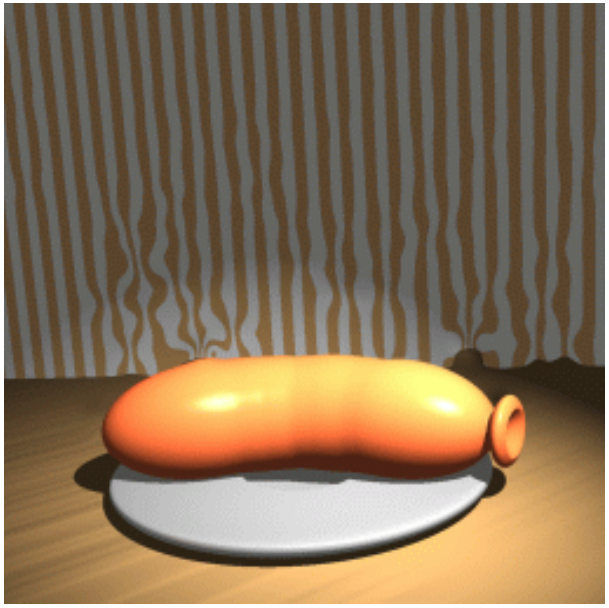
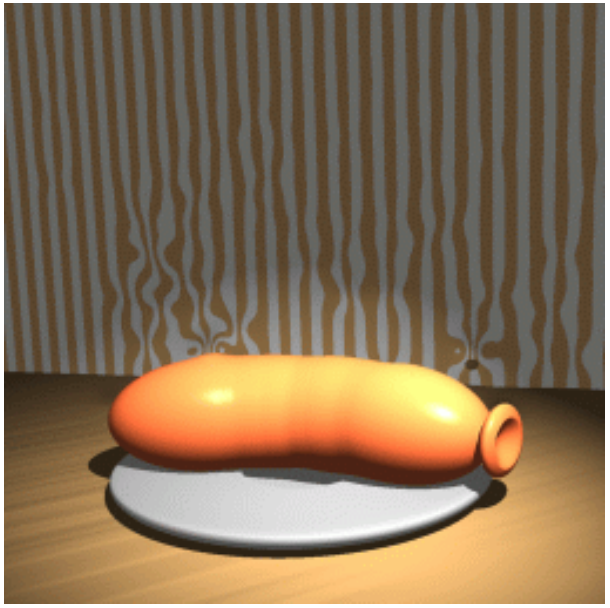
Path Tracing

Prof. Fabio Pellacini

Models of light

- Geometric optics
  - light travel in straight lines
  - light particles, i.e. photons, do not interact with each other
  - describes: emission, reflection/refraction, absorption

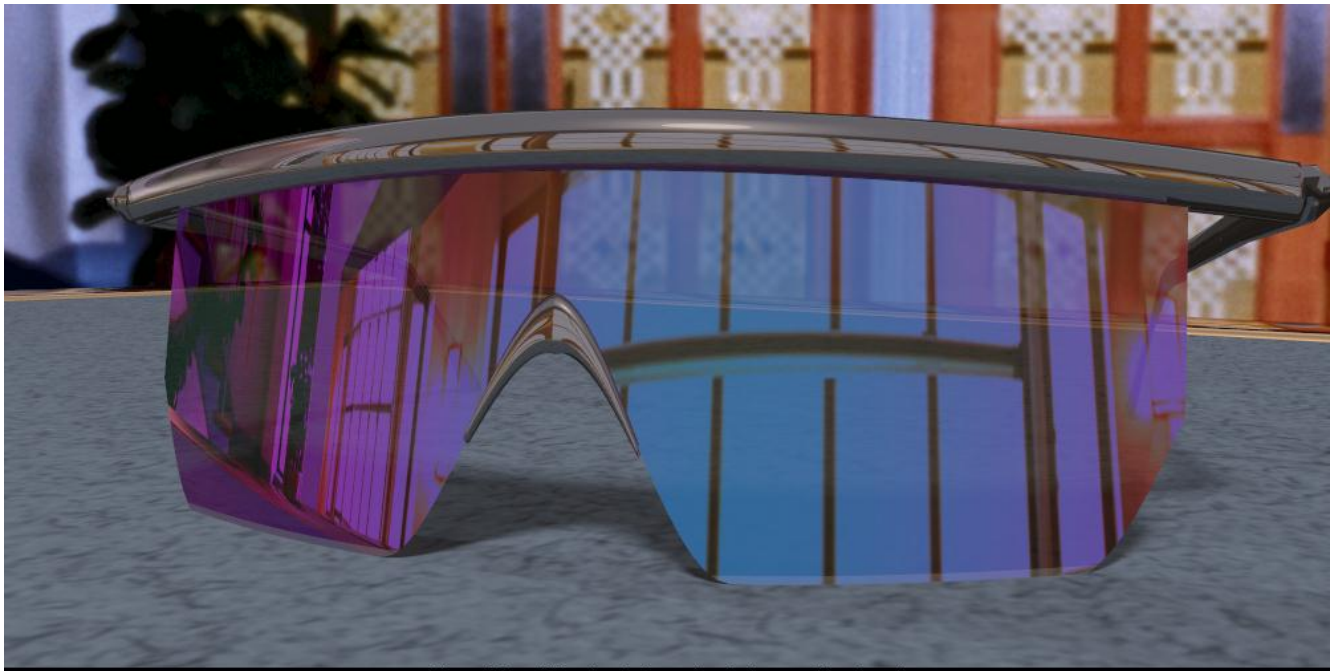




[Stam et al., 1996]

Models of light

- Wave optics
  - light particles interact with each other
  - describes: diffraction, interference, polarization



[Gondek et al., 1997]

Models of light

- Quantum optics
  - light particles are like any other quantum particle
  - describes: fluorescence, phosphorescence

[Glassener et al., 1997]

Real-world image formation

- Assumption: geometric optics
- Images are formed by a continuous process
  - photons are emitted from light sources
  - they travel in straight lines until the hit a surface
  - at the surfaces, they are either absorbed or reflected
  - the reflected ones continue the same process
  - eventually, some hit the eye
  - the eye measures how many photons hit it

Physically-based rendering

- *All realistic image synthesis methods* simulate, to a different degree of approximation, the physics of lights
  - main idea: simulating the physics of light ensures realistic images
    - obviously we also need realistic input scenes
  - some algorithms simulate "virtual photons", tracing them from the lights to the eye as real photons would behave

- *Rendering equation:* describes formally the physics of light
  - common formulation based on geometric optics
  - describes nearly all illumination effects and surface materials

Physically-based rendering

- *Path tracing:* an algorithm that solves the rendering equation
  - produces *physically-correct* images
  - simulates the physics of light using backwards raytracing, i.e. tracing paths from eye to lights
  - used in architecture, product design and movies
  - drawback: slow to compute
  - many other algorithms exists, but not covered here

Path tracing – informal

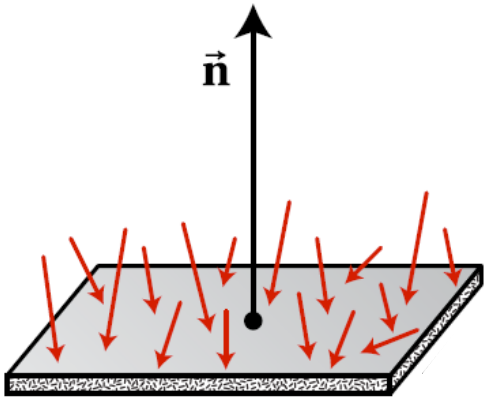
[Disney/Youtube]

Rendering Equation

Radiant power

- Energy *Q* [J]: illumination energy
  - what a detector measures
  - intuition: how many photons hit a detector
- Power *P* or flux [W=Js^{-1}]: energy per unit time
  - intuition: how many photons hit a surface per second
  - other units derived from this

$$P = \frac{dQ}{dt}$$

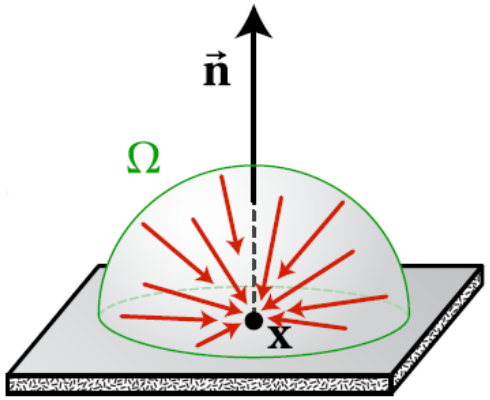


[Woiciech Jaroz]

Irradiance

- Irradiance *E* [Wm^{-2}]: power per unit area
  - measured at each location on a surface
  - intuition: considers photons coming from all direction to a small area around a single surface point
  - irradiance is used for incoming light
  - radiosity is the term used for outgoing light

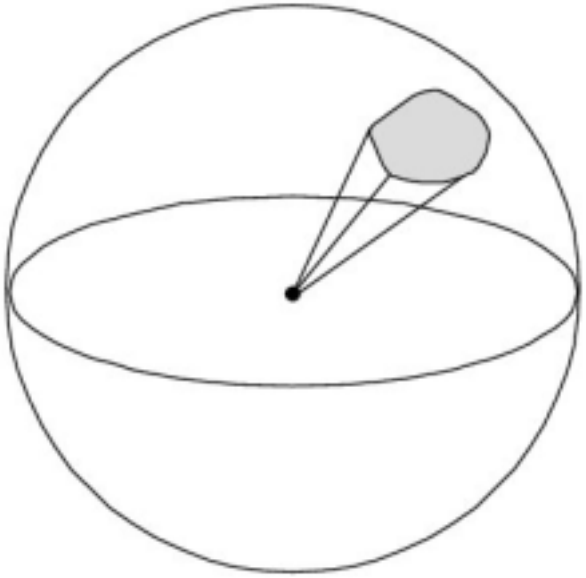
$$E(\mathbf{x}) = \frac{dP}{dA_{\mathbf{x}}}$$



[Woiciech Jaroz]

Solid Angle

- Solid angle [sr]: area of the unit sphere subtended by an object
  - analogous to angles in 2D that are the arc length of the unit circle
  - intuition: measure the "size" of a light beam
- Differential solid angle [sr]: infinitesimal solid angle around a direction
  - for the direction **d**, indicated as $d\omega_{\mathbf{d}}$



[Jaroslav Krivanec]

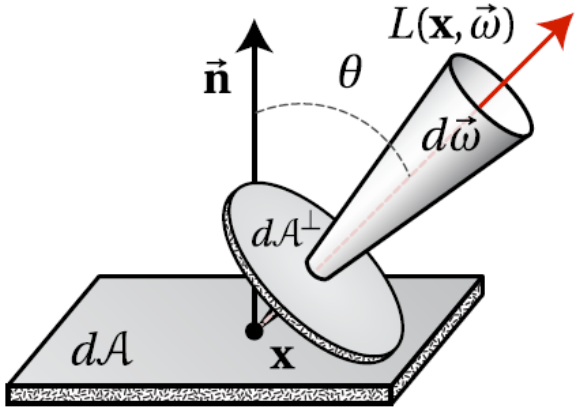
Radiance

- Radiance *L* [$Wm^{-2}sr^{-2}$]: power per unit projected area per unit solid angle
  - measured at each surface location and for each direction
  - intuition: consider only light in a thin beam around a direction
  - "projected area": account for orientation of surface
  - radiance is defined for incoming, outgoing and reflected light
  - most important quantity: *what our eyes see*
  - HDRs store radiance (scaled)

$$L(\mathbf{x}, \mathbf{i}) = \frac{d^2 P}{dA_{\mathbf{x}} d\omega_{\mathbf{i}} (\mathbf{i} \cdot \mathbf{n})}$$

$$dA_{\mathbf{x}}^{\perp} = dA(\mathbf{i} \cdot \mathbf{n})$$

$$d\omega_{\mathbf{i}}^{\perp} = d\omega_{\mathbf{i}}(\mathbf{i} \cdot \mathbf{n})$$



[Woiciech Jaroz]

Radiance

- Typical values [cd m-^{2}]
  - surface of the sun: 2,000,000,000
  - cloudy sky: 30,000
  - clear day: 3,000
  - overcast day: 300
  - surface of the moon: 0.03

[Pat Hanrahan]

Radiance properties

- Radiance depends on the wavelength
  - we just use RGB vector to represent it that samples the spectrum at three wavelength
  - good enough for most cases

$$L(\mathbf{x}, \mathbf{o}, \lambda) \approx [L_R(\mathbf{x}, \mathbf{o}), L_G(\mathbf{x}, \mathbf{o}), L_B(\mathbf{x}, \mathbf{o})]^T$$

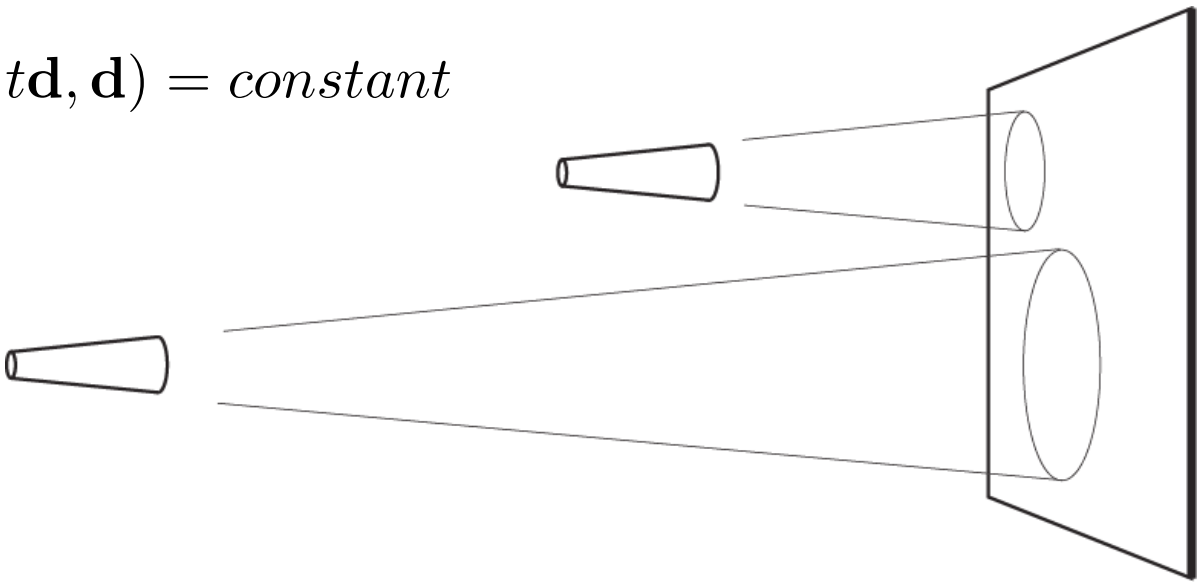
- Incoming radiance L_i is not the same as outgoing radiance L_o
  - photons that reach the surface are not the same as the ones that leave it
  - if not specified, we mean outgoing radiance

$$L_o(\mathbf{x}, \mathbf{d}) = L(\mathbf{x} \rightarrow \mathbf{d}) \neq L(\mathbf{x} \leftarrow \mathbf{d}) = L_i(\mathbf{x}, \mathbf{d})$$

Radiance properties

- Radiance does not change along straight lines in empty space
  - comes from conservation of energy
  - approximately true also in air for short distances
  - reason why radiance is associated with rays in a raytracer
  - corollary: surface colors do not change when we move away

$$L(\mathbf{o} + t\mathbf{d}, \mathbf{d}) = constant$$

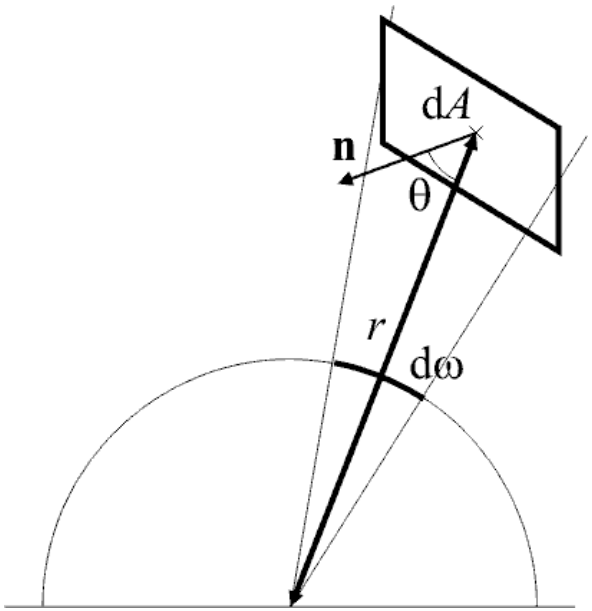


[Shirley]

Differential solid angle of surface

- Differential solid angle of a surface
  - consider a small surface area *dA*
  - project it onto the unit sphere by dividing by distance squared r^2
  - take into account orientation by correcting with the cosine

$$d\omega = \frac{dA \cos\theta}{r^2}$$



[Jaroslav Krivanec]

Proof of radiance invariance

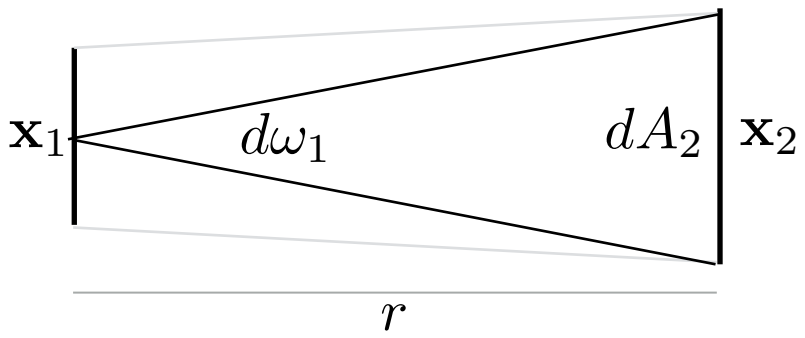
- Consider two surfaces orthogonal to each other
- For energy conservation all photons leaving a surface within the solid angle of the other have to reach the other surface

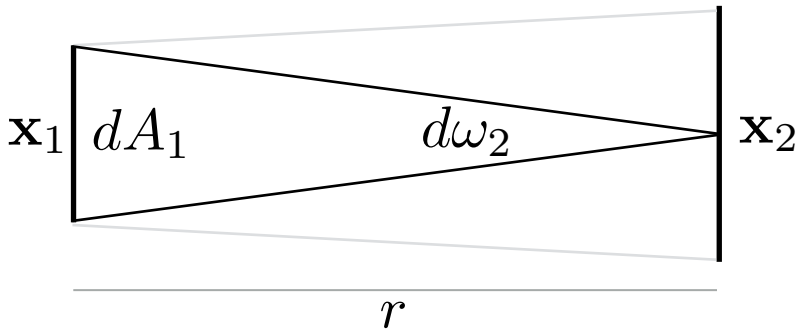
$$d^2P_1 = d^2P_2 \qquad \text{energy conservation}$$

$$L_1 d\omega_1 dA_1 = L_2 d\omega_2 dA_2 \qquad \text{by substitution}$$

$$L_1 (dA_2/r^2) dA_1 = L_2 (dA_1/r^2) dA_2 \qquad \text{by construction}$$

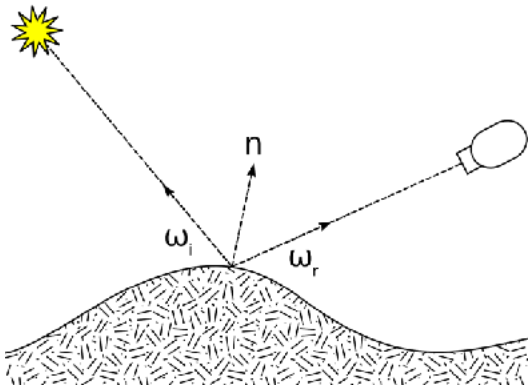
$$L_1 = L_2 = d^2P \frac{dA_1 dA_2}{r^2} \qquad \text{invariance along rays}$$

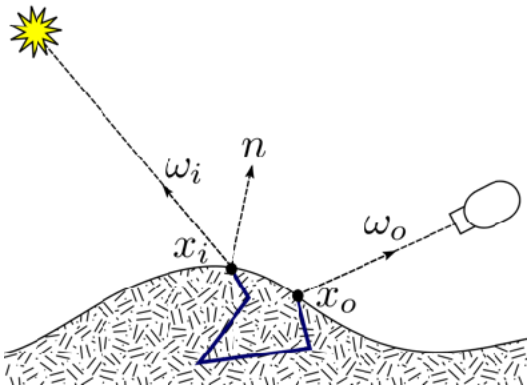




Scattering

- Photons that come to a surface from one direction are scattered in many other directions
- In graphics, we consider mostly three main scattering "modes"
- *Surface scattering* at the surface, e.g. metals and dielectrics
- *Subsurface scattering* in a thin region right below the surface, e.g. skin
- *Volumetric scattering* in a 3D volume, e.g. smoke and fog
- We'll focus on surface scattering

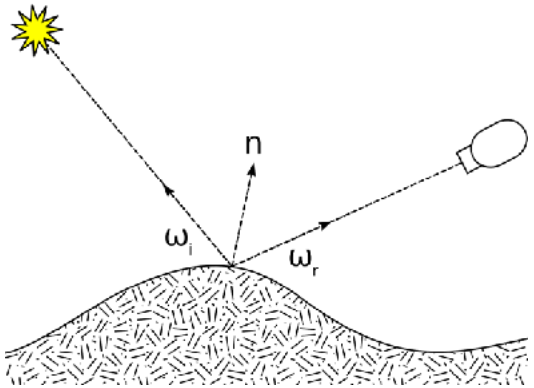




Surface scattering: BRDF

- BRDF [sr^{-1}]: *bidirectional reflectance distribution function*
- Defined as the ratio of the differential reflected radiance over the differential incident irradiance
- Depends on incoming and outgoing directions, and wavelength of light
  - do not track wavelength explicitly: just use colors for BRDF coeff.
- Intuition: probability density that a photon from a direction *i* of solid angle $d\omega_i$ is reflected in direction *o* of solid angle $d\omega_o$

$$f(\mathbf{x}, \mathbf{i}, \mathbf{o}) = \frac{dL_r(\mathbf{x}, \mathbf{o})}{dE_i(\mathbf{x}, \mathbf{i})} = \frac{dL_r(\mathbf{x}, \mathbf{o})}{L_i(\mathbf{x}, \mathbf{i})(\mathbf{n_x} \cdot \mathbf{i}) d\omega_\mathbf{i}}$$



Reflected radiance with BRDF

- From the BRDF definition, we write the differential reflected radiance as the product of the BRDF and the incoming radiance weighted by the cosine
  - write the point explicitly

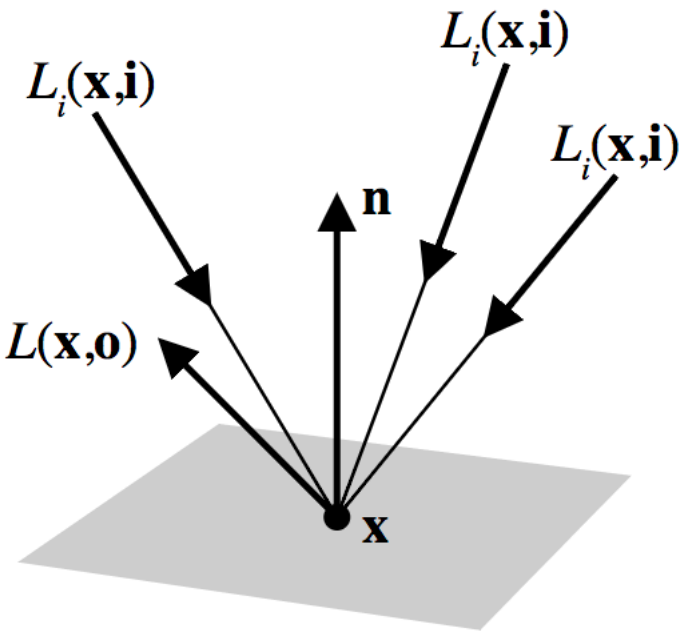
$$dL_r(\mathbf{x}, \mathbf{o}) = L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o})(\mathbf{n_x} \cdot \mathbf{i}) d\omega_\mathbf{i}$$

- The total reflected radiance is the integral of the differential reflected radiance for all directions over the hemisphere

$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H_\mathbf{x}} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o})(\mathbf{n_x} \cdot \mathbf{i}) d\omega_\mathbf{i}$$

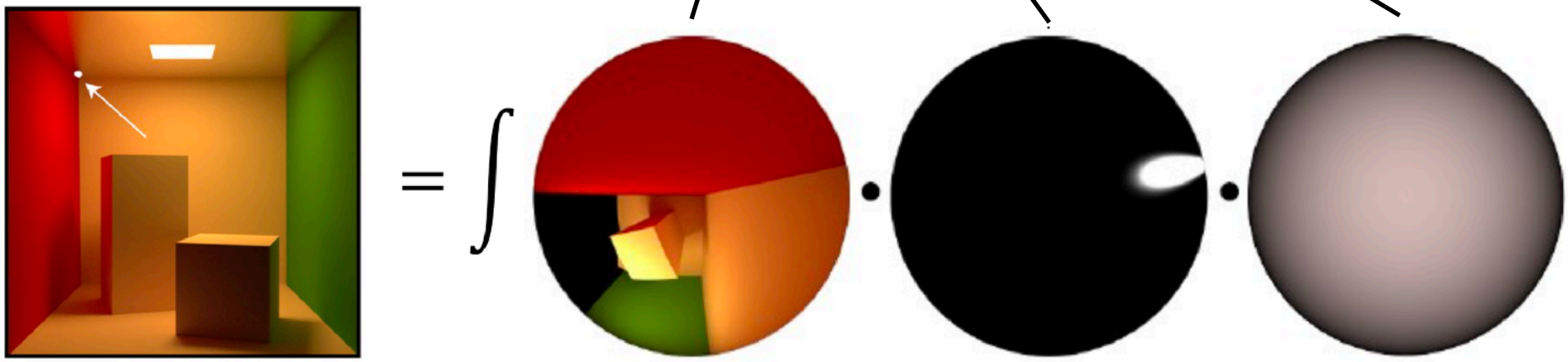
Reflected radiance

$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H_\mathbf{x}} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o})(\mathbf{n_x} \cdot \mathbf{i}) d\omega_\mathbf{i}$$



Reflected radiance

$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H_\mathbf{x}} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o})(\mathbf{n_x} \cdot \mathbf{i}) d\omega_\mathbf{i}$$



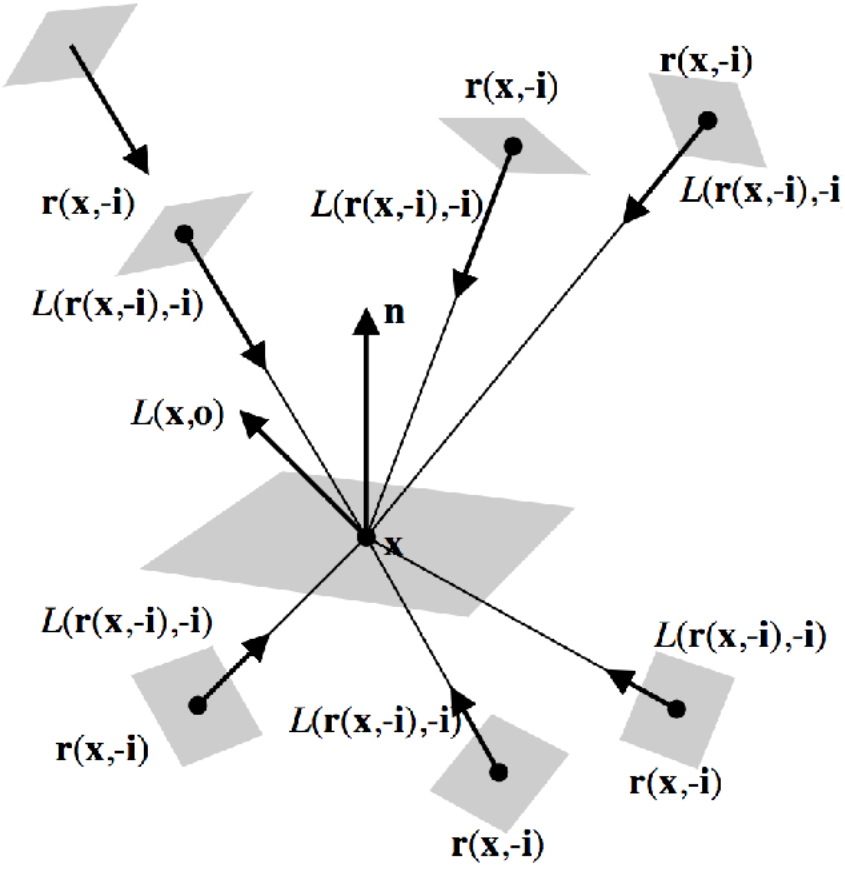
[Krivanec]

BSDF

- So far we only discussed opaque objects
- For translucent objects, such as glass, we have similar definitions
- BTDF: *bidirectional transmittance distribution function*
  - equiv. to BRDF but bottom hemisphere
- BSDF: *bidirectional scattering distribution function*
  - considers both top and bottom hemisphere
  - sum of BRDF and BTDF
- BSDF reflection integral is over full sphere around a point and takes the absolute value of the cosine

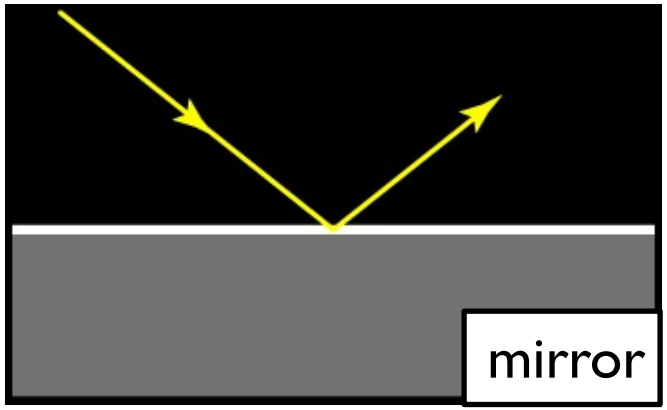
Reflected radiance with BSDF

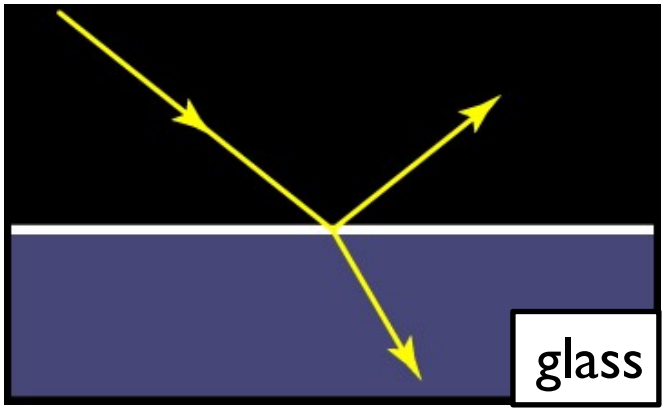
$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H^2_{\mathbf{x}}} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

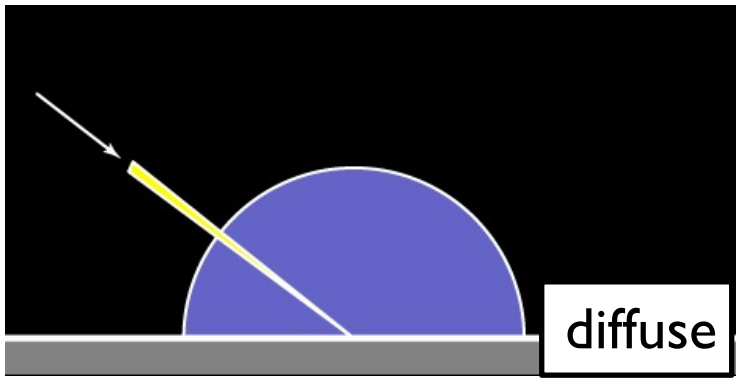


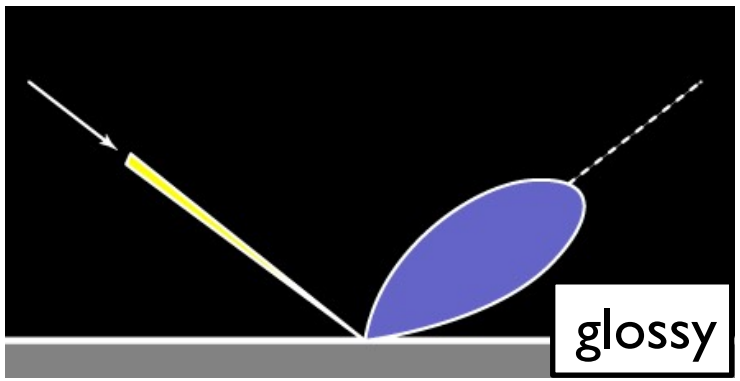
BSDFs for different materials

- Different materials are described by different scattering distributions









Properties of the BSDF

- Positivity: since photons cannot be "negatively" reflected

$$f(\mathbf{x}, \mathbf{i}, \mathbf{o}) \geq 0$$

- Helmotz reciprocity: can invert light paths

$$f(\mathbf{x}, \mathbf{i}, \mathbf{o}) = f(\mathbf{x}, \mathbf{o}, \mathbf{i})$$

- Energy conservation: all photons that comes in are either reflected or absorbed and no new photons is emitted

$$\forall \mathbf{i}. \int_{H^2} f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{o} \cdot \mathbf{n}| d\omega_{\mathbf{o}} \leq 1$$

Rendering equation

- Start from the equation for reflected radiance

$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H^2_{\mathbf{x}}} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

- Consider also the light that is naturally emitted by some surfaces, such as light source with an additional term

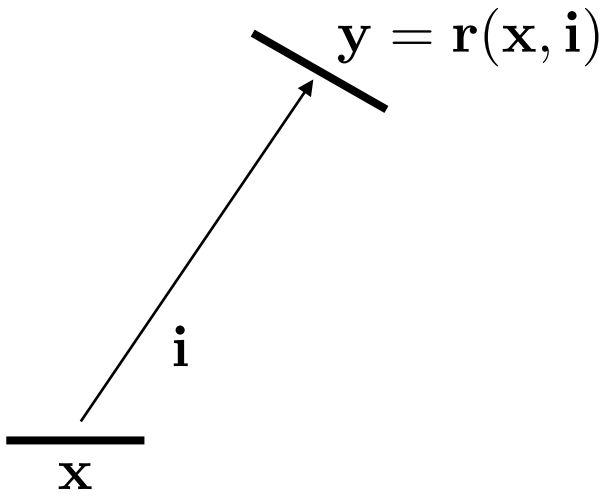
$$L_e(\mathbf{x}, \mathbf{o})$$

- Now we want to express the incoming radiance w.r.t. the outgoing radiance, using the property that radiance along a ray does not change

Radiance along a ray

- From the invariance of radiance along a ray, we can express the radiance incoming to a point as the radiance leaving other points
- Express this with recasting to find the first visible point

$$L_i(\mathbf{x}, \mathbf{i}) = L(\mathbf{r}(\mathbf{x}, \mathbf{i}), -\mathbf{i}))$$



Rendering equation

- Start from the equation for reflected radiance

$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H^2_{\mathbf{x}}} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

- Write incoming as the outgoing radiance os the first visible surface along the ray

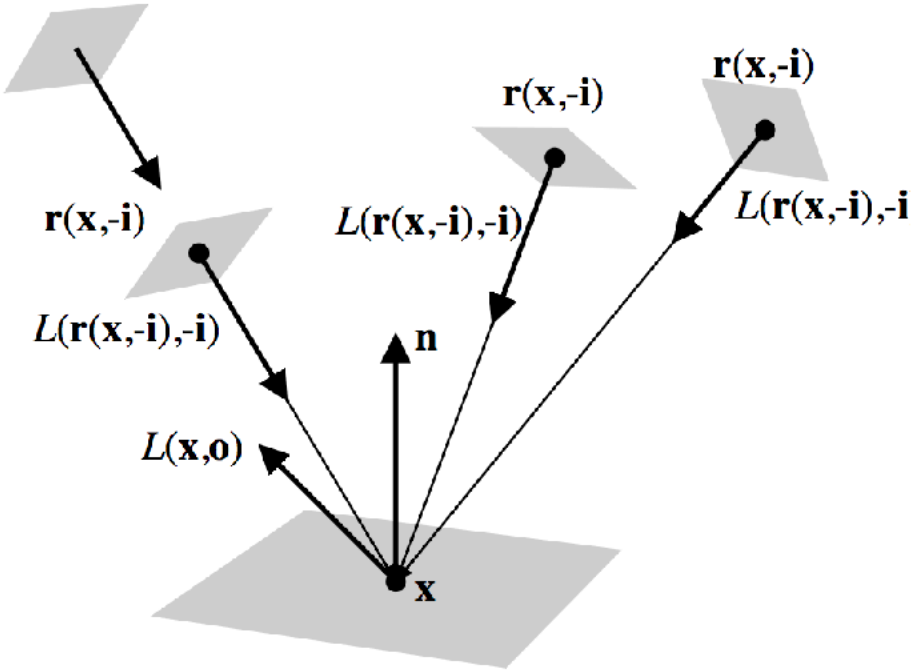
$$L_i(\mathbf{x}, \mathbf{i}) = L(\mathbf{r}(\mathbf{x}, \mathbf{i}), -\mathbf{i})$$

- Substitute the latter in the former and add the emitted radiance

$$L(\mathbf{x}, \mathbf{o}) = L_e(\mathbf{x}, \mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x}, \mathbf{i}), -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

Rendering equation

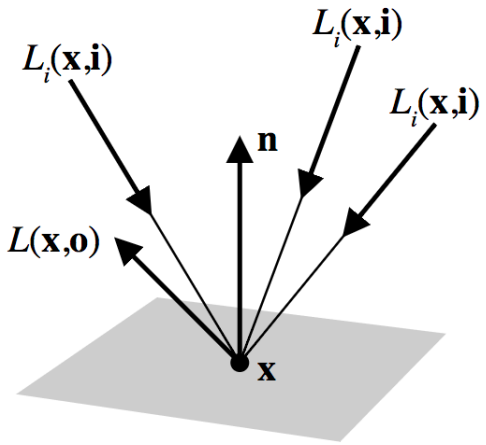
$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

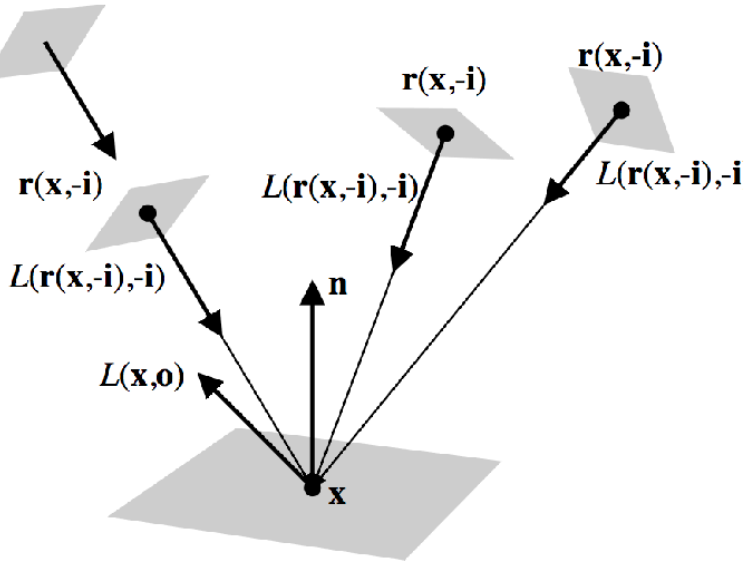


Reflection vs rendering equation

$$L_r(\mathbf{x}, \mathbf{o}) = \int_{H^2_{\mathbf{x}}} L_i(\mathbf{x}, \mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

$$L(\mathbf{x}, \mathbf{o}) = L_e(\mathbf{x}, \mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x}, \mathbf{i}), -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$



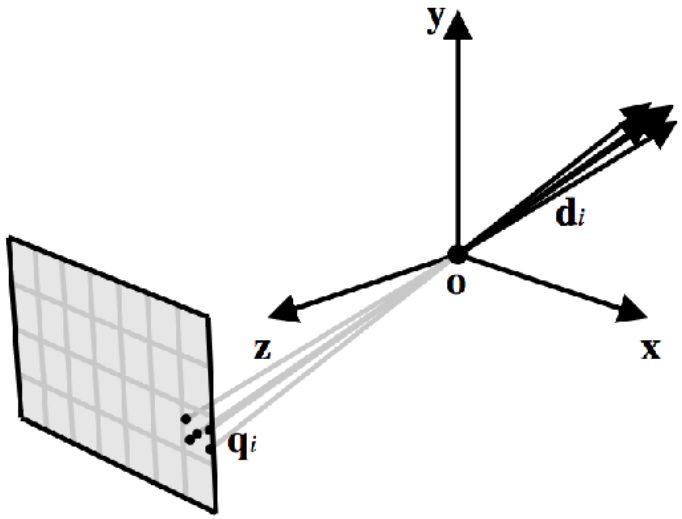


Rendering vs reflection equation

- **Reflection equation**: describes *local light reflection* at a point
  - *Integral* can be used to compute the reflected radiance given the *known incoming radiance*
- **Rendering equation**: condition of the global distribution of light
  - describe the energy balance in the scene at steady state
  - *Integral equation* with *unknown outgoing radiance* on both sides
- **Rendering**: compute the outgoing radiance for all visible points in the images by solving the rendering equation for those points and their view directions

Pixel radiance

- All previous formulation specify the radiance at a point in the scene
- To compute the pixel radiance we can think of measuring the incoming radiance in the sensor area covered by the pixel that can reach the pixel through the lens aperture
- Let us consider first the case of a pinhole camera depicted here



Pixel radiance

- We measure the pixel radiance as the integral, over the pixel surface, of the incoming radiance to each product weighted by the sensitivity of the sensor *s* and the cosine; this gives us proper anti-aliasing
- To be independent of the image resolution we divide by the pixel area

$$L_{pixel} = \frac{1}{A_{pixel}} \int_{\mathbf{q} \in pixel} L_i(\mathbf{q}, \mathbf{d}) s(\mathbf{q}, \mathbf{d}) |\mathbf{d} \cdot \mathbf{n_q}| dA_{\mathbf{q}} \text{ with } \mathbf{d} = \frac{\mathbf{e} - \mathbf{q}}{|\mathbf{e} - \mathbf{q}|}$$

- We now make the approximation that sensor has constant sensitivity and corrects for the cosine (this is your vignetting by the way)

$$L_{pixel} = \frac{1}{A_{pixel}} \int_{\mathbf{q} \in pixel} L_i(\mathbf{q}, \mathbf{d}) dA_{\mathbf{q}} \text{ with } \mathbf{d} = \frac{\mathbf{e} - \mathbf{q}}{|\mathbf{e} - \mathbf{q}|}$$

Monte Carlo integration

Discreet random variable

- Given a random processes with a finite number of outcomes
- A *discreet random variable* is a function that maps outcomes to measurable values, typically real numbers

$$X : \Omega \rightarrow D$$

- Example: rolling a dice
  - outcomes: the face of the dice
  - values of the variable: numbers 1, 2, 3, 4, 5, 6
- For simplicity in these notes, we consider just the range of the random variable and discuss its property over that
  - we assume there is a proper random process whose outcomes can be mapped in the range
- Disclaimer: next slides have math that is correct but not as formal!

Discreet random variable

- A *discreet random variable* is a variable that can randomly take one of a finite number of values with fixed probabilities

$$X \in D = \{v_i\}$$

- *Probability mass function*: probability of each value

$$X \sim p_X \quad p(x) = \Pr(X = x) \quad \sum_i p(v_i) = 1$$

- *Cumulative distribution function*: probability to be within an interval

$$c(x) = \Pr(X \leq x) \quad c(x) = \sum_{v_i \leq x} p(v_i) \quad c(\max(v_i)) = 1$$

- Example: rolling a dice

$$p(v_i) = 1/6 \qquad v_i = \{1, 2, 3, 4, 5, 6\}$$

Expected value and variance

- Expected value: weighted average of the variable values

$$E[X] = \sum_i v_i p(v_i)$$

- Variance: the expected value of the difference from the average

$$\sigma^2[X] = E[(X - E[X])^2] = E[X^2] - E[X]^2$$

- Example: expected value of rolling a dice

$$E[X] = (1 + 2 + 3 + 4 + 5 + 6)/6 = 3.5$$

$$\sigma^2[X] = ... \approx 2.92$$

Estimating expected value

- Estimate expected value by running an experiment
  - pick at random one of the variable values with probably *p*
  - lets call this value x_i
  - repeat *n* times and average the results

$$E[X] \approx \frac{1}{N} \sum_i^N x_i$$

- Larger *n* give better estimates
- Example: rolling a dice
  - roll 3 times: $\{3, 1, 6\} \rightarrow E[X] \approx (3 + 1 + 6)/3 = 3.33$
  - roll 9 times: $\{3, 1, 6, 2, 5, 3, 4, 6, 2\} \rightarrow E[X] \approx 3.51$

Law of large numbers

- As the number of trials goes to infinity, the probability that the average opt the observations is equal to the expected value will be equal to one.

$$\Pr\left(E[X] = \lim_{N\to\infty} \frac{1}{N}\sum\nolimits_i x_i\right) = 1$$

- This means that *the estimate of the expected value converges to the correct value*

Continuous random variable

- *One-dimensional continuous random variables*: similar to discreet random variables but the values are in the continuous domain

$$X \in D = [a, b)$$

- *Probability density function (pdf)*: probability that the variable is in within a differential interval around a given value

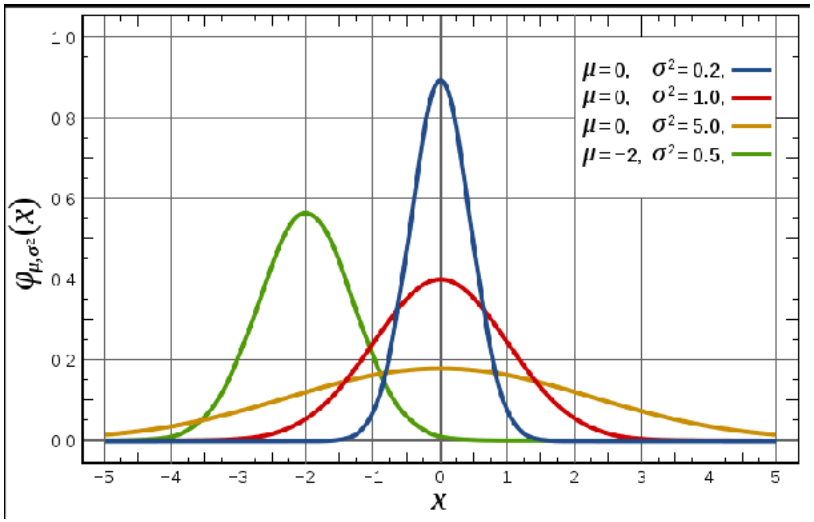
$$X \sim p_X \quad p(x)dx = d\Pr(x \leq X \leq x + dx) \quad \int_a^b p(x)dx = 1$$

- *Cumulative distribution function*: probability to be within an interval

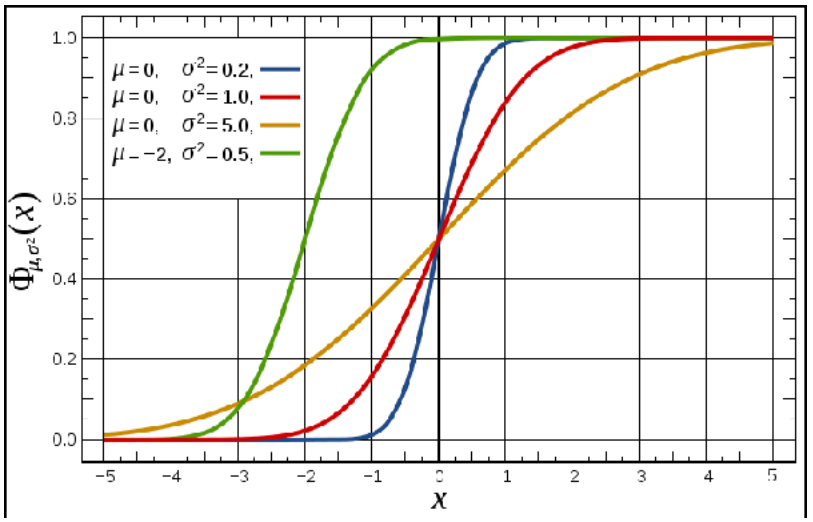
$$c(x) = \Pr(X \leq x) \quad c(x) = \int_a^x p(y)dy \quad c(b) = 1$$

Continuous random variable

example pdf: gaussian *p(x)*



example cdf: erf *c(x)*



Continuous random variable

- More generally, a *continuous random variable* is a multi-dimensional variable that can take a value in a arbitrary continuous domain

$$\mathbf{X} \in D$$

- *Probability density function*: probability that the variable takes a value within a differential domain around a point

$$\mathbf{X} \sim p_{\mathbf{X}} \quad p(\mathbf{x}) dA_{\mathbf{x}} = d\Pr(\mathbf{x} \in dD_{\mathbf{x}}) \quad \int_D p(\mathbf{x}) dA_{\mathbf{x}} = 1$$

- *Cumulative distribution function*: general formulation is complex
- Expected values and variance

$$E[\mathbf{X}] = \int_{\mathbf{x} \in D} \mathbf{x} p(\mathbf{x}) dA_{\mathbf{x}} \quad \sigma[\mathbf{X}] = \int_{\mathbf{x} \in D} |\mathbf{x} - E[\mathbf{X}]|^2 p(\mathbf{x}) dA_{\mathbf{x}}$$

Expected value of a function

- Expected value of a function of a continuous random variable

$$E[g(\mathbf{X})] = \int_{\mathbf{x} \in D} g(\mathbf{x}) p(\mathbf{x}) dA_{\mathbf{x}}$$

- Variance

$$\sigma^2[g(\mathbf{X})] = E[g(\mathbf{X}) - E[g(\mathbf{X})]]^2 = \int_{\mathbf{x} \in D} (g(\mathbf{x}) - E[g(\mathbf{X})])^2 p(\mathbf{x}) dA_{\mathbf{x}}$$

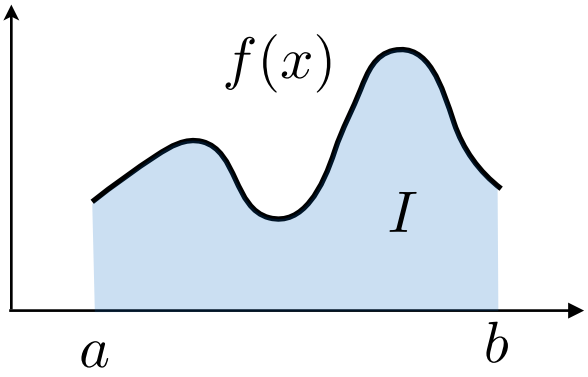
- Estimated expected value

$$E[g(\mathbf{X})] \approx \frac{1}{N} \sum\nolimits_i g(\mathbf{x}_i)$$

Numerical integration

- Rendering amounts to solving numerical integrals
- Let us consider first the case of one dimensional integrals
  - in this case, the integral is the area below the curve

$$I = \int_a^b f(x)dx$$



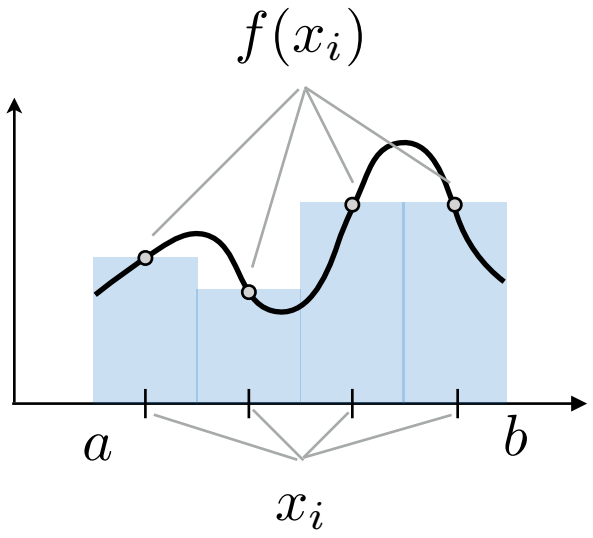
Numerical quadrature

- The classic method for solving numerical integrals is to sample the function at *n* equal spaces positions and approximate the integral with the corresponding boxes

$$I = \int_a^b f(x)dx$$

$$x_i = a + (i - 0.5) \cdot \frac{b-a}{N}$$

$$I \approx \sum_{i=1}^{N} f(x_i) \cdot \frac{b-a}{N}$$



Monte Carlo integration

we want to evaluate $$I = \int_a^b f(x)dx$$

by definition $$E[g(X)] = \int_a^b g(x)p(x)dx$$

can be estimated as $$E[g(X)] \approx \frac{1}{n}\sum_{i=0}^{n-1} g(x_i)$$

by setting $$g(x) = \frac{f(x)}{p(x)} \text{ with uniform } p(x) = \frac{1}{b-a}$$

we have that $$I = \int_a^b \frac{f(x)}{p(x)}p(x)dx$$

can be evaluated as $$I \approx \frac{1}{n}\sum_{i}^{n} \frac{f(x_i)}{p(x_i)}$$

Monte Carlo integration

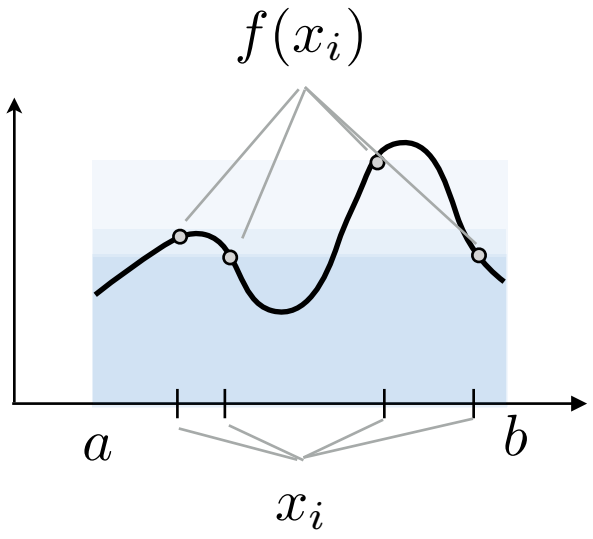
- Approximately evaluate the integral using a *uniform random variable*, by averaging function values at random positions in [a,b)

$$I = \int_a^b f(x)dx$$

$$x \sim p(x) = \frac{1}{b-a}$$

$$I \approx \frac{1}{N} \sum_i \frac{f(x_i)}{p(x_i)} =$$

$$= \frac{1}{N} \sum_i f(x_i)(b-a)$$



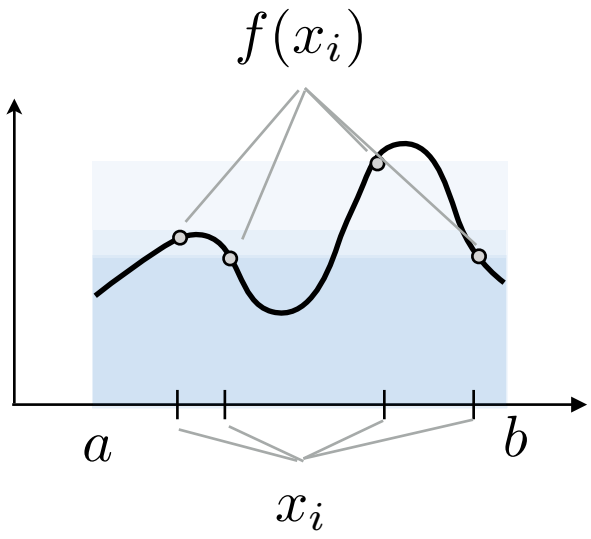
Monte Carlo integration

- Approximately evaluate the integral using a *uniform random variable*, by averaging function values at random positions in [a,b)

$$I = \int_a^b f(x)dx$$

$$x \sim p(x) = \frac{1}{b-a}$$

$$I \approx \frac{1}{N} \sum_i f(x_i)(b-a)$$



$$I \approx \frac{1}{N} \sum$$

Monte Carlo vs. quadrature

- Evaluate the same integral

$$I = \int_a^b f(x)dx$$

- with stochastic vs deterministic positions

$$x \sim p(x) = \frac{1}{b-a} \qquad\qquad x_i = a + (i + 0.5) \cdot \frac{b-a}{n}$$

- using averaging vs reconstruction

$$I \approx \frac{1}{N} \sum_i f(x_i)(b-a) \qquad\qquad I \approx \sum_{i=1}^{N} f(x_i) \cdot \frac{b-a}{N}$$

Curse of dimensionality

- Evaluate an integral of a function of *k* variables, here in a "box" domain

$$I = \int_{\mathbf{x} \in D} f(\mathbf{x}) dA_{\mathbf{x}} = \int_{a_1}^{b_1} \int_{a_2}^{b_2} ... \int_{a_k}^{b_k} f(x_1, x_2, ..., x_k) dx_1 dx_2 ... dx_k$$

- With numerical quadrature we split each axis separately
- The integral is then approximated by summing the function values of the n^k *k*-dimensional intervals

$$I \approx \sum_{i_1}^{N_1} \sum_{i_2}^{N_2} ... \sum_{i_k}^{N_k} f(x_{1,i_1}, x_{2,i_2}, ..., x_{k,i_k}) w_1 w_2 ... w_k$$

- Error in the estimate goes down as

$$error \propto \frac{1}{\sqrt[k]{N}}$$

Monte Carlo integration

- The estimation of the expected values works in *any dimension*
  - just pick random numbers in the proper high dimensional domain

$$I = \int_{\mathbf{x} \in D} f(\mathbf{x}) dA_{\mathbf{x}} \approx \frac{1}{N} \sum_{i} \frac{f(\mathbf{x})}{p(\mathbf{x})}$$

- Works for any shape of the domain, provided that you can picks random numbers in it
- The expected relative error of the random estimate depends only on the number of samples, and not the dimensionality of the domain, and goes down with the square root of *n*

$$error \propto \frac{1}{\sqrt{N}}$$

Example: computing *pi*

- *Pi* can be evaluated by computing the integral of a function that is one inside a unit circle and 0 outside

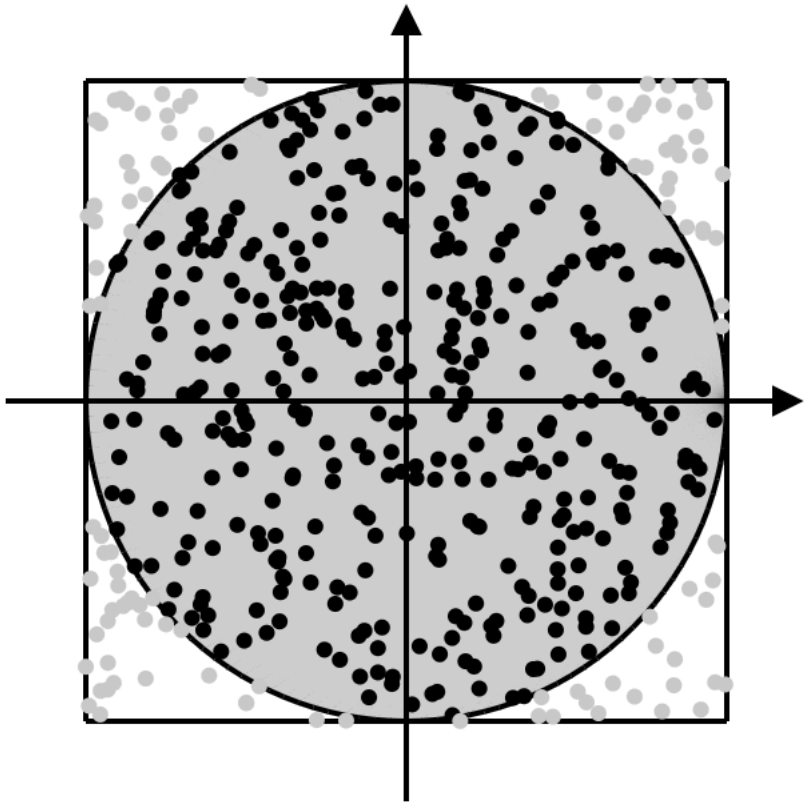
$$\pi = \int_{-1}^{1}\int_{-1}^{1} f(x,y)dxdy \text{ with } f(x,y) = \begin{cases} 1 & \text{if } x^2+y^2 \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

- With Monte Carlo integration, we can estimate *pi* as

$$\pi \approx \frac{4}{N}\sum_{i}^{N} f(x_i,y_i) \text{ with } (x_i,y_i) \in [-1,1]^2, p(x_i,y_i) = \frac{1}{4}$$

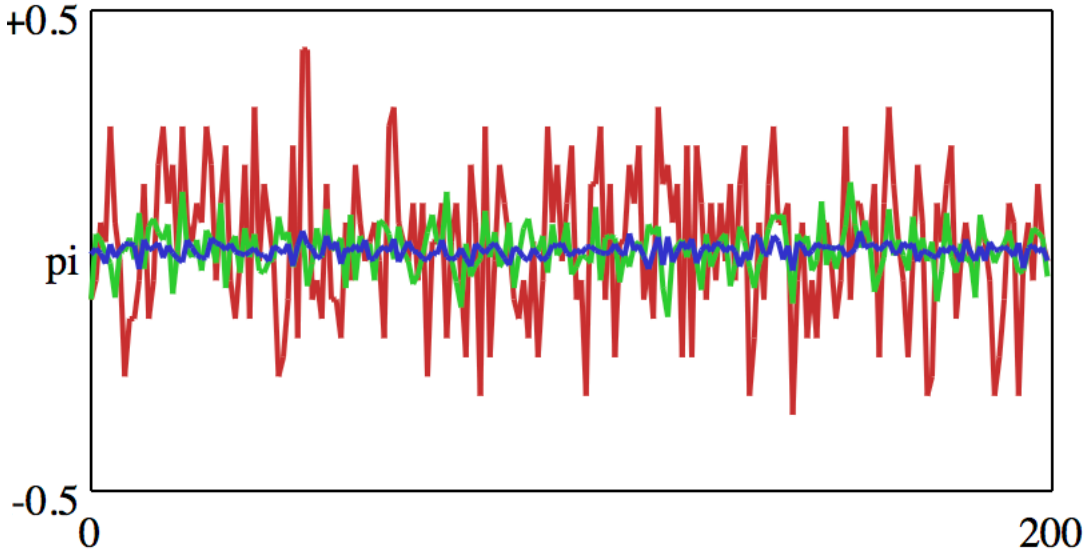
Example: computing *pi*

- If we run one experiment we obtain the following image



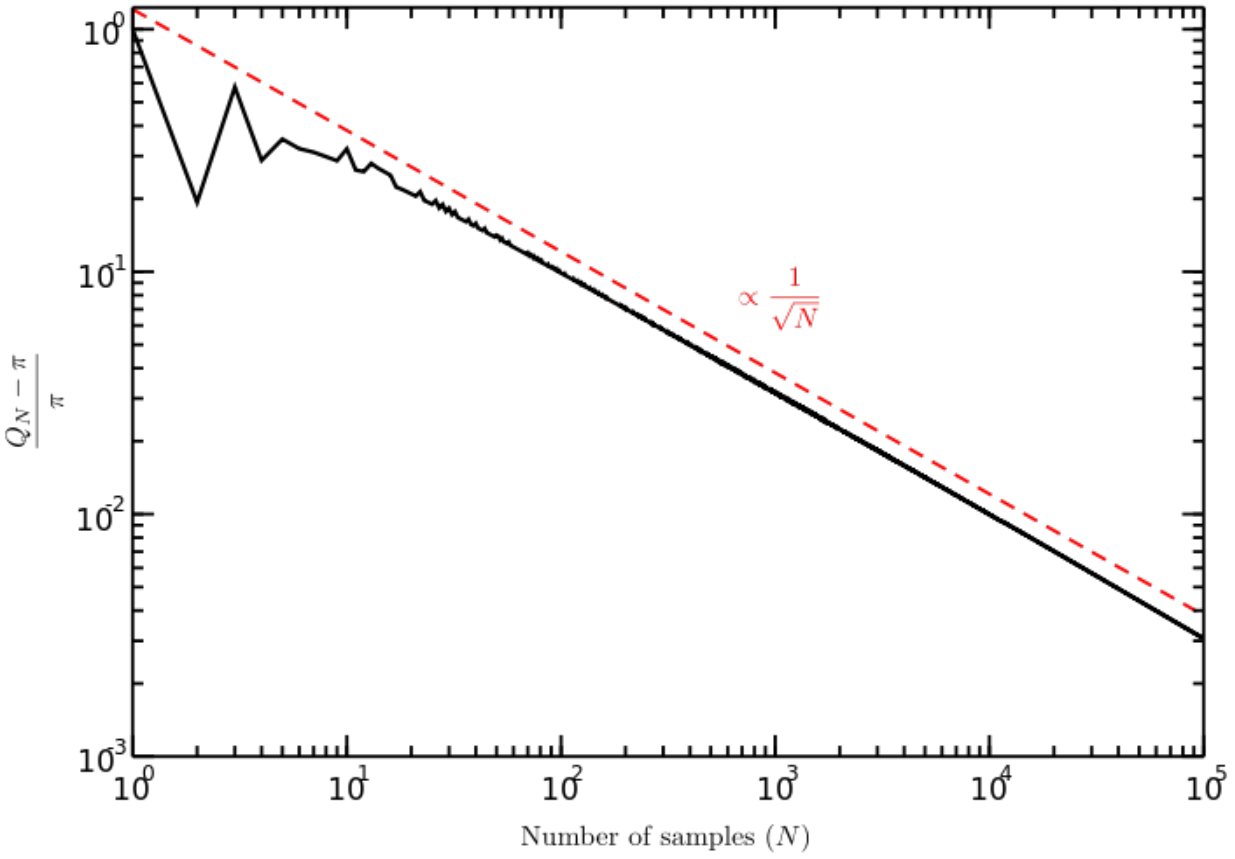
Example: computing *pi*

- Each time we run an experiment we get a slightly different value of *pi*
- The estimates get better as *N* increases



Example: computing *pi*

- Variance decreases with the square root of *N*



[Wikipedia]

Generating random numbers

- Monte Carlo methods requires random numbers
- In general, we use *pseudo random numbers*, i.e. numbers that are deterministically generated, but appear random
  - true random numbers are not always available and are expensive
- Many generators exists with different tradeoffs between random quality and speed/memory
- All generators produce sequences pseudo-random bits, that can be used to generate floating point values in the [0,1) range
- I particularly like the PCG family of random numbers by Melissa E. O'Neill, since they are simple, have good statistical properties, are efficient and they allow for multiple independent streams of random numbers

Implementing the *pi* estimate

- Now we have a way to generate uniform random floats in [0,1)
- But we need uniform values in [-1,1]x[-1,1]
- To generate random numbers in higher dimension, we simply generate multiple random numbers with our generators since the numbers are statistically independent
  - the pdf is the product of the one-dimensional pdfs
- To generate random numbers in a different domain, we *warp* the random numbers to that domain; in out case this is just scaling
  - the pdf needs to be appropriately "rescaled" to take into account the new domain size and the change of variable
- If *r* is our generator of uniform random numbers in [0,1) we can write

$$r \sim p(r) = 1 \rightarrow \mathbf{x} = (2r_1 - 1, 2r_2 - 1) \sim p(\mathbf{x}) = 1/4$$

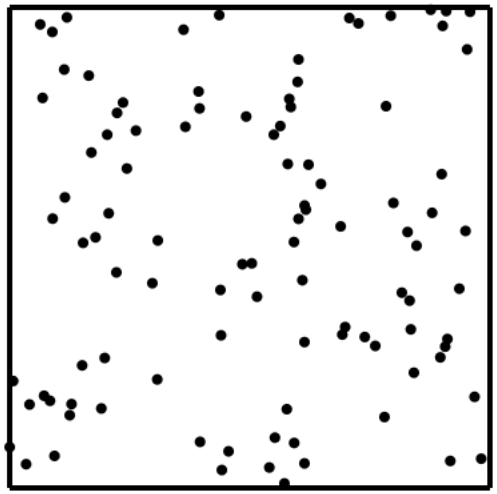
Implementing the *pi* estimate

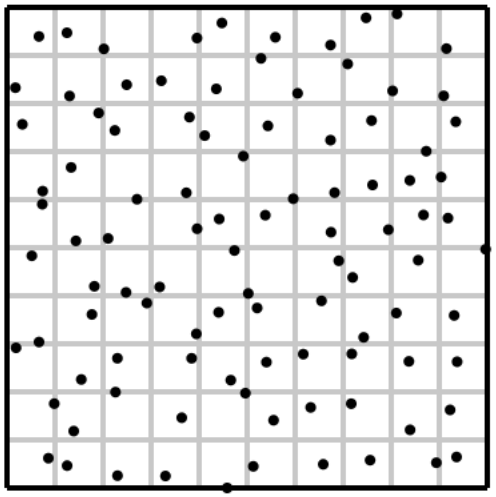
```
float f(vec2f x) {
    return (x.x * x.x + x.y * x.y < 1) ? 1 : 0;
}

float pi(int nsamples, rng_state& rng) {
    auto integral = 0.0f;
    for (auto s = 0; s < nsamples; s++) {
       integral += f({randf(rng)*2-1,randf(rng)*2-1});
    }
    integral *= 4.0f / nsamples;
    return integral;
}
```


Stratified sampling

- Main issue with Monte Carlo is the noise in the solution of integrals
- Several methods exists to reduce noise, which amount to choosing samples at better locations, but still randomly
- *Stratified sampling:* avoid "clumps" in uniform random numbers by forcing them to be distributed in smaller domains
  - not so convenient to implement in rendering

$$\mathbf{x} = \left(\frac{i + r_1}{N_1}, \frac{j + r_2}{N_2} \right)$$





Importance sampling

- *Importance sampling:* sample with a pdf that is as close as possible to the function we want to integrate
  - intuition: the closer the pdf is to the integrand, the smaller in the ratios of f(x)/p(x)
  - most important variance reduction in rendering

$$\mathbf{x} \sim p(\mathbf{x}) \approx f(\mathbf{x})$$

- We will introduce later appropriate warp functions to sample according the advantageous pdfs

Monte Carlo integration

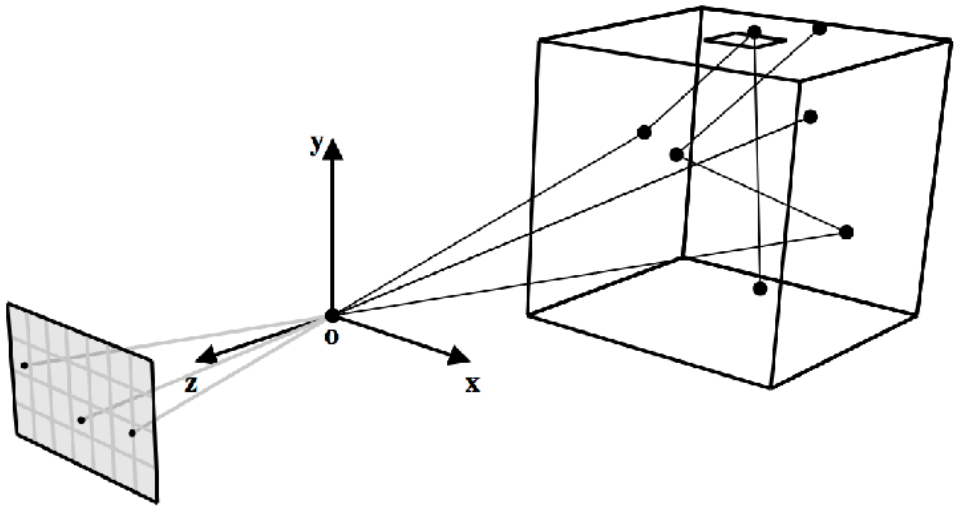
- Pros: works for every domain
- Pros: convergence independent of dimensionality
- Pros: works for functions with discontinuities
- Pros: simple implementation

- Cons: relatively slow convergence
- Cons: variance (noise) in the estimate

Naive Path Tracing

Path tracing

- Solves the rendering equation with Monte Carlo integration supporting all lighting effects shown above
- For each pixel independently, we solve the rendering equation by integrating the incoming radiance over several paths that go through that pixel – paths are traced backwards starting at the pixel and going towards the scene



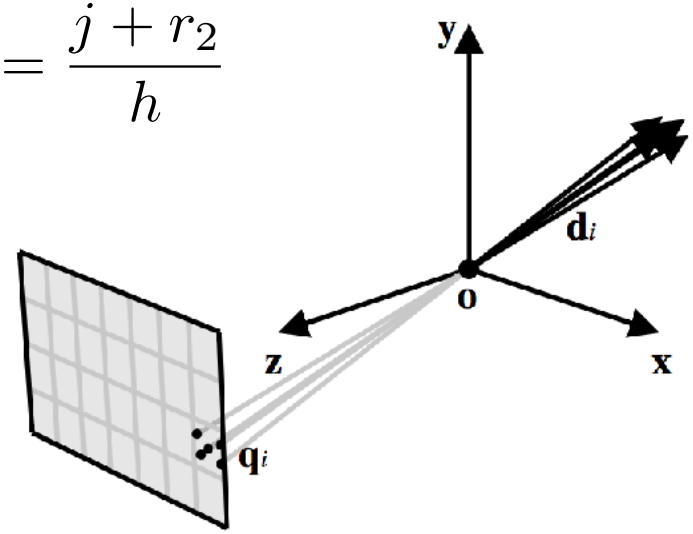
Pixel sampling

- We compute the pixel integral by straightforward Monte Carlo integration of the incoming radiance

$$L_{pixel} = \frac{1}{A_{pixel}} \int_{\mathbf{q} \in pixel} L_i(\mathbf{q}, \overrightarrow{\mathbf{eq}}) dA_{\mathbf{q}} \approx \frac{1}{N} \sum_{i}^{N} L_i(\mathbf{q}_i, \overrightarrow{\mathbf{eq}_i})$$

- To generate the random positions $\mathbf{q}_i$ we pick two random numbers in the unit interval and warp them to the camera *(u,v)* coordinates

$$u = \frac{i + r_1}{w} \qquad v = \frac{j + r_2}{h}$$



Estimating incoming radiance

- We estimate the incoming radiance by tracing a ray toward the scene
- If the ray hits, we return an estimate of the outgoing radiance computed at the intersection point and with the intersection BRDF
- If not, we return the background radiance at that direction

$$L_i(\mathbf{q}, \mathbf{d}) = \begin{cases} L(\mathbf{r}(\mathbf{q}, \mathbf{d}), -\mathbf{d}) & \text{if ray hits} \\ L_{environment}(\mathbf{d}) & \text{otherwise} \end{cases}$$

Solving the reflection equation

- We first give an "intuitive" solution to the rendering equation that we will then formally prove later
- Consider the rendering equation

$$L(\mathbf{x}, \mathbf{o}) = L_e(\mathbf{x}, \mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x}, \mathbf{i}), -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}| d\omega_{\mathbf{i}}$$

- We can solve this equation by picking random direction in the sphere above the point – we use the hemisphere to handle opaque objects
- A Monte Carlo estimate for integral can be written as

$$L(\mathbf{x}, \mathbf{o}) \approx L_e(\mathbf{x}, \mathbf{o}) + \frac{L(\mathbf{r}(\mathbf{x}, \mathbf{i}_i), -\mathbf{i}_i) f(\mathbf{x}, \mathbf{i}_i, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i)}$$

Hemispherical sampling

- To generate random directions in the hemisphere, we warp two uniform random numbers to an hemispherical direction
- To lower variance, we choose a distribution proportional to the cosine
- In local coordinates, we can write

$$\mathbf{i}^l = [\sqrt{1-r_2}\cos(2\pi r_1), \sqrt{1-r_2}\sin(2\pi r_1), r_2]$$

$$p(\mathbf{i}^l) = \frac{\mathbf{i}^l_z}{\pi}$$

- To get the world coordinates, we construct a frame with z aligned to the normal and transform by it

$$\mathbf{i} = \sqrt{1-r_2}\cos(2\pi r_1)\cdot\mathbf{t_x} + \sqrt{1-r_2}\sin(2\pi r_1)\cdot\mathbf{t_y} + r_2\cdot\mathbf{n}$$

$$p(\mathbf{i}) = \frac{\mathbf{n}\cdot\mathbf{i}}{\pi}$$

Recursive evaluation

- The equation above cannot be directly evaluated since the unknown radiance is present on the right side
- To solve the equation, we recursively evaluate the integral using the same formulation
- The recursion terminates is we do not hit a surface, in which case we accumulate the radiance of the environment map
- But in most cases, say closed rooms, an infinite recursion will happen
  - for now, we stop evaluation after a fixed number of recursive calls
- We start the recursion at each camera pixel

Path tracing – main loop

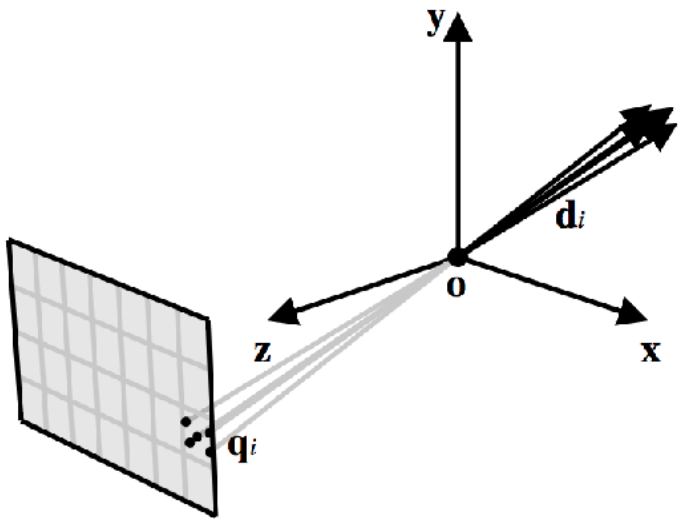
- We can now sketch the pseudocode for a naive path tracer starting from the outer loop

```
auto rngs = make_rngs();
auto acc = image{img.size, zero3f};
for(auto s : range(ns)) {
  for(auto j : range(img.height)) {
    for(auto i : range(img.width)) {
      auto puv = rand2f(rngs[i,j]);
      auto uv = (vec2f{i,j} + puv) / img.size;
      auto ray = eval_camera(cam, uv);
      acc[i,j] += shade(scene, ray, rngs[i,j]);
      img[i,j] = acc[i,j] / (s + 1);
    }
  }
  // view or save img if desired
}
```


Path tracing – pixel sampling

- We then sample the evaluate the camera as before, ignoring the lens for now

```
ray3f eval_camera(camera* camera, vec2f image_uv) {
  auto q = vec3f{camera->film.x * (0.5 - image_uv.x),
    camera->film.y * (image_uv.y - 0.5), camera->lens};
  auto e = vec3f{0}; auto d = normalize(-q - e);
  return ray3f{transform_point(camera->frame, e),
      transform_direction(camera->frame, d)};
}
```



Path tracing – environment maps

- If intersections fail, we can get the environment color by looking values in the environment texture using a lat-lon parametrization

$$L_{env}(\mathbf{i}^l) = k_e * txt \left[\frac{\text{atan2}(\mathbf{i}^l_z, \mathbf{i}^l_x)}{2\pi}, \frac{\text{acos}(\mathbf{i}^l_y)}{\pi} \right]$$

Path tracing – environment maps

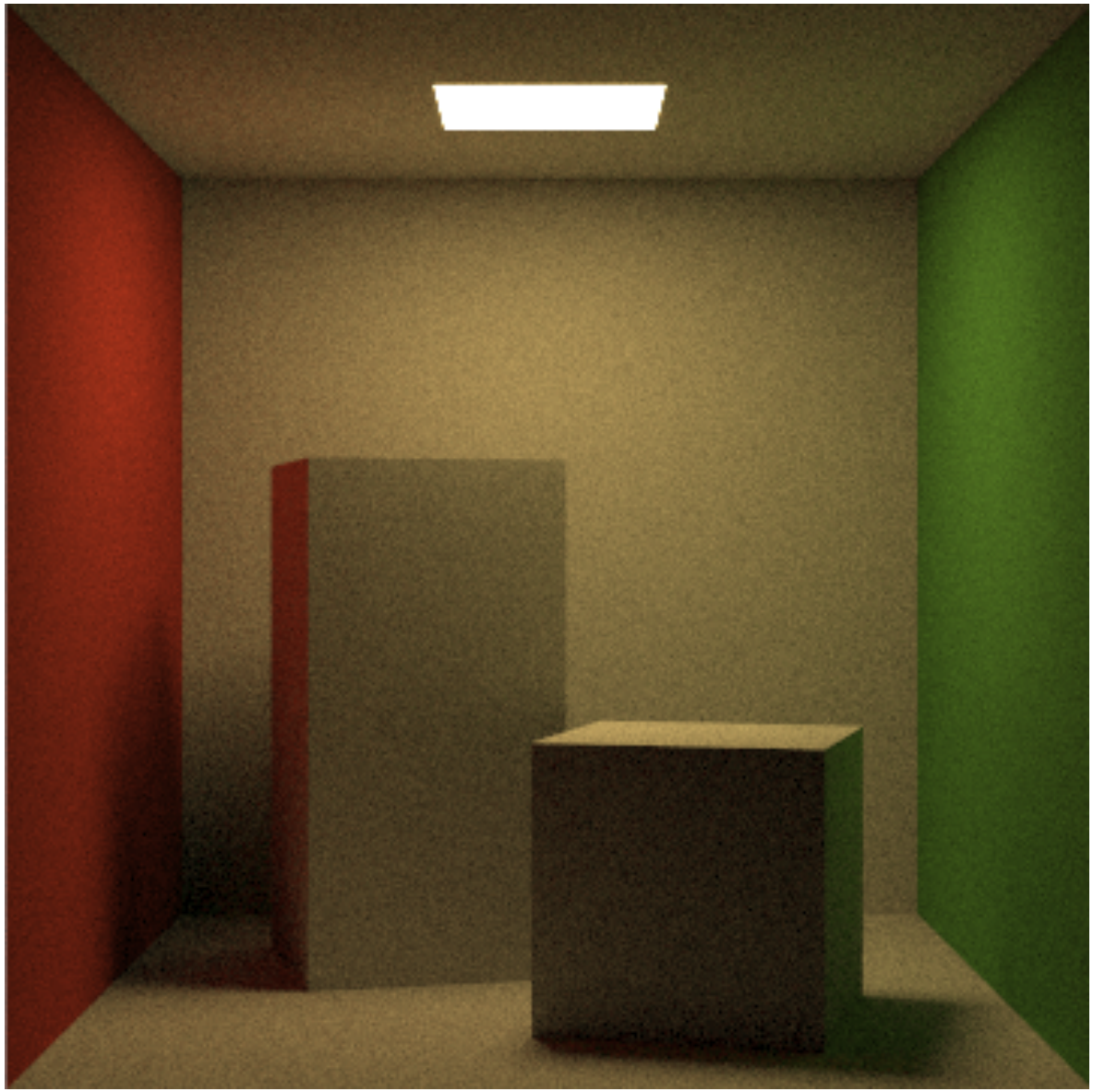
- If intersections fail, we can get the environment color by looking values in the environment texture using a lat-lon parametrization

```
vec3f eval_environment(environment* environment, vec3f dir) {
  auto local_dir = transform_direction(
                       inverse(environment->frame), dir)
  auto texcoord = vec2f{
        atan2(local_dir.z, local_dir.x) / (2 * pi),
        acos(clamp(local_dir.y, -1, 1)) / pi};
  if (texcoord.x < 0) texcoord.x += 1;
  return environment->emission *
            eval_texture(environment->emission_tex, texcoord);
}
```

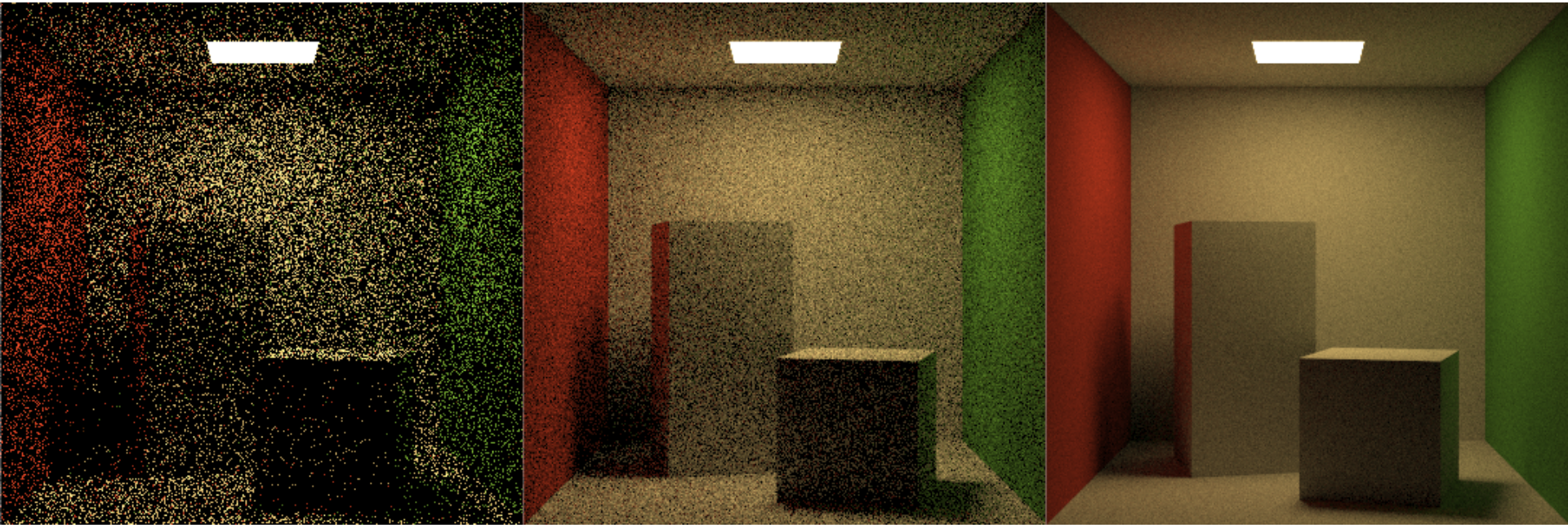

Path tracing – recursive

- We can now write the path tracer with the previous equation

```
vec3f shade_recursive(scene* scn, ray3f ray, int bounce) {
  auto isec = intersect(scn, ray);
  if(!isec.hit) return eval_environment(scn, ray.d);

  auto [p, n] = eval_point(isec);    // eval pos, norm
  auto [e, f] = eval_material(isec); // eval bsdf
  auto o = -ray.d;

  auto l = eval_emission(e, n, o);   // for now, return le
  if(bounce >= max_bounce) return l; // exit

  auto i = sample_hemisphere_cos(n, rand2f(rng));
  l += shade_recursive(scn, {p, i, bounce+1}) *
       eval_brdf(f,n,i,o) * abs(dot(i,n)) /
       sample_hemisphere_cos_pdf(n,i);
  return l;
}
```

Naive path tracing – convergence

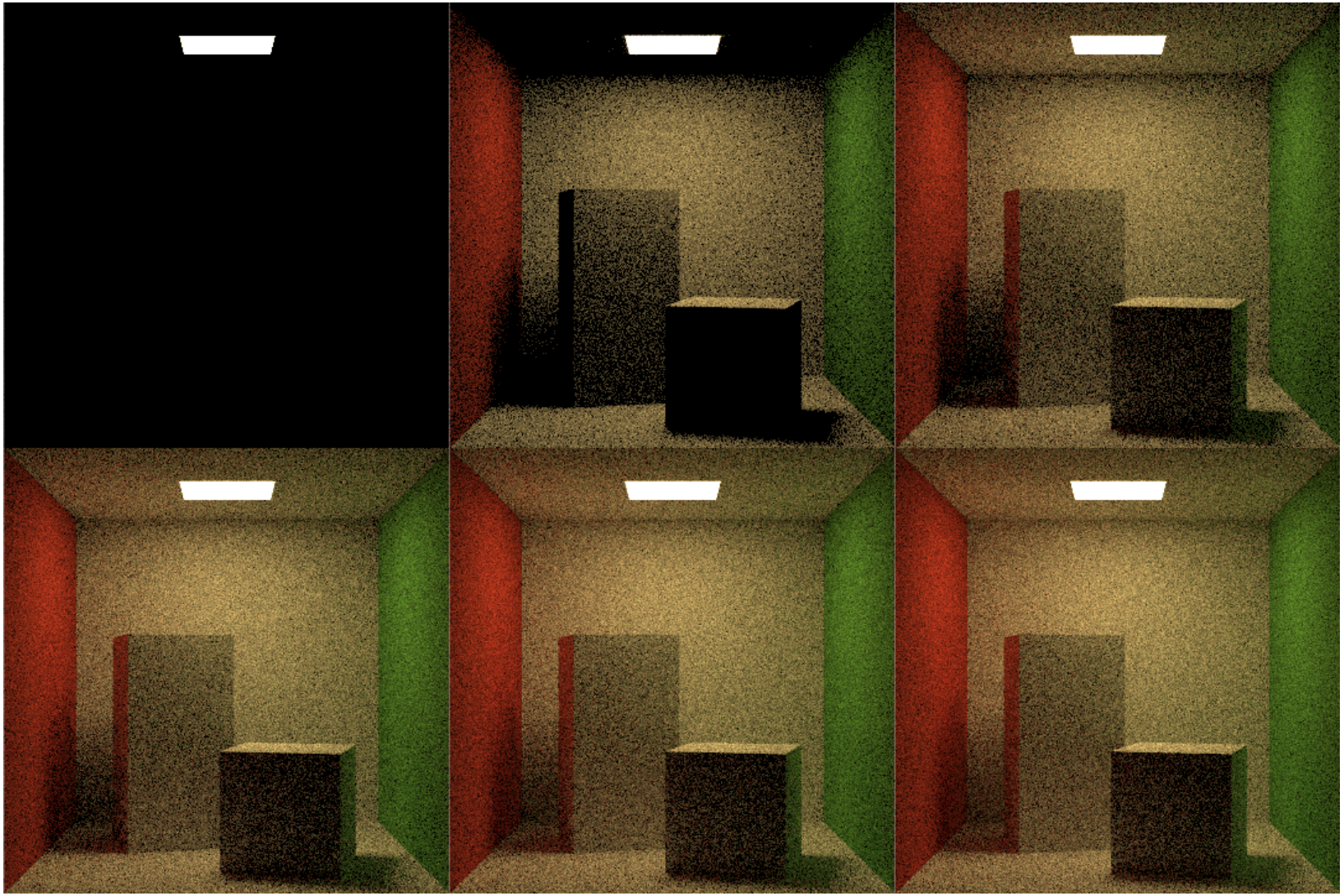


16 samples

256 samples

4096 samples

Naive path tracing – bounces



Russian roulette

- We want to avoid stopping at the arbitrary bounce since the energy associate with it depends on the scene configuration
- *Russian roulette*: stop with a given probability and scale the value of the remaining samples to the inverse of that probability to account for the energy loss due to stopping

$$g(\mathbf{x}) \text{ is an estimator for I}$$

$$g_r(\mathbf{x}) = \begin{cases} \frac{1}{p_r} g(\mathbf{x}) & \text{with probability } p_r \\ 0 & \text{with probability } 1 - p_r \end{cases}$$

$$E[g_r(\mathbf{x})] = p_r \cdot E[(1/p_r) g(\mathbf{x})] + (1 - p_r) \cdot 0 = E[g(\mathbf{x})]$$

Russian roulette

- In the context of Path tracing, we check whether a path needs to be terminate and, if not, correct the recursive radiance
- For the probability of termination, we could use the surface reflectivity since that would match the real world behavior
- But computing that might not be easy – alternatively, we compute the survival probability as

$$p_r = \min\left(1, \frac{f(\mathbf{i}, \mathbf{o})|\mathbf{i} \cdot \mathbf{n}|}{p(\mathbf{i})}\right)$$

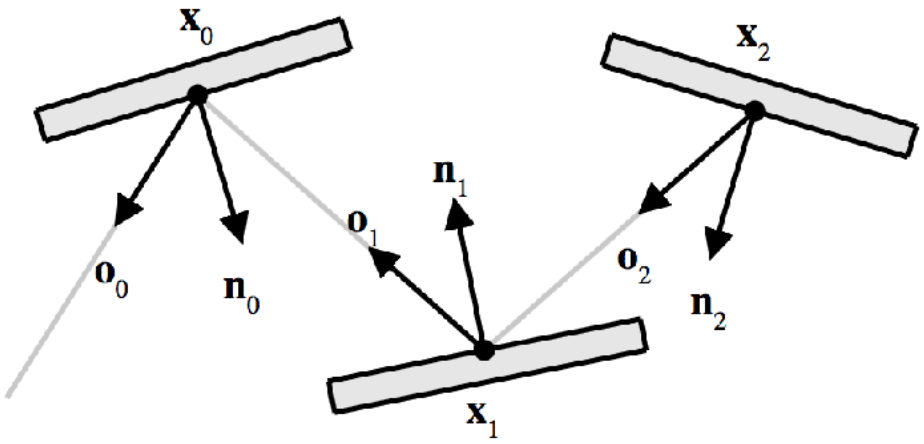
Path tracing – Russian roulette

```
vec3f shade_russian_roulette(scene* scene, ray3f ray) {
  auto isec = intersect(scene, ray);
  if(!isec.hit) return eval_environment(scene, ray.d);

  auto [p, n]  = eval_point(isec);   // eval pos, norm
  auto [e, f] = eval_material(isec); // eval bsdf
  auto o = -ray.d;

  auto l = eval_emission(e, n, o);   // for now, return le
  if(bounce >= max_bounce) return l; // exit
  auto rr = min(1,max(f));           // continuation probability
  if(rand1f() >= rr) return l;       // russian roulette

  auto i = sample_hemisphere_cos(n, rand2f(rng));
  l += shade_recursive(scn, {x, i}) * eval_brdf(f,n,i,o) *
       abs(dot(i,n)) / (rr * sample_hemisphere_cos_pdf(n,i));
  return l;
}
```


Path tracing – product form

- One problem of the recursive evaluation is that it is harder to improve since the recursion "hides" the overall path behavior
- An alternative way to write a path tracer is to unroll the recursion
- Let us write the Monte Carlo estimator for the *j*-th point along a path

$$L(\mathbf{x}_j, \mathbf{o}_j) = L_e(\mathbf{x}_j, \mathbf{o}_j) + \left[f(\mathbf{x}_j, \mathbf{i}_j, \mathbf{o}_j)(\mathbf{n}(\mathbf{x}_j) \cdot \mathbf{i}_j)/p(\mathbf{i}_j) \right] L(\mathbf{x}_{j+1}, \mathbf{o}_{j+1}) =$$
$$= L_e(\mathbf{x}_j, \mathbf{o}_j) + w_j L(\mathbf{x}_{j+1}, \mathbf{o}_{j+1}) \quad \text{with } \mathbf{i}_j = -\mathbf{o}_{j+1}$$

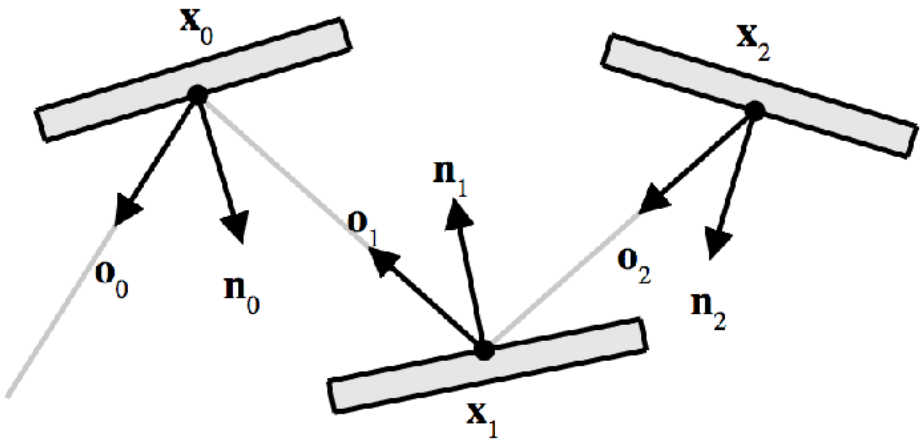


Path tracing – product form

- To compute the radiance of the *whole path* we unroll the previous recursion as

$$L_{path} = L_e(\mathbf{x}_0, \mathbf{o}_0) + w_1 L_e(\mathbf{x}_1, \mathbf{o}_1) + w_1 w_2 L_e(\mathbf{x}_2, \mathbf{o}_2) + \ldots =$$

$$= \sum_j W_j L_e(\mathbf{x}_j, \mathbf{o}_j) \quad \text{with } W_j = w_j W_{j-1} = \prod_{k=0}^{j} w_k$$



Path tracing - product form

```
vec3f shade_product(scene* scene, ray3f ray) {
  auto l = vec3f{0}, w = vec3f{1};
  for(auto bounce : range(max_bounce)) {
    auto isec = intersect(scene, ray);
    if(!isec.hit) {
      l += w * eval_environment(scene, ray.d); break; }
    auto [p, n] = eval_point(isec);    // eval pos, norm
    auto [e, f] = eval_material(isec); // eval bsdf
    auto o = -ray.d;                   // outgoing
    l += w * eval_emission(e, n, o);   // emission
    auto i = sample_hemisphere_cos(n, rand2f(rng)); // incoming
    w *= eval_brdf(f,n,i,o)  * abs(dot(i,n)) / // update weight
         sample_hemisphere_cos_pdf(n,i);
   if(rand1f() >= min(1,max(w))) break;  // russian roulette
    w *= 1 / min(1,max(w));              // update weight
    ray = {p, i};                        // recurse
 }
  return l;
}
```


Path tracing - handle materials

- To handle sharp materials, we can add a special case to the rendering loop that samples their directions directly
- As a convenience, we can also fold the cosine inside the brdf functions
- Finally, we will later pick directions based on the full BRDF, not just the cosine


```
vec3f shade_naive(scene* scene, ray3f ray) {
  auto l = zero3f, w = vec3f{1};
  for(auto bounce : range(max_bounce)) {
    auto isec = intersect(scene, ray);
    if(!isec.hit) {
      l += w * eval_environment(scene, ray.d); break; }
    auto [p, n]  = eval_point(isec);    // eval pos, norm
    auto [e, f] = eval_material(isec);  // eval bsdf
    auto o = -ray.d;                    // outgoing
    l += w * eval_emission(e, n, o);    // emission
    if(!is_delta(f)) { // sample smooth brdfs (fold cos into f)
      auto i = sample_brdfcos(f,n,o,rand2f(rng)); // incoming
      w *= eval_brdfcos(f,n,i,o) / sample_brdfcos_pdf(f,n,i,o);
    } else {          // sample sharp brdfs
      auto i = sample_delta(f,n,o,rand2f(rng));   // incoming
      w *= eval_delta(f,n,i,o) / sample_delta_pdf(n,i,o);
    }
    if(rand1f() >= min(1,max(w))) break;    // russian roulette
    w *= 1 / min(1,max(w));
    ray = {x, i};                           // recurse
 }
  return l;
}
```


Path tracing – product form

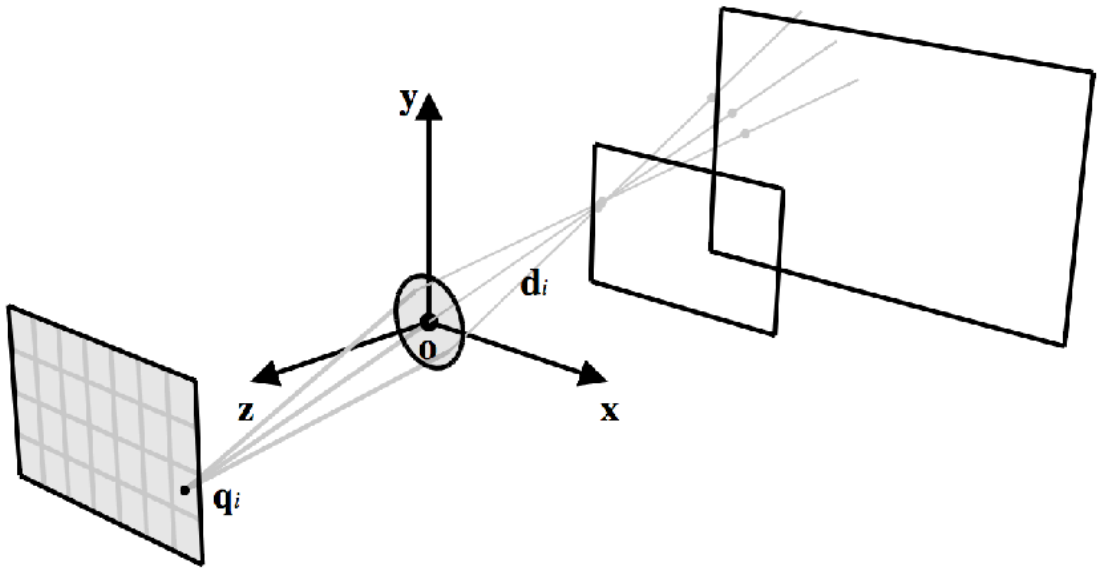
```
vec3f shade_naive(scene* scene, ray3f ray) {
 <init>
 for(auto bounce : range(max_bounce)) {
    <intersection and environment>
    <eval point and material>
    <emission>
    if(!is_delta(f)) <handle smooth>
    else <handle delta>
    <russian roulette>
    <recurse>
  }
  return l;
}
```


Realistic camera

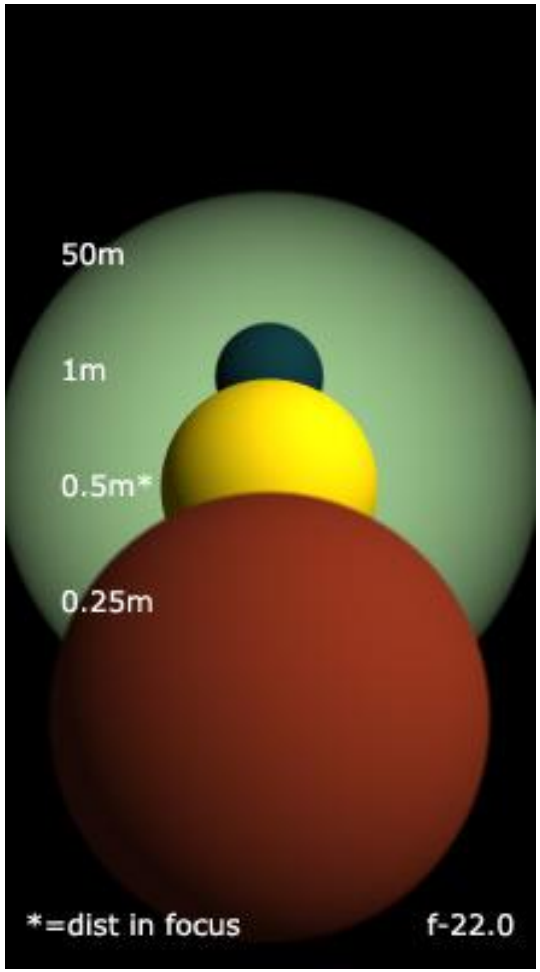
Pixel radiance

- We account for lenses with finite aperture by also integrating for all possible positions on the lens itself; this give us defocus blur

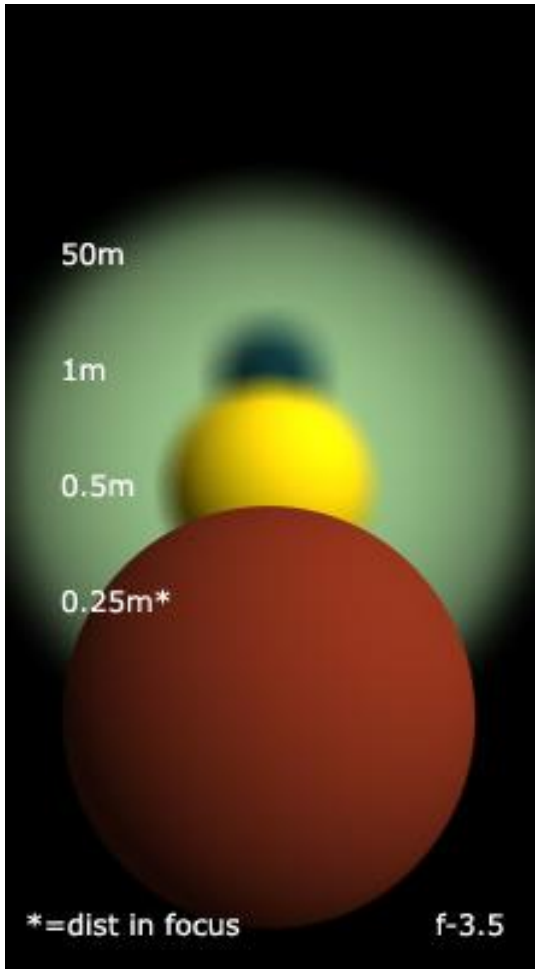
$$L_{pixel} = \frac{1}{A_{pixel}} \frac{1}{A_{lens}} \int_{\mathbf{q} \in pixel} \int_{\mathbf{e} \in lens} L_i(\mathbf{q}, \mathbf{d}) dA_{\mathbf{q}} dA_{\mathbf{e}} \text{ with } \mathbf{d} = \frac{\mathbf{e} - \mathbf{q}}{|\mathbf{e} - \mathbf{q}|}$$



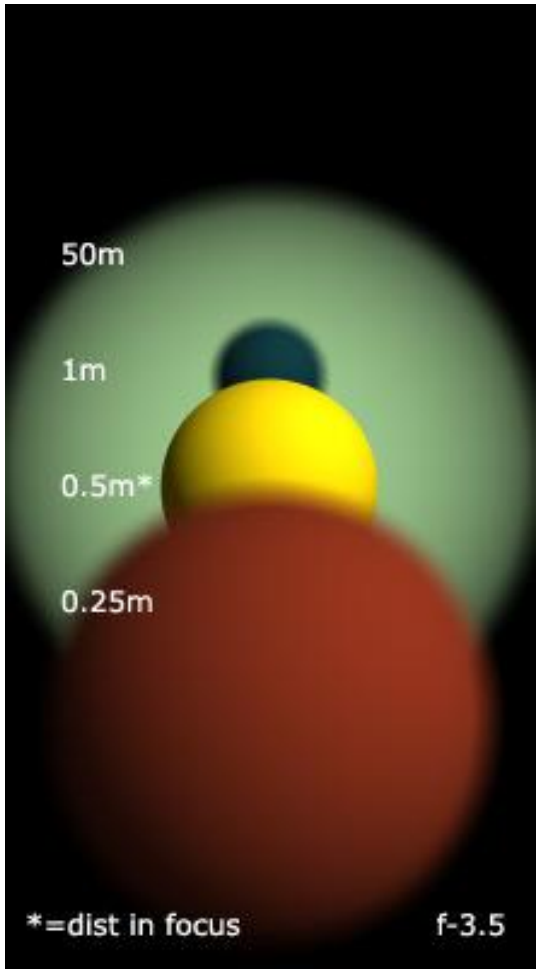
Pixel radiance



pinhole



close focus



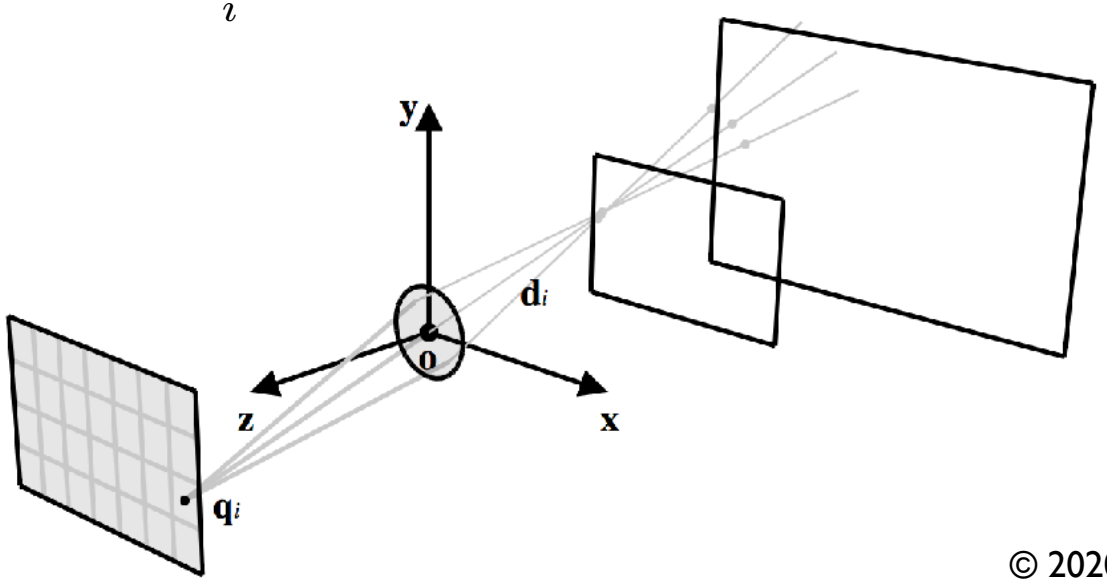
far focus

Sampling the lens

- Monte Carlo integration works just fine when considering the lens too
  - note how we numerically compute two integrals at once

$$L_{pixel} = \frac{1}{A_{pixel}} \frac{1}{A_{lens}} \int_{\mathbf{q} \in pixel} \int_{\mathbf{e} \in lens} L_i(\mathbf{q}, \overrightarrow{\mathbf{eq}}) dA_{\mathbf{q}} dA_{\mathbf{e}}$$

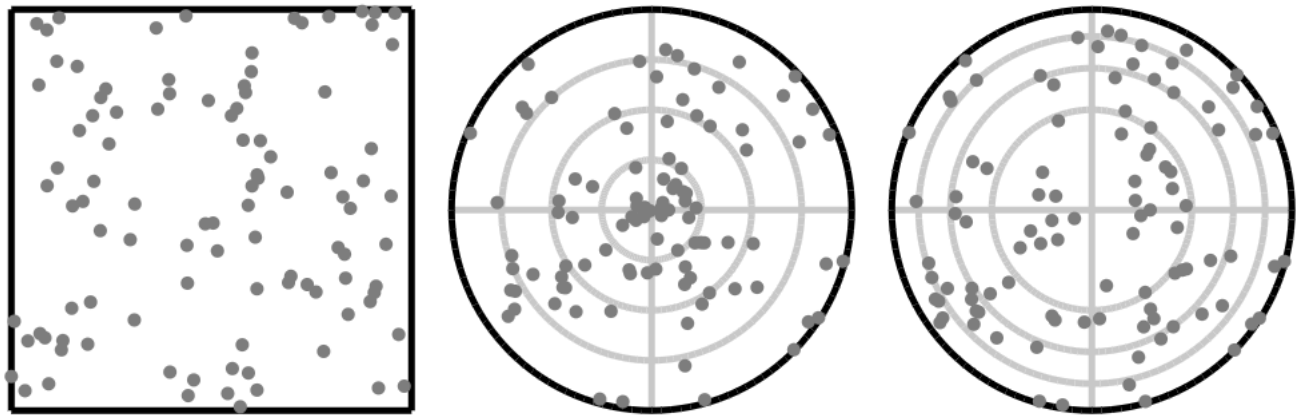
$$\approx \frac{1}{N} \sum_{i}^{N} L_i(\mathbf{q}_i, \overrightarrow{\mathbf{e}_i \mathbf{q}_i})$$



Sampling the lens

- For square lens, we could generate the lens positions by warping to a small square like we did not far – but lenses are really circular
- If we warp with polar coordinates we get points on the circles that are not equally distributed since the areas change
- To sample the circle, we *warp with an area-preserving method*

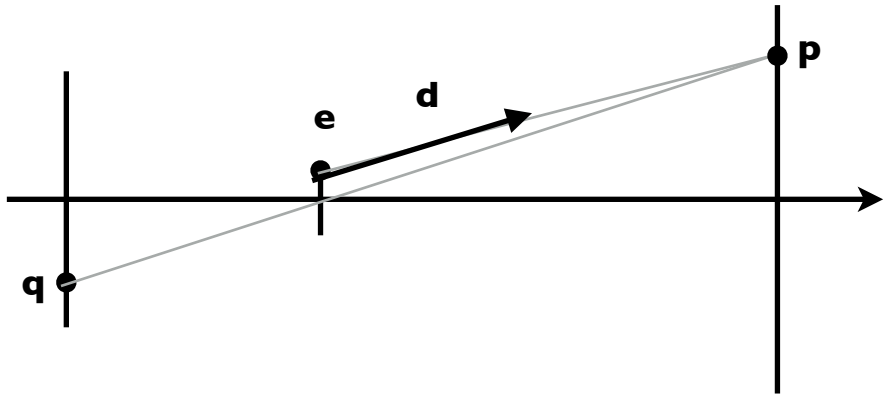
$$\mathbf{e}^l = [ar\cos\theta, ar\sin\theta, 0] \quad \text{with } \theta = 2\pi r_1, r = \sqrt{r_2}$$



Sampling the lens

- To get the ray direction, we assume that we have a thin lens and connect the lens position to the position on the focal plane
  - pick a point on the film, compute direction as through lens center
  - find position on focus plane of the ray
  - pick a point on the lens, compute direction to focus point

$$\mathbf{d}^l = \frac{\overrightarrow{\mathbf{ep}}}{|\overrightarrow{\mathbf{ep}}|} \quad \mathbf{p}^l = \mathbf{c}^l \frac{f}{\mathbf{c}_z^l} \quad \mathbf{c}^l = -\frac{\mathbf{q}^l}{|\mathbf{q}^l|}$$



Sampling the lens

```
ray3f eval_camera(camera* camera,vec2f image_uv,vec2f lens_uv) {
  // point on the image plane
  auto q = vec3f{camera->film.x * (0.5f - image_uv.x),
      camera->film.y * (image_uv.y - 0.5f), camera->lens};
  // ray direction through the lens center
  auto dc = -normalize(q);
  // point on the lens
  auto e = vec3f{lens_uv.x * camera->aperture / 2,
      lens_uv.y * camera->aperture / 2, 0};
  // point on the focus plane
  auto p = dc * camera->focus / abs(dc.z);
  // correct ray direction to account for camera focusing
  auto d = normalize(p - e);
  // done
  return ray3f{transform_point(camera->frame, e),
      transform_direction(camera->frame, d)};
}
```

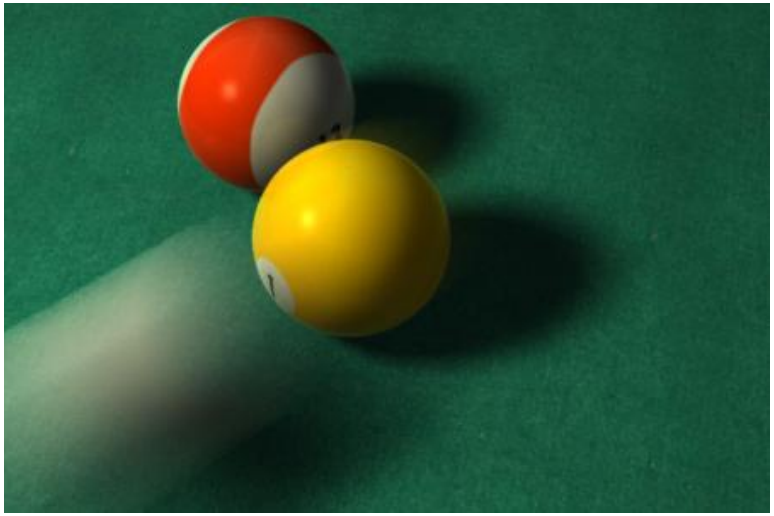

Sampling the lens

```
vec2f sample_disk(vec2f ruv) {
 return {cos(2 * pi * ruv.x) * sqrt(ruv.y),
         sin(2 * pi * ruv.x) * sqrt(ruv.y)};
}

ray3f sample_camera(camera* camera, vec2i ij, vec2i image_size,
                    vec2f puv, vec2f luv) {
  auto uv = vec2f{(ij.x + puv.x) / image_size.x,
                  (ij.y + puv.y) / image_size.y};
  return eval_camera(camera, uv, sample_disk(luv));
}

… main loop …
auto ray = sample_camera(camera, {i,j}, image.size,
                         rand2f(rng), rand2f(rng));
```


Rendering animation

- In general the camera stays open for a finite amount of time while the objects may move in the scene; this gives us motion blur
- We can compute this effect by integrating over time and considering a time-varying scene, say due to animation

$$L_{pixel} = \frac{1}{A_{pixel}} \frac{1}{A_{lens}} \frac{1}{t_1 - t_0} \int_{t_0}^{t_1} \int_{\mathbf{q} \in pixel} \int_{\mathbf{e} \in lens} L_i(\mathbf{q}, \mathbf{d}, t) dA_{\mathbf{q}} dA_{\mathbf{e}} dt$$



Direct Illumination

Splitting direct and indirect

- The main concern of the previous renderer is that all paths that do not touch the light will end up with zero contribution
- Solution: connect each point in the path to at least one light
  - in jargon this is called next event estimation
- To do this, we split the integral of the reflected radiance *direct illumination*, i.e. the reflected radiance originating directly from light sources, and *indirect illumination*, i.e. all the rest
- For indirect illumination, we will proceed as before
- For direct illumination, we want to sample the lights directly to ensure that we hit one

Splitting direct and indirect

- Let us now write the reflection equation by splitting direct and indirect illumination

$$L_r(\mathbf{x},\mathbf{o}) = \int_{H^2} L_i(\mathbf{x},\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_\mathbf{i} =$$
$$= \int_{H^2} L_e(\mathbf{x},\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_\mathbf{i} +$$
$$+ \int_{H^2} L_r(\mathbf{x},\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_\mathbf{i} =$$
$$= L_d(\mathbf{x},\mathbf{o}) + L_{ind}(\mathbf{x},\mathbf{o})$$

- Direct illumination can be thought of as the solution of the reflection equation where the incoming illumination is just the emission term

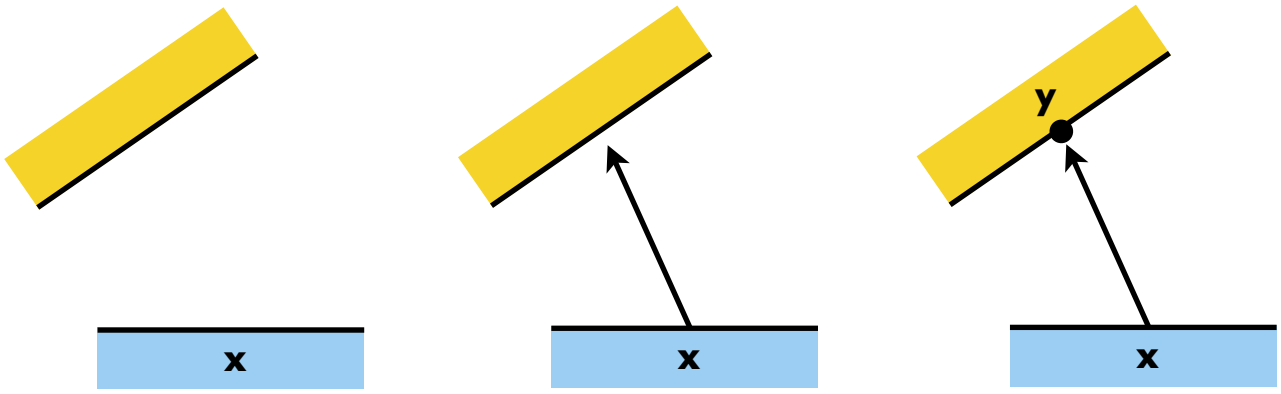
$$L_d(\mathbf{x},\mathbf{o}) = \int_{H^2_\mathbf{x}} L_e(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_\mathbf{i}$$

Direct illumination – hemisphere

- We can solve the direct illumination equation with Monte Carlo by picking random directions and evaluating the integrand function

$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e(\mathbf{r}(\mathbf{x}, \mathbf{i}_i), -\mathbf{i}_i) f(\mathbf{x}, \mathbf{i}_i, \mathbf{o}) |\mathbf{n}_\mathbf{x} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i)}$$

- The problem here is becomes obvious – each ray with L_e zero has no contribution

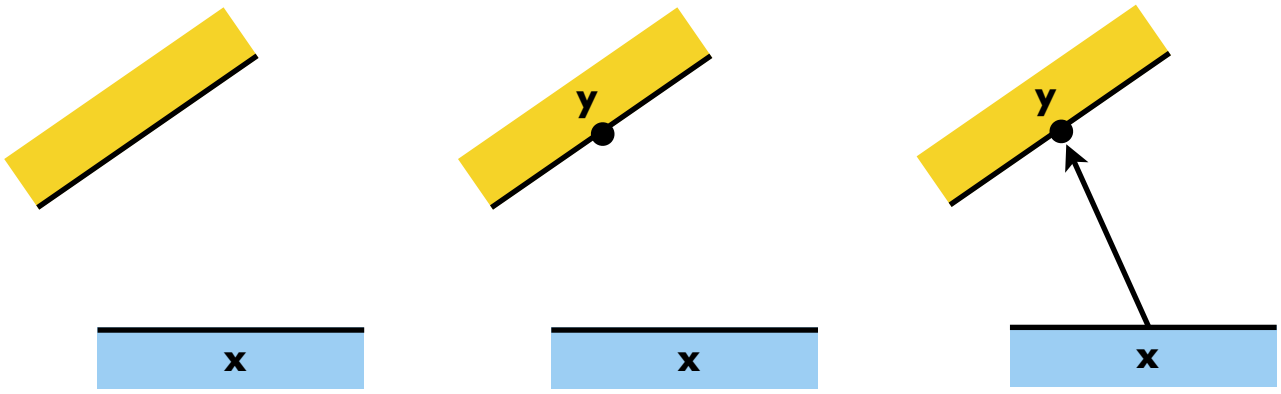


Direct illumination – light

- A better alternative is to pick a point on the light and compute the direction from that;

$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e(\mathbf{r}(\mathbf{x}, \mathbf{i}_i), -\mathbf{i}_i) f(\mathbf{x}, \mathbf{i_i}, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i | \mathbf{y}_i)} \qquad \mathbf{i}_i = \frac{\mathbf{y}_i - \mathbf{x}}{|\mathbf{y}_i - \mathbf{x}|}$$

- We still need to cast a ray to check whether we did hit the light
- We now have to compute the direction PDF from the point PDF



Sampling the light surface

- To evaluate it, we have to pick points uniformly over the light surface
- For a square light of area *A*, we can warp two random numbers as

$$\mathbf{y}^l = \sqrt{A}[r_1 - 0.5, r_2 - 0.5, 0]$$

$$\mathbf{y}^l \sim p(\mathbf{y}^l) = \frac{1}{A}$$

- To move to world coordinates, we use a frame that describes the orientation of the square light

$$\mathbf{y} = \sqrt{A}(r_1 - 0.5) \cdot \mathbf{F_x} + \sqrt{A}(r_2 - 0.5) \cdot \mathbf{F_y} + \mathbf{F_o}$$

$$\mathbf{y} \sim p(\mathbf{y}) = \frac{1}{A}$$

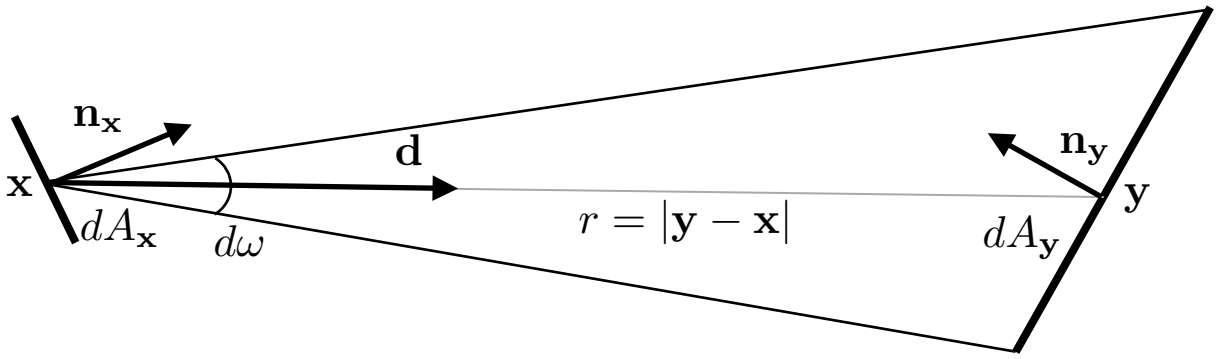
Sampling the light surface

- To compute the PDF of picking a direction from a point, we consider the differential solid angle subtended by a surface w.r.t. to another

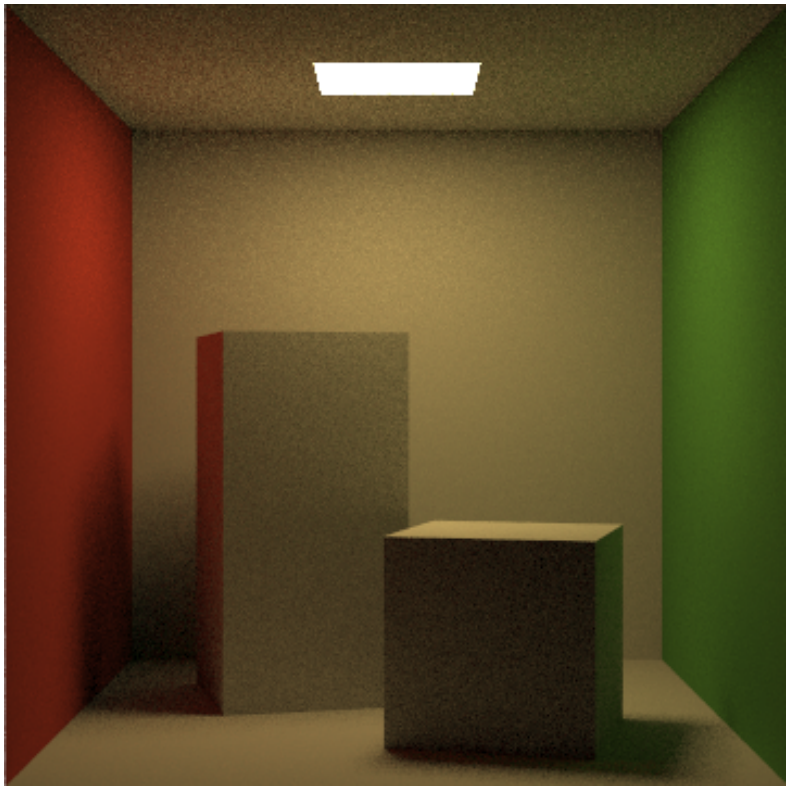
$$d\omega_{\mathbf{i}} = \frac{dA_{\mathbf{y}} |-\mathbf{i} \cdot \mathbf{n}_{\mathbf{y}}|}{|\mathbf{y} - \mathbf{x}|^2}$$

- From this, we derive the PDF of the direction as the product of the PDF of picking the point and the change of variable above

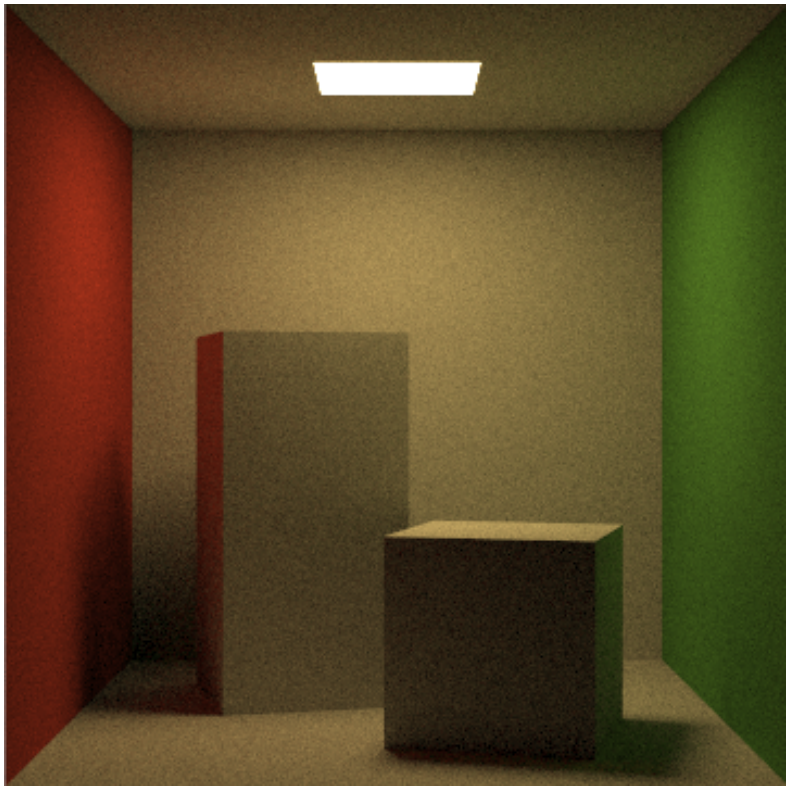
$$p(\mathbf{i}|\mathbf{y}) = p(\mathbf{y}) \frac{|\mathbf{y} - \mathbf{x}|^2}{|-\mathbf{i} \cdot \mathbf{n}_{\mathbf{y}}|} = \frac{|\mathbf{y} - \mathbf{x}|^2}{A|-\mathbf{i} \cdot \mathbf{n}_{\mathbf{y}}|}$$



Hemispherical vs light

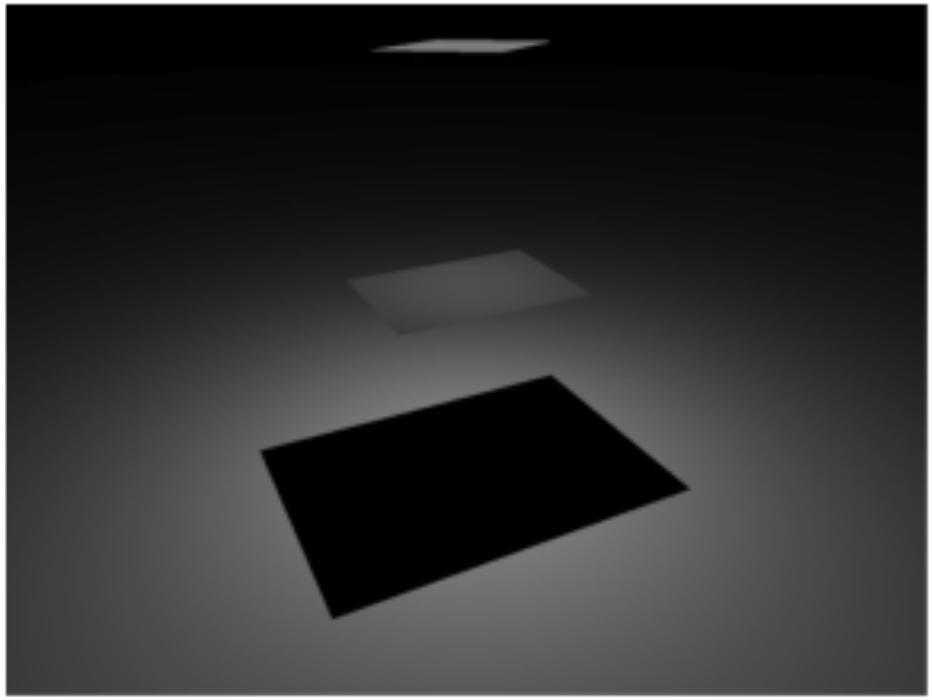


64 light samples

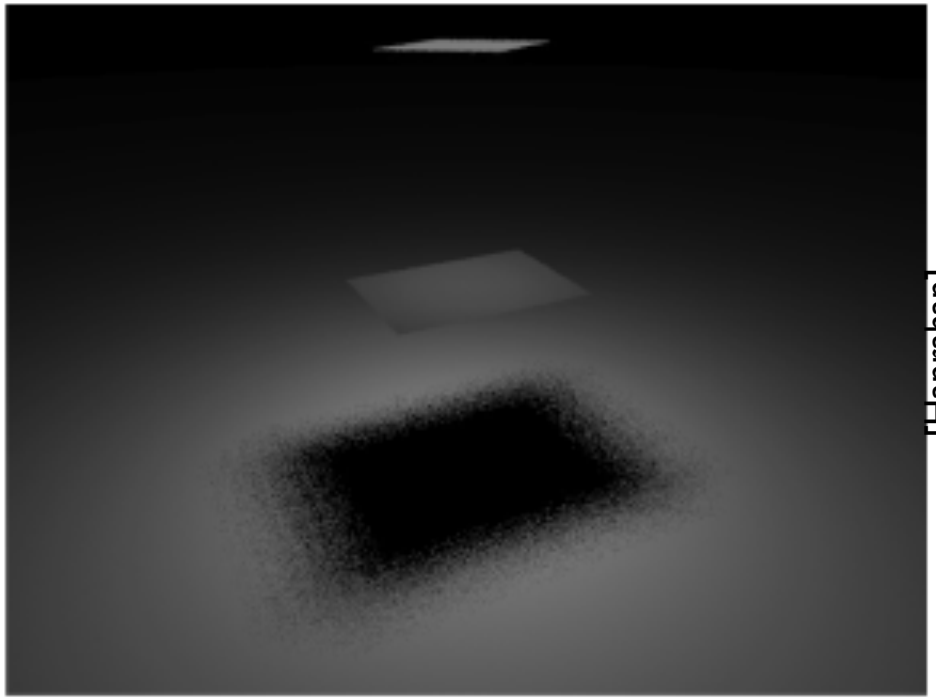


4096 hemispherical samples

Monte Carlo vs quadrature



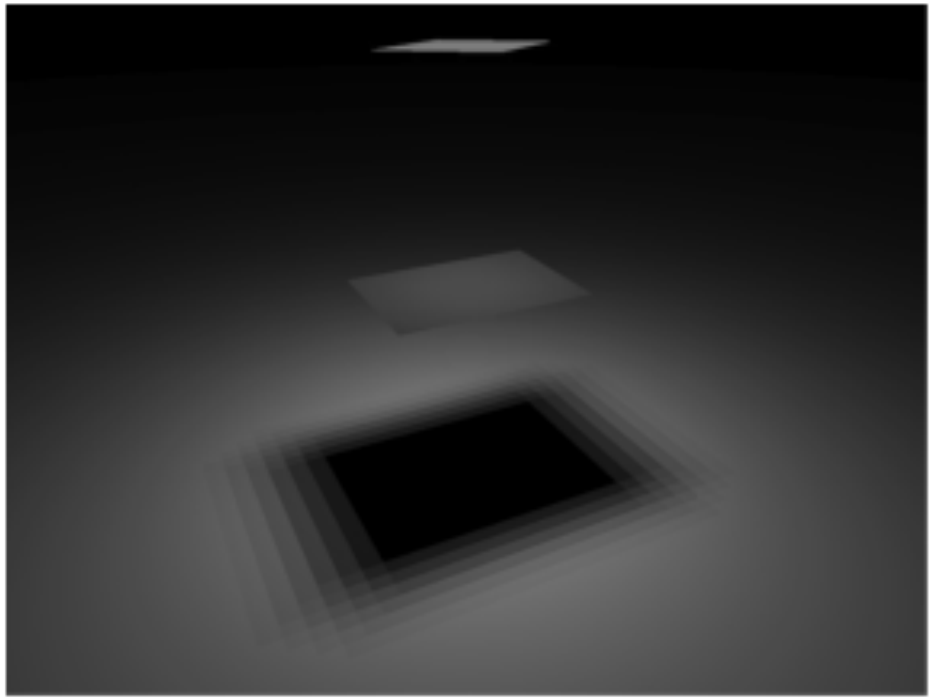
1 sample – center



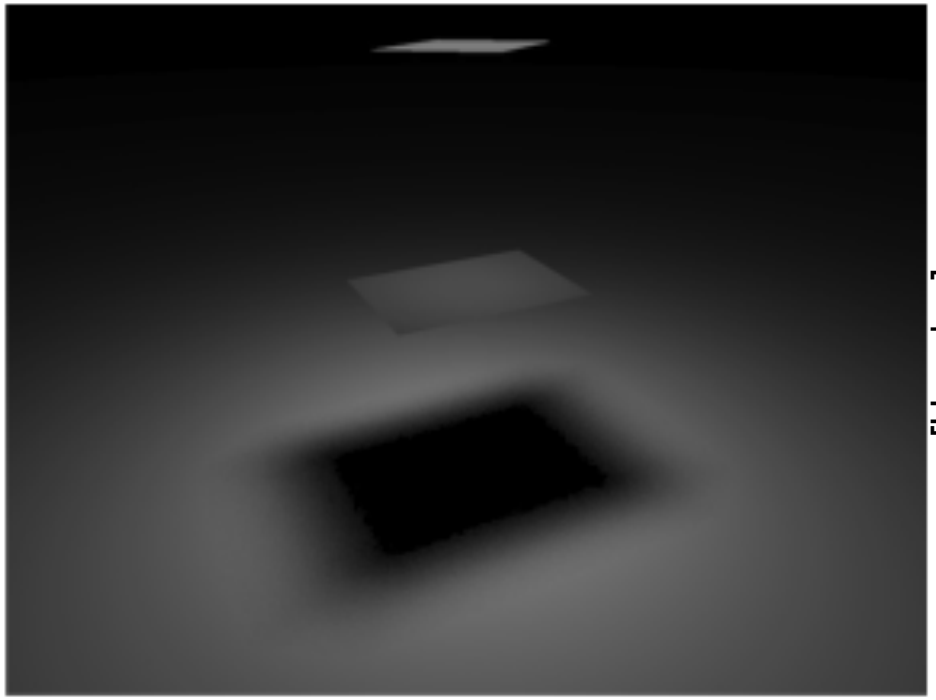
16 samples – random

[Hanrahan]

Monte Carlo vs quadrature



16 samples – center



16 samples – stratified

[Hanrahan]

Sampling multiple lights

- To handle multiple lights, we can estimate their contributions independently since each light integral is independent

$$L_d(\mathbf{x},\mathbf{o}) = \int_{H^2_\mathbf{x}} L_e(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_\mathbf{i} =$$

$$= \sum_l^{n_l} \int_{H^2_\mathbf{x}} L^l_e(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_\mathbf{i}$$

- We can estimate this by picking a direction for each light independently

$$L_d(\mathbf{x},\mathbf{o}) \approx \sum_l^{n_l} \frac{L^l_e(\mathbf{r}(\mathbf{x},\mathbf{i}^l_i),-\mathbf{i}^l_i) f(\mathbf{x},\mathbf{i}^l_i,\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}^l_i|}{p(\mathbf{y}^l_i|\mathbf{i}^l_i)}$$

Sampling multiple lights

- As the number of lights increases, this formulation becomes inefficient
- The alternative is to pick a *light at random for each sample*
  - here we are sampling from a uniform *discrete* distribution

$$l_i = \text{floor}(r \cdot n_l) \quad l \sim p(l) = \frac{1}{n_l}$$

- We can estimate the illumination by considering our light choice and folding its PDF into the complete PDF

$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e^{l_i}(\mathbf{r}(\mathbf{x}, \mathbf{i}_i), -\mathbf{i}_i) f(\mathbf{x}, \mathbf{i}_i, \mathbf{o}) |\mathbf{n_x} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i|\mathbf{y}_i|l_i)}$$

$$p(\mathbf{i}|\mathbf{y}|l) = p(\mathbf{i}|\mathbf{y})p(l) = \frac{|\mathbf{y} - \mathbf{x}|^2}{n_l A_l |-\mathbf{i} \cdot \mathbf{n_y}|}$$

Path tracing – direct illumination

- Integrate direct illumination in the path tracer by just inserting it in the main loop; emission is now computed only only first bounce since for other bounces it is integrated into the direct illumination estimation

```
vec3f shade_direct(scene* scene, ray3f ray) {
 <init>
 for(auto bounce : range(max_bounce)) {
    <intersection and environment>
    <eval point and material>
    if(!bounce) <emission>    // direct only on first bounce
    if(bounce < max_bounce-1) <handle direct>
    if(!is_delta(f)) <handle smooth>
    else <handle delta>
    <russian roulette>
    <recurse>
  }
  return l;
}
```


Path tracing – direct illumination

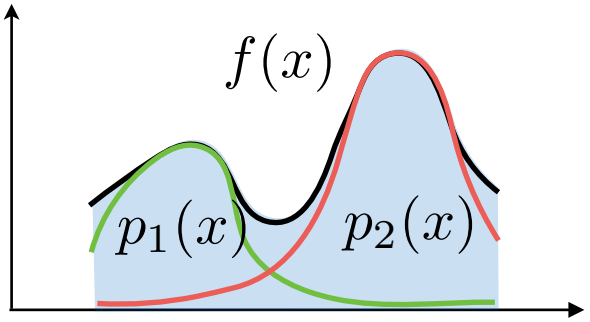
- Here we assume lights are only quads and that we have stored a list of objects that have emissive materials.

```
<handle direct> {
  // pick a light, a quad index and a quad uv
  auto light = sample_uniform(scene->lights, rand1f(rng));
  auto qid = sample_uniform(light->quads.size(), rand1f(rng));
  auto uv  = rand2f(rng);
  // evaluate the light point and material
  auto [lp, ln] = eval_point(light, qid, uv);
  auto [le, _ ] = eval_material(light, qid, uv);
  auto li = normalize(lp - p); auto ld = length(lp - p);
  // if the light is visible, accumulate its contribution
  if(!intersect(scene, {p, li, eps, ld})) {
  auto lpdf = (ld*ld) / (dot(li,nl) * quad_area(light,quad) *
      scene->lights.size() * light->quads.size());
    l += w * eval_emission(le, ln, -li) *
             eval_brdfcos(f, n, li, o) * dot(n, li) / lpdf;
 }
}
```


Multiple Importance Sampling

Multiple importance sampling

- We want to sample a complex function with importance sampling, but we do not know hoe to sample it well over the entire domain
- But we can sample different parts of the domain well with different sampling techniques
- We want to combine all techniques into a single robust one
- *Multiple importance sampling*: pick a technique at random and sample according to that that one
- Insight: avoid issues when a technique would not sample well a region



Multiple importance sampling

- Simple formulation: pick a technique at random, based on predetermined probabilities, and sample according to that that one
- Consider *M* sampling techniques, each with pdf p_j, that we want to sample with probability w_j
- Pick a distribution at random with probability w_j
- Pick a sample from that distribution
- Compute the combined pdf as weighted sum of all pdfs

$$E[I] = \frac{1}{N} \sum_{i}^{N} \frac{f(\mathbf{x}_i)}{\sum_{j}^{M} w_j p_j(\mathbf{x}_i)}$$

$$j \sim \{w_j\} \rightarrow \mathbf{x}_i \sim p_j(\mathbf{x})$$

$$\sum_{j} w_j = 1$$

Multiple importance sampling

- General formulation: pick samples from multiple techniques
- Consider *M* sampling techniques with pdfs p_j
- Pick N_j samples from each technique
- The combined estimator for these samples is

$$E[I] = \sum_j^M \frac{1}{N_j} \sum_i^{N_j} w_j(\mathbf{x}_{j,i}) \frac{f(\mathbf{x}_{j,i})}{p_j(\mathbf{x}_{j,i})}$$

$$\mathbf{x}_{j,i} \sim p_j(\mathbf{x})$$

$$\sum_j w_j(\mathbf{x}) = 1$$

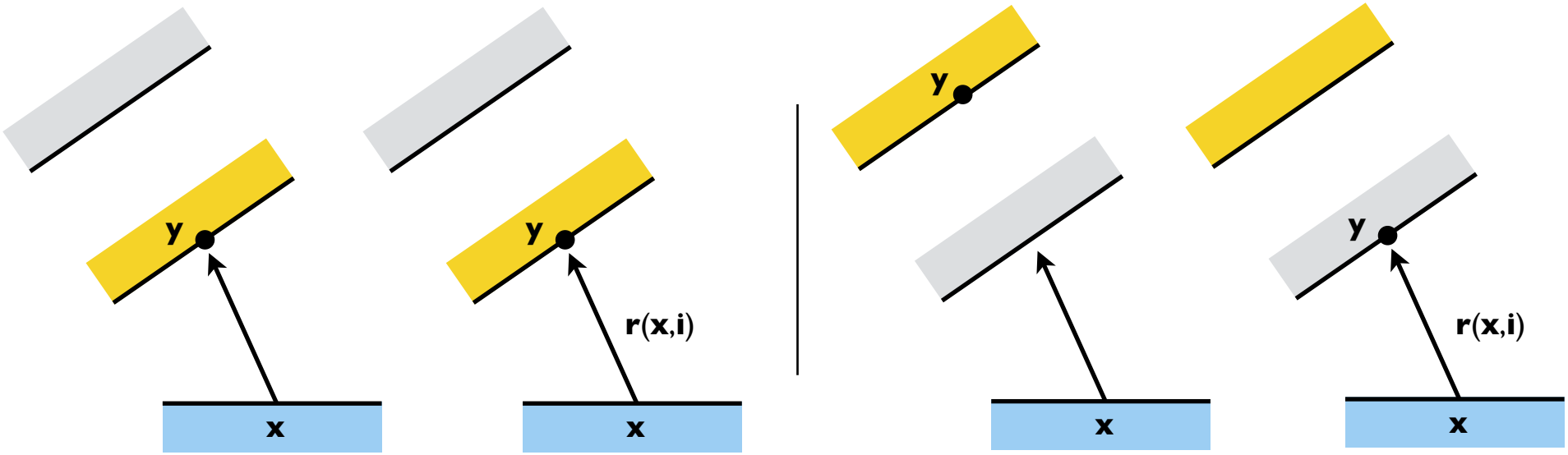
- Choose weights w_j to reduce variance by applying heuristics
- Balance heuristic: near optimal weights

$$w_j(\mathbf{x}) = \frac{N_j p_j(\mathbf{x})}{\sum_k^M N_k p_k(\mathbf{x})}$$

Sampling multiple lights – MIS

- A drawback of the previous formulation is that we might waste samples for lights that are occluded by other lights since we consider illumination only from the light we have chosen

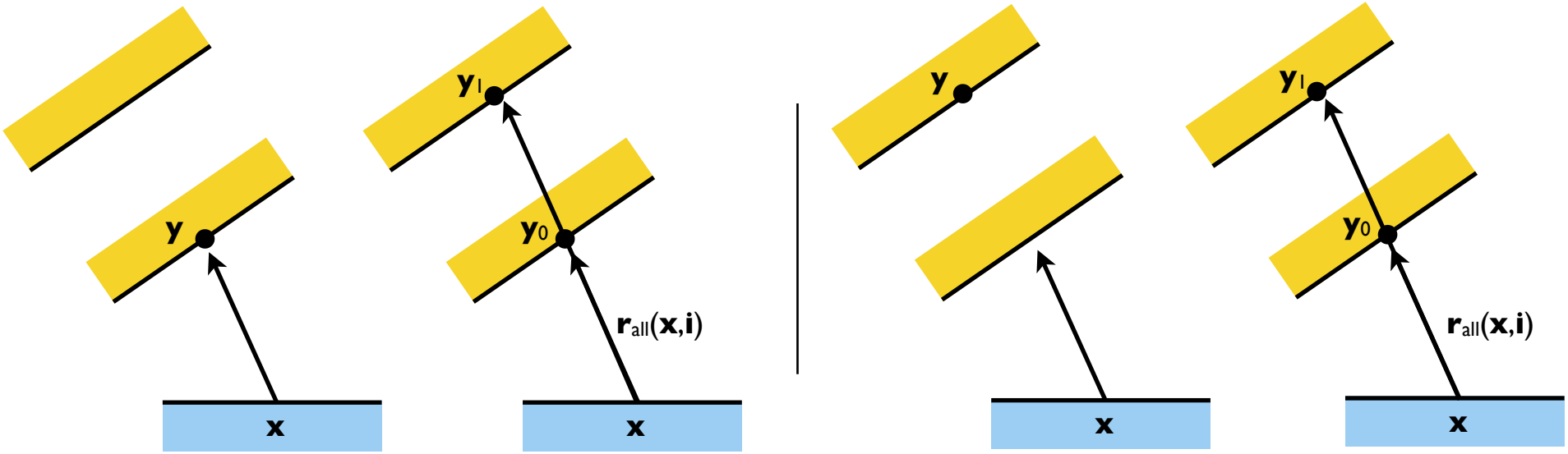
$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e^{l_i}(\mathbf{r}(\mathbf{x}, \mathbf{i}_i), -\mathbf{i}_i) f(\mathbf{x}, \mathbf{i}_i, \mathbf{o}) |\mathbf{n}_\mathbf{x} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i | \mathbf{y}_i | l_i)}$$



Sampling multiple lights – MIS

- The alternative is to always consider the first visible light, but correct the PDF to account for the fact that the sampled direction *i* might have come from any point y_j along the ray from the shading point *x*

$$L_d(\mathbf{x},\mathbf{o}) \approx \frac{L_e(\mathbf{r}(\mathbf{x},\mathbf{i}_i),-\mathbf{i}_i)f(\mathbf{x},\mathbf{i}_i,\mathbf{o})|\mathbf{n}_\mathbf{x}\cdot\mathbf{i}_i|}{p(\mathbf{i}_i|\{\mathbf{y}_l\})} \qquad \{\mathbf{y}_l\} = \mathbf{r}_{\text{all}}(\mathbf{x},\mathbf{i}_i)$$



Sampling multiple lights – MIS

- This is an example of multiple importance sampling, where we pick a direction based on different sampling strategies, in this case we have one distribution for each light

$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e(\mathbf{r}(\mathbf{x}, \mathbf{i}_i), -\mathbf{i}_i) f(\mathbf{x}, \mathbf{i}_i, \mathbf{o}) |\mathbf{n}_\mathbf{x} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i | \{\mathbf{y}_l\})} \qquad \{\mathbf{y}_l\} = \mathbf{r}_{\text{all}}(\mathbf{x}, \mathbf{i}_i)$$

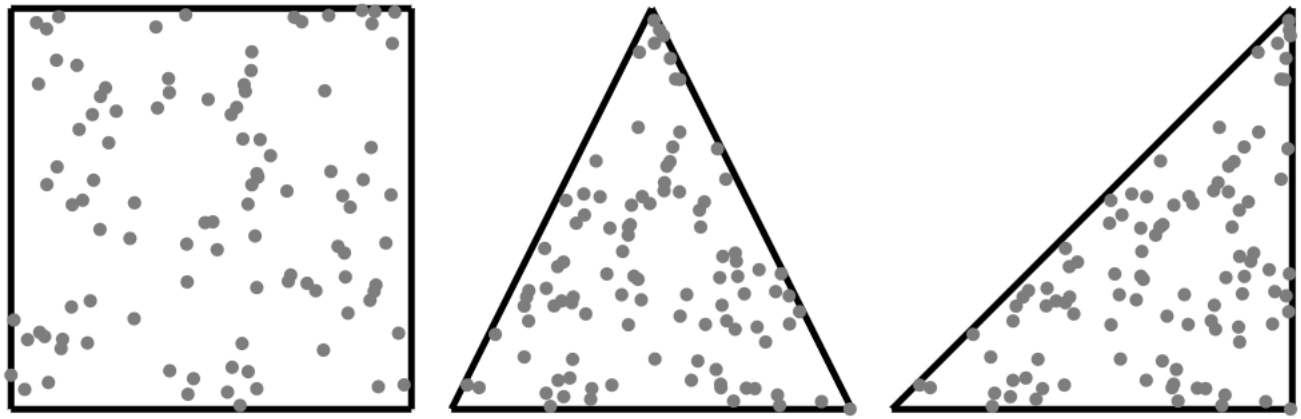
- We can compute the PDF by summing the PDFs for each light weighted by the probability of picking that light

$$p(\mathbf{i}, \{\mathbf{y}_l\}) = \sum_{\mathbf{y}_l \in \mathbf{r}_{\text{all}}(\mathbf{x}, \mathbf{i})} p(\mathbf{i}|\mathbf{y}_l) p(l) = \frac{|\mathbf{y}_l - \mathbf{x}|}{n_l A_l |-\mathbf{i} \cdot \mathbf{n}_{\mathbf{y}_l}|}$$

Sampling triangle meshes

- In general, our surfaces are described by triangle meshes
- We want to a procedure to pick point *uniformly* on the entire surface
- Let us start by picking points on a triangle with uniform probability by warping from the unit square

$$\mathbf{y} = \sqrt{r_0}(1 - r_1)\mathbf{v}_0 + (1 - \sqrt{r_1})\mathbf{v}_1 + r_1\sqrt{r_0}\mathbf{v}_2$$



Sampling triangle meshes

- Now we want to pick a triangle at random from the set of triangles with probability proportional to its area
- This is a non-uniform discrete distribution
- To sample it, we warp a random number in [0,1) to the distribution by
  - compute the triangle areas
  - from these, derive the cdf of the distribution
  - for each sample, we use a random number in [0,1) to find the smallest cdf value that is larger than the given one
  - its index is the wanted triangle index

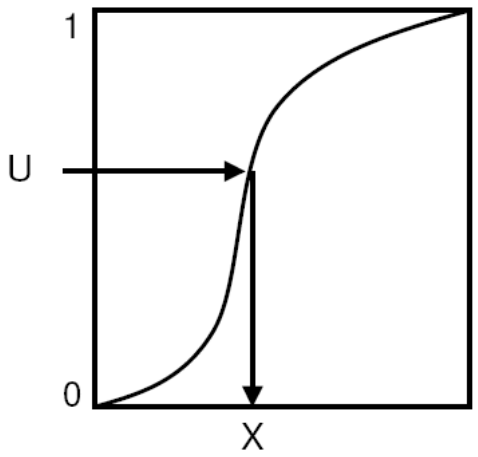
$$p_j = \frac{A_j}{\sum_k A_k} \quad c_j = \sum_{k=0}^{j} p_k \quad t_i = \arg\min_j c_j \geq r$$

Sampling discreet distribution

- Theorem: given the uniform random variable *U* in [0,1) and a cumulative distribution function *c*, then the random variable $X=c^{-1}(U)$ has *c* as its distribution

$$U \in [0,1) \sim p_U = 1 \rightarrow X = c^{-1}(U) \sim p_X = c$$

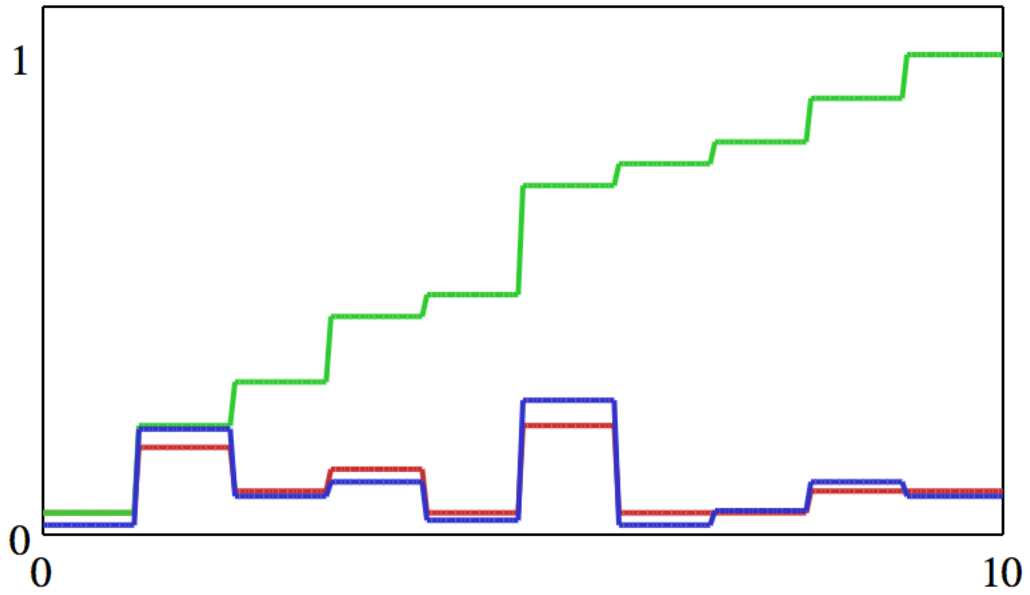
- To sample a random variable according to a distribution *p* do
  - compute its cumulative distribution function *c* and its inverse c^{-1}
  - use c^{-1} to warp uniform samples to the new domain



[Krivanec]

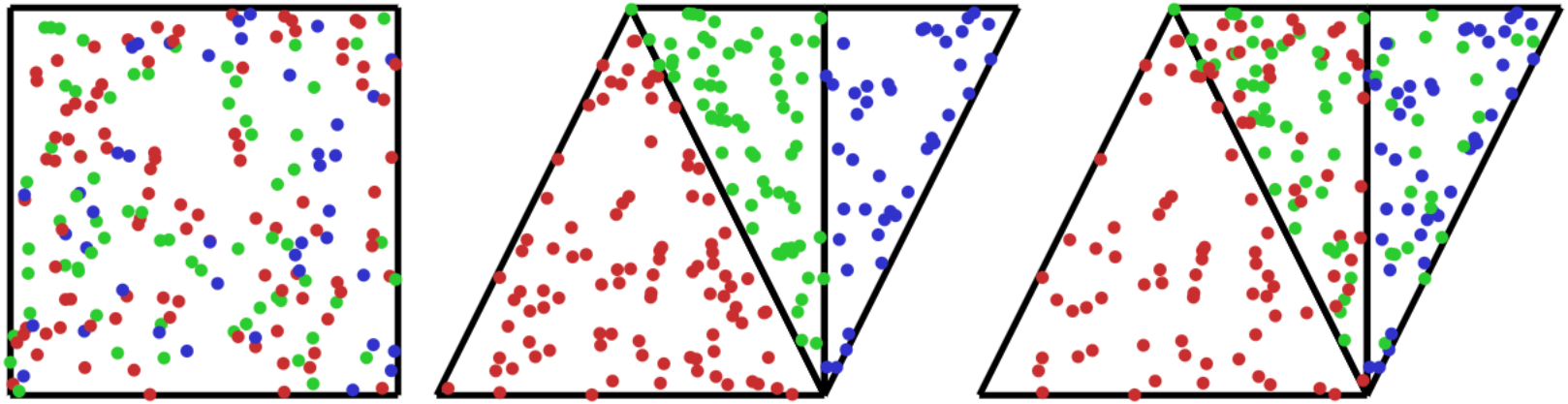
Sampling triangle meshes

- Example case with 10 triangles



Sampling triangle meshes

- Comparison between sampling uniformly the triangle index or sampling proportionally to area



Environment map

- So far we sampled environment maps uniformly from the surface
- But, just like small lights, small areas of high values in the environment will likely be missed when sampling
- As an alternative, we can consider the environment as a light and compute directions by picking its pixels with probability proportional to the pixel intensity weighted by its size
- This is a discreet distribution similar to the one used to pick between triangles based on their areas, so we can use the same procedure

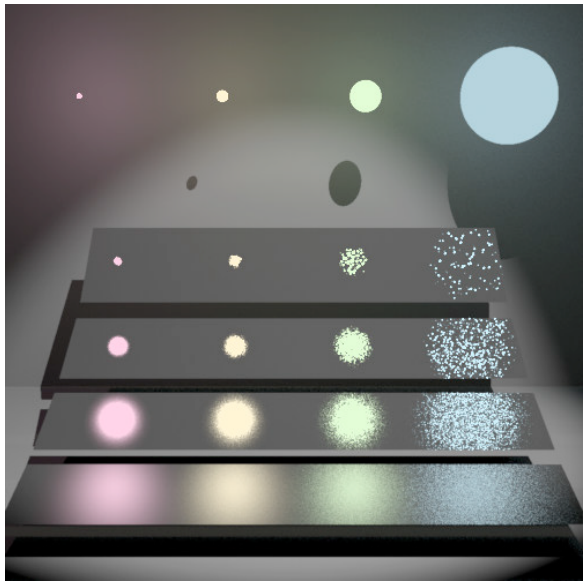
$$p_{(i,j)} = |\text{texture}(i,j)| \sin(\pi j/h)$$

$$\mathbf{i}^l = [\cos(2\pi i/w)\sin(\pi j/h), \cos(\pi j/h), \sin(2\pi i/w)\sin(\pi j/h)]$$

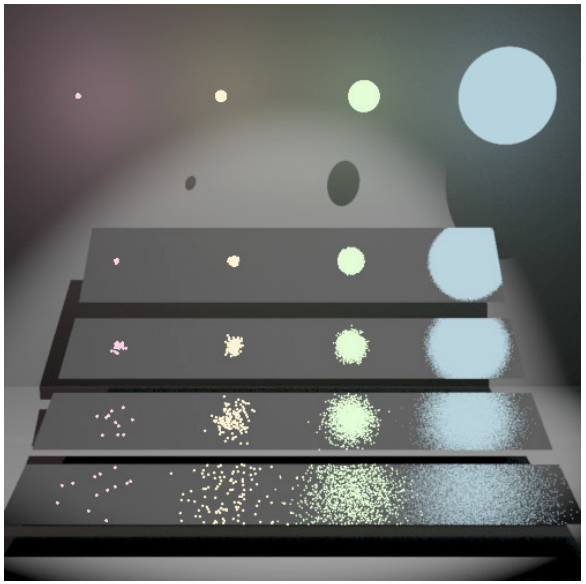
Sampling both light and BSDF

- Light sampling is not efficient for sharp material since we might miss the material lobe
- We can apply one more time multiple importance sampling to include both strategies

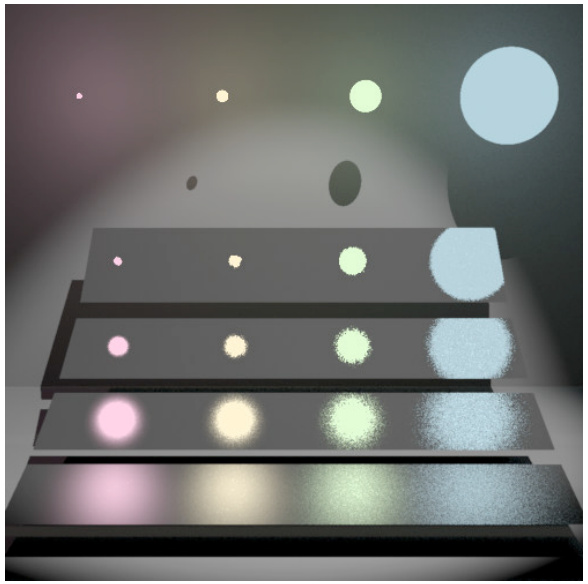
Sampling light



Sampling BSDF

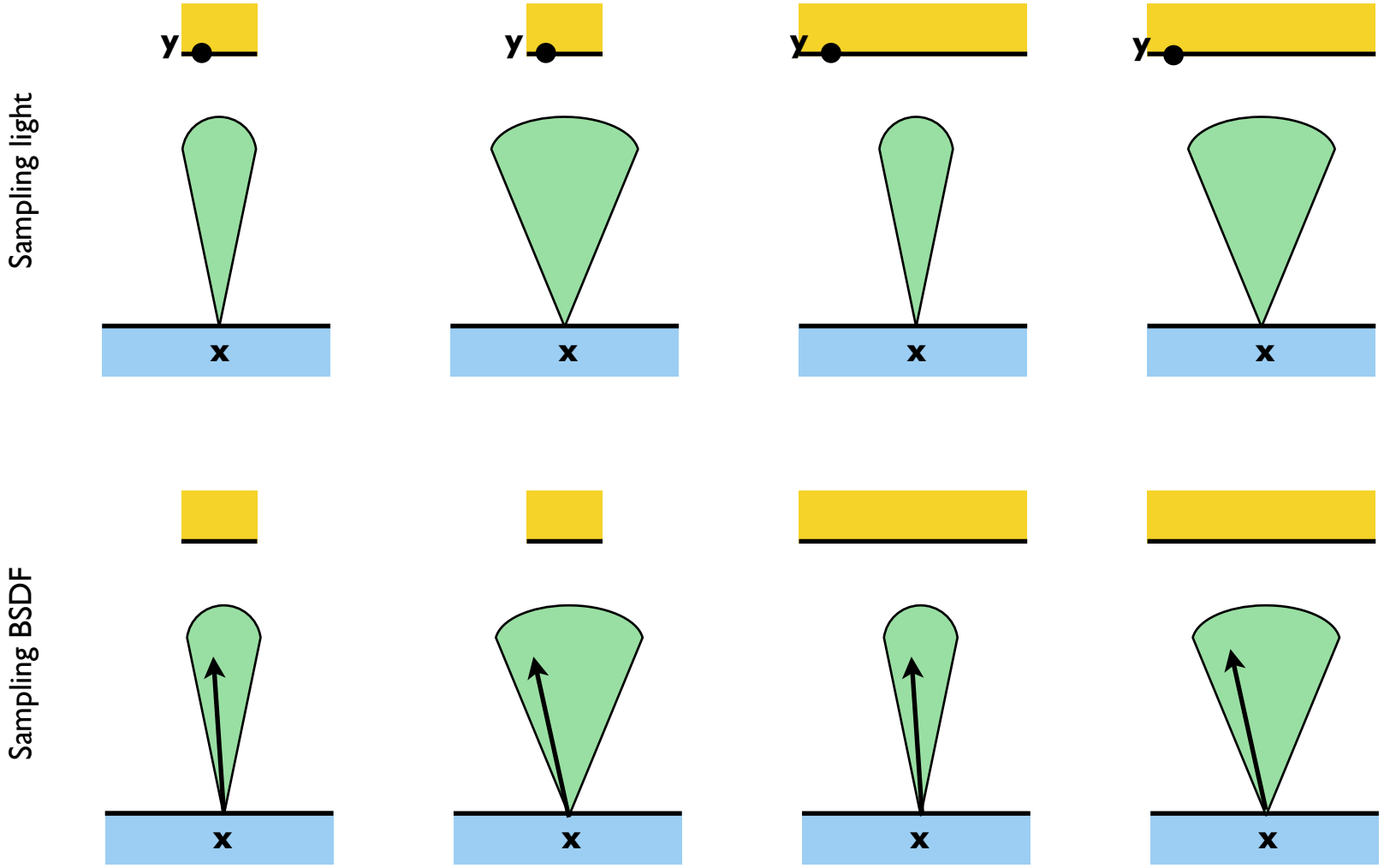


Sampling both



[Veach]

Sampling both light and BSDF



Sampling both light and BSDF

- We can use multiple importance sampling to sample according to both lights and BSDF, to get the benefit of both
- To do it, pick a technique at random, say with probability 0.5
- Pick a direction according to that
- Compute the final PDF as the sum of both PDFs

$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e(\mathbf{r}(\mathbf{x}, \mathbf{i}_i), -\mathbf{i}_i) f(\mathbf{x}, \mathbf{i}_i, \mathbf{o}) |\mathbf{n} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i | \mathbf{i}_i^b \text{ or } \mathbf{i}_i^l)}$$

$$\mathbf{i}_i = \begin{cases} \mathbf{i}_i^b & r < w_b \\ \mathbf{i}_i^l & r \geq w_b = 1 - w_l \end{cases} \qquad p(\mathbf{i} | \mathbf{i}^b \text{ or } \mathbf{i}^l) = w_b p_b(\mathbf{i}) + w_l p_l(\mathbf{i})$$

Sampling both light and BSDF

- The basic idea of sampling both BSDF and light can be use also for picking the direction for indirect illumination
- In fact, if we integrate BSDF and light sampling in a naive path tracer, without sampling direct explicitly, we get a simple solution that works well in a variety of cases

$$L(\mathbf{x},\mathbf{o}) \approx L_e(\mathbf{x},\mathbf{o}) + \frac{L(\mathbf{r}(\mathbf{x},\mathbf{i}_i),-\mathbf{i}_i) f(\mathbf{x},\mathbf{i}_i,\mathbf{o}) |\mathbf{n}_\mathbf{x} \cdot \mathbf{i}_i|}{p(\mathbf{i}_i | \mathbf{i}_i^b \text{ or } \mathbf{i}_i^l)}$$

$$\mathbf{i}_i = \begin{cases} \mathbf{i}_i^b & r < w_b \\ \mathbf{i}_i^l & r \geq w_b = 1 - w_l \end{cases} \qquad p(\mathbf{i} | \mathbf{i}^b \text{ or } \mathbf{i}^l) = w_b p_b(\mathbf{i}) + w_l p_l(\mathbf{i})$$

Path tracing – MIS

- We can implement a path tracer with MIS starting from a naive one and changing only the part when the next direction is computed
- Before rendering, we store an array of lights and their cdfs for sampling

```
void init_light(scene* scene) { // initialize lights and cdfs }
vec3f shade_mis(scene* scene, ray3f ray) {
 <init>
 for(auto bounce : range(max_bounce)) {
    <intersection and environment>
    <eval point and material>
    <emission>
    if(!is_delta(f))
      <handle smooth>
    else <handle delta>
    <russian roulette>
    <recurse>
  }
  return l;
}
```



```
vec3f shade_mis(scene* scene, ray3f ray) {
  auto l = vec3f{0}, w = vec3f{1};
  for(auto bounce : range(max_bounce)) {
    auto isec = intersect(scene, ray);
    if(!isec.hit) {
      l += w * eval_environment(scene, ray.d); break; }
    auto [p, n] = eval_point(isec);     // eval pos, norm
    auto [e, f] = eval_material(isec);  // eval bsdf
    auto o = -ray.d;                    // outgoing
    l += w * eval_emission(e, n, o);    // emission
    if(!is_delta(f)) { // sample smooth brdfs (fold cos into f)
      auto i = rand1f(rng) < 0.5 ?            // incoming
          sample_brdfcos(f,n,o,rand1f(rng),rand2f(rng)) :
          sample_lights(p,rand1f(rng),rand1f(rng),rand2f(rng));
      w *= eval_brdfcos(f,n,i,o) * 2 /
        (sample_brdfcos_pdf(f,n,i,o)+sample_lights_pdf(p,i));
    } else {          // sample sharp brdfs
      auto i = sample_delta(n, rand2f(rng)); // incoming
      w *= eval_delta(f,n,i,o) / sample_delta_pdf(i);
    }
    if(rand1f() >= min(1,max(w))) break;     // russian roulette
    w *= 1 / min(1,max(w));
    ray = {x, i};                           // recurse
  }
  return l;
}
```


Path tracing – sampling lights

- A light is either an emissive object or an environment
- We store a separate array mostly as convenience

```
// a light is either an object or an environment and their cdf
struct light {
  object*       object = nullptr;
  environment*  environment = nullptr;
  vector<float> cdf = {};
};

// initialize an array of lights and compute cdfs
void init_light(scene* scene) {
  <handle objects>
  <handle environments>
}
```


Path tracing – sampling lights

- For each object, we add a light if the object is emissive
- For sampling later, we store an sampling cdf computed as the accumulate triangle areas, which are treated as their weights

```
<handle_objects> =
  for (auto object : scene->objects) {
    if (object->material->emission == zero3f) continue;
    if (object->shape->triangles.empty()) continue;
    auto light = add_light(scene, object);
    auto shape = object->shape;
    light->cdf = vector<float>(shape->triangles.size());
    for(auto idx : range(shape->cdf.size())) {
      auto t = shape->triangles[idx];
      light->cdf[idx] = triangle_area(shape->positions[t.x],
            shape->positions[t.y], shape->positions[t.z]);
      if(idx) shape->cdf[idx] = shape->cdf[idx-1];
    }
 }
```


Path tracing – sampling lights

- For each environment, we add a light if the environment is emissive
- The environment map cdf is computed as the accumulation of the pixel weights, which is the pixels' intensities scaled by their relative size

```
<handle environments> =
  for (auto environment : scene->environments) {
    if (environment->emission == zero3f) continue;
    auto light = add_light(scene, environment);
    auto texture = environment->emission_tex;
    auto size    = texture_size(texture);
    light->cdf = vector<float>(size.x * size.y);
    for (auto idx : range(light->cdf.size())) {
      auto ij = vec2i{idx % size.x, idx / size.x};
      auto theta = (ij.y + 0.5) * pi / size.y;
      auto value = lookup_texture(texture, ij);
      light->cdf[idx] = max(value) * sin(theta);
      if (idx) light->cdf[i] += light->cdf[i - 1];
    }
```


Path tracing – sampling lights

- To sample a direction towards lights, we first pick a light at random

```
vec3f sample_lights(scene* scene, vec3f p,
                    float rl, float rel, vec2f ruv) {
  auto lid = sample_uniform(scene->lights.size(), rl);
  auto light = scene->lights[lid];
  if (light->object) {
    <handle object>
  } else if (light->environment) {
    <handle environment>
  }
}
```


Path tracing – sampling lights

- For the object case, we then pick a random triangle and a random uv for that triangle, based on the previous sampling helpers
- From these, we compute the position on the light and set the direction accordingly

```
<handle object> =
  if (light->object) {
    auto tid   = sample_discrete_cdf(light->cdf, rel);
    auto uv    = sample_triangle(ruv);
    auto [lp, _] = eval_point(light->object, tid, uv);
    return normalize(lp - position);
  }
```


Path tracing – sampling lights

- For the environment case, we then pick a texel at random, and compute the environment uv at the center of that telex, assuming that environment map pixels are small
- We then compute the direction corresponding to this uv

```
if (light->environment) {
 auto texture = light->environment->emission_tex;
 auto tid  = sample_discrete_cdf(light->cdf, rel);
 auto size = texture_size(texture);
 auto uv   = vec2f{(tid % size.x + puv.x) / size.x,
                   (tid / size.x + puv.y) / size.y};
 return transform_direction(environment->frame,
        {cos(uv.x * 2 * pi) * sin(uv.y * pi),
         cos(uv.y * pi),
         sin(uv.x * 2 * pi) * sin(uv.y * pi)});
}
```


Path tracing – lights pdf

- To lights pdf for a direction is computed as the sum of each light pdf

```
float sample_lights_pdf(scene* scene, vec3f p, vec3f d) {
  auto pdf = 0.0f;
  for (auto light : scene->lights) {
    if (light->object) {
      <handle object>
    } else if(light->environment) {
      <handle object>
    }
  }
  pdf *= sample_uniform_pdf(scene->lights.size());
  return pdf;
}
```


Path tracing – lights pdf

- For the object case, the pdf is computed as the pdfs of all triangles that lie along the given direction from the shaded point
- The triangle pdfs are given by the formulas before

```
<handle object> =
  // accumulate pdf over all intersections
  auto lpdf = 0.0f; auto np = p;
  for (auto bounce = 0; bounce < 100; bounce++) {
    auto isec = intersect(light->object, np, d);
    if (!isec) break;
    // compute pdf of this triangle
    auto [lp, ln] = eval_point(isec);
    // prob triangle * area triangle = area triangle mesh
    auto area = light->cdf.back();
    lpdf += distance_squared(lp, p) / (abs(dot(ln, d)) * area);
    np = lp + direction * 1e-3f; // continue ray
  }
  pdf += lpdf;
```

Path tracing – lights pdf

- For the environment case, the pdf is computed as the pdf of choosing that pixel over the solid angle of the selected pixel

```
<handle environment> =
  auto texture = light->environment->emission_tex;
  auto size = texture_size(texture);
  auto wl = transform_direction(
            inverse(light->environment->frame), direction);
  auto texcoord = vec2f{atan2(wl.z, wl.x) / (2 * pif),
                  acos(clamp(wl.y, -1.0f, 1.0f)) / pif};
  if (texcoord.x < 0) texcoord.x += 1;
  auto i    = clamp((int)(texcoord.x * size.x), 0, size.x - 1);
  auto j    = clamp((int)(texcoord.y * size.y), 0, size.y - 1);
  auto prob = sample_discrete_cdf_pdf(light->cdf, j*size.x+i) /
                      light->cdf.back();
  auto angle = (2 * pif / size.x) * (pif / size.y) *
                       sin(pif * (j + 0.5f) / size.y);
  pdf += prob / angle;
```

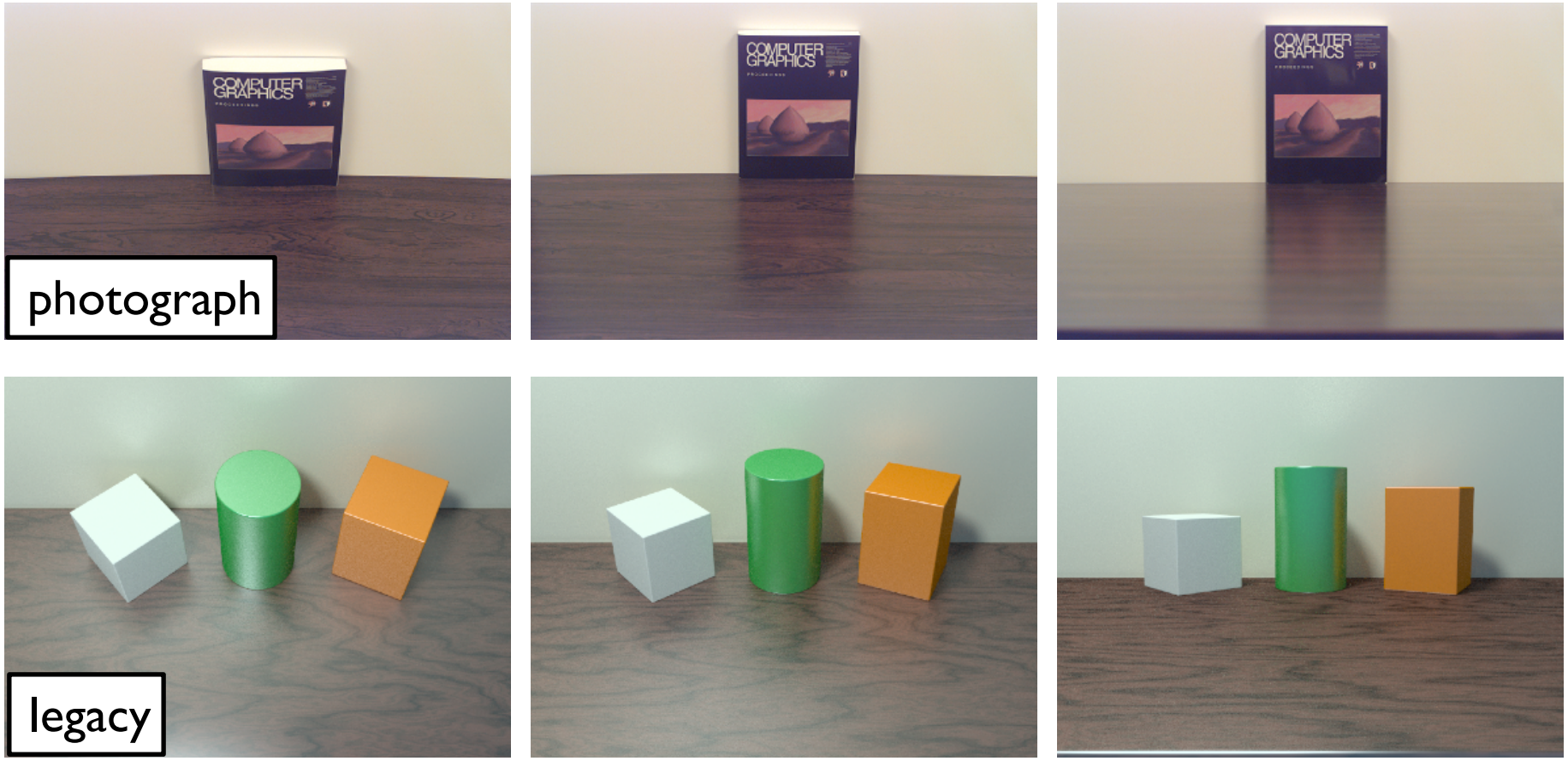

Path tracing – lights pdf

- Overall we made use a few simple support functions

```
// sample an index with uniform probability
int sample_uniform(int size, float r) {
  return clamp((int)(r * size), 0, size - 1); }
float sample_uniform_pdf(int size) {
  return 1.0 / (float)size; }
// sample an index with the given cdf
int sample_discrete_cdf(vector<float> cdf, float r) {
  r = clamp(r * cdf.back(), 0, cdf.back() - 0.00001);
  auto idx = (int)(std::upper_bound(cdf.begin(),
                  cdf.end(), r) - cdf.begin());
  return clamp(idx, 0, cdf.size() - 1); }
float sample_discrete_cdf_pdf(vector<float> cdf, int idx) {
  if (idx == 0) return cdf.at(0);
  else return cdf.at(idx) - cdf.at(idx - 1); }
// sample a triangle uv
vec2f sample_triangle(vec2f ruv) {
  return {1 - sqrt(ruv.x), ruv.y * sqrt(ruv.x)}; }
```

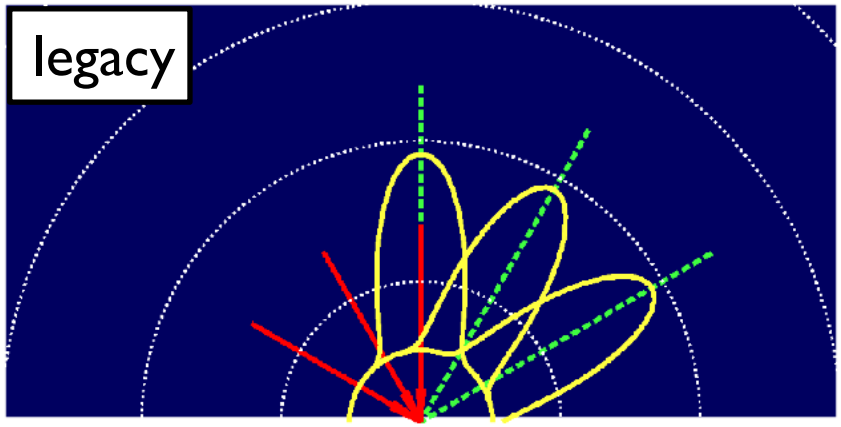

Materials

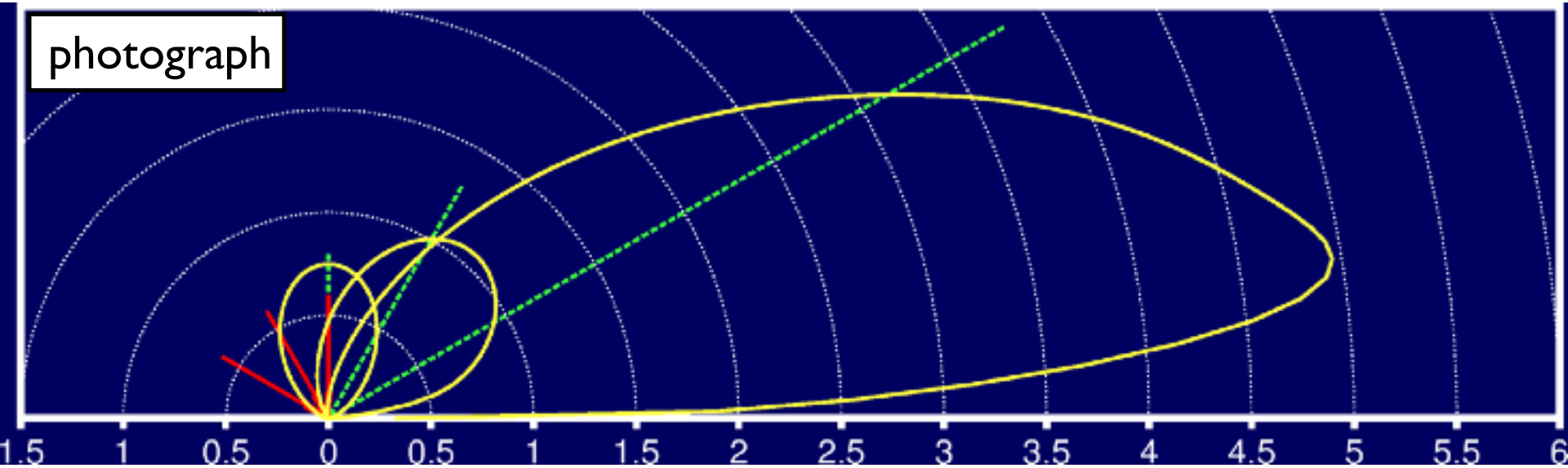
Legacy vs physically-based



[Lafortune et al]

Legacy vs physically-based

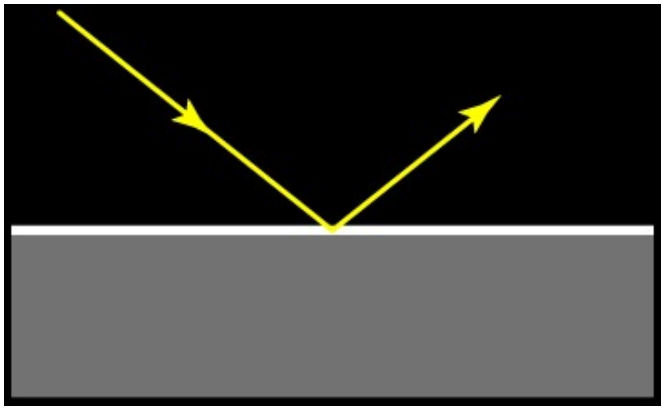


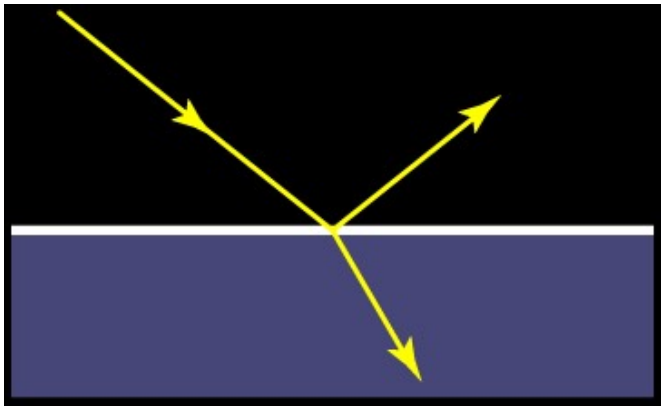


Materials

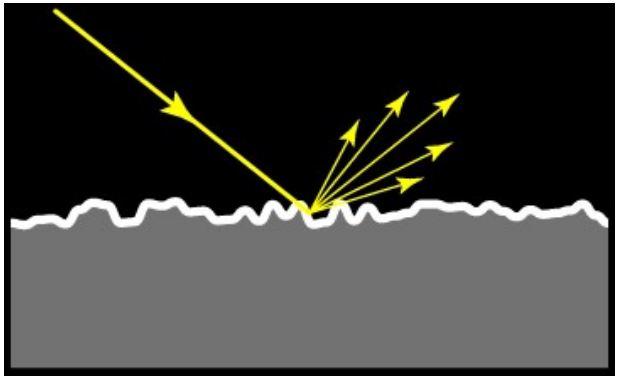
- We'll focus on BSDF and in particular in covering opaque materials
- Different BSDFs comes from different optical behavior and "structure" of the surface
- In theory, most materials would need an entirely different BSDF to account for their peculiarities
  - not practical since editing and exchanging them becomes hard
- In practice, use one "good-enough" physical model for most cases and custom materials only when absolutely necessary

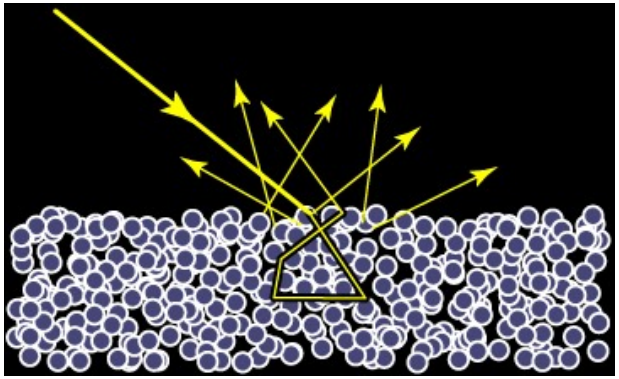
Smooth surfaces





Rough surfaces

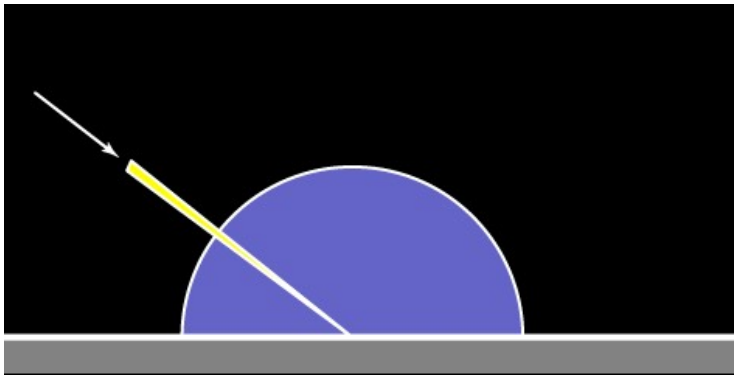


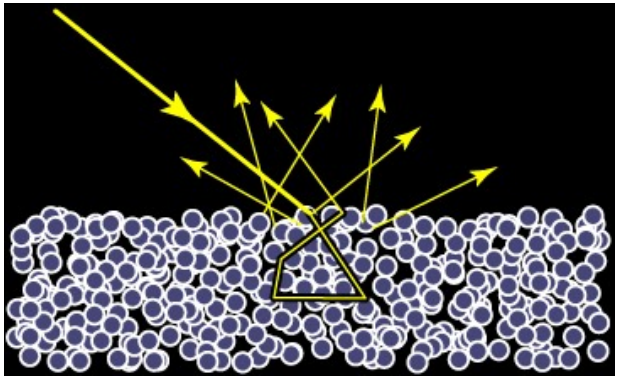


Diffuse reflection

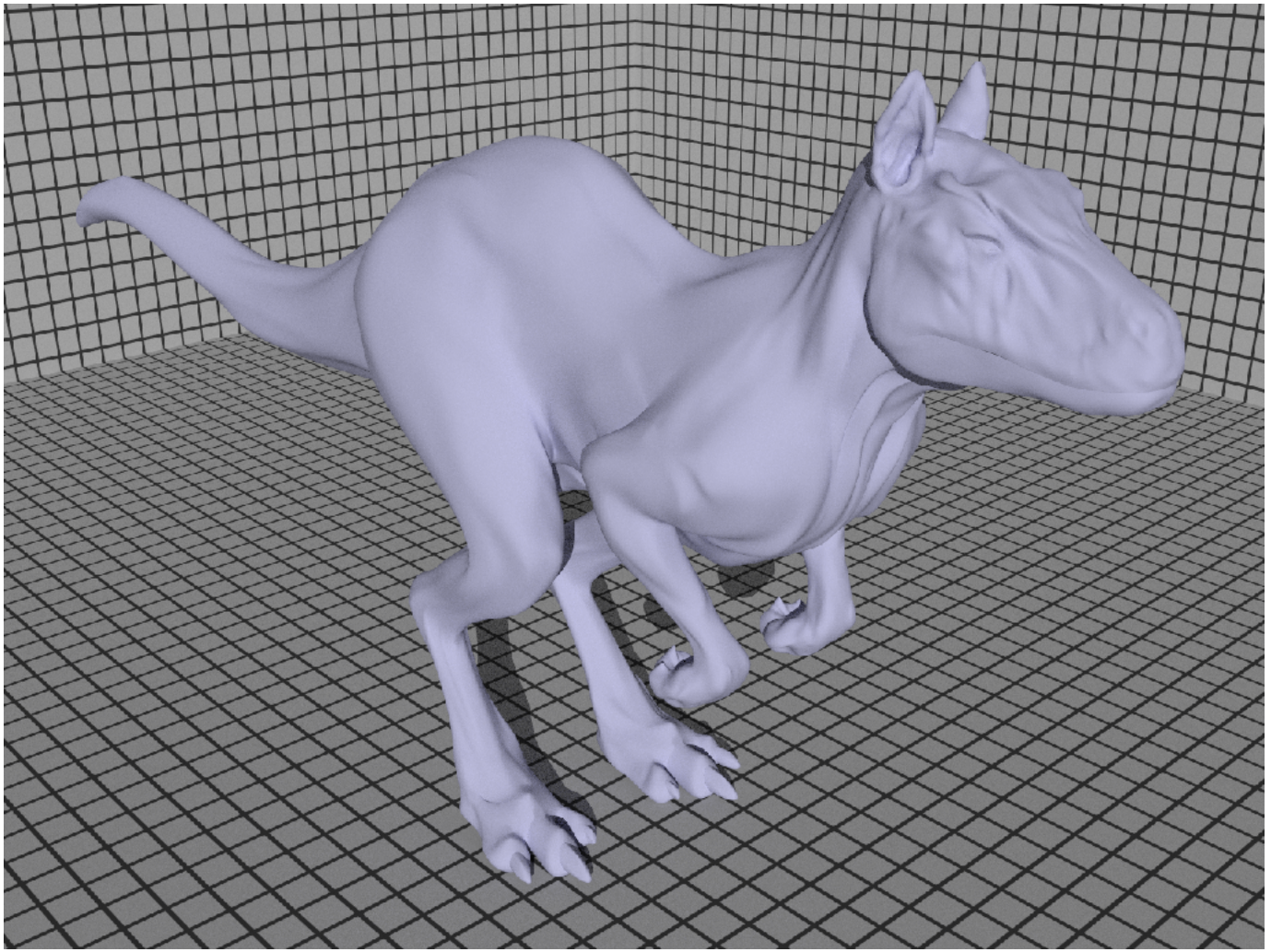
- Diffuse reflection from a completely matte surface
- Light is scattered in all direction uniformly
- BRDF is a constant
- For energy conservation, k_d is in [0,1] and normalized by *pi*

$$f_d(\mathbf{i}, \mathbf{o}) = \frac{k_d}{\pi}$$





Diffuse reflection

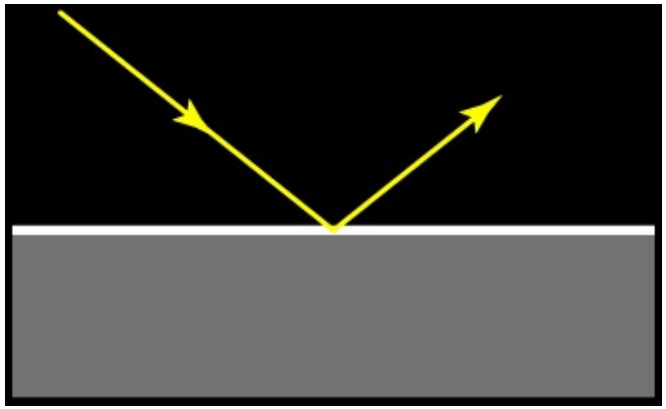


[Pharr et al./pbrt]

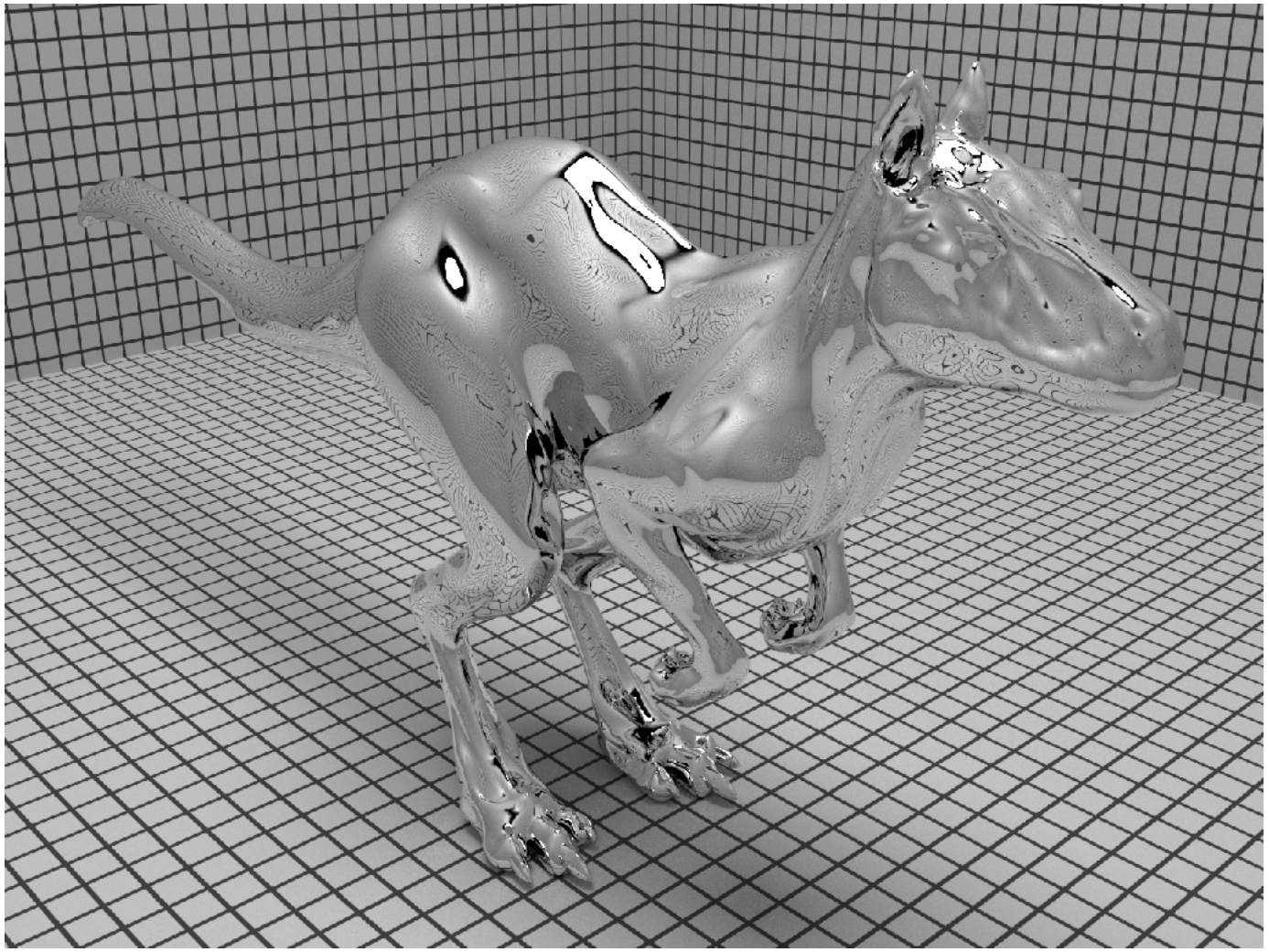
Ideal reflection

- Ideal surface with no imperfection made of an ideal conductor
- Light is scattered entirely along the mirror direction
- BRDF is a "delta function" — informally, zero everywhere except at one point, and with finite integral

$$f_r(\mathbf{i},\mathbf{o},\mathbf{n}) = F(\mathbf{o},\mathbf{n})\frac{\delta(\mathbf{i},\mathbf{r})}{|\mathbf{n}\cdot\mathbf{i}|} \qquad \mathbf{r} = -\mathbf{o} + 2(\mathbf{n}\cdot\mathbf{o})\mathbf{n}$$



Ideal reflection

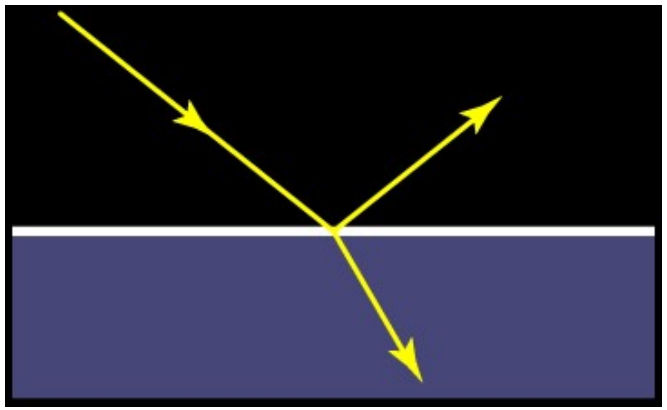


[Pharr et al./pbrt]

Ideal refraction

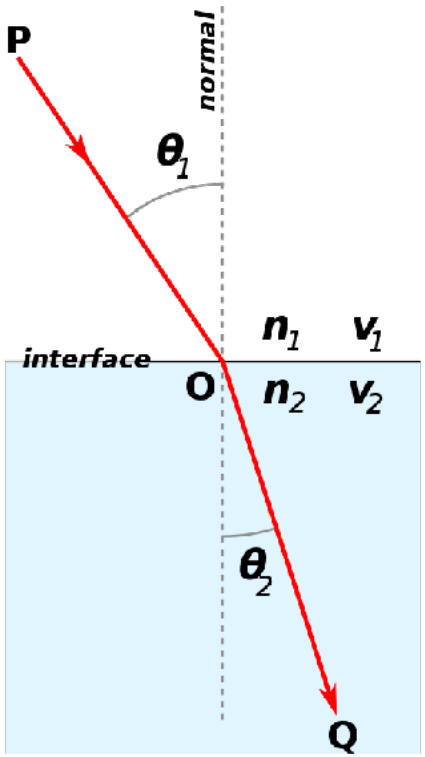
- Ideal surface made of a dialectic has both reflection and refraction
- Light is scattered in two directions, above and below the surface
- BRDF refraction is a "delta function" around the refracted direction

$$f_t(\mathbf{i},\mathbf{o},\mathbf{n}) = \frac{1}{\eta^2}(1 - F(\mathbf{o},\mathbf{n}))\frac{\delta(\mathbf{i},\mathbf{t})}{|\mathbf{n}\cdot\mathbf{i}|} \quad \mathbf{t} = -\frac{1}{\eta}\mathbf{o} + \left[\frac{1}{\eta}(\mathbf{n}\cdot\mathbf{o}) - \sqrt{1 + \frac{1}{\eta^2}(|\mathbf{n}\cdot\mathbf{o}|^2 - 1)}\right]\mathbf{n}$$

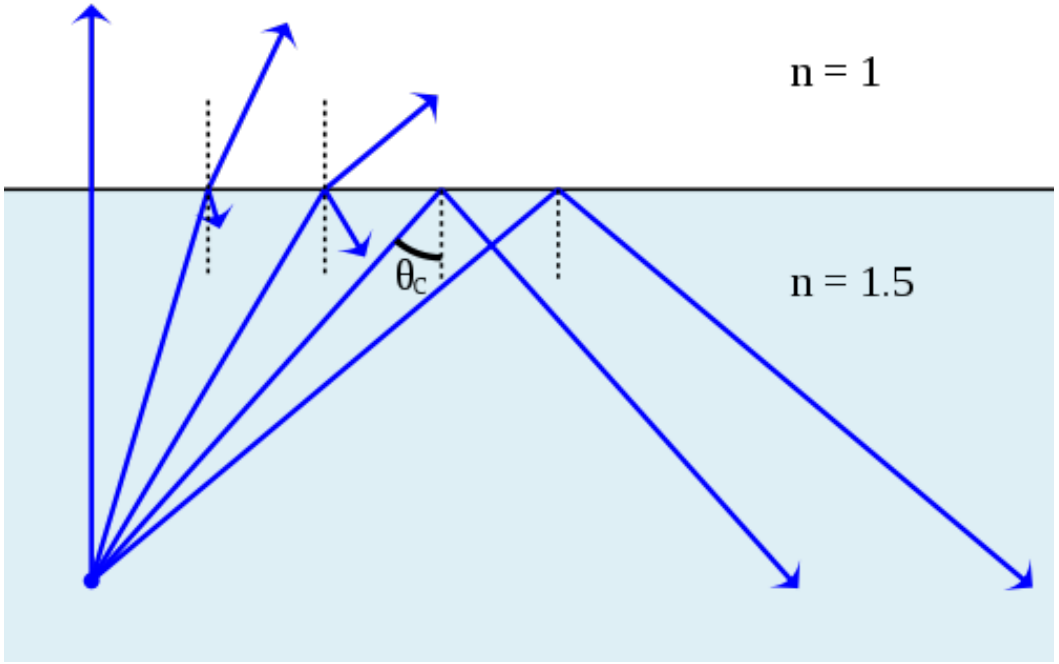


$$\eta = \frac{\eta_i}{\eta_o}$$

Ideal refraction

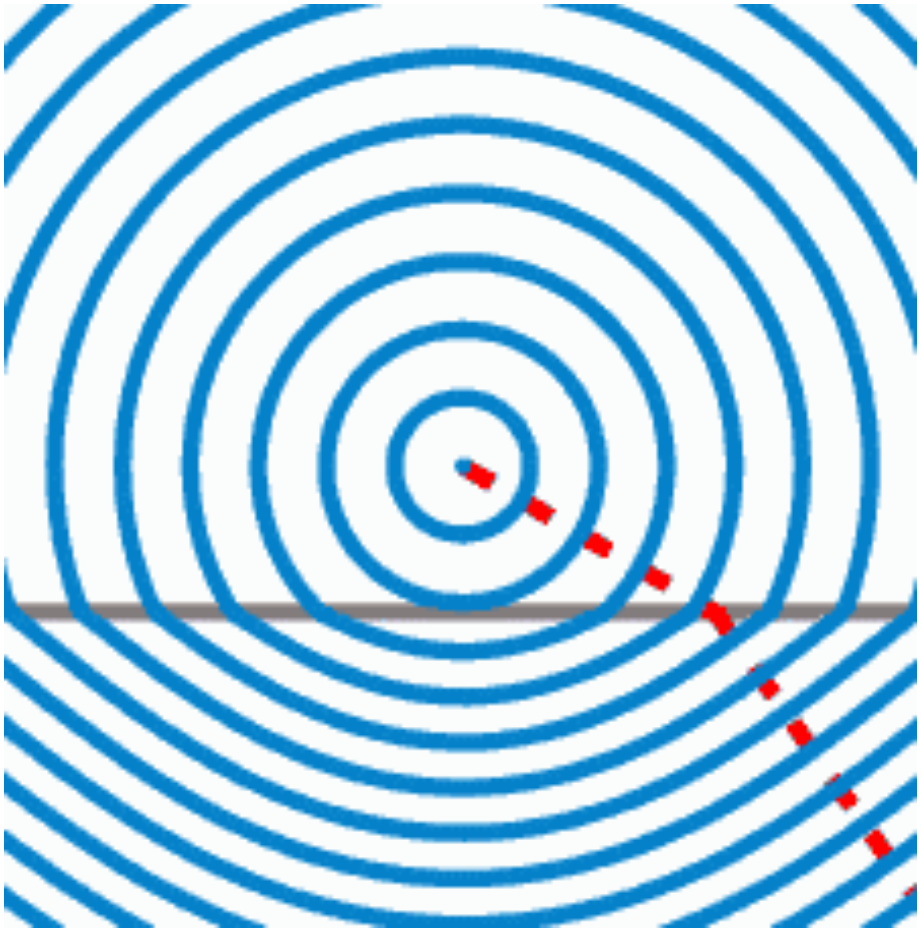


$$\mathbf{t} = -\frac{1}{\eta}\mathbf{o} + \left[\frac{1}{\eta}(\mathbf{n}\cdot\mathbf{o}) - \sqrt{1+\frac{1}{\eta^2}(|\mathbf{n}\cdot\mathbf{o}|^2-1)}\right]\mathbf{n}$$



[Wikipedia]

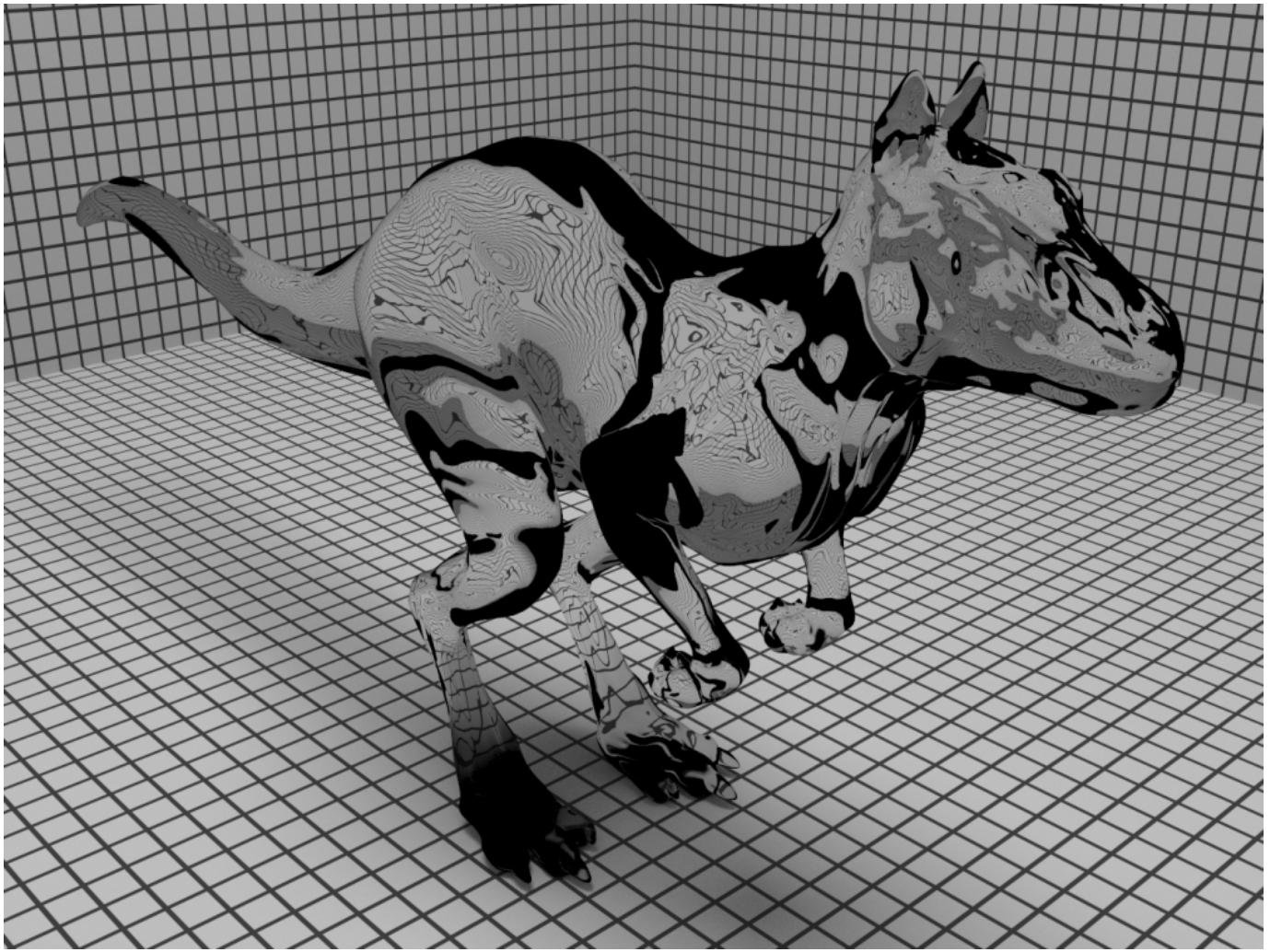
Ideal refraction



$$f_t(\mathbf{i},\mathbf{o},\mathbf{n}) = \frac{1}{\eta^2}(1 - F(\mathbf{o},\mathbf{n}))\frac{\delta(\mathbf{i},\mathbf{t})}{|\mathbf{n}\cdot\mathbf{i}|}$$

[Wikipedia]

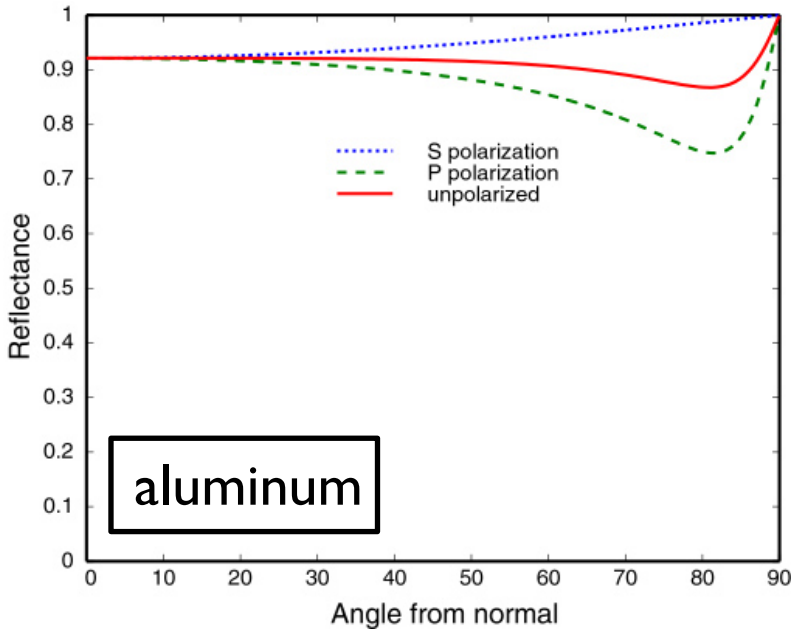
Ideal refraction

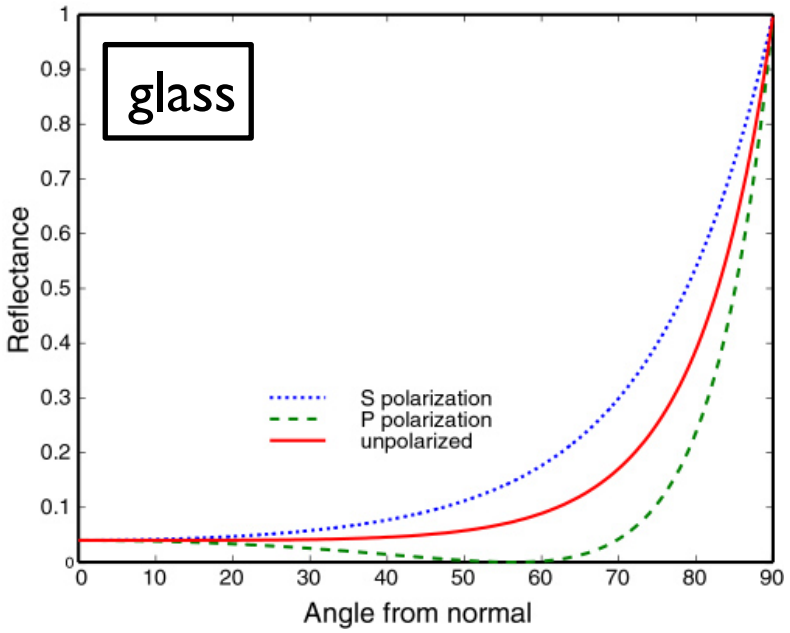


[Pharr et al./pbrt]

Fresnel reflection and transmission

- Fresnel term: determines k_r and k_t from intrinsic material properties
- Conductors: small variation from surface color to 1
- Dielectrics: drastic change from small values (~0.04) to 1
- Depends on wavelength





[Westin]

Schlick's Fresnel approximation

- Real-time graphics uses an approximation for Fresnel due to Schlick
- Mostly valid for dialectics similar to glass
- Less valid for metals, but hard to see the difference
- In general, better to use more appropriate approximations for dielectrics to capture TIR

$$F_s(\mathbf{o},\mathbf{n}) = F_0 + (1 - F_0)(1 - \mathbf{o}\cdot\mathbf{n})^5 \quad F_0 = \frac{(\eta-1)^2}{(\eta+1)^2}$$

Fresnel formulas

- Dielectric Fresnel

$$F_d(\mathbf{o},\mathbf{n}) = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left(1+\frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right)$$

$$g = \sqrt{\eta^2 - 1 + c^2} \quad \text{and} \quad c = |\mathbf{o}\cdot\mathbf{n}|$$

- Conductor Fresnel: too complex to write here
  - https://seblagarde.wordpress.com/2013/04/29/memo-on-fresnel-equations/
  - http://www.pbr-book.org/3ed-2018/Reflection_Models/Specular_Reflection_and_Transmission.html

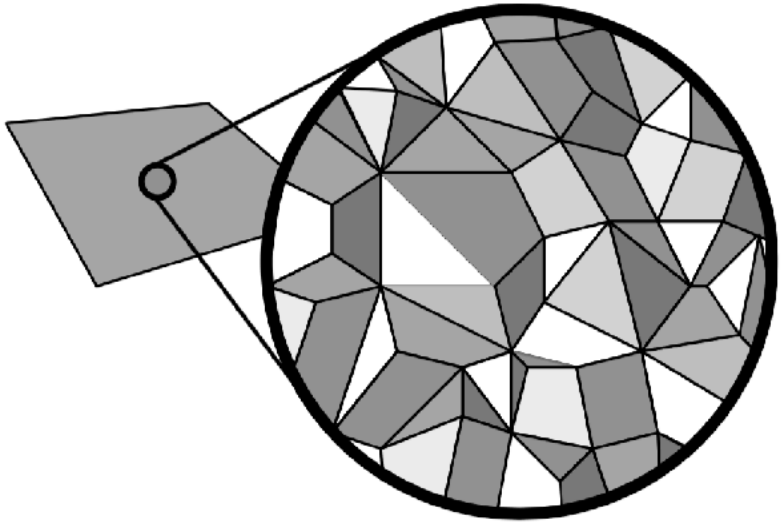
Fresnel term

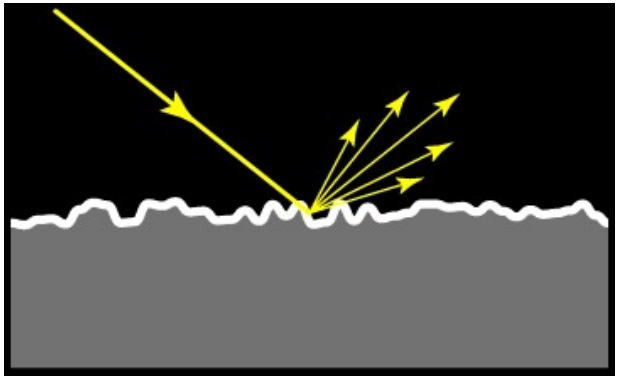
[Wikipedia]

Microfacet reflection

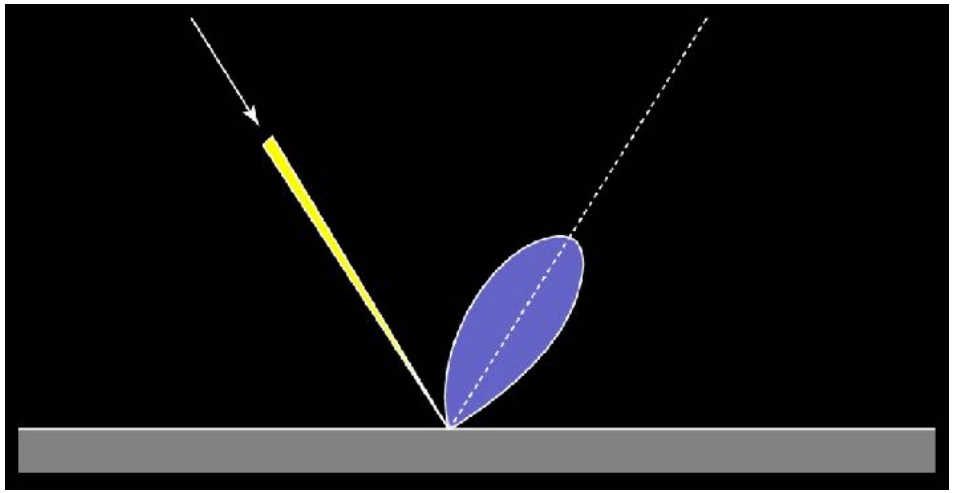
- Surface modeled as a collection of randomly oriented microfacet
- When light hits a surface, it hits one microfacet
- BRDF is the "average" value of all these reflection events
  - reflection centered around the bisector h
- See "Microfacet Models for Refraction through Rough Surfaces"

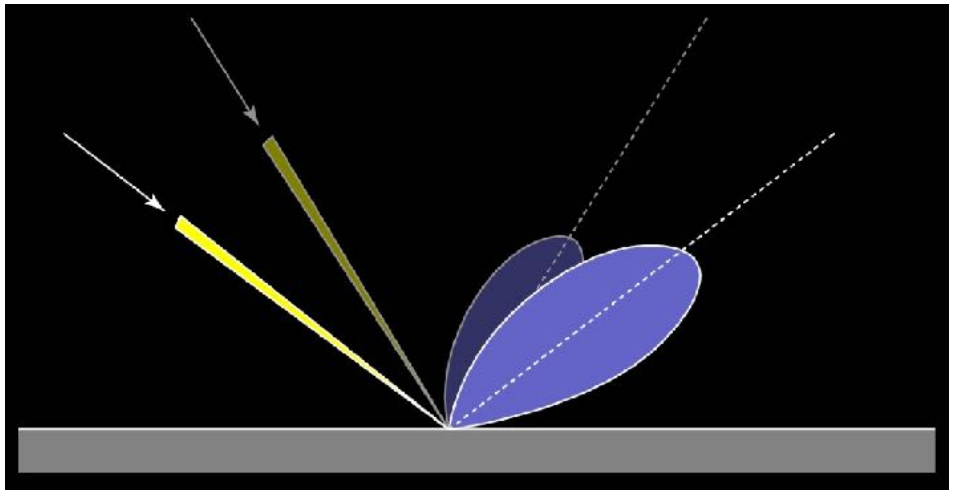
[Westin]

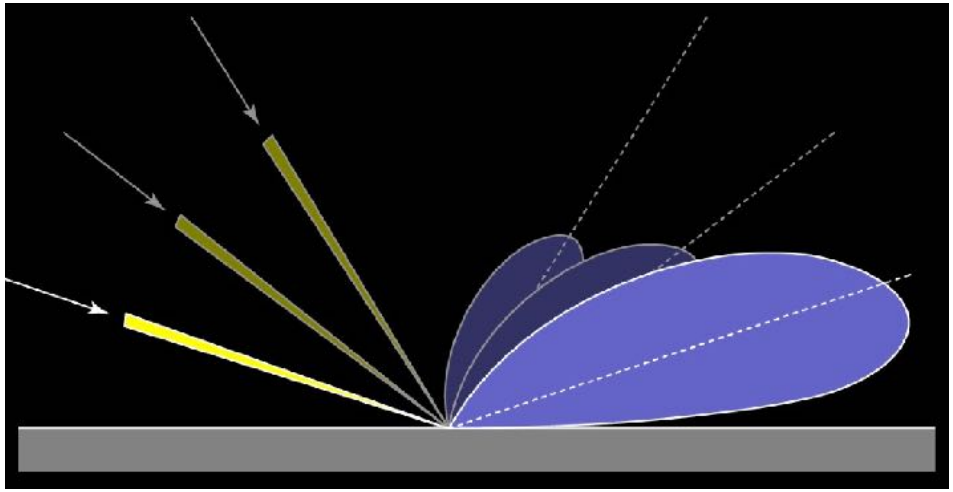




Microfacet reflection



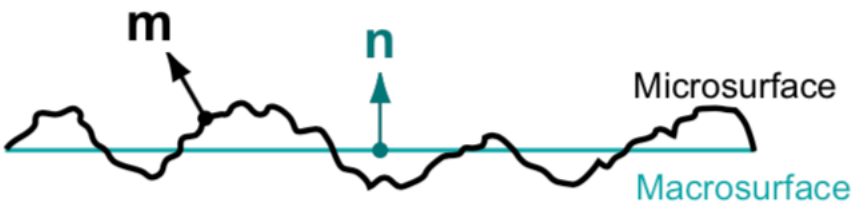




Microfacet reflection

- Normalized product of three terms: fresnel *F*, microfacet distribution *D* and shadow-masking term *G*
- Fresnel *F* accounts for difference at grazing angle
- Distribution *D* describes the statistical distribution of microfacets
- Shadow-making *G* corrects for facets not visible from light or view

$$f_r(\mathbf{i},\mathbf{o},\mathbf{n}) = \frac{F(\mathbf{o},\mathbf{n})D(\mathbf{h},\mathbf{n})G(\mathbf{i},\mathbf{o},\mathbf{n})}{4|\mathbf{n}\cdot\mathbf{i}||\mathbf{n}\cdot\mathbf{o}|} \qquad \mathbf{h} = \frac{\mathbf{i}+\mathbf{o}}{|\mathbf{i}+\mathbf{o}|}$$



[Walter et al.]

Microfacet reflection

- Different choices for *D*, mostly quite similar
- Blinn-Phong: simple, but lacks physical basis

$$D_p(\mathbf{h}, n) = \frac{n+2}{2\pi}(\mathbf{n}\cdot\mathbf{h})^n \qquad D_p(\mathbf{h}, \alpha) = \frac{1}{\pi\alpha^2}(\mathbf{n}\cdot\mathbf{h})^{\frac{2}{\alpha^2}-2}$$

- Beckman: first distribution used in graphics

$$D_b(\mathbf{h}, \alpha) = \frac{1}{\pi\alpha^2(\mathbf{n}\cdot\mathbf{h})^4}\exp\left(\frac{(\mathbf{n}\cdot\mathbf{h})^2 - 1}{\alpha^2(\mathbf{n}\cdot\mathbf{h})^2}\right)$$

- GGX: currently the most used one – slightly better fit to real data

$$D_g(\mathbf{h}, \alpha) = \frac{\alpha^2}{\pi((\mathbf{n}\cdot\mathbf{h})^2(\alpha^2 - 1) + 1)^2}$$

Microfacet reflection

- Different choices for *G*, quite a bit different – use Smith formulation

$$G(\mathbf{i}, \mathbf{o}, \mathbf{n}) = G_1(\mathbf{i}, \mathbf{n})G_1(\mathbf{o}, \mathbf{n})$$

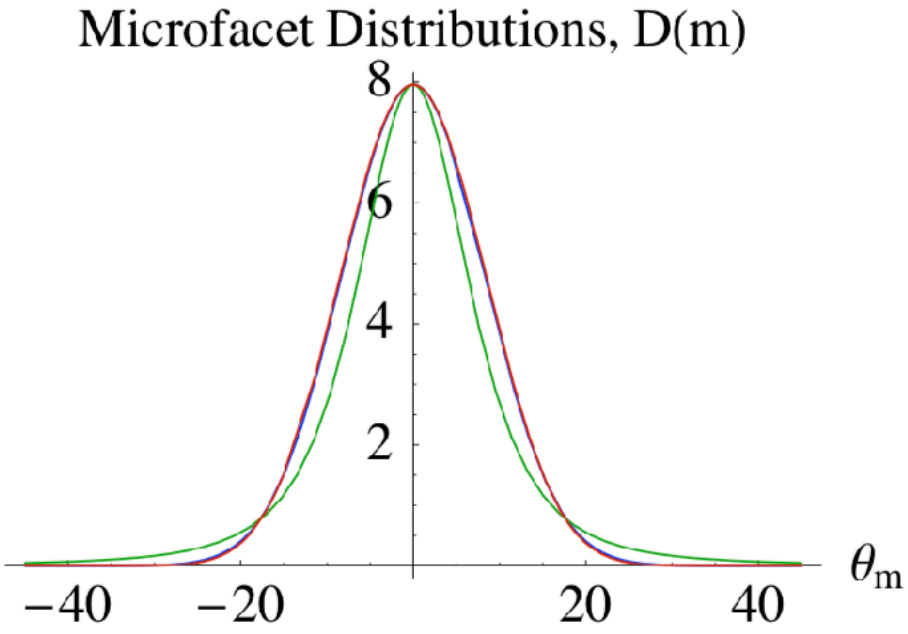
- Blinn-Phong: approximate as Beckmann
- Beckmann: slow analytic solution, better to use approximation

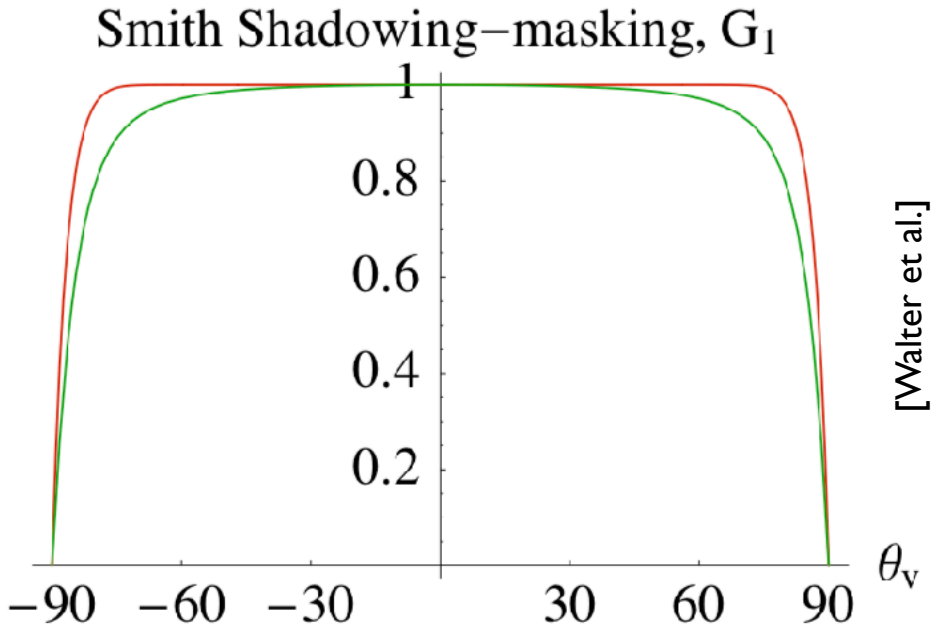
$$G_{1,b}(\mathbf{v}, \mathbf{n}) \approx \begin{cases} \frac{3.535a+2.181a^2}{1+2.276a+2.577a^2} & a < 1.6 \\ 1 & \text{otherwise} \end{cases} \qquad a = \frac{|\mathbf{n} \cdot \mathbf{v}|}{\alpha_b \sqrt{1 - |\mathbf{n} \cdot \mathbf{v}|^2}}$$

- GGX: analytic solution

$$G_{1,g}(\mathbf{v}, \mathbf{n}) = \frac{2|\mathbf{n} \cdot \mathbf{v}|}{|\mathbf{n} \cdot \mathbf{v}| + \sqrt{\alpha_g^2 + (1 - \alpha_g^2)|\mathbf{n} \cdot \mathbf{v}|^2}}$$

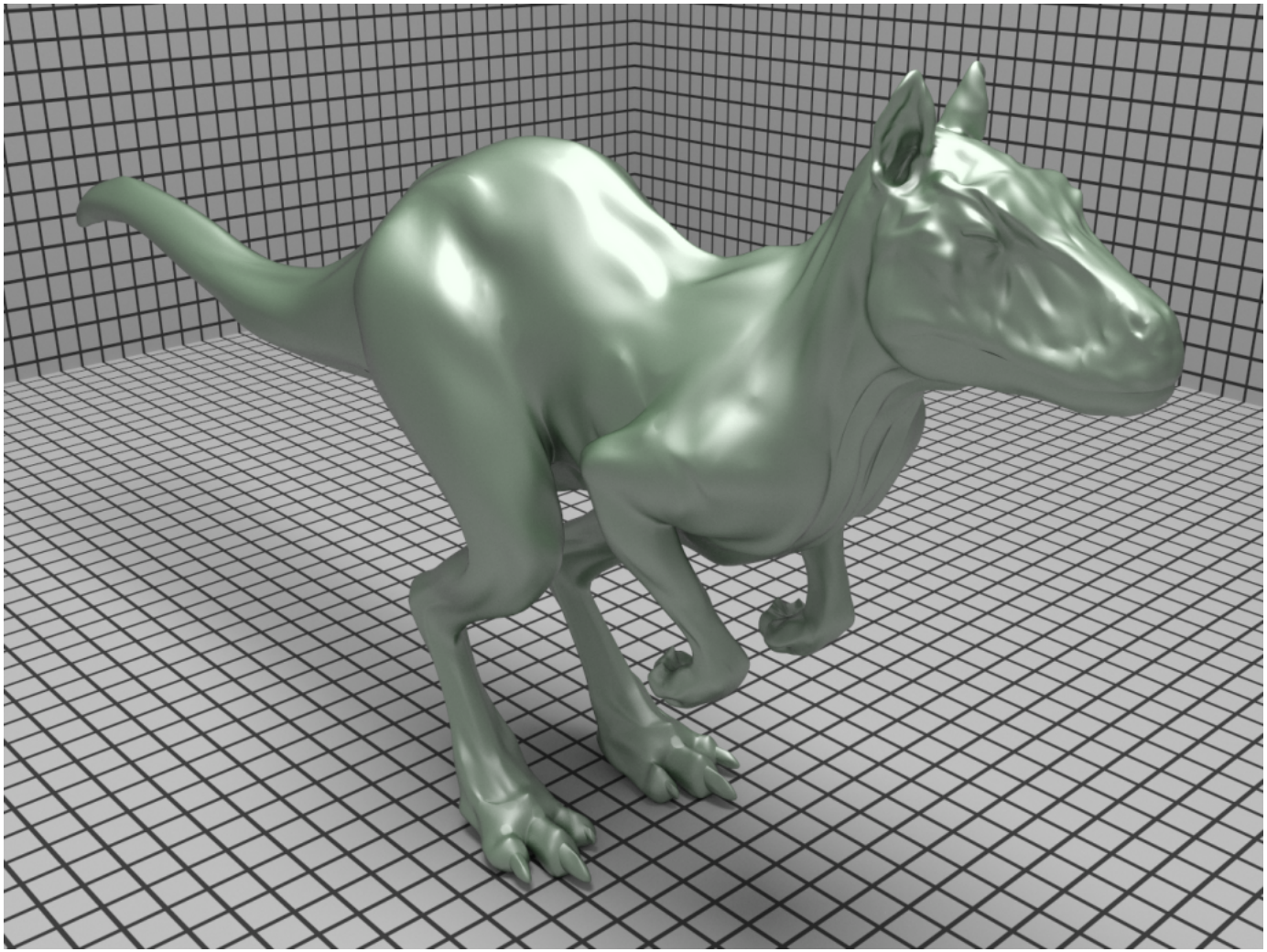
Microfacet reflection





[Walter et al.]

Microfacet reflection



[Pharr et al./pbrt]

Microfacet transmission

- Derived from previous ideas while correctly accounting for change of variable when moving below the surface
  - Here we write equations using the relative index of refraction which is eta on entering and 1/eta on exiting

$$f_t(\mathbf{i},\mathbf{o},\mathbf{n}) = \frac{|\mathbf{h}\cdot\mathbf{i}|\ |\mathbf{h}\cdot\mathbf{o}|}{|\mathbf{n}\cdot\mathbf{i}|\ |\mathbf{n}\cdot\mathbf{o}|}\frac{(1-F(\mathbf{o},\mathbf{h}))D(\mathbf{h},\mathbf{n})G(\mathbf{i},\mathbf{o},\mathbf{n})}{(\eta(\mathbf{h}\cdot\mathbf{i})+(\mathbf{h}\cdot\mathbf{o}))^2}$$

$$\mathbf{h} = \frac{-\eta\mathbf{i}+\mathbf{o}}{|-\eta\mathbf{i}+\mathbf{o}|}\mathrm{sign}(\mathbf{n}\cdot\mathbf{o})$$

Generic materials

- These scattering behaviors can be combined together to *fit* data from real-world surfaces by using weighted sums of lobes

$$f = \sum_i k_i f_i$$

- Polished/rough glass: combine reflection and transmission terms
- Plastics: weighted sum of diffuse and specular reflections
  - Accounting for Fresnel still an open problem here, but we can use a simple solution that conserve energy

$$f_{plastic} = k_d(1 - F(\mathbf{n} \cdot \mathbf{o}))f_d + k_s f_s$$

Sampling BSDF-cosine

- When picking directions for hemispherical sampling, we want to sample a function that is close to the integrand
- Often, best we can do is to sample the BSDF scaled by the cosine
- For mirror reflection and transmission, we only have one direction
- For diffuse reflection, we use the cosine sampling presented previously
- For convenience, we can rewrite that sampling in spherical coordinates

$$\mathbf{i}^l = (\phi, \theta) = (2\pi r_1, \mathrm{acos}(r_2))$$

$$p(\mathbf{i}^l) = \frac{\mathbf{i}^l_z}{\pi}$$

Sampling BSDF-cosine

- For microfacet BRDFs, we sample the *D* term since this is dominant
- We proceed by picking a half-vector **h** according to *D* as

$$\mathbf{h}_p^l = \left(2\pi r_1, \operatorname{acos}\left(r_2^{\alpha/2}\right)\right)$$

$$\mathbf{h}_b^l = \left(2\pi r_1, \operatorname{atan}\left(\sqrt{-\alpha^2 \log(1-r_2)}\right)\right)$$

$$\mathbf{h}_g^l = \left(2\pi r_1, \operatorname{atan}\left(\frac{\alpha\sqrt{r_2}}{\sqrt{1-r_2}}\right)\right)$$

$$p(\mathbf{h}) = D(\mathbf{h})|\mathbf{n}\cdot\mathbf{h}|$$

Sampling BSDF-cosine

- For reflection, we derive the incoming angle from the half-vector and correct the pdf for the change of variable as

$$\mathbf{i} = -\mathbf{o} + (\mathbf{h} \cdot \mathbf{o})\mathbf{h} \qquad p(\mathbf{i}) = \frac{D(\mathbf{h}, \mathbf{n})|\mathbf{n} \cdot \mathbf{h}|}{4|\mathbf{h} \cdot \mathbf{i}|}$$

- For refraction, the incoming angle and pdf are written as

$$\mathbf{i} = -\frac{1}{\eta}\mathbf{o} + \left[\frac{1}{\eta}(\mathbf{h} \cdot \mathbf{o}) - \sqrt{1 + \frac{1}{\eta^2}(|\mathbf{h} \cdot \mathbf{o}|^2 - 1)}\right]\mathbf{h}$$

$$p(\mathbf{i}) = \frac{D(\mathbf{h}, \mathbf{n})|\mathbf{n} \cdot \mathbf{h}|}{4|\mathbf{h} \cdot \mathbf{i}|} \frac{|\mathbf{h} \cdot \mathbf{o}|}{(\eta(\mathbf{h} \cdot \mathbf{i}) + (\mathbf{h} \cdot \mathbf{o}))^2}$$

Sampling BSDF-cosine

- For BSDF that are sums of lobes, we use multiple importance sampling
- We pick the lobe at random where lobes treated as a discreet distribution, with weights set as the maximum of the lobe weight
- We then sample that lobe
- The final PDF is the sum of PDFs

$$f = \sum_i k_i f_i \qquad w_i = \frac{\max(k_i)}{\sum_i \max(k_i)}$$

$$\mathbf{i} = \mathbf{i}_i \text{ if } \sum_j^i w_j > r \qquad p(\mathbf{i}) = \sum_i p_i(\mathbf{i})$$

Diffuse reflection

- For diffuse reflection, the implementation is straightforward
- We always need to check in which hemisphere we are

```
vec3f eval_diffuse(vec3f n, vec3f o, vec3f i) {
  if (dot(n, i) <= 0 || dot(n, o) <= 0) return vec3f{0};
  return vec3f{1} / pi * dot(n, i);
}

vec3f sample_diffuse(vec3f n, vec3f o, vec2f rn) {
  if (dot(n, o) <= 0) return vec3f{0};
  return sample_hemisphere_cos(n, rn);
}

float sample_diffuse_pdf(vec3f n, vec3f o, vec3 i) {
  if (dot(n, i) <= 0 || dot(n, o) <= 0) return 0;
  return sample_hemisphere_cos_pdf(n, i);
}
```


Diffuse reflection

- We base the implementation on hemispherical sampling functions

```
vec3f sample_hemisphere_cos(vec3f n, vec2f ruv) {
  auto il = vec3f{sqrt(1 - ruv.y*ruv.y) * cos(2*pi*ruv.x),
                  sqrt(1 - ruv.y*ruv.y) * sin(2*pi*ruv.x),
                  sqrt(ruv.y)};
  return basis_framez(n) * il;
}

float sample_hemisphere_cos_pdf(vec3f n, vec3f i) {
  auto cosw = dot(n, d);
  return (cosw <= 0) ? 0 : cosw / pi;
}
```


Microfacet reflection

- For micro facet reflections, our implementation follows the math formulation precisely

```
vec3f eval_specular(float ior, float a,
    vec3f n, vec3f o, vec3f i) {
  if (dot(n, i) <= 0 || dot(n, o) <= 0) return vec3f{0};
  auto h = normalize(incoming + outgoing);
  auto F = fresnel_dielectric(ior, h, i);
  auto D = microfacet_distribution(a, n, h);
  auto G = microfacet_shadowing(a, n, h, o, i);
  return F * D * G / (4 * dot(n, o) * dot(n, i)) *
         dot(n, i);
}
```


Microfacet reflection

- For micro facet reflections, our implementation follows the math formulation precisely

```
vec3f sample_specular(float ior, float a,
    vec3f n, vec3f o, vec2f rn) {
  if (dot(n, o) <= 0) return vec3f{0};
  auto h = sample_microfacet(a, n, rn);
  return reflect(o, h);
}

float sample_specular_pdf(float ior, float a,
    vec3f n, vec3f o, vec3f i) {
  if (dot(n, i) <= 0 || dot(n, o) <= 0) return 0;
  auto h = normalize(o + i);
  return sample_microfacet_pdf(a, n, h) /
         (4 * abs(dot(o, h)));
}
```


Microfacet reflection

- Metals and dielectric reflections differ only on their fresnel term

```
vec3f eval_metal(vec3f eta, vec3f etak, float a,
    vec3f n, vec3f o, vec3f i) {
  if (dot(n, i) <= 0 || dot(n, o) <= 0) return vec3f{0};
  auto h = normalize(incoming + outgoing);
  auto F = fresnel_conductor(eta, etak, h, i);
  auto D = microfacet_distribution(a, n, h);
  auto G = microfacet_shadowing(a, n, h, o, i);
  return F * D * G / (4 * dot(n, o) * dot(n, i)) *
         dot(n, i);
}
vec3f sample_metal(vec3f eta, vec3f etak, float a,
    vec3f n, vec3f o, vec2f rn) {<same as sample_specular>}
float sample_specular_pdf(vec3f eta, vec3f etak, float a,
    vec3f n, vec3f o, vec3f i) {<sample_specular_pdf>}
```


Microfacet reflection

- The micro facet functions are just an implementation of the formulae

```
float microfacet_distribution(float a, vec3f n, vec3f h) {
  auto c = dot(n, h); if (c <= 0) return 0;
  auto a2 = a * a; auto c2 = c * c;
  return a2 / (pi * (c2 * a2 + 1 - c2) * (c2 * a2 + 1 - c2));
}

float microfacet_shadowing1(float a, vec3f n,vec3f h, vec3f d) {
  auto c  = dot(n, d); auto ch = dot(h, d);
  if (c * ch <= 0) return 0;
  auto a2 = a * a; auto c2 = c * c;
  return 2 * abs(c) / (abs(c) + sqrt(c2 - a2 * c2 + a2));
}

float microfacet_shadowing(float a, vec3f n,
    vec3f h, vec3f o, vec3f i) {
  return microfacet_shadowing1(a, n, h, o) *
         microfacet_shadowing1(a, n, h, i);
}
```

Microfacet reflection

- The microfacet functions are just an implementation of the formulae

```
vec3f sample_microfacet(float a, vec3f n, vec2f rn) {
  auto phi   = 2 * pi * rn.x;
  auto theta = atan(roughness * sqrt(rn.y / (1 - rn.y)));
  auto lh = vec3f{
      cos(phi) * sin(theta), sin(phi) * sin(theta), cos(theta)};
  return transform_direction(basis_fromz(n), lh);
}

float sample_microfacet_pdf(
    float a, vec3f n, const vec3f h) {
  auto c = dot(n, h); if (c < 0) return 0;
  return microfacet_distribution(a, n, h) * c;
}
```


Generic BRDF

- As an example of a generic BRDF, we can write a BRDF as a weighted sum of predefined lobes; here we specify the lobes explicitly, but we could have also used an array of lobes

```
struct brdf {
  vec3f diffuse      = {0, 0, 0}; // diffuse weight
  vec3f specular     = {0, 0, 0}; // specular weight
  vec3f metal        = {0, 0, 0}; // metal weight
  vec3f transmission = {0, 0, 0}; // transmission weight
  float roughness    = 0;         // surface roughness
  float ior          = 1;         // specular ior
  vec3f meta         = {0, 0, 0}; // metal complex ior
  vec3f metak        = {0, 0, 0};
  float diffuse_pdf      = 0; // diffuse pdf
  float specular_pdf     = 0; // specular pdf
  float metal_pdf        = 0; // metal pdf
  float transmission_pdf = 0; // transmission pdf
};
```


Generic BRDF

- The evaluation functions just sum each lobe contribution

```
vec3f eval_brdfcos(brdf brdf, vec3f n, vec3f o, vec3f i) {
 auto brdfcos = zero3f;
 if (brdf.diffuse)
   brdfcos += brdf.diffuse * eval_diffuse(n, o, i);
 if (brdf.specular)
   brdfcos += brdf.specular * eval_specular(brdf.ior,
                brdf.roughness, n, o, i);
 if (brdf.metal)
   brdfcos += brdf.metal * eval_metal(brdf.meta, brdf.metak,
                brdf.roughness, n, o, i);
 if (brdf.transmission)
   brdfcos += brdf.transmission * eval_transmission(brdf.ior,
                brdf.roughness, n, o, i);
 return brdfcos;
}
```


Generic BRDF

- The sample function picks a lobe by maintaining a running sum over the lobe pdfs

```
vec3f sample_brdfcos(brdf brdf, vec3f n, vec3f o,
  float rnl, vec2f rn) {
  auto cdf = 0.0f;
  if (brdf.diffuse_pdf) {
    cdf += brdf.diffuse_pdf;
    if (rnl < cdf) return sample_diffuse(n, o, rn); }
  if (brdf.specular_pdf) {
    cdf += brdf.specular_pdf;
    if (rnl < cdf) return sample_specular(brdf.r, n, o, rn); }
  if (brdf.metal_pdf) {
    cdf += brdf.metal_pdf;
    if (rnl < cdf) return sample_metal(brdf.r, n, o, rn); }
  if (brdf.transmission_pdf) {
    cdf += brdf.transmission_pdf;
    if (rnl < cdf)return sample_transmission(brdf.r, n, o, rn);}
  return zero3f;
}
```

Generic BRDF

- The sample pdf is computed as the sum of each lobe pdf

```
float sample_brdfcos_pdf(brdf brdf, vec3f n, vec3f o, vec3f i) {
  auto pdf = 0.0f;
  if (brdf.diffuse_pdf) {
    pdf += brdf.diffuse_pdf*sample_diffuse_pdf(n,o,i); }
  if (brdf.specular_pdf) {
    pdf += brdf.specular_pdf*sample_specular_pdf(brdf.r,n,o,i);}
  if (brdf.metal_pdf) {
    pdf += brdf.metal_pdf*sample_metal_pdf(brdf.r,n,o,i);}
  if (brdf.transmission_pdf) {
    pdf += brdf.transmission_pdf*
              sample_transmission_pdf(brdf.r,n,o,i);}
  return pdf;
}
```

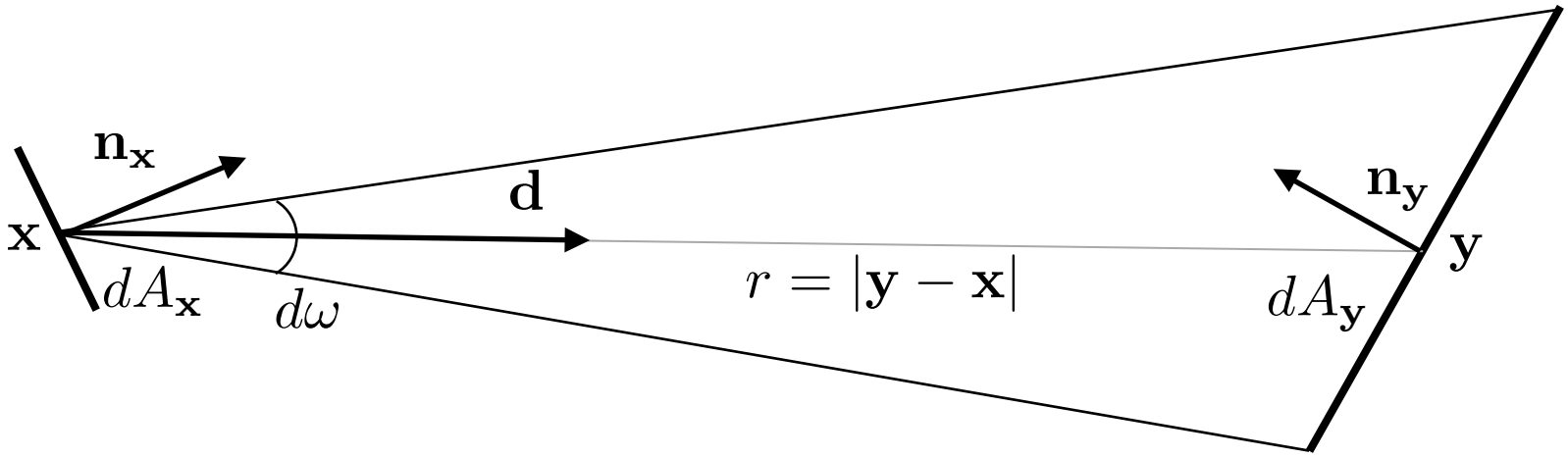

Area Formulation

Radiance between differential areas

- Radiance between surfaces can be computed considering the solid angle subtended by one surface w.r.t. the other

$$L(\mathbf{x}, \mathbf{d}_{\mathbf{x}\to\mathbf{y}}) = \frac{d^2P}{dA_{\mathbf{x}}(\mathbf{d}\cdot\mathbf{n}_{\mathbf{x}})d\omega_{\mathbf{d}}} = \frac{d^2P|\mathbf{y}-\mathbf{x}|^2}{dA_{\mathbf{x}}(\mathbf{d}\cdot\mathbf{n}_{\mathbf{x}})dA_{\mathbf{y}}(-\mathbf{d}\cdot\mathbf{n}_{\mathbf{x}})} =$$

$$\text{with } \mathbf{d}_{\mathbf{x}\to\mathbf{y}} = \frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|} \quad d\omega_{\mathbf{d}} = \frac{dA_{\mathbf{y}}(-\mathbf{d}\cdot\mathbf{n}_{\mathbf{x}})}{|\mathbf{y}-\mathbf{x}|^2}$$



Rendering equation – area form

- Angular form of the rendering equation

$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_{\mathbf{i}}$$

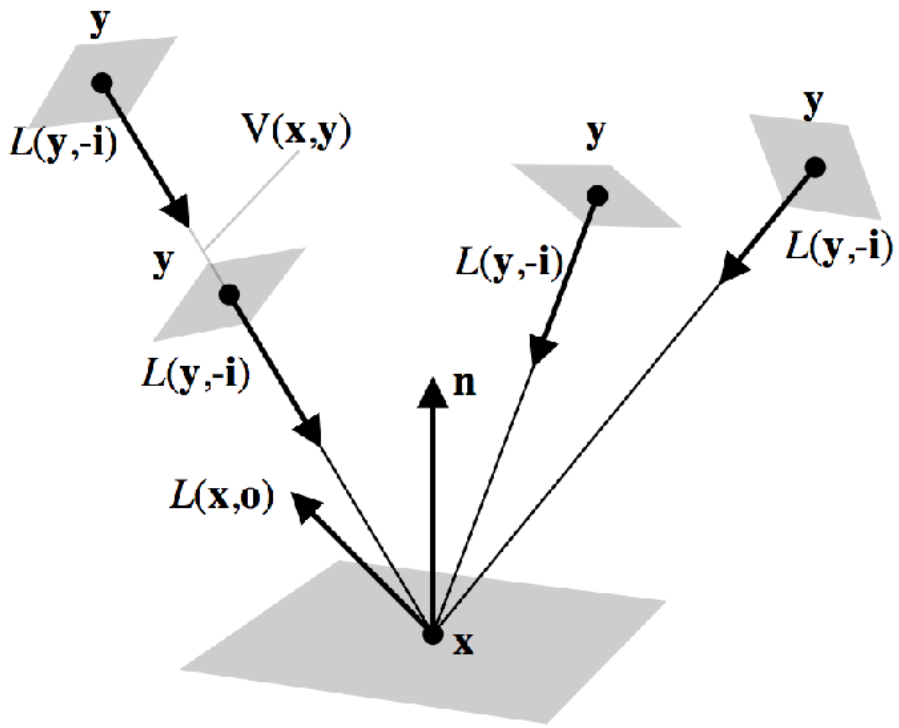
- Express the integral in terms of integrating over all possible surfaces in the scene; change the variable of integration from solid angle to area using the expression of the solid angle w.r.t. area
  - include a term *V* to decide whether two points are visible

$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{\mathbf{y}\in S} L(\mathbf{y},-\mathbf{i}) V(\mathbf{x},\mathbf{y}) f(\mathbf{x},\mathbf{i},\mathbf{o}) G(\mathbf{x},\mathbf{y}) dA_{\mathbf{y}}$$

$$\text{with } \mathbf{i} = \frac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|} \quad d\omega_{\mathbf{i}} = \frac{dA_{\mathbf{y}}|-\mathbf{i}\cdot\mathbf{n_y}|}{|\mathbf{y}-\mathbf{x}|^2} \quad G(\mathbf{x},\mathbf{y}) = \frac{|\mathbf{i}\cdot\mathbf{n_x}||-\mathbf{i}\cdot\mathbf{n_y}|}{|\mathbf{y}-\mathbf{x}|^2}$$

Rendering equation – area form

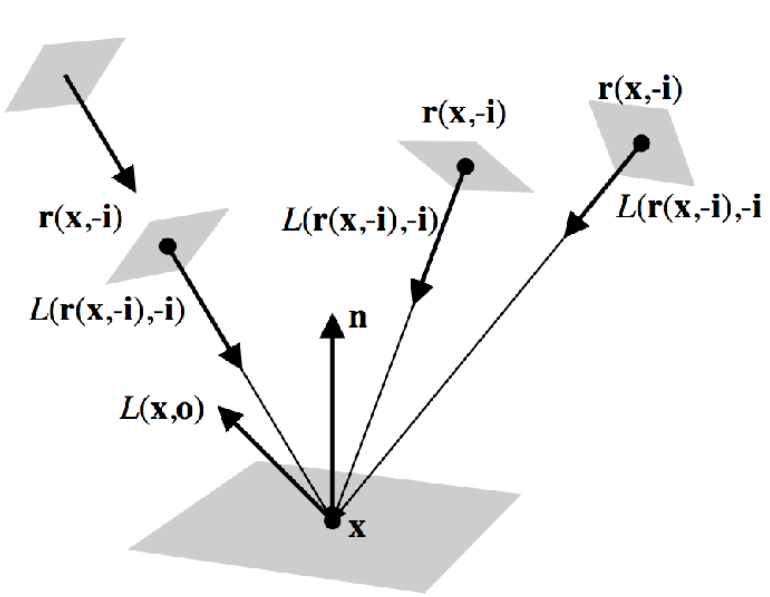
$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{\mathbf{y}\in S} L(\mathbf{y},-\mathbf{i})V(\mathbf{x},\mathbf{y})f(\mathbf{x},\mathbf{i},\mathbf{o})G(\mathbf{x},\mathbf{y})dA_{\mathbf{y}}$$

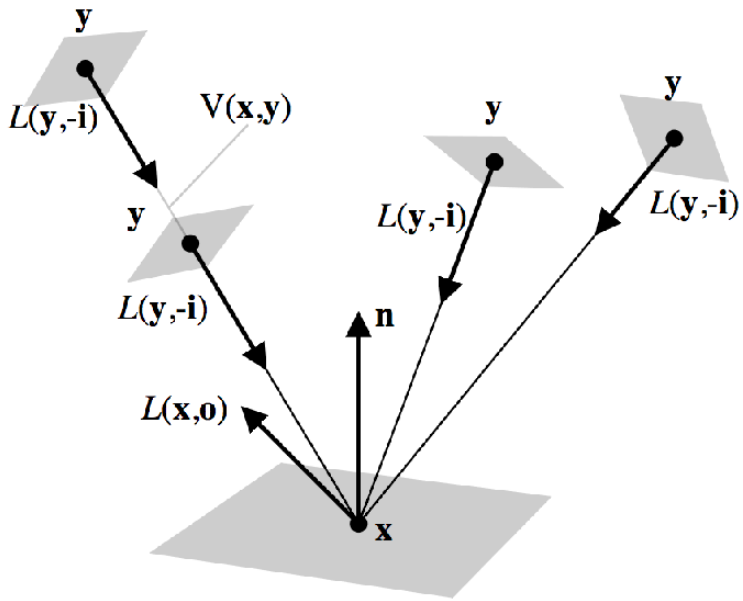


Angular vs area form

$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{H^2_{\mathbf{x}}} L(\mathbf{r}(\mathbf{x},\mathbf{i}),-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) |\mathbf{n_x}\cdot\mathbf{i}| d\omega_{\mathbf{i}}$$

$$L(\mathbf{x},\mathbf{o}) = L_e(\mathbf{x},\mathbf{o}) + \int_{\mathbf{y}\in S} L(\mathbf{y},-\mathbf{i}) V(\mathbf{x},\mathbf{y}) f(\mathbf{x},\mathbf{i},\mathbf{o}) G(\mathbf{x},\mathbf{y}) dA_{\mathbf{y}}$$





Angular vs area form

- **Angular form**: integrate over incoming angle
  - use raytracing to determine the visible surface from where the light comes from
- **Area form**: integrate over all scene's surfaces
  - use raytracing to determine whether a given point is visible from another, i.e. whether a point contributes to the integral (shadow rays)

Direct illumination – area

- Consider the same solution but now with the area formulation

$$L_d(\mathbf{x}, \mathbf{o}) = \int_{\mathbf{y} \in A_{light}} L_e(\mathbf{y}, -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) V(\mathbf{x}, \mathbf{y}) G(\mathbf{x}, \mathbf{y}) dA_{\mathbf{y}}$$

- The estimator for this formulation is

$$L_d(\mathbf{x}, \mathbf{o}) \approx \frac{L_e(\mathbf{y}_i, -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) V(\mathbf{x}, \mathbf{y}_i) G(\mathbf{x}, \mathbf{y}_i)}{p(\mathbf{y}_i)}$$

- To evaluate it, we have to pick points uniformly over the light surface

Sampling multiple lights

- To handle multiple lights, we can estimate their contributions independently since each light integral is independent

$$L_d(\mathbf{x},\mathbf{o}) = \int_{\mathbf{y}\in A_{lights}} L_e(\mathbf{y},-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) V(\mathbf{x},\mathbf{y}) G(\mathbf{x},\mathbf{y}) dA_{\mathbf{y}} =$$

$$= \sum_l^{n_l} \int_{\mathbf{y}\in A_l} L_e(\mathbf{y},-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) V(\mathbf{x},\mathbf{y}) G(\mathbf{x},\mathbf{y}) dA_{\mathbf{y}}$$

$$L_d(\mathbf{x},\mathbf{o}) \approx \sum_l^{n_l} \frac{A_{l_i}}{N} \sum_i A_{l_i} L_e(\mathbf{y}_i,-\mathbf{i}) f(\mathbf{x},\mathbf{i},\mathbf{o}) V(\mathbf{x},\mathbf{y}_i) G(\mathbf{x},\mathbf{y}_i)$$

Sampling multiple lights

- As the number of lights increases, this formulation becomes inefficient
- The alternative is to pick a *light at random for each sample*
- Here we are sampling from a uniform *discreet* distribution

$$L_d(\mathbf{x}, \mathbf{o}) = \sum_l^{n_l} \int_{\mathbf{y} \in A_l} L_e(\mathbf{y}, -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) V(\mathbf{x}, \mathbf{y}) G(\mathbf{x}, \mathbf{y}) dA_{\mathbf{y}}$$

$$L_d(\mathbf{x}, \mathbf{o}) \approx A_{l_i} L_e(\mathbf{y}_i, -\mathbf{i}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) V(\mathbf{x}, \mathbf{y}_i) G(\mathbf{x}, \mathbf{y}_i)$$

$$l_i = \text{floor}(r \cdot n_l) \quad l \sim p(l) = \frac{1}{n_l}$$

Proof of Recursive Solution

Rendering eq. - operator form

- We can think of reflection as transport operator *T* that takes as input the radiance function and return a new radiance function

$$T : L(\mathbf{x}, \mathbf{o}) \rightarrow L_r(\mathbf{x}, \mathbf{o})$$

$$(TL)(\mathbf{x}, \mathbf{o}) = \int_{\mathbf{y} \in S} L(\mathbf{y}, -\mathbf{i}) V(\mathbf{x}, \mathbf{y}) f(\mathbf{x}, \mathbf{i}, \mathbf{o}) G(\mathbf{x}, \mathbf{y}) dA_{\mathbf{y}}$$

- Rendering equation

$$L(\mathbf{x}, \mathbf{o}) = L_e(\mathbf{x}, \mathbf{o}) + (TL)(\mathbf{x}, \mathbf{o})$$

- In shorthand notation

$$L = L_e + TL$$

Rendering eq. - operator form

- Formal solution of the rendering equation is

$$L = L_e + TL \rightarrow (I - T)L = L_e \rightarrow L = (I - T)^{-1} L_e$$

- Unusable in practice since we cannot compute the inverse
- Solve by recursive substitution

$$L = L_e + TL = L_e + T(L_e + TL) = L_e + TL_e + T^2 L =$$
$$= L_e + TL_e + T^2(L_e + TL) = L_e + TL_e + T^2 L_e + T^3 L = ...$$

- This leads to the solution written as a Neumann series

$$L = \sum_{i=0}^{\infty} T^i L_e$$

Rendering eq. - operator form

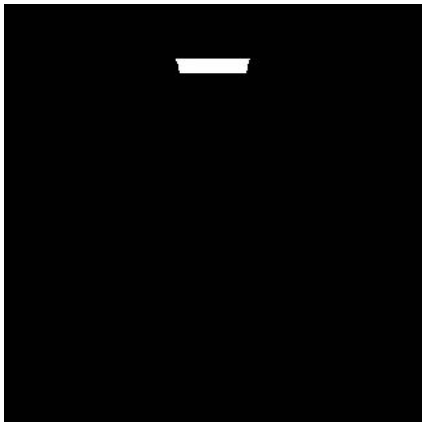
- We can show that the Neumann series is the right solution by equating it in the formal solution

$$L = \sum_{i=0}^{\infty} T^i L_e \qquad L = (I - T)^{-1} L_e$$

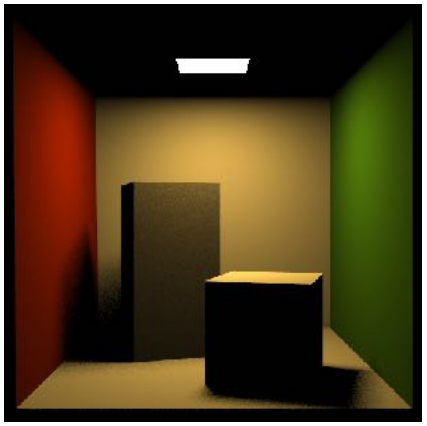
- Since these equations are true for every *Le*, we have that

$$\begin{aligned}
(I - T)^{-1} &= \sum_{i=0}^{\infty} T^i L_e \\
(I - T)(I - T)^{-1} &= (I - T) \sum_{I=0}^{\infty} T^i L_e \\
I &= \sum_{i=0}^{\infty} T^i L_e - \sum_{i=1}^{\infty} T^i L_e \\
I &= I
\end{aligned}$$

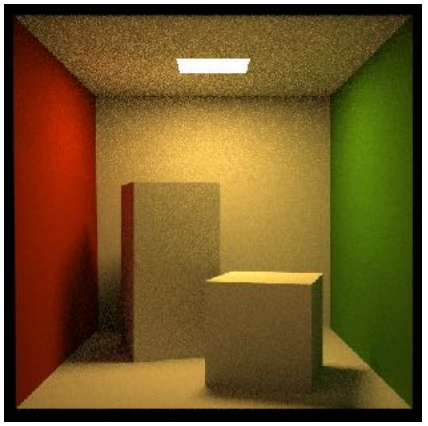
Rendering eq. - operator form



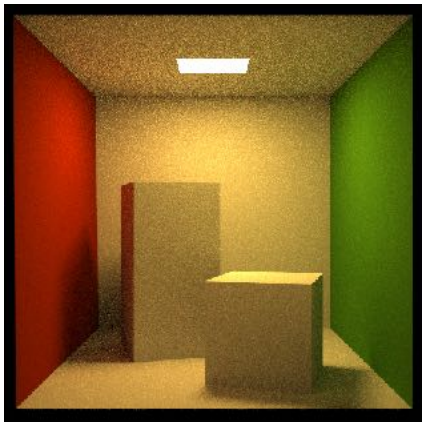
L_e



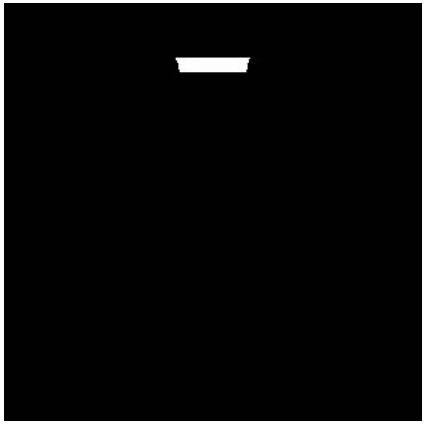
$L_e + TL_e$



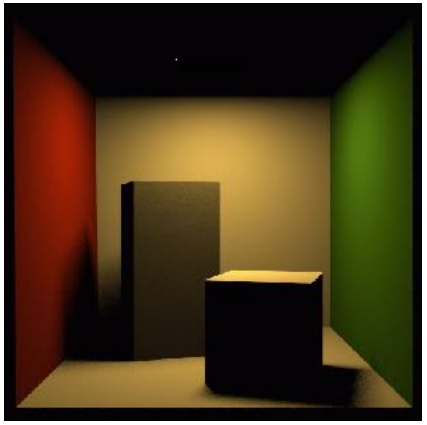
$L_e + TL_e + T^2L_e$



$L_e + ... + T^3L_e$

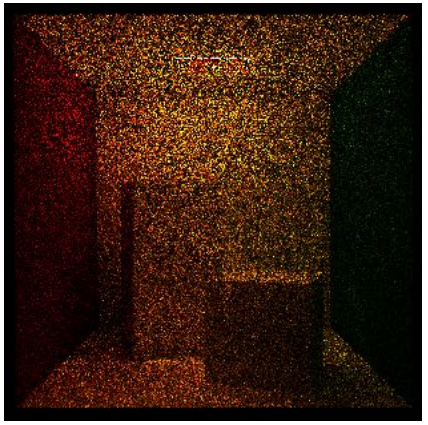


L_e



TL_e

T^2L_e (16x)



T^3L_e (16x)

[Dutrè, Bala]

Properties of TL

- Intuition: *TL* bounces light once
- From the conservation of energy, we know that the norm of *TL* will be less or equal to the one of *L*
  - since light is either reflected of absorbed
  - similar to energy conservation for the BSDF
- The higher the BSDFs of the scene the more bounces we need since energy decreases less at each bounce

Recursive solution

- We know now how to recursively solve the rendering equation

$$L = \sum_{i=0}^{\infty} T^i L_e$$

- Let us rewrite this in terms of the original integrals

$$\begin{aligned} L(\mathbf{x}_0, \mathbf{o}_0) = L_e(\mathbf{x}_0, \mathbf{o}_0) + \\ + \int_{\mathbf{x}_1} L_e(\mathbf{x}_1, \mathbf{o}_1) V(\mathbf{x}_0, \mathbf{x}_1) f(\mathbf{x}_0, \mathbf{i}_0, \mathbf{o}_0) G(\mathbf{x}_0, \mathbf{x}_1) dA_{\mathbf{x}_1} + \\ + \int_{\mathbf{x}_1} \left(\int_{\mathbf{x}_2} L_e(\mathbf{x}_2, \mathbf{o}_2) V(\mathbf{x}_1, \mathbf{x}_2) f(\mathbf{x}_1, \mathbf{i}_1, \mathbf{o}_1) G(\mathbf{x}_1, \mathbf{x}_2) dA_{\mathbf{x}_2} \right) \cdot \\ \cdot V(\mathbf{x}_0, \mathbf{x}_1) f(\mathbf{x}_0, \mathbf{i}_0, \mathbf{o}_0) G(\mathbf{x}_1, \mathbf{x}_2) dA_{\mathbf{x}_2} + \ldots \end{aligned}$$

Recursive solution

- We can write the recursive solution by aggregating the product of all visibility, BRDF and geometric terms in a single expression called *path throughput*

$$\mathrm{Tr}(\mathbf{x}_0, ..., \mathbf{x}_i) = \prod_{j=0}^{i-1} V(\mathbf{x}_j, \mathbf{x}_j + 1) f(\mathbf{x}_j, \mathbf{i}_j, \mathbf{o}_j) G(\mathbf{x}_j, \mathbf{x}_{j+1})$$

$$L(\mathbf{x}, \mathbf{o}) = \sum_{i=0}^{\infty} \int_{\mathbf{x}_1} ... \int_{\mathbf{x}_i} L_e(\mathbf{x}_i, \mathbf{o}_i) \mathrm{Tr}(\mathbf{x}_0, ..., \mathbf{x}_i) dA_{\mathbf{x}_1} ... dA_{\mathbf{x}_i}$$

Rendering eq. – path integral form

- From the previous formulation, we can get some intuition on yet another form of the rendering equation called *path formulation*
- The basic idea is to think about integrating not recursively over position but in *space of all possible paths* that the light can travel into
- A light path is a sequence of points of *any length*

$$\mathbf{p} = \{\mathbf{x}_0, ..., \mathbf{x}_i\}$$

- We can rewrite the previous solution as the integral of the radiance of all possible path that end at the camera

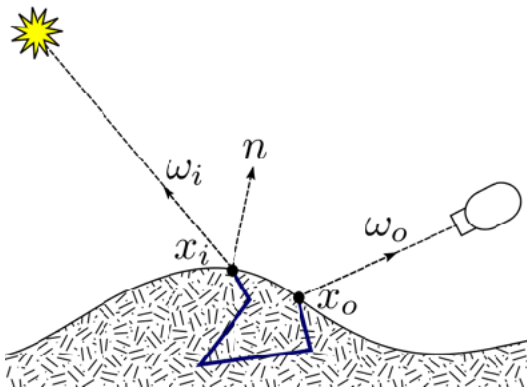
$$L(\mathbf{x}, \mathbf{o}) = \int_{\mathbf{p} \in \mathcal{P}} L_e(\mathbf{p}_{end}) \mathrm{Tr}(\mathbf{p}) d\mu_{\mathbf{p}}$$

Subsurface Scattering

Subsurface scattering: BSSRDF

- BSSRDF [$m^{-2}sr^{-1}$]: *bidirectional scattering surface-reflectance distr. func.*
- Defined as the ratio of the differential reflected radiance over the differential incident flux
- Depends on incoming and outgoing directions, incoming and outgoing positions, and wavelength of light
- Main different w.r.t. BSDF: photons leave at different points then they enter, so consider two both locations in the equation

$$s(\mathbf{x_i}, \mathbf{x_o}, \mathbf{i}, \mathbf{o}) = \frac{dL_r(\mathbf{x_o}, \mathbf{o})}{dP_i(\mathbf{x_i}, \mathbf{i})} = \frac{dL_r(\mathbf{x_o}, \mathbf{o})}{L_i(\mathbf{x_i}, \mathbf{i})|\mathbf{i} \cdot \mathbf{n_i}| d\omega_\mathbf{i} dA_{\mathbf{x_i}}}$$



Subsurface scattering: BSSRDF

- Reflected radiance can be obtained by integrating the above equation over all points on the surface and for each incoming direction around each point

$$L_r(\mathbf{x_o}, \mathbf{o}) = \int_{\mathbf{x_i} \in S} \int_{\mathbf{i} \in H^2} L_i(\mathbf{x_i}, \mathbf{i}) s(\mathbf{x_i}, \mathbf{x_o}, \mathbf{i}, \mathbf{o}) |\mathbf{i} \cdot \mathbf{n_i}| d\omega_\mathbf{i} dA_{\mathbf{x_i}}$$

Properties of the BSSRDF

- Same properties of BSSRDF
- Positivity: since photons cannot be "negatively" reflected

$$s(\mathbf{y}, \mathbf{x}, \mathbf{i}, \mathbf{o}) \geq 0$$

- Helmotz reciprocity: can invert light paths

$$s(\mathbf{y}, \mathbf{x}, \mathbf{i}, \mathbf{o}) = s(\mathbf{x}, \mathbf{y}, \mathbf{o}, \mathbf{i})$$

- Energy conservation: all photons that comes in are either reflected or absorbed and no new photons is emitted

$$\int_S \int_{H^2} s(\mathbf{y}, \mathbf{x}, \mathbf{i}, \mathbf{o}) |\mathbf{i} \cdot \mathbf{o}| d\omega_{\mathbf{i}} dA_{\mathbf{y}} \leq 1$$

Rendering with BSSRDF

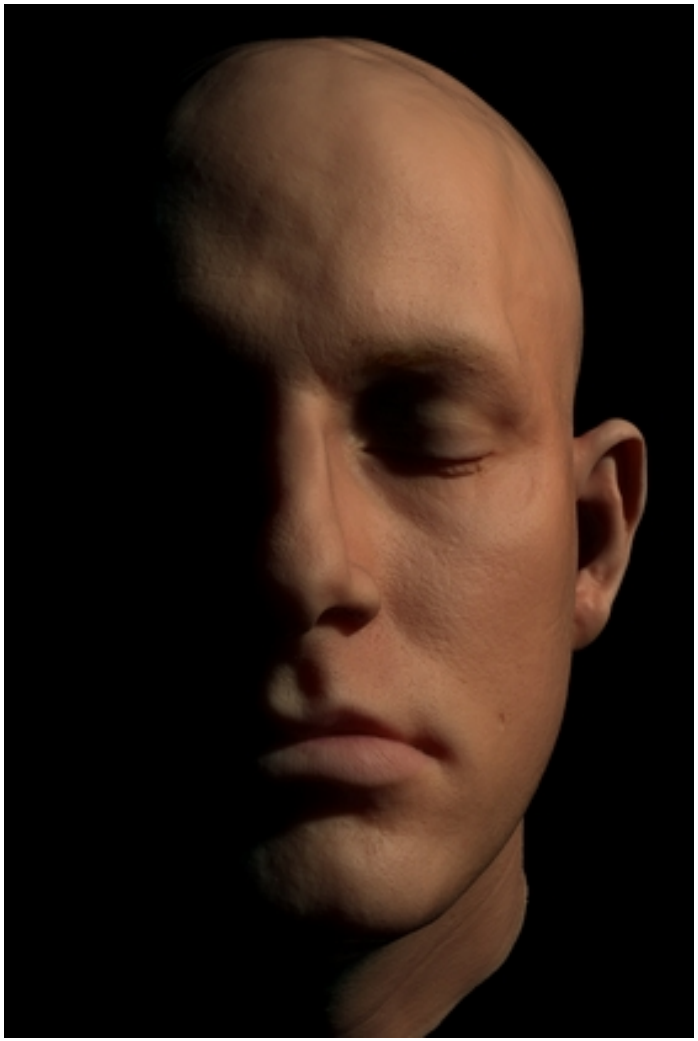
BRDF

BSSRDF

[Jensen et al]

Rendering with BSSRDF

[Conner et al]



[Jensen et al]

Volumetric Rendering

Volume scattering processes

- Emission: photons are emitted by the medium

[https://wikipedia.org]

Volume scattering processes

- Absorption: photons are absorbed by the medium

[https://common.wikipedia.org]

Volume scattering processes

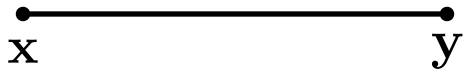
- Scattering: photons change direction in the medium

[https://coclouds.com]

Volume scattering processes

- Surface-only scattering: radiance is constant along a ray

$$L_i(\mathbf{x}, \mathbf{i}) = L_o(\mathbf{y}, -\mathbf{i})$$

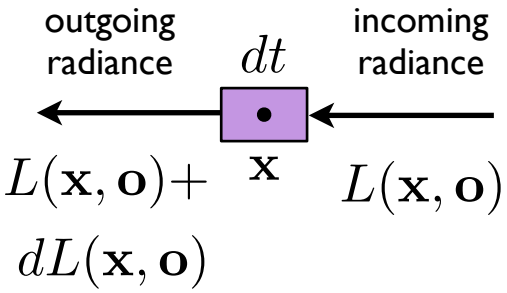


- Volume scattering: describe how radiance changes along a ray

$$L_i(\mathbf{x}, \mathbf{i}) \neq L_o(\mathbf{y}, -\mathbf{i})$$

Volume scattering processes

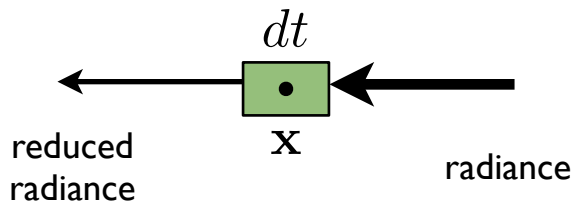
- Main idea: describe how radiance changes on a differential beam of length *dt* and area *dA* at a point *x* and oriented in direction *o*



Volume scattering processes

- Absorption: reduction of radiance due to absorption of radiance
- Absorption coefficient [m^{-1}]: describes the absorbed radiance per unit length

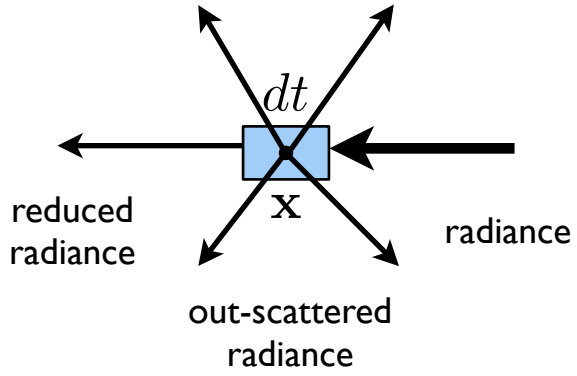
$$dL(\mathbf{x}, \mathbf{o}) = -\sigma_a(\mathbf{x})L(\mathbf{x}, \mathbf{o})dt$$



Volume scattering processes

- Out scattering: reduction of radiance due to scattering in other directions
- Scattering coefficient [m^{-1}]: describes the scattered radiance per unit length

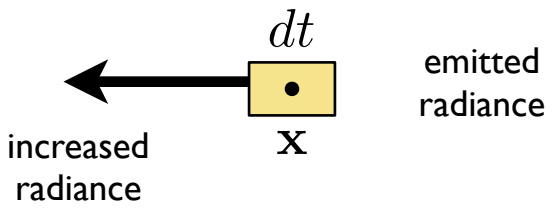
$$dL(\mathbf{x}, \mathbf{o}) = -\sigma_s(\mathbf{x})L(\mathbf{x}, \mathbf{o})dt$$



Volume scattering processes

- Emission: increase of radiance due to emission of radiance
- Absorption coefficient [m^{-1}]: describes the absorbed radiance per unit length
- Emitted radiance as before

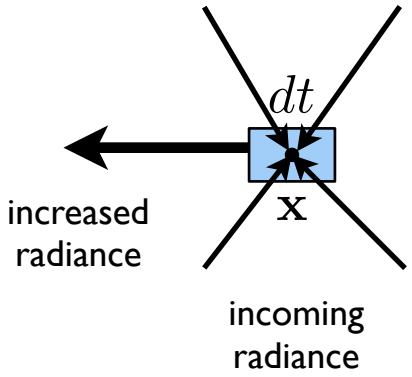
$$dL(\mathbf{x}, \mathbf{o}) = \sigma_a(\mathbf{x}) L_e(\mathbf{x}, \mathbf{o}) dt$$



Volume scattering processes

- In scattering: increase of radiance due to scattering from other directions
- Scattering coefficient [m^{-1}]: describes the scattered radiance per unit length

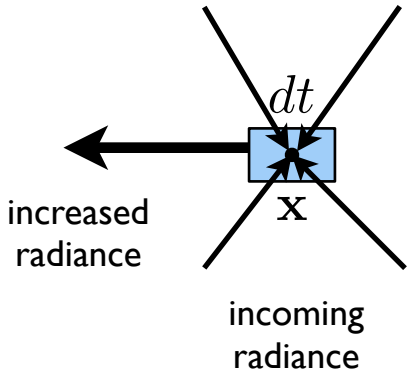
$$dL(\mathbf{x}, \mathbf{o}) = \sigma_s(\mathbf{x}) L_s(\mathbf{x}, \mathbf{o}) dt$$



Volume scattering processes

- In-scattered radiance: radiance scattered from all directions into this one, computed as the integral over the sphere of the incoming radiance weighted by the phase function
- Phase function: describes the angular distribution of scattering

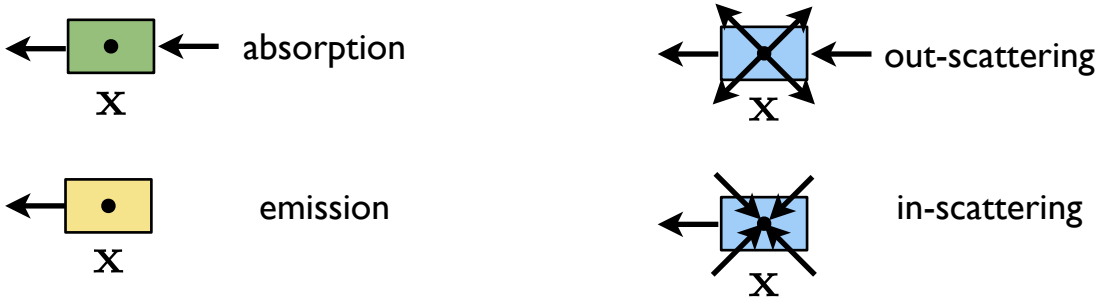
$$L_s(\mathbf{x}, \mathbf{o}) = \int_S f_p(\mathbf{x}, \mathbf{o}, \mathbf{i}) L_i(\mathbf{x}, \mathbf{i}) d\omega_{\mathbf{i}}$$



Volume scattering processes

- In summary, at a point *x*, radiance decreases due to absorption and out-scattering and increases due to emission and in-scattering

$$dL(\mathbf{x},\mathbf{o}) = -\sigma_a(\mathbf{x})L(\mathbf{x},\mathbf{o})dt \; -\sigma_s(\mathbf{x})L(\mathbf{x},\mathbf{o})dt$$
$$+\sigma_a(\mathbf{x})L_e(\mathbf{x},\mathbf{o})dt + \sigma_s(\mathbf{x})L_s(\mathbf{x},\mathbf{o})dt$$



Volume scattering processes

- For convenience, we can parametrize the previous equation
- Extinction coefficient [m^{-1}]: measures the total decrease in radiance per unit length; defined as the sum of absorption and scattering
- Albedo [unit]: measures the amount of scattered radiance independently of extinction; defined as ratio of scattering and extinction

$$dL(\mathbf{x},\mathbf{o}) = -\sigma_t(\mathbf{x})L(\mathbf{x},\mathbf{o})dt + \sigma_a(\mathbf{x})L_e(\mathbf{x},\mathbf{o})dt + \sigma_s(\mathbf{x})L_s(\mathbf{x},\mathbf{o})dt$$

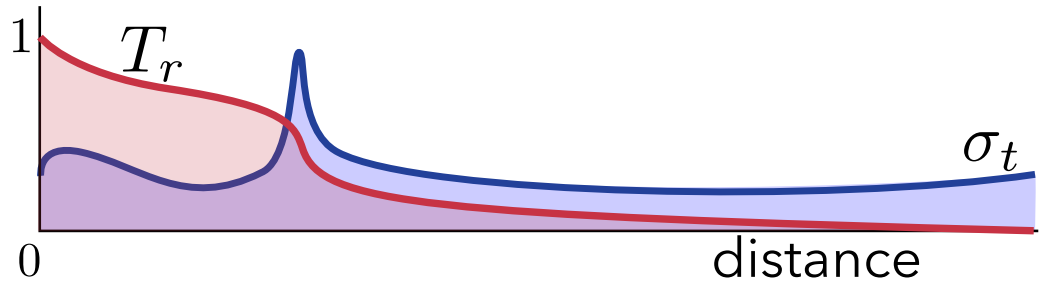
$$\sigma_t(\mathbf{x}) = \sigma_a(\mathbf{x}) + \sigma_s(\mathbf{x}) \qquad \rho_s(\mathbf{x}) = \sigma_s(\mathbf{x})/\sigma_t(\mathbf{x})$$

Volumetric transmittance

- Volumetric transmittance [unit]: decrease in radiance between 2 points
- Transmittance is the points radiances and is between 0 and 1
- We can compute by integrating the excitation terms along a ray

$$T(\mathbf{x}, \mathbf{x}_t) = e^{-\int_0^t \sigma_t(\mathbf{x}_s) ds}$$

$$\mathbf{x}_t = \mathbf{x} - t\mathbf{o}$$

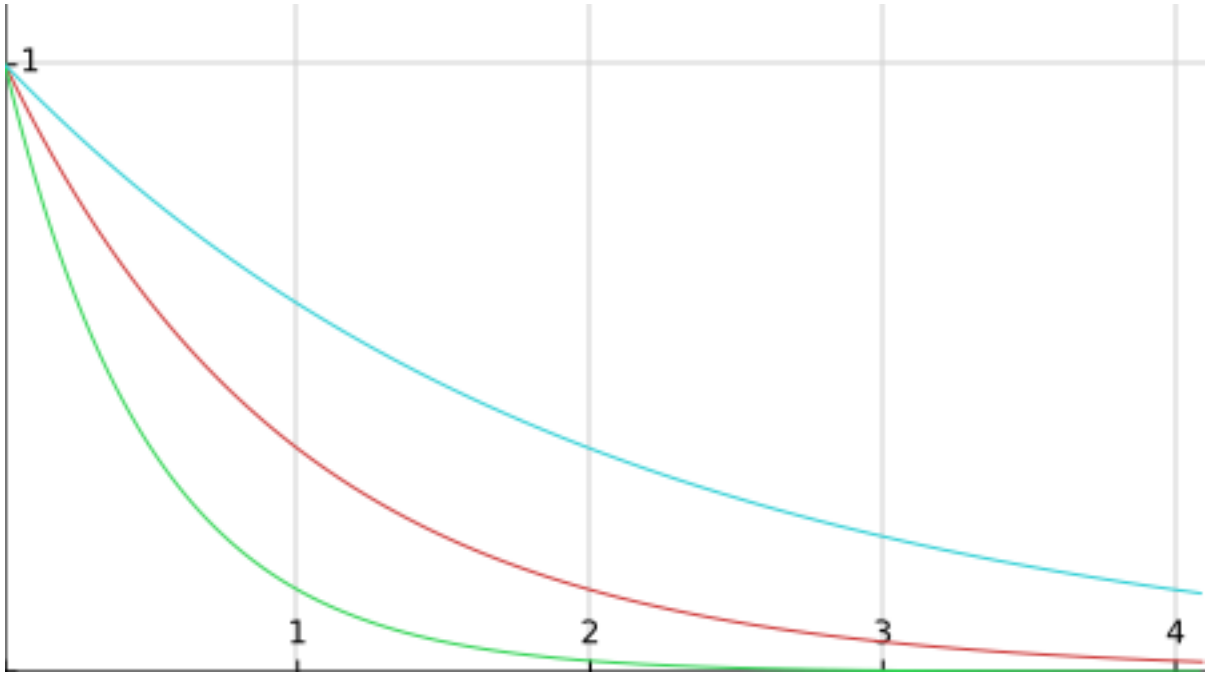


[Jaroz]

Volumetric transmittance

- For homogenous media, that has a constance extinction coefficient, we get the know simpler Beer's law

$$T(\mathbf{x}, \mathbf{x}_t) = e^{-\sigma_t t}$$



Volumetric rendering equation

- We can now derive the volumetric rendering equation by integrating along a ray from a point to another
- The derivation is non-trivial and hardly required to be able to solve the equation itself
- So we will skip the derivation and focus on under the equation itself
- To make integration easier in a path tracer, we are going to use the direction conventions introduce for surface scattering

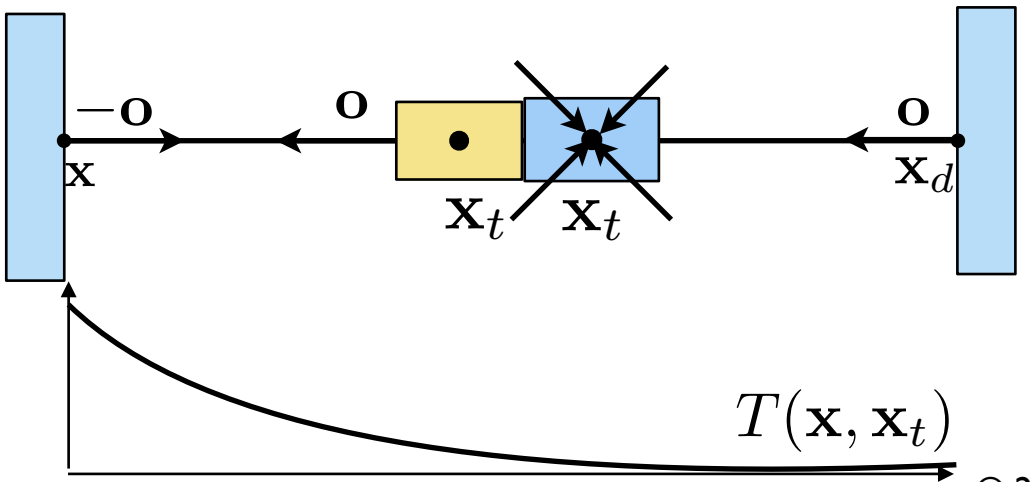
Volumetric rendering equation

- The volumetric rendering equation is the sum of three terms
  - Background radiance rescaled by transmittance
  - Emitted radiance rescaled and accumulated along the ray
  - In-scattered radiance rescaled and accumulated along the ray

$$
\begin{aligned}
L_i(\mathbf{x}, -\mathbf{o}) = & \, T(\mathbf{x}, \mathbf{x}_d) L_o(\mathbf{x}_d, \mathbf{o}) \\
& + \int_0^d T(\mathbf{x}, \mathbf{x}_t) \sigma_a(\mathbf{x}_t) L_e(\mathbf{x}_t, \mathbf{o}) dt \\
& + \int_0^d T(\mathbf{x}, \mathbf{x}_t) \sigma_s(\mathbf{x}_t) L_s(\mathbf{x}_t, \mathbf{o}) dt
\end{aligned}
$$

Volumetric rendering equation

$$L_i(\mathbf{x}, -\mathbf{o}) = T(\mathbf{x}, \mathbf{x}_d) L_o(\mathbf{x}_d, \mathbf{o}) + \int_0^d T(\mathbf{x}, \mathbf{x}_t) \sigma_a(\mathbf{x}_t) L_e(\mathbf{x}_t, \mathbf{o}) dt + \int_0^d T(\mathbf{x}, \mathbf{x}_t) \sigma_s(\mathbf{x}_t) L_s(\mathbf{x}_t, \mathbf{o}) dt$$



Volumetric path tracing

- We want to extend our path tracer to integrate volumes
- There are several manner to do this, with different tradeoffs
- In our case, we want to extend a path along one direction only, as before, to avoid exponential growth
- For simplicity, we will model volumes as enclosed into surfaces that are either the surface of real-world objects, like glass, as fictitious surfaces to bound the volume, like a fog proxy volume
- In this case, a path can either
  - reflect/refract on a surface
  - scatter inside the medium
- Once again, the resulting Monte Carlo algorithm is simpler than the math that describes it

Volumetric path tracing

- Basic idea: split the integral into surface and volume contributions

surface contribution

$$L_i(\mathbf{x}, -\mathbf{o}) = T(\mathbf{x}, \mathbf{x}_d) L_o(\mathbf{x}_d, \mathbf{o}) + \int_0^d T(\mathbf{x}, \mathbf{x}_t)[\sigma_a(\mathbf{x}_t) L_e(\mathbf{x}_t, \mathbf{o}) + \sigma_s(\mathbf{x}_t) L_s(\mathbf{x}_t, \mathbf{o})] dt$$

volume contribution

Volumetric path tracing

- To estimate it, sample a distance along a path and either estimate the surface contribution or the volume one
- Now we have *Le*, but need to estimate *Ls*

surface estimate
using path tracing

$$L_i(\mathbf{x}, -\mathbf{o}) \approx \frac{T(\mathbf{x}, \mathbf{x}_d) L_o(\mathbf{x}_d, \mathbf{o})}{P(d)} + \frac{T(\mathbf{x}, \mathbf{x}_t)[\sigma_a(\mathbf{x}_t) L_e(\mathbf{x}_t, \mathbf{o}) + \sigma_s(\mathbf{x}_t) L_s(\mathbf{x}_t, \mathbf{o})]}{p(t)}$$

probability of
picking surface
at distance d

pdf for sampling
distance t

volume estimate
by scattering in volume

Volumetric path tracing

- To estimate the in-scattered radiance, pick one direction according to the phase function and continue the path; can also integrate with MIS

$$L_i(\mathbf{x}, -\mathbf{o}) \approx \frac{T(\mathbf{x}, \mathbf{x}_d) L_o(\mathbf{x}_d, \mathbf{o})}{P(d)} + \frac{T(\mathbf{x}, \mathbf{x}_t)[\sigma_a(\mathbf{x}_t) L_e(\mathbf{x}_t, \mathbf{o}) + \sigma_s(\mathbf{x}_t) L_s(\mathbf{x}_t, \mathbf{o})]}{p(t)}$$

$$L_s(\mathbf{x}_t, \mathbf{o}) \approx \frac{f_p(\mathbf{x}_t, \mathbf{i}, \mathbf{o}) L_i(\mathbf{x}_t, \mathbf{i})}{p(\mathbf{i})}$$

pdf for sampling direction i

Volumetric path tracing

- We can write a volumetric path tracer starting from a standard one
- Here we consider the simplification of having only smooth birds

```
vec3f shade_path(scene* scene, ray3f ray) {
 <init>
 for(auto bounce : range(max_bounce)) {
    <intersection and environment>
    <eval point and material>
    <emission>
    <sample brdf direction>
    <russian roulette>
    <recurse>
  }
  return l;
}
```


Volumetric path tracing

- Volumes are contained into surfaces that are used to delimit them
- We send a ray to compute the maximum distance to the surface
- After intersection, we sample a distance and either scatter in the volume or on the surface

```
vec3f shade_volpath(scene* scene, ray3f ray) {
 <init>
 for(auto bounce : range(max_bounce)) {
    <intersection and environment>
    <sample transmittance>
    if(!in_volume) <handle surface>
    else <handle volume>
    <russian roulette>
    <recurse>
  }
  return l;
}
```

Volumetric path tracing

- We track the objects we are in with a stack
- We then sample the free-flight distance and update the weight by the transmission over its pdf; for simplicity here we handle both surface and volume interactions

```
<sample transmittance>
  auto in_volume = false;
  if (!vstack.empty()) {
    auto extinction = volume_stack.back().extinction;
    auto distance = sample_transmittance(
          extinction, isec.distance, rand1f(rng), rand1f(rng));
    W *= eval_transmittance(extinction, distance) /
         sample_transmittance_pdf(extinction, distance,
                                  isec.distance);
    in_volume = distance < isec.distance;
    isec.distance = distance;
  }
```

Volumetric path tracing

- The surface interaction is the same as a standard path tracer, except that we update the volume stack upon entering and exiting objects.

```
<handle surface> {
  auto [p, n] = eval_point(isec);     // eval pos, norm
  auto [e, f] = eval_material(isec);  // eval bsdf
  auto o = -ray.d;                    // outgoing
  l += w * eval_emission(e, n, o);    // emission
  auto i = rand1f(rng) < 0.5 ?        // incoming
     sample_brdfcos(n, o, rand1f(rng),rand2f(rng)) :
     sample_lights(p, rand1f(rng),rand1f(rng),rand2f(rng));
  w *= eval_brdfcos(f,n,i,o) * 2 /
    (sample_brdfcos_pdf(f,n,i,o) + sample_lights_pdf(p,i));
  ray = {p, i};                       // setup recurse
  if (has_volume(isec) && dot(n,o)*dot(n,i)<0){// if change side
    if (volume_stack.empty())         // update volume stack
      vstack.push_back(eval_volume(isec));
    else vstack.pop_back();
}
```

Volumetric path tracing

- The volume interaction is similar to the surface one, but we scatter with respect to the phase function rather than the bsdf

```
<handle volume> {
  auto p = eval_ray(ray, isec.distance); // eval pos
  auto vol = vstack.back();              // get volume props
  auto o = -ray.d;                       // outgoing
  l += w * eval_emission(vol, p, o);  // emission
  auto i = rand1f(rng) < 0.5 ?        // incoming
     sample_scattering(vol,o,rand2f(rng)) :
     sample_lights(p,rand1f(rng),rand1f(rng),rand2f(rng));
  w *= eval_scattering(vol,i,o) * 2 /
    (sample_scattering_pdf(vol,i,o) + sample_lights_pdf(p,i));
  ray = {p, i};                          // setup recurse
}
```


Volumetric path tracing

- To write our estimator, we need to write functions to
  - Sample a distance in the medium, and compute the distance pdf
  - Compute the transmission at a given distance
  - Sample the phase function and compute its PDF
- We'll now introduce each of these pieces

Homogeneous volumes

- Handling distance sampling and computing transmittance is easy for homogenous volumes, i.e. volumes that have constant parameters, since analytic solutions are available for all computations
- Homogenous volumes are particularly important since they can be used to model skin and refractive glass.

Homogeneous volumes

- For homogeneous volumes, transmittance is just Beer's law

$$T(t) = e^{-\sigma_t t}$$

- We want to sample distances with a PDF that is proportional to transmittance; this makes for an efficient algorithm since volumes with low transmittance will more likely sample the surface and vice-versa
- As before we use a warp for which we do not derive the formula

$$t = -\frac{\ln(1-r)}{\sigma_t} \qquad p(t) = \sigma_t e^{-\sigma_t t}$$

- From this pdf, we can compute the probability of hitting the surface

$$P(d) = \int_d^\infty p(t)dt = e^{-\sigma_t d}$$

Homogeneous volumes

- For colored extinction, we cannot directly use the previous formulae, since sampling distance depends on wavelength
- In this case, we can simple apply MIS over the channels
- The sampling function first pick a channel and then samples according to that, while the pdf is computed as the weighted sum of pdfs

Homogeneous volumes

- The transmittance functions are straightforward implementations

```
float sample_transmittance(
    vec3f extinction, float max_distance, float rl, float rd) {
  auto channel  = clamp((int)(rl * 3), 0, 2);
  auto distance = (extinction[channel] == 0) ?
       float_max : -log(1 - rd) / extinction[channel];
  return min(distance, max_distance);
}
float sample_transmittance_pdf(
    vec3f& extinction, float distance, float max_distance) {
  if(distance < max_distance) {
    return sum(extinction * exp(-extinction*distance)) / 3;
  } else {
    return sum(exp(-extinction*max_distance)) / 3;
  }
}
vec3f eval_transmittance(vec3f dextinction, float distance) {
  return exp(-extinction * distance);
}
```


Phase function

- Directional scattering is controlled by the phase function
- As before, many model exists to capture different behaviors
- Phase functions are the equivalent of the BSDF for volumes
- The simplest phase function is just a constant, indicating that light is scattered in every direction equally

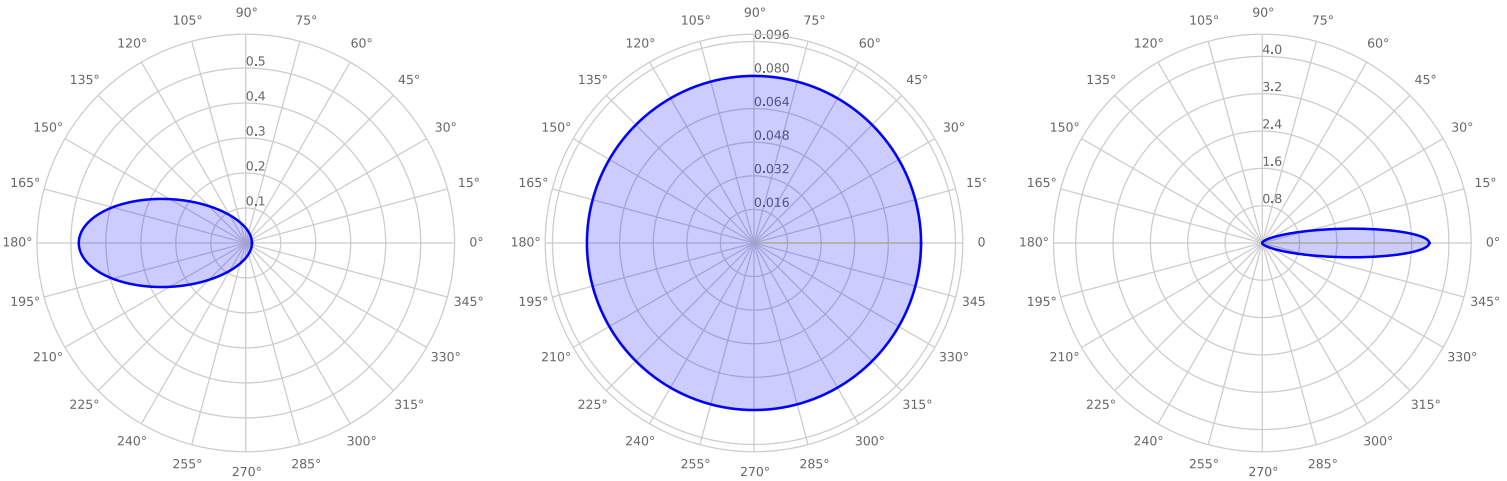
$$f_p(\mathbf{x}, \mathbf{i}, \mathbf{o}) = \frac{1}{4\pi}$$

Phase function

- The most commonly used phase function is the Henyey-Greenstein
- It can describe isotropic materials, but also anisotropic materials, that scatter mostly into one direction

$$f_p(\mathbf{i}, \mathbf{o}) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 - 2g(\mathbf{i} \cdot \mathbf{o}))^{3/2}}$$

$g = -0.5$ $g = 0$ $g = 0.8$



[Jaroz]

Phase function

- For the Henyey-Greenstein phase functions, we can use a warp to pick directions with a pdf proportional to the phase function itself

$$\cos\theta = \frac{1}{2g}\left(1 + g^2 - (\frac{1-g^2}{1-g+2gr_2})^2\right)$$

$$\phi = 2\pi r_2$$

$$p(\mathbf{i}, \mathbf{o}) = f_p(\mathbf{i}, \mathbf{o})$$

- We need special handling for g=0

$$\cos\theta = 1 - 2r_2$$

Phase function

- The volume functions are just wrappers over simpler functions

```
vec3f eval_emission(vsdf vol, vec3f p, vec3f o) {
  return vol.extinction * (1 - vol.albedo) * vol.emission;
}

vec3f eval_scattering(vsdf vol, vec3f i, vec3f o) {
  return vol.extinction * vol.albedo *
    eval_phasefunc(vol.anisotropy, i, o);
}

vec3f sample_scattering(vsdf vol, vec3f i, vec3f o) {
  return sample_phasefunc(vol.anisotropy, o);
}

vec3f pdf_scattering(vsdf vol, vec3f i, vec3f o) {
  return pdf_phasefunc(vol.anisotropy, i, o);
}
```


Phase function

- The scattering function functions are straightforward implementation

```
vec3f eval_phasefunc(float g, vec3f i, vec3f o) {
  auto c = -dot(i, o);
  return (1 - g*g) / (4 * pi * pow(1 + g*g - 2*g*c,3/4));
}
vec3f sample_phasefunc(float g, vec3f o, vec2f rn) {
  auto cos_theta = 0.0f;
  if (abs(g) < 1e-3f) { cos_theta = 1 - 2 * rn.y; } else {
    auto square = (1 - g * g) / (1 - g + 2 * g * rn.y);
    cos_theta    = (1 + g * g - square * square) / (2 * g);
  }
  auto sin_theta = sqrt(max(0, 1 - cos_theta * cos_theta));
  auto phi       = 2 * pif * rn.x;
  auto il = {sin_theta*cos(phi), sin_theta*sin(phi), cos_theta};
  return basis_fromz(-o) * il;
}
vec3f sample_phasefunc_pdf(float g, vec3f i, vec3f o) {
  return eval_phasefunc(vol.anisotropy, i, o);
}
```