An introduction to Global Illumination

Tomas Akenine-Möller
Modified by Ulf Assarsson
Department of Computer Engineering
Chalmers University of Technology

DAT295/DIT221 Advanced Computer Graphics - Seminar Course, 7.5p

- If you are interested, register to that course
- http://www.cse.chalmers.se/edu/course/TDA362/Advanced Computer Graphics/
- ~13 seminars in total, sp4
- Project (no exam)
  - Self or in groups
- Project examples include:
  - realistic explosions, clouds, smoke, procedural textures
  - fractal mountains, CUDA program, Spherical Harmonics, SSAO, Displacement mapping, Collision detection
  - 3D Game
  - real-time ray tracer, enhanced path tracing.
  - HDRI

GFX Companies Gothenburg

3D software development:
Rapid Images
Epic Games
EA Sports
NVIDIA
Smart Eye AB,
EON Reality,
Spark Vision
(Autodesk)
MindArk
Mentice
Vizendo
Surgical Science
Combitech
Fraunhofer (Chalmers Teknikpark)
RD&T Technology

And many more that I have forgotten now…

For graphics artists:
Rapid Images
AFRY
Zoink games
Industriromantik
Stark Film
Edit House
Bobby Works
Filmgate
Ord och bild
Magoo 3D Studios
Tenjin Visual
Silverbullet Film
Tengbom
MFX – www.mfx.se

Non-Gothenburg

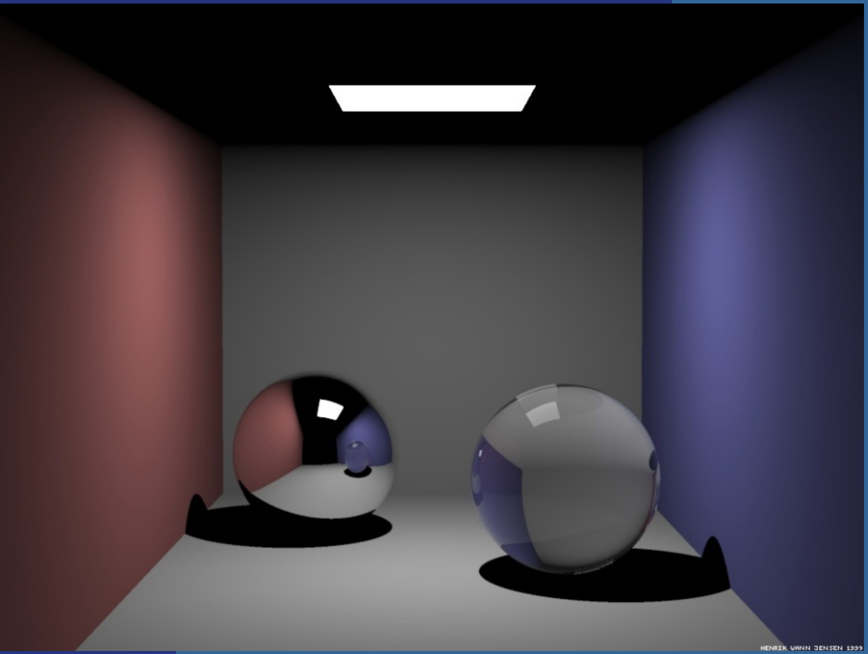
Game Studios:
Avalanche studios (Sthlm)
DICE (Sthlm)
Massive (Malmö)

Architects
Arcitec – (Sthlm)– visualization of buildings for architects

Architects, graphics artists:
White
Wingårdhs
Volvo Personvagnar
Semcon
Ramböll
Zynka
CAP AB

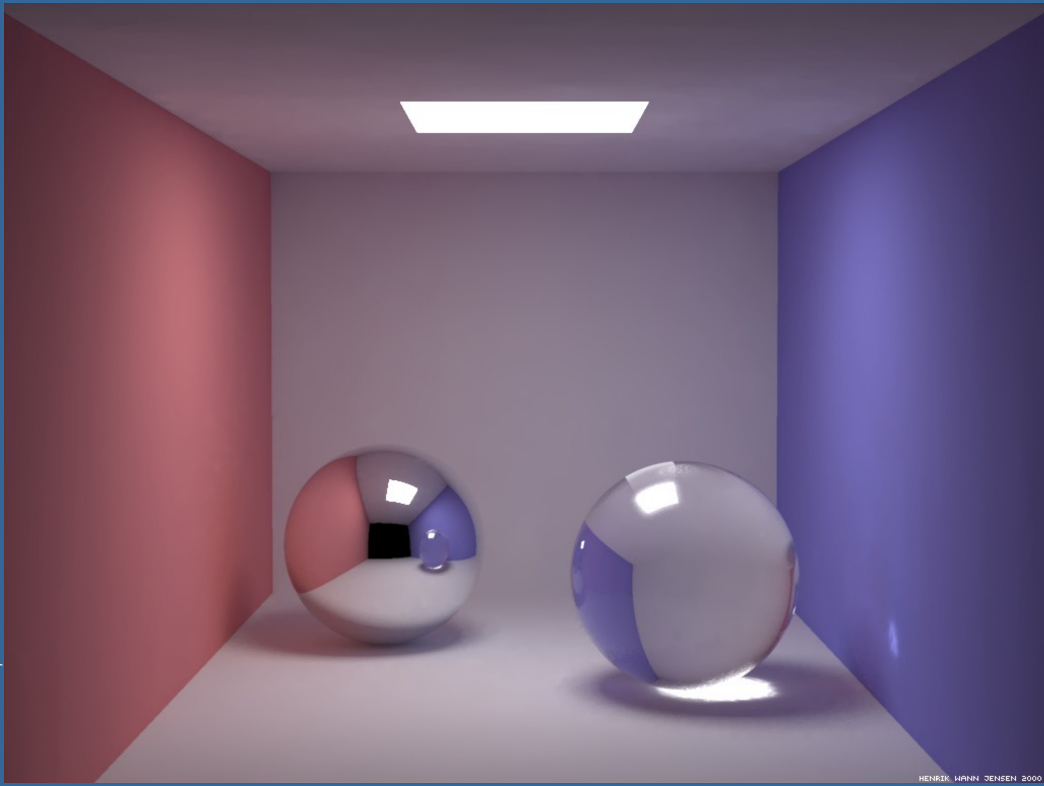
Isn't classic ray tracing enough?

Effects to note in Global Illumination image:
1) Indirect lighting (light reaches the roof)
2) Soft shadows (light source has area)
3) Color bleeding (example: roof is red near red wall) (same as 1)
4) Caustics (concentration of refracted light through glass ball)
5) Materials have no ambient component



Whitted Ray tracing
(reflections, refractions, shadows)

Which are
the differences?



Global
Illumination

Images courtesy of Henrik Wann Jensen

Global Illumination

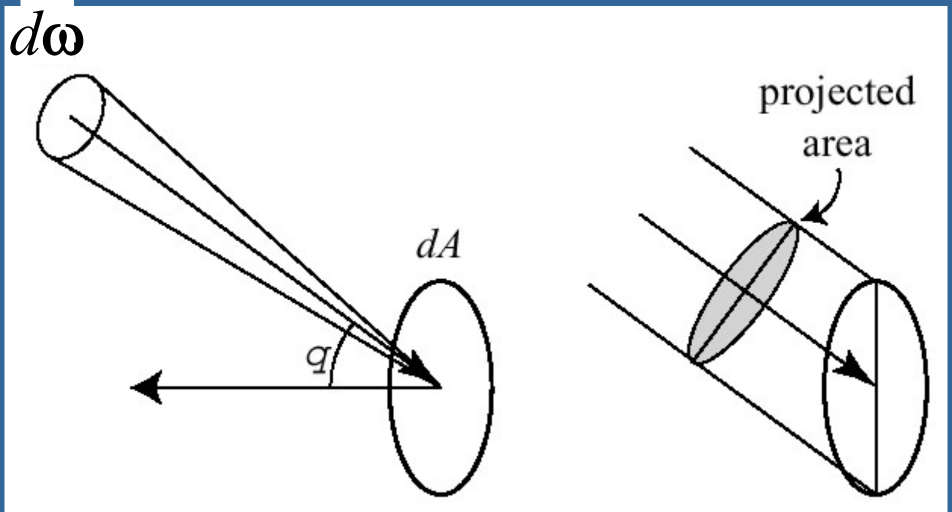
- The goal: **follow** all **photon/ray** bounces through a scene, in order to render images with all kinds of light paths.
- This will give incredibly realistic images
- This lecture will treat:
  - Background: radiance, *the rendering equation*
  - **Monte Carlo** ray tracing:
    - Path tracing
    - Bidirectional Path tracing
    - Denoising - Final Gathering or AI denoising
    - Photon mapping
- Great book on global illumination:
  - Pharr, Humphreys, Physically Based Rendering, 2010
    - With source code.

Background: Radiance

- Radiance, L : a radiometric term. What we store in a pixel is the radiance towards the eye
  - the amount of electromagnetic radiation leaving or arriving at a point on a surface (per unit solid angle per unit projected area)
- Five-dimensional (or 6, including wavelength):
  - Position (3)
  - Direction (2) – horizontal + vertical angle
- Radiance is ”power per unit projected area per unit solid angle”

Solid angle: measured in Steradians (4π is whole sphere).

Uses differentials, so the cone of the solid angle becomes infinitesmally small: a ray



Background: The rendering equation

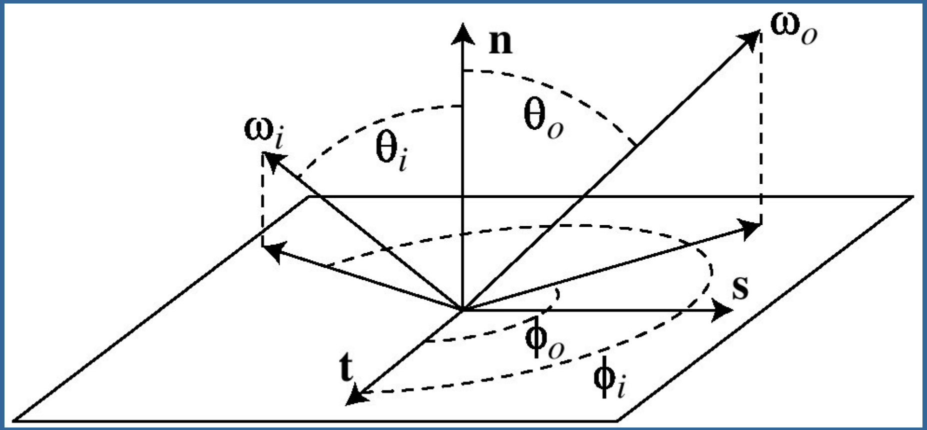
- Paper by Kajiya, 1986 (see course website).
- Is the basis for all rendering, but especially for global illumination algorithms
- $L_o(\mathbf{x},\omega)=L_e(\mathbf{x},\omega)+L_r(\mathbf{x},\omega)$ (slightly different terminology than Kajiya)
  - outgoing=emitted+reflected radiance
  - $\mathbf{x}$ is position on surface, ω is direction vector
- Extend the last term $L_r(\mathbf{x},\omega)$

$$L_o = L_e + \int_{\Omega} f_r(\mathbf{x},\boldsymbol{\omega},\boldsymbol{\omega}')L_i(\mathbf{x},\boldsymbol{\omega}')(\boldsymbol{\omega}'\cdot\mathbf{n})d\boldsymbol{\omega}'$$

- f_r is the BRDF (next slide), ω' is incoming direction, $\mathbf{n}$ is normal at point $\mathbf{x}$, Ω is hemisphere ”around” $\mathbf{x}$ and $\mathbf{n}$, L_i is incoming radiance

Background: Briefly about BRDFs

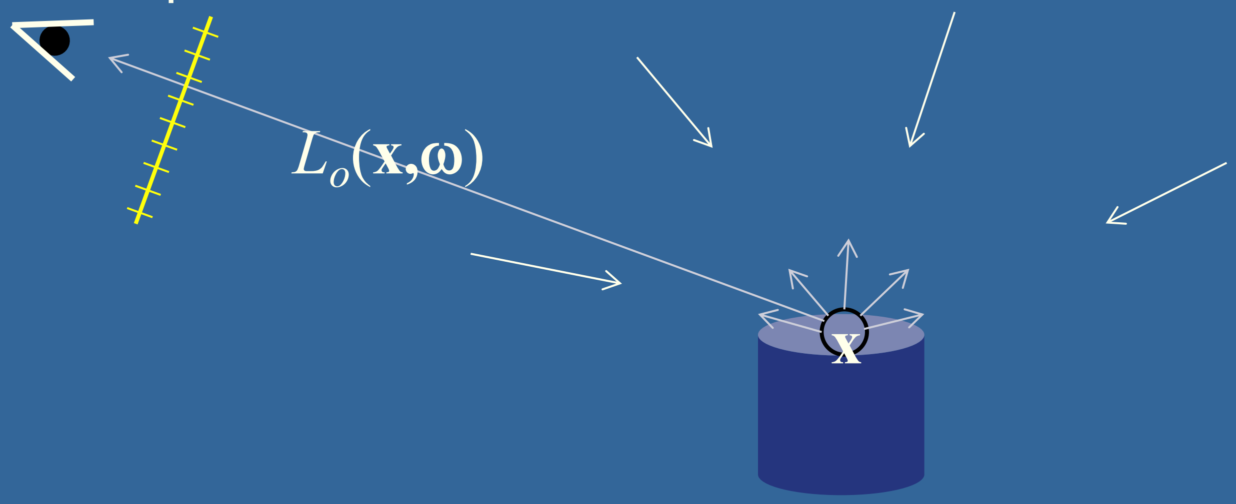
- Bidirectional Reflection Distribution Function
- A more accurate description of material properties
- What it describes: the probability that an incoming photon will leave in a particular outgoing direction



- i is incoming
- o is outgoing
- Many different ways to get BRDF:s
  - Measurement
  - Models:
    - amb+diff+spec
    - Diffuse color, roughness value, metal percentage, …

Radiance/strålning

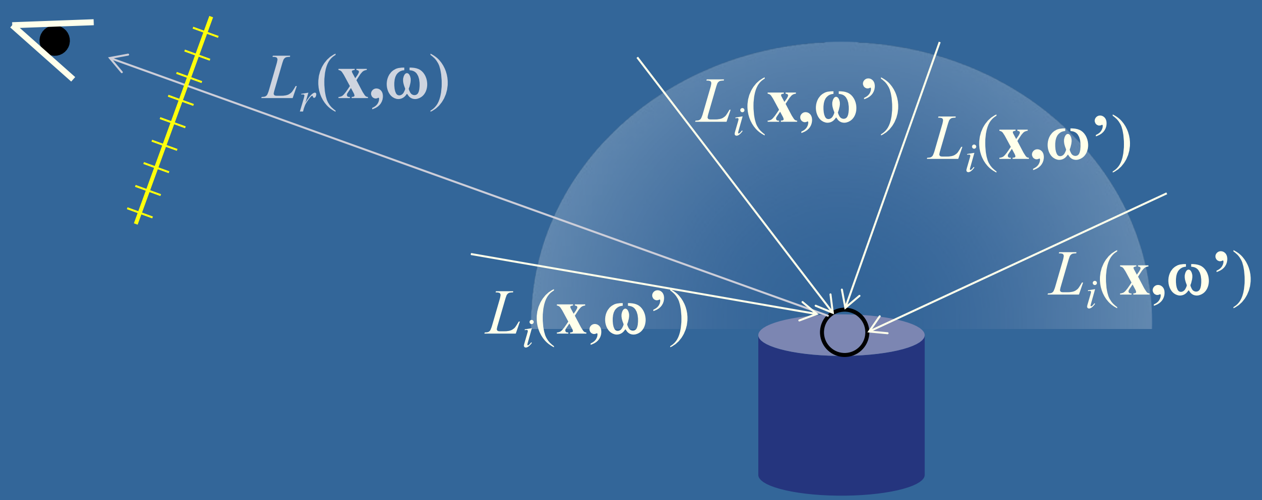
- Radiance, L : a radiometric term. What we store in a pixel is the radiance towards the eye
  - the amount of electromagnetic radiation leaving or arriving at a point on a surface



- L_o = outgoing radiation from a point to a certain direction
- Radiation = color and its intensity, i.e., rbg-value
- $\mathbf{x}$ = x,y,z-position in space
- ω = outgoing direction

The rendering equation - Summary

- Paper by Kajiya, 1986.
- Is the basis for all global illumination algorithms
- $L_o(\mathbf{x},\omega)=L_e(\mathbf{x},\omega)+L_r(\mathbf{x},\omega)$
  - outgoing=emitted+reflected radiance



Integrate over all incoming directions **ω'** to get how much radiance is reflected in outgoing direction **ω**.

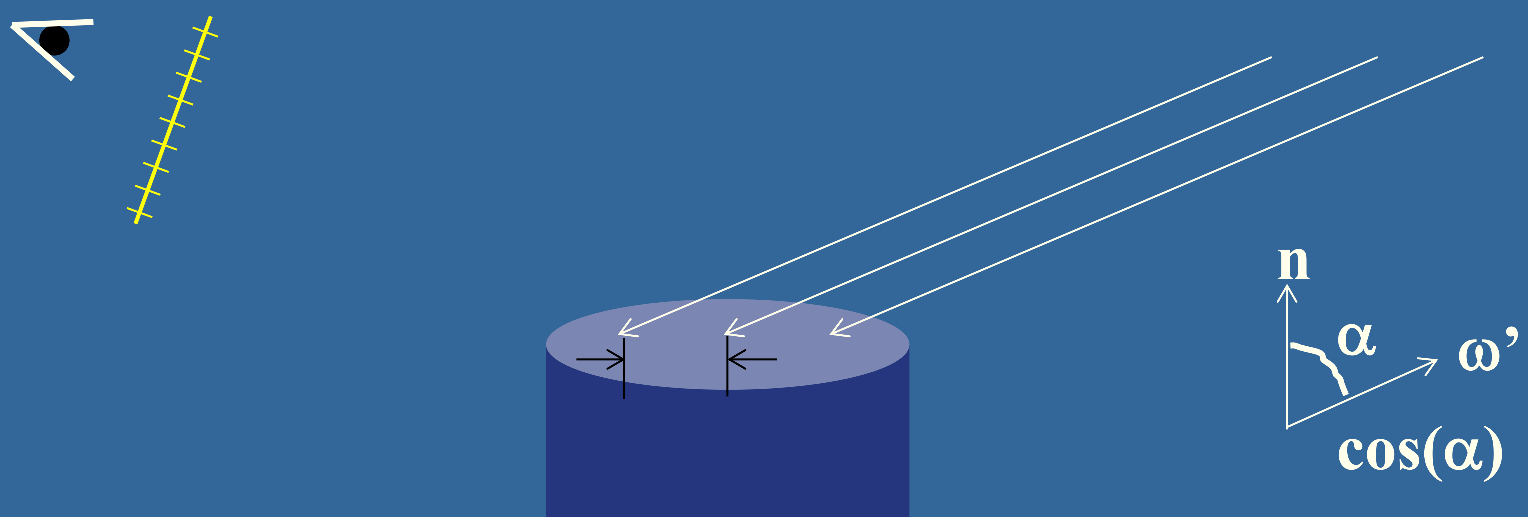
$$L_o = L_e + \int_{\Omega} f_r(\mathbf{x},\boldsymbol{\omega},\boldsymbol{\omega}')L_i(\mathbf{x},\boldsymbol{\omega}')(\boldsymbol{\omega}'\cdot\mathbf{n})\,d\boldsymbol{\omega}'$$

- f_r is the BRDF, **ω'** is incoming direction, **n** is normal at point **x**, Ω is hemisphere "around" **x** and **n**, L_i is incoming radiance

The rendering equation

BRDF = Bidirectional Reflection Distribution Function

- Paper by Kajiya, 1986.
- Is the basis for all global illumination algorithms
- $L_o(\mathbf{x},\omega)=L_e(\mathbf{x},\omega)+L_r(\mathbf{x},\omega)$
  - outgoing=emitted+reflected radiance

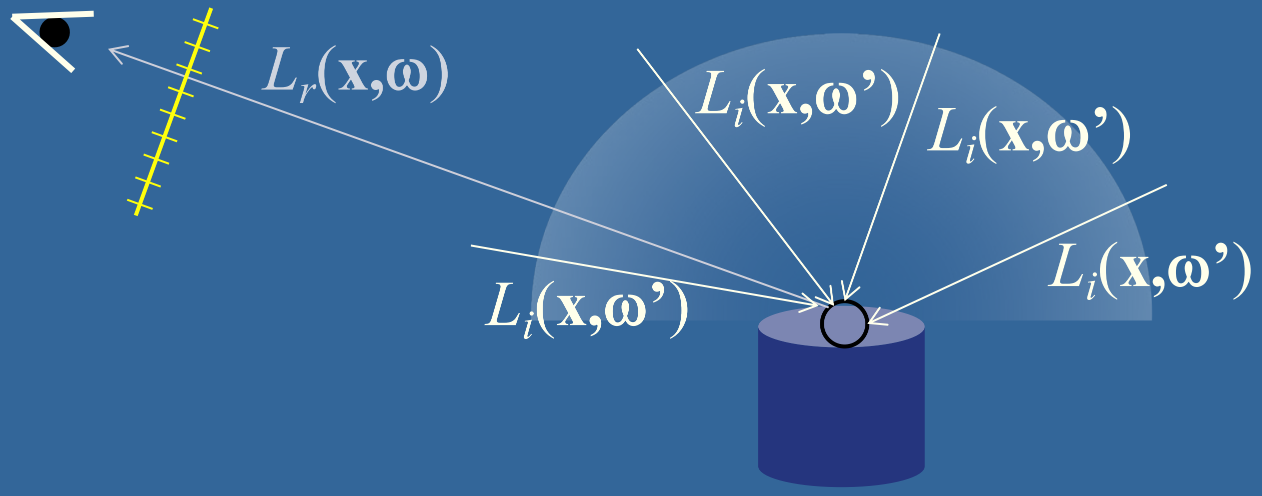


$$L_o = L_e + \int_{\Omega} f_r(\mathbf{x},\omega,\omega')L_i(\mathbf{x},\omega')(\omega'\cdot\mathbf{n})d\omega'$$

- f_r is the BRDF, ω’ is incoming direction, $\mathbf{n}$ is normal at point $\mathbf{x}$, Ω is hemisphere ”around” $\mathbf{x}$ and $\mathbf{n}$, L_i is incoming radiance

The rendering equation

- Paper by Kajiya, 1986.
- Is the basis for all global illumination algorithms
- $L_o(\mathbf{x},\omega)=L_e(\mathbf{x},\omega)+L_r(\mathbf{x},\omega)$
  - outgoing=emitted+reflected radiance



$$L_o = L_e + \int_{\Omega} f_r(\mathbf{x},\boldsymbol{\omega},\boldsymbol{\omega}')L_i(\mathbf{x},\boldsymbol{\omega}')(\boldsymbol{\omega}'\cdot\mathbf{n})d\boldsymbol{\omega}'$$

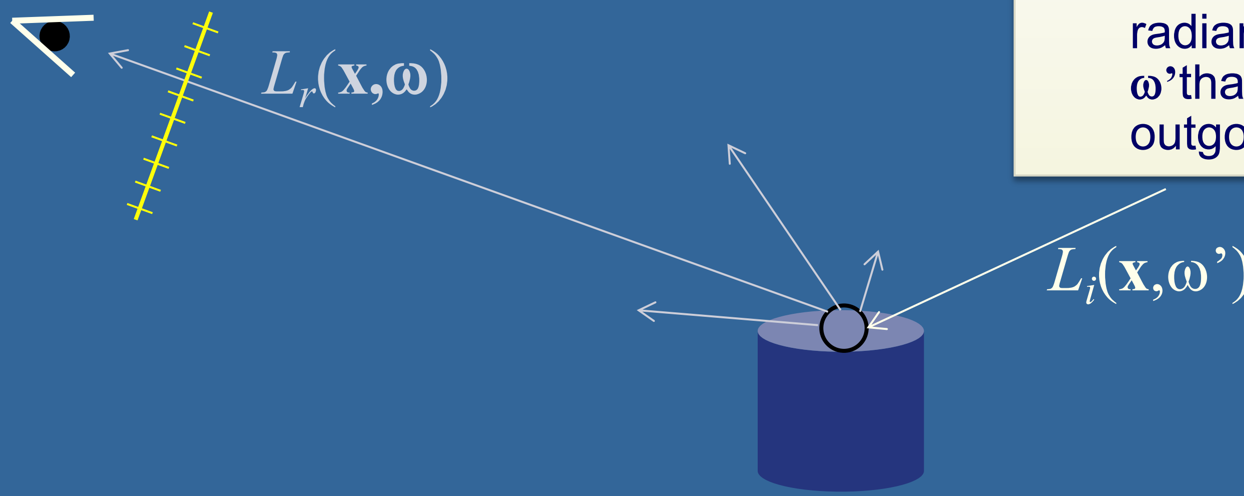
- f_r is the BRDF, ω' is incoming direction, $\mathbf{n}$ is normal at point $\mathbf{x}$, Ω is hemisphere "around" $\mathbf{x}$ and $\mathbf{n}$, L_i is incoming radiance

The rendering equation

BRDF = Bidirectional Reflection Distribution Function

- Paper by Kajiya, 1986.
- Is the basis for all global illumination algorithms
- $L_o(\mathbf{x},\omega)=L_e(\mathbf{x},\omega)+L_r(\mathbf{x},\omega)$
  - outgoing=emitted+reflected radiance

BRDF:
$f_r(\mathbf{x},\omega,\omega') =$
"How much of incoming radiance, L_i, from direction ω' that leaves in an outgoing direction ω"

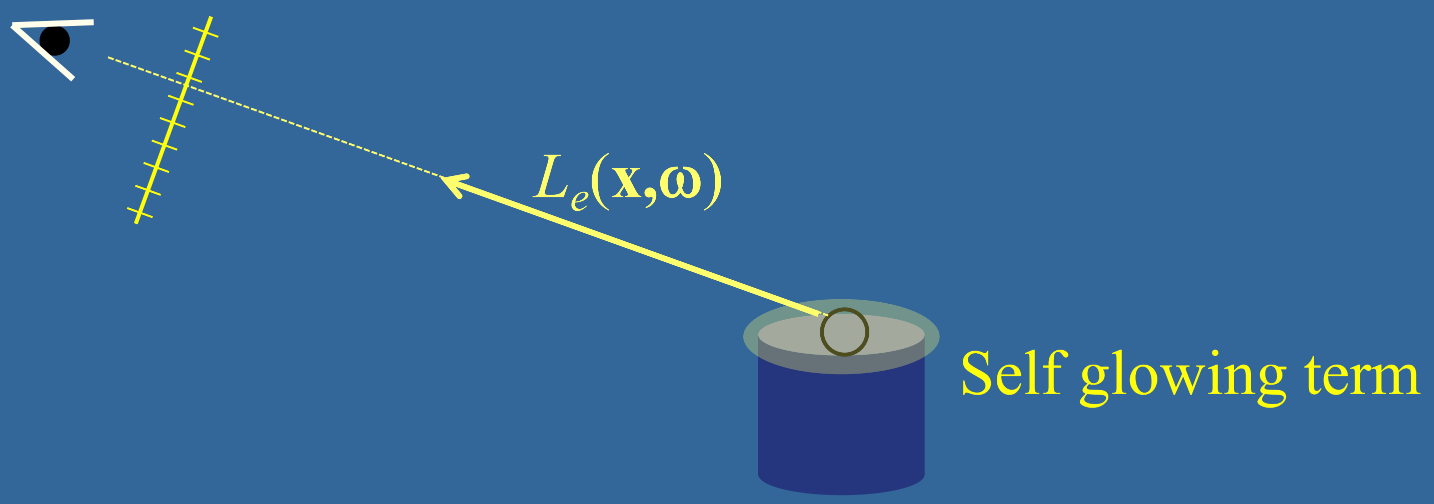


$$L_o = L_e + \int_\Omega f_r(\mathbf{x},\omega,\omega')L_i(\mathbf{x},\omega')(\omega'\cdot\mathbf{n})\,d\omega'$$

- f_r is the BRDF, ω' is incoming direction, $\mathbf{n}$ is normal at point $\mathbf{x}$, Ω is hemisphere "around" $\mathbf{x}$ and $\mathbf{n}$, L_i is incoming radiance

The rendering equation - Summary

- Paper by Kajiya, 1986.
- Is the basis for all global illumination algorithms
- $L_o(\mathbf{x},\omega)=L_e(\mathbf{x}, \omega)+L_r(\mathbf{x}, \omega)$
  - outgoing=emitted+reflected radiance



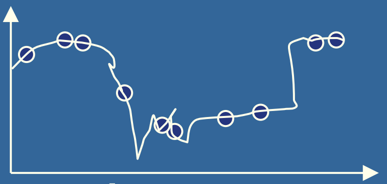
$$L_o = L_e + \int_{\Omega} f_r(\mathbf{x}, \omega, \omega')L_i(\mathbf{x}, \omega')(\omega' \cdot \mathbf{n})d\omega'$$

- f_r is the BRDF, ω' is incoming direction, $\mathbf{n}$ is normal at point $\mathbf{x}$, Ω is hemisphere "around" $\mathbf{x}$ and $\mathbf{n}$, L_i is incoming radiance

Many GI algorithms are built on Monte Carlo Integration

$$L_o = L_e + \int_\Omega f_r(\mathbf{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\mathbf{x}, \boldsymbol{\omega}')(\boldsymbol{\omega}' \cdot \mathbf{n}) d\boldsymbol{\omega}'$$

- Integral in rendering equation:
  - Hard to evaluate numerically
  - But we can sample it.
- MC can estimate integrals: $I = \int_a^b f(x)dx$



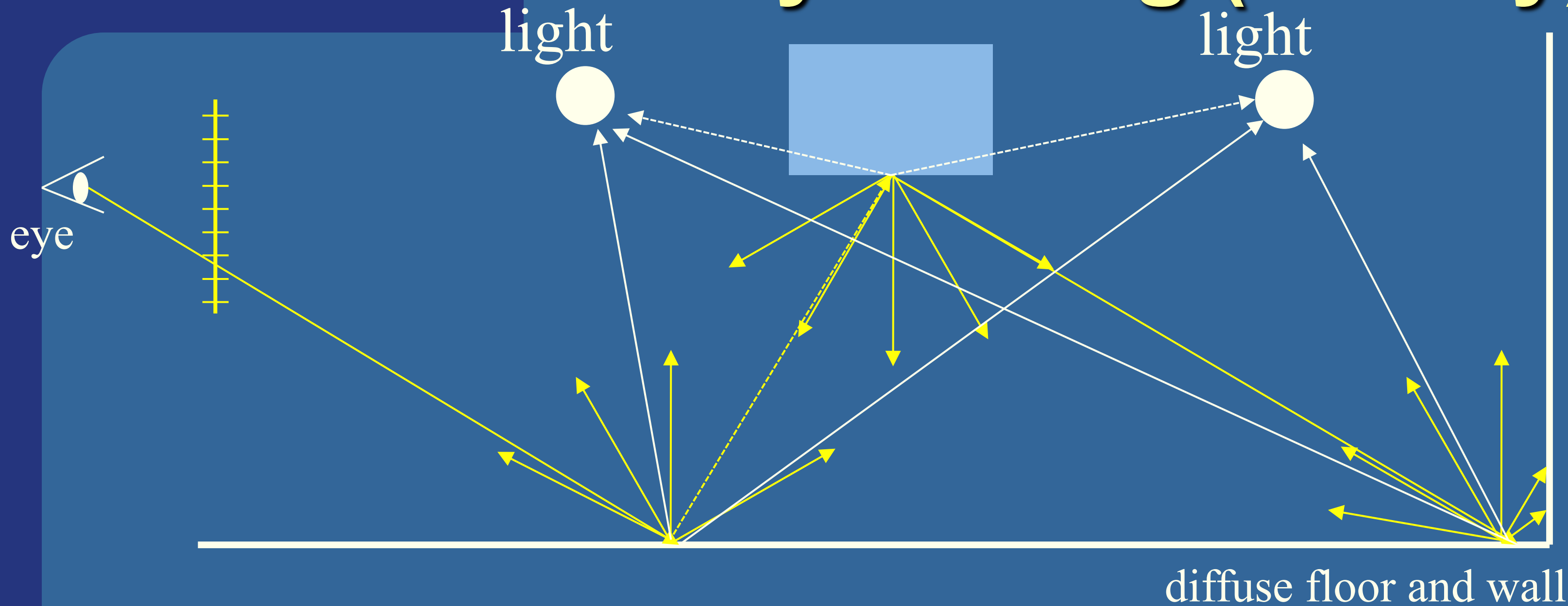
- Assume we can compute the mean of $f(x)$ over the interval $[a,b]$
  - Then the integral is mean*(b-a)
- Thus, focus on estimating mean of $f(x)$
- Idea: sample f at n uniformly distributed random locations, x_i:

$$I_{MC} = (b-a)\frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

Monte Carlo estimate

- When $n \rightarrow infinity$, $I_{MC} \rightarrow I$
- Standard deviation convergence is slow: $\sigma \propto \frac{1}{\sqrt{n}}$
- Thus, to halve error, must use 4x number of samples!

Monte Carlo Ray Tracing (naïvely)

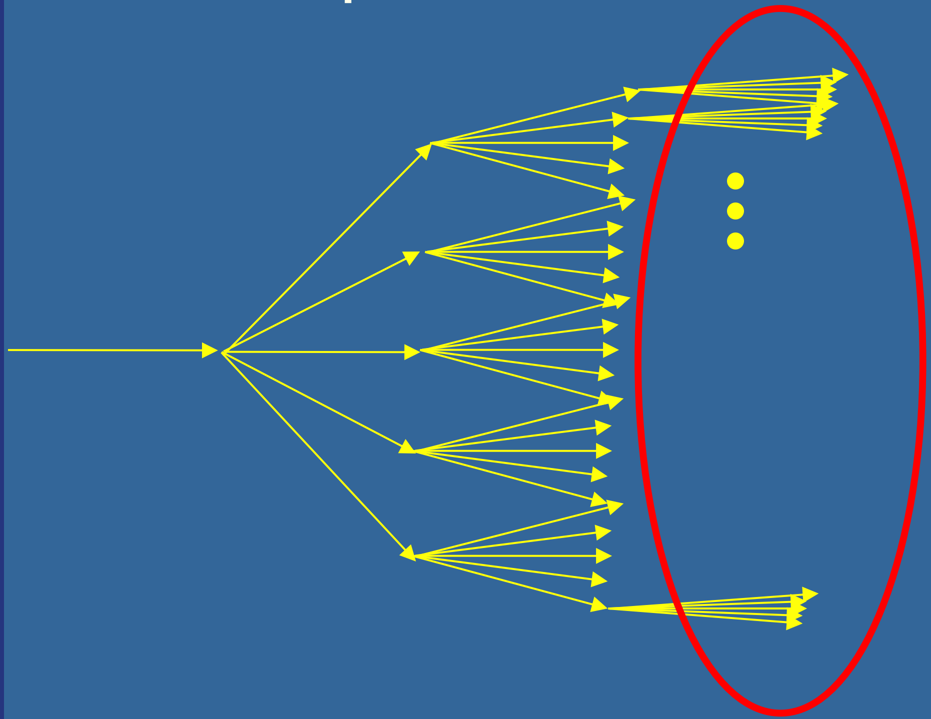


- (Compute local lighting as usual, with a shadow ray per light.)
- Sample indirect illumination by shooting sample rays over the hemisphere, at each hit.

$$L_o = L_e + \int_{\Omega} f_r(\mathbf{x}, \boldsymbol{\omega}, \boldsymbol{\omega}')L_i(\mathbf{x}, \boldsymbol{\omega}')(\boldsymbol{\omega}' \cdot \mathbf{n})d\boldsymbol{\omega}'$$

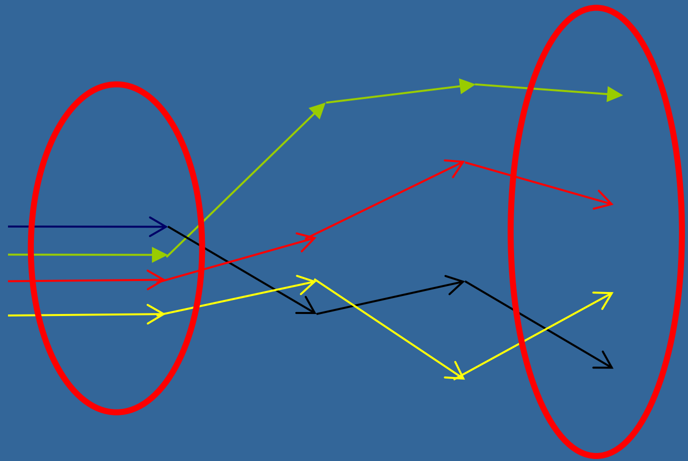
Monte Carlo Ray Tracing (naïvely)

- The indirect-illumination sampling gives a ray tree with most rays at the bottom level. This is bad since these rays have the lowest influence on the pixel color.



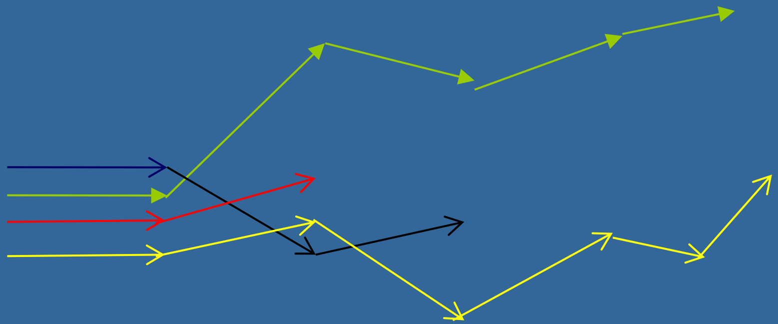
PathTracing
– one efficient Monte-Carlo Ray-Tracing solution

- Path Tracing instead only traces one of the possible ray paths at a time. This is done by randomly selecting only one sample direction at a bounce. Hundreds of paths per pixel are traced.



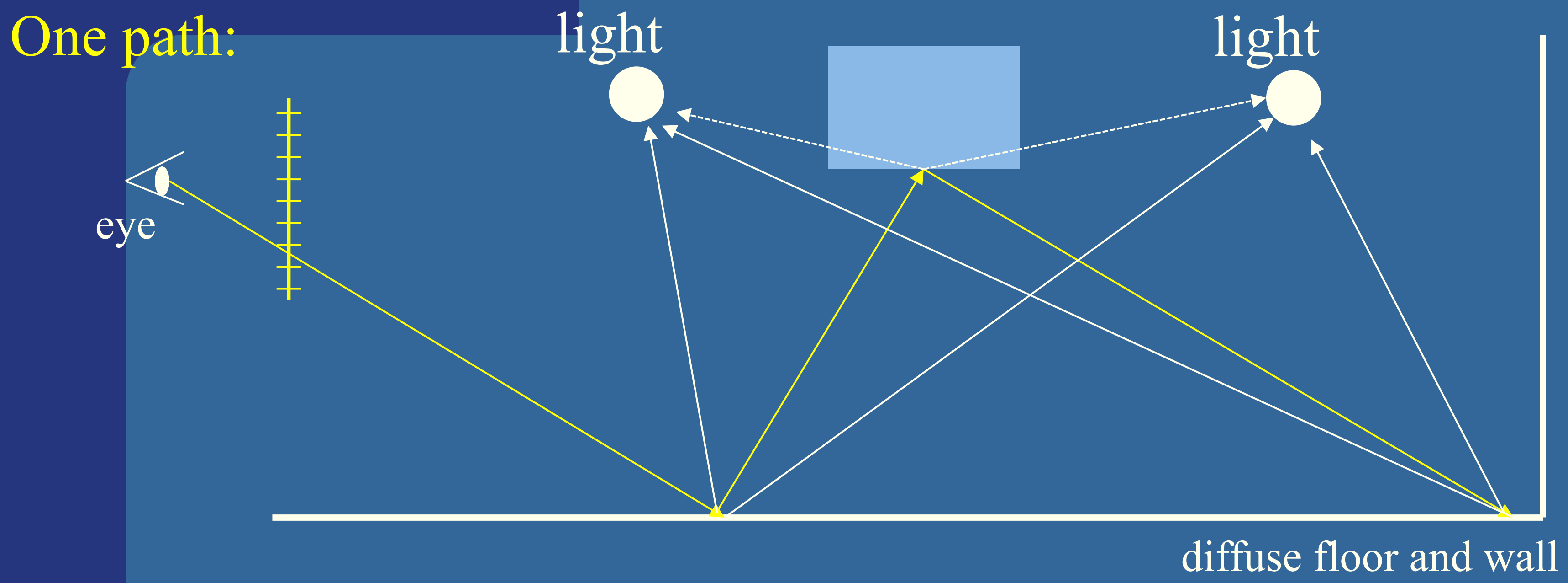
Equally number of rays are traced at each level

Or:



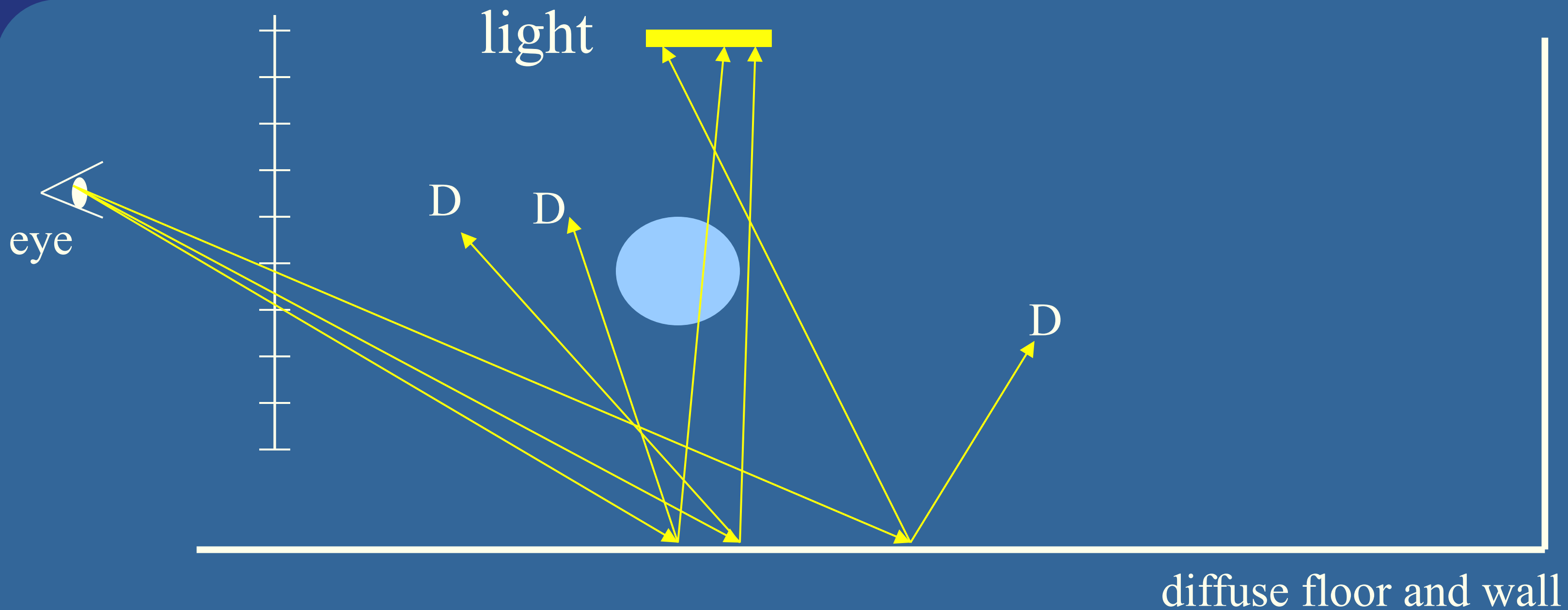
Even smarter: terminate path with some probablility after each level, since they have decreasing importance to final pixel color.

Path Tracing – indirect + direct illumination



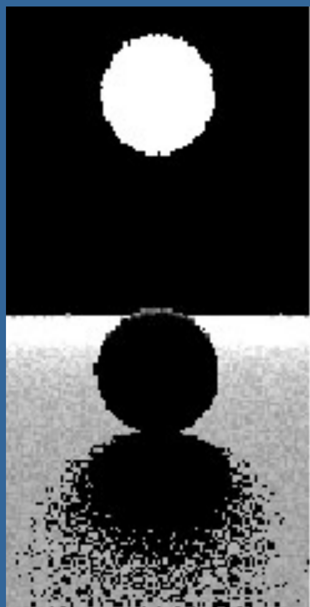
- Shoot many paths per pixel (the image just shows one light path).
  - At each intersection,
    - Shoot one shadow ray per light source
      - at random position on light, for area/volumetric light sources
    - and randomly select one new ray direction.

Path Tracing and area lights

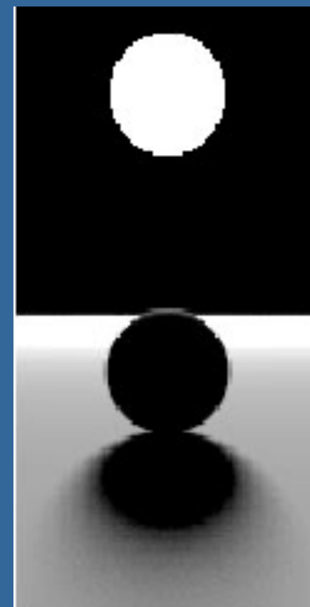


- For area light sources, shoot the shadow ray to one random position on the area light. This gives soft shadows when many paths are averaged for the pixel.
- Example: Three rays for one pixel
  - All three rays hits diffuse floor
  - Pick ***one*** random position on light source
  - Send one random diffuse ray (D's above)
    - To continue the path...

Example of diffuse surface + soft shadows



One sample per pixel



100 samples per pixel

- Need to send many many rays to avoid noisy images
  - Sometimes 1000 or 10,000 rays are needed per pixel!
- Still, it is a simple method to generate high quality images

Images courtesy of Peter Shirley

Path tracing: Summary

- Uses Monte Carlo sampling to solve integration:
  - by shooting many random ray ***paths*** over the integral domain.
  - Algorithm:
    - For each pixel, // we will shoot a number of paths:
      - For each path, generate the primary ray:
        1. Trace the ray. At hitpoint:
        2. Shoot one shadow ray and compute local lighting.
        3. Sample indirect illumination randomly over the possible reflection/refraction directions by generating **one** such new ray.
        4. Repeat from 1, until the path is randomly terminated (or the ray does not hit anything).
- Shorter summary: shoot many paths per pixel, by randomly choosing **one** new ray at each interaction with surface + **one** shadow ray per light.

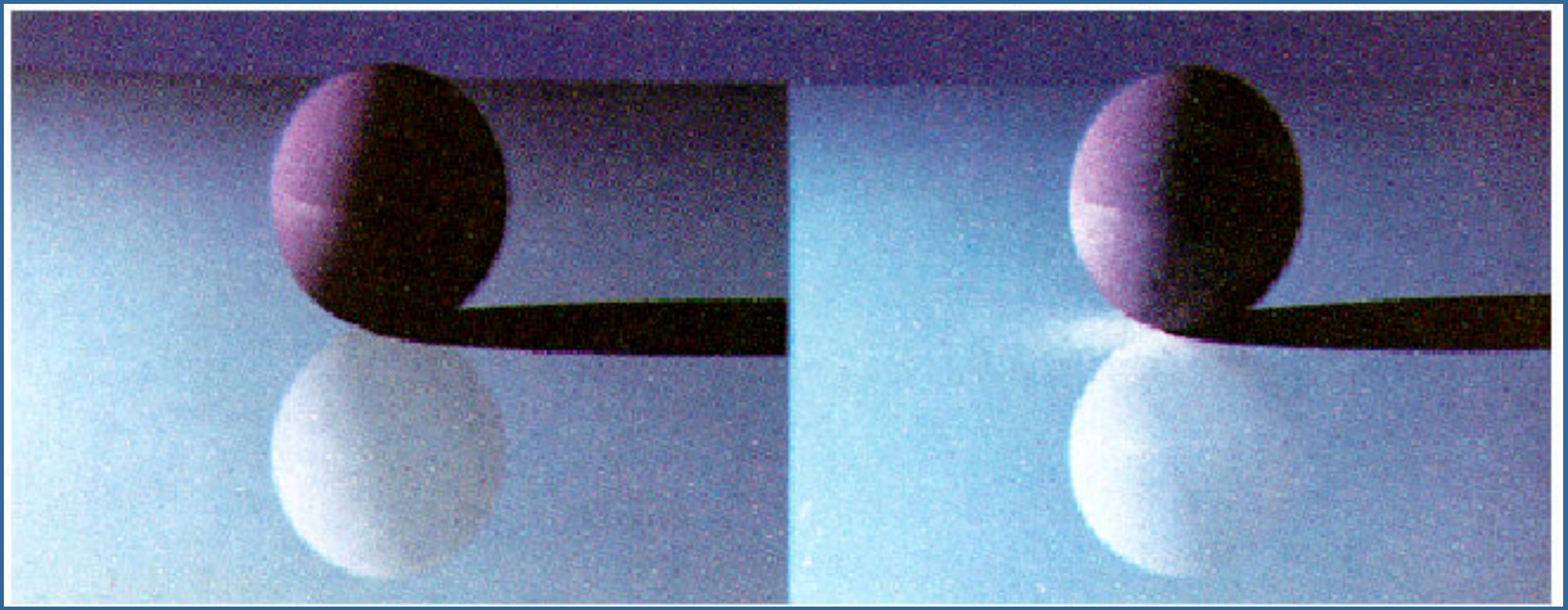
Russian Roulette

E.g., assuming perfectly diffuse and perfectly specular surfaces:

- Assume $k_{diff}+k_{spec}$<=1 (since energy cannot be created)
- When a ray hits such a surface
  - Pick a random number, r in [0,1]
  - If($r < k_{diff}$) → send diffuse ray (e.g. in random direction)
  - Else if($r < k_{diff}+k_{spec}$) → send specular ray (e.g. along reflection dir.)
  - Else absorb ray.
- Or use Fresnel equation:
  - Let k_{spec} = amount of reflectivity for the ray angle
  - Let k_{diff} = amount of refraction for the ray angle if non-transparent mtrl.
    Else, k_{diff} means shooting refractive ray.
- This is called **Russian roulette**.
  - Common for layered materials.
  - and for BRDF's, see path-tracer lab.
- Point: this selects just one ray so we get a path instead of a tree. Also terminates path with a physically-based probability.

A classical example – spec+diff surface + hard shadow

- Path tracing was introduced in 1986 by Jim Kajiya



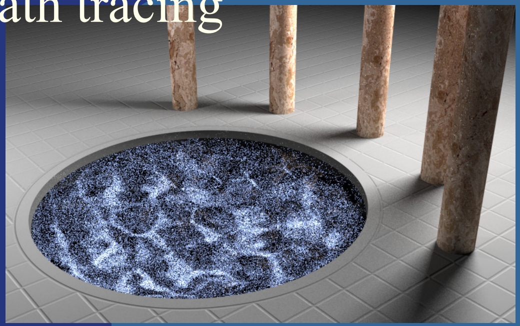
- Note how the right sphere reflects light, and so the ground under the sphere is brighter

What is Caustics?

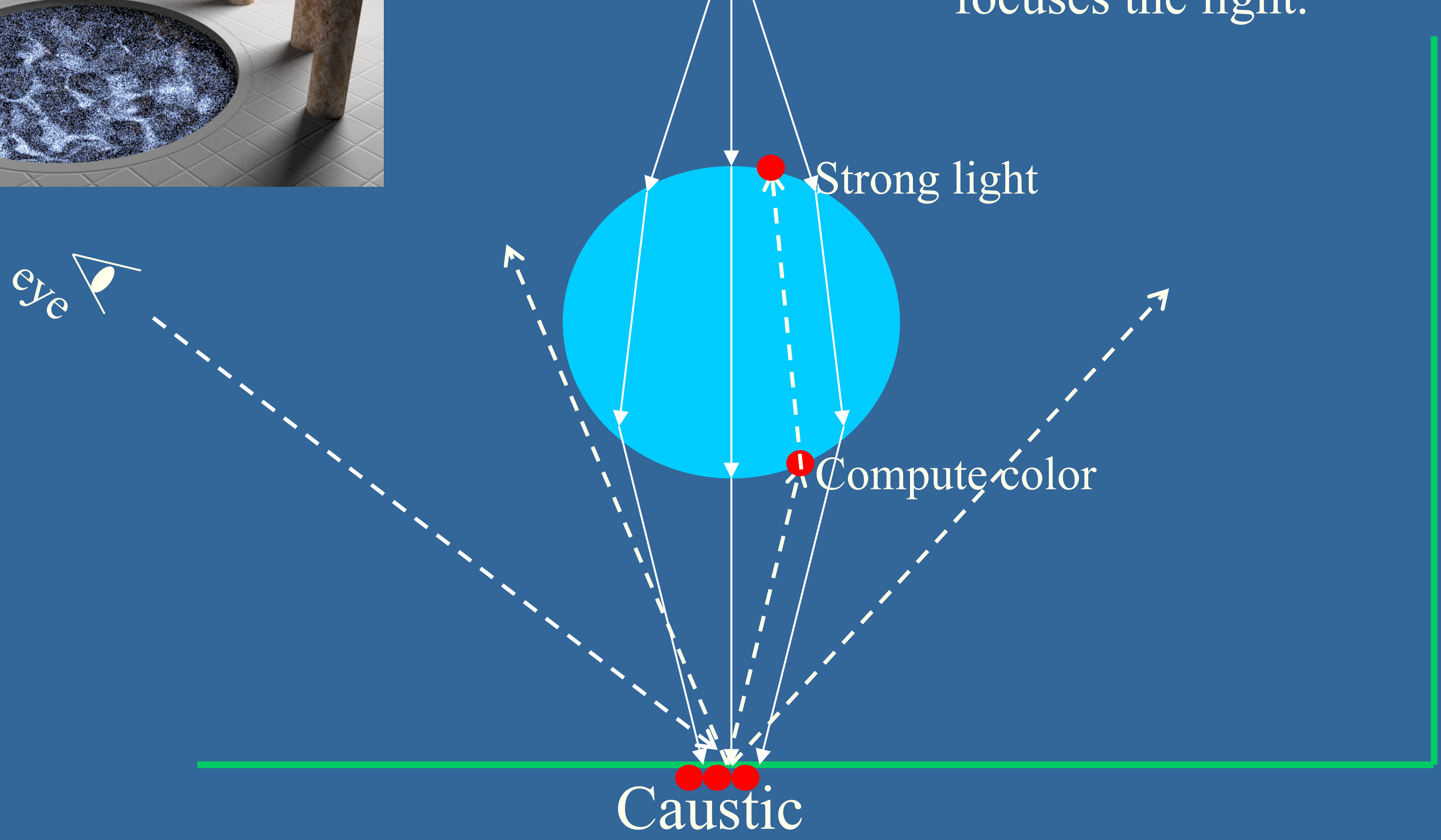
- Caustic's don't work well for path tracing

Reason why forward ray tracing fails to capture caustics well

Path tracing



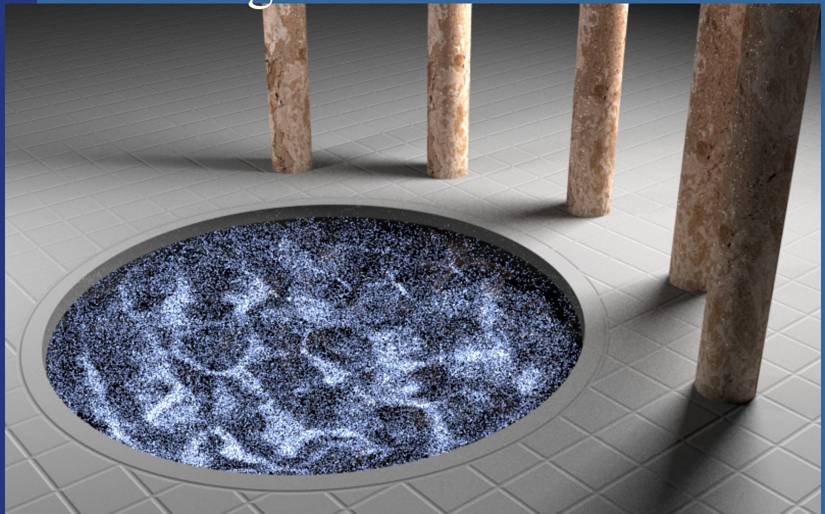
Must be lucky to hit the specular reflector and discover that it focuses the light.



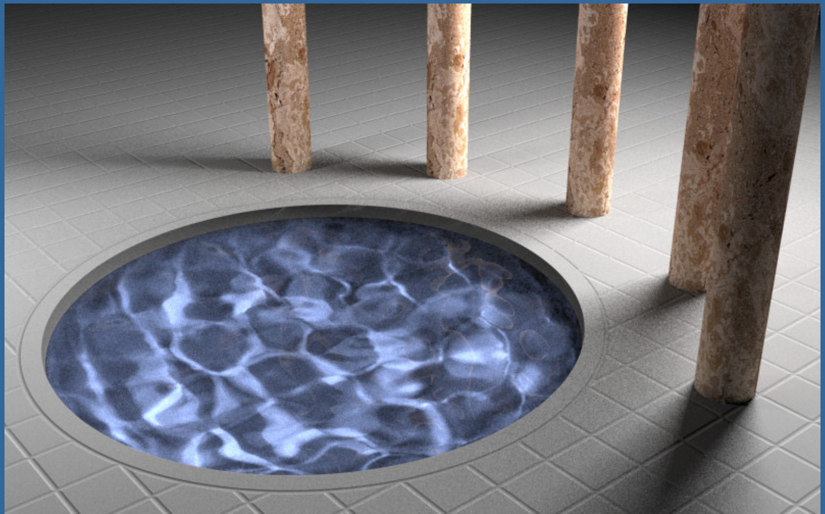
Extensions to path tracing

- Bidirectional path tracing
  - Developed in1993-1994
  - Sends light paths, both from eye and from the light
  - Faster, but still noisy images.
- Metropolis light transport
  - 1997
  - Ray distribution is proportional to unknown function
  - Means that more rays will be sent where they are needed
  - Faster convergence in certain cases (see below)

Path tracing



Metropolis (same rendering time)



Images courtesy of Eric Veach

Bidirectional Path tracing

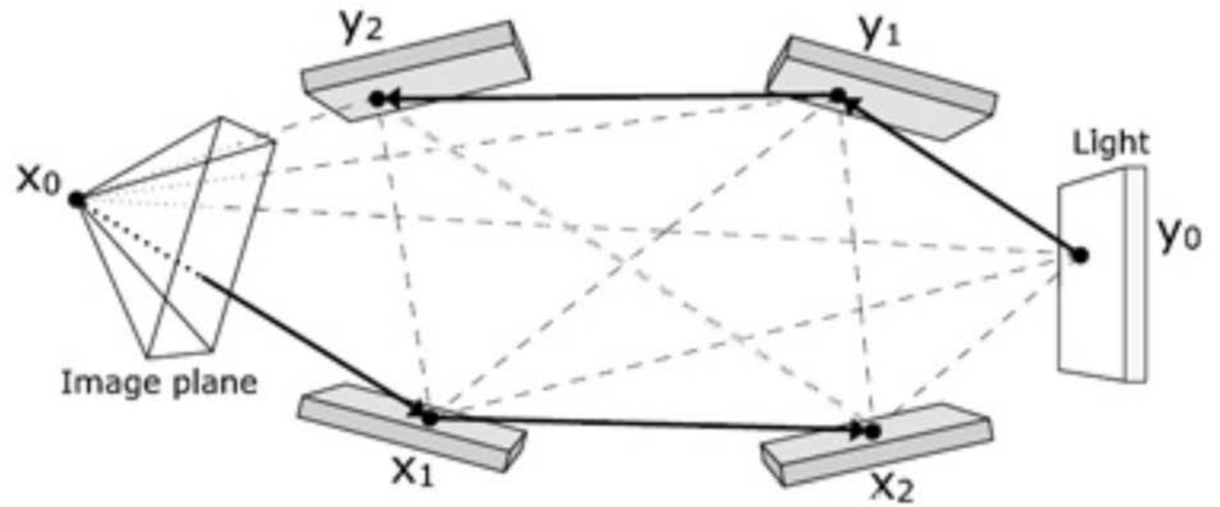


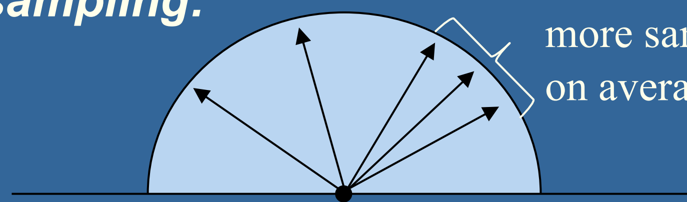
Figure 1: The different ways to combine the eye path and the light path. The eye path starts at the eye in X0 and moves through X1 and X2. The light path starts at the light source in Y0 and moves through Y1 and Y2. To create a complete path, a shadow ray is used to connect X2 and Y2. More paths can be generated by connecting all the different combinations of points on the sub paths.

BONUS

Monte-Carlo Ray tracing – the maths

The weight of the radiance from each sampled ray direction:

- If hemispherical directions are **not** sampled perfectly **randomly**, then the weight for each of the *n* sampled rays is **not** just $w = 1/n$,
  - e.g., when shooting more sampling rays towards the more probable directions (by trying to somewhat regard the BRDF). This is called ***importance sampling:***



- Solutions:
  - In theory, we could look at the actual taken sample directions, and estimate good weights. This is rarely used.
    - Does not work well for path tracing, since we only sample one direction per position.
  - Or, rely on probability theory, which will converge to correct weights when #samples, *n,* goes to infinity.
    - $w_i = 1/\big(n * p(\omega_i)\big), \quad p(\omega_i) = \text{"}probability\_bias\_of\_the\_choosen\_direction\text{"}$

    Where function $p(\omega)$ is our Probability Density Function (PDF)
    - This is what people use today. See our path-tracing tutorial.

How much more/less likely (e.g., *2) direction ω_i is compared to the average direction (avg. should be =1).

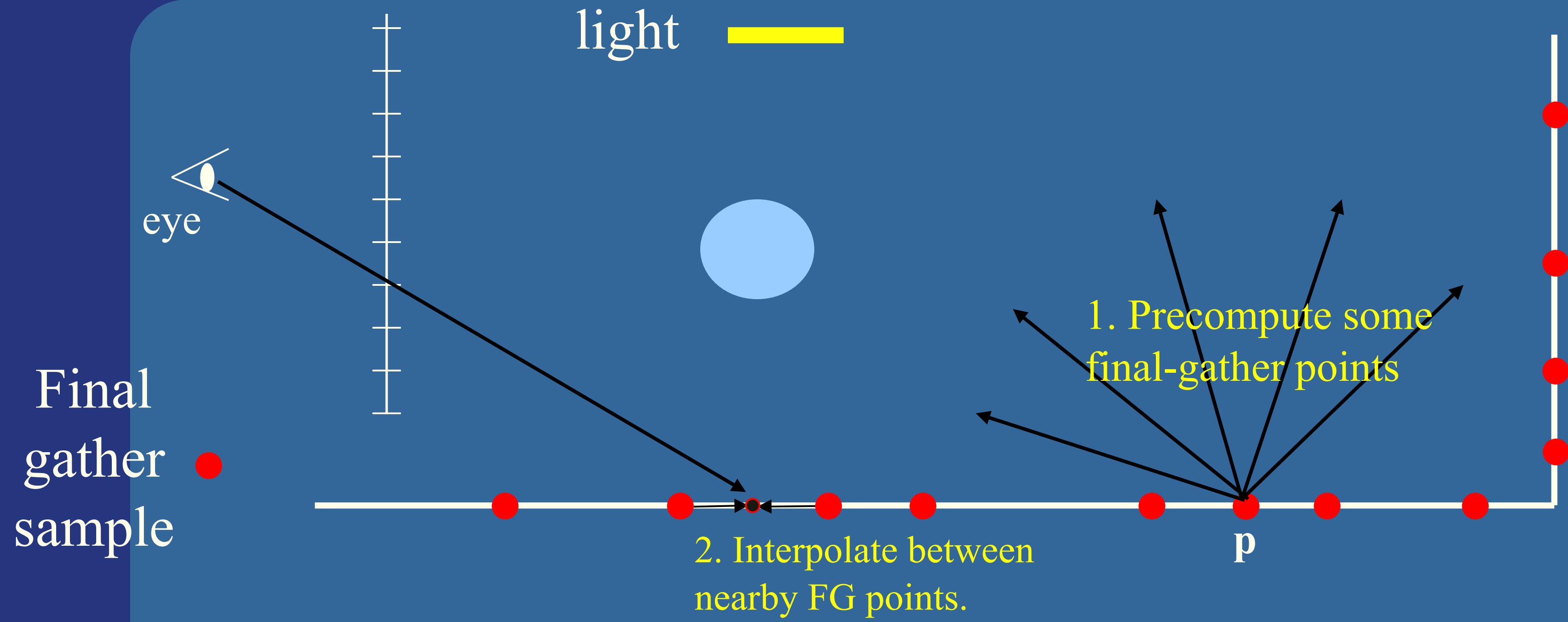
Denoising

- Monte Carlo ray tracing is typically slow or noisy.
- You can denoise by using:
  - Final Gather (older)
  - or AI denoising (new).

Final Gather

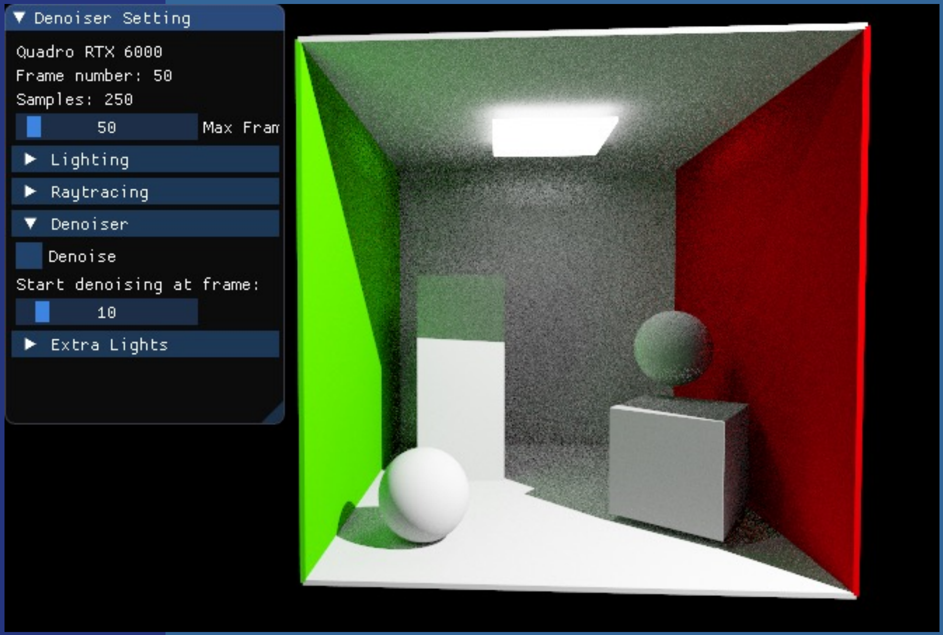
Popular for ray tracing and photon mapping but not path tracing

Idea and good answer:
- Compute indirect illumination somehow, but only at a few positions (final gather points) in the scene.
- Estimate indirect illumination for other positions by interpolation from nearby final-gather points

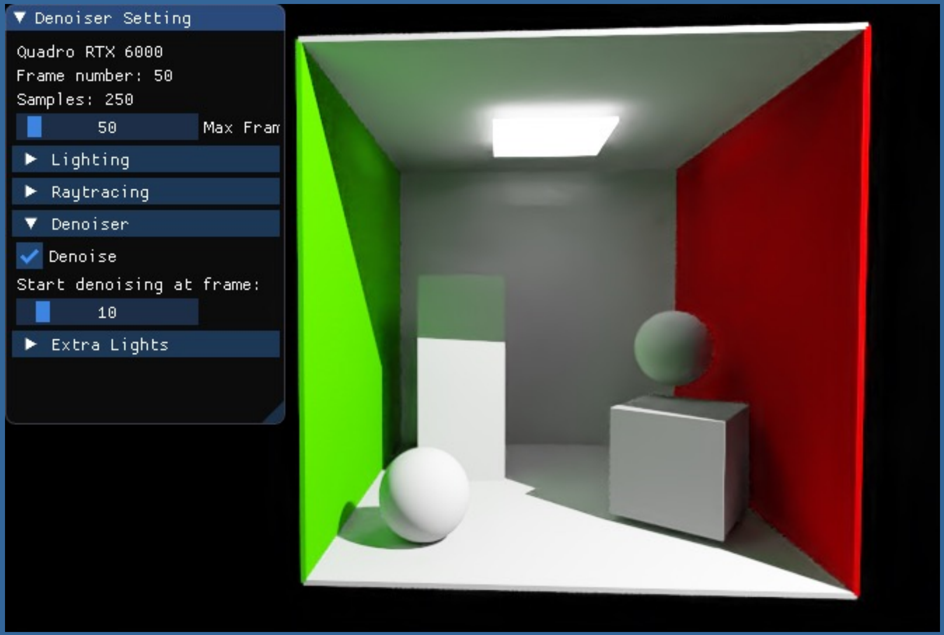


- Many versions of Final Gathering exist.
- E.g., to compute final-gather point **p**:
  - Send hundreds of random rays out from **p** to sample indirect illumination
- To use during ray tracing: interpolate global illumination between nearby Final Gather points, to estimate incoming radiance at the ray's intersection point.

Path tracing + AI Denoising



Before denoising



After denoising

Real-time Denoising - NVIDIA

Need to Start with a Noisy Result and Reconstruct

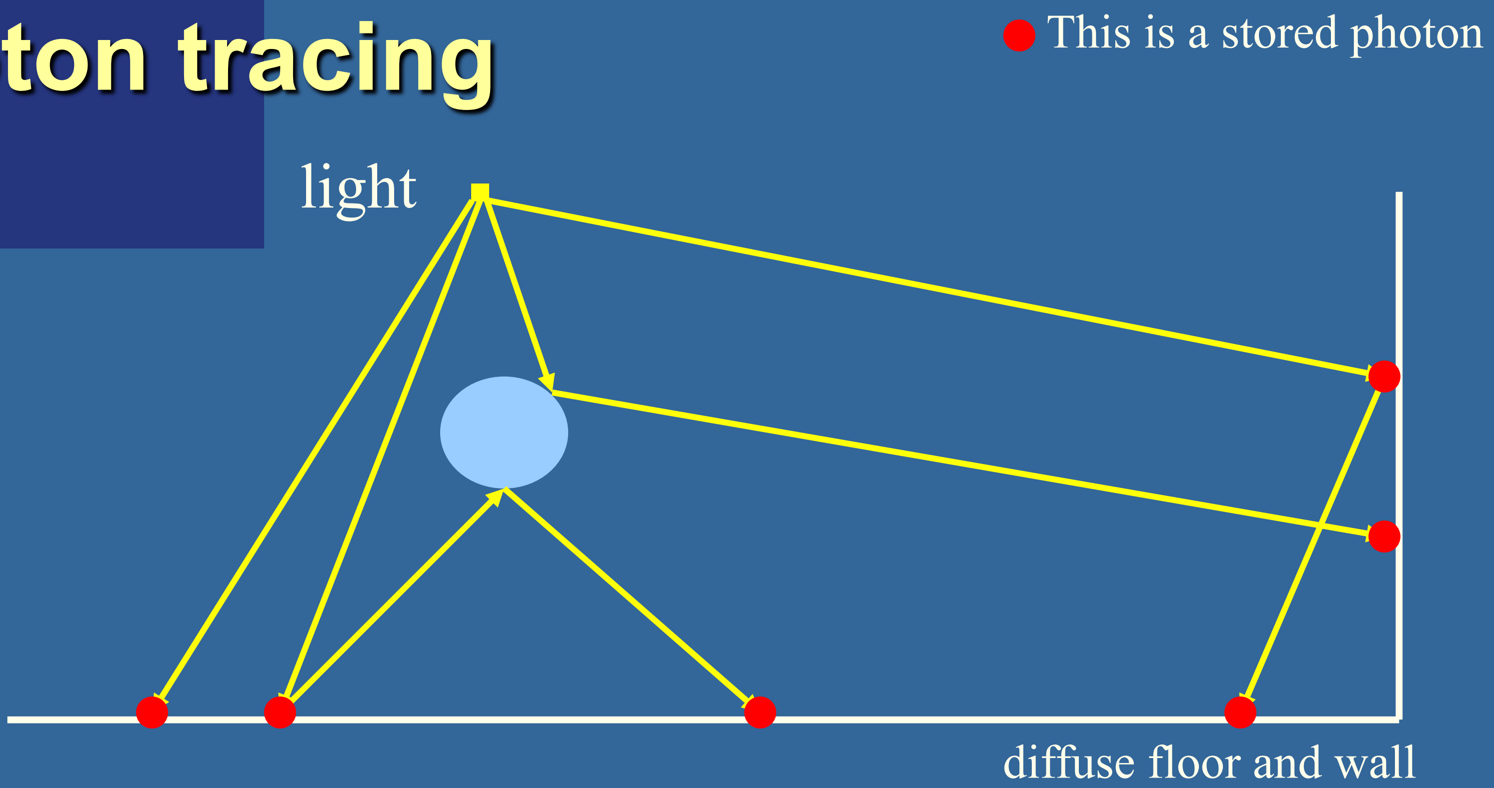
Photon mapping

- Developed by Henrik Wann Jensen (started 1993)
- A clever two-pass algorithm:
  - 1: Shoot photons from light source, and let them bounce around in the scene, and store them where they land
  - 2: ”Ray tracing”-like pass from the eye, but gather the photons from the previous pass
- Advantages:
  - Fast
  - Handles arbitrary geometry (as do path tracing)
  - All global illumination effects can be seen
  - Little noise (in still images)

The first pass: Photon tracing

- Store illumination as points (photons) in a ”photon map” data structure
- In the first pass: photon tracing
  - Emit photons from light sources
  - Trace them through scene
  - Store them in photon map data structure
- More details:
  - When a photon hits a surface (that has a diffuse component), store the photon in photon map
  - Then use Russian roulette to find out whether the photon is absorbed or reflected
  - If reflected, then shoot photon in new random direction

Photon tracing



- Should not store photon at purely specular surfaces, because these effects are fully view dependent
  - only diffuse effect is view independent
- At hit, photon gets colored (looses intensity)
  - E.g., white photon (1,1,1) becomes pink (0.8, 0.5, 0.5), so looses intensity.
  - Instead of decreasing intensity, decrease probability of further scatter the photon. (E.g., probability of absorbing photon ≈ 0.4)
  - Why not just decrease the photon's intensity?
    - Harder to get good filtering by expanding spheres

The photon map data structure

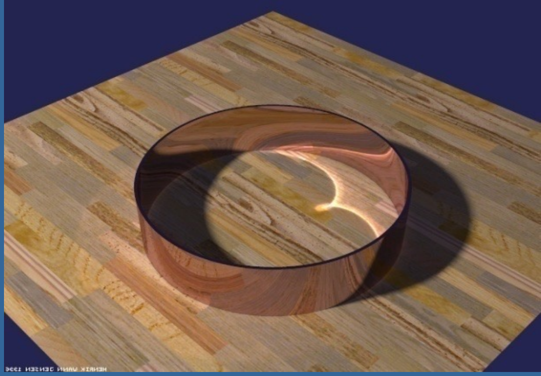
- Keep them in a separate (from geometry) structure
- Store all photons in kD-tree
  - Essentially an axis-aligned BSP tree, since we must alter splitting axis: x,y,z,x,y,z,x,y,z, etc.
  - Each node stores a photon
  - Needed because the algorithm needs to locate the n closest photons to a point
- A photon:
  - float x,y,z;
  - char power[4]; // essentially the color, with more accuracy
  - char phi,theta; // compact representation of incoming direction
  - short flag; // used by KD-tree (stores which plane to split)
- Create balanced KD-tree – simple, done once.
- Photons are stored linearly in memory:
  - Parent node at index: p
  - Left child at: 2p , right child: 2p+1

Locate n closest photons

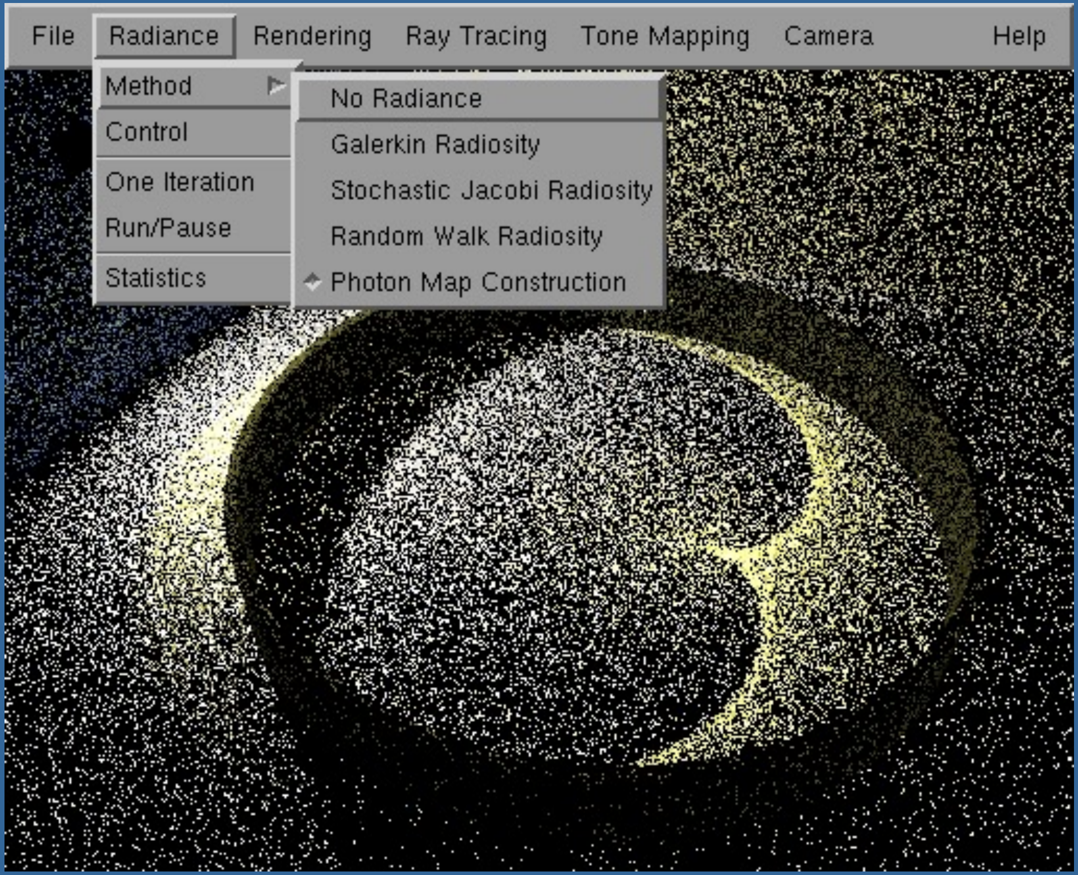
After Henrik Wann Jensen

```
// locate n closest photons around point "pos"
// call with "locate_photons(1)", i.e., with the root as in argument
locate_photons(p)
{
        if(2p+1 < number of photons in photon map structure)
        {       // examine child nodes
                delta=signed distance to plane of node n
                if(delta<0)
                {       // we're to the "left" of the plane
                        locate_photons(2p);
                        if(delta*delta < d*d)
                                locate_photons(2p+1); //right subtree
                }
                else
                {       // we're to the "right" of the plane
                        locate_photons(2p+1);
                        if(delta*delta < d*d)
                                locate_photons(2p);  // left subtree
                }
        }
        delta=real distance from photon p to pos
        if(delta*delta < d*d)
        {       // photon close enough?
                insert photon into priority queue h
                d=distance to photon in root node of h
        }
}
// think of it as an expanding sphere, that stops exanding when n closest
// photons have been found
```

What does it look like?



- Stored photons displayed:



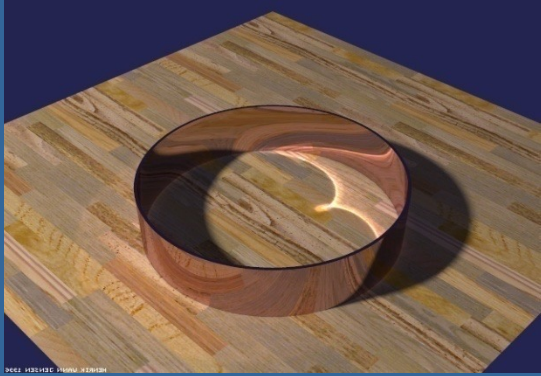
Density estimation

- The density of the photons indicate how much light that point receives
- Radiance is the term for what we display at a pixel
- Complex derivation skipped (see Jensen's book)…
- Reflected radiance at point x:

$$L(\mathbf{x},\boldsymbol{\omega}) \approx \frac{1}{\pi r^2} \sum_{1}^{n} f_r(\mathbf{x},\boldsymbol{\omega}_p,\boldsymbol{\omega}) \Phi_p(\mathbf{x},\boldsymbol{\omega}_p)$$

- L is radiance in $\mathbf{x}$ in the direction of $\mathbf{w}$
- r is radius of expanded sphere
- ω_p is the direction of the stored photon
- Φ_p is the stored power of the photon
- f_r is the BRDF

Two-pass algorithm

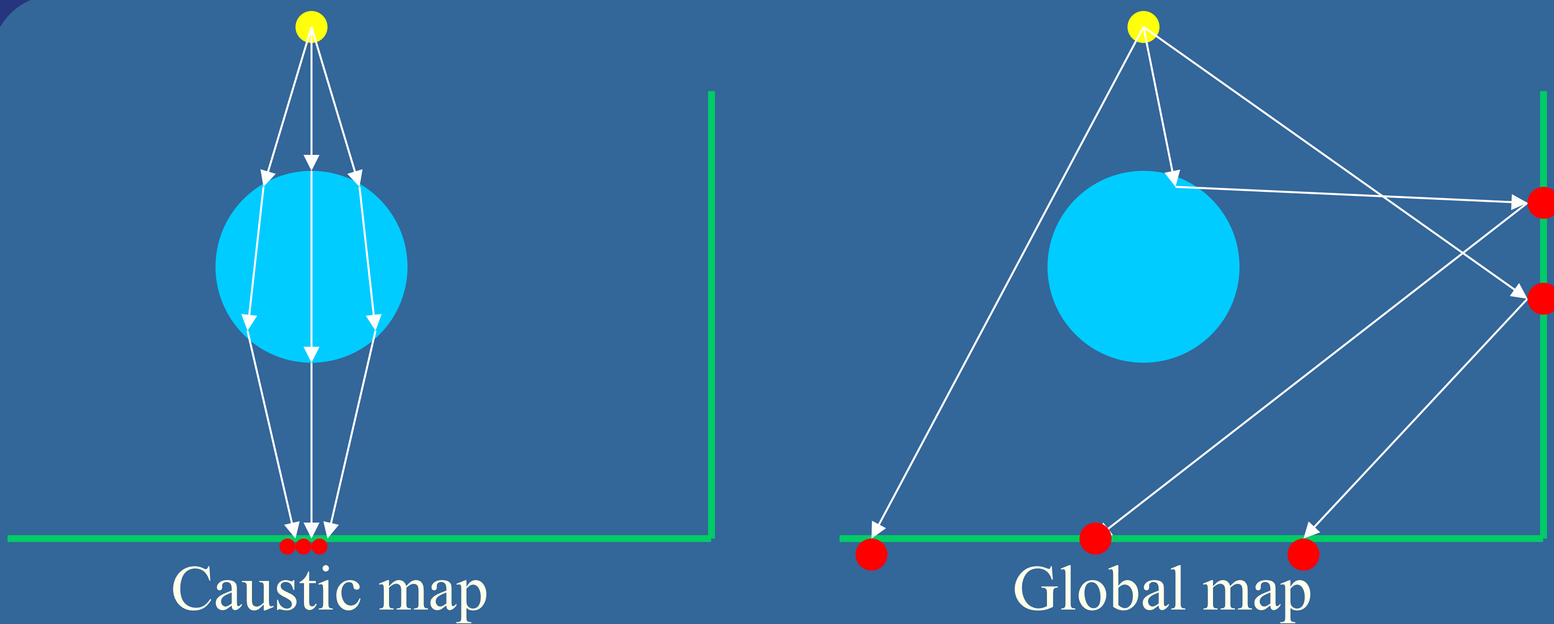


- Already said:
  - 1) Photon tracing, to build photon maps
  - 2) Rendering from the eye + using photon maps
- Pass 1 (create photon maps):
  - Use two photon maps
  - A caustics photon map (for caustics)
    - Stores photons that have been reflected or refracted (via a specular/transparent surface) to a diffuse surface
    - (Light transport notation: LS+D)

+

  - A global photon map (for all illumination)
    - All photons that landed on diffuse surfaces
    - L(S | D)*D

Caustic map and global map



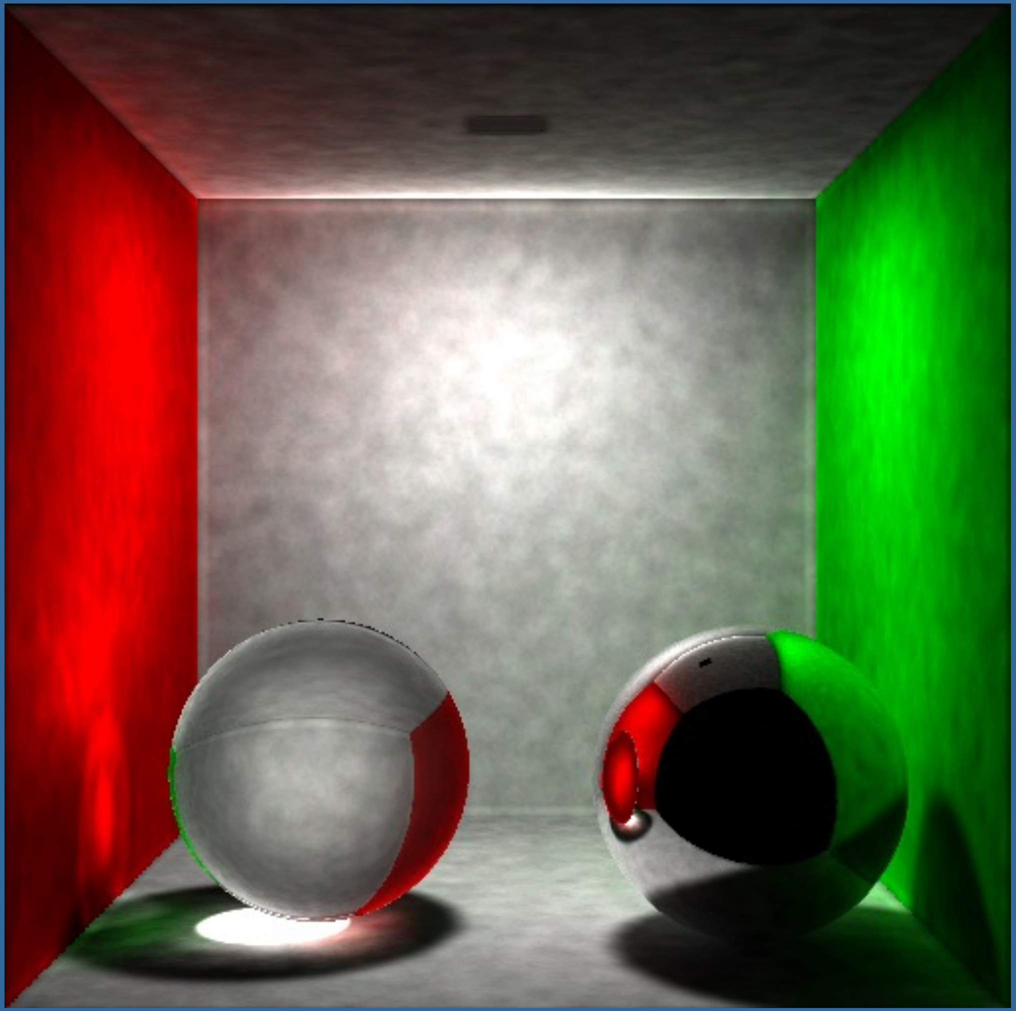
- Caustic map: send photons only towards reflective and refractive surfaces
  - Caustics is a high frequency component of illumination
  - Therefore, need many photons to represent accurately
- Global map - assumption: illumination varies more slowly

Pass 2: Rendering using the photon map

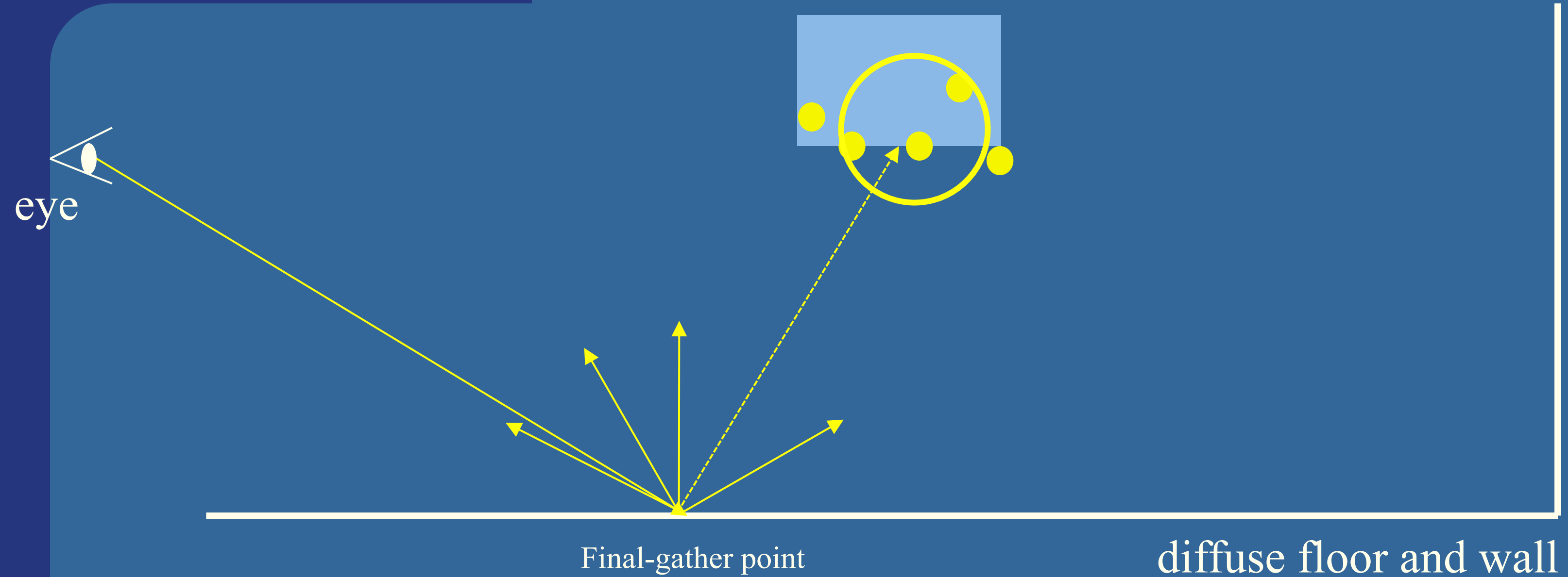
- Render from the eye using a modified ray tracer
  - A number of rays are sent per pixel
  - For each ray, evaluate four terms
    - **Direct illumination** (light that reaches a surface directly from light source)… may need to send many rays to area lights. Done using standard ray tracing.
    - **Specular reflection** (also evaluted using ray tracing, possibly with many rays sent around the reflection direction)
    - **Caustics**: use caustics photon map
    - **Indirect illumination**: use the global photonmap
      - Or *Final Gather* + global photon map...

Example of noise when using the photon maps for the primary rays

- Ugly noise:
- Solution:
  - for the primary rays: use Final Gather instead of using the global photon map



A modification for indirect Illumination – Final Gather



- Too noicy to use the global map for direct visualization
- Remember: eye rays are recursively traced (via reflections/refractions) until a diffuse hit, **p**. There, we want to estimate slow-varying indirect illumination.
  - Instead of growing sphere in global map at **p**, Final Gather shoots 100-1000 indirect rays from **p** and grows sphere in the global map and also caustics map, where each of those rays end at a diffuse surface. Or interpolate from nearby already computed final-gather points.

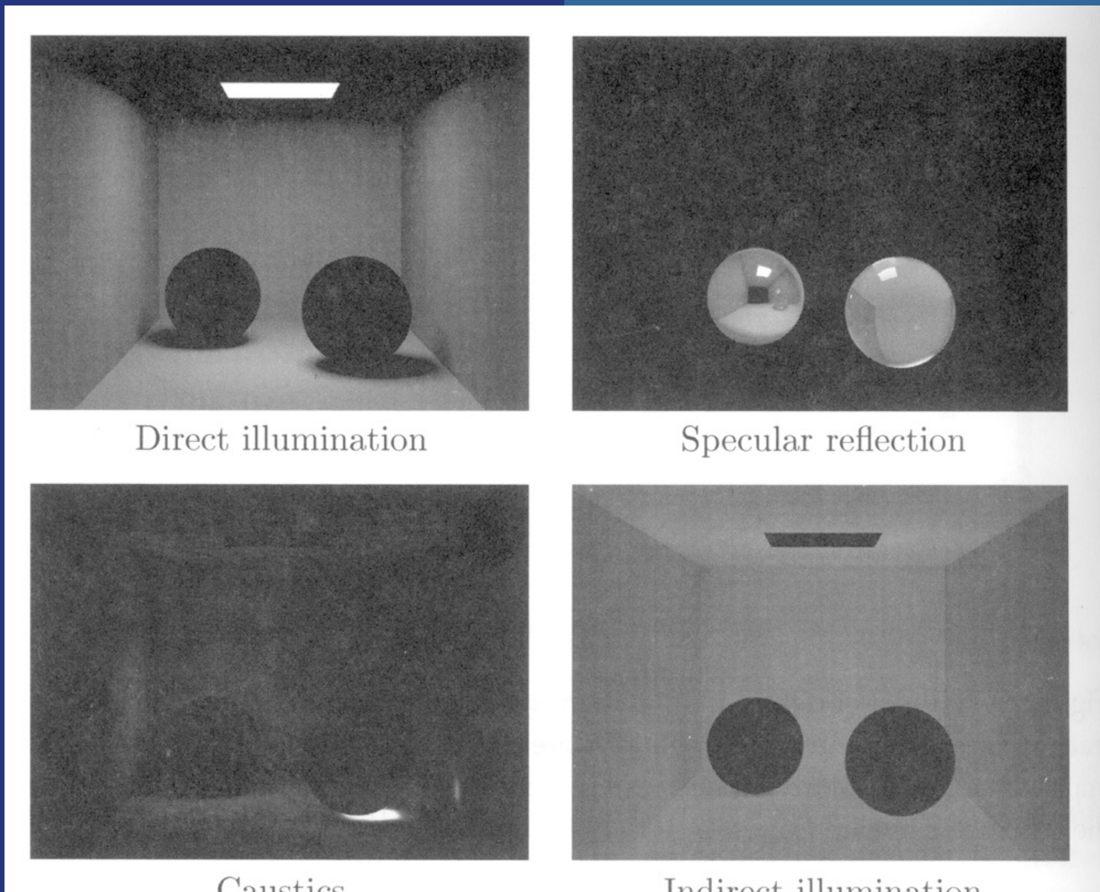
Final Gather for Photon Mapping

- Final gathering is a technique for estimating global illumination for a given point by either sampling a number of directions in the hemisphere over that point (such a sample set is called a *final gather point*), or by averaging a number of final gather points nearby since final gather points are too expensive to compute for every illuminated point.
- For diffuse hits, final gathering often improves the quality of the global illumination solution. Without final gathering, the global illumination on a diffuse surface is computed by estimating the photon density (and energy) near that point. With final gathering, many new rays are sent out to sample the hemisphere above the point to determine the incident illumination. Some of these rays hit diffuse surfaces; the global illumination at those points is then computed by the material shaders at those sample point, using illumination from the photon maps and other material properties. Other rays hit specular surfaces and do not contribute to the final gather color (since that type of light transport is a secondary caustic). Tracing many rays (each with photon-map lookups) is very time-consuming so it is only done when necessary – in most cases, interpolation and extrapolation from previous nearby final gatherings is sufficient.

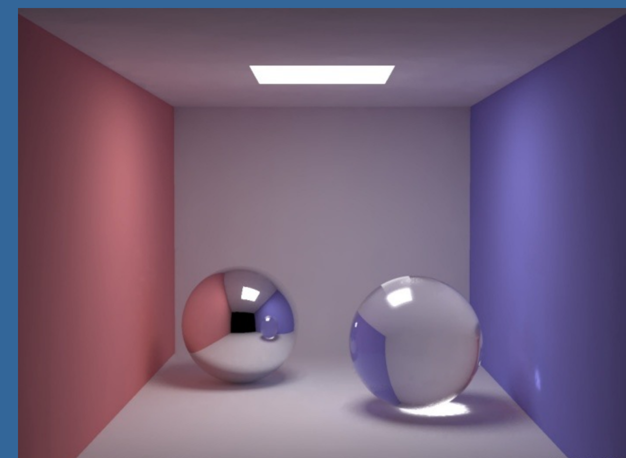
Photon Mapping + AI Denoising

- Or use AI denoising instead of Final Gathering:

Images of the four components



=



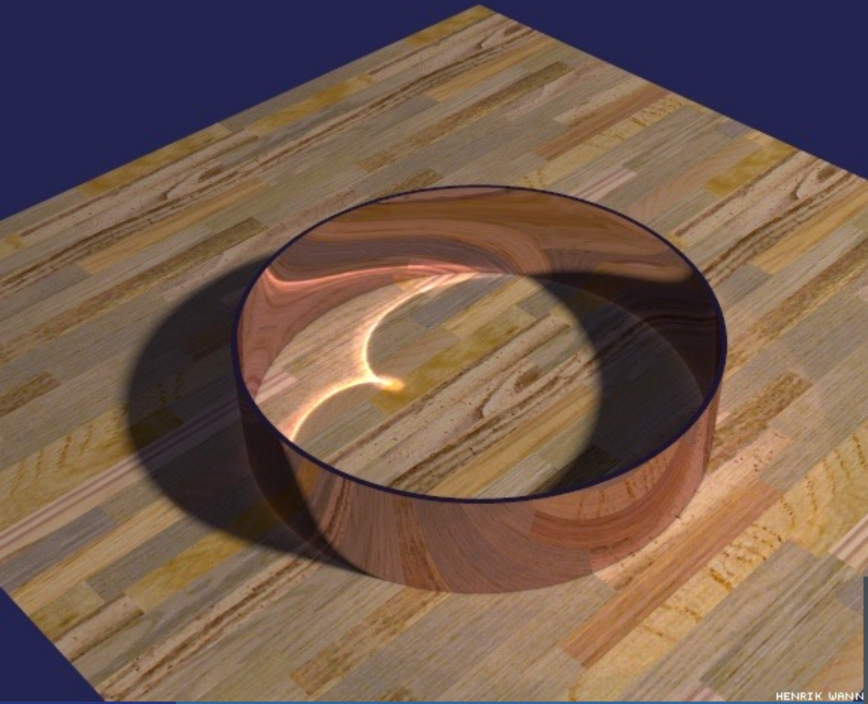
- These together solves the entire rendering equation!

Image scanned from Henrik Wann Jensen's book

Photon Mapping - Summary

- Creating Photon Maps:
  - Trace photons (~100K-1M) from light source. Store them in kd-tree when they hit diffuse surface. Then, use russian roulette to decide if the photon should be absorbed or specularly or diffusively reflected. Create both global map and caustics map. For the Caustics map, we send more of the photons towards reflective/refractive objects.
- Ray trace from eye:
  - As usual: I.e., shooting primary rays and recursively shooting reflection/refraction rays, and at each intersection point **p**, compute direct illumination (shadow rays + shading).
  - Also grow sphere around each **p** in caustics map to get caustics contribution and in global map to get slow-varying indirect illumination.
  - If final gather is used: At the first diffuse hit, instead of using global map directly, sample indirect slow varying light around **p** by sampling the hemisphere with ~100 – 1000 rays and use the two photon maps where those rays hit a surface. Or interpolate from nearby final-gather points.
- Growing sphere:
  - Uses the kd-tree to expand a sphere around **p** until a fixed amount (e.g. 50) photons are inside the sphere. The radius is an inverse measure of the intensity of indirect light at **p**. The BRDF at **p** could also be used to get a more accurate color and intensity value.

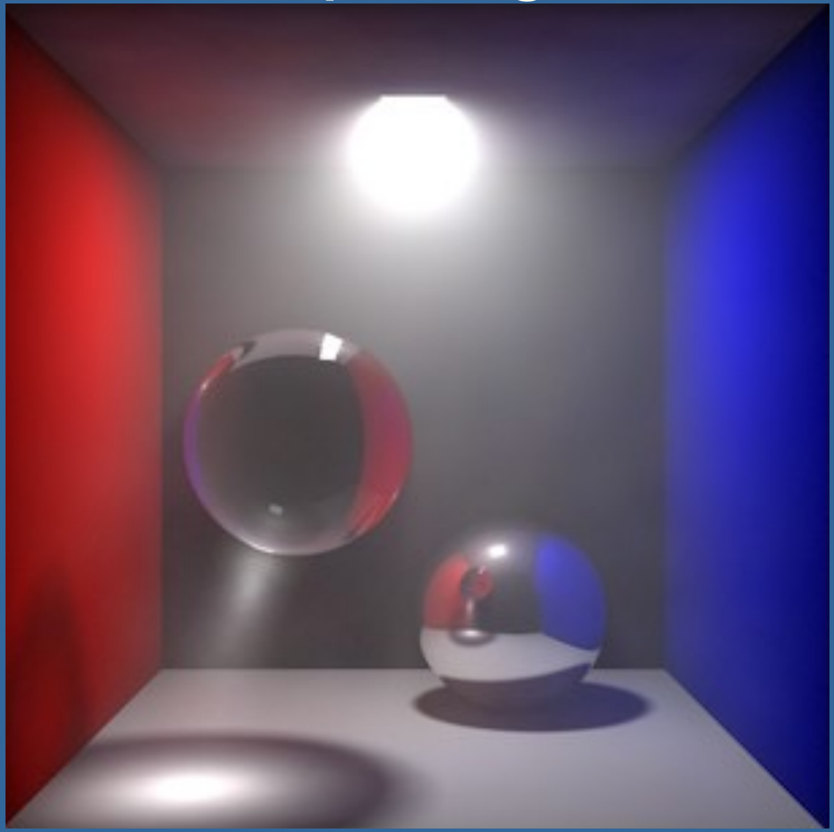
Standard photon mapping

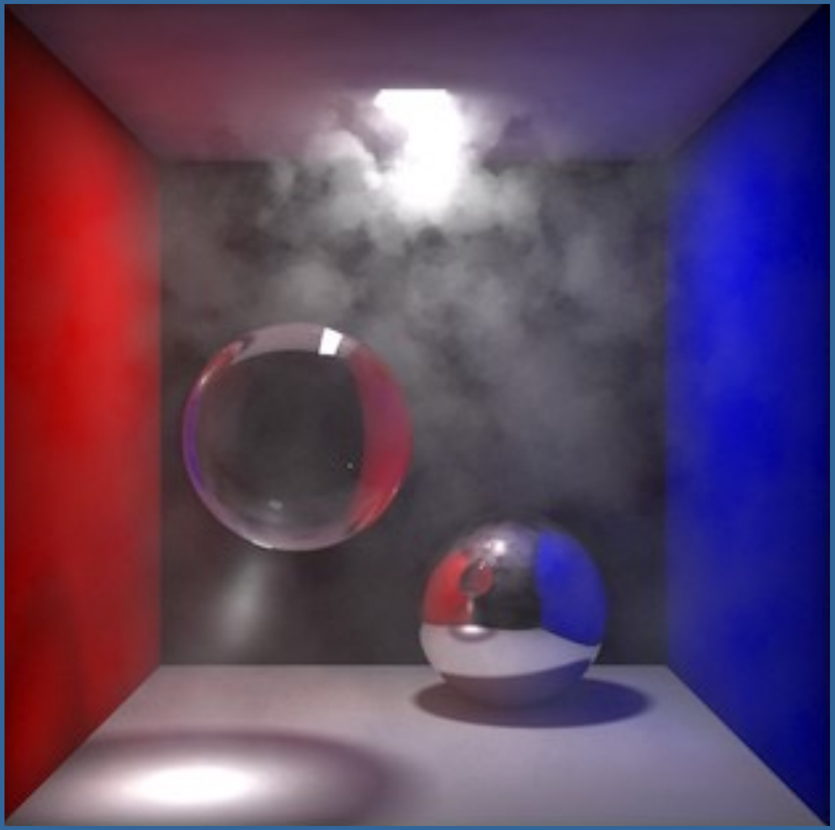


Caustics: concentrated reflected or refracted light

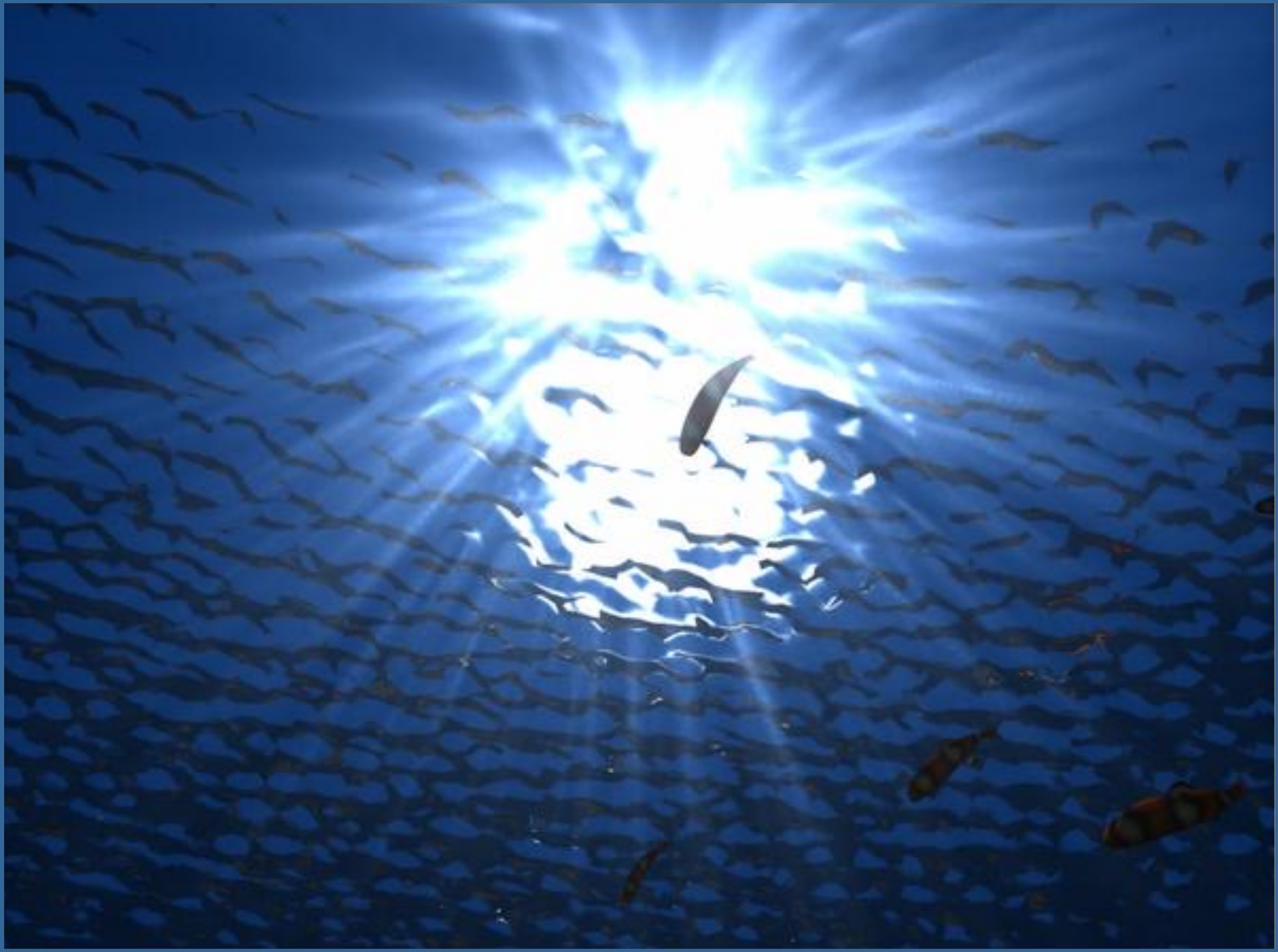
Extensions to photon mapping

- Participating media

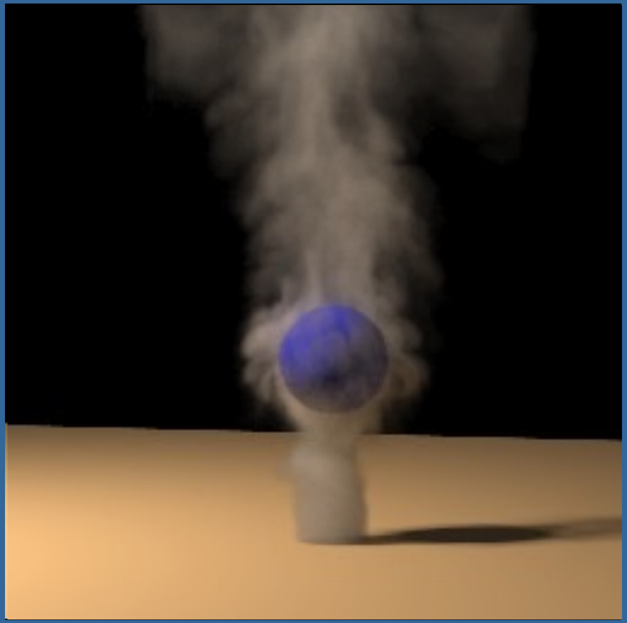




Another one on participating media



Smoke and photon mapping



Press for a movie

In conclusion

- If you want to get global illumination effects, then implement a path tracer
  - Simple to implement
  - Good results
  - Disadvantage: rendering times (many many rays per pixel). Advantage: fast preview.
- If you want a more advanced renderer:
  - Extend with bidirectional path tracing
  - Or Metropolis Light Transport, or Photon Mapping

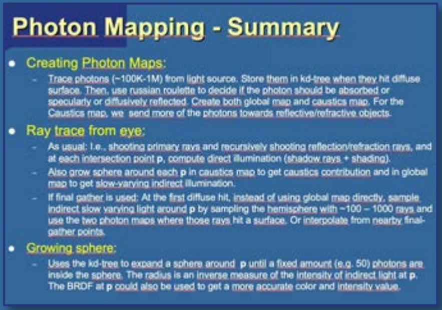
THE END

What you need to know

- The rendering equation + BRDF
  - Be able to explain all its components

$$L_o = L_e + \int_{\Omega} f_r(\mathbf{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\mathbf{x}, \boldsymbol{\omega}') (\boldsymbol{\omega}' \cdot \mathbf{n}) d\boldsymbol{\omega}'$$

- Monte Carlo sampling:
  - The naïve way (an exponentially growing ray tree)
  - Path tracing
    - Why it is good, compared to naive monte-carlo sampling
    - The overall algorithm (on a high level as in these slides).
  - Photon Mapping
    - The overall algorithm. See the summary slide on:
      - Creating Photon Maps…
      - Ray trace from eye…
      - Growing spheres…
  - Bidirectional Path Tracing, Metropolis Light Transport
    - Just their names. Don’t need to know the algorithms.

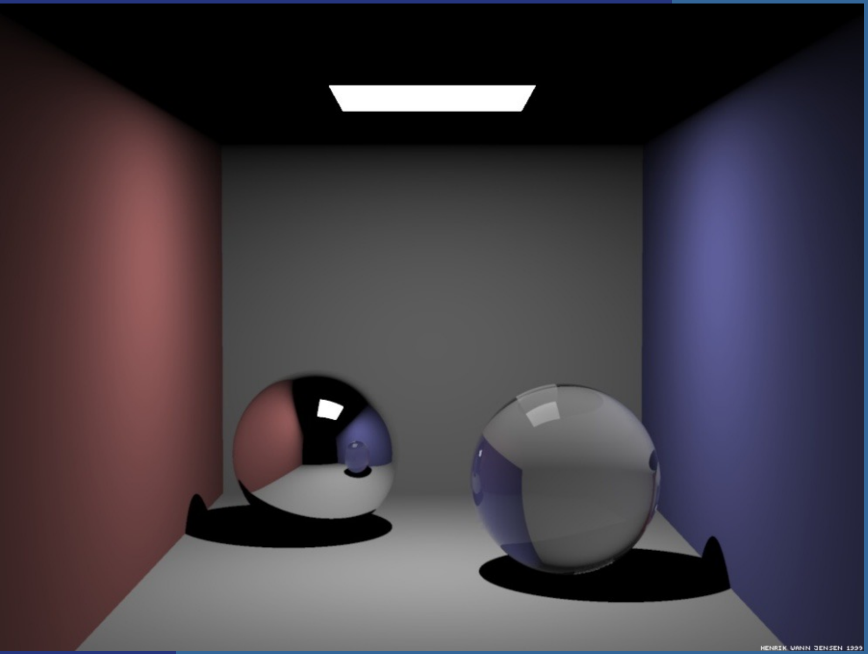


- Denoising by Final Gather or AI
    - Final Gather – sample indirect illumination at some positions in the world (final-gather points). At each ray hit, estimate indirect illumination by interpolation from nearby final-gather points.
    - AI: use some existing Deep Neural Network solution that denoises your images for your kind of scenes.

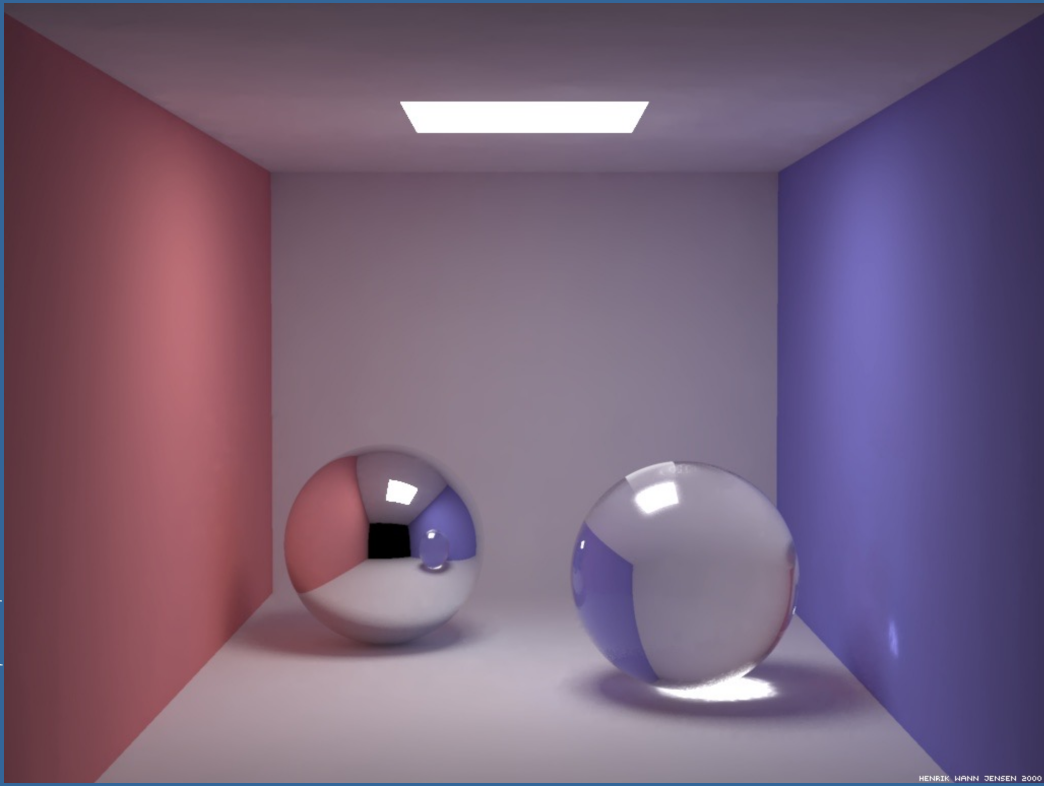
The most important slides →

Isn't ray tracing enough?

Effects to note in Global Illumination image:
1) Indirect lighting (light reaches the roof)
2) Soft shadows (light source has area)
3) Color bleeding (example: roof is red near red wall) (same as 1)
4) Caustics (concentration of refracted light through glass ball)
5) Materials have no ambient component



Ray tracing

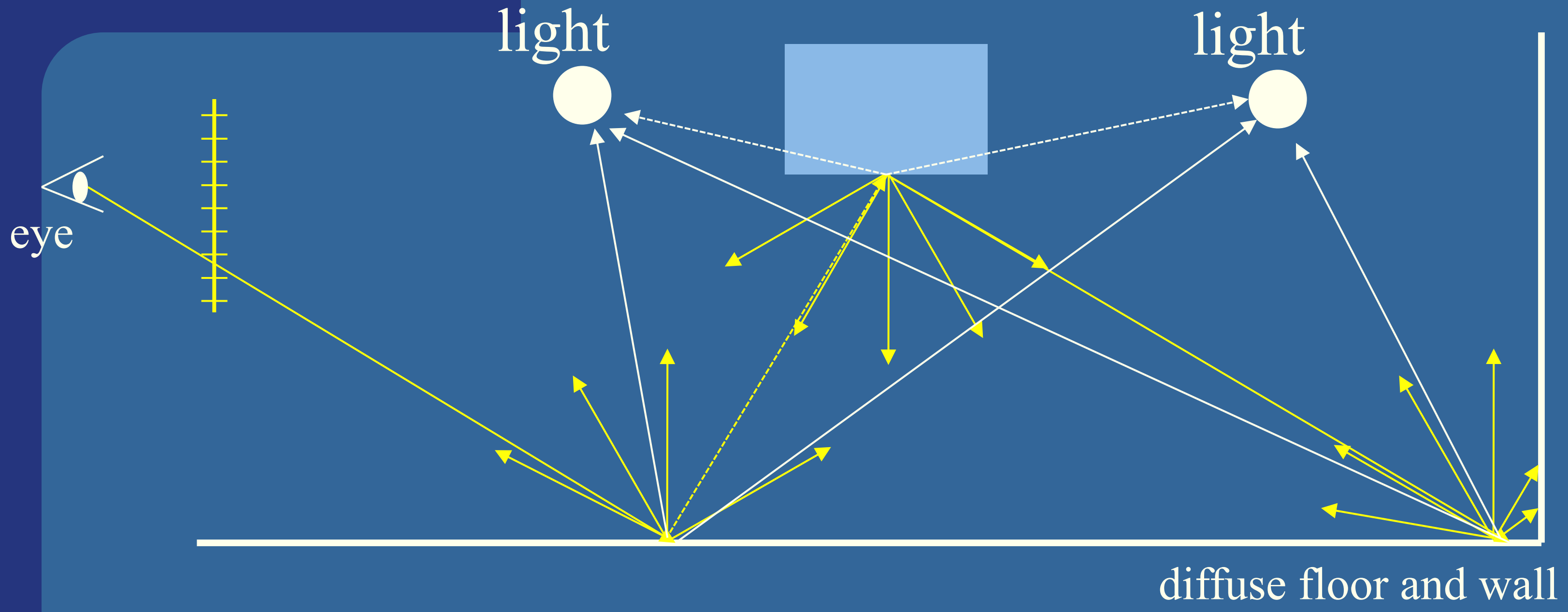


Global Illumination

Which are the differences?

Images courtesy of Henrik Wann Jensen

Monte Carlo Ray Tracing – direct + indirect illumination

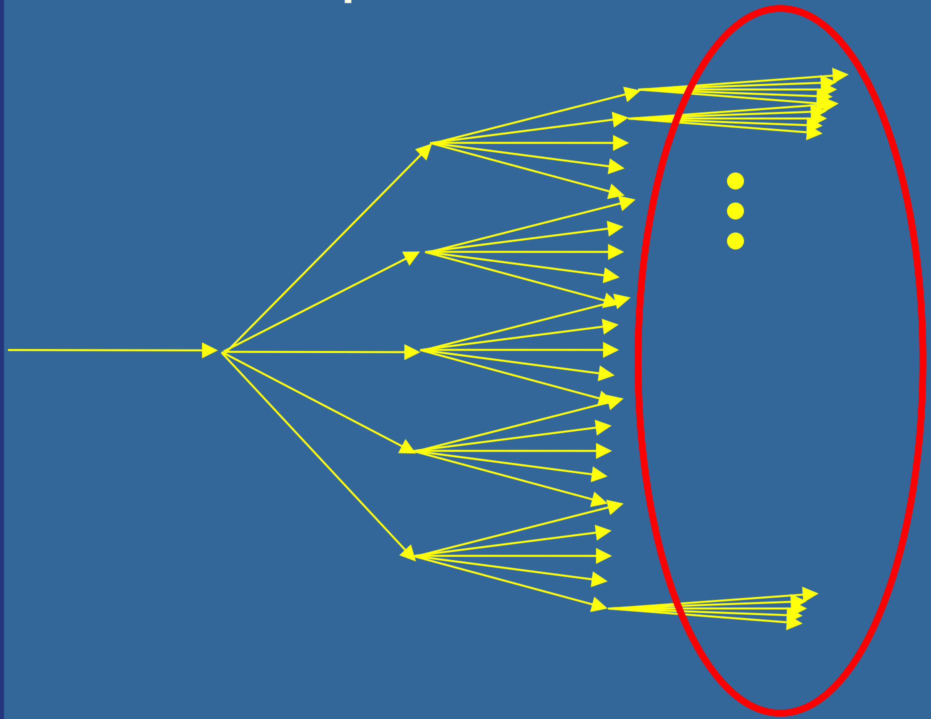


- Sample indirect illumination by shooting sample rays over the hemisphere, at each hit.

$$L_o = L_e + \int_{\Omega} f_r(\mathbf{x}, \boldsymbol{\omega}, \boldsymbol{\omega}') L_i(\mathbf{x}, \boldsymbol{\omega}')(\boldsymbol{\omega}' \cdot \mathbf{n}) d\boldsymbol{\omega}'$$

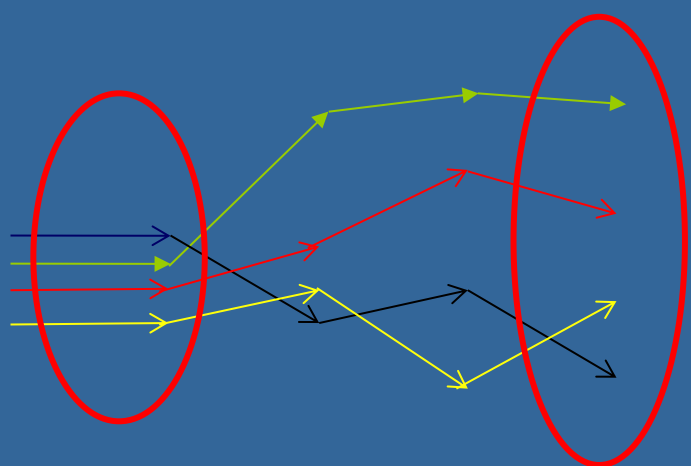
Monte Carlo Ray Tracing (naïvely)

- The indirect-illumination sampling gives a ray tree with most rays at the bottom level. This is bad since these rays have the lowest influence on the pixel color.



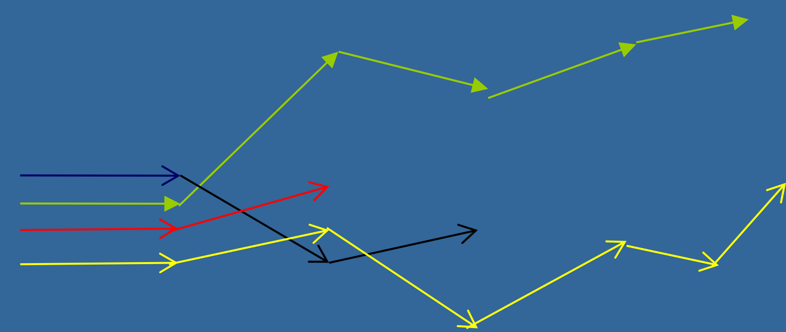
PathTracing
– one efficient Monte-Carlo Ray-Tracing solution

- Path Tracing instead only traces one of the possible ray paths at a time. This is done by randomly selecting only one sample direction at a bounce. Hundreds of paths per pixel are traced.



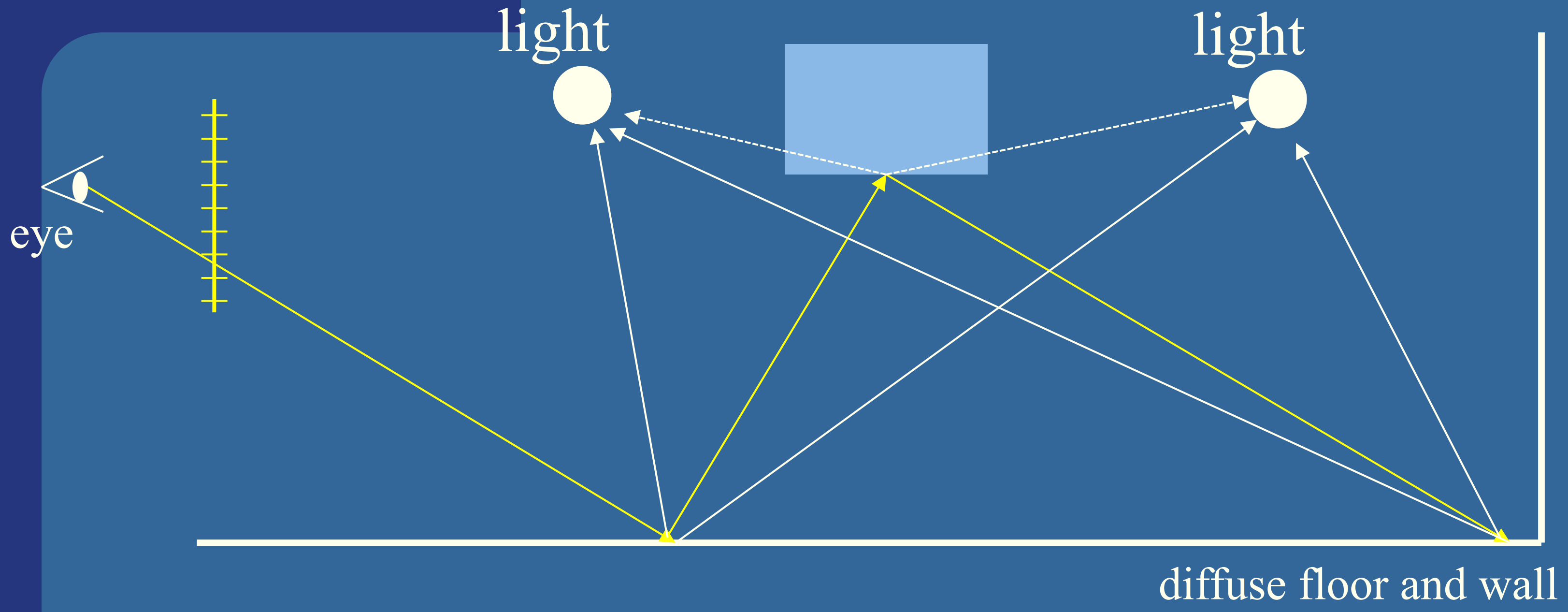
Equally number of rays are traced at each level

Or:



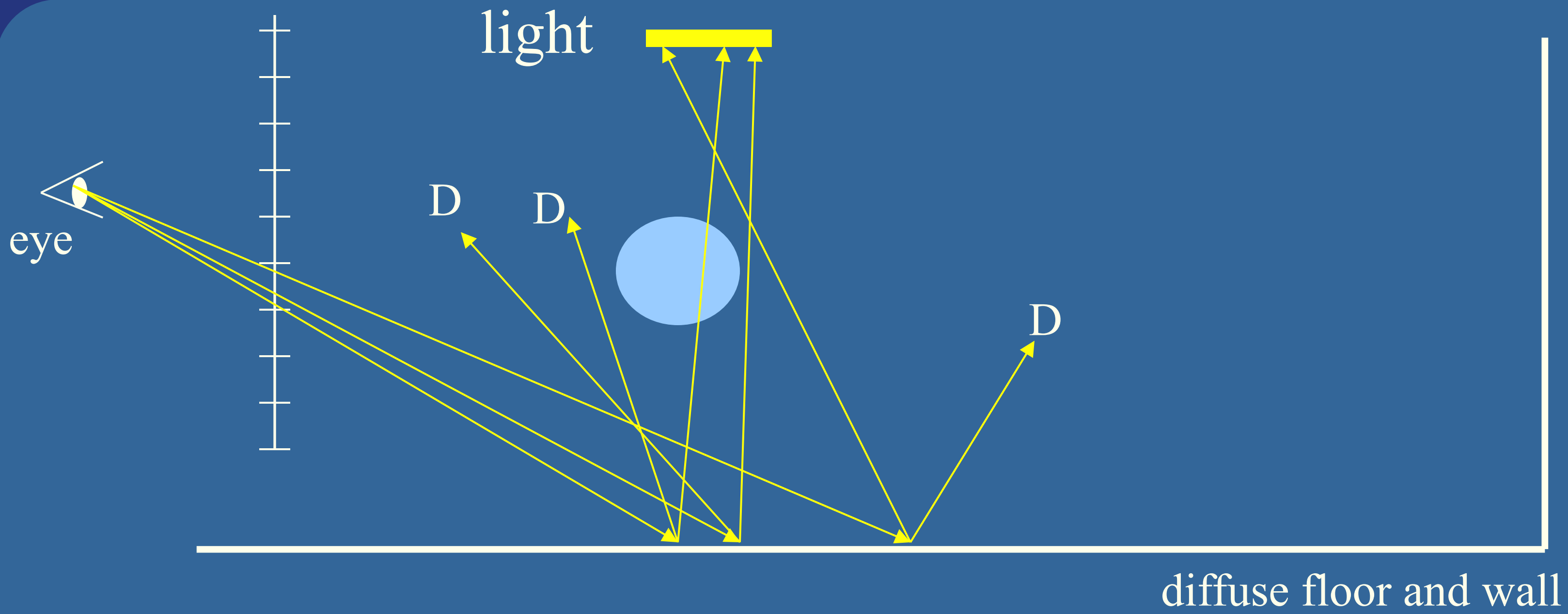
Even smarter: terminate path with some probablility after each level, since they have decreasing importance to final pixel color.

Path Tracing – indirect + direct illumination



- Shoot many paths per pixel (the image just shows one light path).
  - At each intersection,
    - Shoot one shadow ray per light source
      - at random position on light, for area/volumetric light sources
    - and randomly select one new ray direction.

Path tracing with soft shadows (area lights):



- For area lights:
  - For each path, at each intersection
    - Shoot the shadow ray to a random position on the area light source.

For many paths per pixel, this will converge to a soft shadow.

Path tracing: Summary

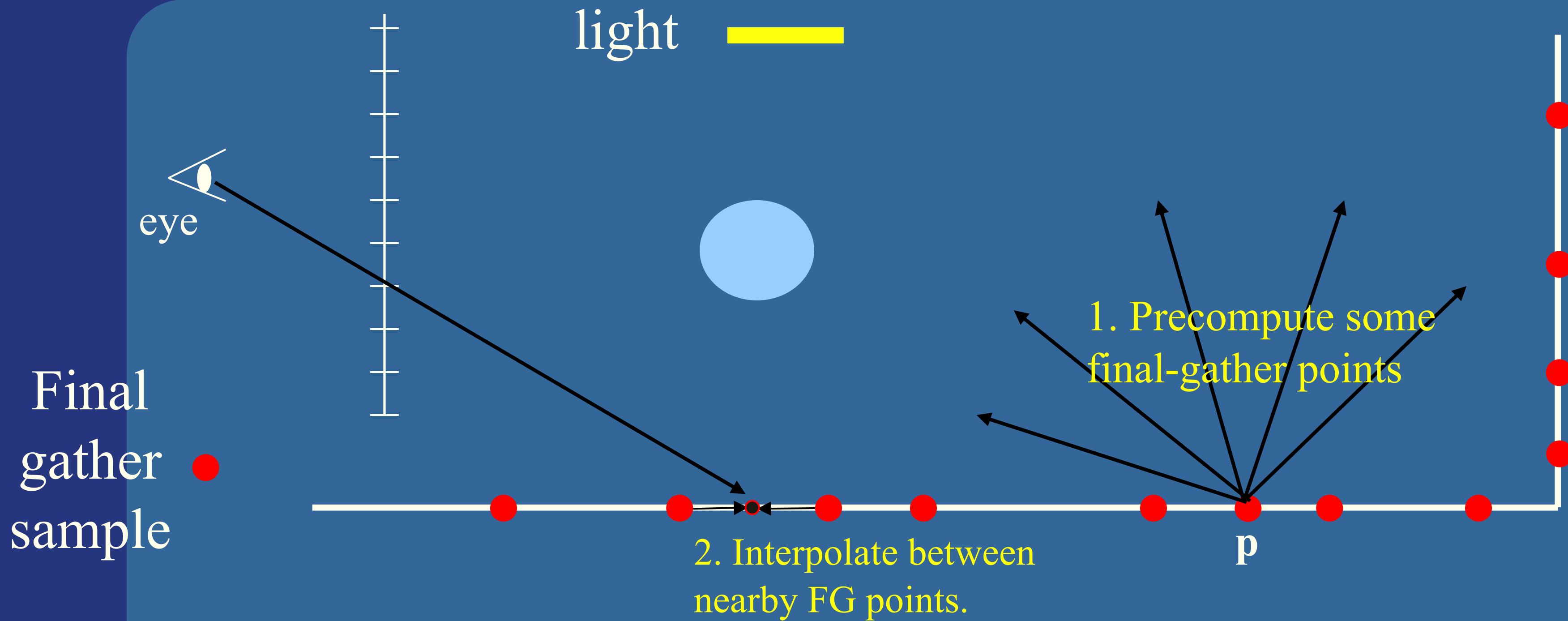
- Uses Monte Carlo sampling to solve integration:
  - by shooting many random ray ***paths*** over the integral domain.
  - Algorithm:
    - For each pixel, // we will shoot a number of paths:
      - For each path, generate the primary ray:
        1. Trace the ray. At hitpoint:
        2. Shoot one shadow ray and compute local lighting.
        3. Sample indirect illumination randomly over the possible reflection/refraction directions by generating **one** such new ray.
        4. Repeat from 1, until the path is randomly terminated (or the ray does not hit anything).
- Shorter summary: shoot many paths per pixel, by randomly choosing **one** new ray at each interaction with surface + **one** shadow ray per light. Terminate the path with a random probability

Final Gather

Popular for ray tracing and photon mapping but not path tracing

Idea and good answer:

- Compute indirect illumination somehow, but only at a few positions (final gather points) in the scene.
- Estimate indirect illumination for other positions by interpolation from nearby final-gather points



- Many versions of Final Gathering exist.
- E.g., to compute final-gather point **p**:
  - Send hundreds of random rays out from **p** to sample indirect illumination
- To use during ray tracing: interpolate global illumination between nearby Final Gather points, to estimate incoming radiance at the ray's intersection point.

Photon Mapping - Summary

- Creating Photon Maps:
  - Trace photons (~100K-1M) from light source. Store them in kd-tree when they hit diffuse surface. Then, use russian roulette to decide if the photon should be absorbed or specularly or diffusively reflected. Create both global map and caustics map. For the Caustics map, we send more of the photons towards reflective/refractive objects.
- Ray trace from eye:
  - As usual: I.e., shooting primary rays and recursively shooting reflection/refraction rays, and at each intersection point **p**, compute direct illumination (shadow rays + shading).
  - Also grow sphere around each **p** in caustics map to get caustics contribution and in global map to get slow-varying indirect illumination.
  - If final gather is used: At the first diffuse hit, instead of using global map directly, sample indirect slow varying light around **p** by sampling the hemisphere with ~100 – 1000 rays and use the two photon maps where those rays hit a surface. Or interpolate from nearby final-gather points.
- Growing sphere:
  - Uses the kd-tree to expand a sphere around **p** until a fixed amount (e.g. 50) photons are inside the sphere. The radius is an inverse measure of the intensity of indirect light at **p**. The BRDF at **p** could also be used to get a more accurate color and intensity value.