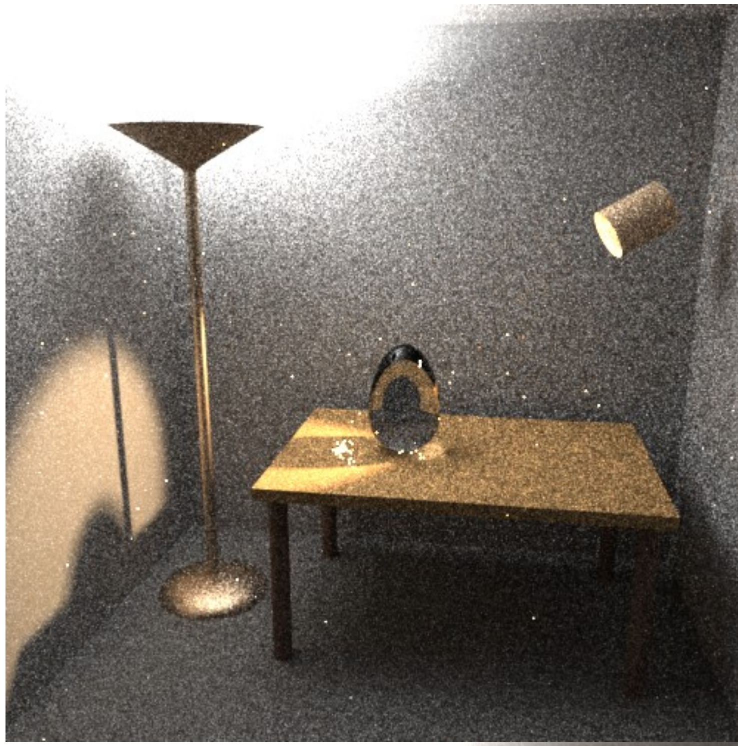
Path Tracing

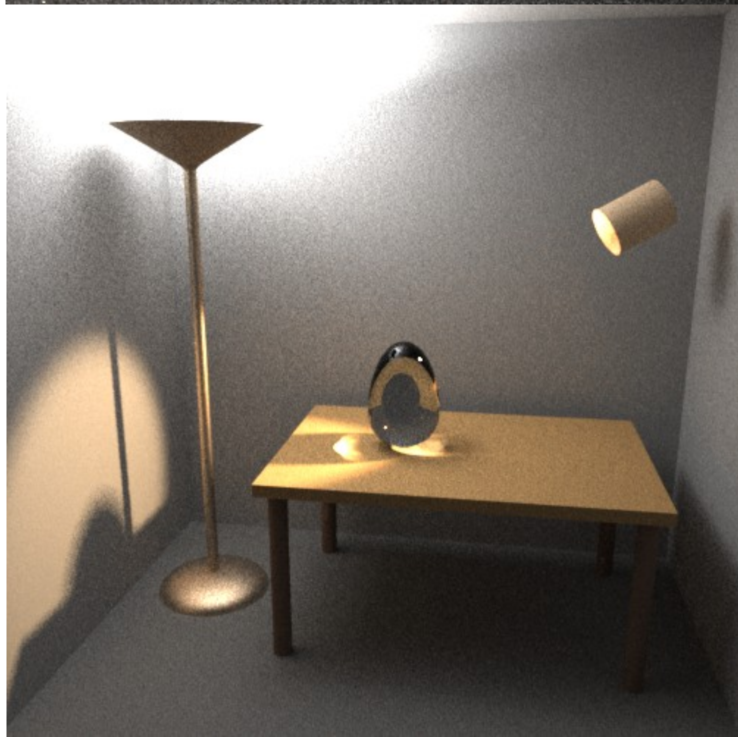
Bridage 2 real-time path tracer

Aalto CS-E5520 Spring 2019
Jaakko Lehtinen

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





What can it do ~today? (pre-RTX, though)

- Path Tracing + Deep Learning for noise removal

Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder

CHAKRAVARTY R. ALLA CHAITANYA, NVIDIA, University of Montreal and McGill University
ANTON S. KAPLANYAN, NVIDIA
CHRISTOPH SCHIED, NVIDIA and Karlsruhe Institute of Technology
MARCO SALVI, NVIDIA
AARON LEFOHN, NVIDIA
DEREK NOWROUZEZAHRAI, McGill University
TIMO AILA, NVIDIA

SIGGRAPH 2017

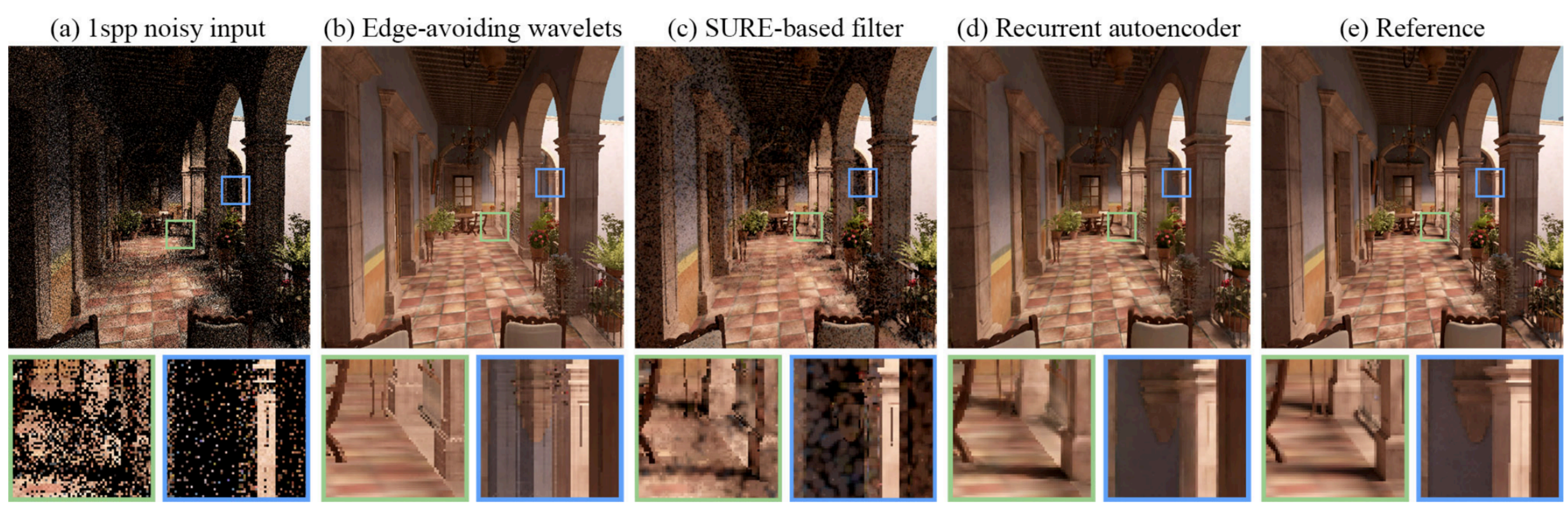


Fig. 1. Left to right: (a) noisy image generated using path-traced global illumination with one indirect inter-reflection and 1 sample/pixel; (b) edge-avoiding wavelet filter [Dammertz et al. 2010] (10.3ms at 720p, SSIM: 0.7737); (c) SURE-based filter [Li et al. 2012] (74.2ms, SSIM: 0.5960); (d) our recurrent denoising autoencoder (54.9ms, SSIM: 0.8438); (e) reference path-traced image with 4096 samples/pixel.

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)

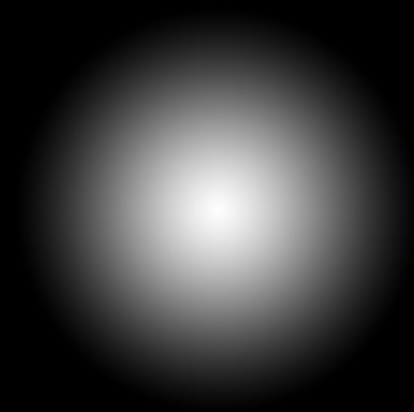
Let’s Go Back to Pixel Filtering

- 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

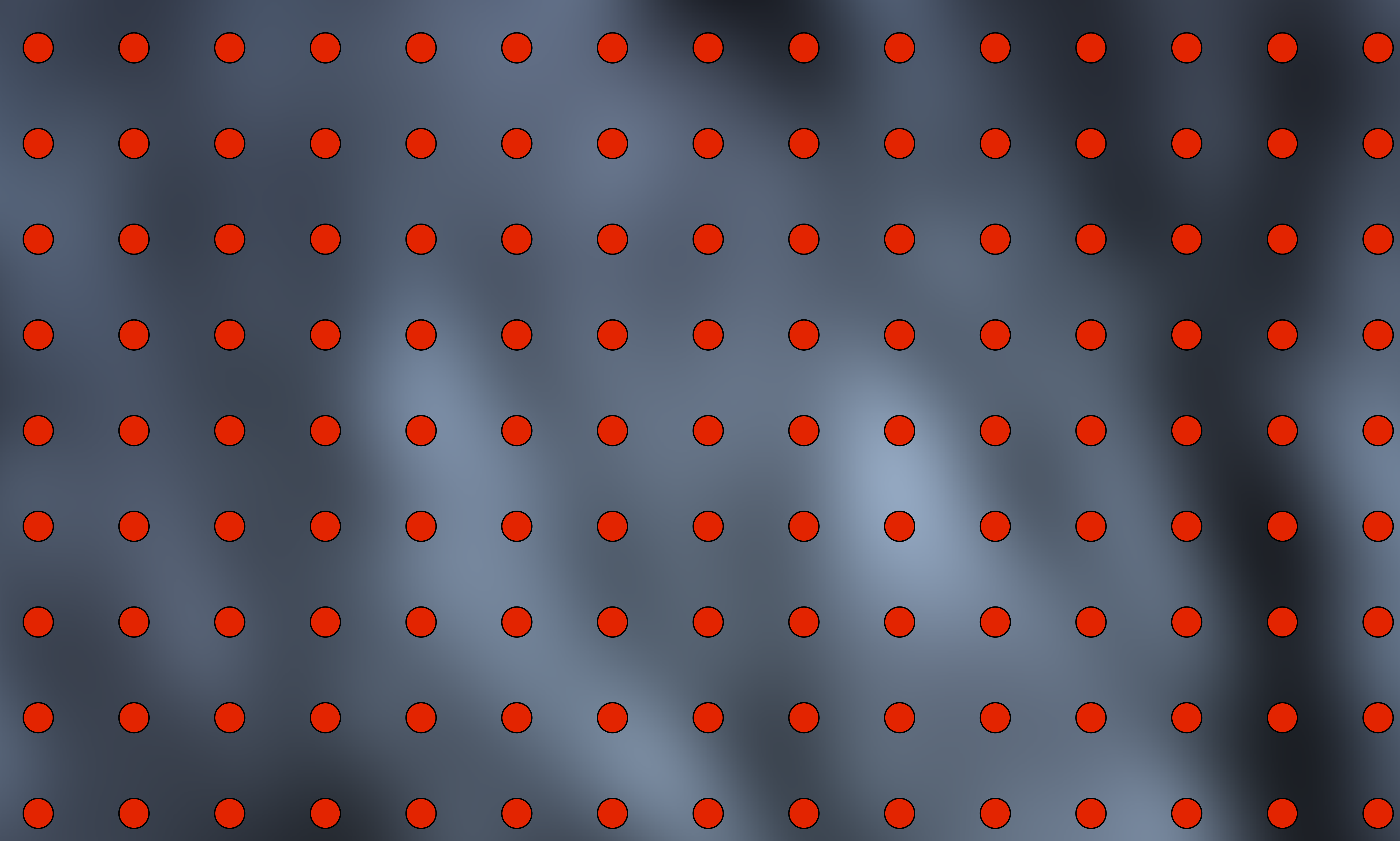
- But we found this was impossible so we turned to supersampling (average many samples in pixel)
  - There is a "proper" way to look at that as well, and here it is..
- (And separate tricks for textures)
  - MIP-maps



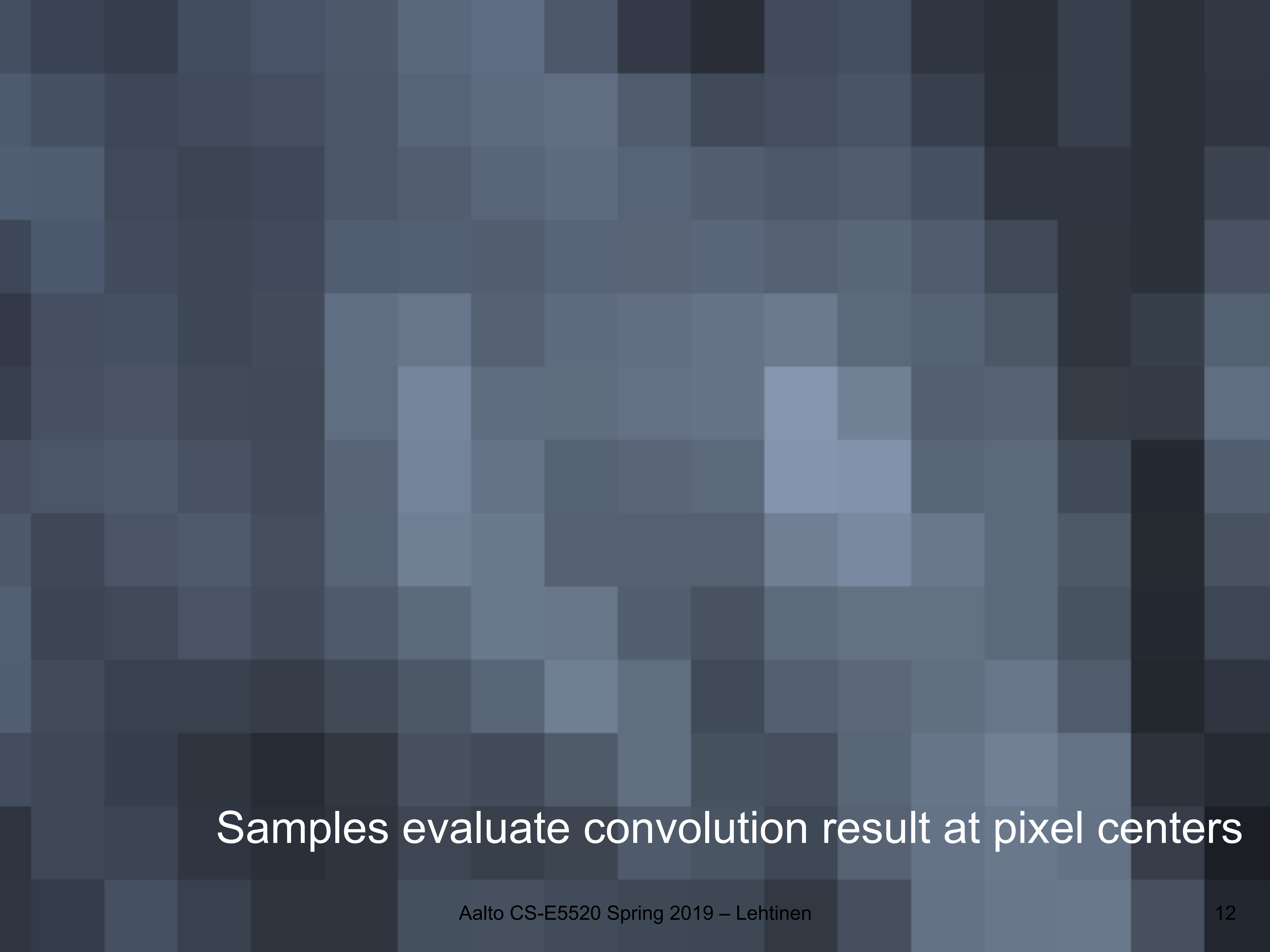
Filter $f(x-x_j, y-y_j)$ centered at pixel (x_j, y_j)

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 *f* and input image; we can actually never compute this exactly)



Samples at pixel centers



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:

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

- I_j is the (discrete) intensity/radiance value of jth 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

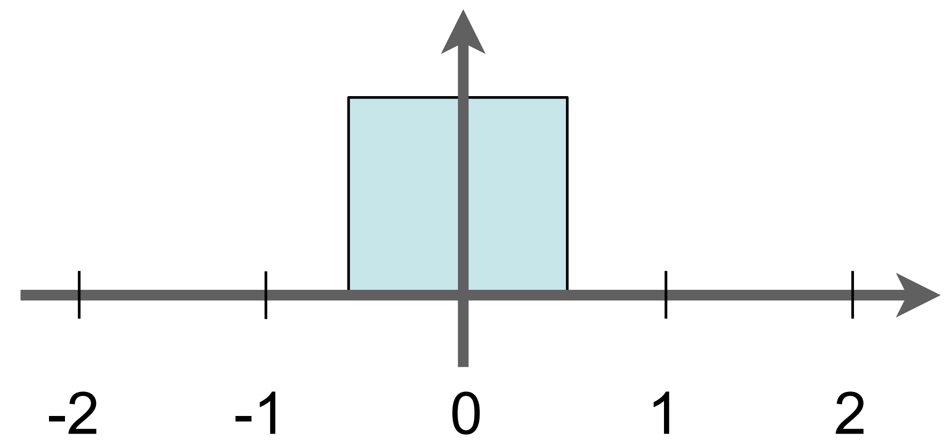
- In practice, we don't care about normalizing the filters analytically, but do it numerically instead

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

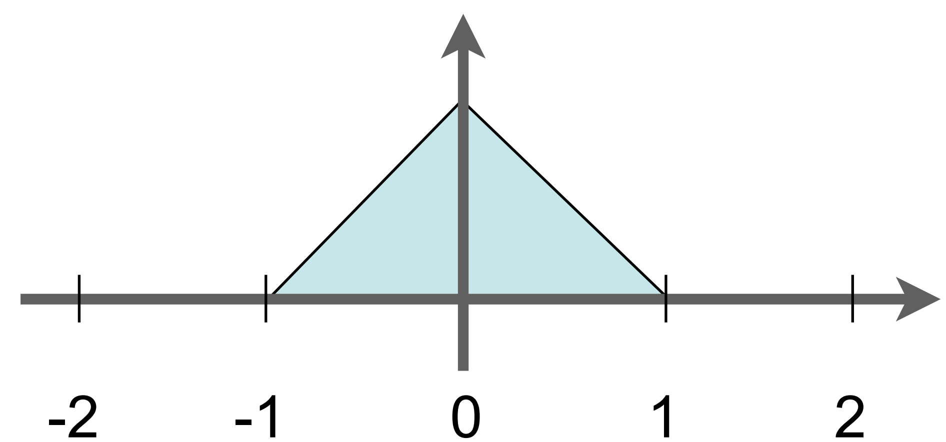
- Intuitive: when we evaluate the above using MC, we sum the "filter weights" from each sample and divide by the sum in the end
  - Note that 1/N cancels out as it's both above and below
  - **IMPORTANT** do it this way; don't rely on $\int f(x, y) = 1$

Common Pixel Filters, 1D profiles

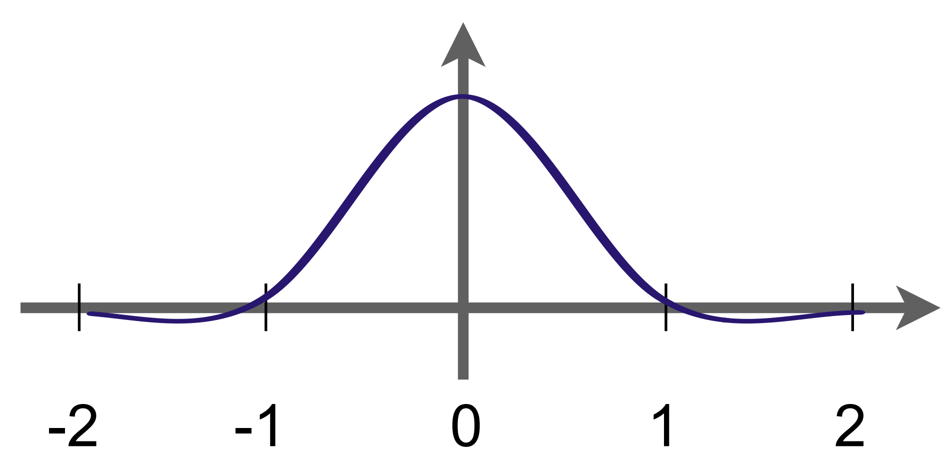
$$f_{\text{box}}(x) = \begin{cases} 1, & -0.5 \le x \le 0.5 \\ 0, & \text{otherwise} \end{cases}$$



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



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



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

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

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

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

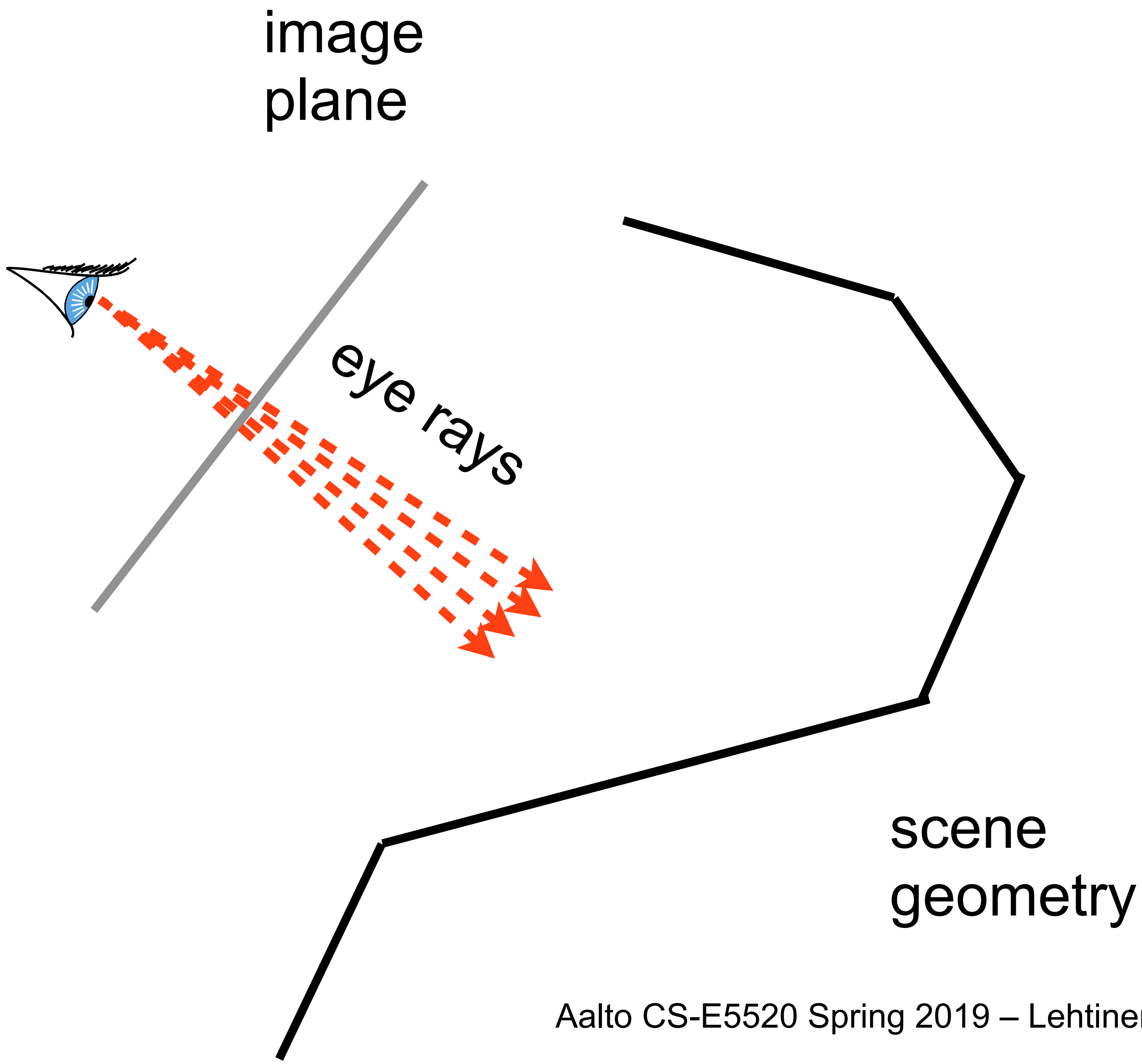
- Each pixel value given by

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

- (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)

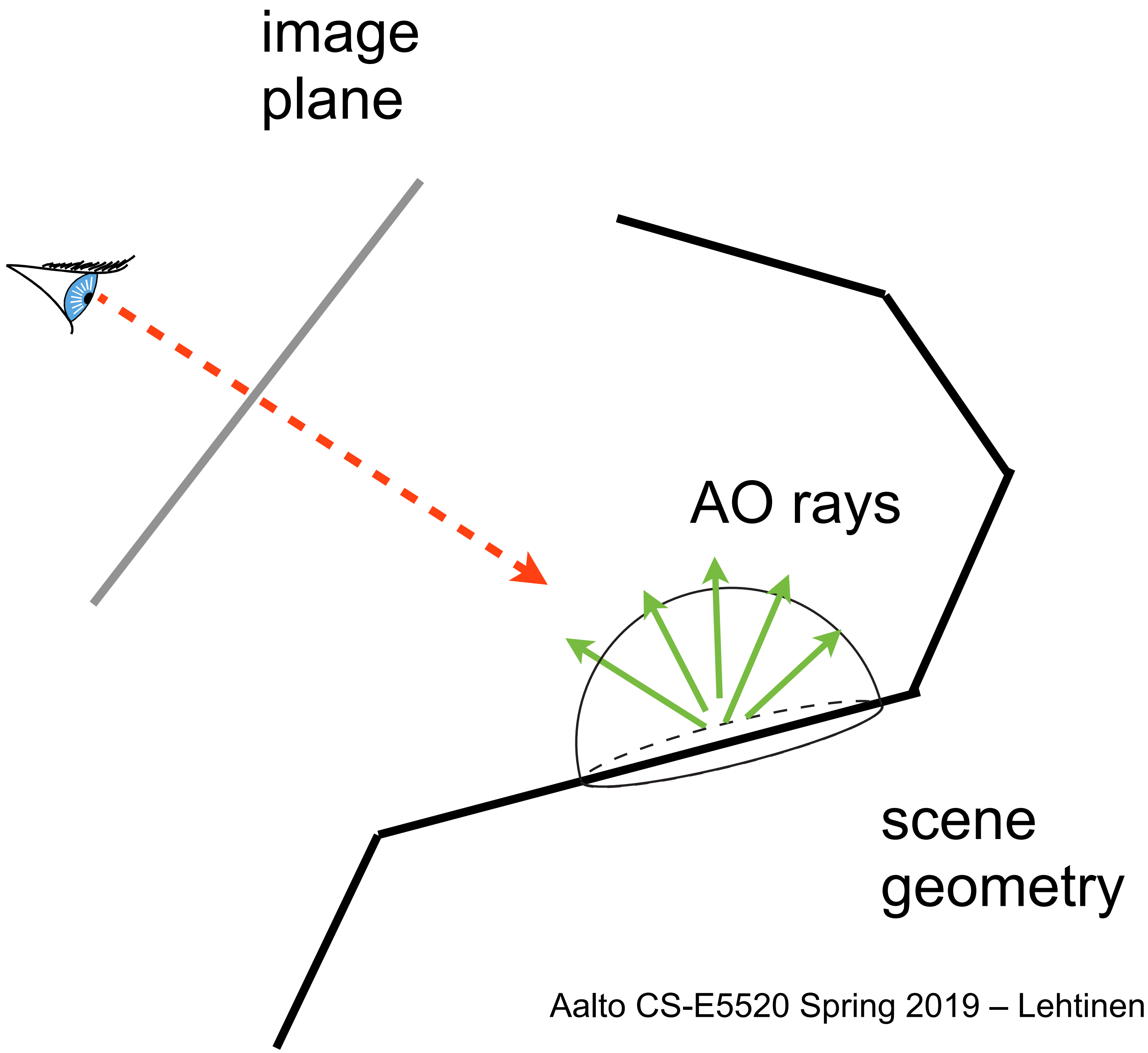
Outer Integral

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



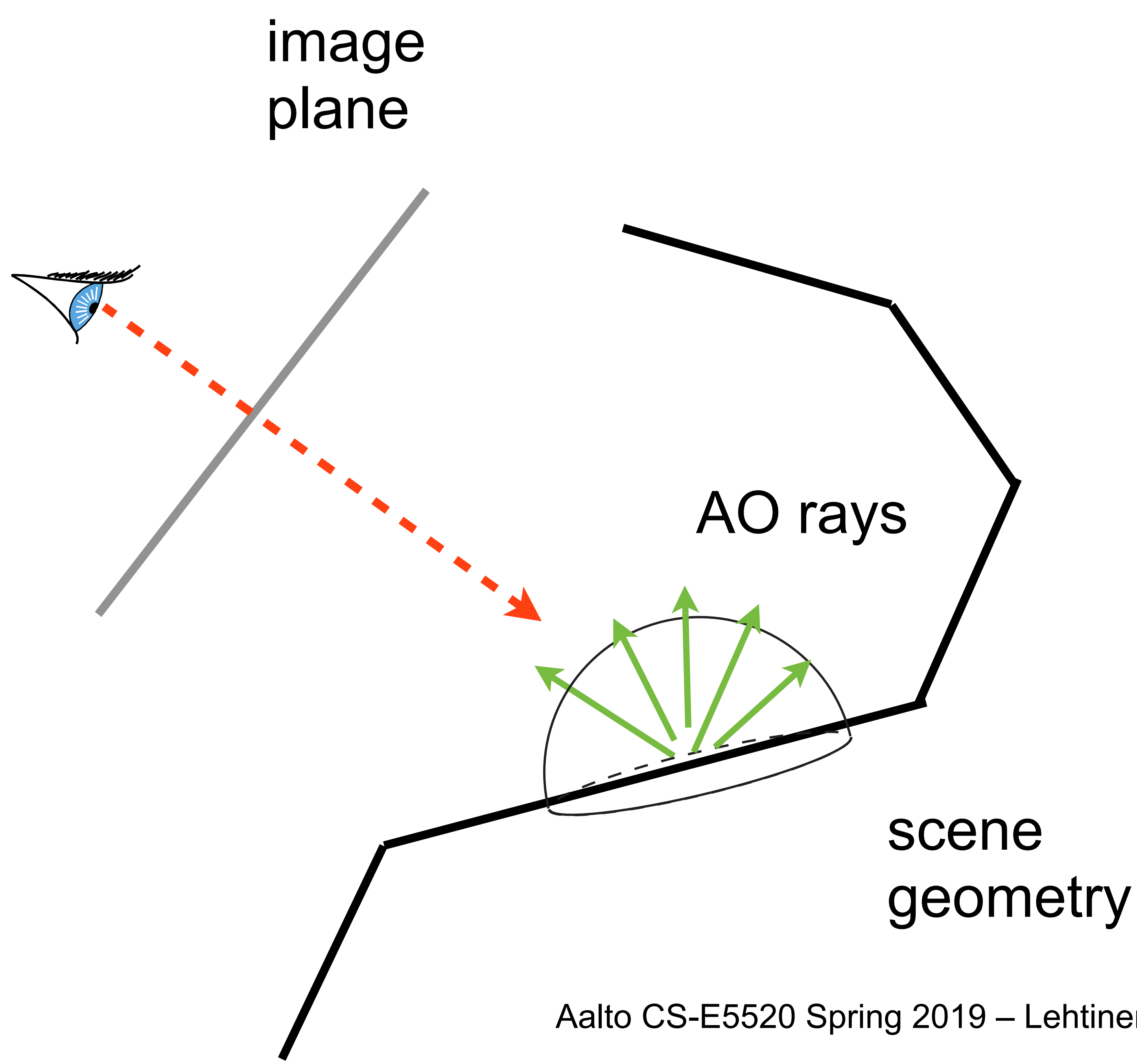
Inner Integral, for each eye ray

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



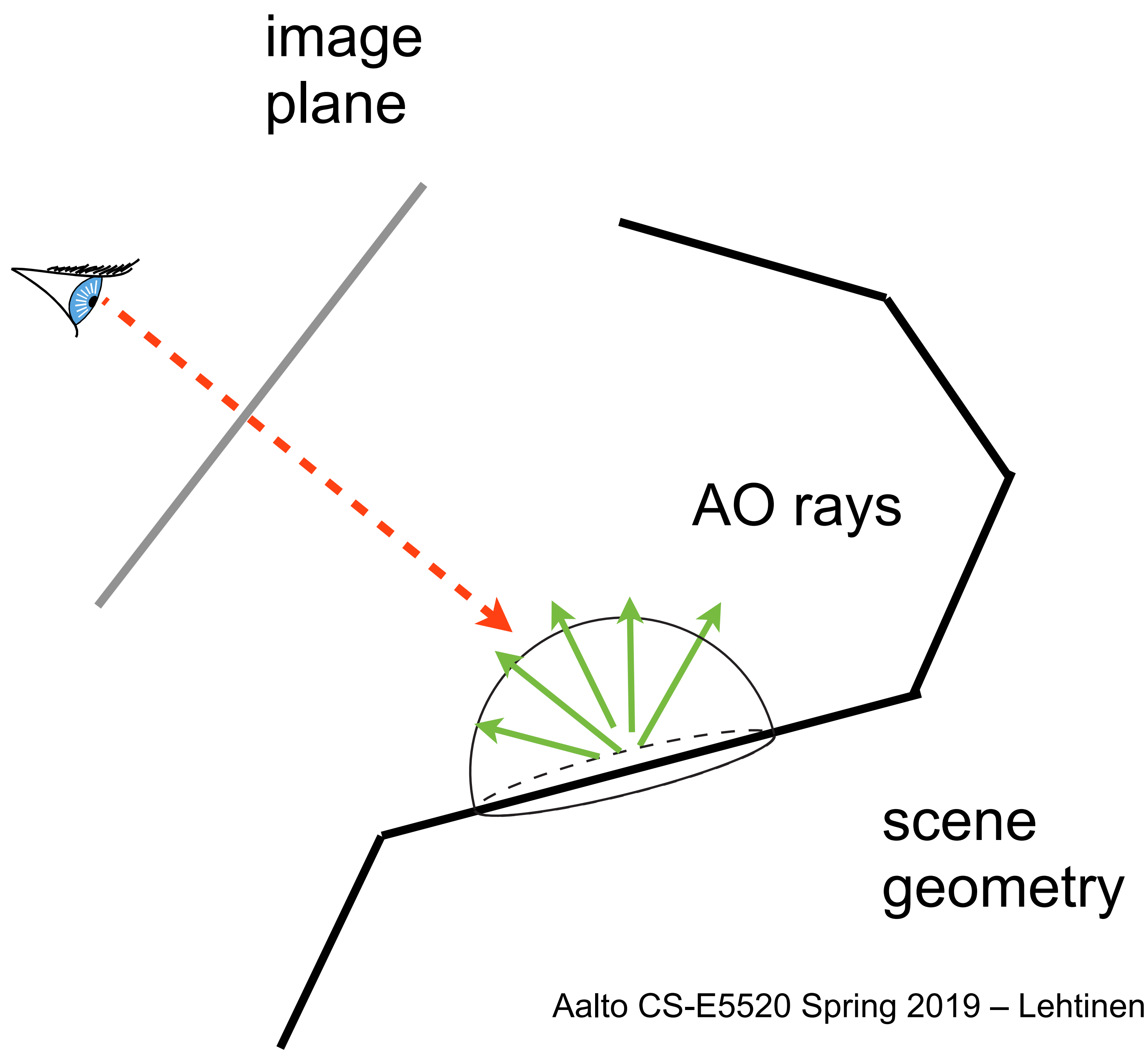
Inner Integral, for each eye ray

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



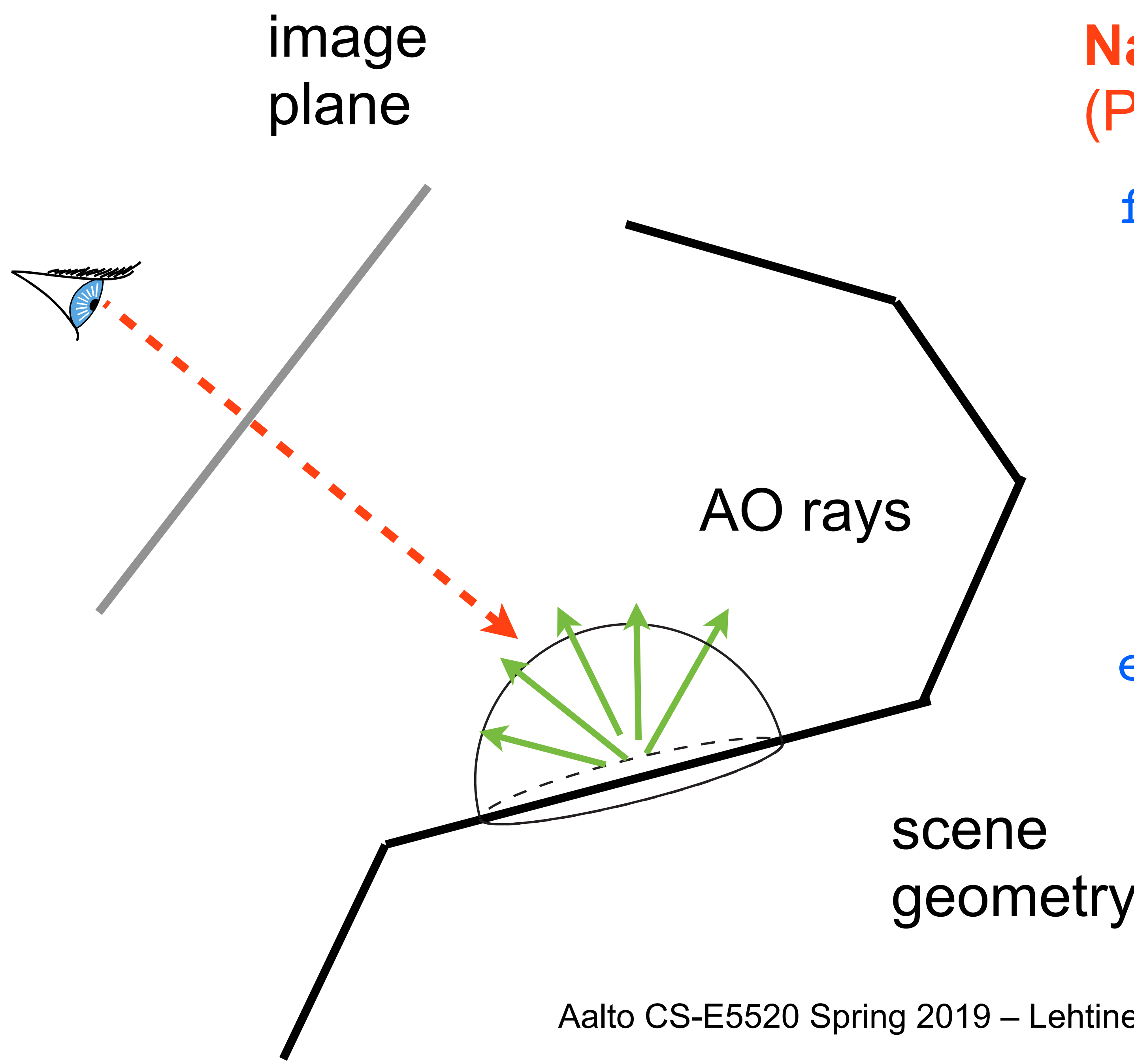
Inner Integral, for each eye ray

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



Inner Integral, for each eye ray

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

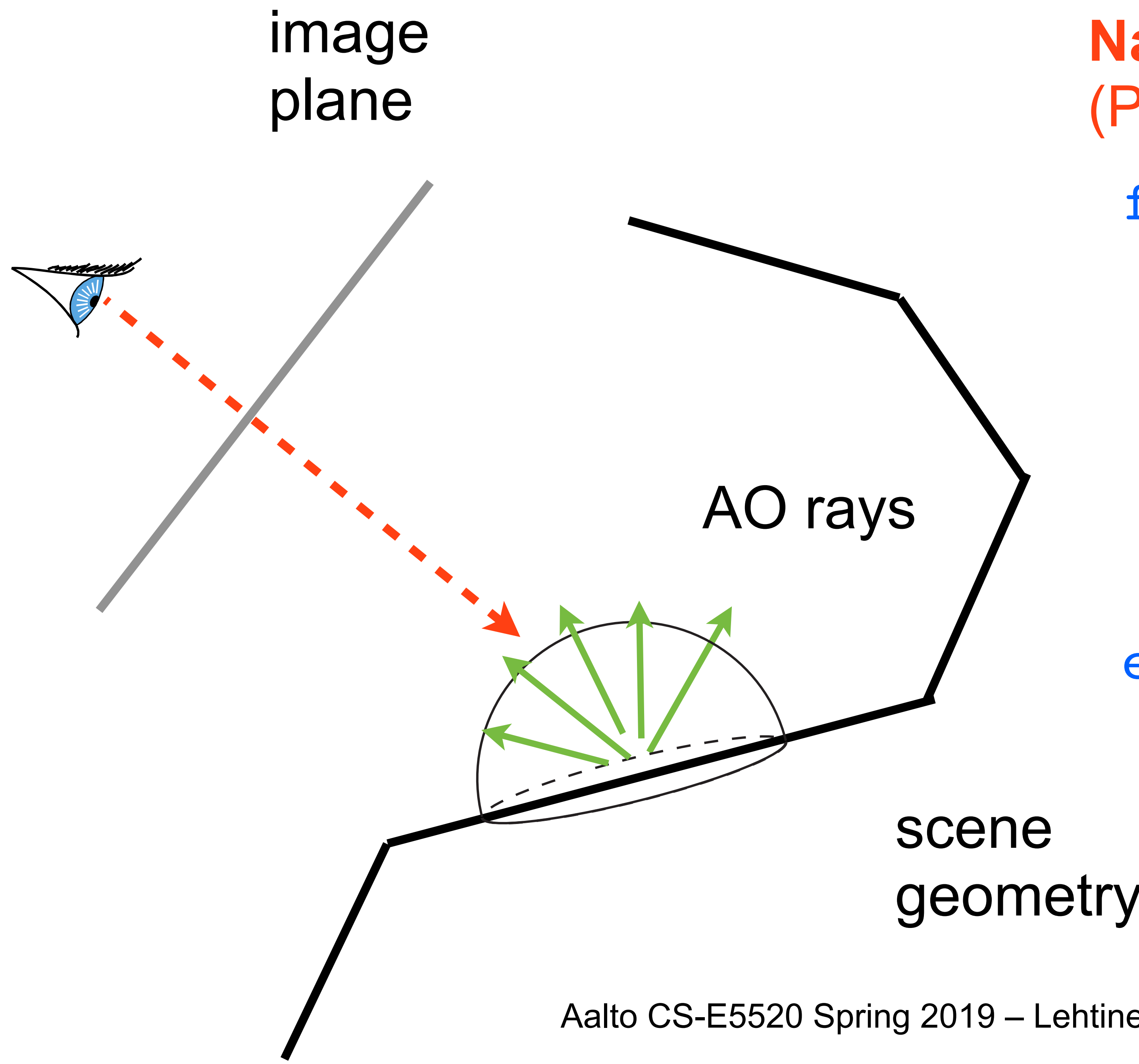


Naive MC implementation:
(PDFs, accumulation not shown)

```
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

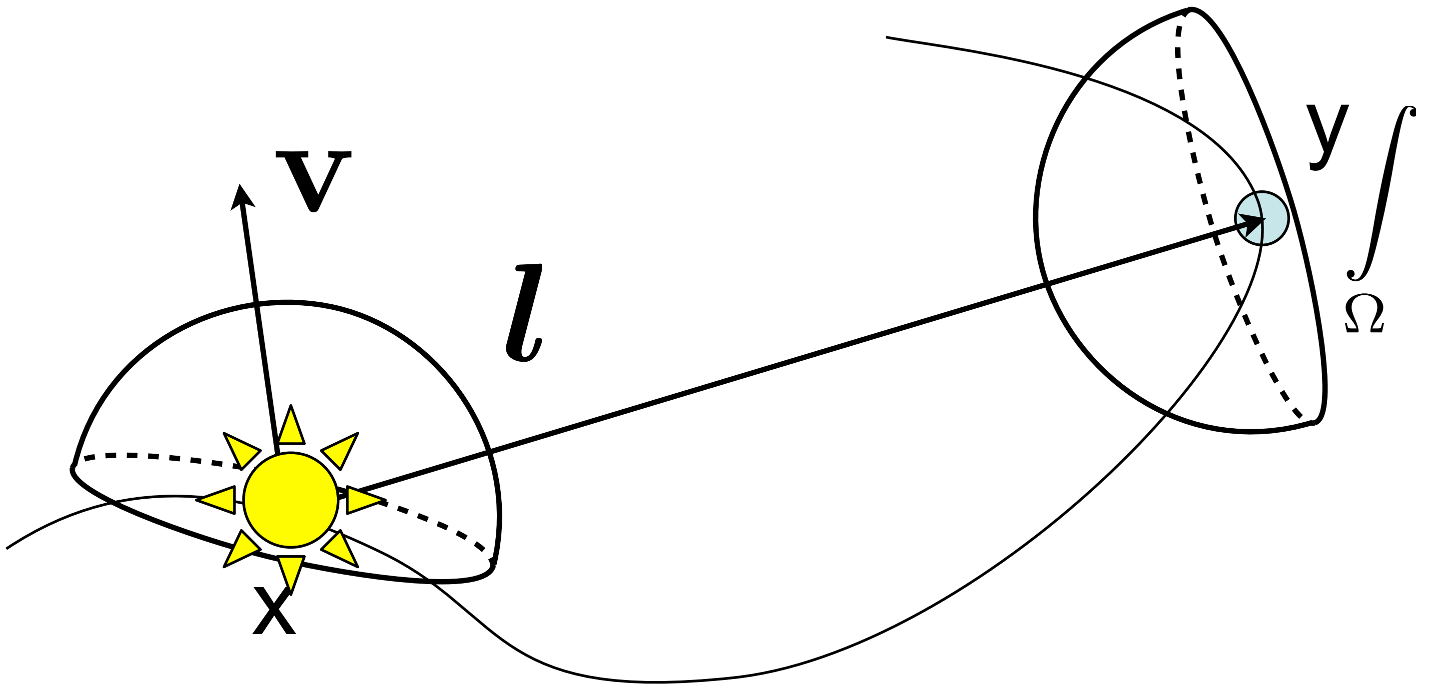
- Difficult to control number of rays cast in the pixel
  - You have two knobs to tweak
- What if we had even further integrals...?

Recap: Rendering Equation

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

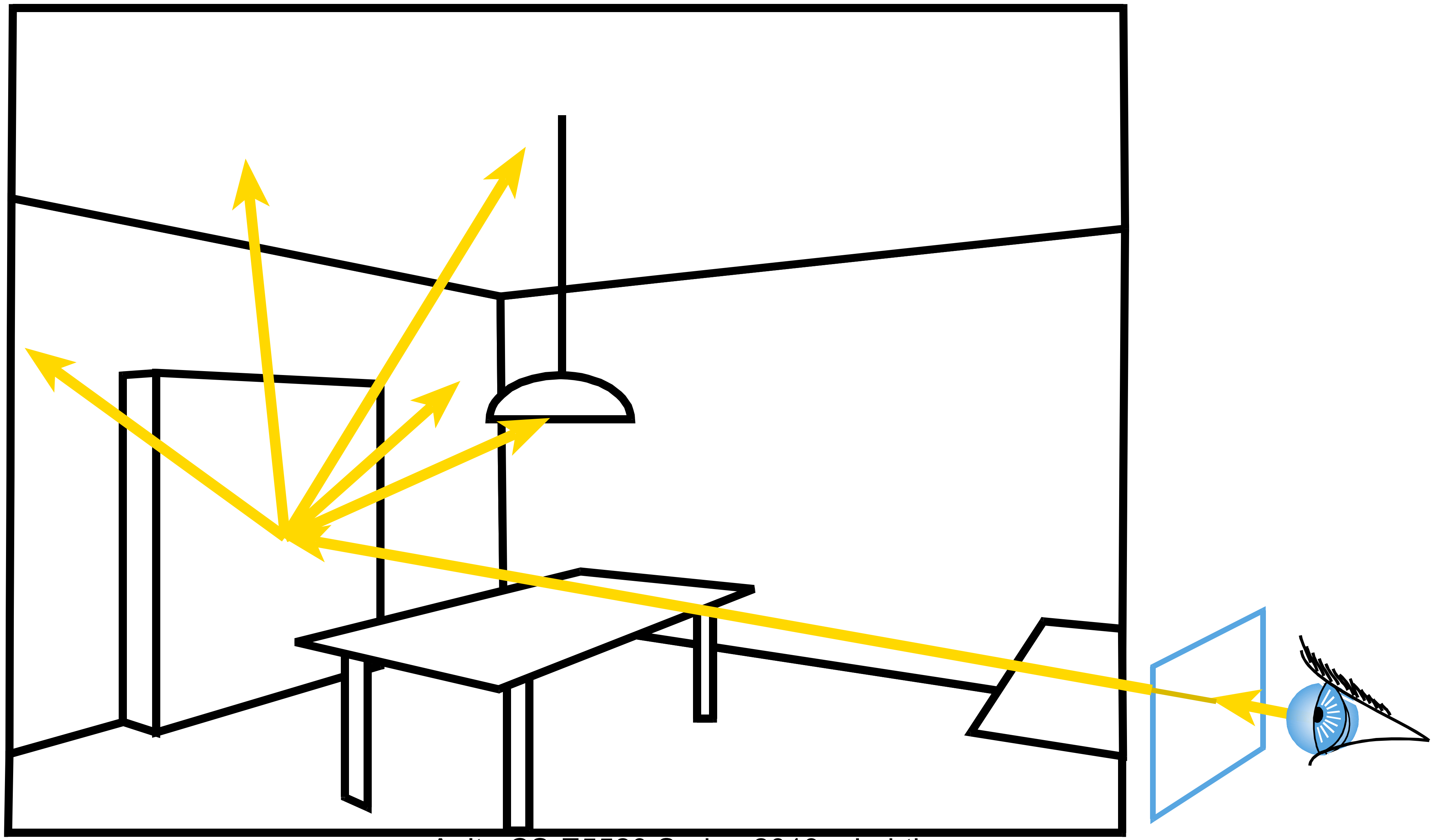
$$+\ E(x \to \mathbf{v})$$

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



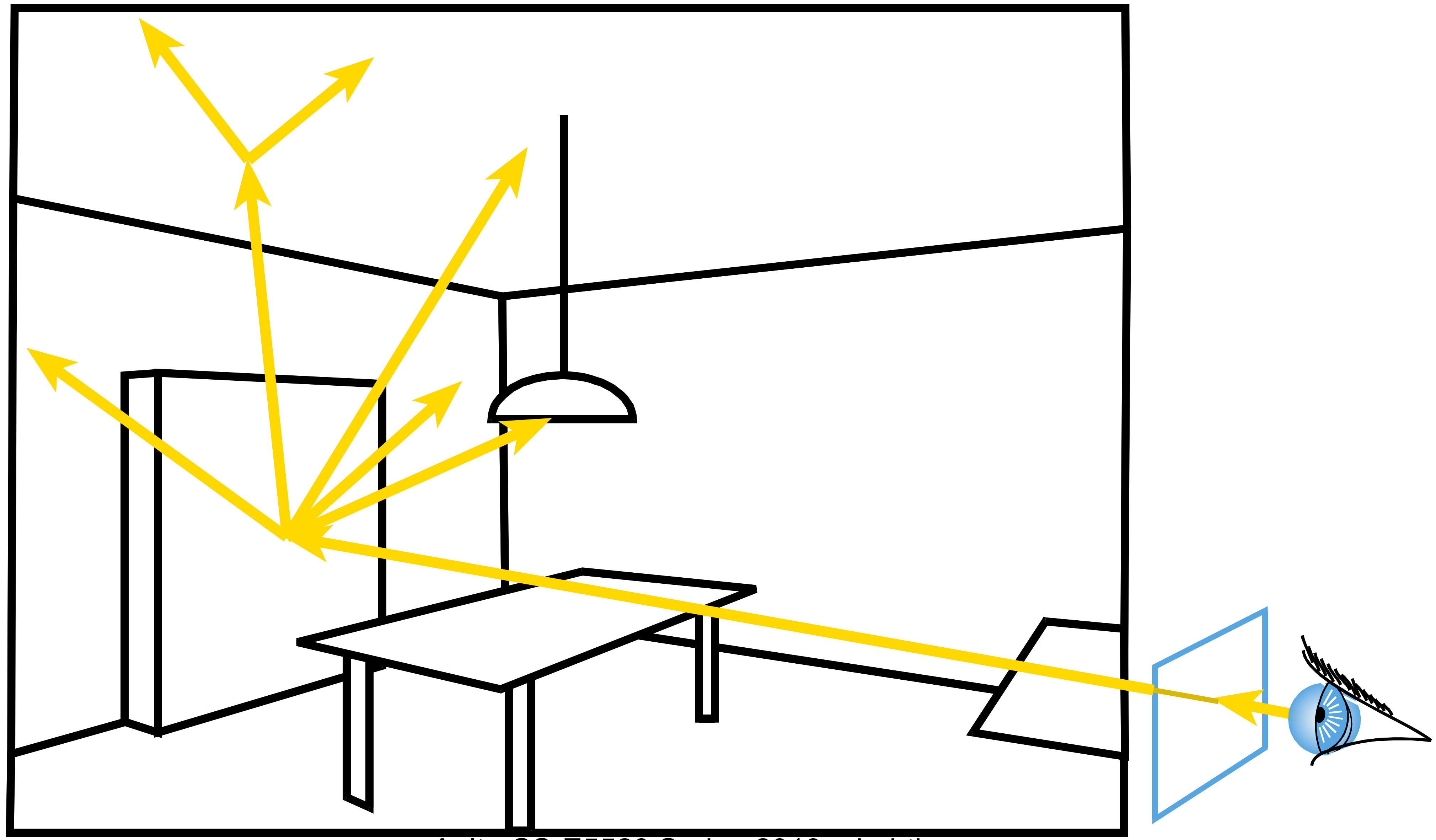
“Monte-Carlo Ray Tracing”

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



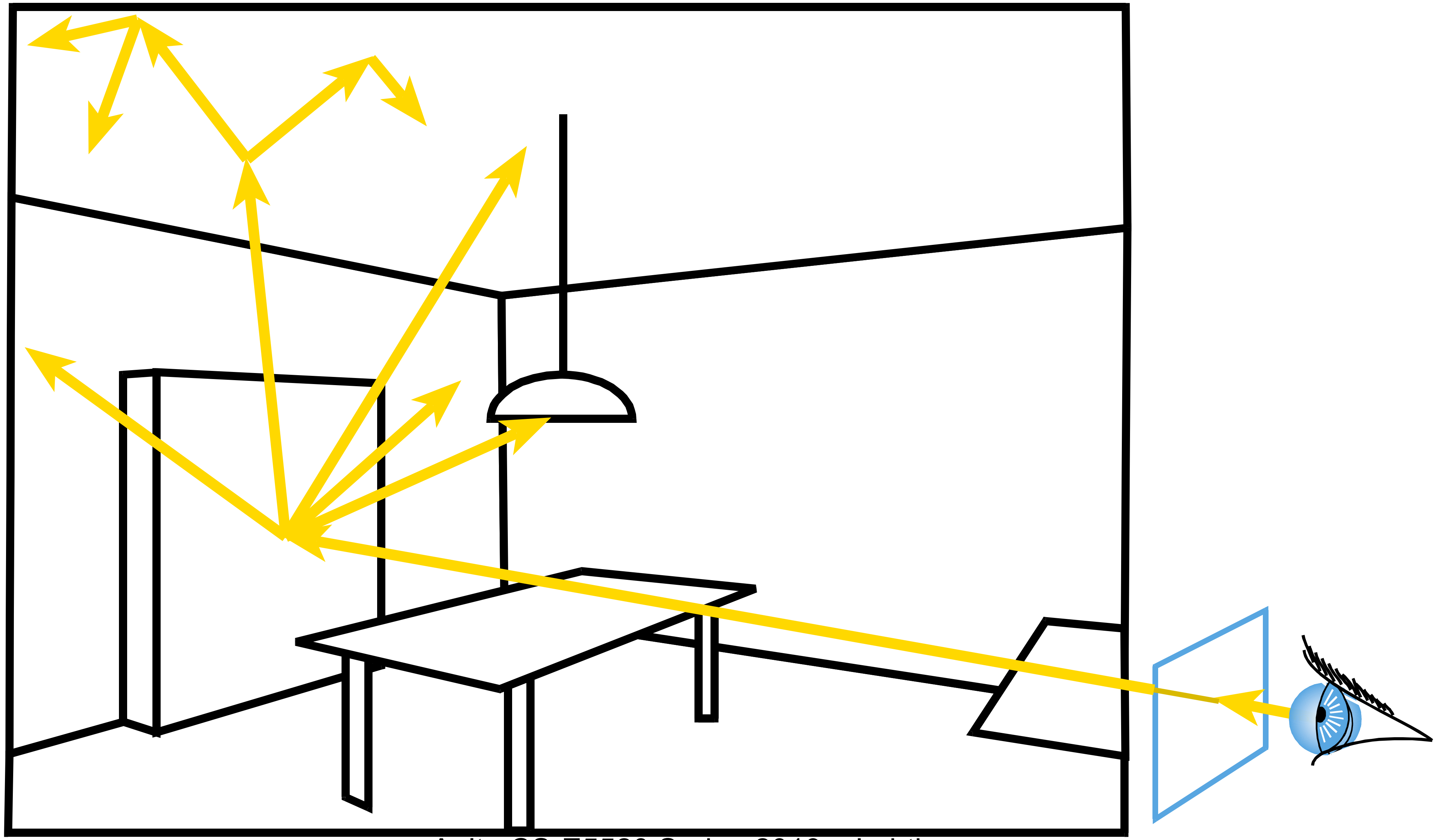
“Monte-Carlo Ray Tracing”

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



“Monte-Carlo Ray Tracing”

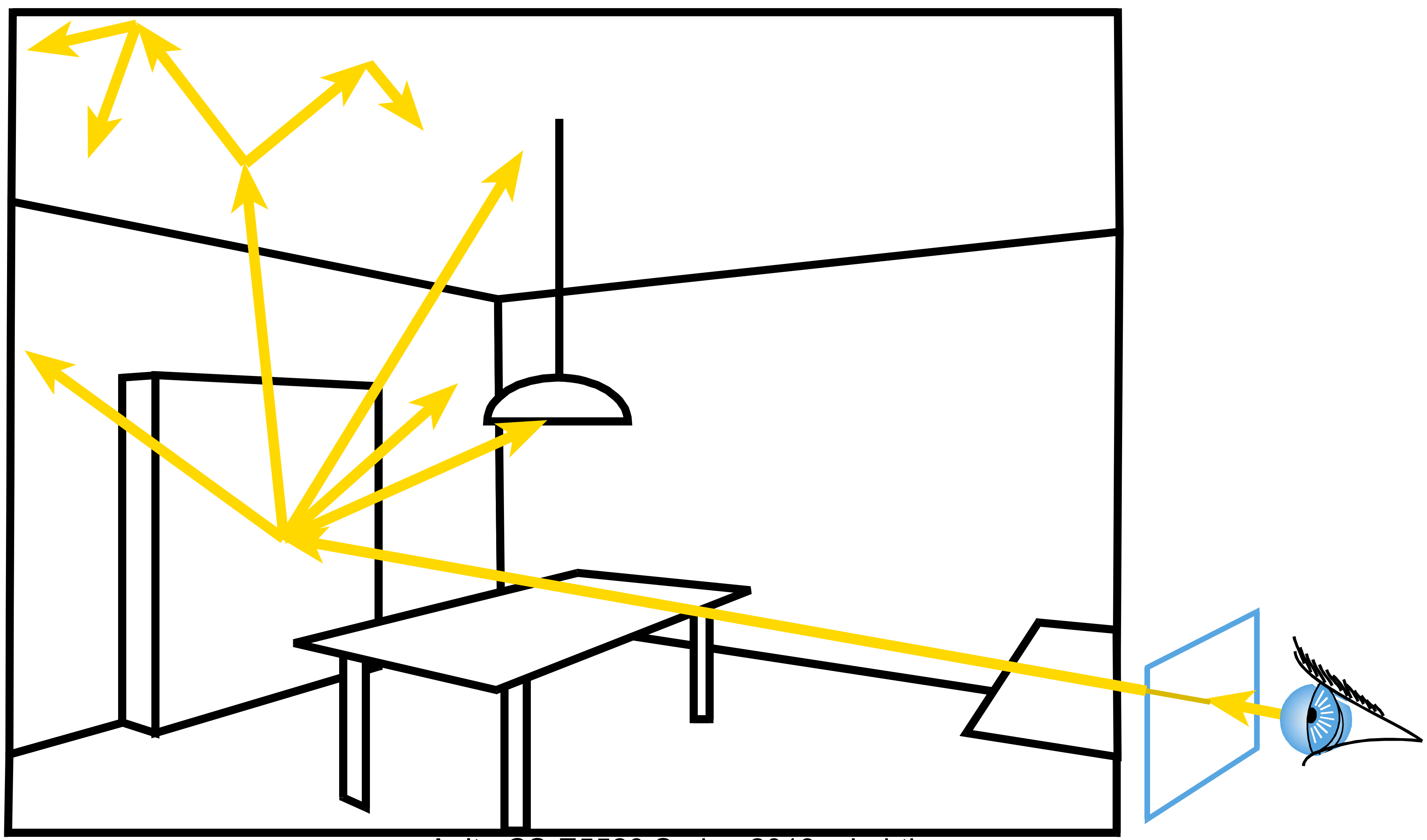
- Cast a ray from the eye through each pixel
- Cast N random rays from the visible point
- Recurse



"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:

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

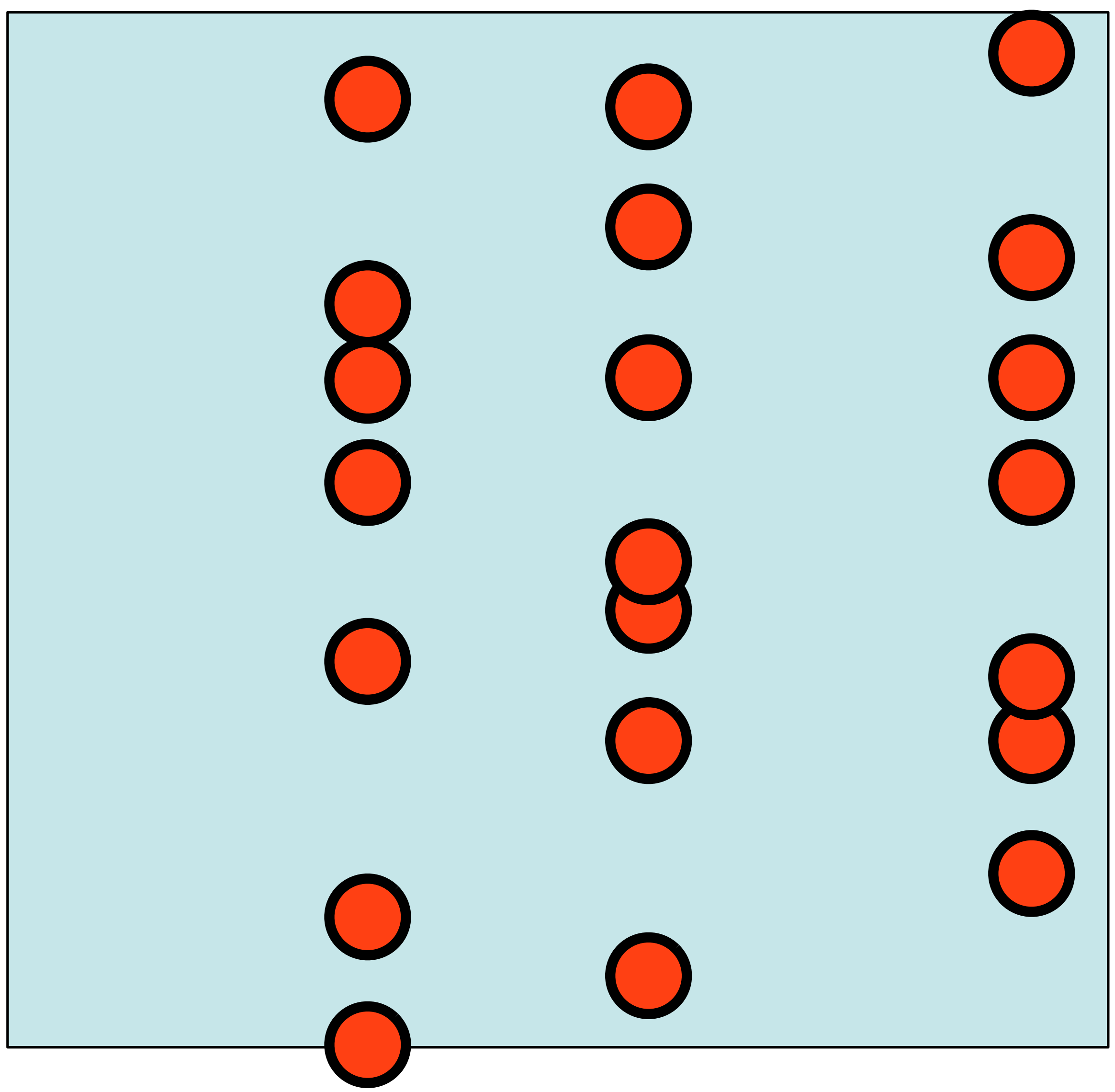
with integrand

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

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

Nested 1D + 1D, naive

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

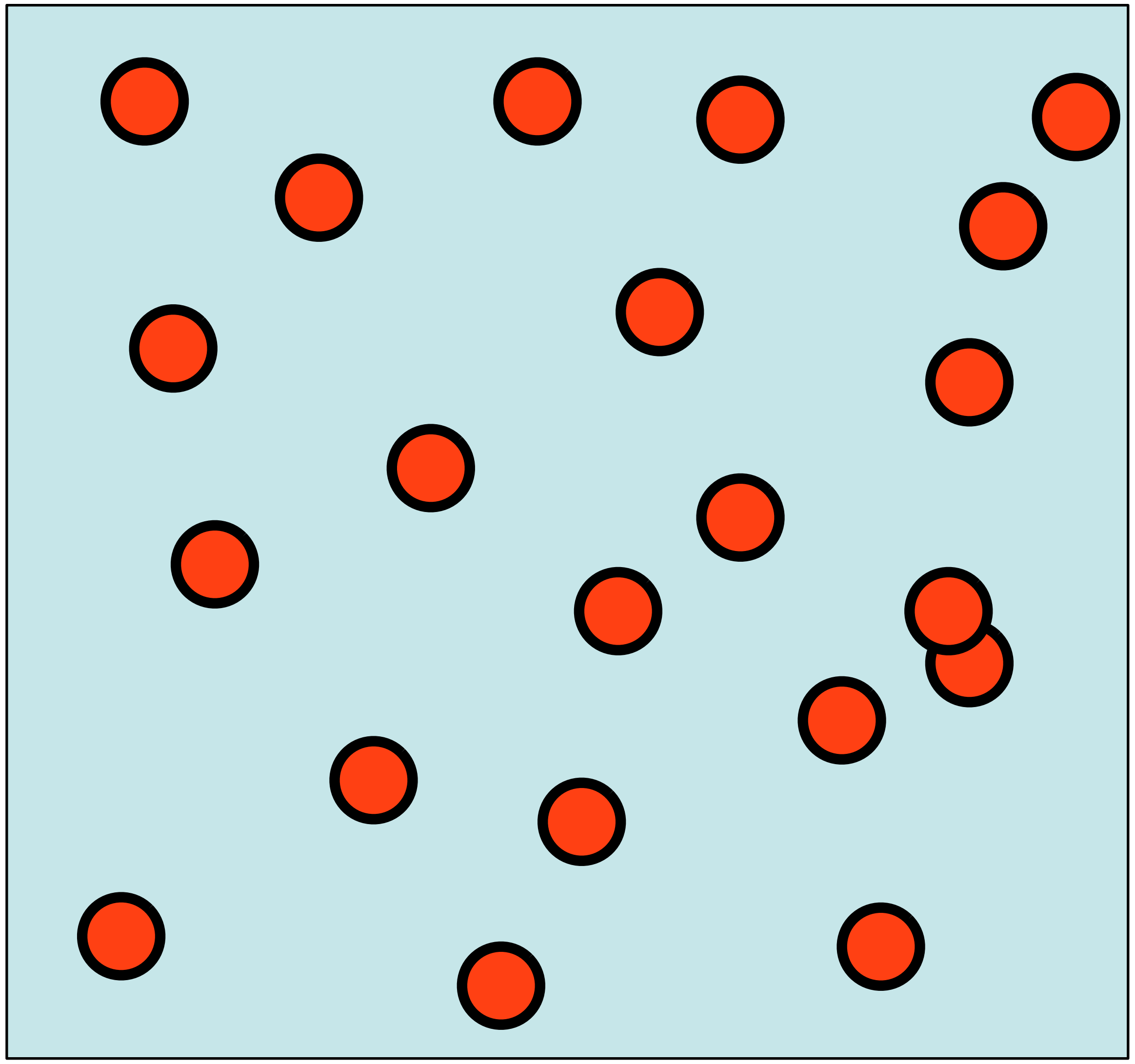


First pick x,
then pick a
bunch of ys

Repeat

Nested 1D + 1D, treat as 2D

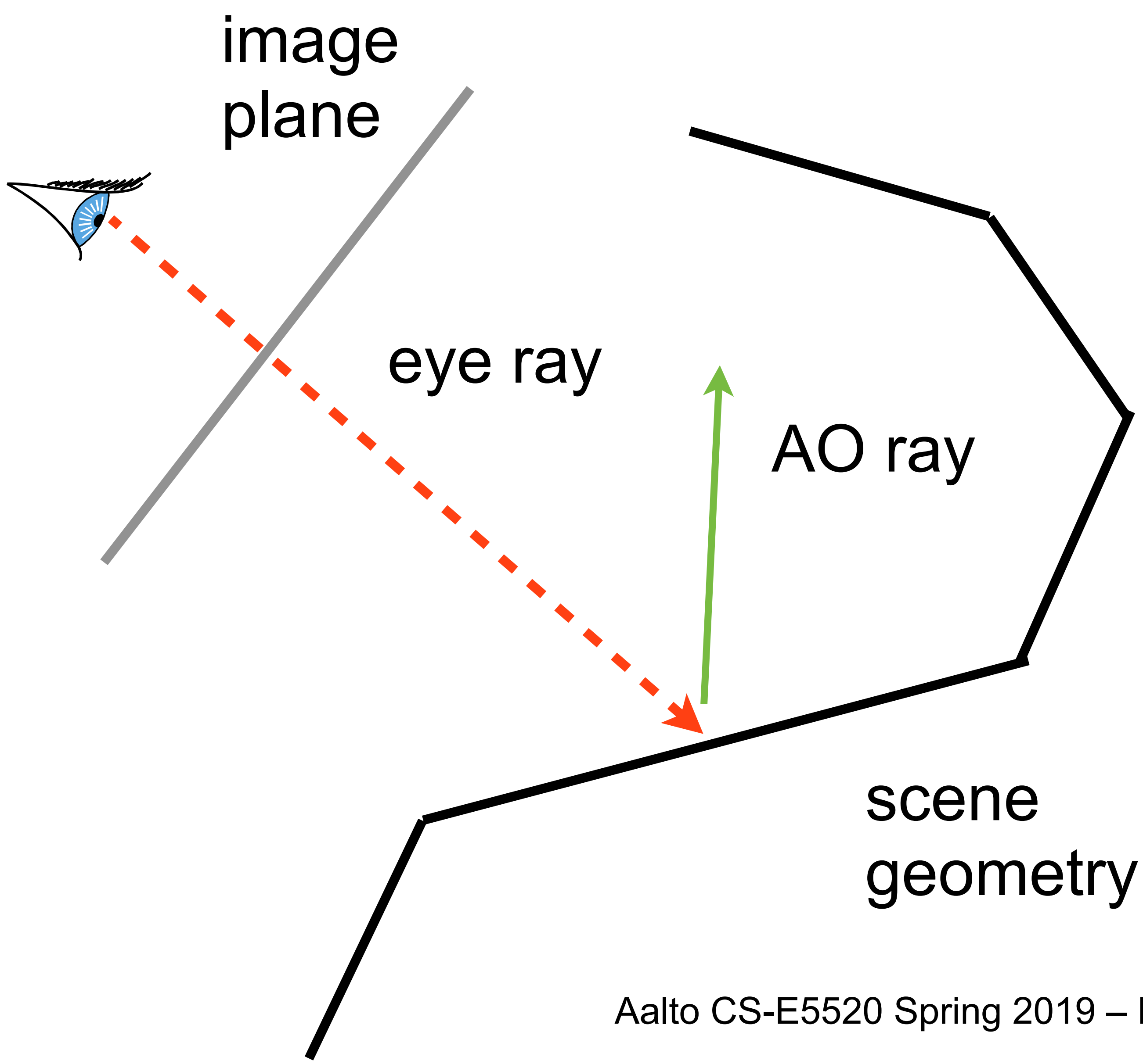
$$\int \left(\int f(x,y) \mathrm{d}y \right) \mathrm{d}x = \int \int f(x,y)\, \mathrm{d}x \mathrm{d}y$$



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

Visually: One sample is Two Rays

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

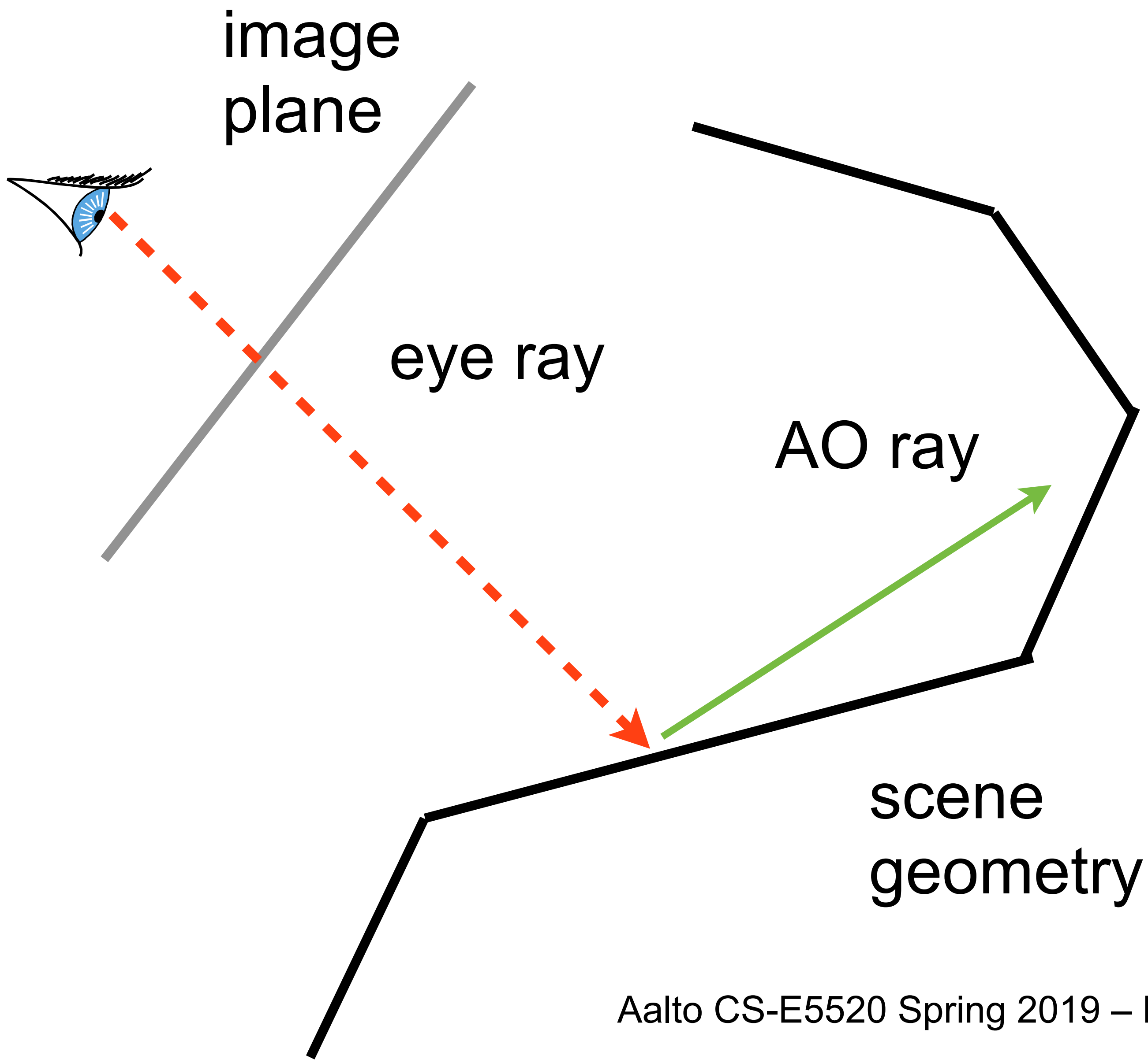


Better MC implementation:

```
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,V)/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,V)/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

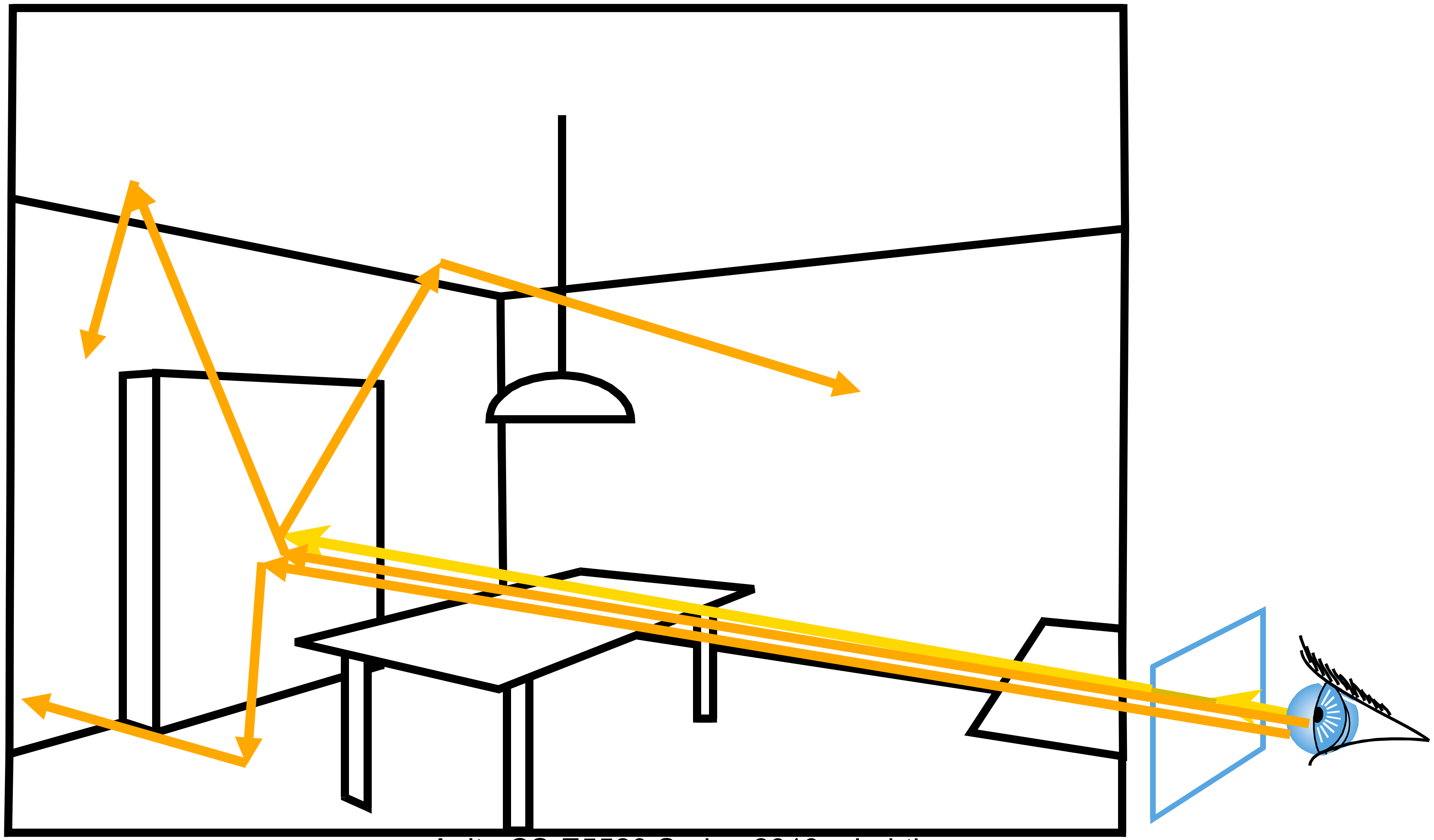
```
res[k] = weight[k] = 0 for all pixels k
for each pixel k
  for i=1 to #samplesperpixel
    pick sample (x,y,omega)          // e.g. 4D Sobol'
    pdf=p(x,y)*p(omega)              // usually p(x,y) == 1
    P=castray(x,y)                   // find primary hit
    V=castray(P,omega).length()>D    // 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

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

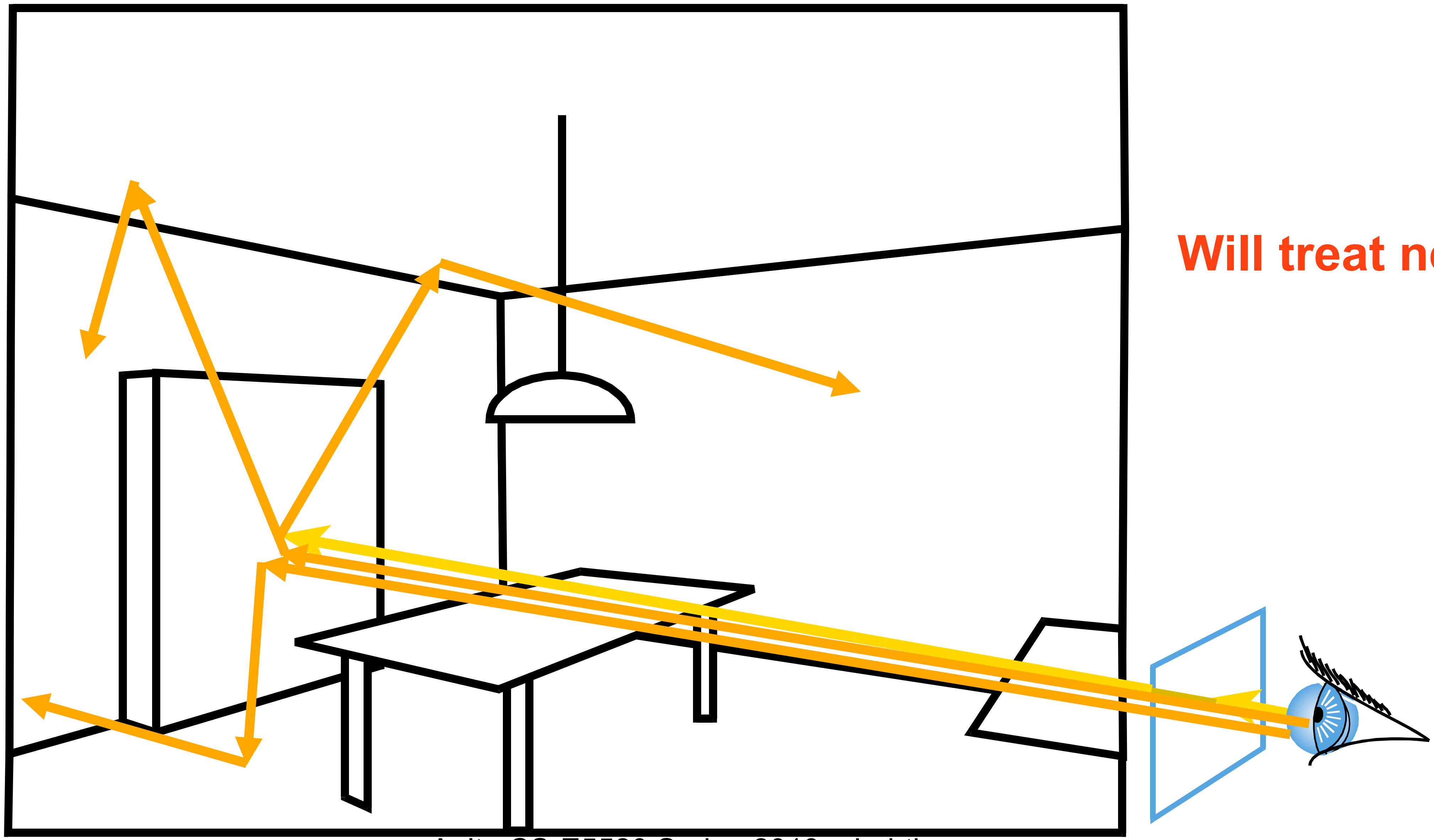
Monte Carlo Path Tracing

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



Monte Carlo Path Tracing

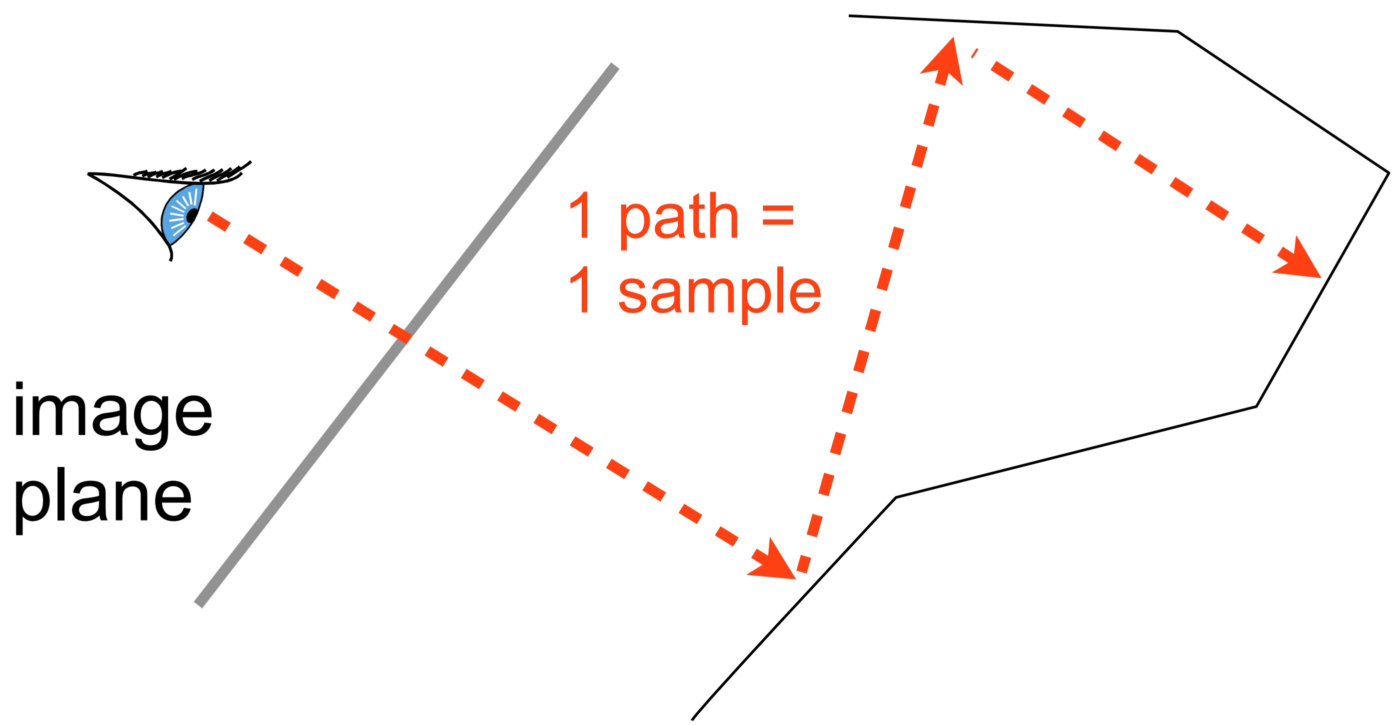
- Trace only one secondary ray per recursion
  - Otherwise number of rays explodes!
- But send many primary rays per pixel (antialiasing)



Will treat next time!

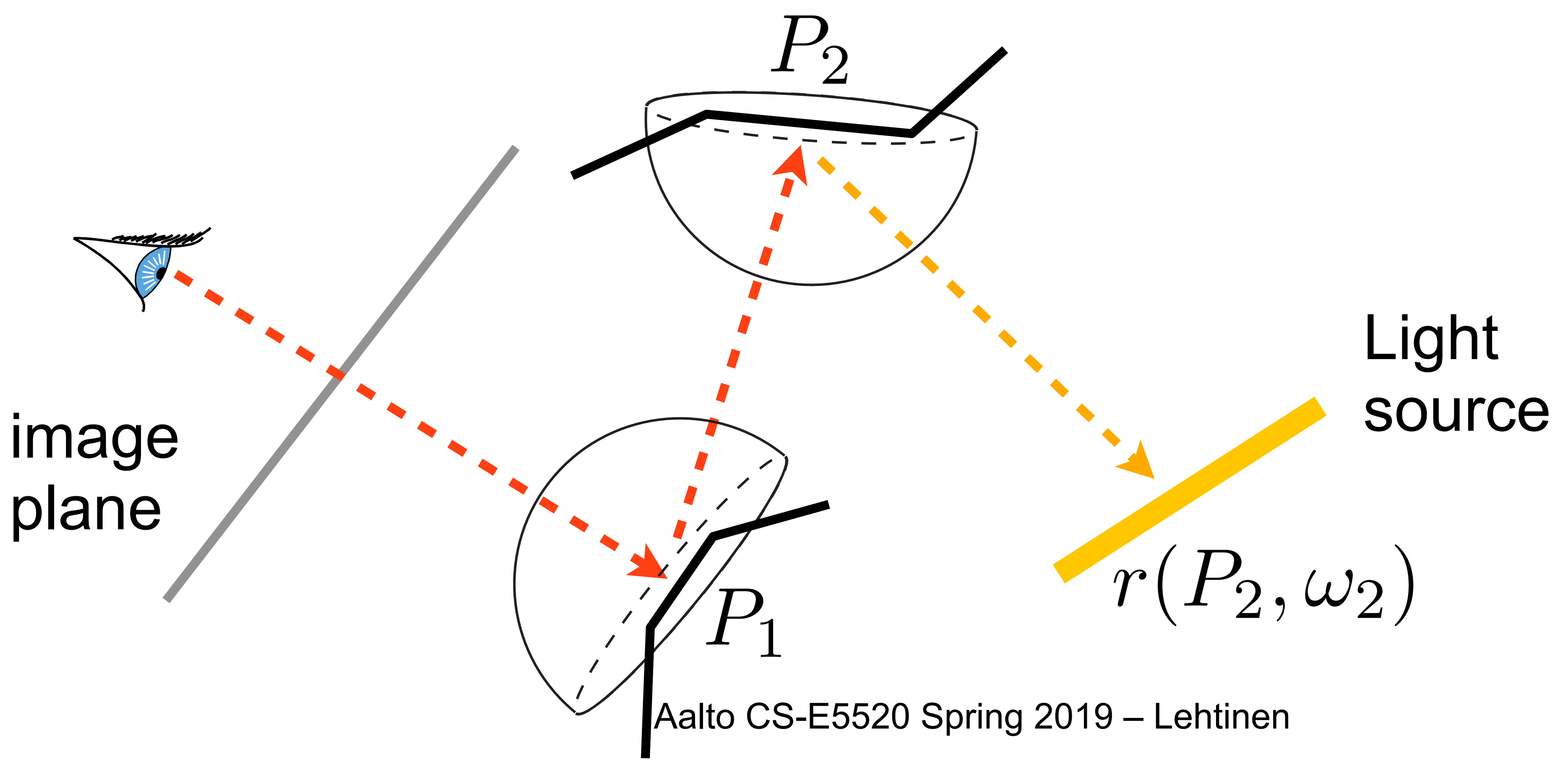
Monte Carlo Path Tracing

- The idea is just the same as before with AO+filter
  - Instead of thinking about nested integrals over hemispheres at each bounce, let's think of one integral over the Cartesian product of all the hemispheres
  - For n bounces, the domain is $\text{screen} \times \underbrace{\Omega \times \ldots \times \Omega}_{n \text{times}}$
  - Each sample is a *path* = *sequence of rays*



Example: 1 Indirect Bounce

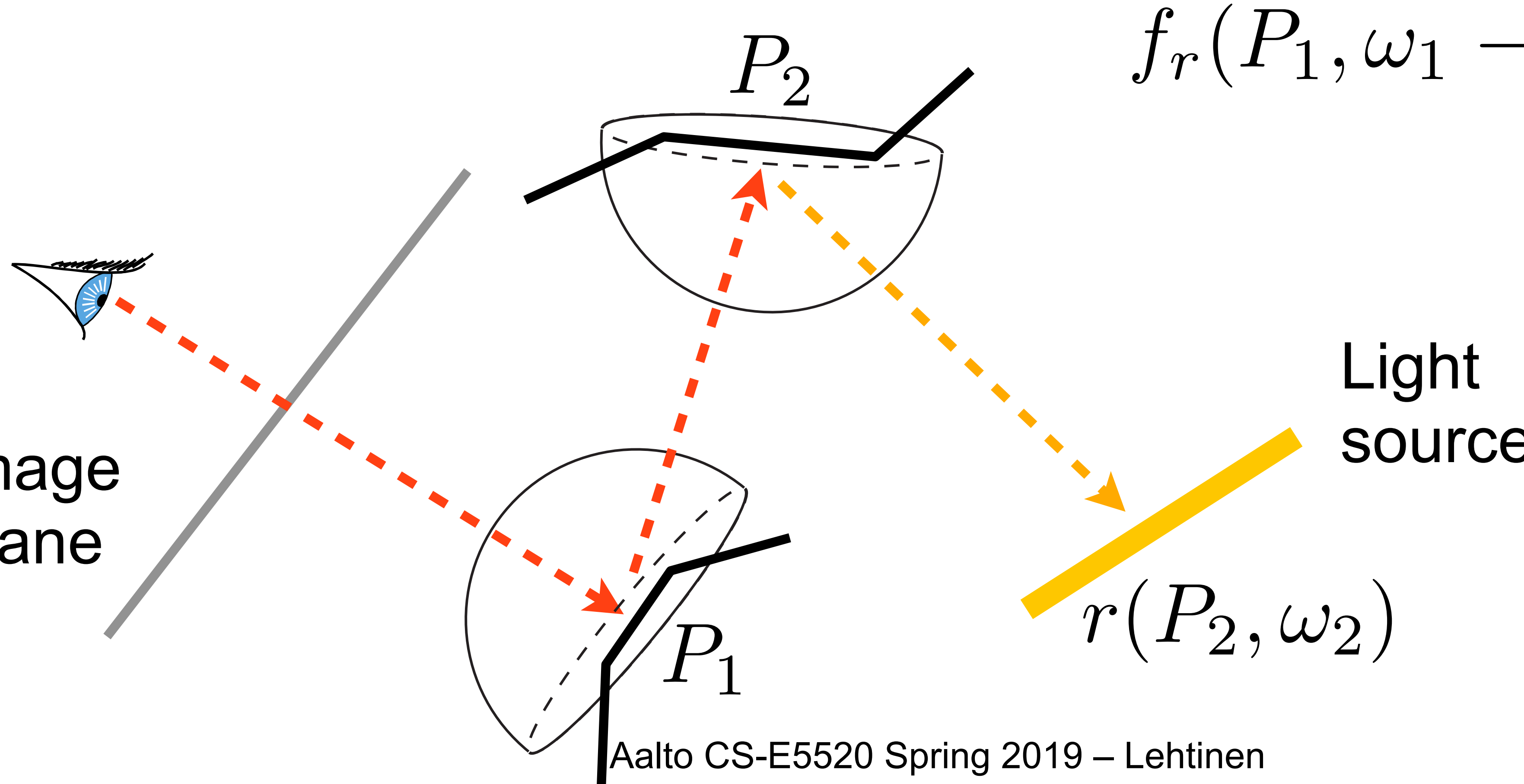
- Nested version (P_1, P_2 are ray hit points)



Example: 1 Indirect Bounce

- Nested version (P_1, P_2 are ray hit points)

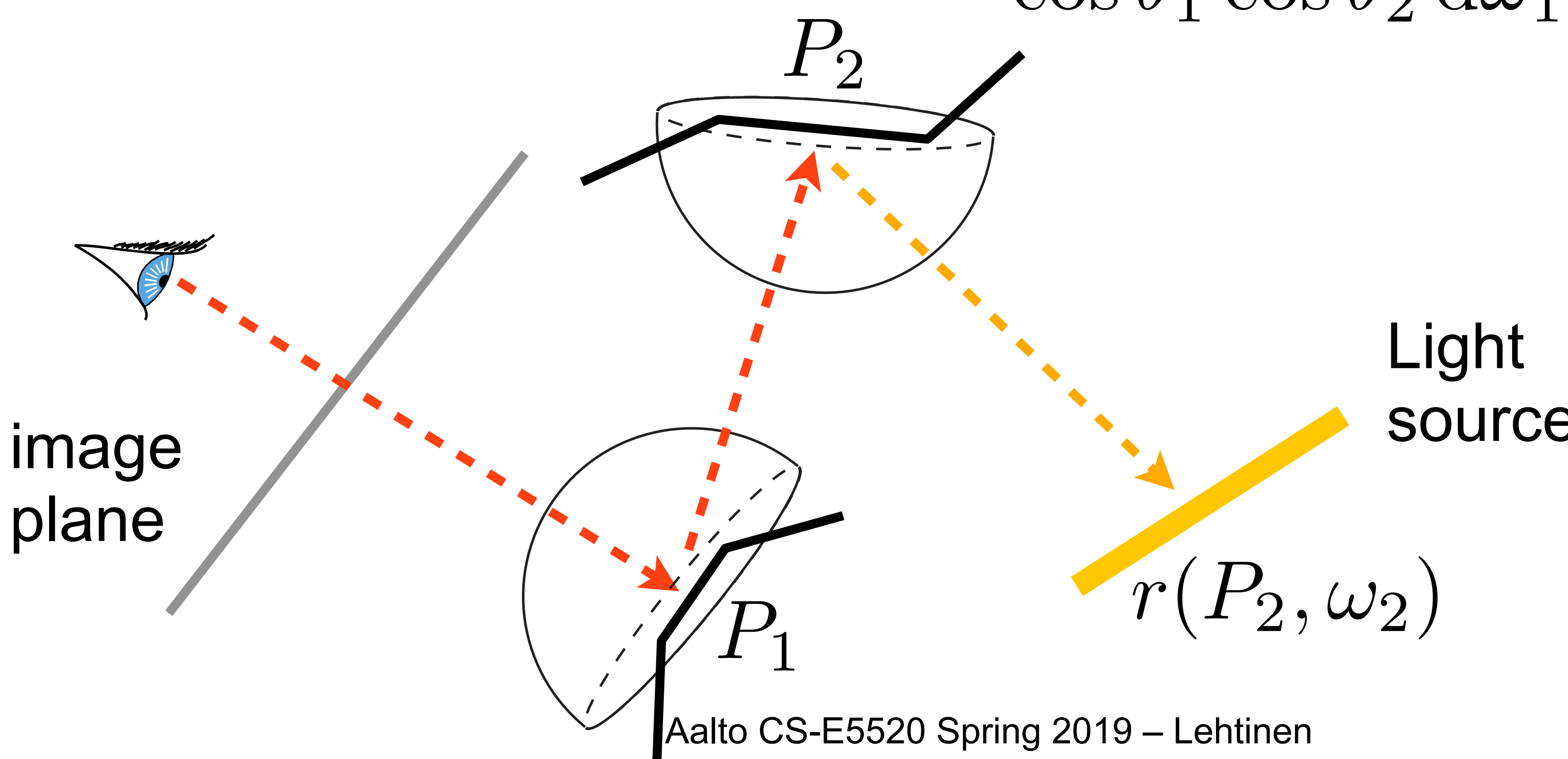
$$L_2(x,y) = \int_{\Omega(P_1)} \overbrace{\left[\int_{\Omega(P_2)} E(r(P_2,\omega_2) \to P_2) f_r(P_2, \omega_2 \to -\omega_1) \cos\theta_2 \mathrm{d}\omega_2\right]}^{L(P_1 \leftarrow \omega_1)} f_r(P_1, \omega_1 \to \mathrm{eye}) \cos\theta_1 \mathrm{d}\omega_1$$



Example: 1 Indirect Bounce

- Flat version, 4D integral

$$L_2(x,y) = \int_{\Omega(P_1)\times\Omega(P_2)} E(r(P_2,\omega_2) \to P_2) \times$$
$$f_r(P_2,\omega_2 \to -\omega_1) f_r(P_1,\omega_1 \to \text{eye}) \times$$
$$\cos\theta_1 \cos\theta_2 \, \mathrm{d}\omega_1 \mathrm{d}\omega_2$$



This really is just as simple as going from two nested 1D integrals to a 2D area integral!

Full Solution

- The full lighting solution is a sum over paths of all lengths

$$L(x,y) = \sum_{i=0}^{\infty} L_i(x,y), \qquad \text{with } L_0(x,y) = E(P_1 \leftarrow \text{eye})$$

- Notice how we've "unwrapped" the recursive rendering equation into a sum of terms
  - n bounce lighting is an integral over $\text{screen} \times \underbrace{\Omega \times \ldots \times \Omega}_{n \text{times}}$
  - This is really the same as directly evaluating the terms of the Neumann series E + TE + TTE + ...

Sampling Paths

- "Local path sampling" proceeds bounce to bounce, always importance sampling according to local BRDF
- That is, for each sample (path):
  - First sample screen (x, y), then trace ray
  - At primary hit, choose outgoing direction ω_1, trace ray
  - At secondary hit, choose outgoing direction ω_2
  - Apply local PDFs at each step.. justification below

- Denote the full path $\bar{x} = (x, y, \omega_1, \omega_2, \ldots)$
  - Then $p(\bar{x}) = p(x, y)\, p(\omega_1)\, p(\omega_2) \ldots$
  - (This assumes independent choices at each bounce)
  - Easy to implement

Brute Force Path Tracing, Eye Part

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

$$+\ E(x \to \mathbf{v})$$

```
for each pixel
 Lout = 0, w=0
 for i=1 to #samples
  generate xi,yi inside pixel with p(x,y)
  ray_i = generatecameraray(xi,yi)
  Lout += f(xi,yi) * trace(ray_i)/p(x,y)
  w += f(xi,yi)/p(x,y)
 endfor
 L(pixel) = Lout/w
endfor
```

(Assuming, for simplicity, that only one pixel filter is nonzero. Look back a few slides for full treatment.)

Brute Force Path Tracing

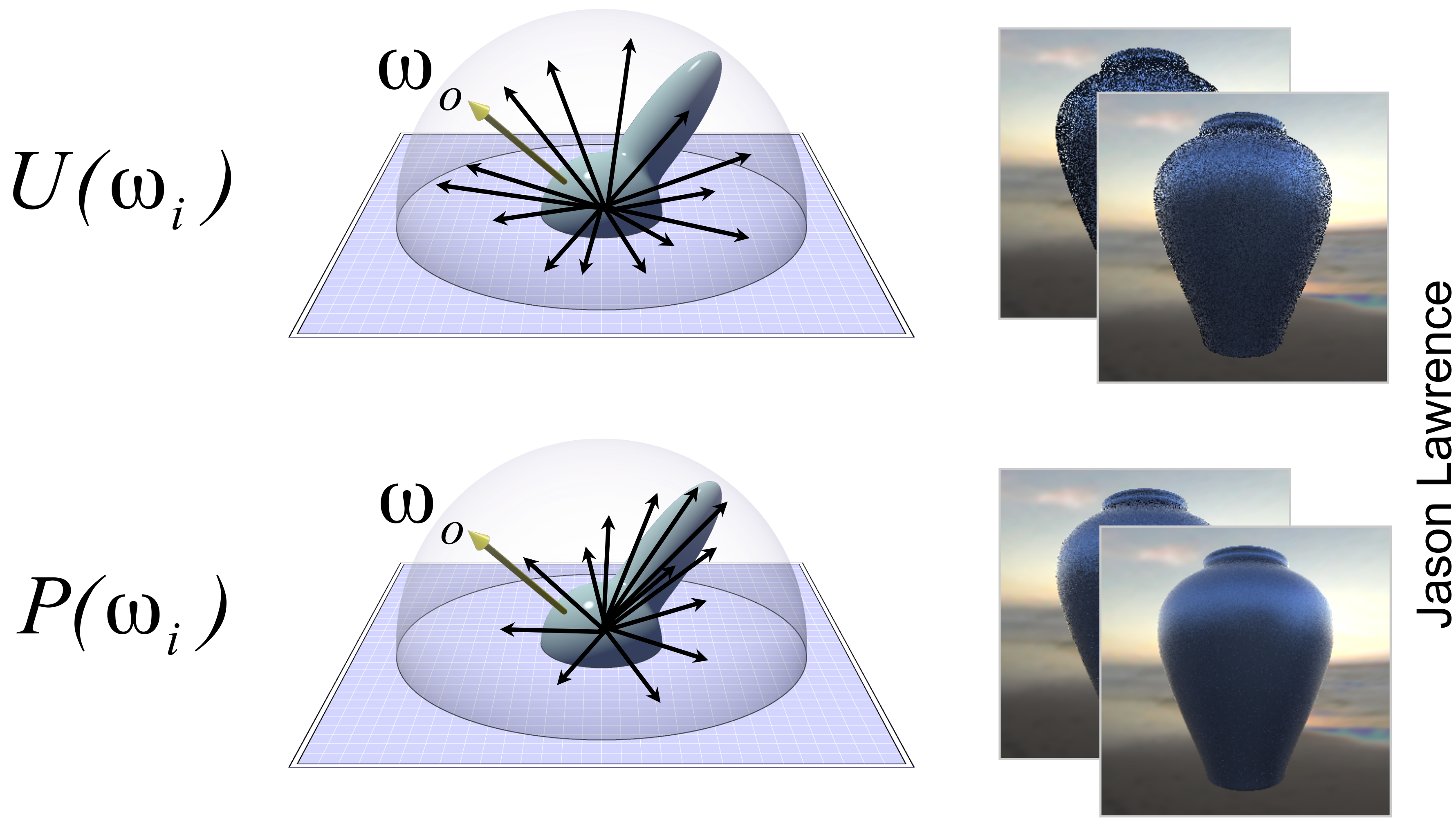
$$L(x \rightarrow \mathbf{v}) = \int_\Omega L(x \leftarrow \mathbf{l})\, f_r(x, \mathbf{l} \rightarrow \mathbf{v})\, \cos\theta \,\mathrm{d}\mathbf{l}$$

$$+ E(x \rightarrow \mathbf{v})$$

```
trace(ray)
 hit = intersect(scene, ray)
 result = emission(hit,-dir(ray))  // 0 if no light
 // sample outgoing direction
 [w,pdf] = sampleReflection(hit,dir(ray))
 // recursively estimate incoming radiance, apply BRDF
 result += BRDF(hit,-dir(ray),w)*
           cos(theta)*
           trace(ray(hit,w))/pdf
 return result
// when we apply the PDF like this, we are implicitly
// multiplying them for all bounces like shown before
```

Notes

- `sampleReflection()` chooses a direction with which to estimate reflectance integral for indirect part
  - I.e. importance sample according to BRDF

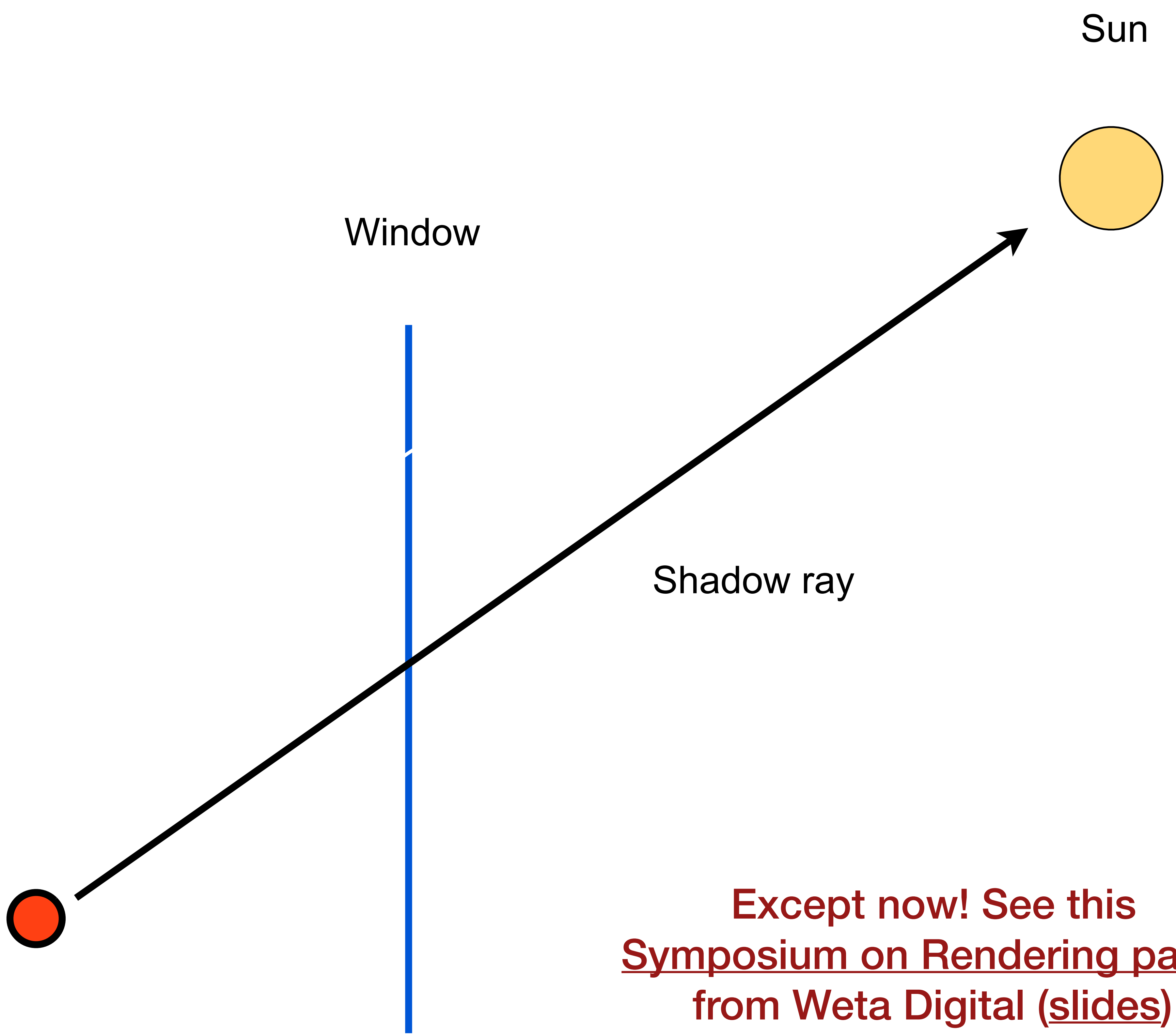


Why “Brute Force”?

- We’re waiting for the sampler to hit the light on its own
  - Often not a good idea
  - But sometimes we can’t do too much else
  - Think of an architectural model where all the light comes through several specular bounces through windows

- In simple cases we can help by adding an explicit direct light sampling step to each bounce

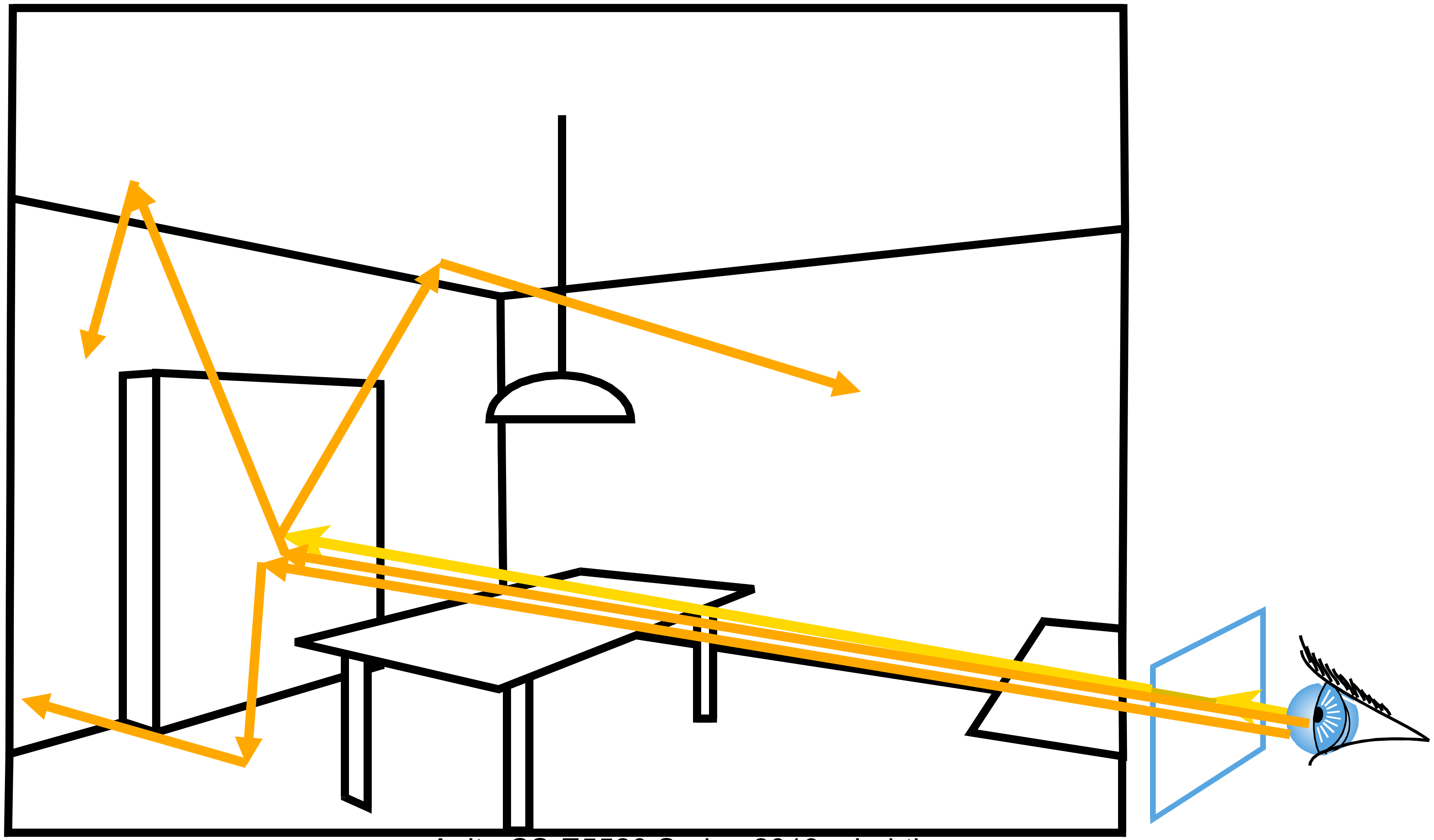
~~This Doesn't Work!~~



Except now! See this
Symposium on Rendering paper
from Weta Digital (slides)

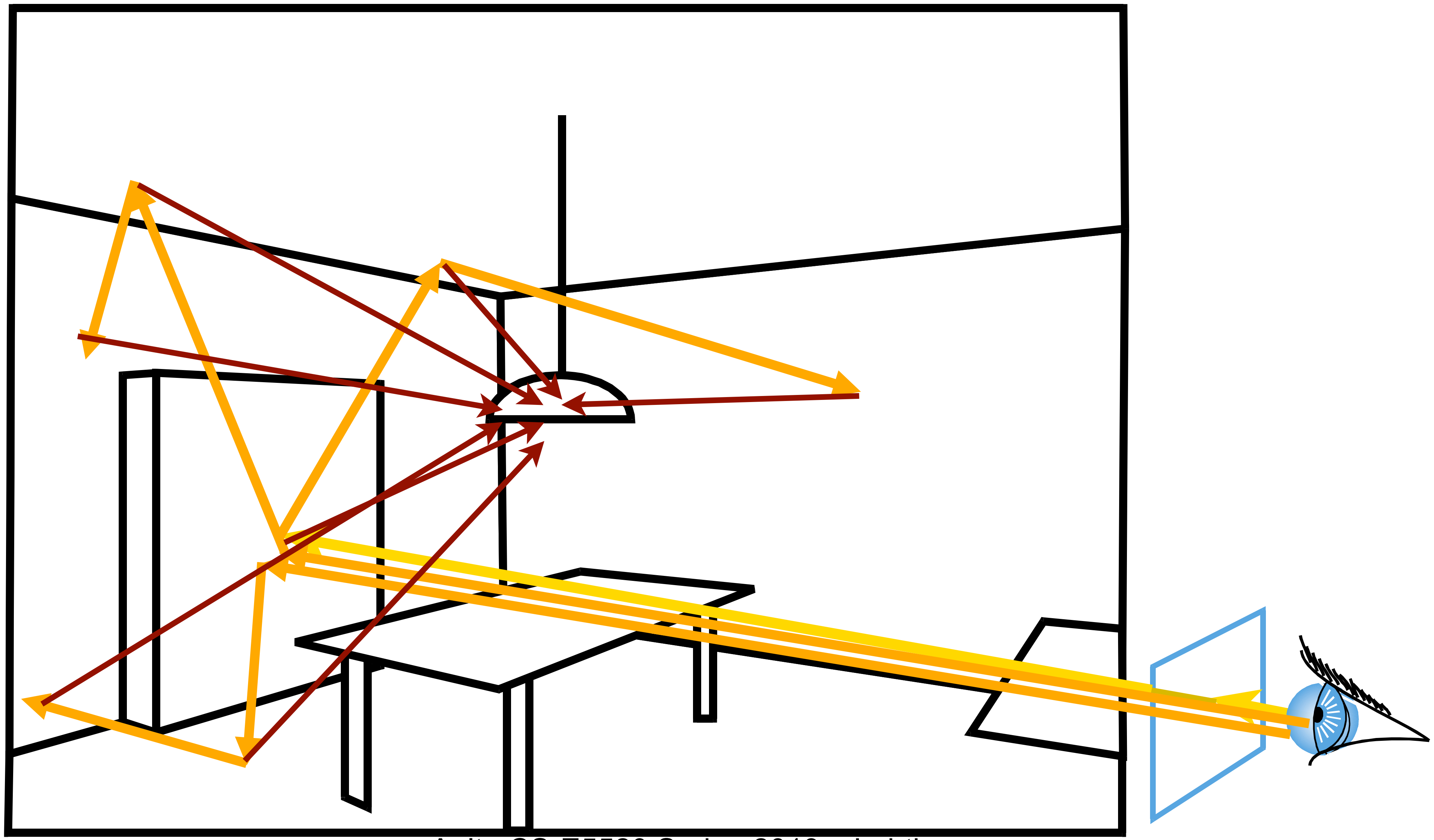
Brute Force Path Tracing

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



Path Tracing w/ Light Sampling

- At each hit, also sample a light and shoot a shadow ray
- The standard way of doing path tracing
- Also called “next event estimation”



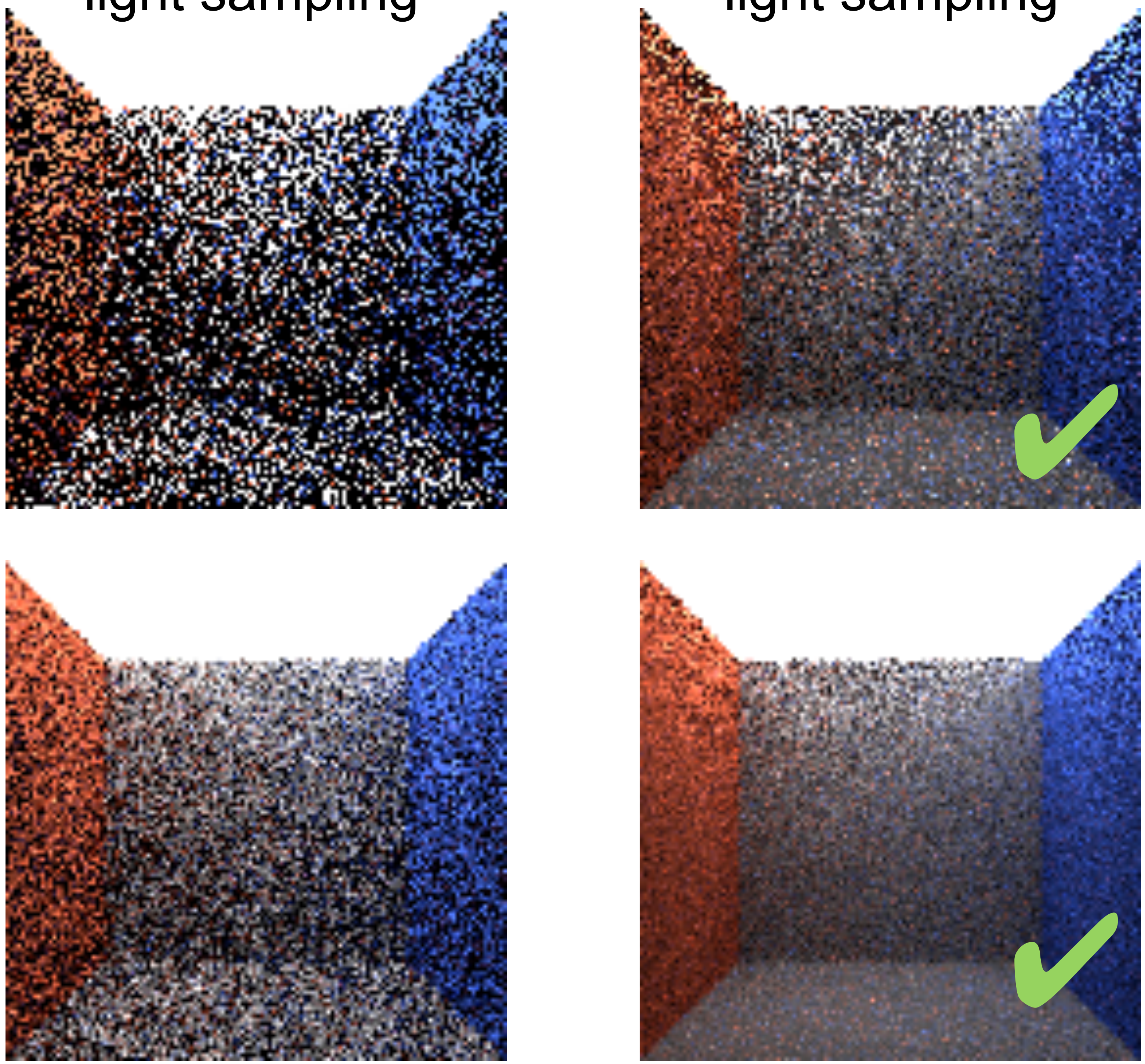
Importance of Sampling the Light

Without explicit light sampling

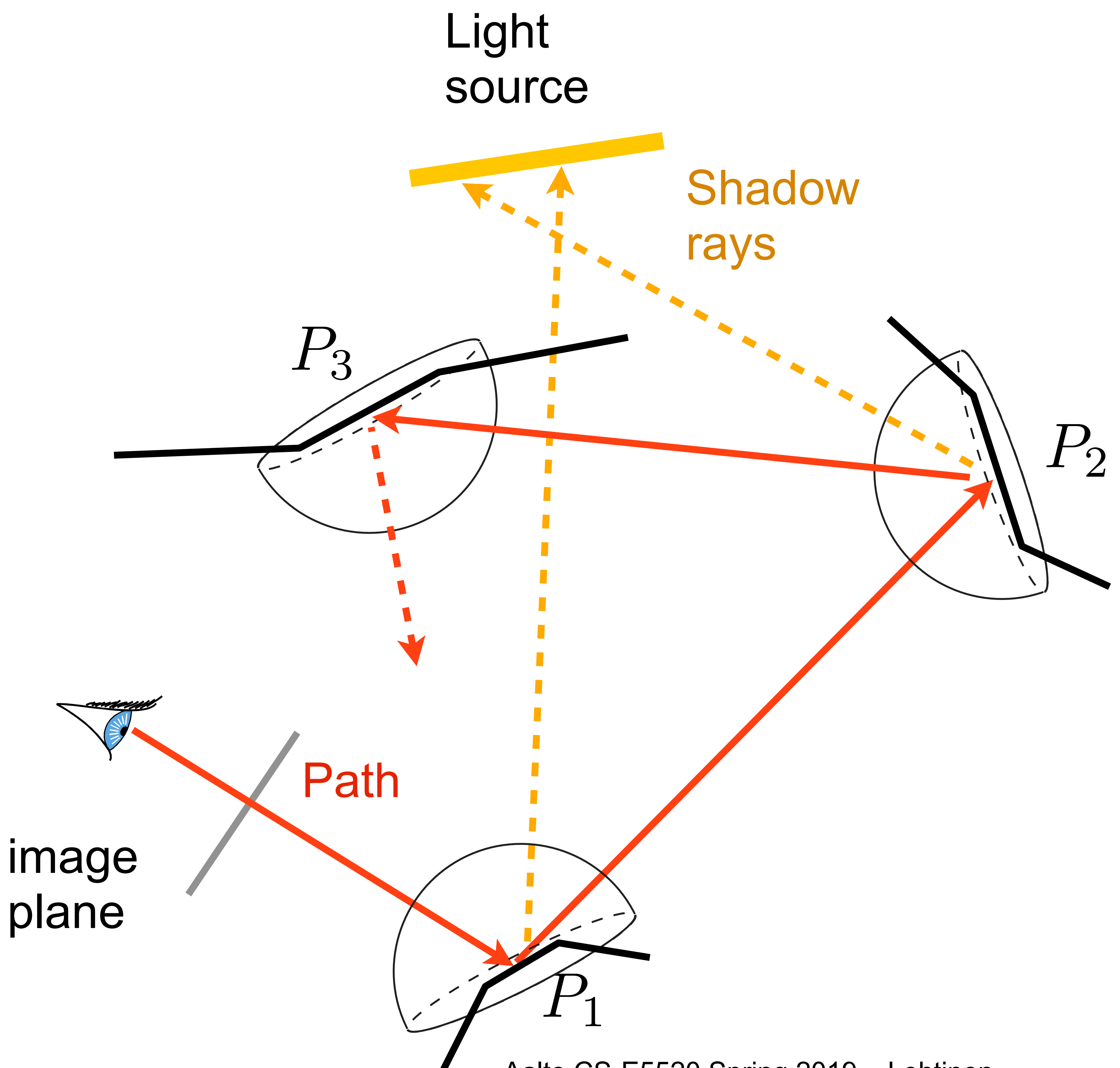
With explicit light sampling

1 path per pixel

4 paths per pixel



Path Tracing w/ Light Sampling



Interpretation of Shadow Rays

- Recall: the full lighting solution is a sum over paths of all lengths

$$L(x,y) = \sum_{i=0}^{\infty} L_i(x,y), \qquad \text{with } L_0(x,y) = E(P_1 \leftarrow \text{eye})$$

- Notice how we've "unwrapped" the recursive rendering equation into a sum of terms
  - n bounce lighting is an integral over $\text{screen} \times \underbrace{\Omega \times \ldots \times \Omega}_{n\text{times}}$ (brute force PT)
  - But now we've replaced the final hemisphere with lights by solid-angle-to-area conversion: $\text{screen} \times \omega \times \omega \ldots \times \text{lights}$

Path Tracing Pseudocode

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

$$+\ E(x \rightarrow \mathbf{v})$$

```
trace(ray)
 hit = intersect(scene, ray)
 if ray is from camera // only add “very direct” light here
  result = emission(hit,-dir(ray))
 [y,pdf1] = sampleLightsource()     // pick shadow ray dest.
 // G(hit,y) contains the usual cosine/r^2 of the
 // hemisphere-to-area variable change
 result += V(hit,y)*E(y,y->hit)*BRDF*cos*G(hit,y)/pdf1
 [w,pdf] = sampleReflection(hit,dir(ray)) // like before
 result += BRDF(hit,-dir(ray),w)*
           cos(theta)*
           trace(ray(hit,w))/pdf
 return result
```

Notes 2

- `sampleLightsource`() picks a point on the light source and evaluates its PDF
  - You're doing this in the first part of your radiosity assignment
  - ..and we saw this already on the first MC lecture
  - We're (again) applying the solid angle-to-area variable change (i.e. we're integrating over the surface of the light source)

- When you have multiple light sources, you pick *one* at random, and build this into the PDF
  - Simple: just multiply the light source $p(y)$ with the probability of picking that particular light source

Picking Lights

- It makes sense to importance sample the light you pick
- E.g. doesn’t make sense to sample dim, far-away lights as often as bright, nearby ones!

One Small Problem

One Small Problem

- Yes, it doesn't terminate if you just keep going
  - Fortunately, there's still something we can do!

Russian Roulette

- The usual MC estimate is $E\{\frac{f(x)}{p(x)}\}_p$
  - f/p is a random variable because x is a random variable

Russian Roulette

- The usual MC estimate is $E\{\frac{f(x)}{p(x)}\}_p$
  - *f*/*p* is a random variable because *x* is a random variable
- Let's multiply this by another specially constructed random variable R
  - R(x)=0 with probability $\alpha(x)$, and $R = 1/(1-\alpha)$ otherwise
  - Also assume α and *x* are uncorrelated (independent). Then:

$$E\{\frac{R \cdot f(x)}{p(x)}\} = E\{R\}\, E\{\frac{f(x)}{p(x)}\} = E\{\frac{f(x)}{p(x)}\}$$

This step requires independence

Russian Roulette: What is Going On?

- R(x)=0 with probability $\alpha(x)$, and $R = 1/\alpha$ otherwise

$$E\{\frac{R \cdot f(x)}{p(x)}\} = E\{R\}\, E\{\frac{f(x)}{p(x)}\} = E\{\frac{f(x)}{p(x)}\}$$

- *We've given ourselves permission to sometimes replace the value of the integrand with zero without introducing bias to the result*
  - When we don't set it to zero, we multiply the result by $1/\alpha$
- This means, for instance, that we can probabilistically terminate light paths without tracing them to infinity

Path Tracing w/ RR

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

$$+ \; E(x \to \mathbf{v})$$

```
trace(ray)
 hit = intersect(scene, ray)
 if ray is from camera // only add "very direct" light here
  result = emission(hit,-dir(ray))
 [y,pdf1] = sampleLightsource()     // pick shadow ray dest.
 result += E(y,y->hit)*BRDF*cos*G(hit,y)/pdf1
 [w,pdf] = sampleReflection(hit,dir(ray))
 // russian roulette with alpha=0.5
 terminate = uniformrandom() < 0.5
 if !terminate
  result += BRDF(hit,-dir(ray),w)*
            cos(theta)*
            trace(ray(hit,w))/pdf/0.5   // 1/0.5 =mult. by 2!
 return result
```

“Path Space”

- Earlier we wrote n-bounce lighting as a simultaneous integral over n hemispheres
- We can just as well integrate over surfaces instead
  - We just need to add in the geometry terms like before
    - $1/r^2$, visibility, the other cosine
- The space of paths of length n is then simply

$$\underbrace{S \times \ldots \times S}_{n \text{ times}}$$

  with S being the set of 2D surfaces of the scene
- See Eric Veach’s PhD

What Does It Look Like?

- Jacco Bikker's Brigade Real Time GPU Path Tracer (video)
  - Multiple GPUs + post processing for removing noise
  - A few years old already, but gets the point across

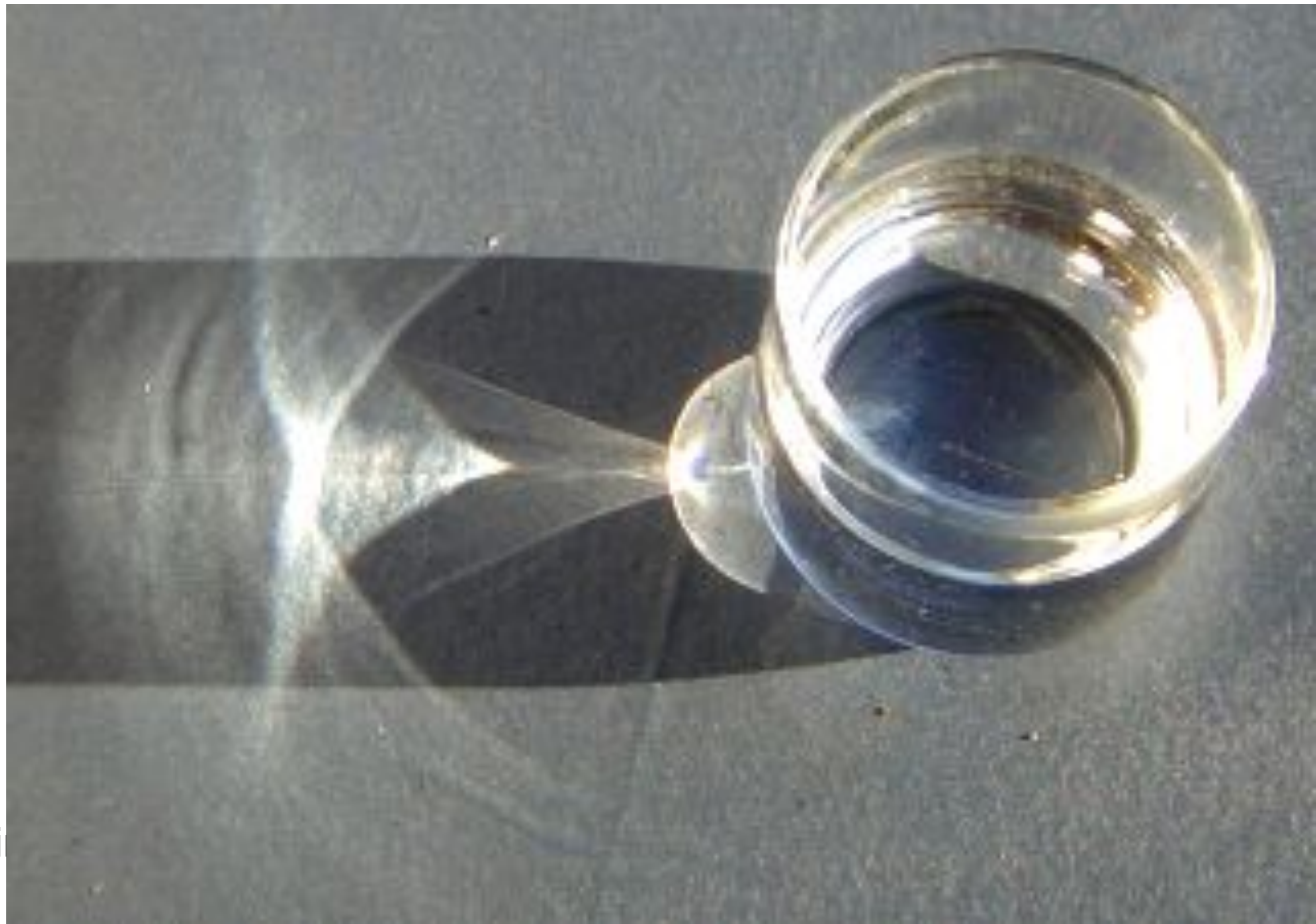
- See http://raytracey.blogspot.co.nz/2012/08/real-time-path-traced-brigade-demo-at.html

Bigger Picture

- We are shooting rays from the camera, propagating them along, and kind of hoping we will find light
  - Actively try to hit it by the light source samples

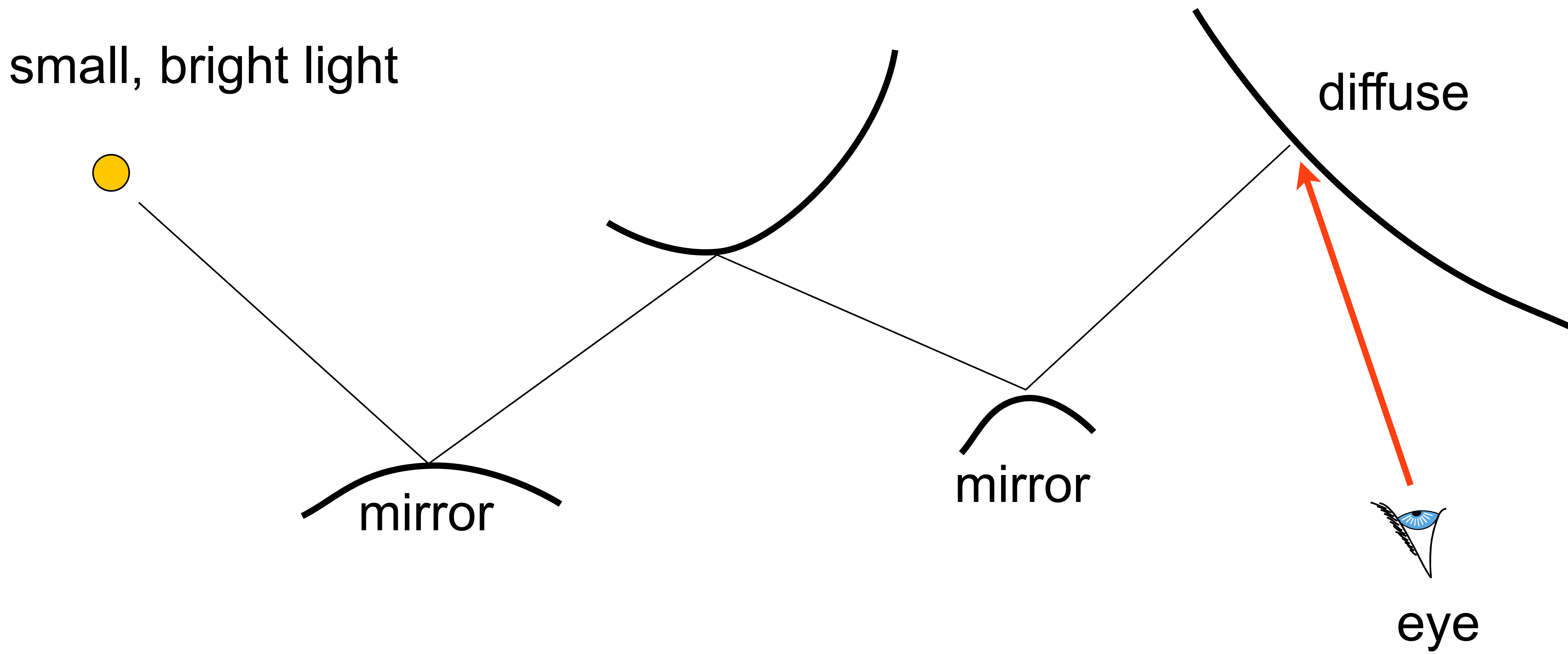
- What about more difficult cases?
  - In a *caustic*, the light propagates through a series of specular refractions and reflections before hitting a diffuse surface

wikipedia



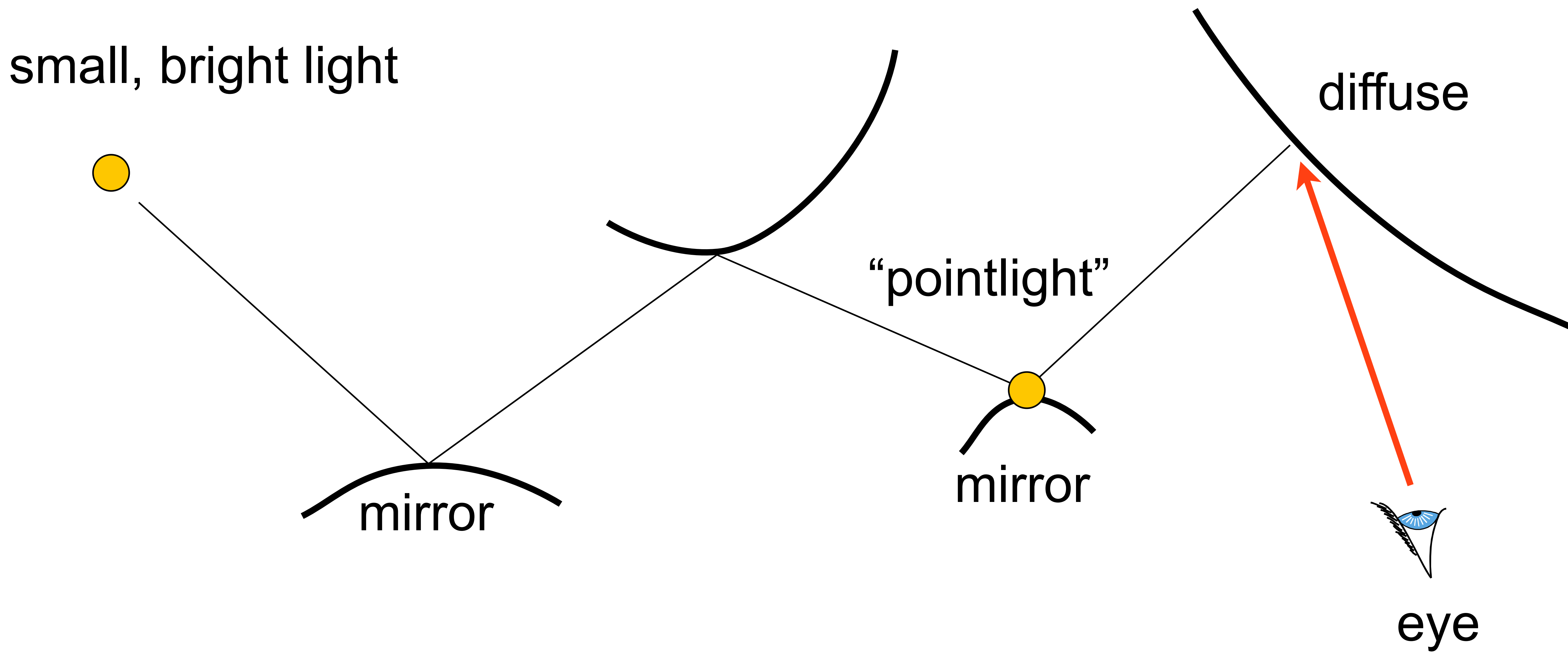
Problem With Caustics

- Think of an almost pointlike light shining through a sequence of curved mirrors onto a receiver



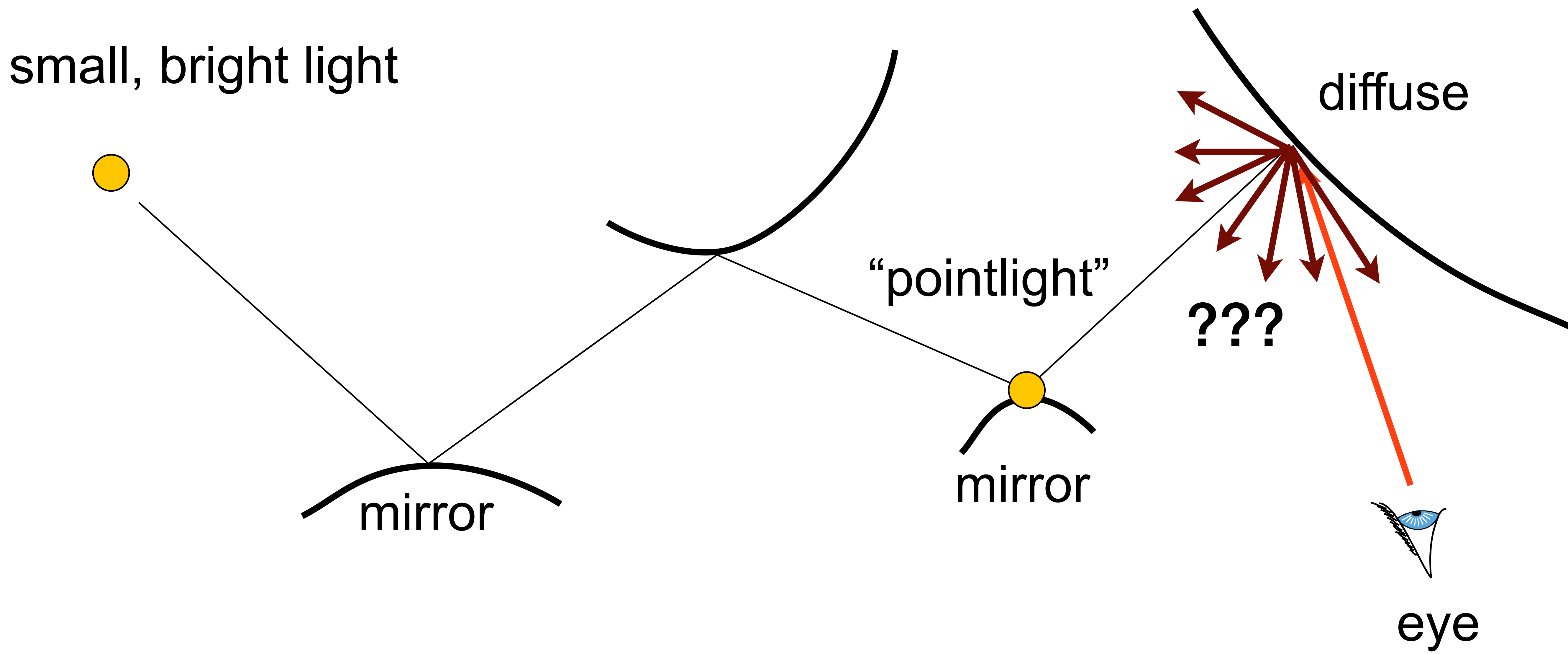
Problem With Caustics

- The point hit by the eye ray effectively sees a pointlight in the direction of the last mirror



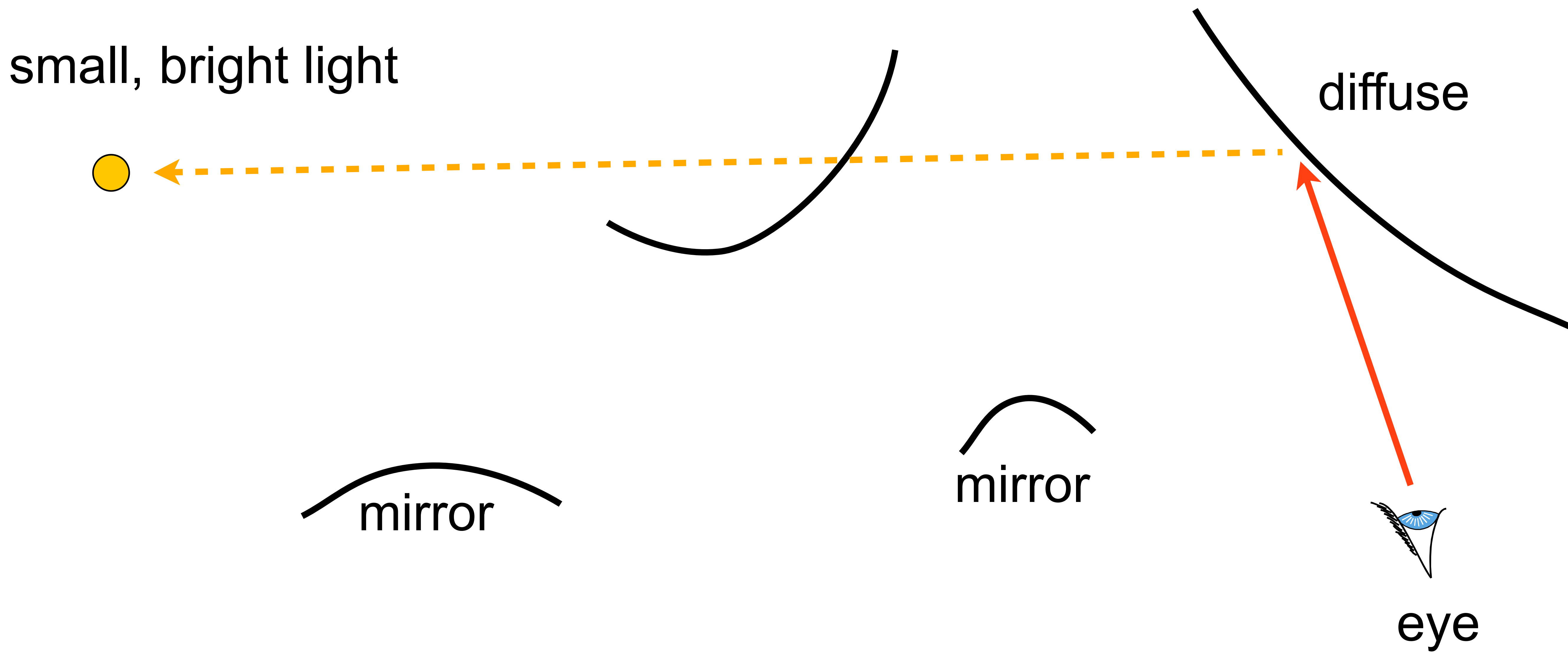
Problem With Caustics

- The point hit by the eye ray effectively sees a pointlight in the direction of the last mirror
  - How does the cosine importance sampler know that?



Problem With Caustics

- All we can do is shoot shadow rays towards the light
  - Not very helpful here!



Problem With Caustics

- All we can do is shoot shadow rays towards the light
  - Not very helpful here!

To be continued…