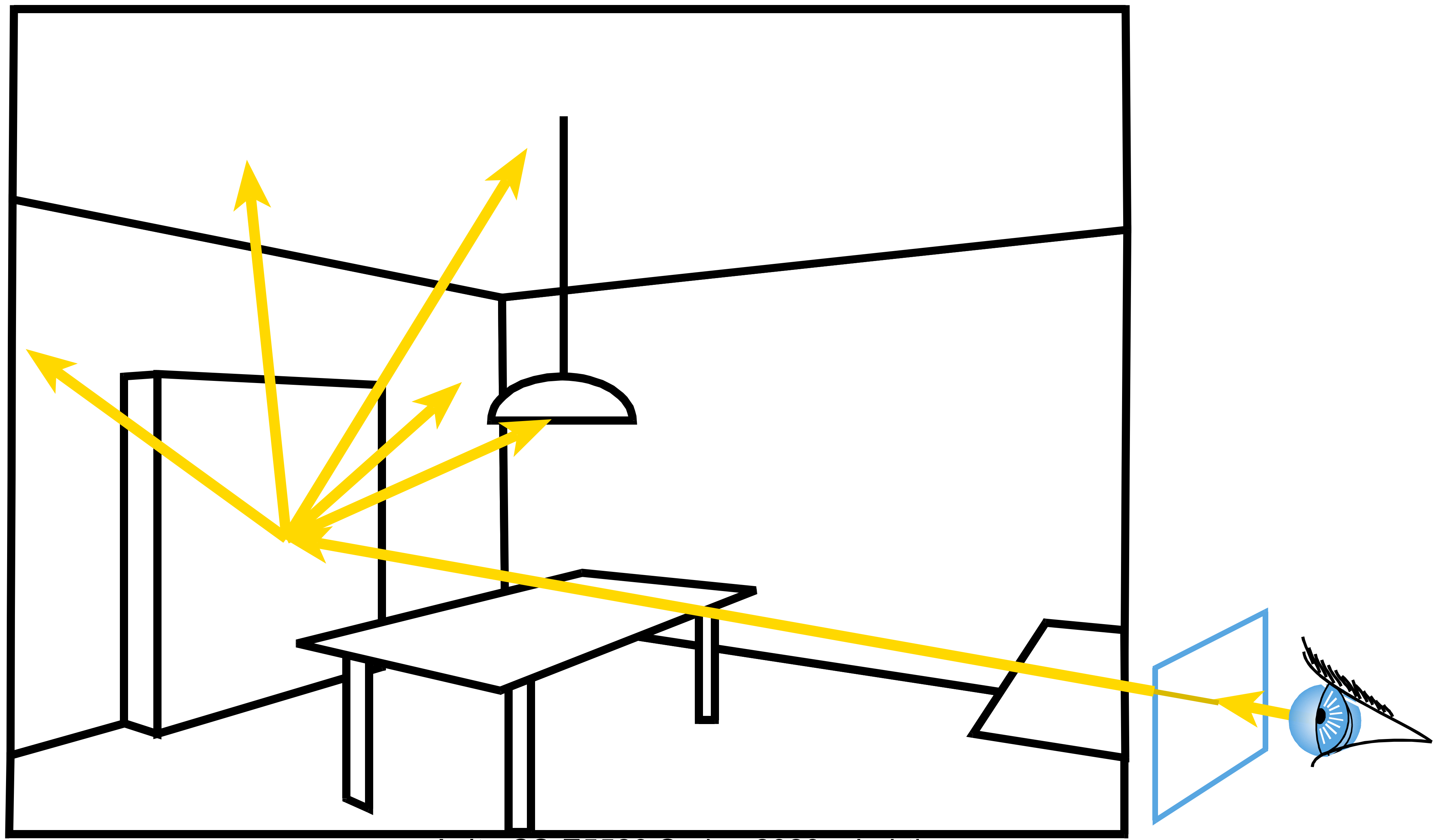
$$L(x \to \mathbf{v}) = \int_\Omega L(x \leftarrow \mathbf{l})\, f_r(x, \mathbf{l} \to \mathbf{v})\, \cos\theta \,\mathrm{d}\mathbf{l}$$
$$+ E(x \to \mathbf{v})$$

to know incoming radiance, must know outgoing radiance elsewhere => recursion!



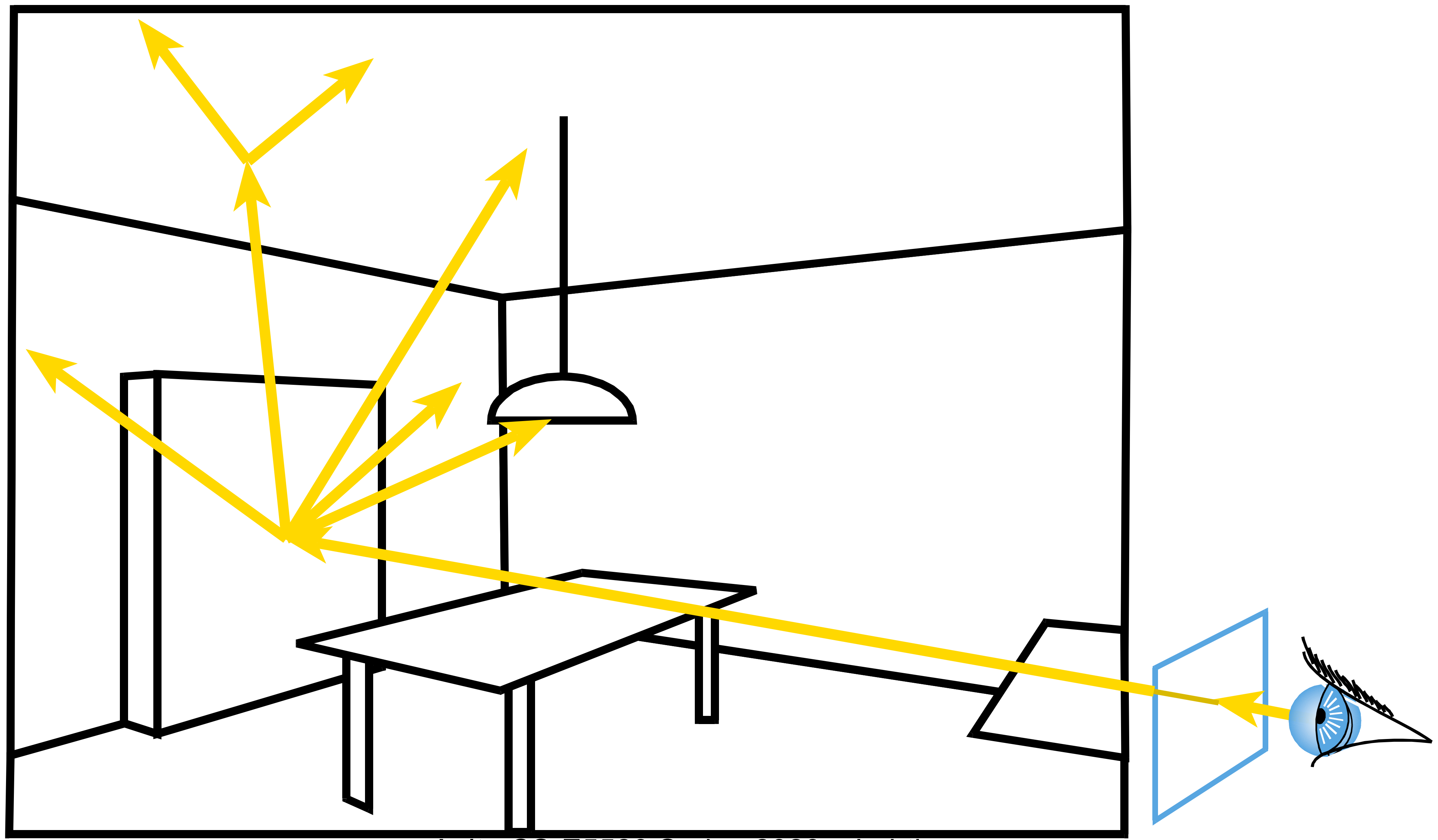
“Monte-Carlo Ray Tracing”

- Cast a ray from the eye through each pixel
- Cast N random rays from the hit point to evaluate hemispherical integral using random sampling



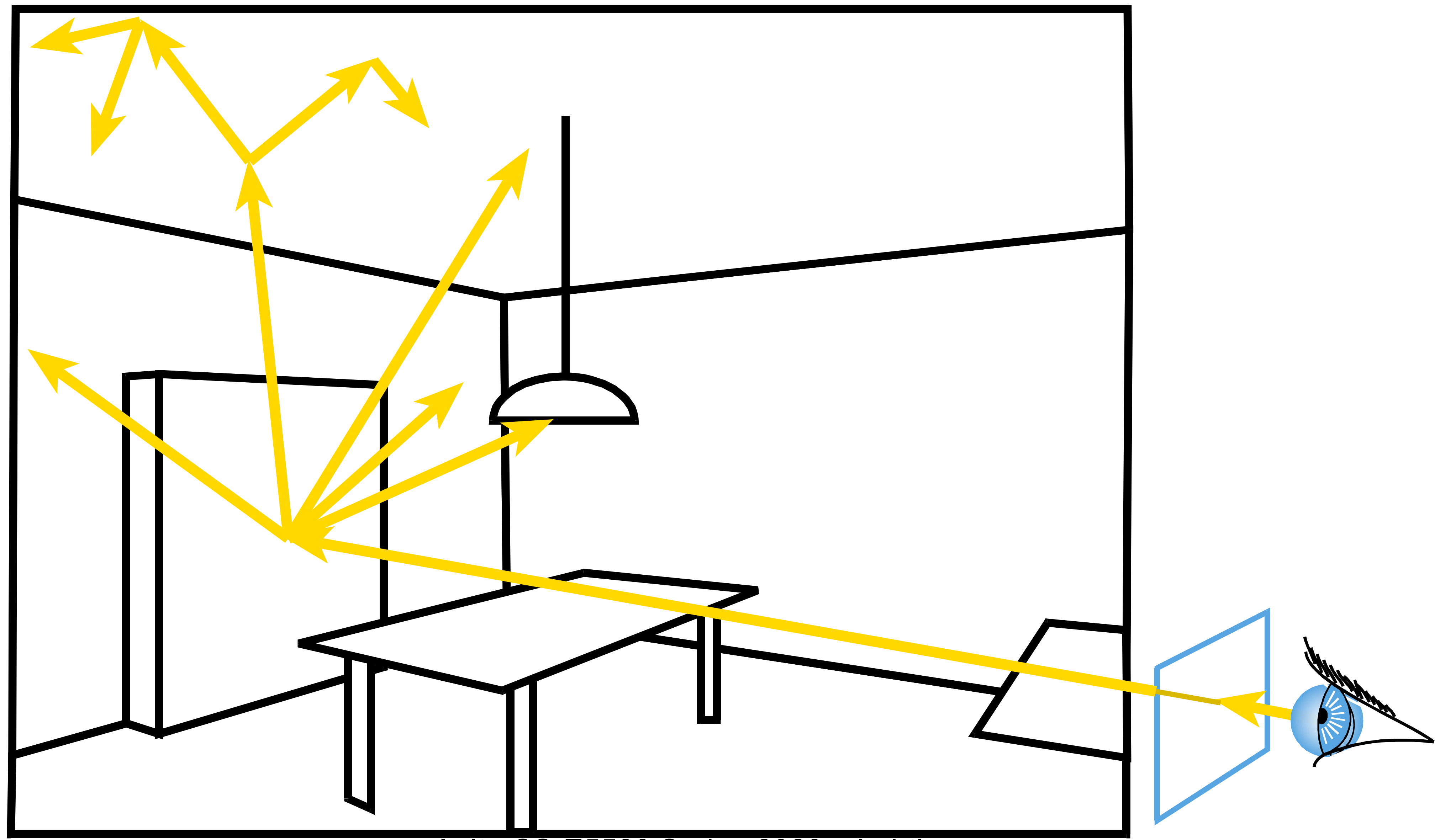
“Monte-Carlo Ray Tracing”

- Cast a ray from the eye through each pixel
- Cast N random rays from the visible point
- Recurse



“Monte-Carlo Ray Tracing”

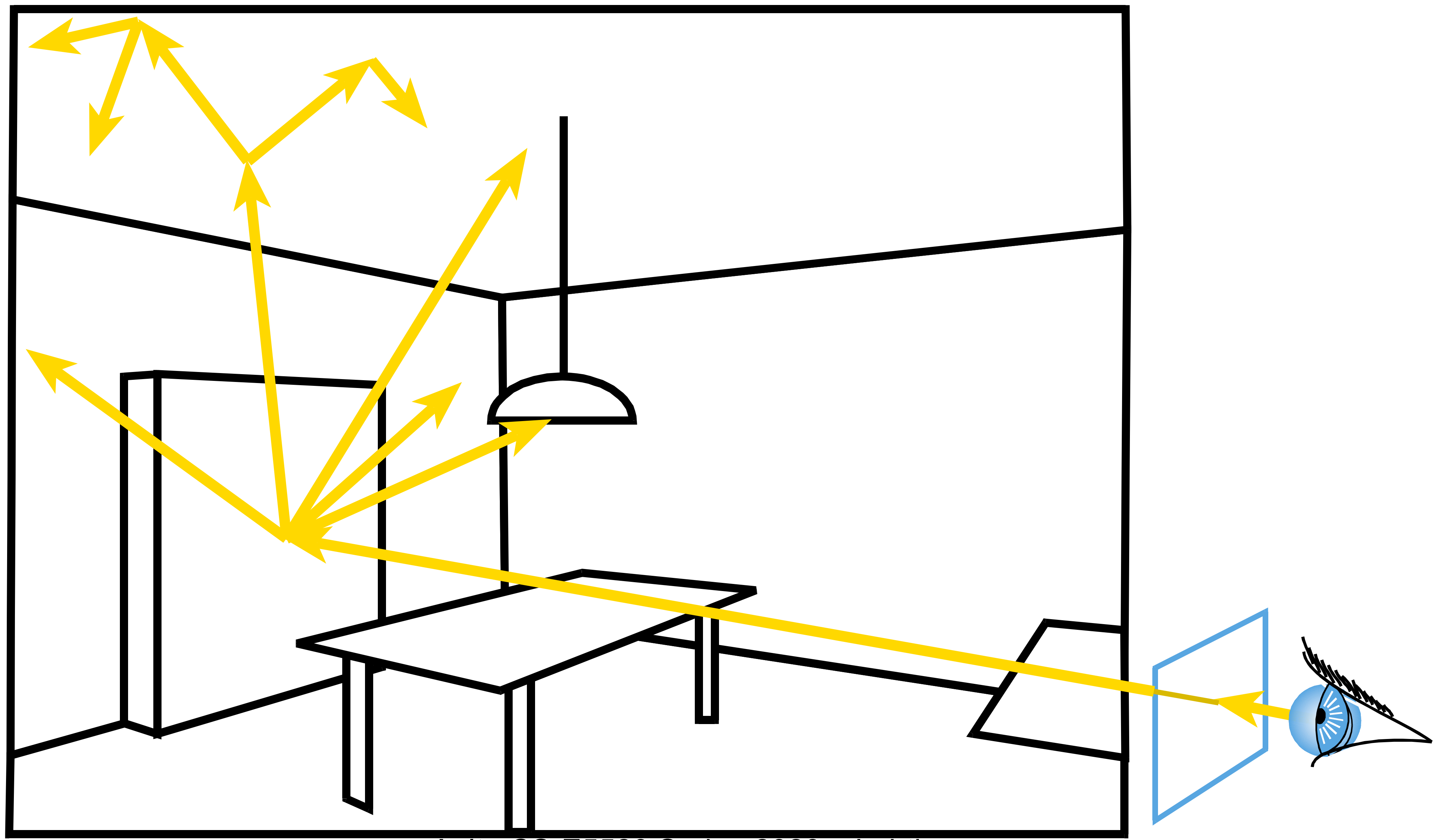
- Cast a ray from the eye through each pixel
- Cast N random rays from the visible point
- Recurse



“Monte-Carlo Ray Tracing”

- Cast a ray from the eye through each pixel
- Cast N random rays from the visible point
- Recurse

Combinatorial explosion!



Combinatorial Explosion

- Sample indirect illumination with 100 rays
- Each ray results in N more rays.. grows exponentially
- For N=100
  - 1 eye ray
  - 100 indirect rays at primary hit
  - 10 000 indirect rays at the secondary hits
  - 1 000 000 at the tertiary hits
  - You get the picture

Back to AO: Better Way

- Rather than 2D x 2D, one integral over 4D domain:

$$I_j = \int_{\text{screen}\times\Omega} g(x, y, \omega)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}\omega$$

with integrand

$$g(x, y, \omega) = f(x - x_j, y - y_j)\,V(P(x, y), \omega)\,\cos\theta$$

Back to AO: Better Way

- Rather than 2D x 2D, one integral over 4D domain:

$$I_j = \int_{\mathrm{screen}\times\Omega} g(x,y,\omega)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}\omega$$

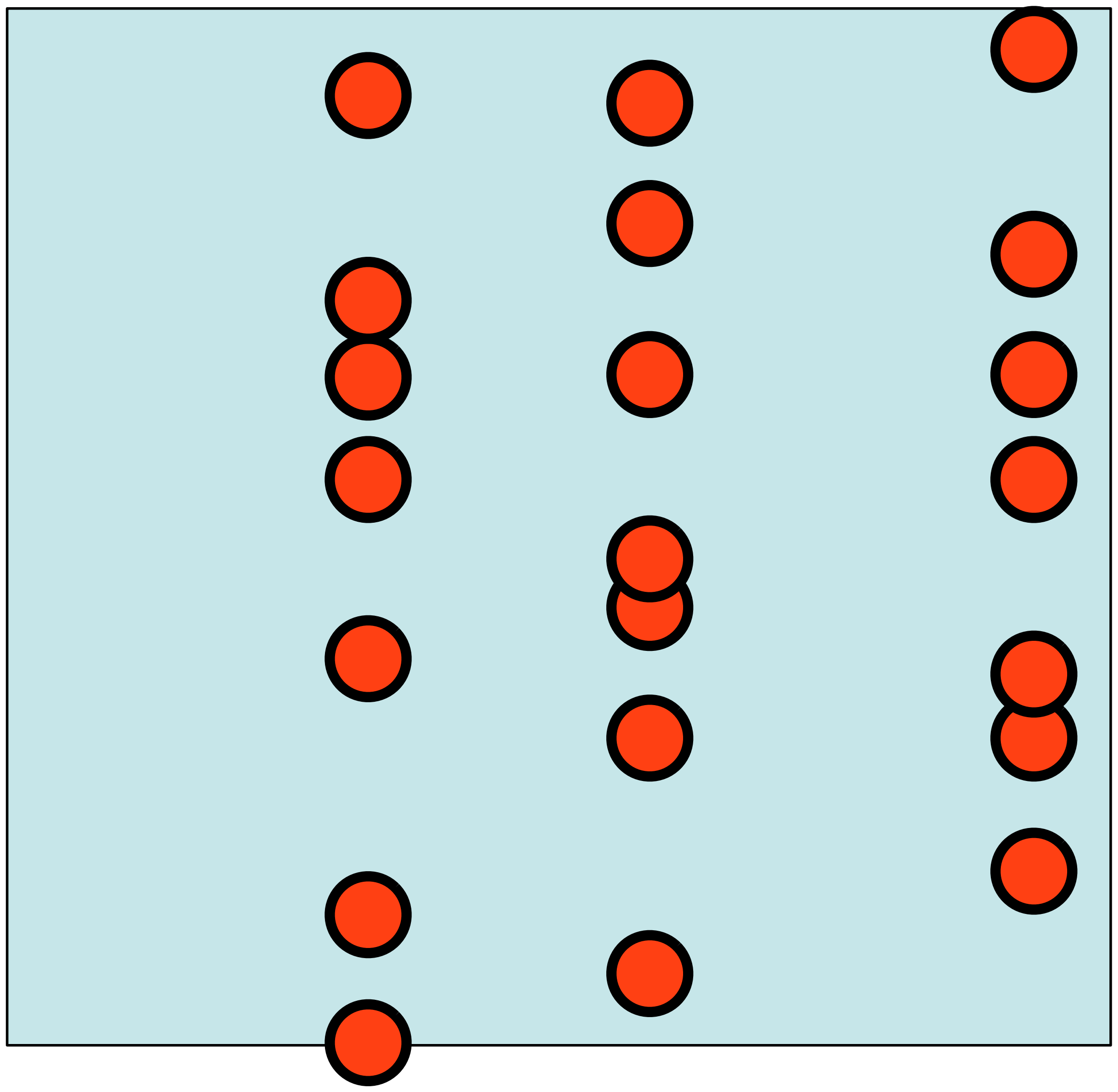
with integrand

$$g(x,y,\omega) = f(x-x_j, y-y_j)\,V(P(x,y),\omega)\,\cos\theta$$

- This is strictly equivalent; just another point of view
  - *Think of 1D vs. 2D integrals*

Nested 1D + 1D, naive

$$\int \left(\int f(x,y) \mathrm{d}y \right) \mathrm{d}x$$

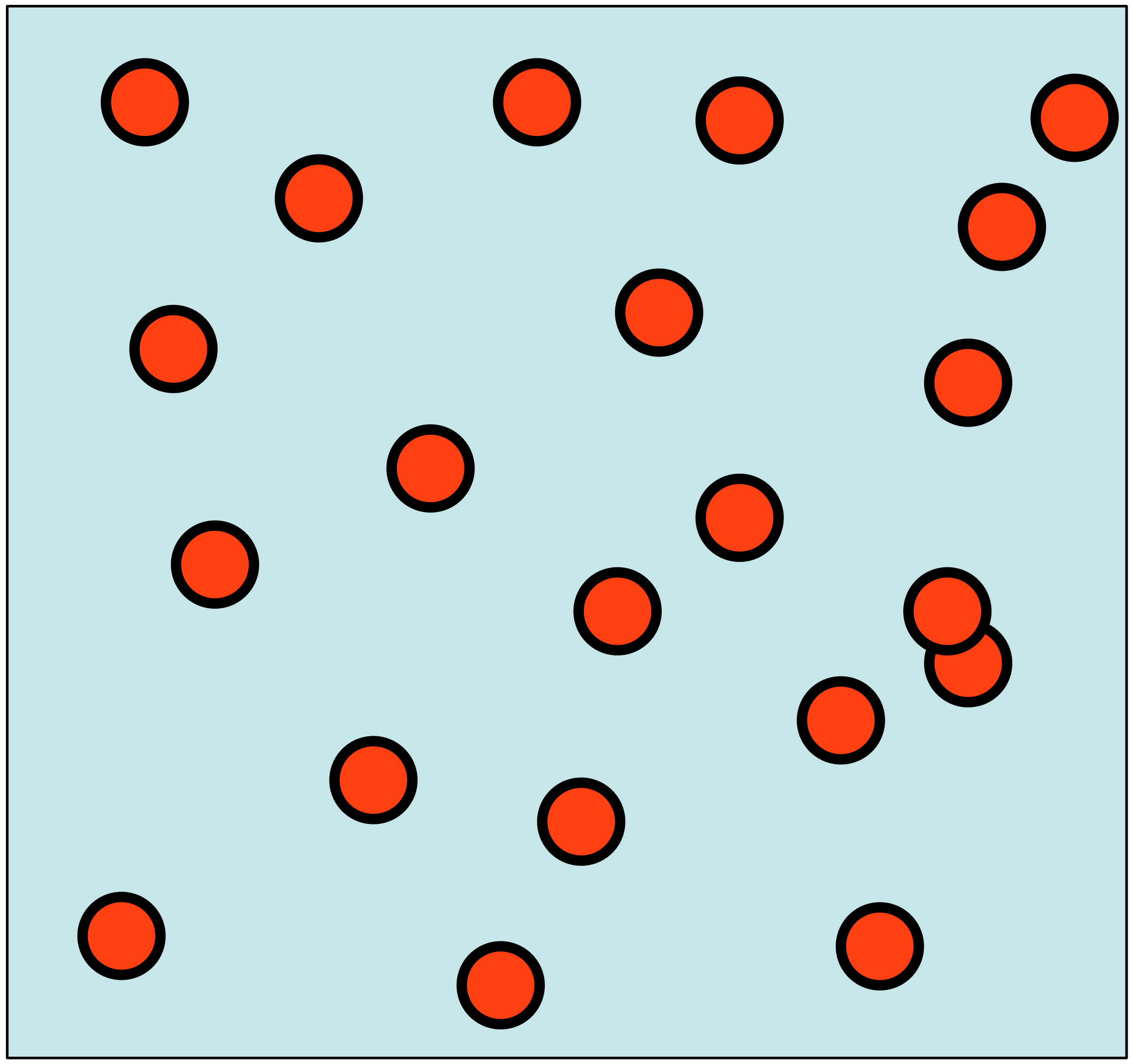


First pick x, then pick a bunch of ys

Repeat

Nested 1D + 1D, treat as 2D

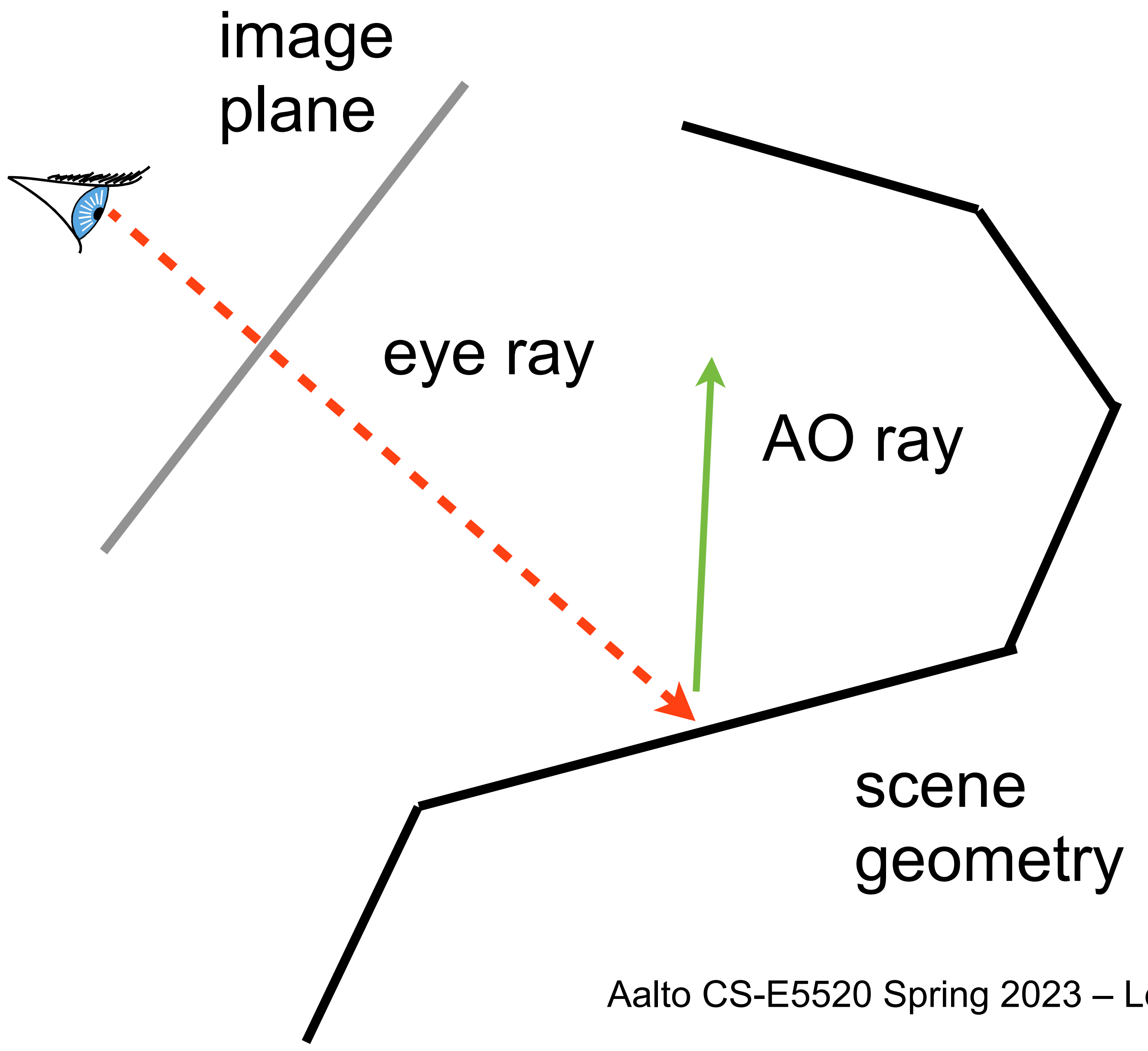
$$\int \left(\int f(x,y) \mathrm{d}y \right) \mathrm{d}x = \int \int f(x,y) \, \mathrm{d}x \mathrm{d}y$$



Draw 2D samples (x,y) from 2D pdf

Visually: One sample is Two Rays

$$I_j = \int_{\mathrm{screen} \times \Omega} g(x, y, \omega) \, \mathrm{d}x \, \mathrm{d}y \, \mathrm{d}\omega$$

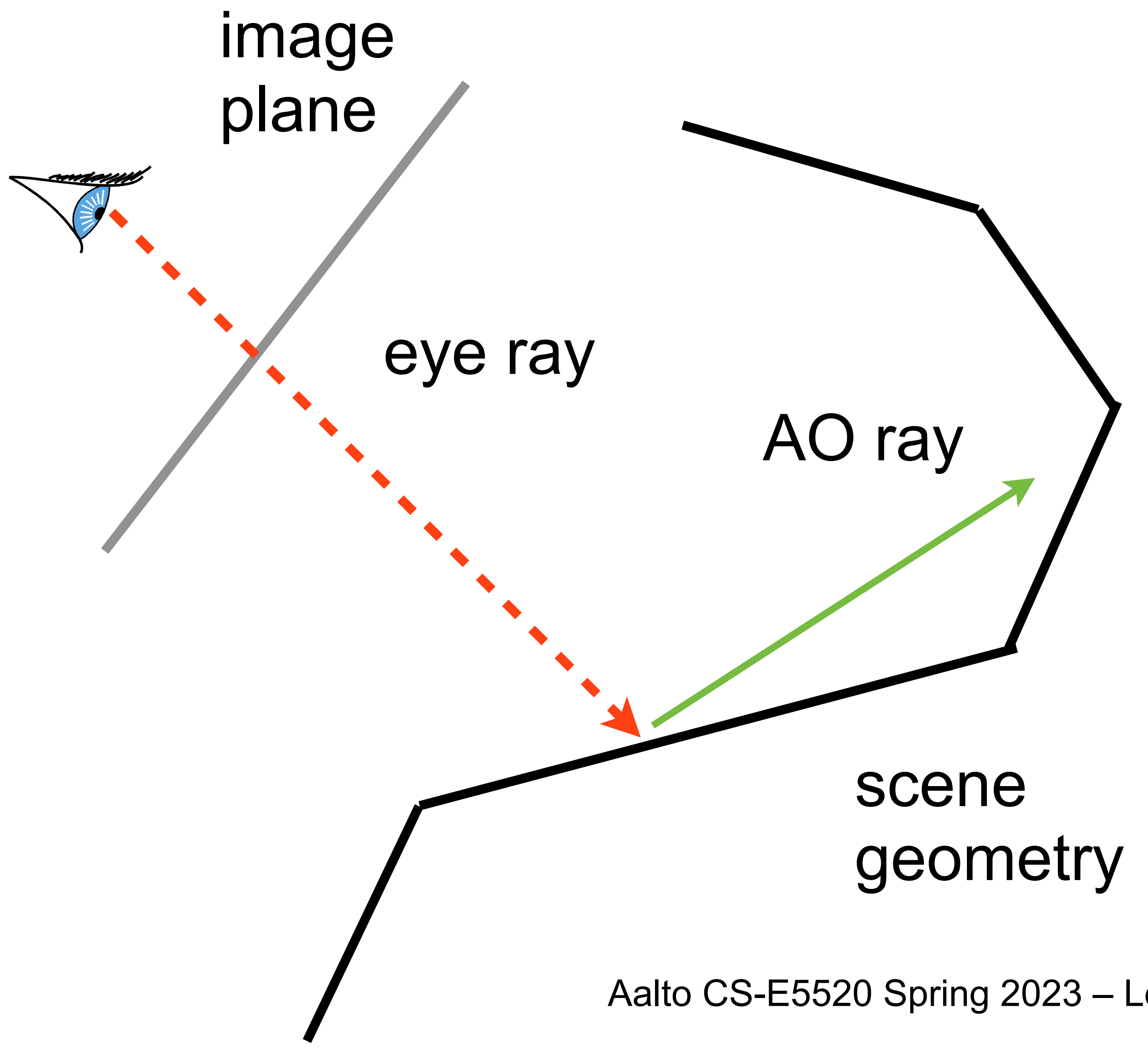


Better MC implementation:

```
res = 0
for i=1 to #samples
  pick sample (x,y,w_out)
  pdf=p(x,y)*p(w_out)
  P=castray(x,y)
  V=castray(P,w_out)
  res += g(x,y,w_out)/pdf
end
res = res/#samples
```

Visually: One sample is Two Rays

$$I_j = \int_{\text{screen}\times\Omega} g(x,y,\omega)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}\omega$$



Better MC implementation:

```
res = 0
for i=1 to #samples
  pick sample (x,y,w_out)
  pdf=p(x,y)*p(w_out)
  P=castray(x,y)
  V=castray(P,w_out)
  res += g(x,y,w_out)/pdf
end
res = res/#samples
```

Implementation Details

- Naturally, if your pixel filters overlap, you use the same samples for updating all the pixels with nonzero filter responses

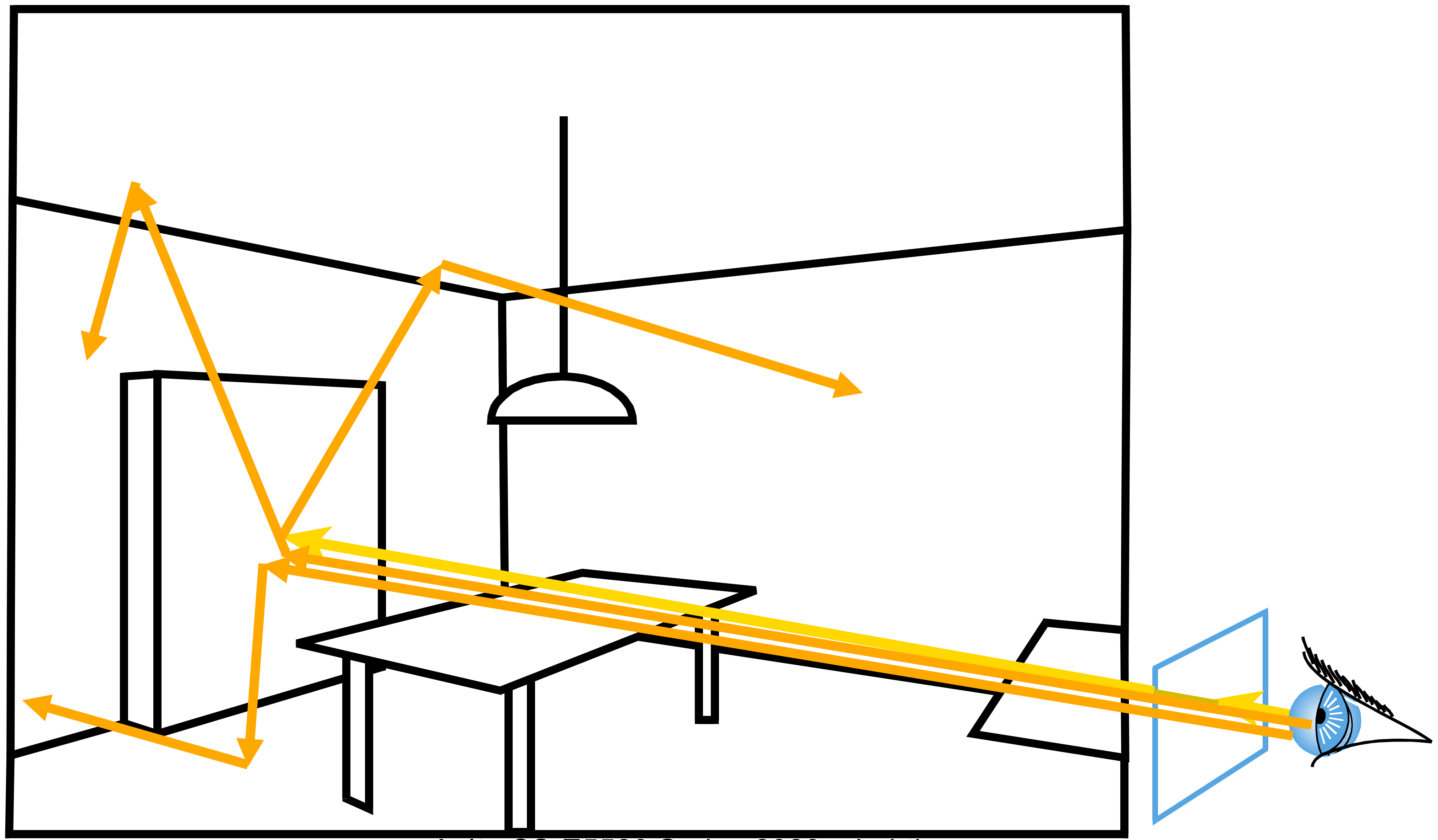
```
res[k] = weight[k] = 0 for all pixels k
for each pixel k
 for i=1 to #samplesperpixel
  pick sample (x,y,omega)           // e.g. 4D Sobol’
  pdf=p(x,y)*p(omega)               // usually p(x,y) == 1
  P=castray(x,y)                    // find primary hit
  V=castray(P,omega).length()>D     // evaluate AO shadow term
  for each pixel j where f_j(x,y) is nonzero
   res[j] += f_j(x,y)*cos(theta)*V/pdf
   weight[j] += f_j(x,y)/p(x,y)
  end
 end
end
res[k] = res[k]/weight[k]
```

Filter of *j*th pixel

$$f_j(x,y) = f(x - x_j, y - y_j)$$

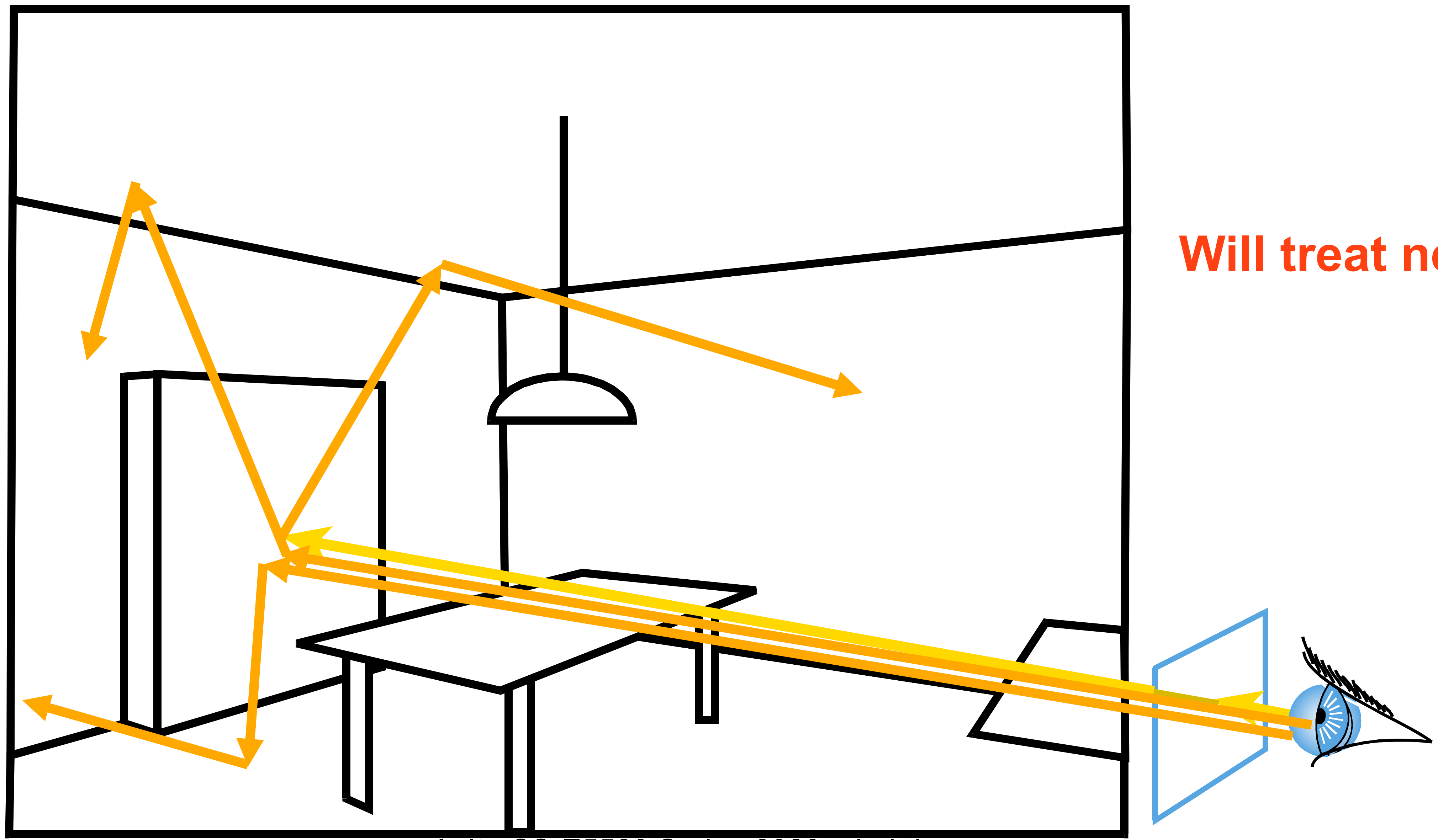
Monte Carlo Path Tracing

- Trace only one secondary ray per recursion
  - Otherwise number of rays explodes!
- But send many primary rays per pixel (antialiasing)



Monte Carlo Path Tracing

- Trace only one secondary ray per recursion
  - Otherwise number of rays explodes!
- But send many primary rays per pixel (antialiasing)



Will treat next time!