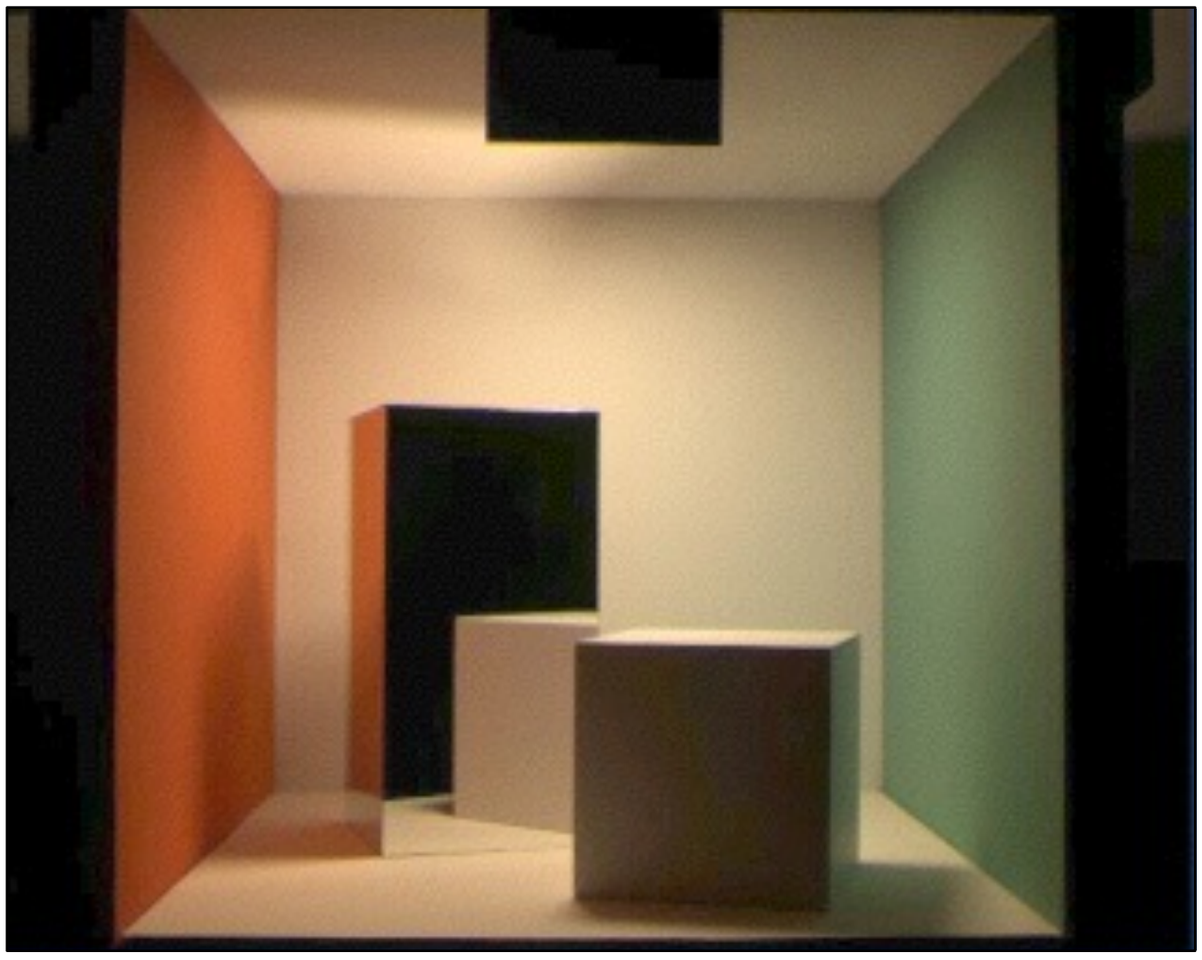
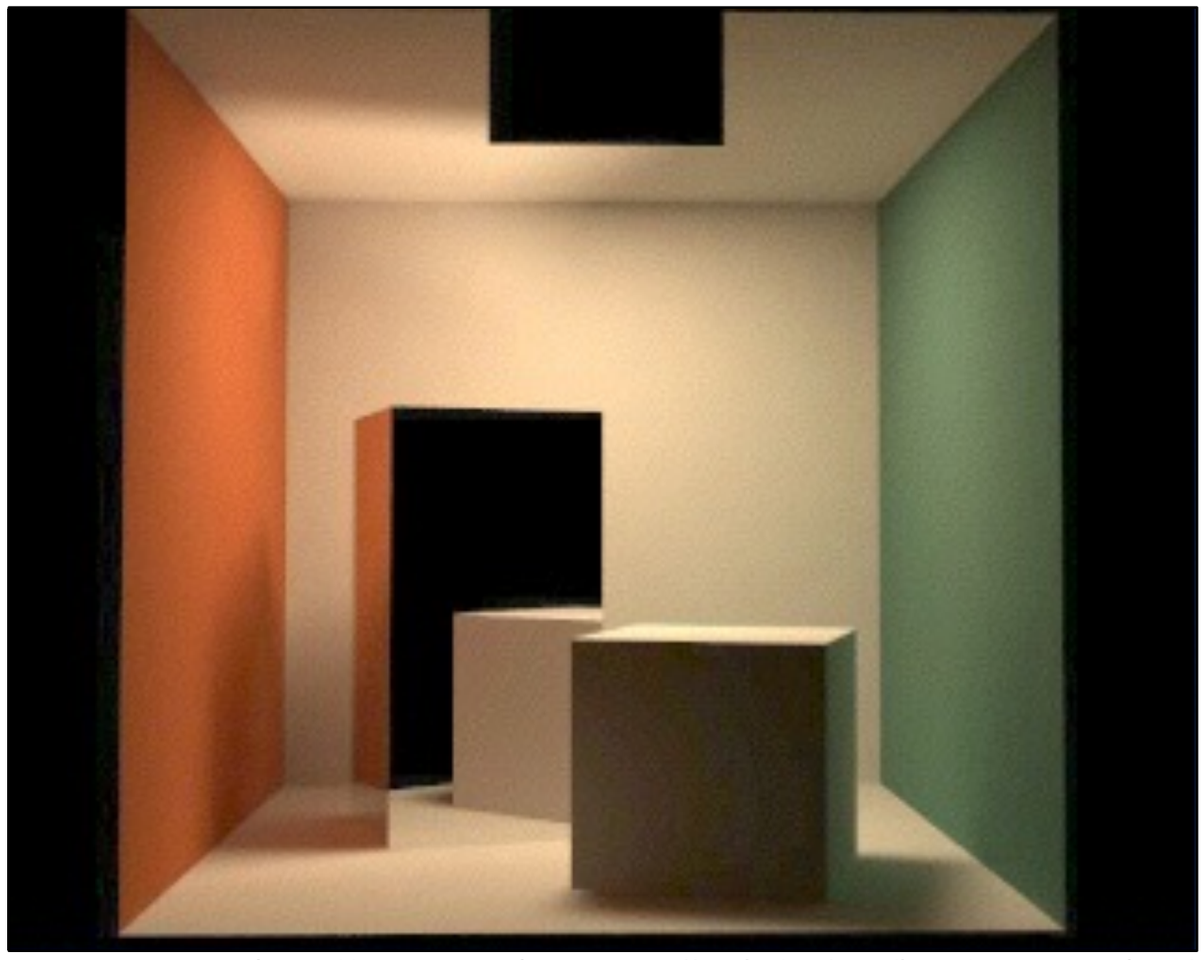
CS 87/187, Spring 2016

Rendering Algorithms

Global Illumination I: Path Tracing





http://www.graphics.cornell.edu/online/box/compare.html

Prof. Wojciech Jarosz

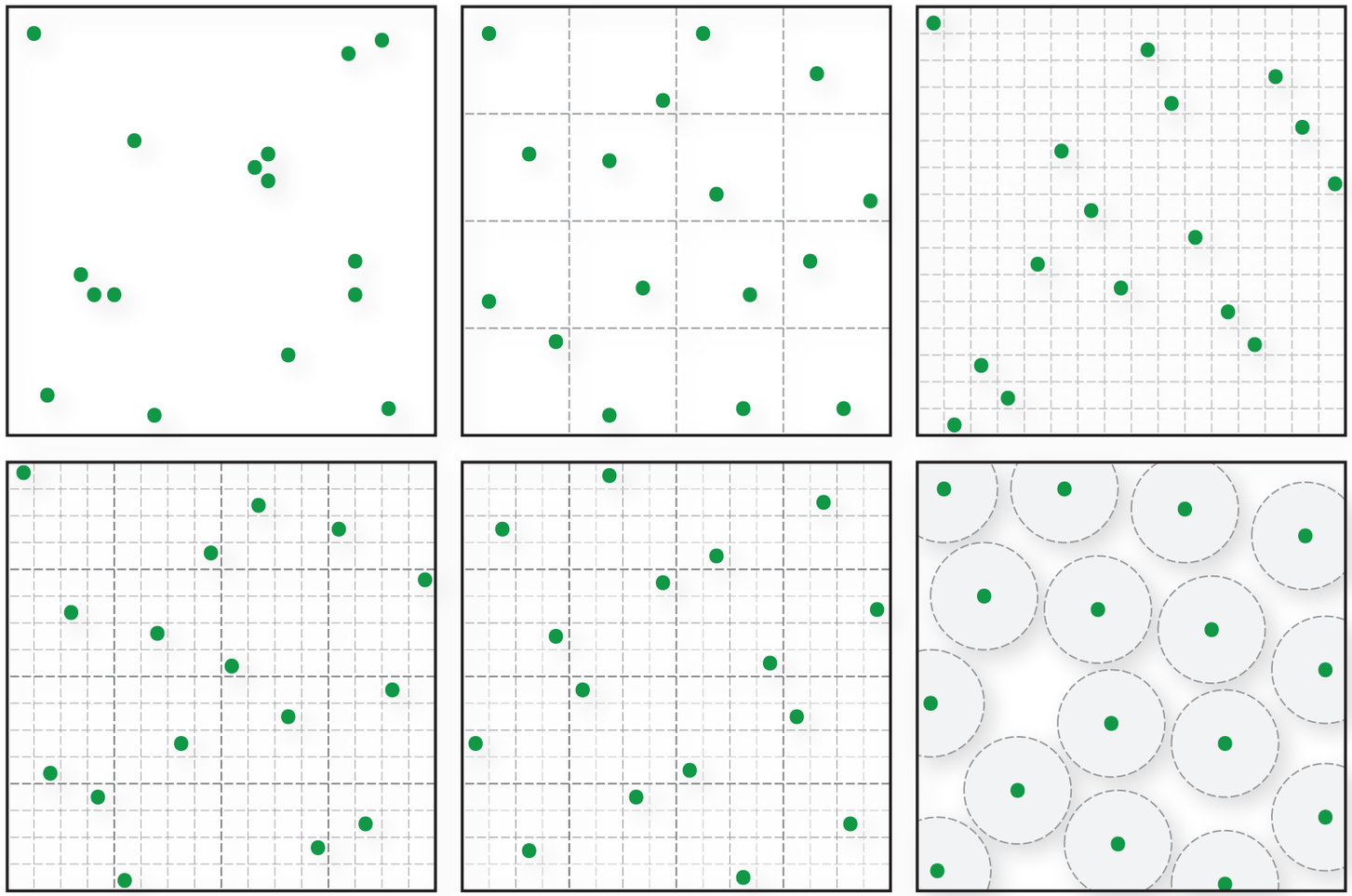
wojciech.k.jarosz@dartmouth.edu

(with some slides by Jan Novák and Wenzel Jakob)

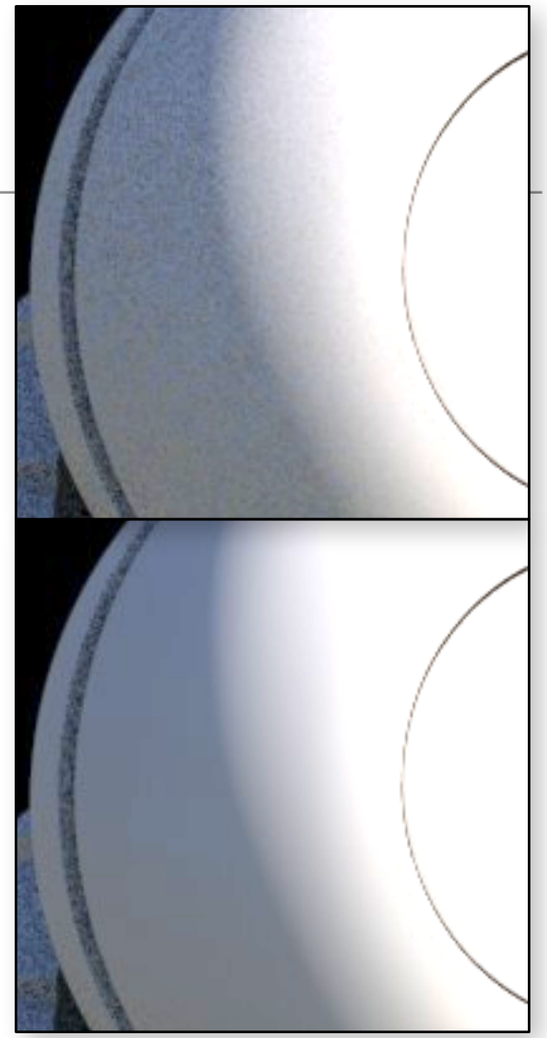
Dartmouth

Last time

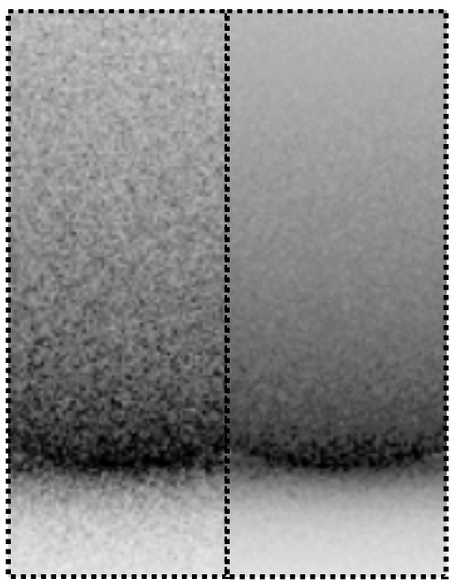
- Variance reduction using
  - Control variates
  - Careful sample placement

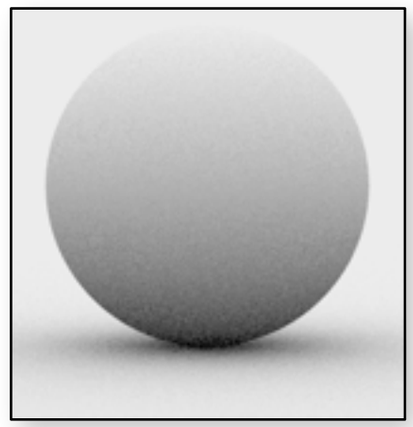


Control variates off



Control variates on

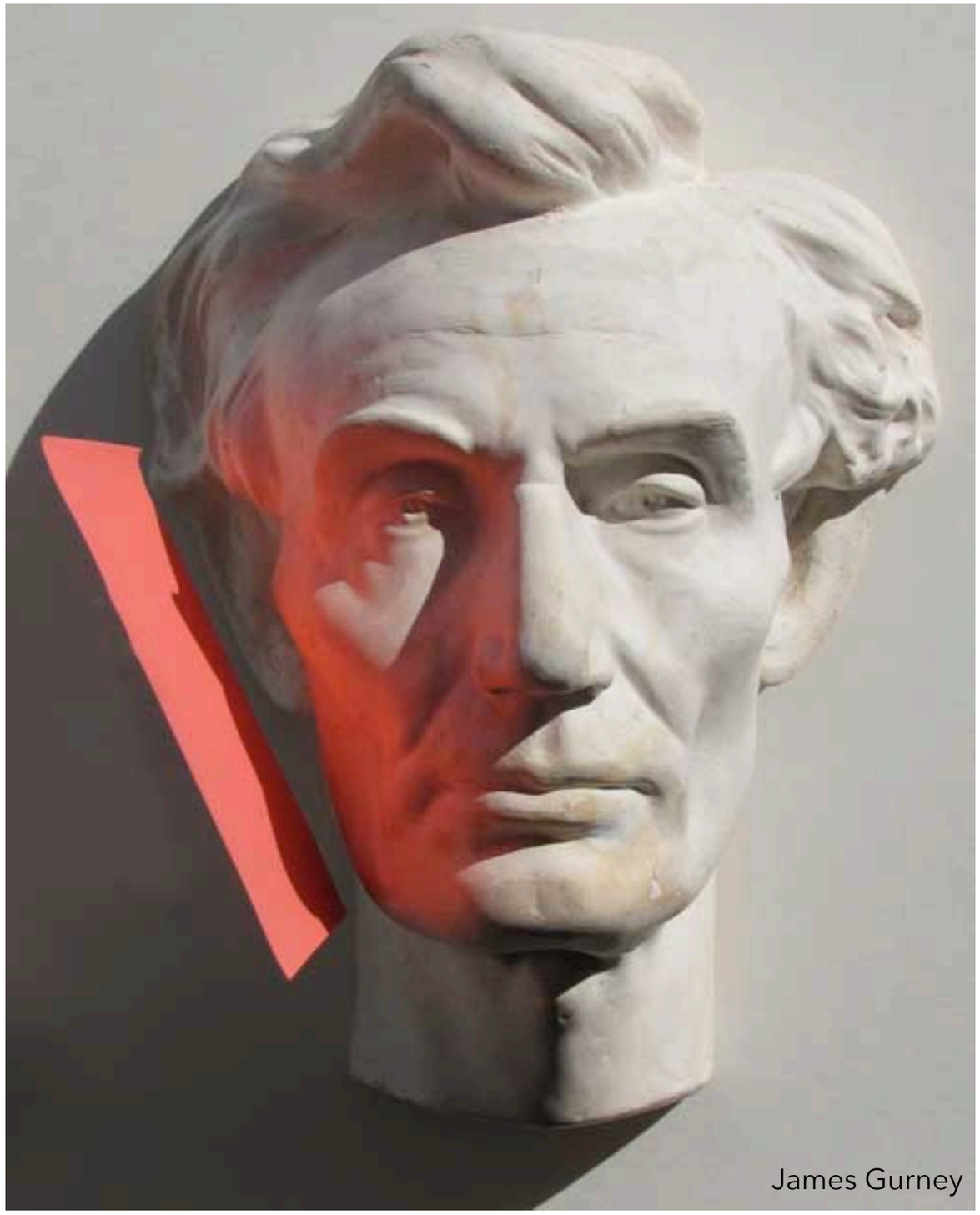




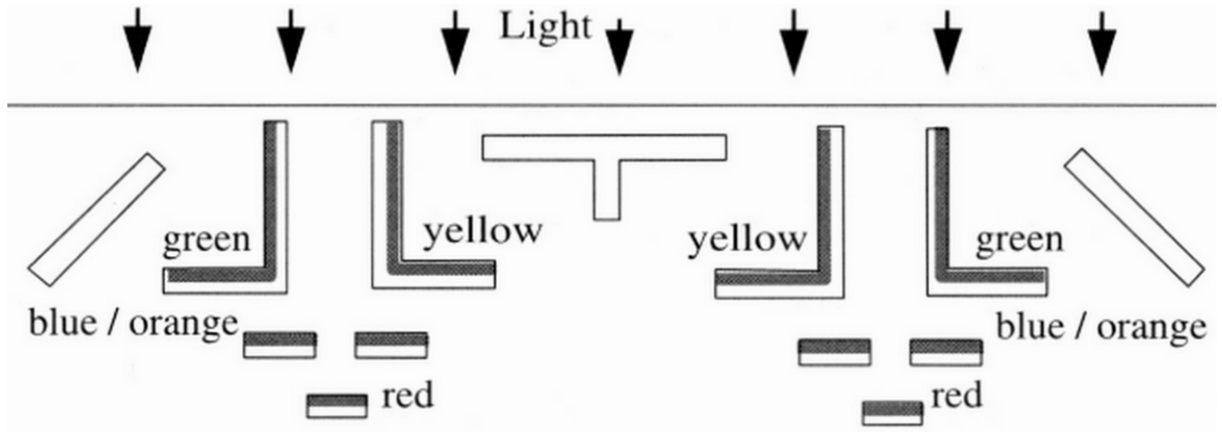
Today's Menu

- Light paths
- Heckbert notation
- Rendering Equation
- Solving the Rendering Equation
  - Recursive Monte Carlo ray tracing
  - Path tracing

Color Bleeding



Color Bleeding



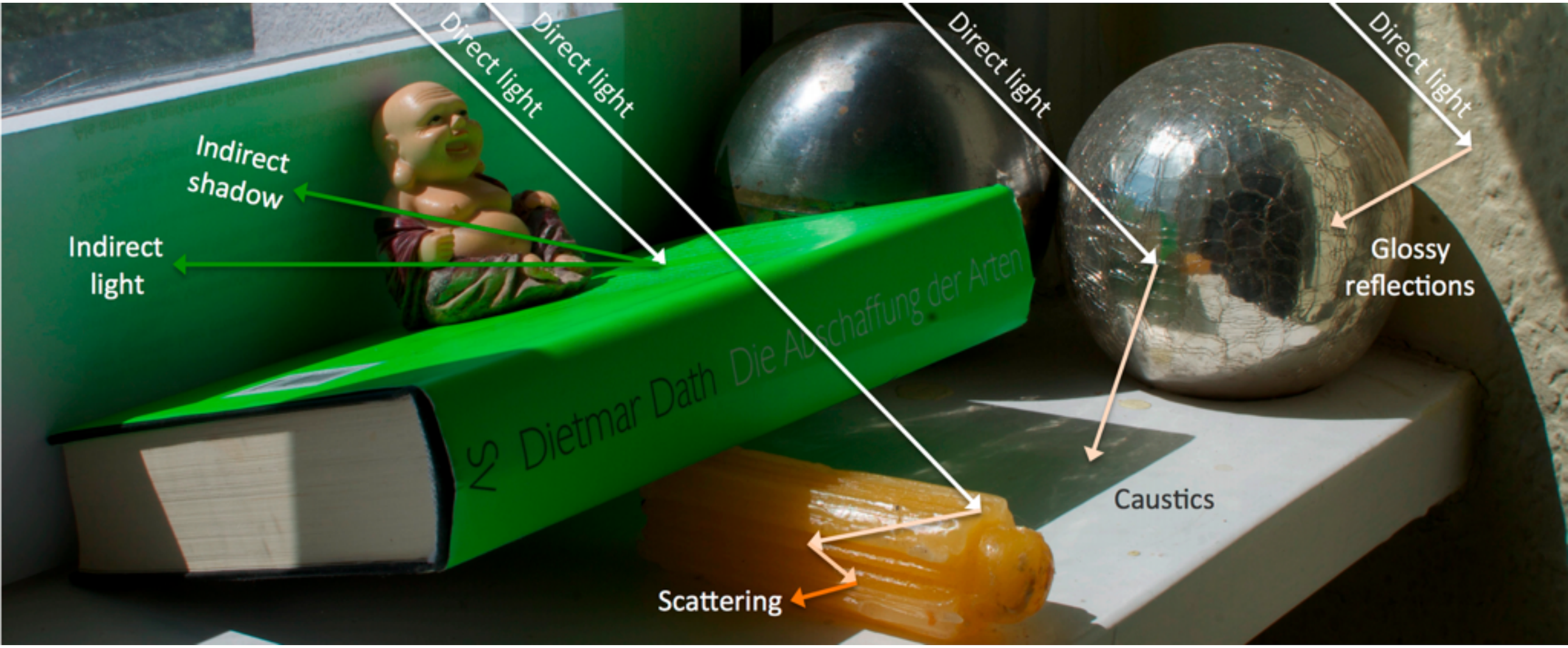
Sculpture by
John Ferren

All visible surfaces are painted white, the color is only due to inter-reflections

Caustics

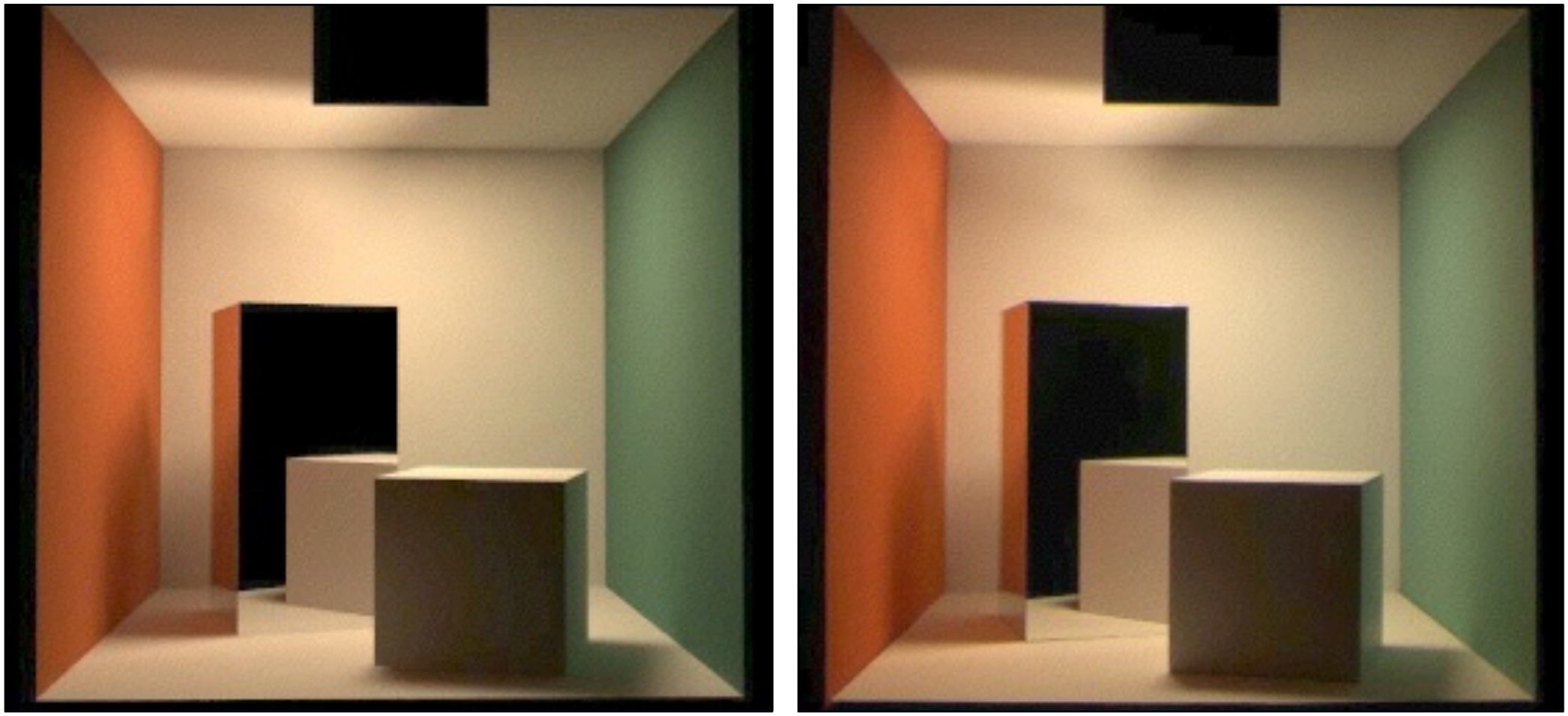
Subsurface Scattering

All-in-One!



Ritschel et al. [2012]

Simulating Global Illumination



http://www.graphics.cornell.edu/online/box/compare.html

One image here was captured from a physical model under controlled lighting conditions using a Photometrics CCD camera. The other image was rendered using a geometric model with material properties and lighting set to values identical to the physical conditions.
Which one is which?

Light Paths

Light Paths

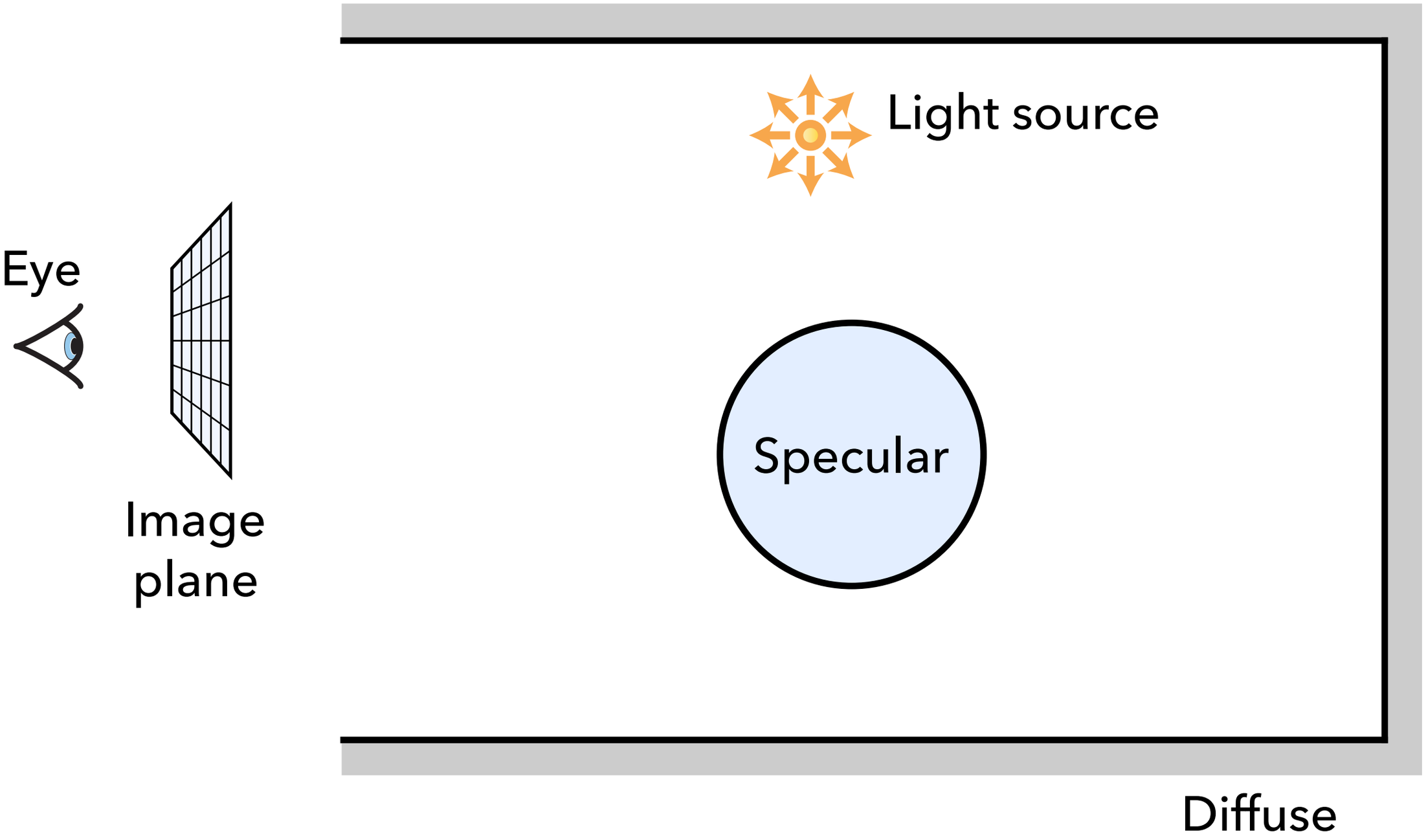
- Express light paths in terms of the surface interactions that have occurred
- A light path is a chain of linear segments joined at event “vertices”

Heckbert's Classification

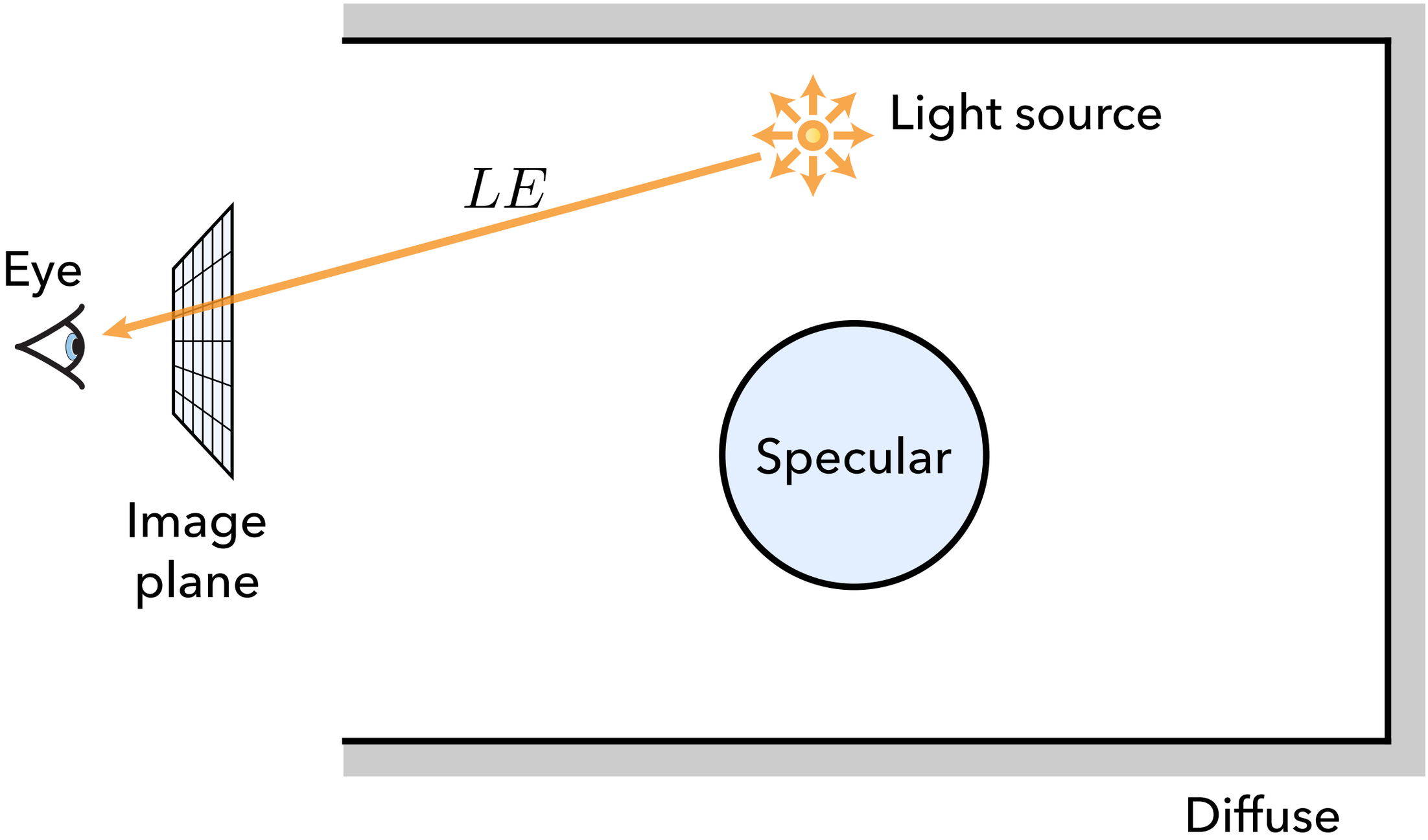
- Classification of "vertices":
  - L : a light source
  - E : the eye
  - S : a specular reflection
  - D : a diffuse reflection

classification by Paul Heckbert

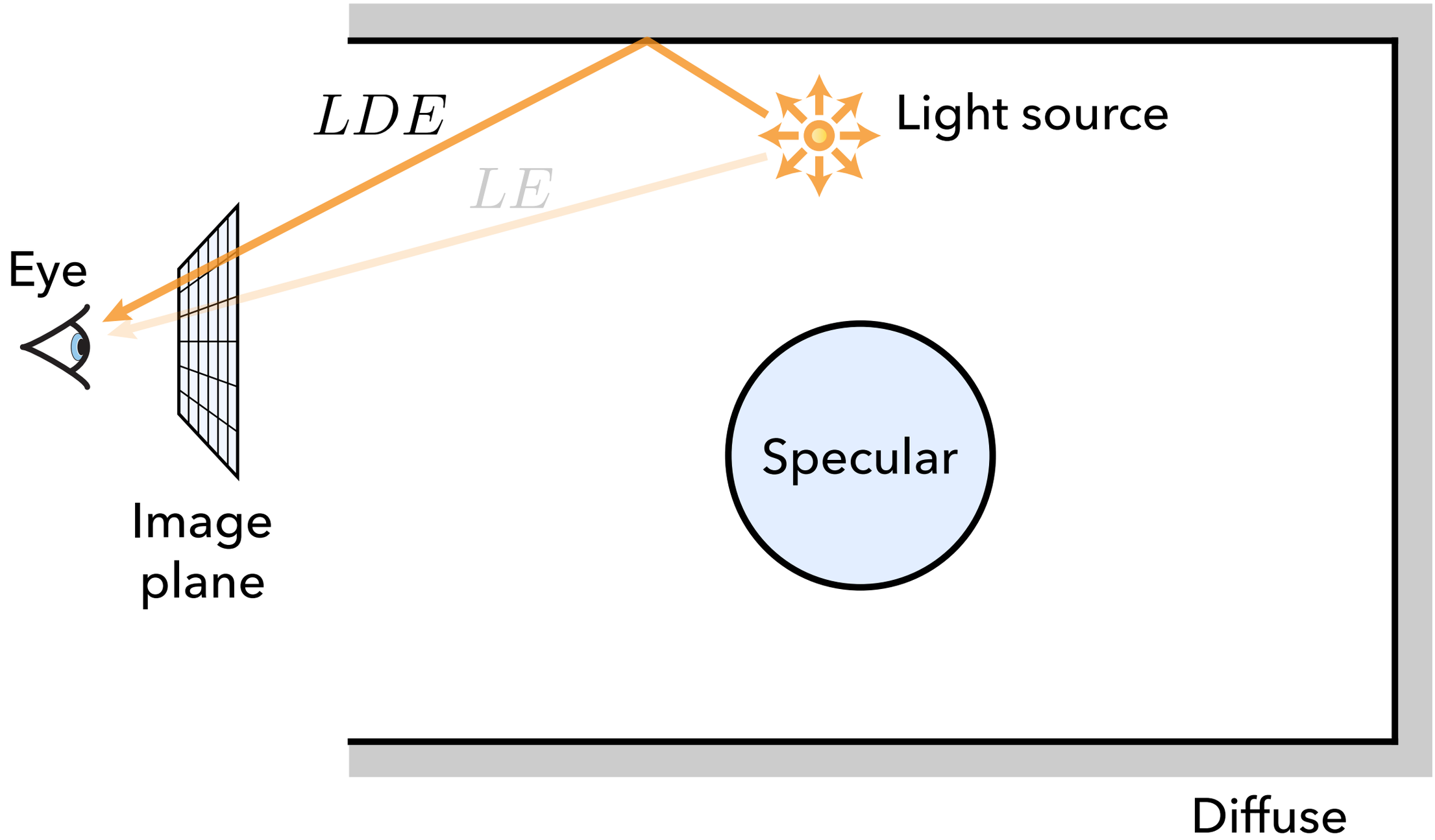
Heckbert's Classification



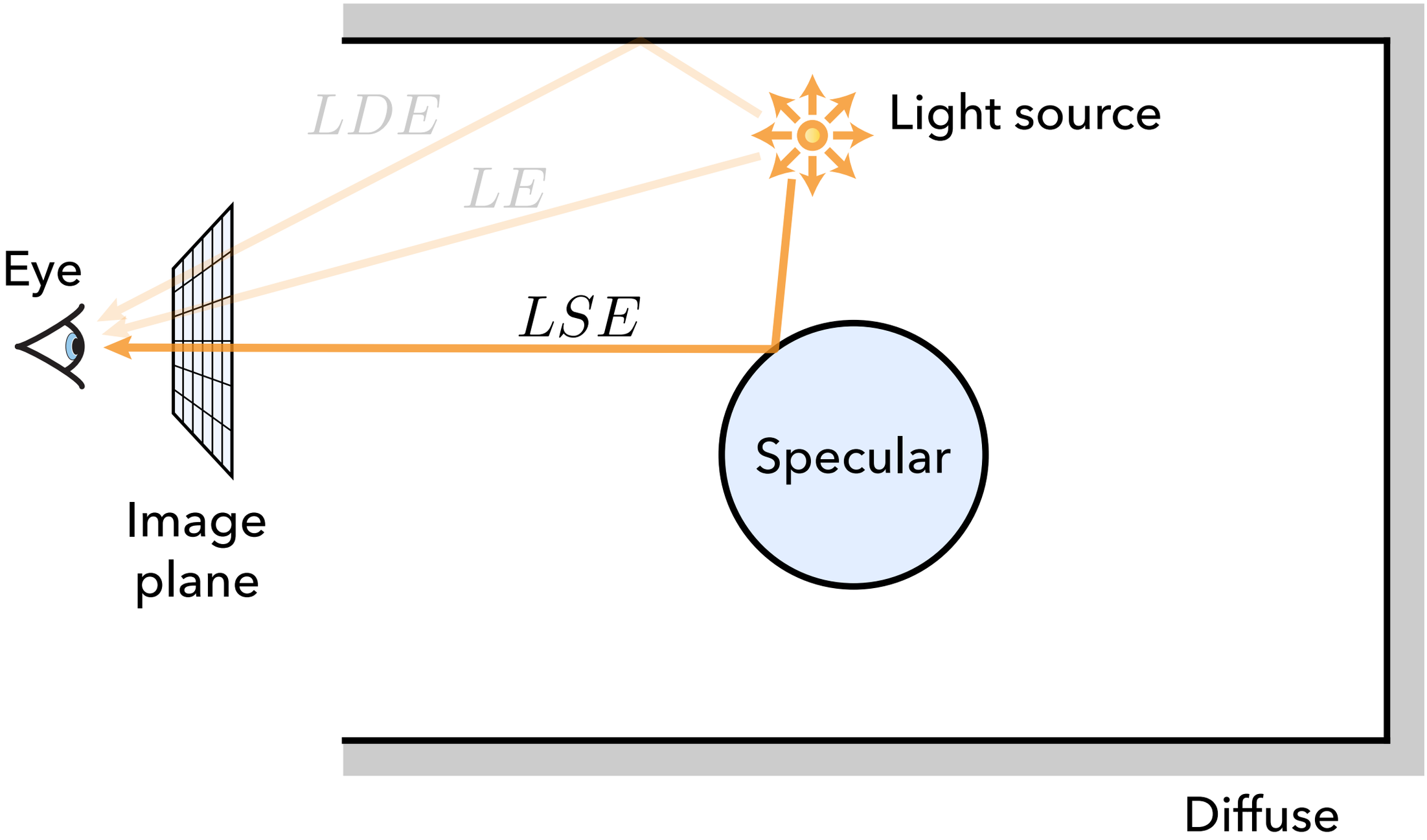
Heckbert's Classification



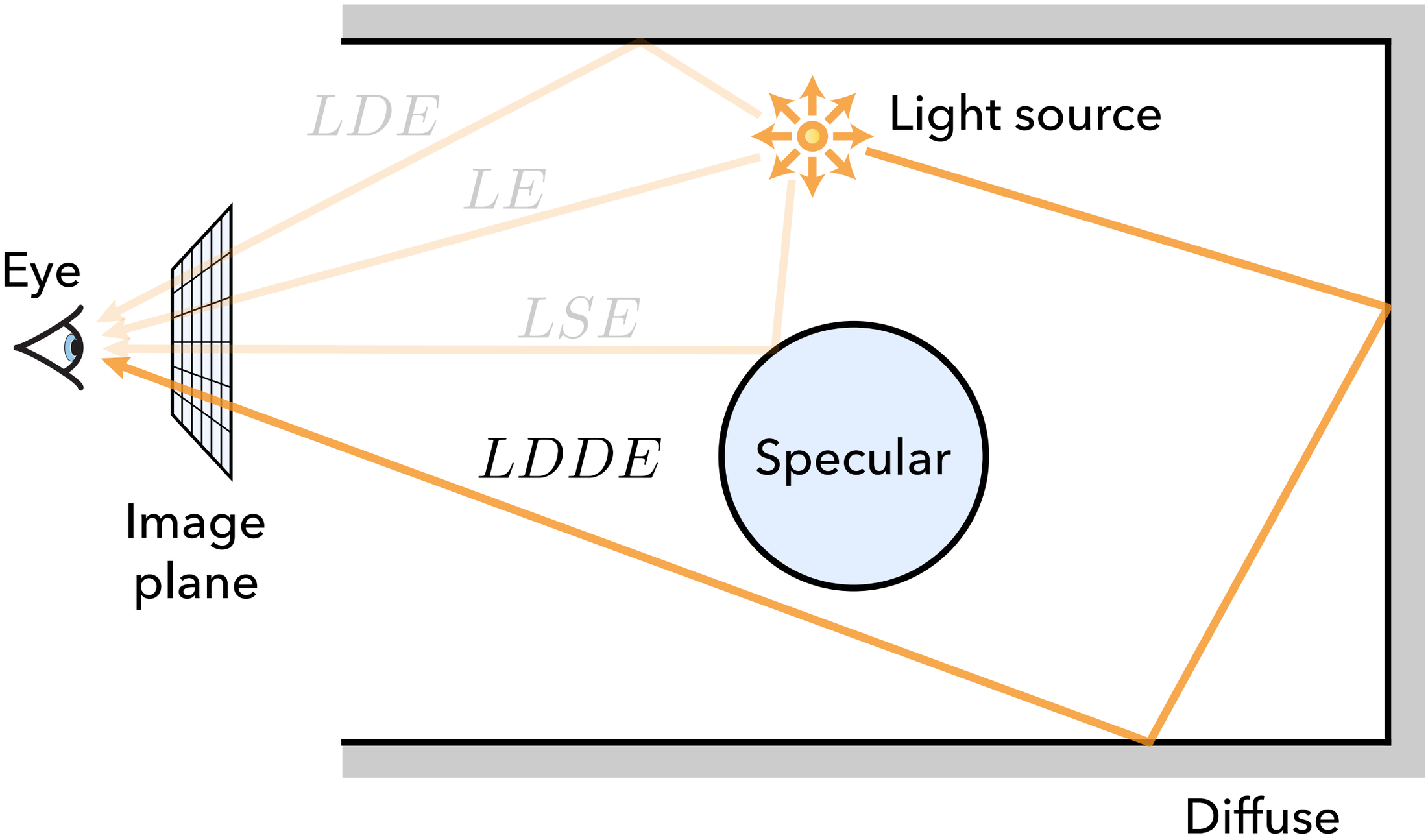
Heckbert's Classification



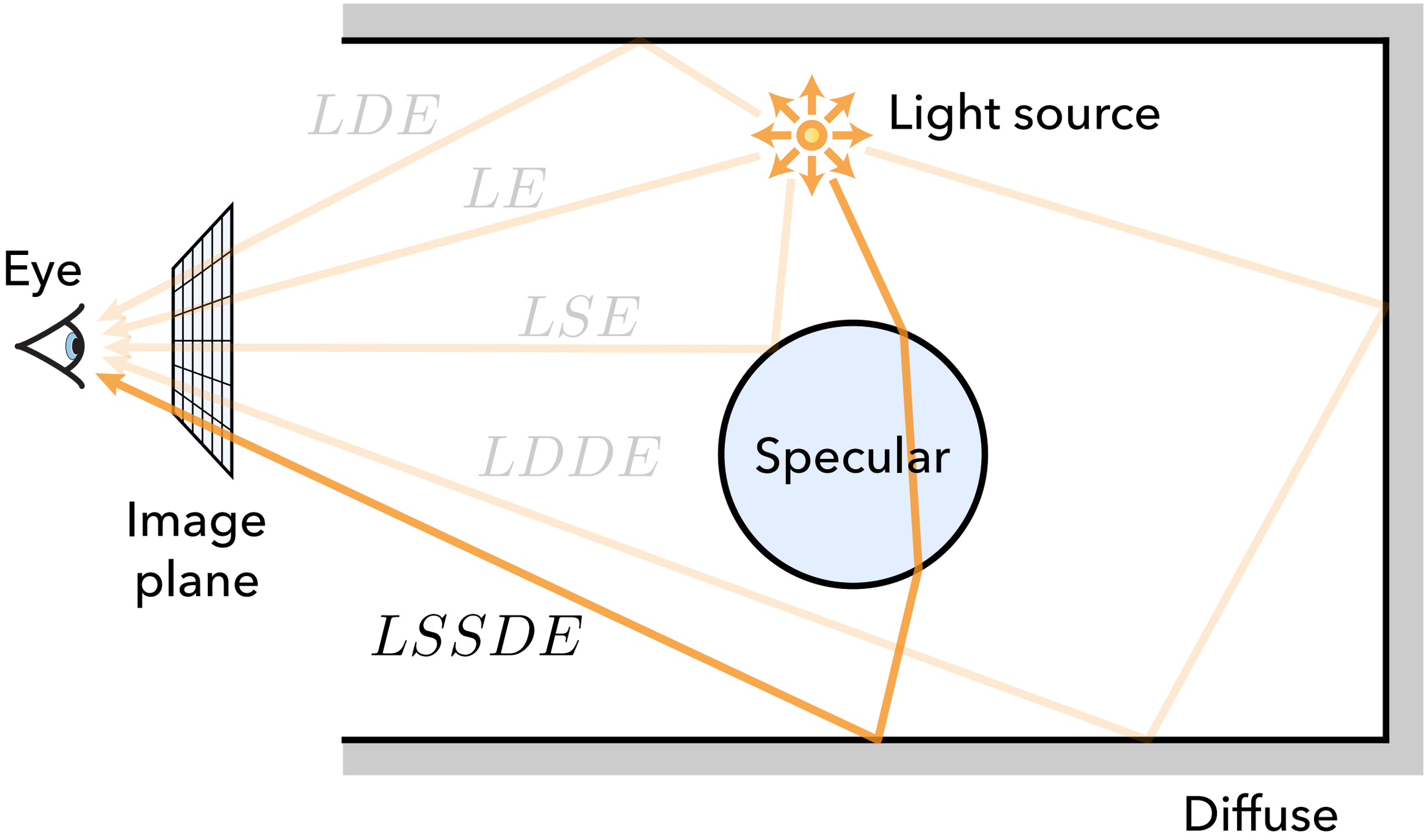
Heckbert's Classification



Heckbert's Classification



Heckbert's Classification

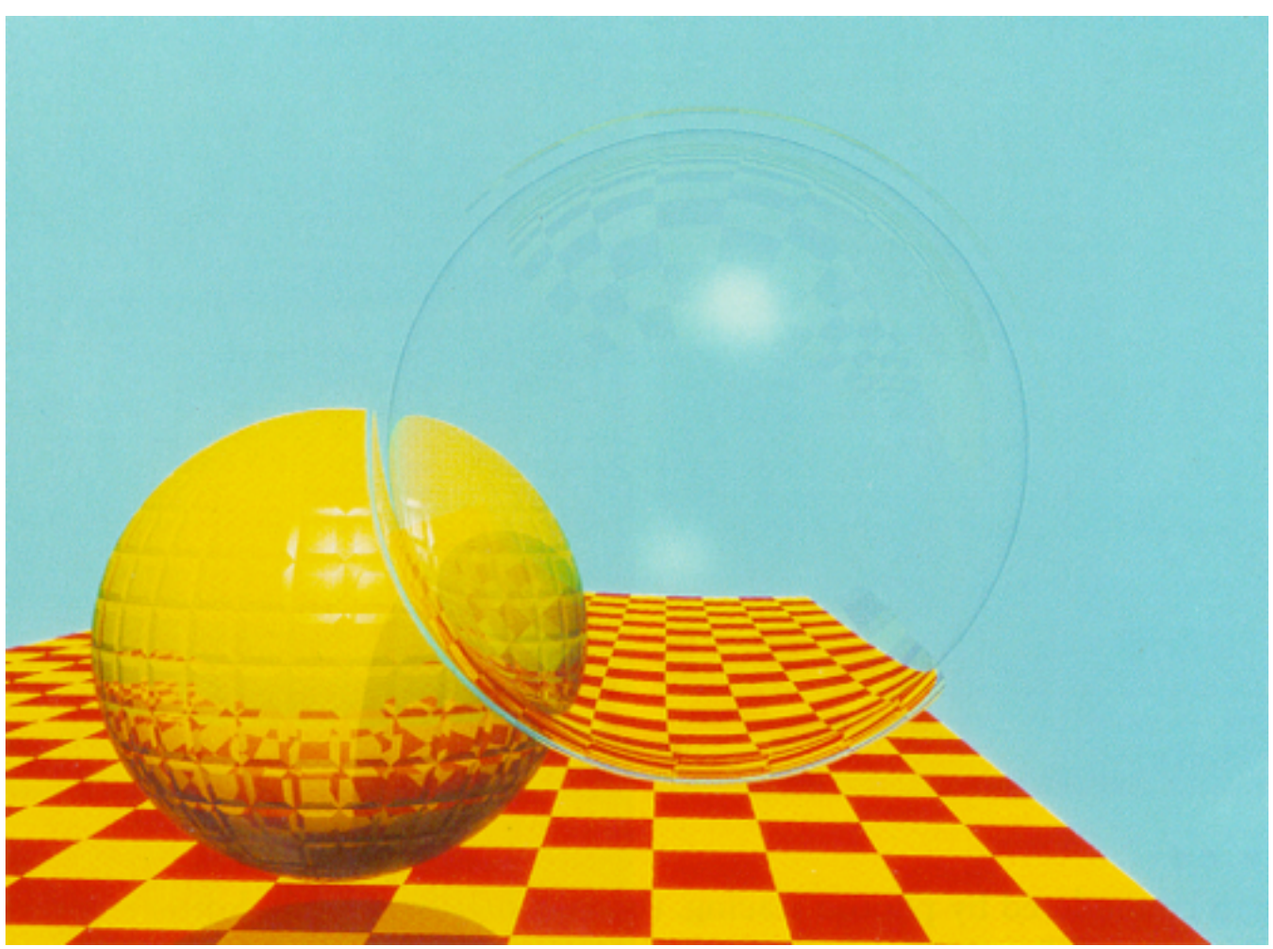


Heckbert's Classification

- Can express arbitrary classes of paths using a regular expression type syntax:
  - $k+$: one or more of event k
  - $k*$: zero or more of event k
  - $k?$: zero or one k events
  - $(k|h)$: a k or h event

Heckbert's Classification

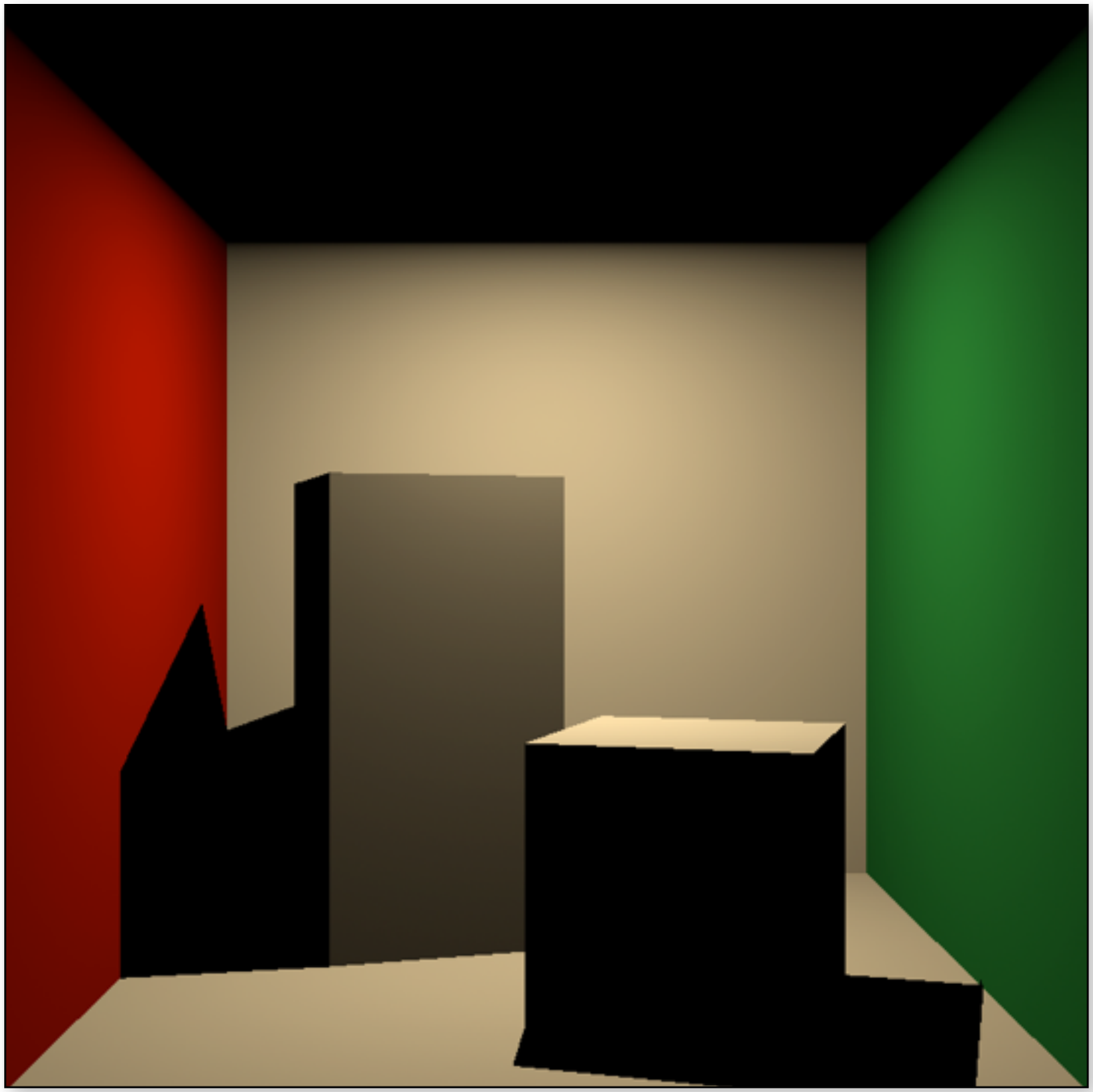
- Direct illumination: $L(D|S)E$
- Indirect illumination: $L(D|S)(D|S)+E$
- Classical (Whitted-style) ray tracing: $LDS*E$



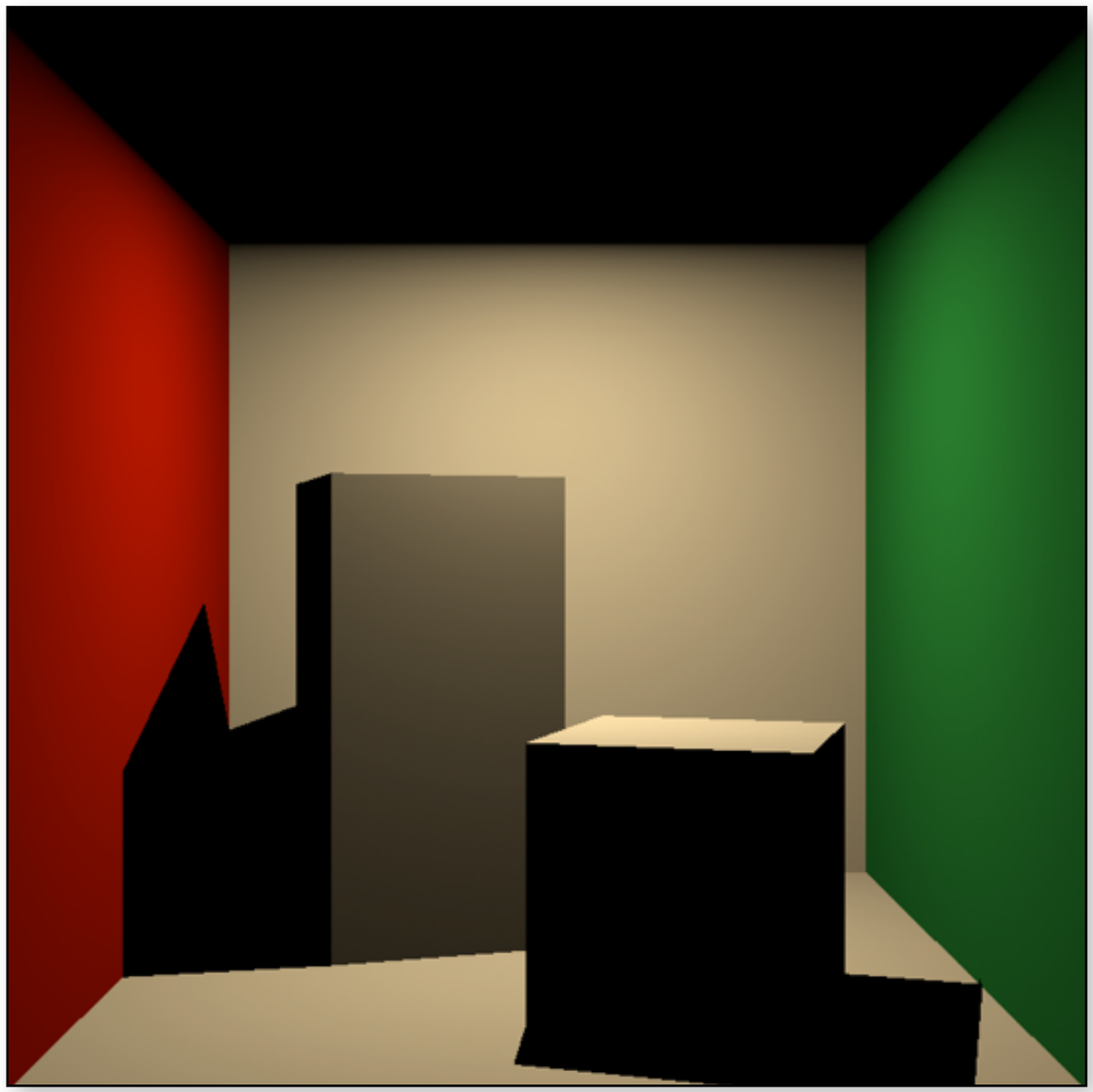
Heckbert's Classification

- Direct illumination: $L(D|S)E$
- Indirect illumination: $L(D|S)(D|S)+E$
- Classical (Whitted-style) ray tracing: $LDS*E$
- Full global illumination: $L(D|S)*E$
  - diffuse inter-reflections: $LDD+E$
  - caustics: $LS+DE$

Ray Tracing



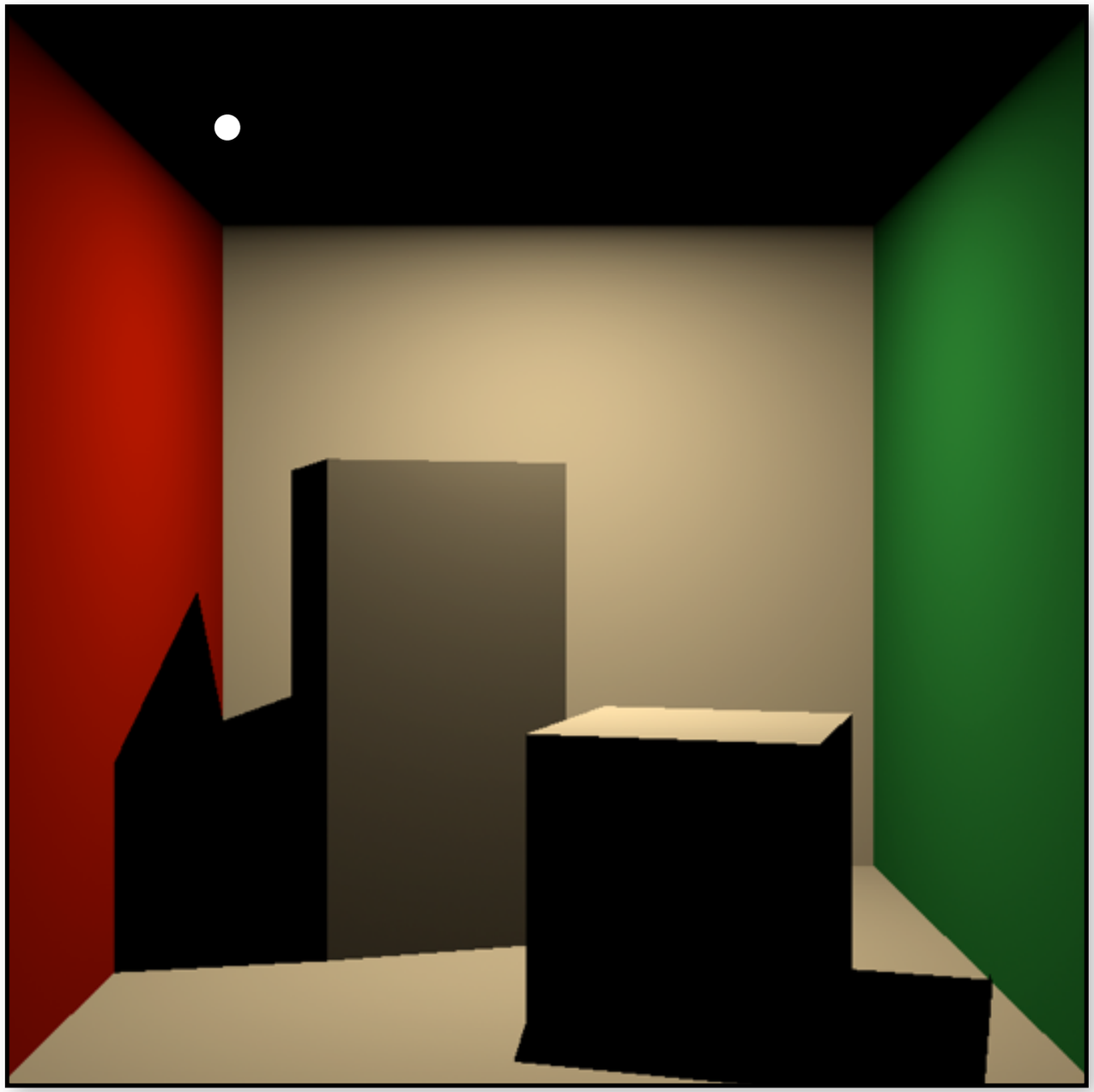
Beyond Ray Tracing

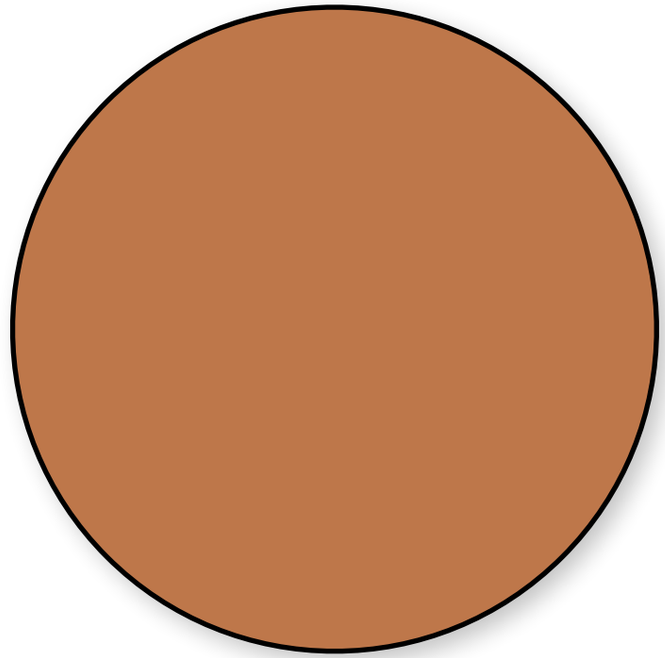


Ray-traced

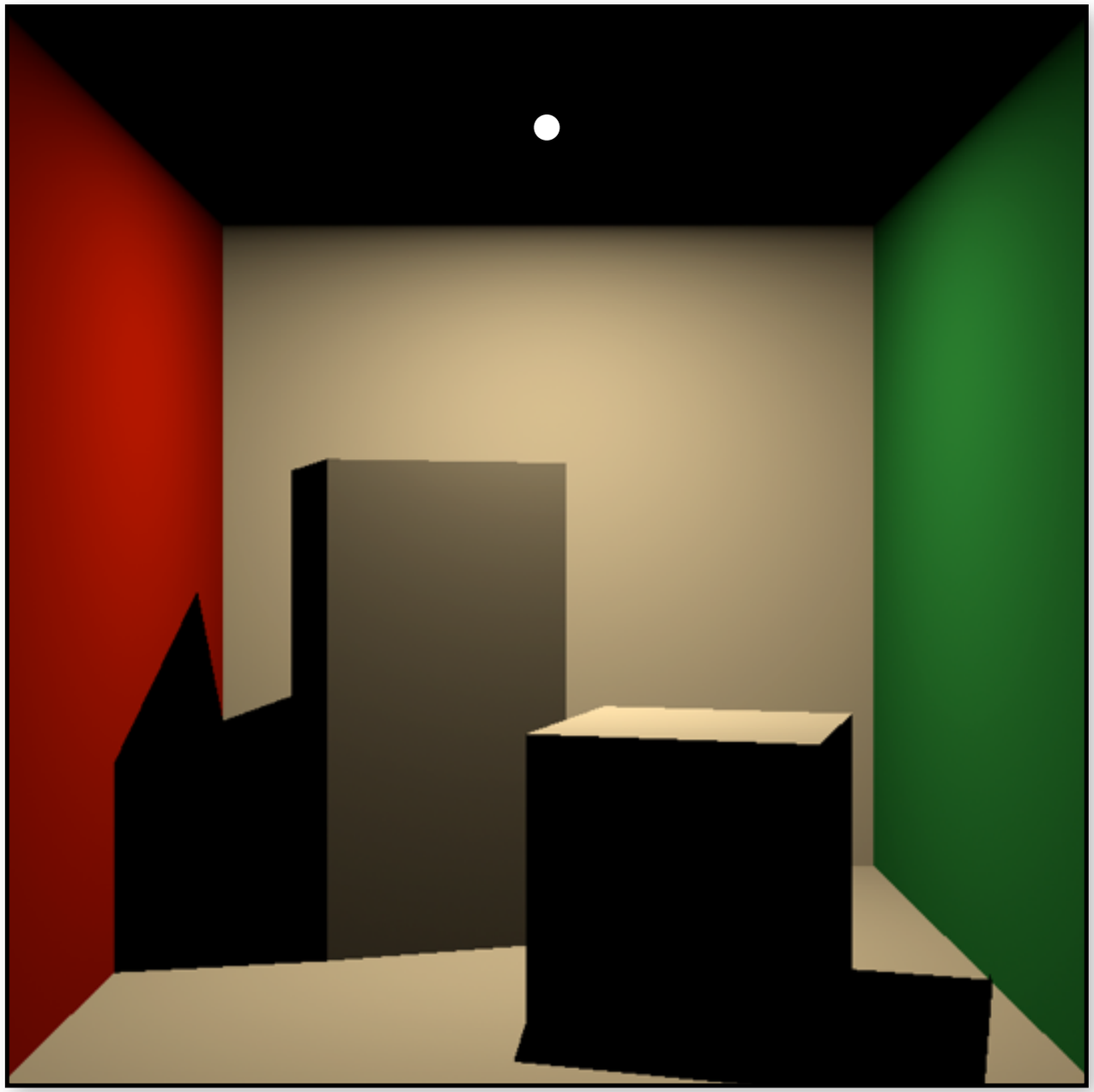
Real

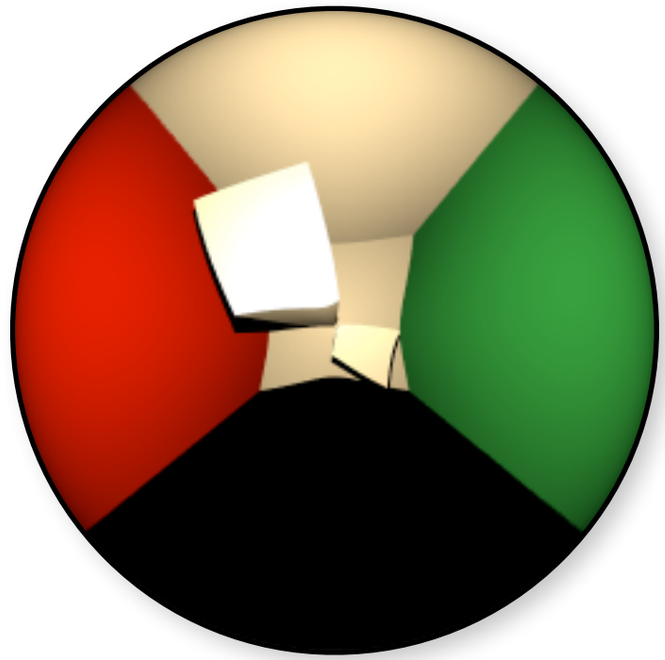
What is Indirect Illumination?

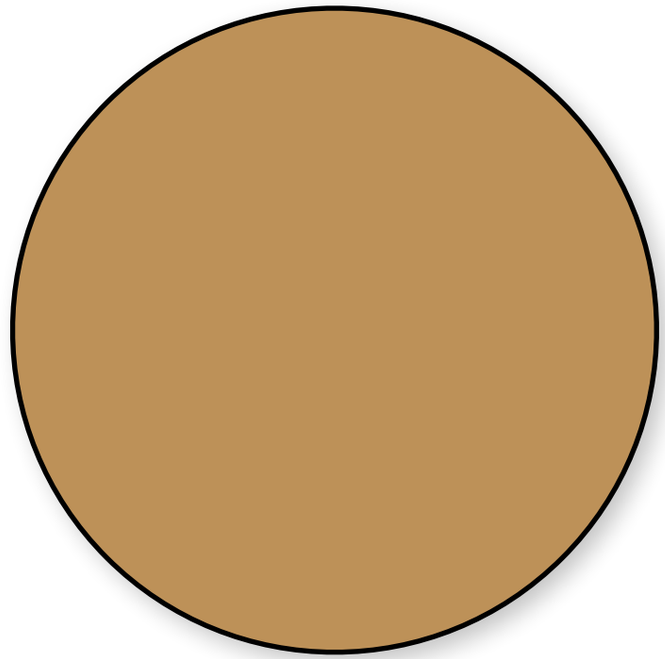




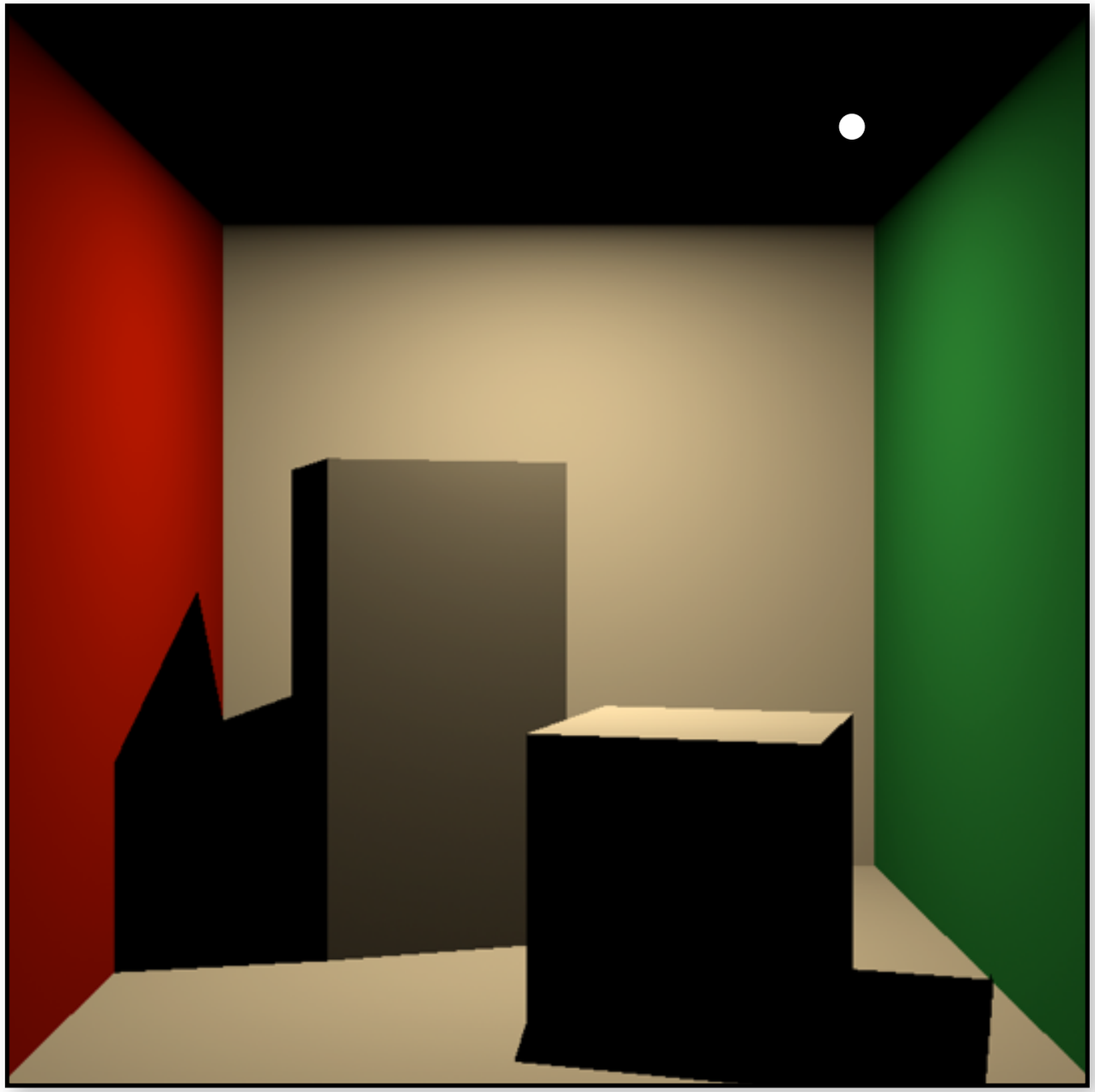
What is Indirect Illumination?

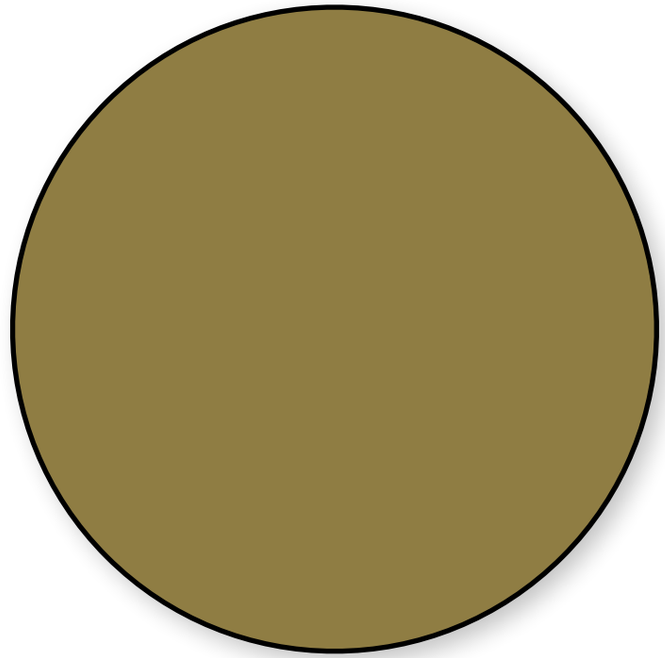




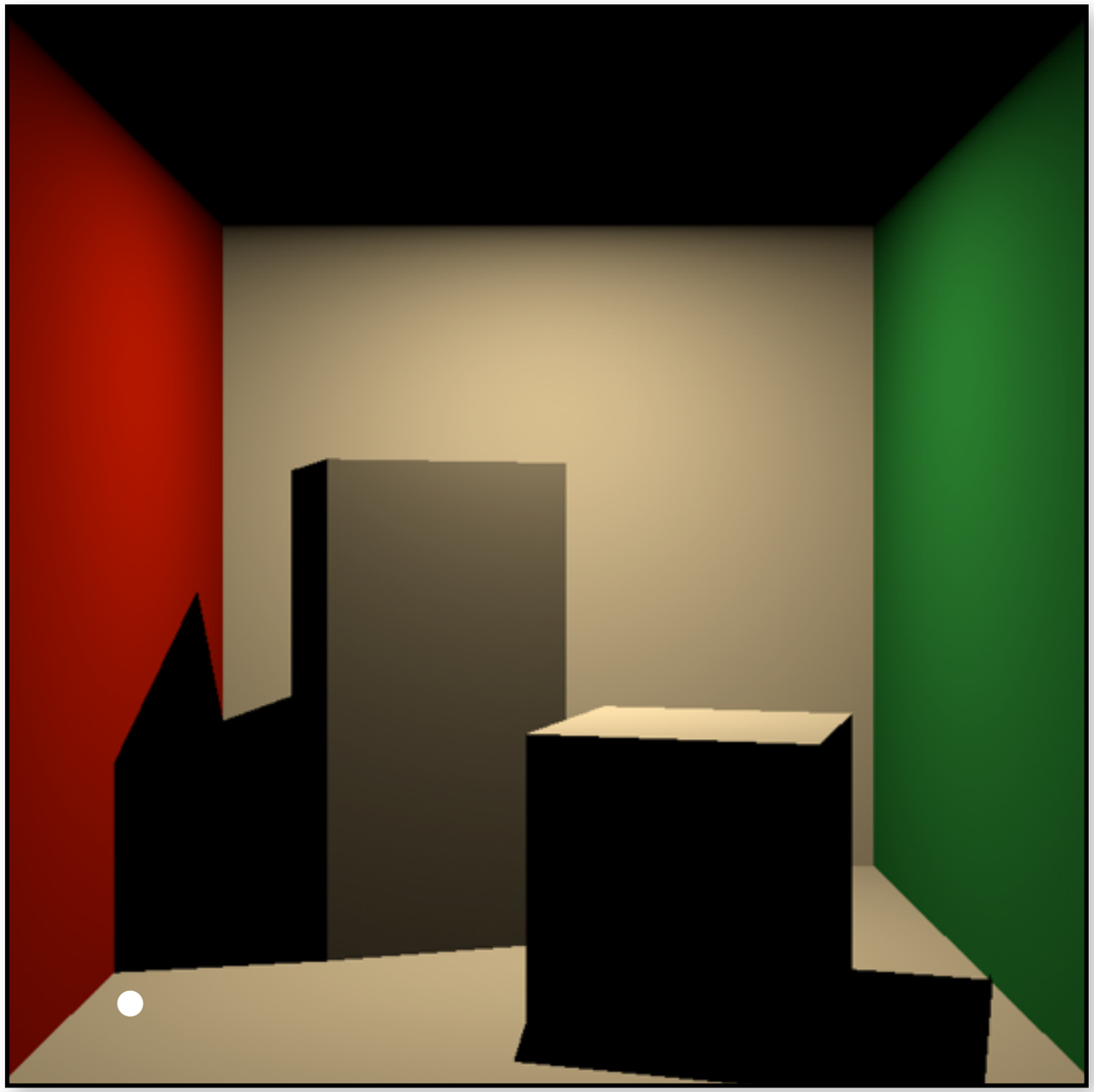


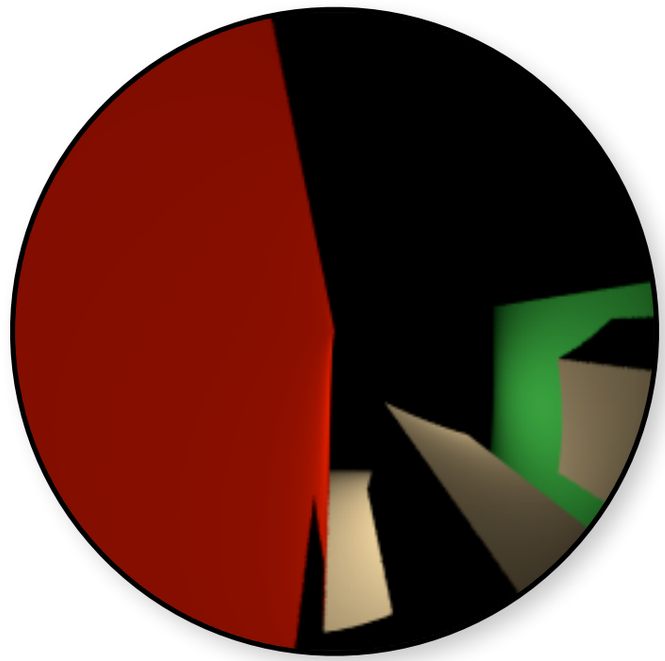
What is Indirect Illumination?

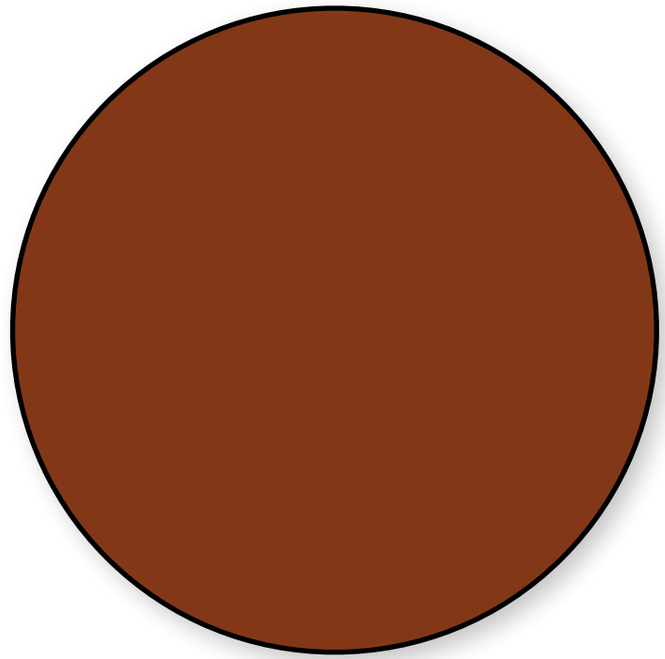




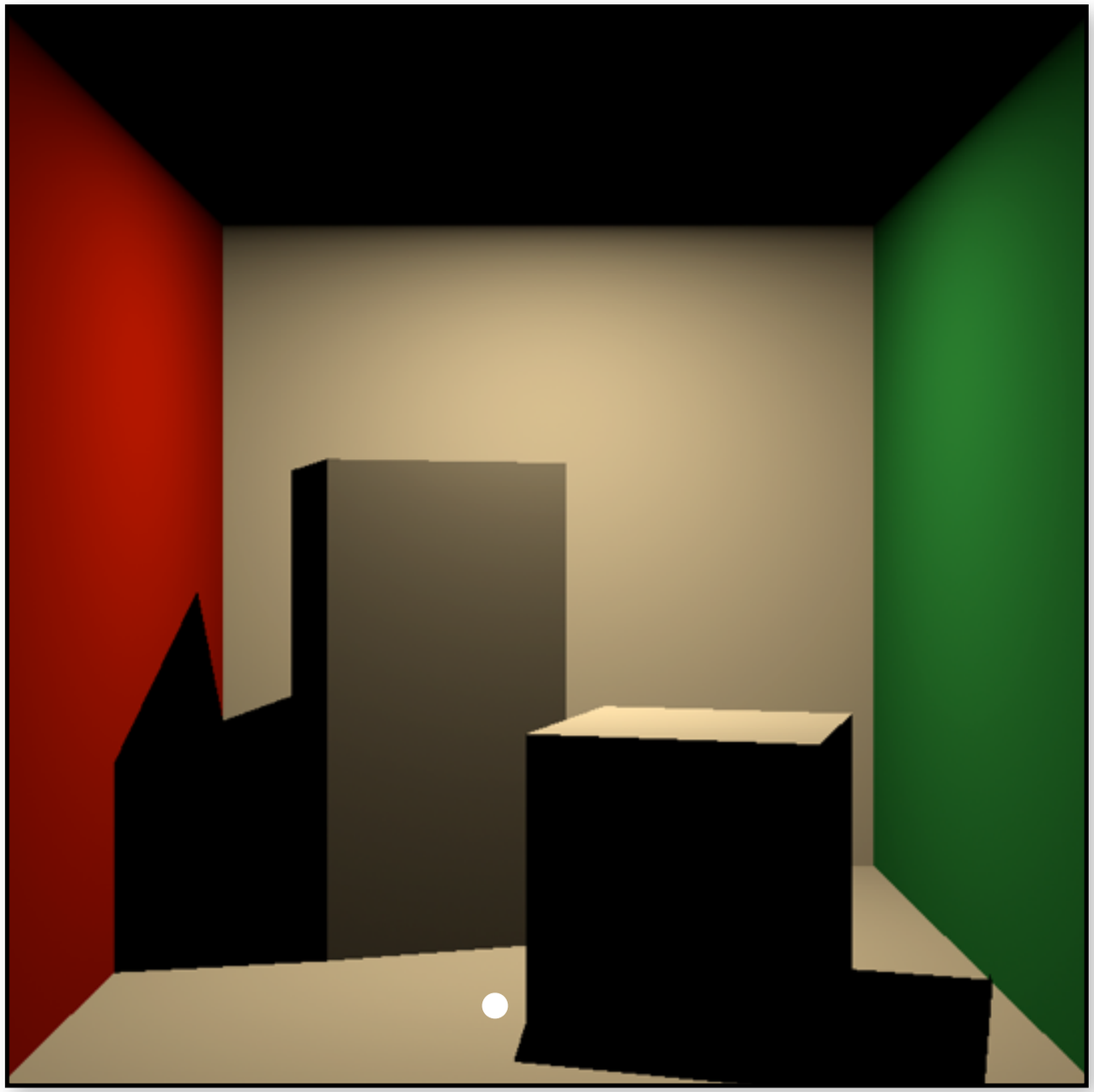
What is Indirect Illumination?

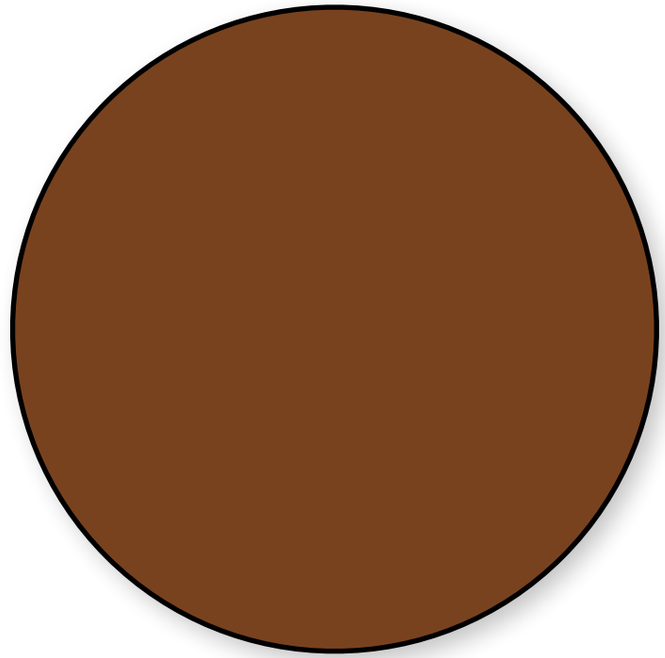




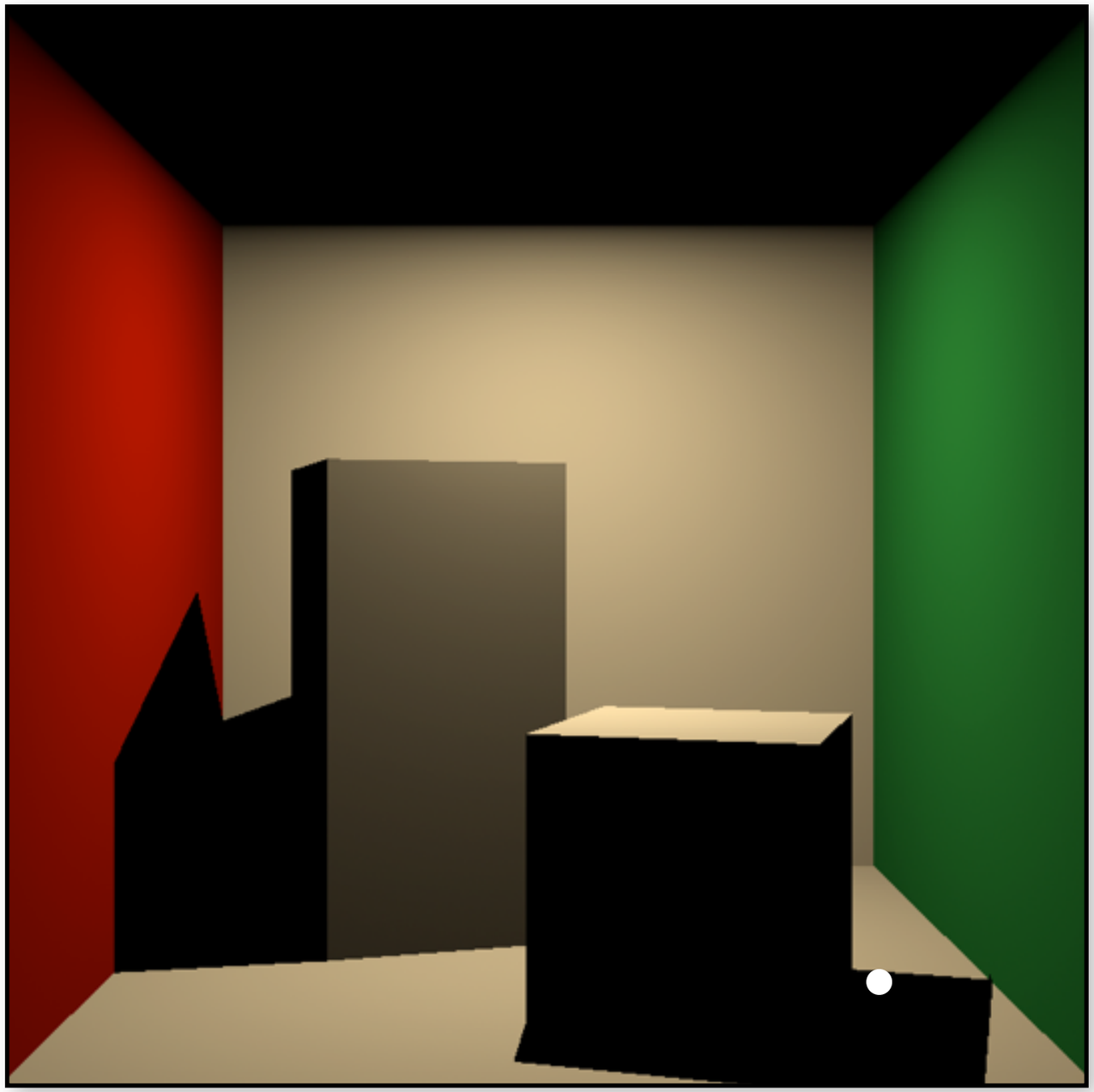


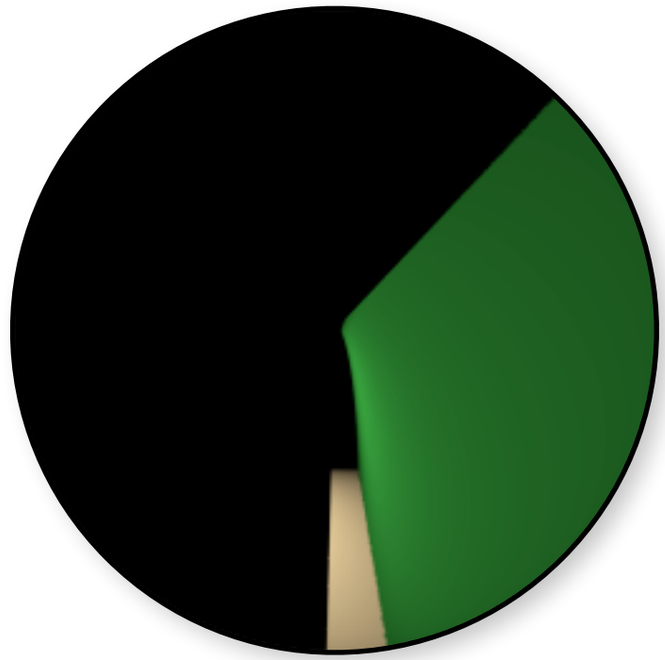
What is Indirect Illumination?

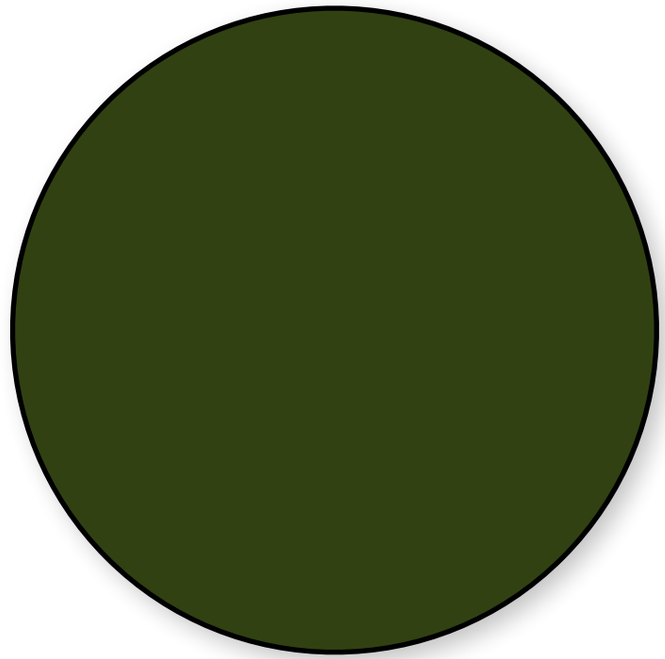




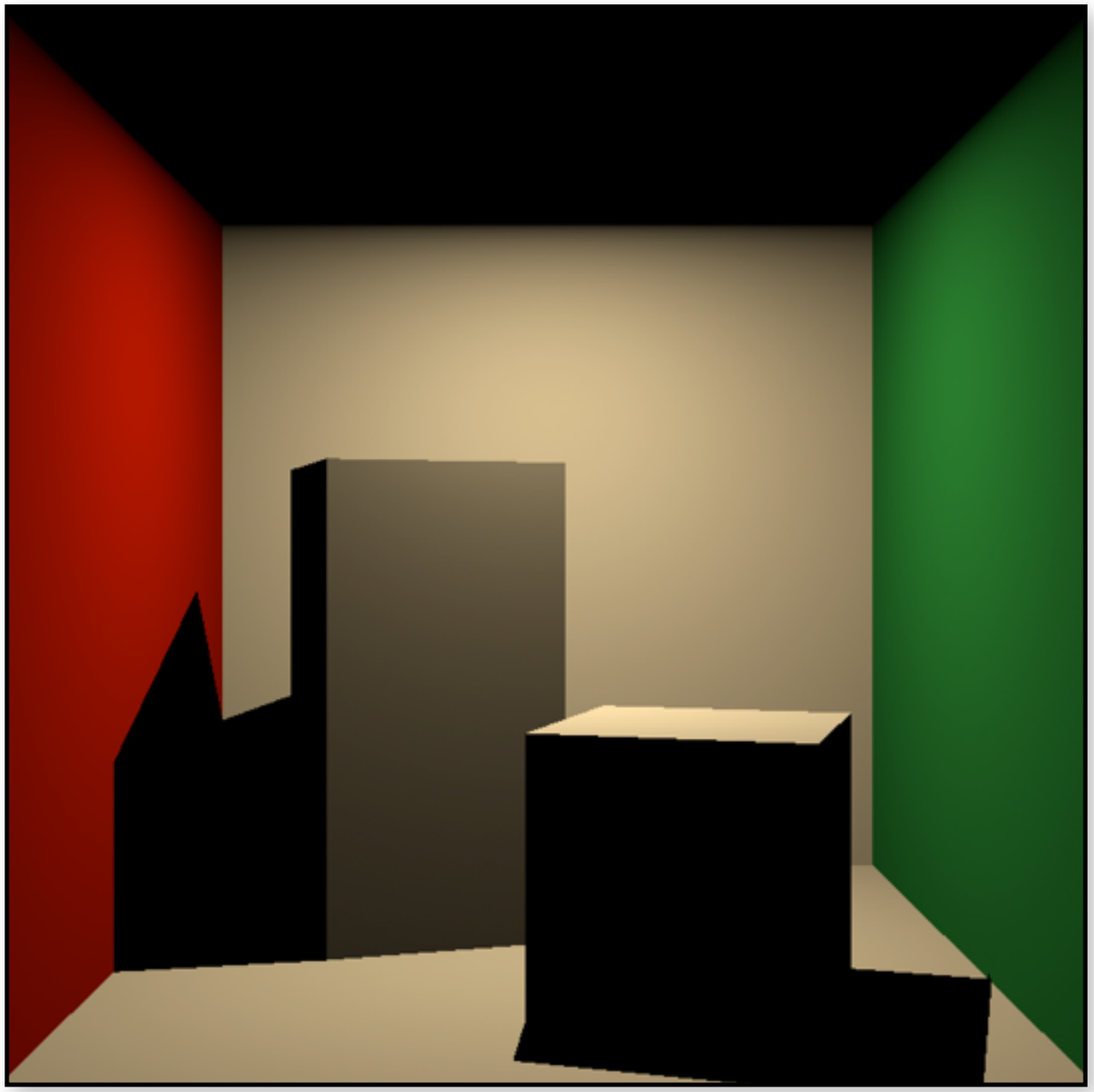
What is Indirect Illumination?



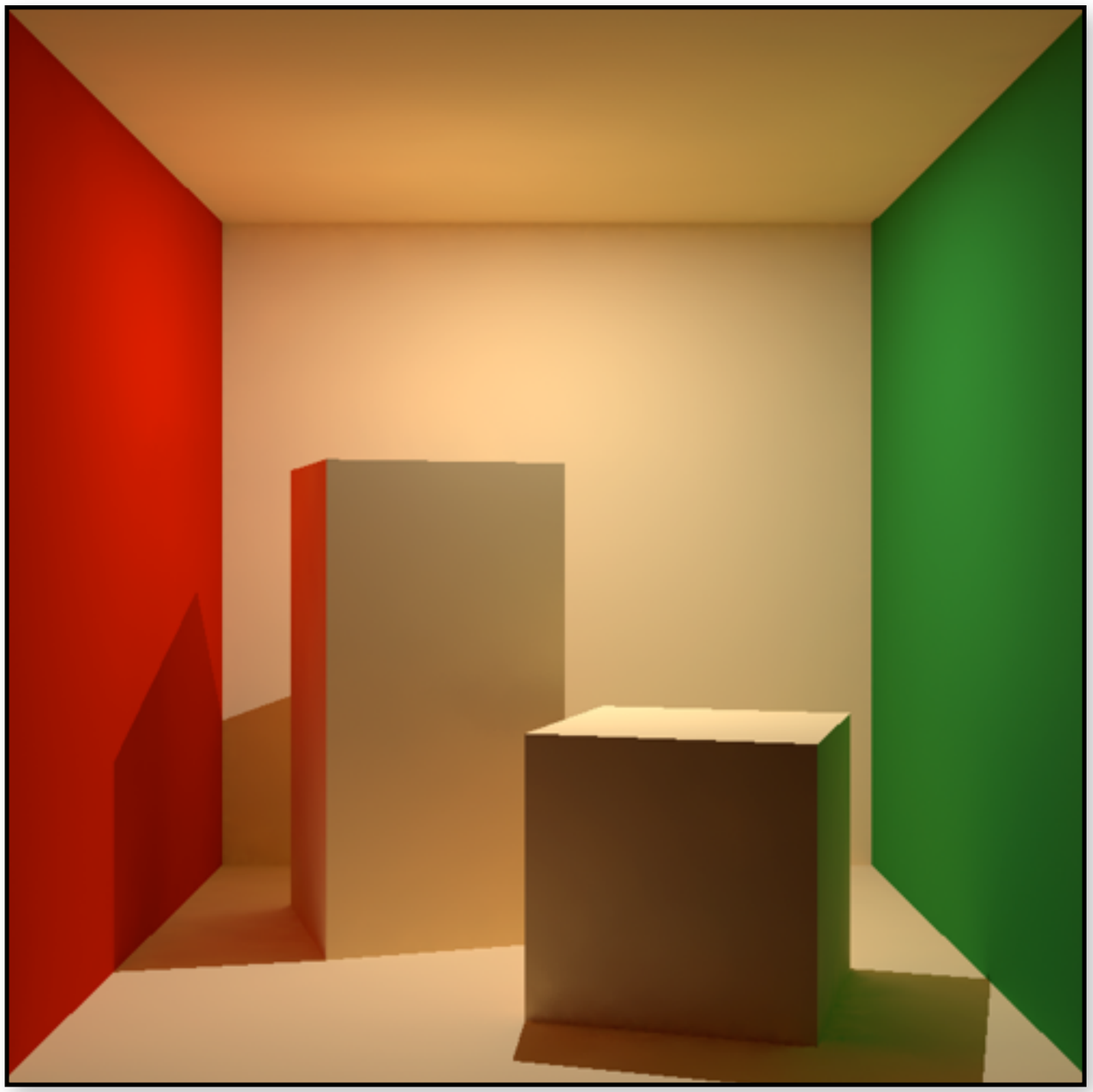




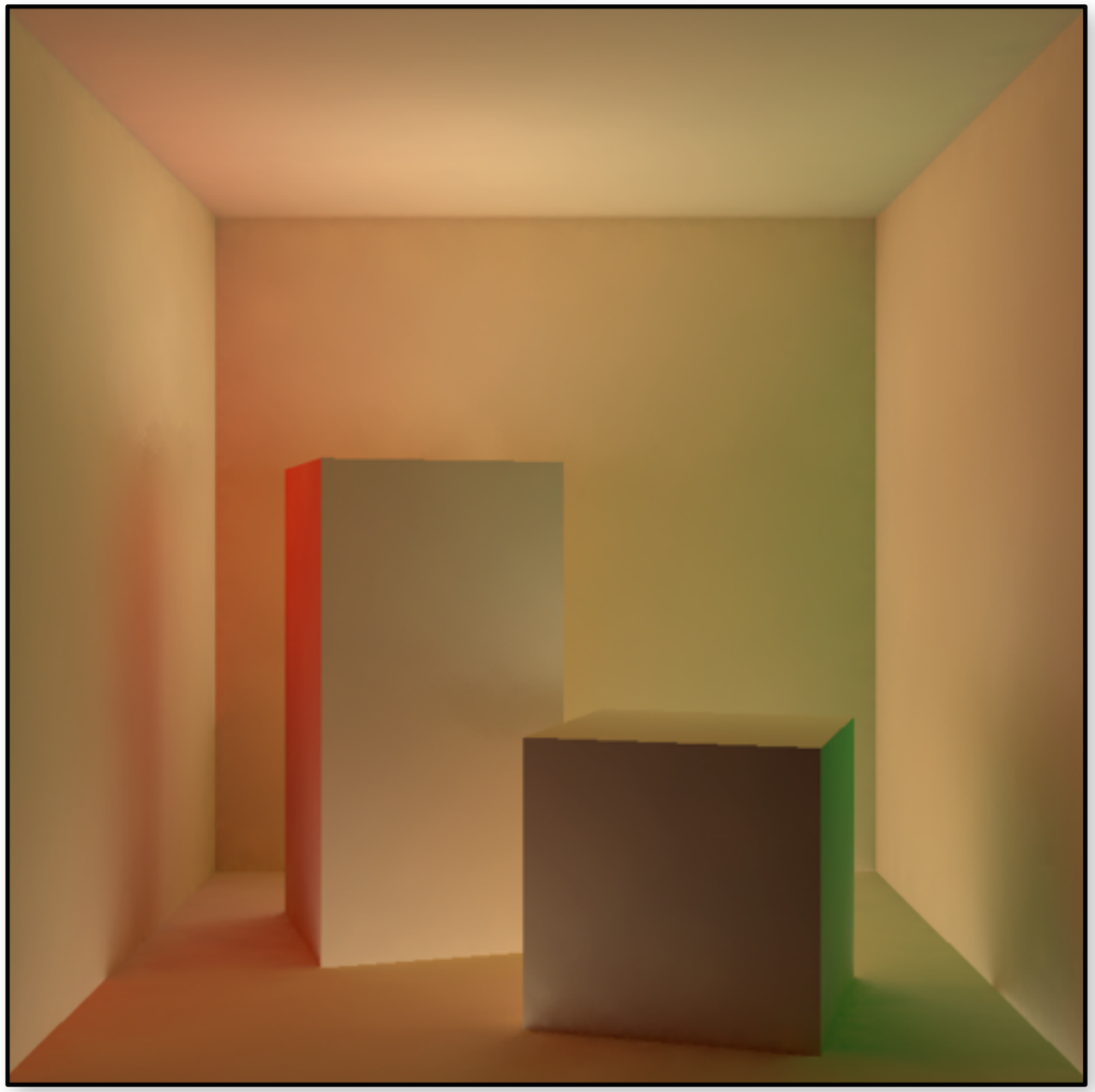
Direct Illumination



Direct + Indirect Illumination

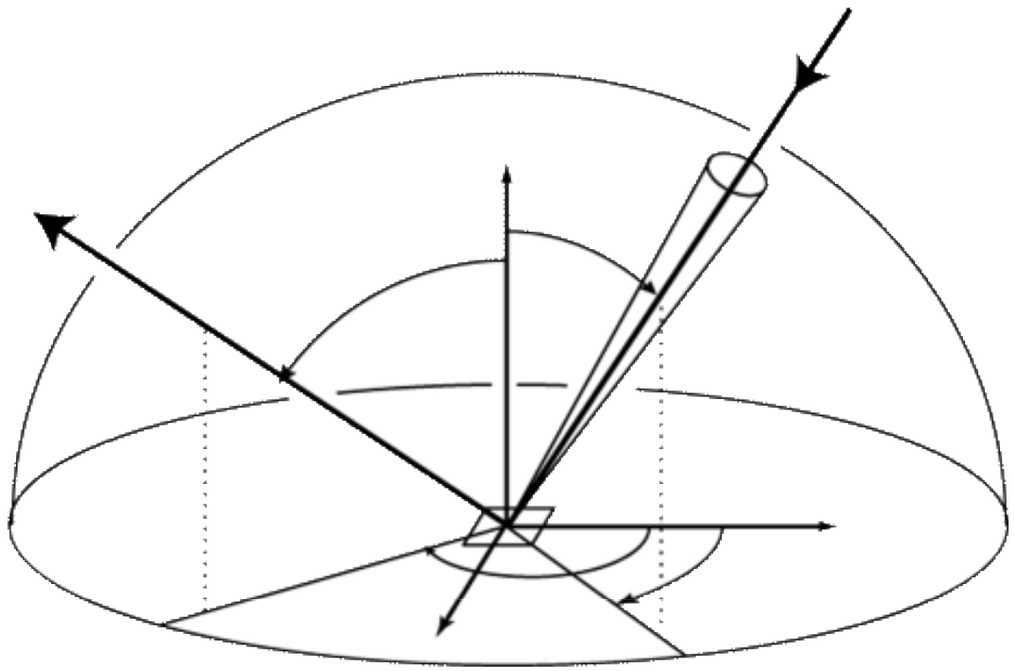


Indirect Illumination



Reflection Equation

- reflected radiance depends on incident radiance



$$L_r(\mathbf{x}, \vec{\omega}_r) = \int_{H^2} f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_r) L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i \, d\vec{\omega}_i$$

Rendering Equation

- James Kajiya, "The Rendering Equation." *SIGGRAPH* 1986.
- Energy equilibrium:

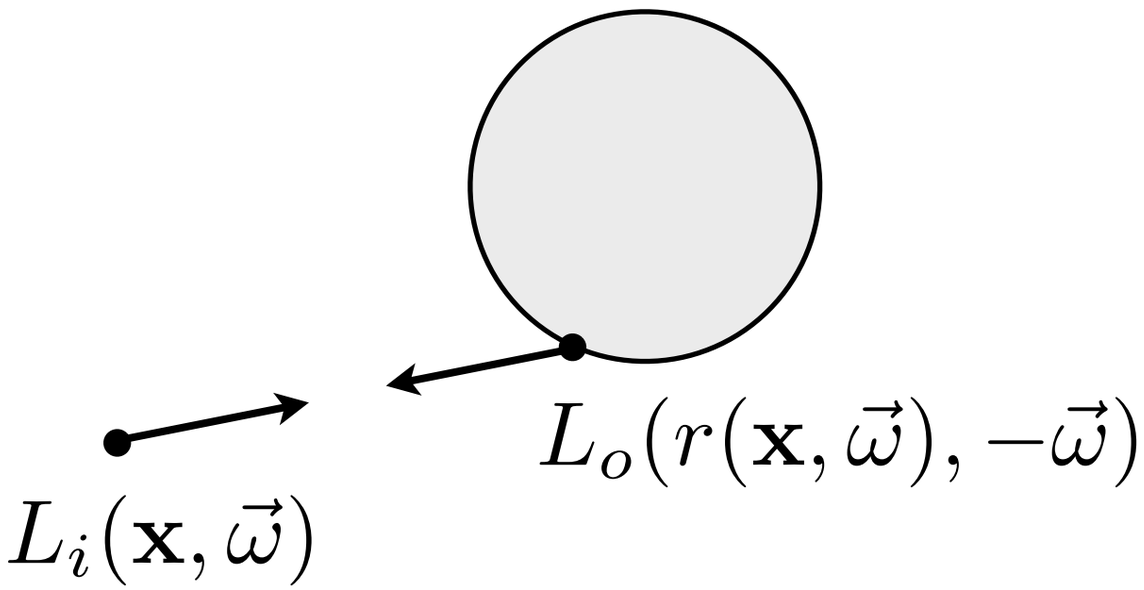
$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + L_r(\mathbf{x}, \vec{\omega}_o)$$

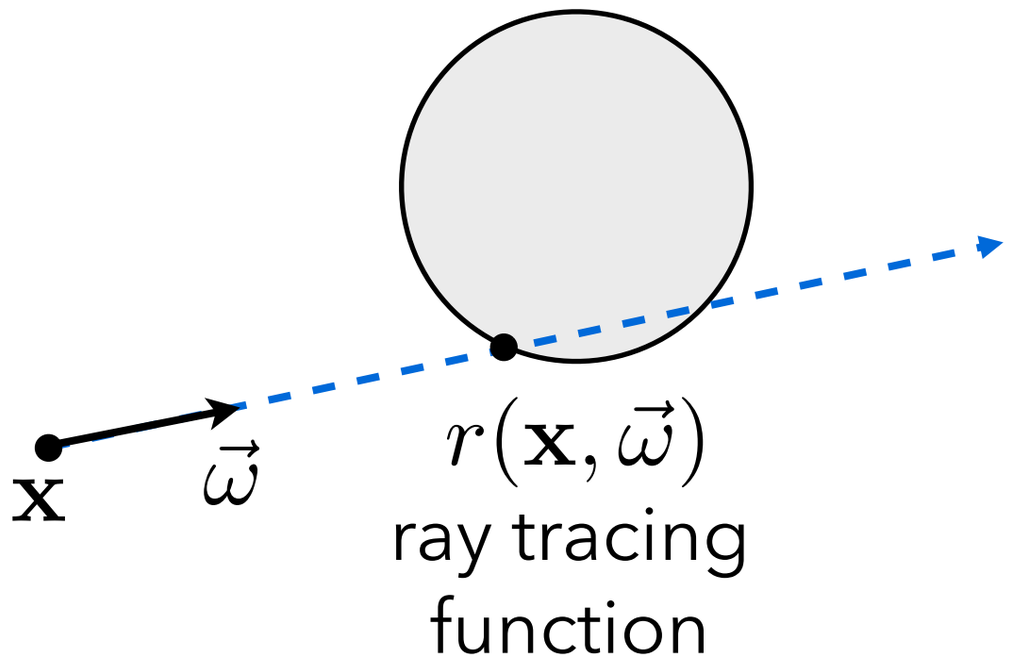
outgoing emitted reflected

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + \int_{H^2} f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i \, d\vec{\omega}_i$$

Light Transport

- No participating media → radiance is constant along ray
- We can relate incoming radiance to outgoing radiance $L_i(\mathbf{x}, \vec{\omega}) = L_o(r(\mathbf{x}, \vec{\omega}), -\vec{\omega})$





Rendering Equation

- Hemispherical form

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + \int_{H^2} f_r(\mathbf{x}, \vec{\omega}', \vec{\omega}) L(r(\mathbf{x}, \vec{\omega}'), -\vec{\omega}') \cos\theta' d\vec{\omega}'$$

ray tracing function

- Only outgoing radiance (except for L_e) on both sides
  - we drop the "o" subscript
  - Fredholm equation of the second kind (recursive)

Rendering Equation

- Surface area form

$$L(\mathbf{x}', \mathbf{x}) = L_e(\mathbf{x}', \mathbf{x}) + \int_A f_r(\mathbf{x}', \mathbf{x}'', \mathbf{x}) L(\mathbf{x}'', \mathbf{x}') G(\mathbf{x}'', \mathbf{x}') \, dA(\mathbf{x}'')$$

$$G(\mathbf{x}'', \mathbf{x}') = V(\mathbf{x}'', \mathbf{x}') \frac{\cos\theta_i'' \cos\theta_o'}{\|\mathbf{x}'' - \mathbf{x}'\|^2}$$

$$V(\mathbf{x}'', \mathbf{x}') = \begin{cases} 1: & \text{visible} \\ 0: & \text{not visible} \end{cases}$$

Rendering Equation

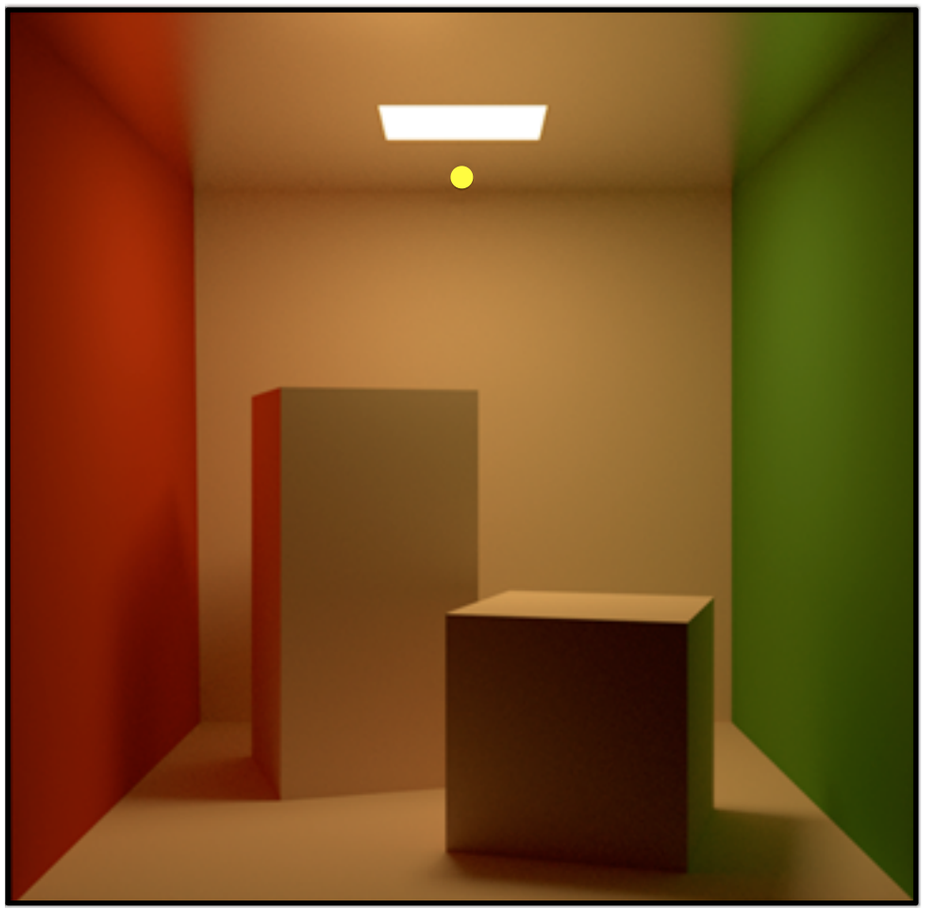
- How to evaluate these integrals?
- Closed-form solution almost never exists
  - multidimensional, complex BRDF
  - discontinuous integrand (visibility)
- Numerical Methods
  - Gaussian quadrature
  - curse of dimensionality → inefficient

Solving the Rendering Equation

- Monte Carlo methods
  - Today: unbiased methods
    - Recursive Monte Carlo ray tracing
    - Path tracing and light tracing
    - Bidirectional path tracing
  - Later: biased methods
    - Many-light algorithms
    - Density estimation
    - (Probabilistic) progressive photon mapping
    - Irradiance caching

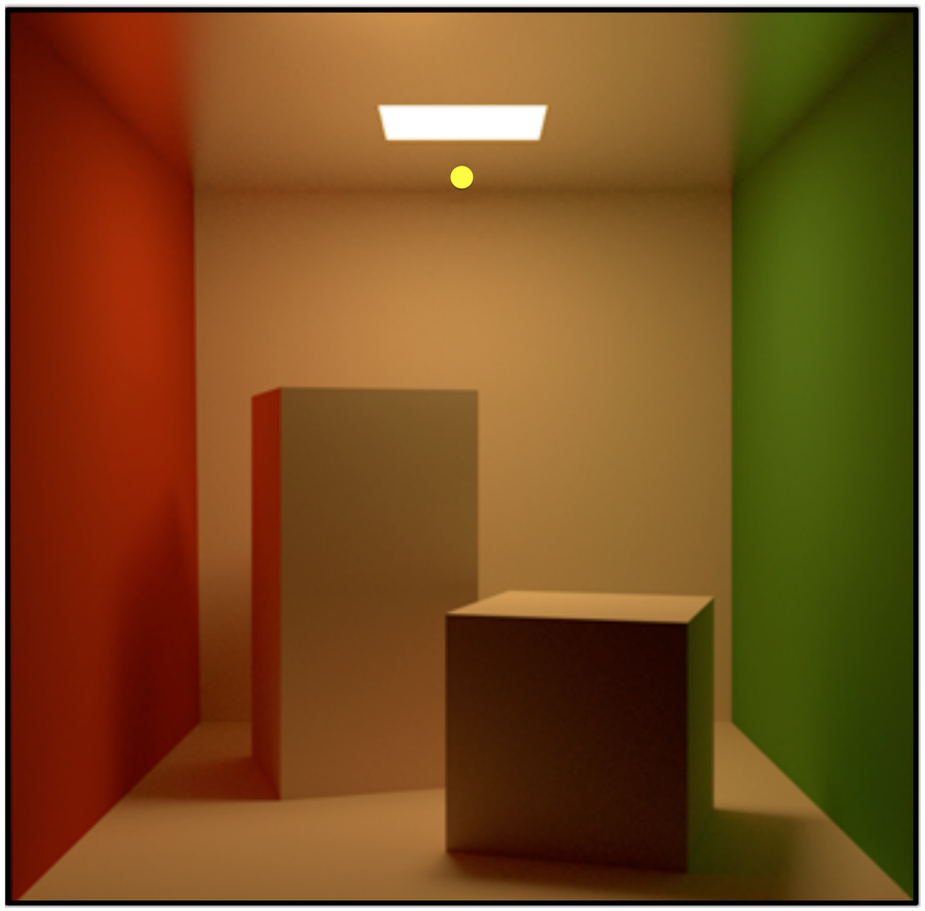
Solving the Rendering Equation

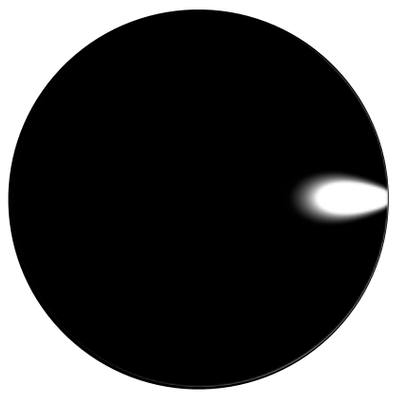
$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + \int_{H^2} f_r(\mathbf{x}, \vec{\omega}', \vec{\omega}) L(r(\mathbf{x}, \vec{\omega}'), -\vec{\omega}') \cos\theta' d\vec{\omega}'$$

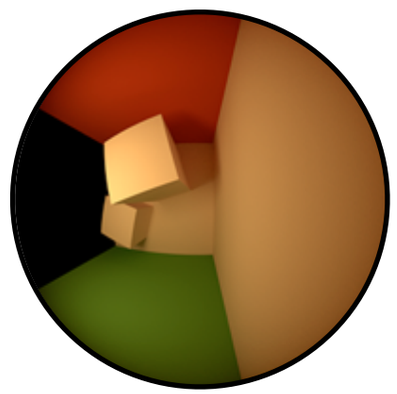


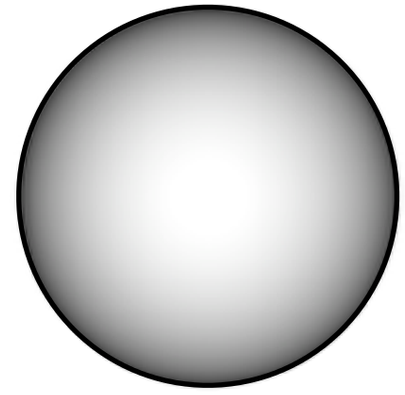
Solving the Rendering Equation

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + \int_{H^2} f_r(\mathbf{x}, \vec{\omega}', \vec{\omega}) L(r(\mathbf{x}, \vec{\omega}'), -\vec{\omega}') \cos\theta' \, d\vec{\omega}'$$



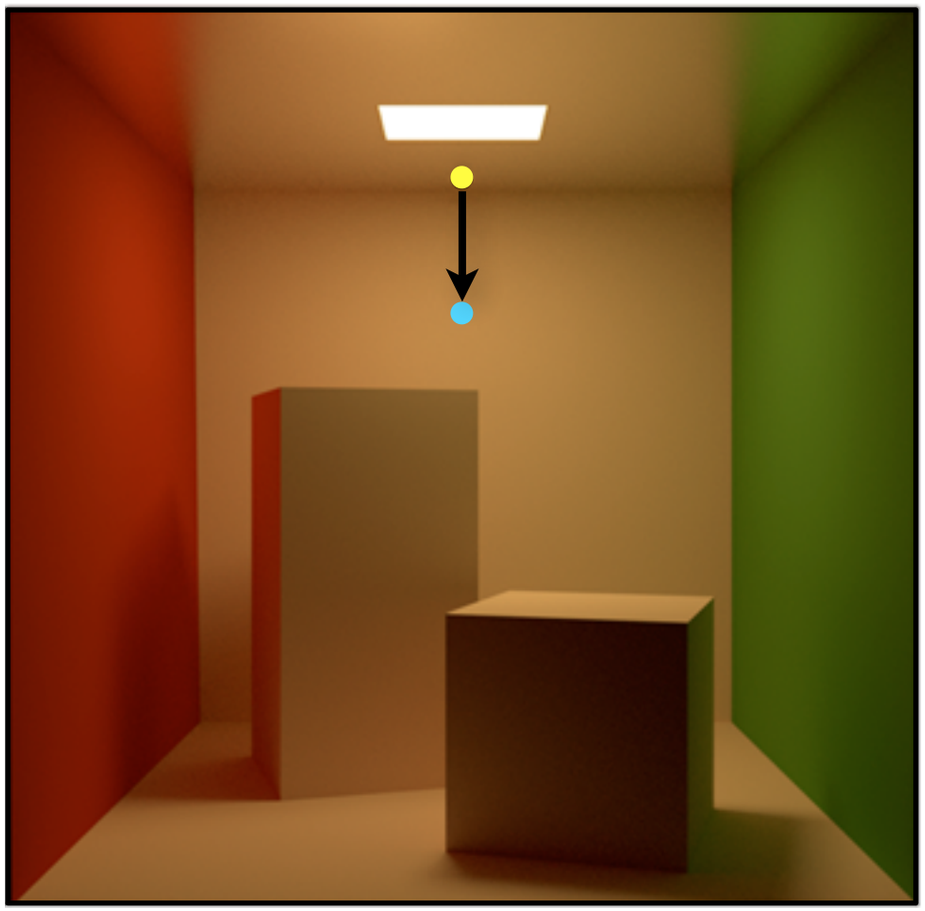


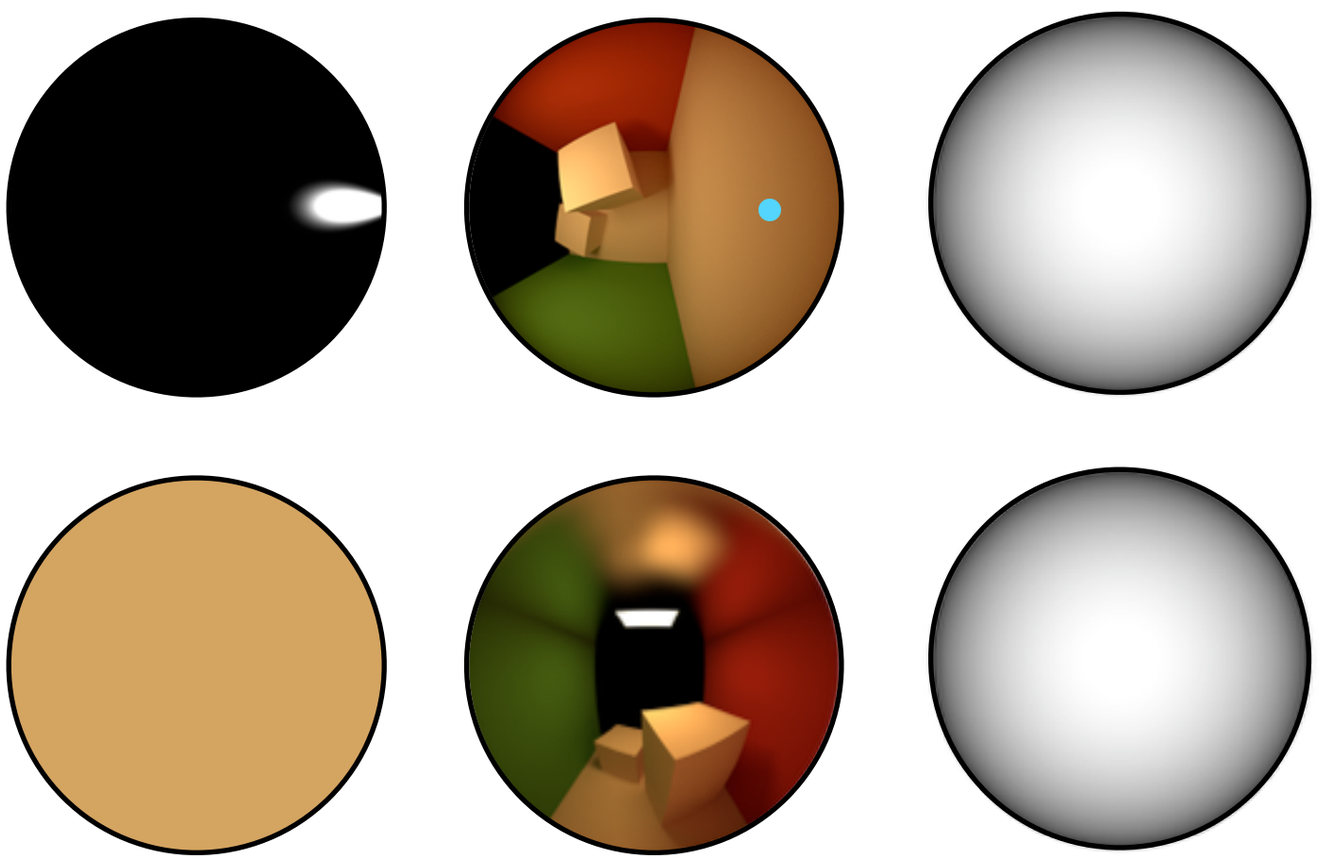




Solving the Rendering Equation

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + \int_{H^2} f_r(\mathbf{x}, \vec{\omega}', \vec{\omega}) L(r(\mathbf{x}, \vec{\omega}'), -\vec{\omega}') \cos\theta' \, d\vec{\omega}'$$

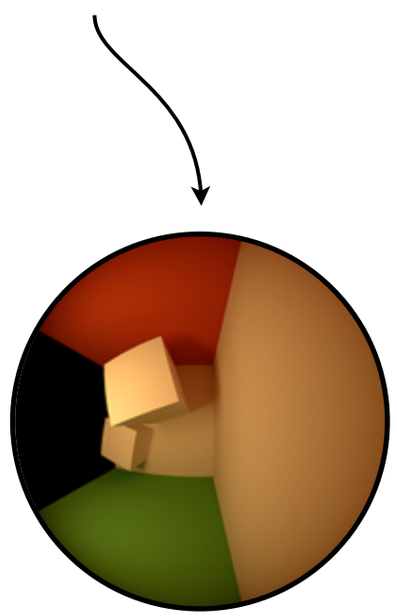




Solving the Rendering Equation

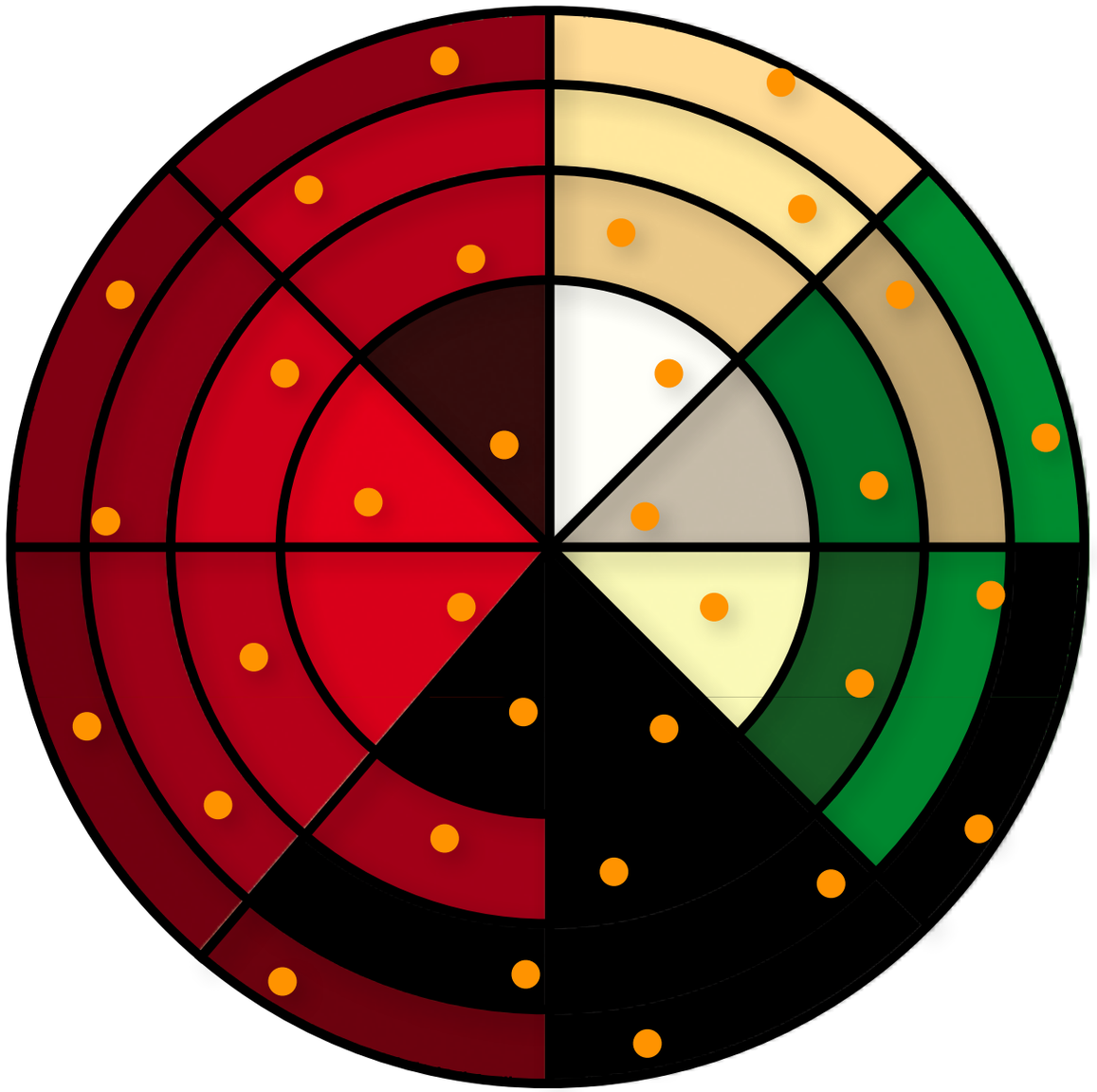
$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + \int_{H^2} f_r(\mathbf{x}, \vec{\omega}', \vec{\omega}) L(r(\mathbf{x}, \vec{\omega}'), -\vec{\omega}') \cos\theta' d\vec{\omega}'$$

How do we get these?

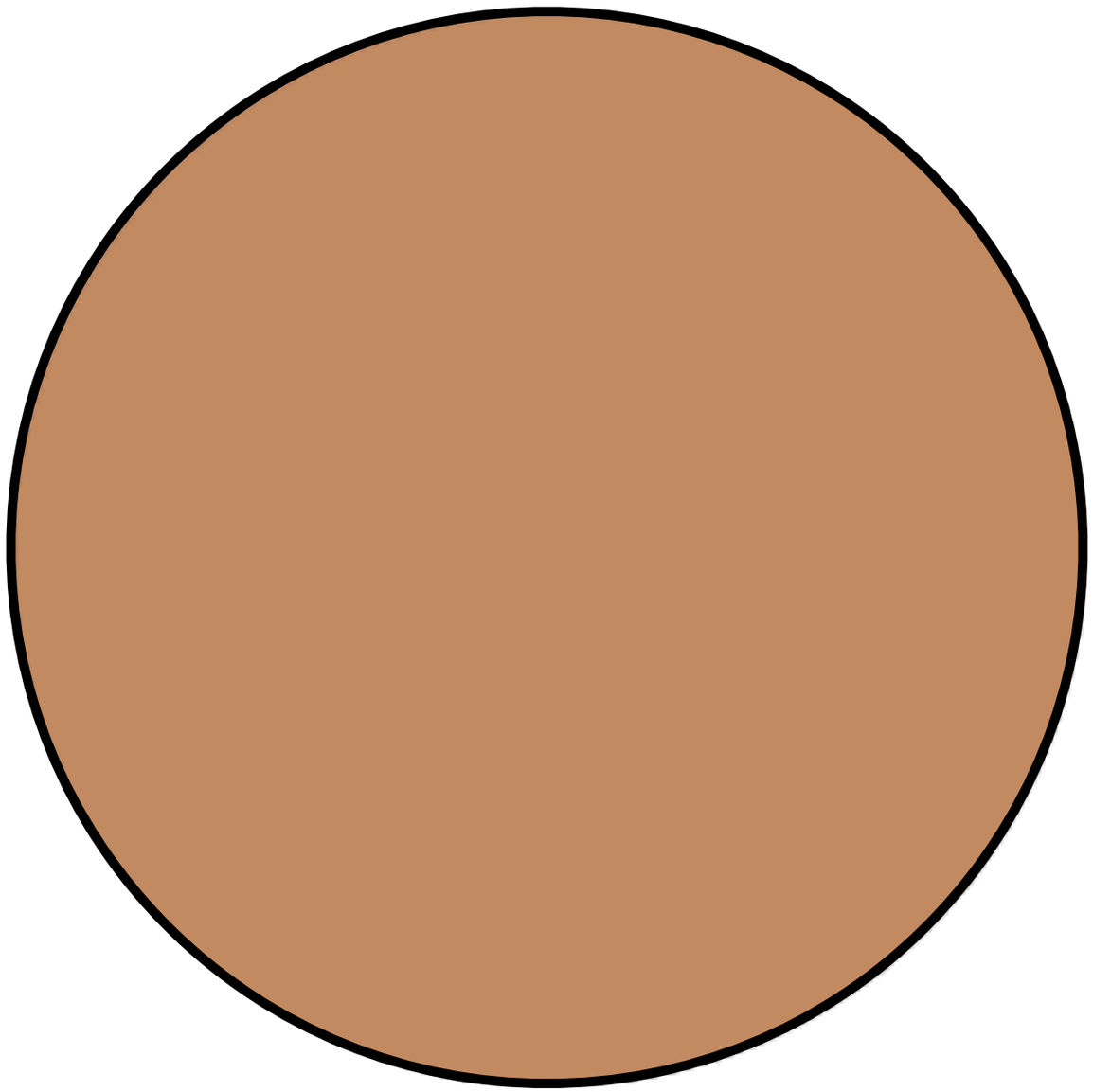


Recursive Monte Carlo Ray Tracing

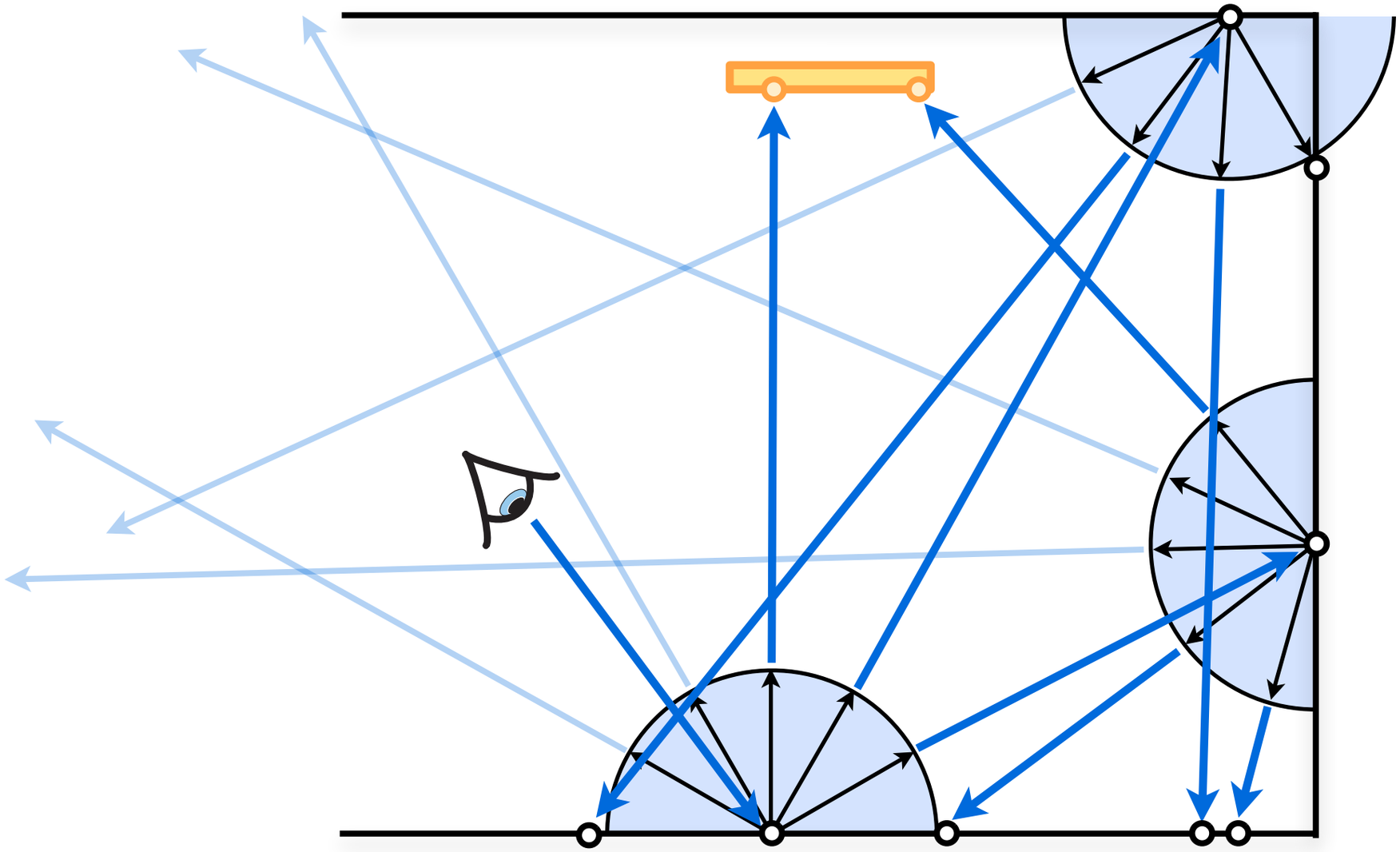
Solving the Rendering Equation



Solving the Rendering Equation



Recursive Monte Carlo Ray Tracing



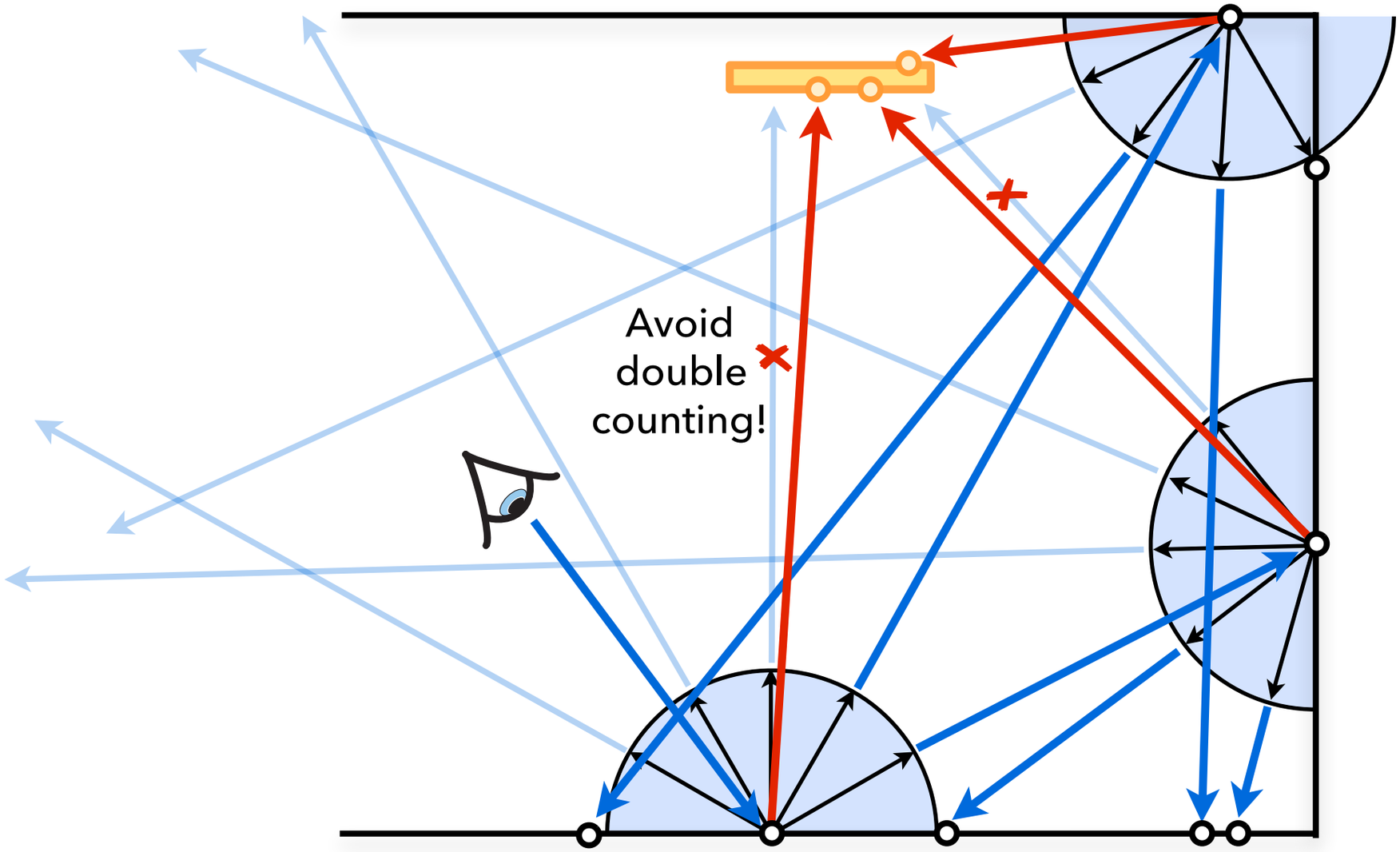
$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + \int_{H^2} f_r(\vec{\mathbf{x}}, \vec{\omega}', \vec{\omega}) L(r(\mathbf{x}, \vec{\omega}'), -\vec{\omega}') \cos\theta' d\vec{\omega}'$$

$$L(\mathbf{x}, \vec{\omega}) \approx L_e(\mathbf{x}, \vec{\omega}) + \frac{1}{N} \sum_{i=1}^{N} \frac{f_r(\vec{\mathbf{x}}, \vec{\omega}', \vec{\omega}) L(r(\mathbf{x}, \vec{\omega}'), -\vec{\omega}') \cos\theta'}{p(\vec{\omega}')}$$

Partitioning the Integrand

- Direct illumination: sometimes better estimated by sampling emissive surfaces
- Let's estimate direct illumination separately from indirect illumination, then add the two
  - i.e. shoot shadow rays (direct lighting) and gather rays (indirect lighting)
  - be careful *not to double-count*!
- Explicit estimation of direct illumination is also known as *next-event estimation*

Recursive Monte Carlo Ray Tracing



with explicit shadow rays

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + L_d(\mathbf{x}, \vec{\omega}) + L_i(\mathbf{x}, \vec{\omega})$$

$$L(\mathbf{x}, \vec{\omega}) = L_e + \int_A \ldots L_e \ldots dA(\mathbf{x}') + \int_{H^2} \ldots L \ldots d\vec{\omega}'$$

Monte Carlo Ray Tracing Algorithm

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + L_d(\mathbf{x}, \vec{\omega}) + L_i(\mathbf{x}, \vec{\omega})$$

```
color shade (point x, normal n)
{
    return Le + Ld + Li;
}
```

Monte Carlo Ray Tracing Algorithm

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + L_d(\mathbf{x}, \vec{\omega}) + L_i(\mathbf{x}, \vec{\omega})$$

```
color shade (point x, normal n)
{
    for all lights  // direct illumination
        Ld += contribution from light;

    return Le + Ld + Li;
}
```

Monte Carlo Ray Tracing Algorithm

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + L_d(\mathbf{x}, \vec{\omega}) + L_i(\mathbf{x}, \vec{\omega})$$

```
color shade (point x, normal n)
{
    for all lights  // direct illumination
        Ld += contribution from light;
    for all N indirect sample rays  // indirect illumination
        ω' = random direction in hemisphere above n;
        Li += brdf * shade(trace(x, ω')) * dot(n, ω') / (p(ω') * N);

    return Le + Ld + Li;
}
```

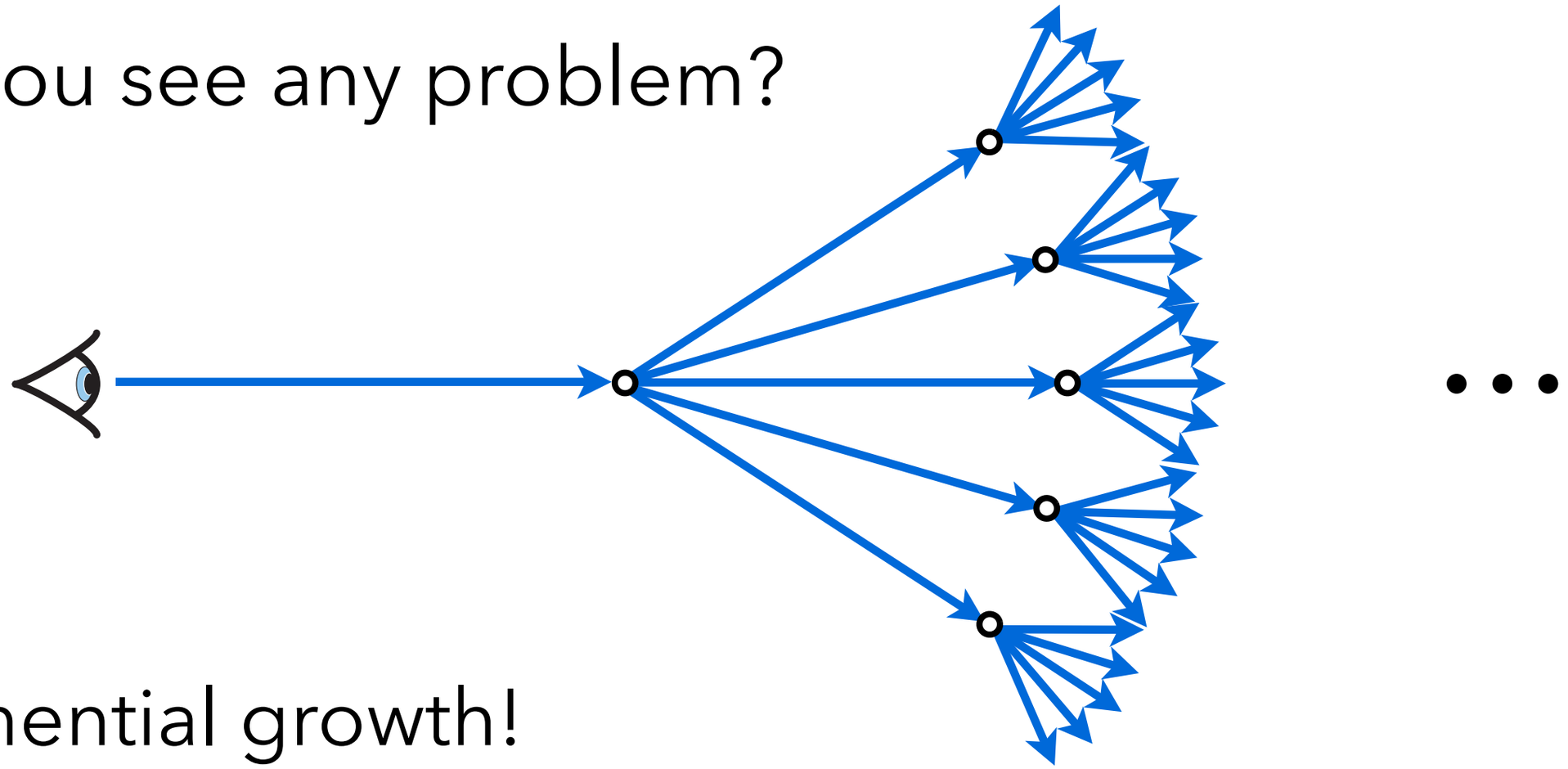
Monte Carlo Ray Tracing Algorithm

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + L_d(\mathbf{x}, \vec{\omega}) + L_i(\mathbf{x}, \vec{\omega})$$

```
color shade (point x, normal n)
{
    for all lights  // direct illumination
        Ld += contribution from light;

    for all N indirect sample rays  // indirect illumination
        ω' = random direction in hemisphere above n;
        Li += brdf * shade(trace(x, ω')) * dot(n, ω') / (p(ω') * N);

    if last bounce not specular  // prevent double-counting

        return Ld + Li;

    return Le + Ld + Li;
}
```

Monte Carlo Ray Tracing Algorithm

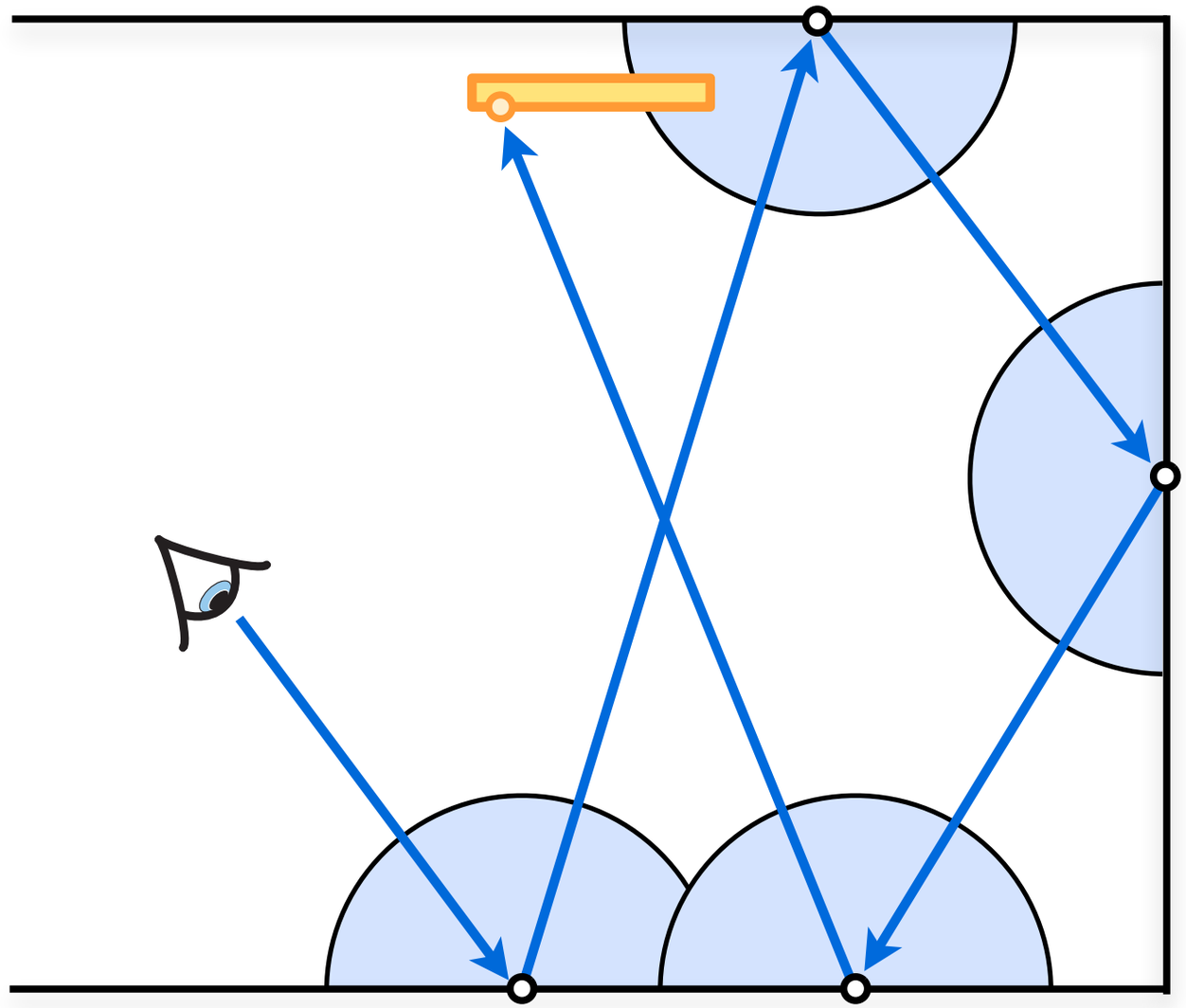
- Formulation of L is recursive!
- Can you see any problem?



- Exponential growth!
- 3-bounce contributes less than 1-bounce transport, but we estimate it with 25× as many samples!

Path Tracing

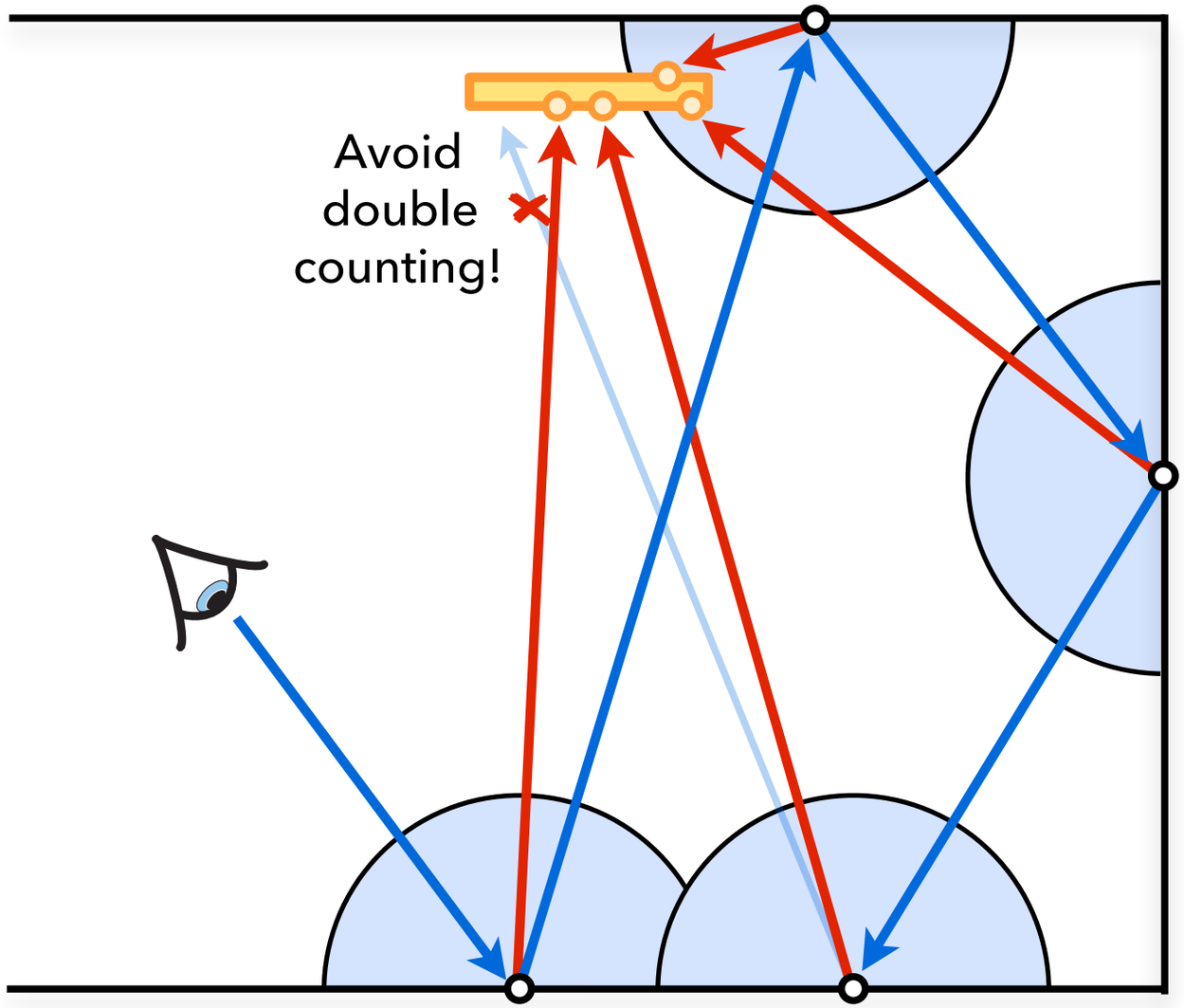
Path Tracing



$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + \int_{H^2} f_r(\vec{\mathbf{x}}, \vec{\omega}', \vec{\omega}) L(r(\mathbf{x}, \vec{\omega}'), -\vec{\omega}') \cos\theta' d\vec{\omega}'$$

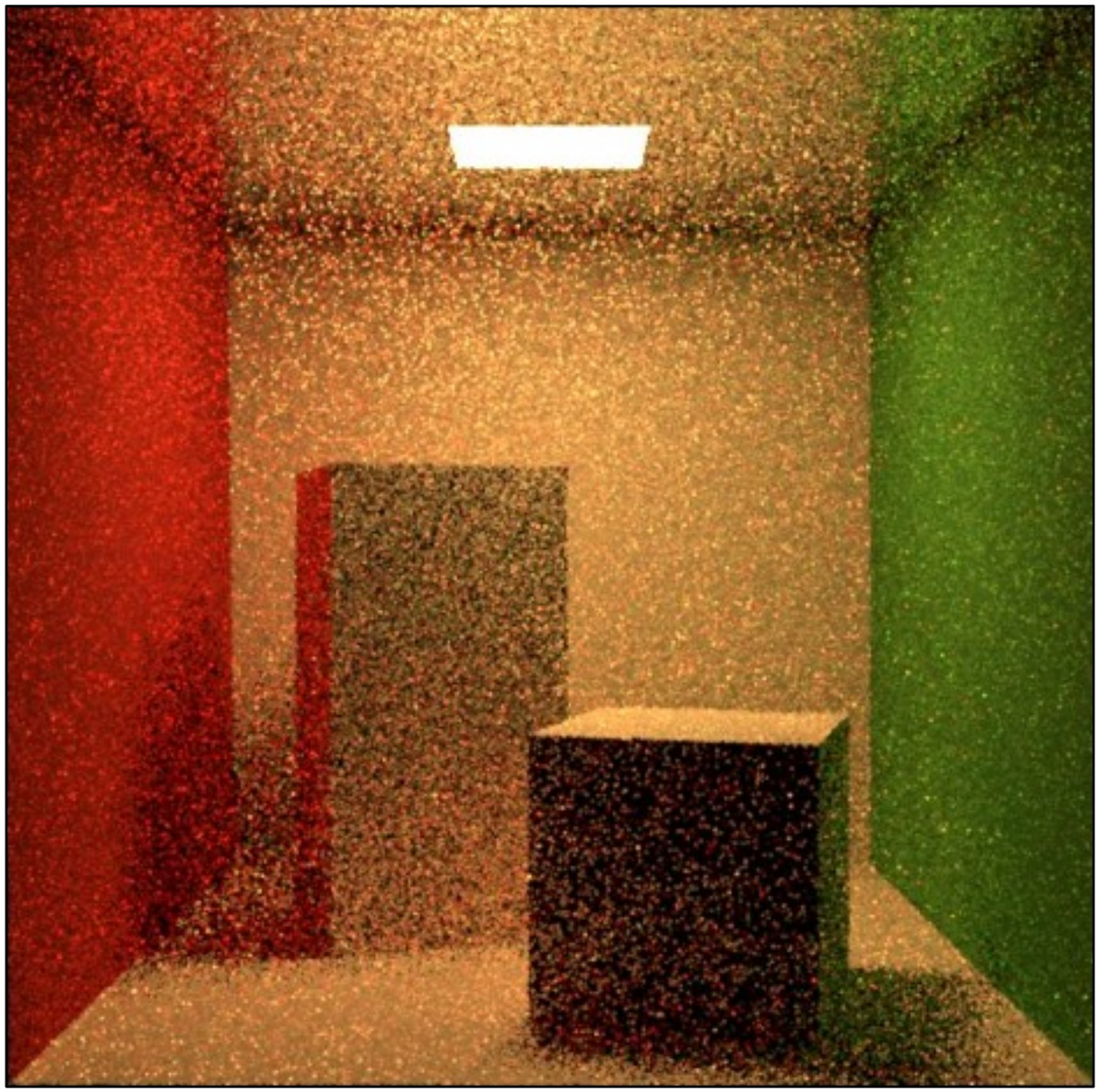
$$L(\mathbf{x}, \vec{\omega}) \approx L_e(\mathbf{x}, \vec{\omega}) + \frac{f_r(\vec{\mathbf{x}}, \vec{\omega}', \vec{\omega}) L(r(\mathbf{x}, \vec{\omega}'), -\vec{\omega}') \cos\theta'}{p(\vec{\omega}')}$$

Path Tracing with Shadow Rays



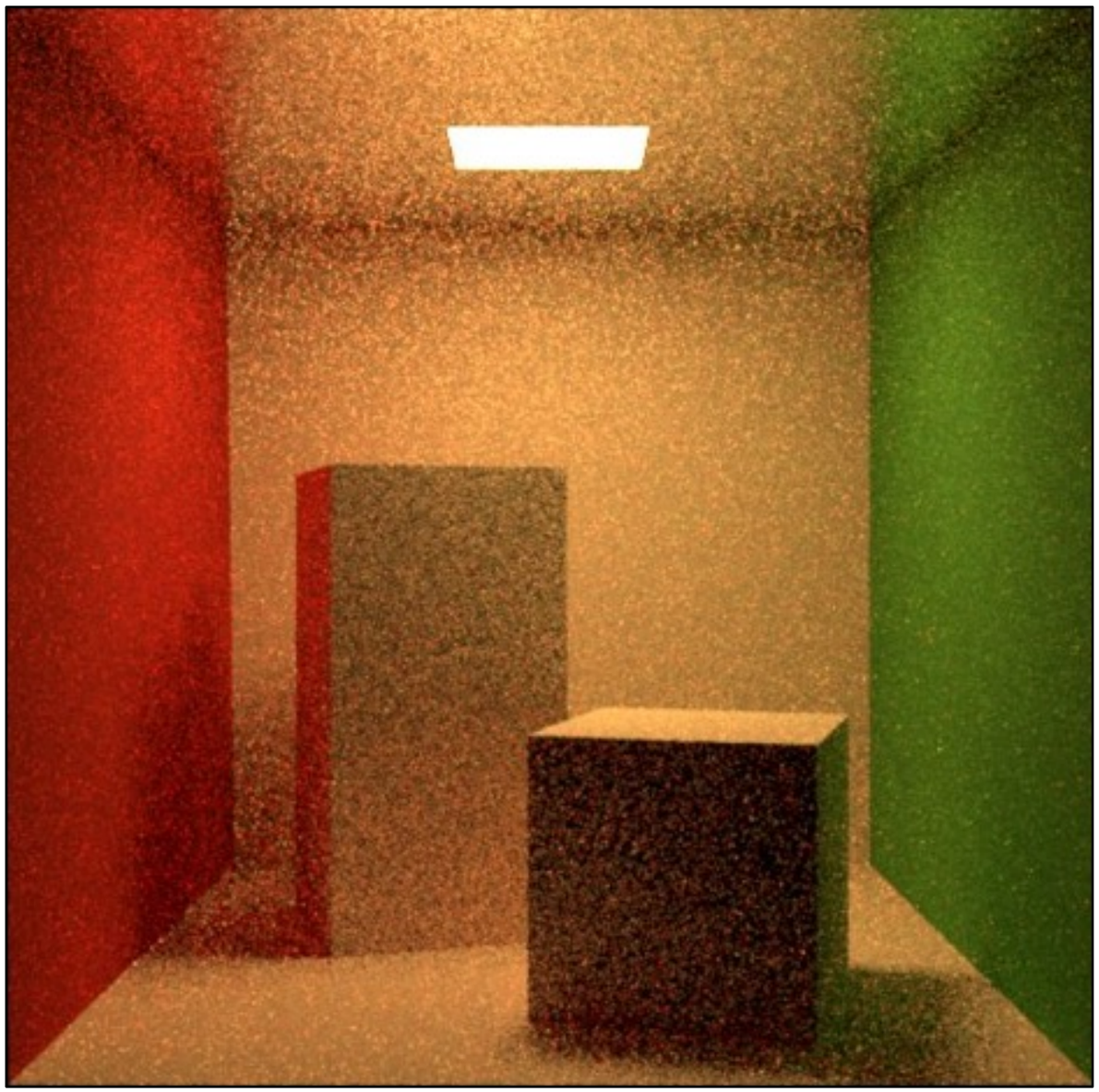
$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + \int_A \ldots L_e \ldots dA(\mathbf{x}') + \int_{H^2} \ldots L \ldots d\vec{\omega}'$$

Path Tracing with Shadow Rays



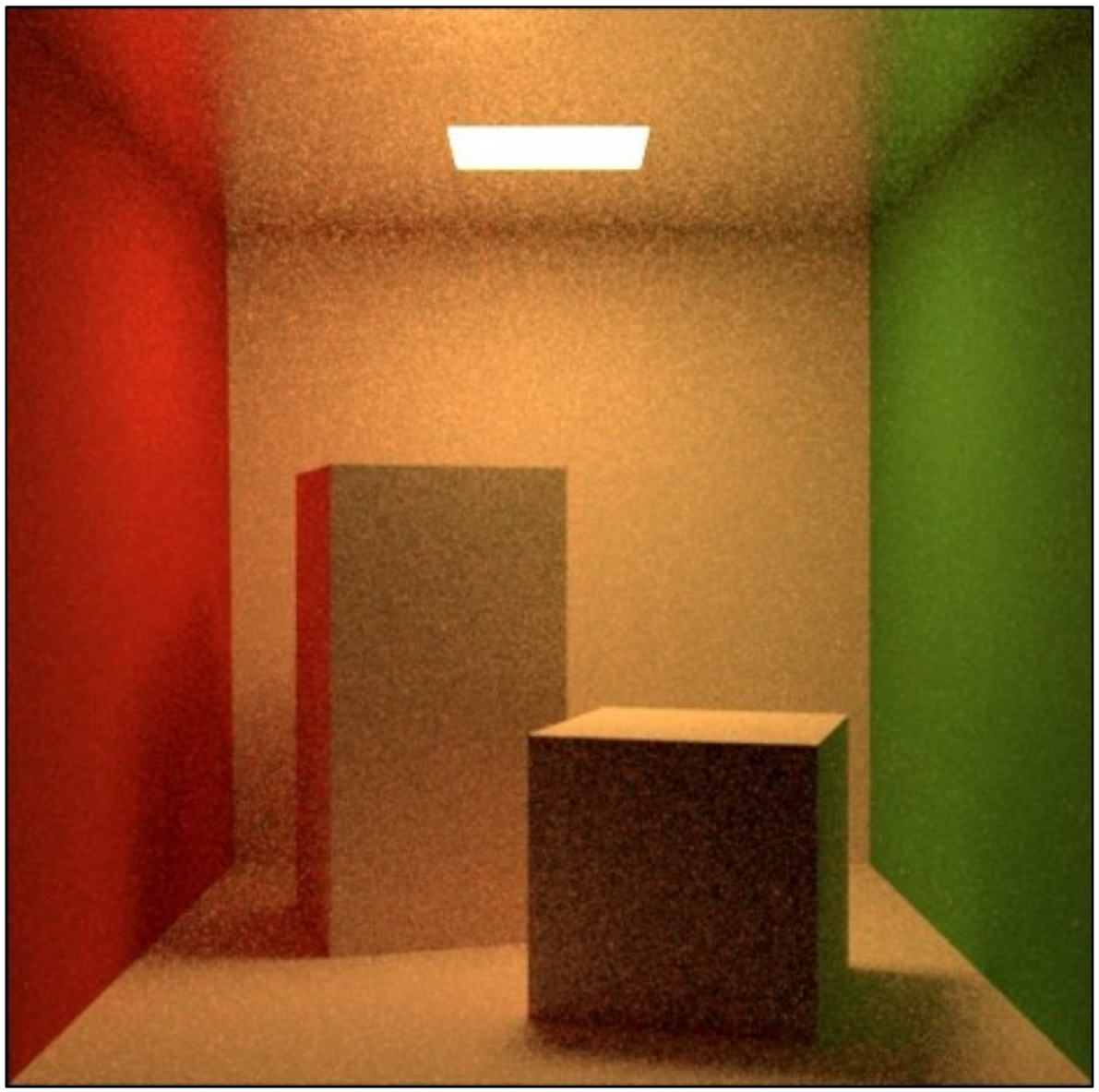
1 path/pixel

Path Tracing with Shadow Rays



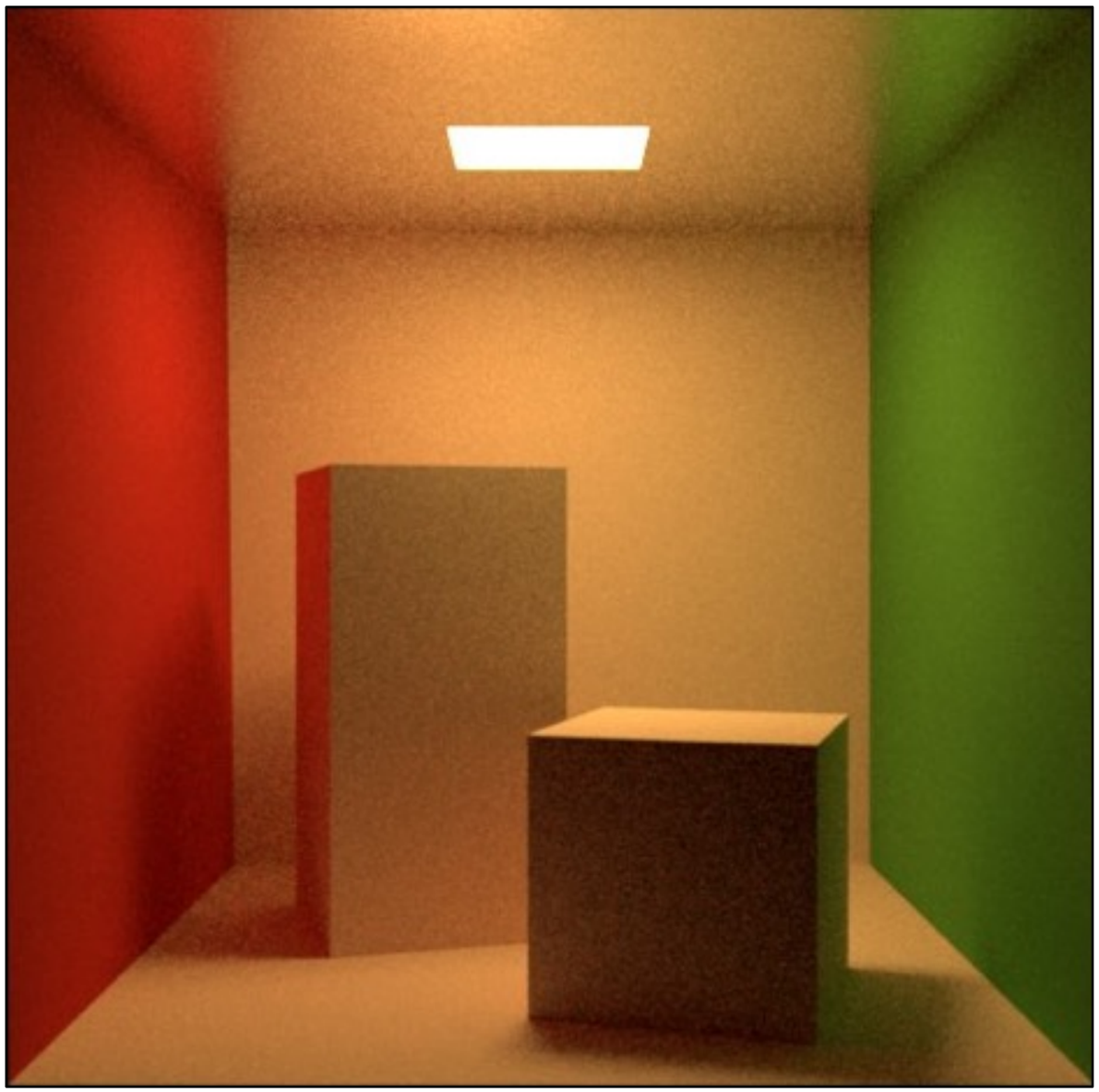
4 paths/pixel

Path Tracing with Shadow Rays



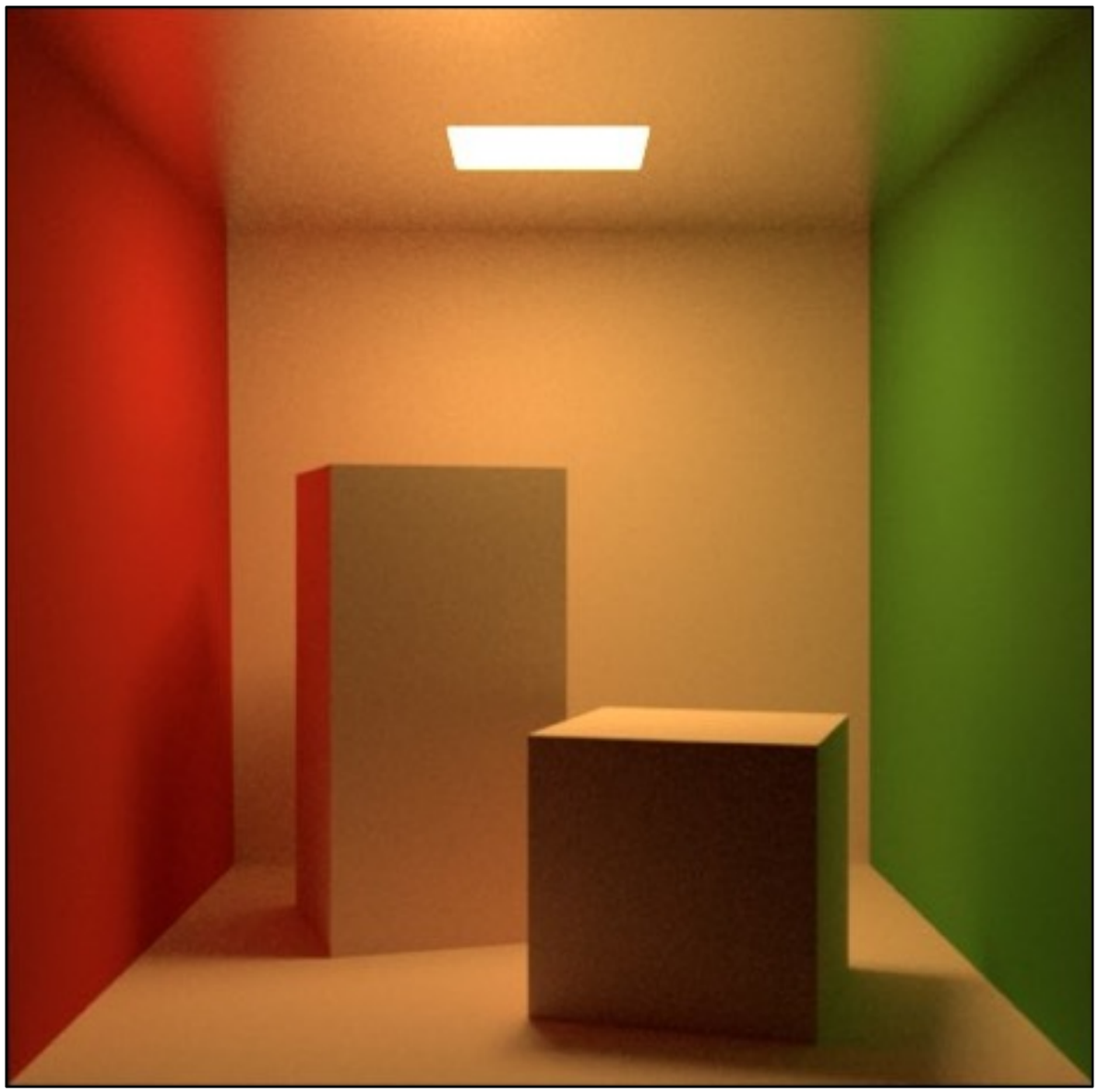
16 paths/pixel

Path Tracing with Shadow Rays



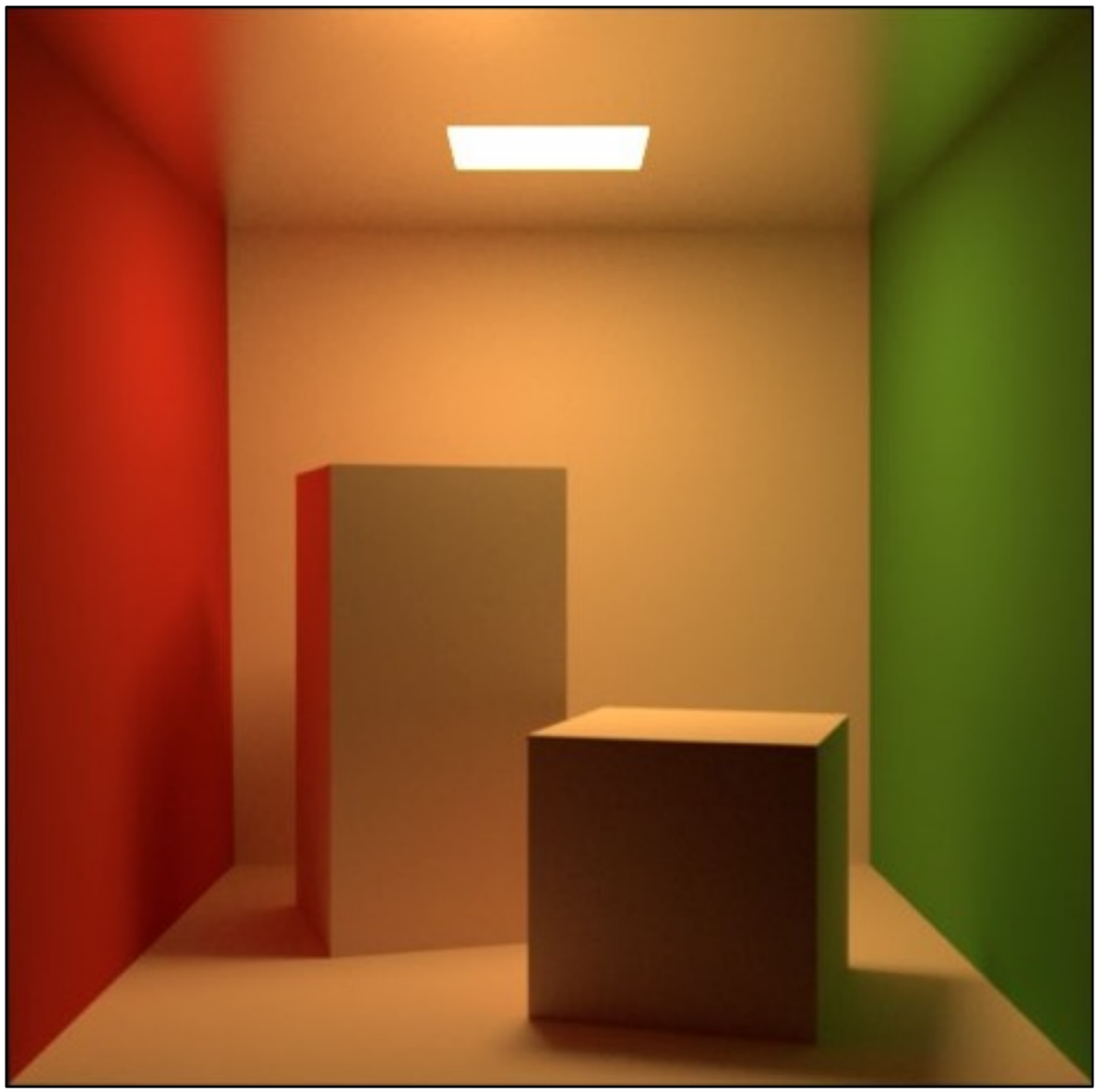
64 paths/pixel

Path Tracing with Shadow Rays



256 paths/pixel

Path Tracing with Shadow Rays



1024 paths/pixel

Monte Carlo Ray Tracing Algorithm

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + L_d(\mathbf{x}, \vec{\omega}) + L_i(\mathbf{x}, \vec{\omega})$$

```
color shade (point x, normal n)
{
    for all lights  // direct illumination
        Ld += contribution from light;

    for all N indirect sample rays  // indirect illumination
        ω' = random direction in hemisphere above n;
        Li += brdf * shade(trace(x, ω')) * dot(n, ω') / (p(ω') * N);

    if last bounce not specular  // prevent double-counting

        return Ld + Li;

    return Le + Ld + Li;
}
```

Monte Carlo Ray Tracing Algorithm

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + L_d(\mathbf{x}, \vec{\omega}) + L_i(\mathbf{x}, \vec{\omega})$$

```
color shade (point x, normal n)
{
    for all lights  // direct illumination
        Ld += contribution from light;

    for all N indirect sample rays  // indirect illumination
        ω' = random direction in hemisphere above n;
        Li += brdf * shade(trace(x, ω')) * dot(n, ω') / (p(ω') * N);

    if last bounce not specular  // prevent double-counting

        return Ld + Li;

    return Le + Ld + Li;
}
```

Path Tracing Algorithm

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + L_d(\mathbf{x}, \vec{\omega}) + L_i(\mathbf{x}, \vec{\omega})$$

```
color shade (point x, normal n)
{
    for all lights  // direct illumination
        Ld += contribution from light;

    // indirect illumination
    ω' = random direction in hemisphere above n;
    Li += brdf * shade(trace(x, ω')) * dot(n, ω') / (p(ω'));

    if last bounce not specular  // prevent double-counting

        return Ld + Li;

    return Le + Ld + Li;
}
```

Path Tracing

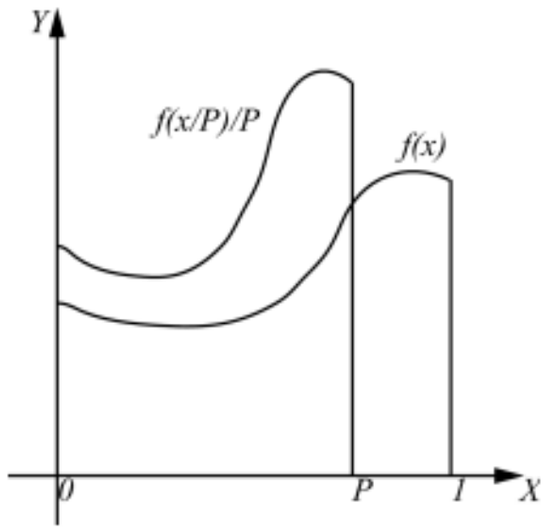
- Shoot multiple rays/pixel to improve quality of indirect illumination estimation
- Since we are already shooting multiple rays/pixel, we can achieve antialiasing/depth of field/motion blur at the same time “for free”!

Russian Roulette

- When do we stop recursion?
- Truncating at some fixed depth introduces *bias*
- Russian roulette:
  - probabilistically terminate the recursion
  - choose some termination probability $q \in (0, 1)$
  - generate a random number ξ

$$F' = \begin{cases} \frac{F}{1-q} & \xi > q \\ 0 & \text{otherwise} \end{cases}$$

$$E[F'] = (1-q) \cdot \left(\frac{E[F]}{1-q} \right) + q \cdot 0 = E[F]$$



Path Tracing Algorithm

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + L_d(\mathbf{x}, \vec{\omega}) + L_i(\mathbf{x}, \vec{\omega})$$

```
color shade (point x, normal n)
{
    for all lights  // direct illumination
        Ld += contribution from light;

    // indirect illumination
    ω' = random direction in hemisphere above n;
    Li += brdf * shade(trace(x, ω')) * dot(n, ω') / (p(ω'));

    if last bounce not specular  // prevent double-counting

        return Ld + Li;

    return Le + Ld + Li;
}
```

Path Tracing Algorithm

$$L(\mathbf{x}, \vec{\omega}) = L_e(\mathbf{x}, \vec{\omega}) + L_d(\mathbf{x}, \vec{\omega}) + L_i(\mathbf{x}, \vec{\omega})$$

```
color shade (point x, normal n)
{
    for all lights  // direct illumination
        Ld += contribution from light;

    if rand() > q  // indirect illumination
        ω' = random direction in hemisphere above n;
        Li += brdf * shade(trace(x, ω')) * dot(n, ω') / (p(ω'));

    if last bounce not specular  // prevent double-counting

        return Ld + Li / (1-q);

    return Le + Ld + Li / (1-q);
}
```

Path Tracing on 99 Lines of C++

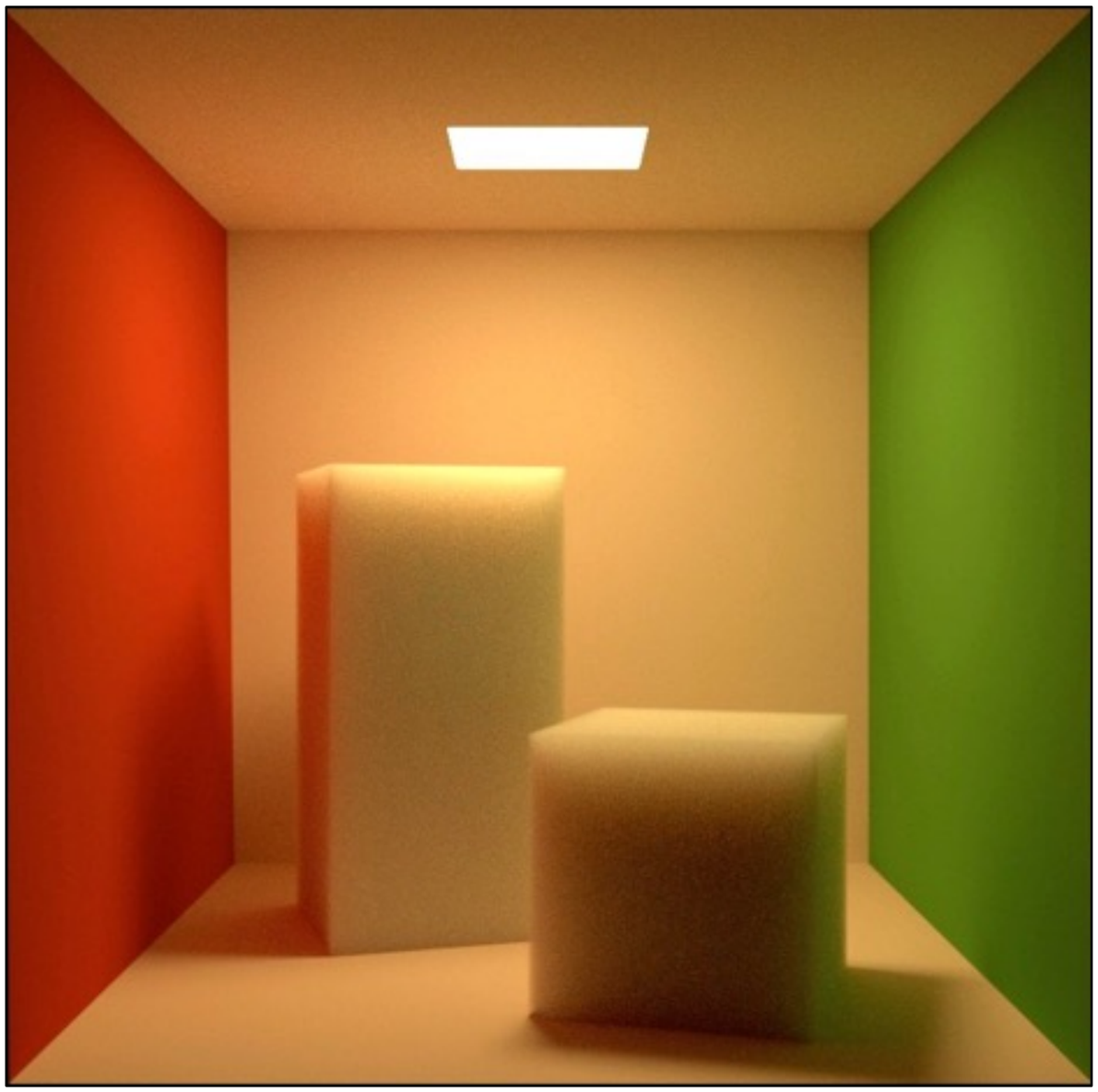
Minimalistic PT by Kevin Beason

```
#include <math.h>   // smallpt, a Path Tracer by Kevin Beason, 2008
#include <stdlib.h> // Make : g++ -O3 -fopenmp smallpt.cpp -o smallpt
#include <stdio.h>  //        Remove "-fopenmp" for g++ version < 4.2
struct Vec {        // Usage: time ./smallpt 5000 && xv image.ppm
  double x, y, z;                  // position, also color (r,g,b)
  Vec(double x_=0, double y_=0, double z_=0){ x=x_; y=y_; z=z_; }
  Vec operator+(const Vec &b) const { return Vec(x+b.x,y+b.y,z+b.z); }
  Vec operator-(const Vec &b) const { return Vec(x-b.x,y-b.y,z-b.z); }
  Vec operator*(double b) const { return Vec(x*b,y*b,z*b); }
  Vec mult(const Vec &b) const { return Vec(x*b.x,y*b.y,z*b.z); }
  Vec& norm(){ return *this = *this * (1/sqrt(x*x+y*y+z*z)); }
  double dot(const Vec &b) const { return x*b.x+y*b.y+z*b.z; } // cross:
  Vec operator%(Vec&b){return Vec(y*b.z-z*b.y,z*b.x-x*b.z,x*b.y-y*b.x);}
};
struct Ray { Vec o, d; Ray(Vec o_, Vec d_) : o(o_), d(d_) {} };
enum Refl_t { DIFF, SPEC, REFR };  // material types, used in radiance()
struct Sphere {
  double rad;       // radius
  Vec p, e, c;      // position, emission, color
  Refl_t refl;      // reflection type (DIFFuse, SPECular, REFRactive)
  Sphere(double rad_, Vec p_, Vec e_, Vec c_, Refl_t refl_):
    rad(rad_), p(p_), e(e_), c(c_), refl(refl_) {}
  double intersect(const Ray &r) const { // returns distance, 0 if nohit
    Vec op = p-r.o; // Solve t^2*d.d + 2*t*(o-p).d + (o-p).(o-p)-R^2 = 0
    double t, eps=1e-4, b=op.dot(r.d), det=b*b-op.dot(op)+rad*rad;
    if (det<0) return 0; else det=sqrt(det);
    return (t=b-det)>eps ? t : ((t=b+det)>eps ? t : 0);
  }
};
Sphere spheres[] = {//Scene: radius, position, emission, color, material
  Sphere(1e5, Vec( 1e5+1,40.8,81.6), Vec(),Vec(.75,.25,.25),DIFF),//Left
  Sphere(1e5, Vec(-1e5+99,40.8,81.6),Vec(),Vec(.25,.25,.75),DIFF),//Rght
  Sphere(1e5, Vec(50,40.8, 1e5),     Vec(),Vec(.75,.75,.75),DIFF),//Back
  Sphere(1e5, Vec(50,40.8,-1e5+170), Vec(),Vec(),           DIFF),//Frnt
  Sphere(1e5, Vec(50, 1e5, 81.6),    Vec(),Vec(.75,.75,.75),DIFF),//Botm
  Sphere(1e5, Vec(50,-1e5+81.6,81.6),Vec(),Vec(.75,.75,.75),DIFF),//Top
  Sphere(16.5,Vec(27,16.5,47),       Vec(),Vec(1,1,1)*.999, SPEC),//Mirr
  Sphere(16.5,Vec(73,16.5,78),       Vec(),Vec(1,1,1)*.999, REFR),//Glas
  Sphere(600, Vec(50,681.6-.27,81.6),Vec(12,12,12),  Vec(), DIFF) //Lite
};
inline double clamp(double x){ return x<0 ? 0 : x>1 ? 1 : x; }
inline int toInt(double x){ return int(pow(clamp(x),1/2.2)*255+.5); }
inline bool intersect(const Ray &r, double &t, int &id){
  double n=sizeof(spheres)/sizeof(Sphere), d, inf=t=1e20;
  for(int i=int(n);i--;) if((d=spheres[i].intersect(r))&&d<t){t=d;id=i;}
  return t<inf;
}
Vec radiance(const Ray &r, int depth, unsigned short *Xi){
  double t;                               // distance to intersection
  int id=0;                               // id of intersected object
  if (!intersect(r, t, id)) return Vec(); // if miss, return black
  const Sphere &obj = spheres[id];        // the hit object
  Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;
  double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z; // max refl
  if (++depth>5) if (erand48(Xi)<p) f=f*(1/p); else return obj.e; //R.R.
  if (obj.refl == DIFF){                  // Ideal DIFFUSE reflection
    double r1=2*M_PI*erand48(Xi), r2=erand48(Xi), r2s=sqrt(r2);
    Vec w=nl, u=((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm(), v=w%u;
    Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
    return obj.e + f.mult(radiance(Ray(x,d),depth,Xi));
  } else if (obj.refl == SPEC)            // Ideal SPECULAR reflection
    return obj.e + f.mult(radiance(Ray(x,r.d-n*2*n.dot(r.d)),depth,Xi));
  Ray reflRay(x, r.d-n*2*n.dot(r.d));     // Ideal dielectric REFRACTION
  bool into = n.dot(nl)>0;                // Ray from outside going in?
  double nc=1, nt=1.5, nnt=into?nc/nt:nt/nc, ddn=r.d.dot(nl), cos2t;
  if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)    // Total internal reflection
    return obj.e + f.mult(radiance(reflRay,depth,Xi));
  Vec tdir = (r.d*nnt - n*((into?1:-1)*(ddn*nnt+sqrt(cos2t)))).norm();
  double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
  double Re=R0+(1-R0)*c*c*c*c*c,Tr=1-Re,P=.25+.5*Re,RP=Re/P,TP=Tr/(1-P);
  return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ?   // Russian roulette
    radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP) :
    radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
}
int main(int argc, char *argv[]){
  int w=1024, h=768, samps = argc==2 ? atoi(argv[1])/4 : 1; // # samples
  Ray cam(Vec(50,52,295.6), Vec(0,-0.042612,-1).norm()); // cam pos, dir
  Vec cx=Vec(w*.5135/h), cy=(cx%cam.d).norm()*.5135, r, *c=new Vec[w*h];
#pragma omp parallel for schedule(dynamic, 1) private(r)       // OpenMP
  for (int y=0; y<h; y++){                       // Loop over image rows
    fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1));
    for (unsigned short x=0, Xi[3]={0,0,y*y*y}; x<w; x++)   // Loop cols
      for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++)     // 2x2 subpixel rows
        for (int sx=0; sx<2; sx++, r=Vec()){        // 2x2 subpixel cols
          for (int s=0; s<samps; s++){
            double r1=2*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1);
            double r2=2*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2);
            Vec d = cx*( ( (sx+.5 + dx)/2 + x)/w - .5) +
                    cy*( ( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;
            r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
          } // Camera rays are pushed ^^^^^ forward to start in interior
          c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
        }
  }
  FILE *f = fopen("image.ppm", "w");          // Write image to PPM file.
  fprintf(f, "P3\n%d %d\n%d\n", w, h, 255);
  for (int i=0; i<w*h; i++)
    fprintf(f,"%d %d %d ", toInt(c[i].x), toInt(c[i].y), toInt(c[i].z));
}
```

Questions?

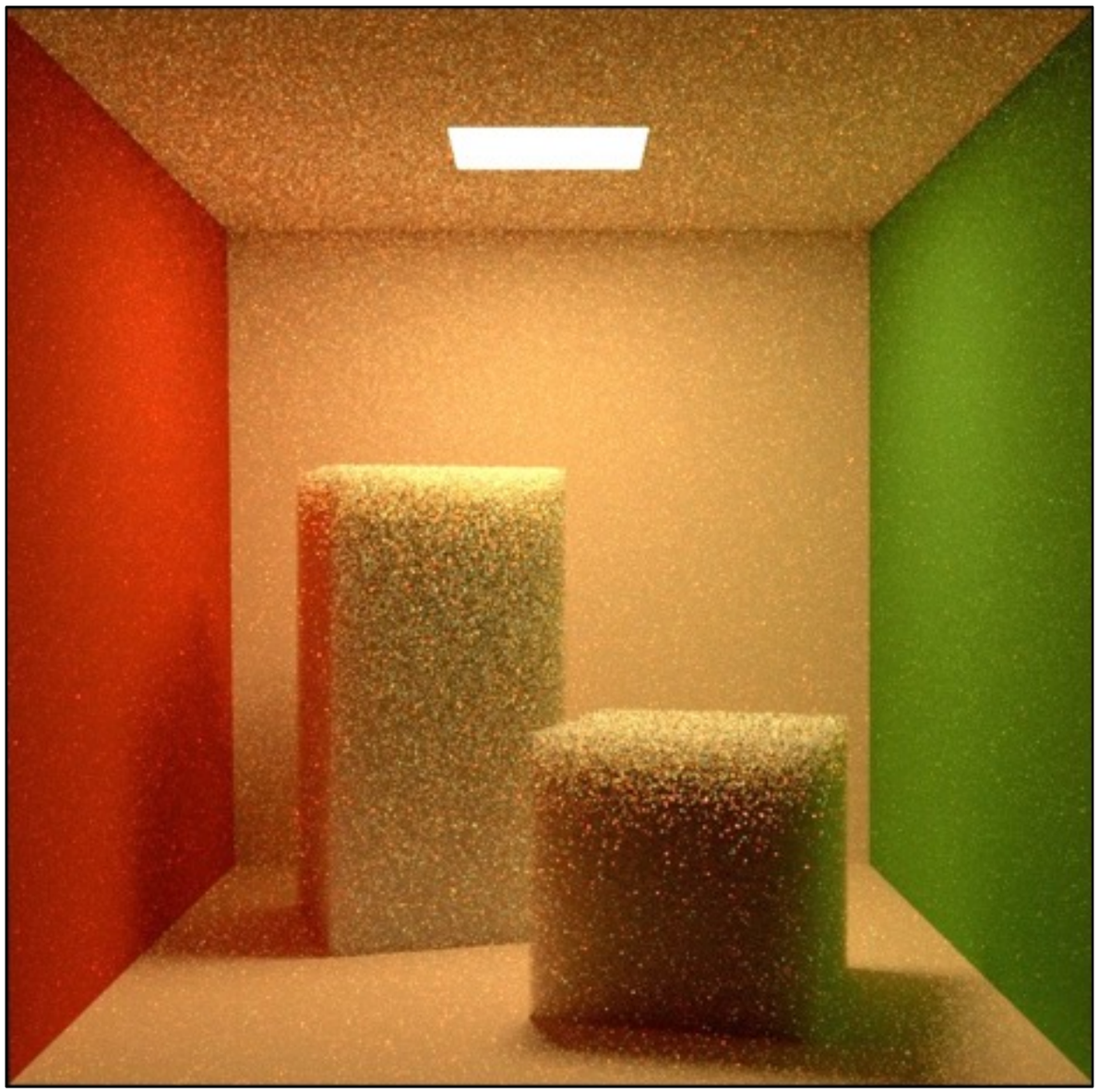
- Use MIS at least for direct illumination!

A Simple Example Scene



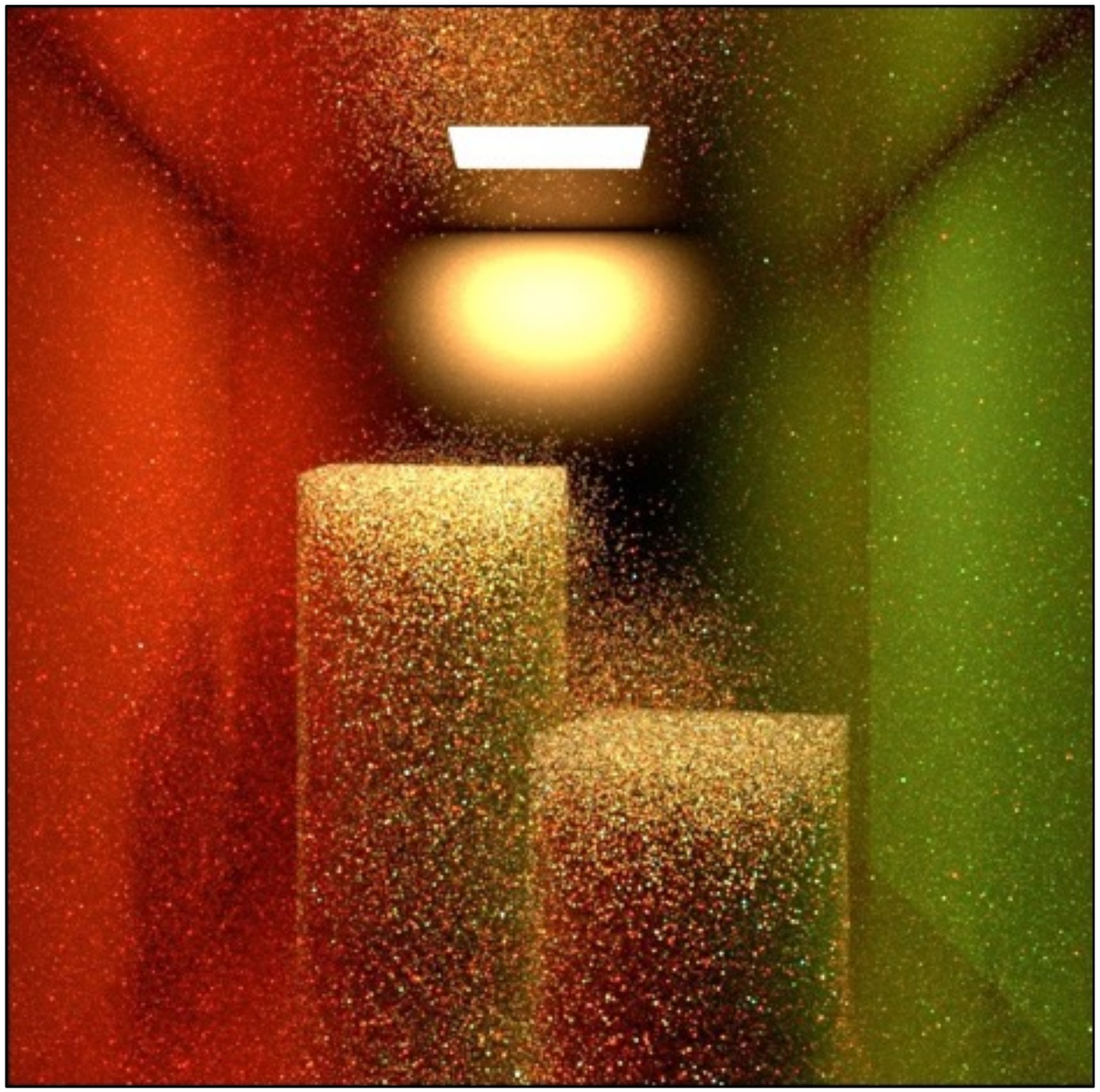
Scattering media with index matched boundary, i.e.$\eta = 1$

+ Dielectric Boundary

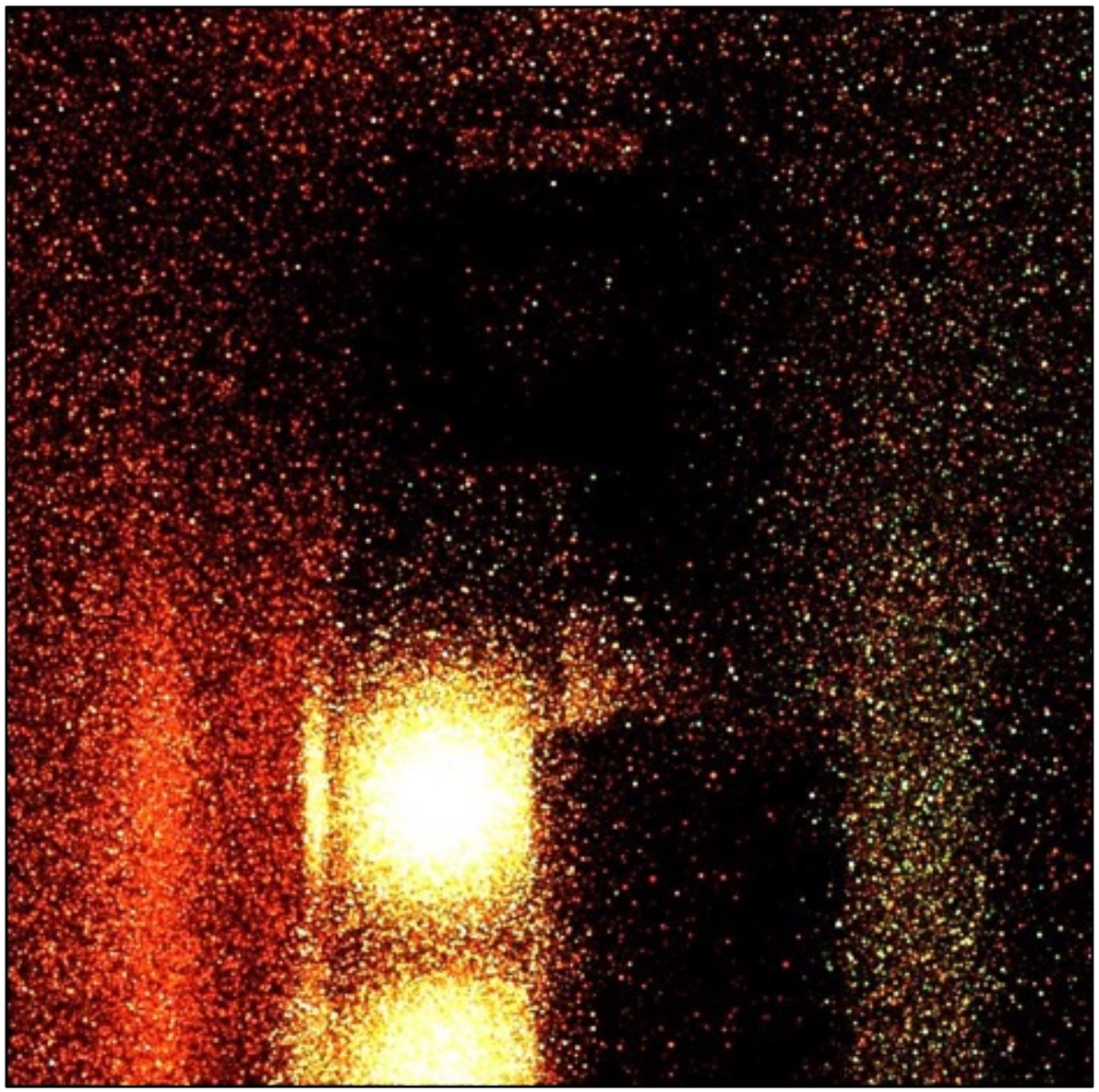


$$\eta = 1.00001$$

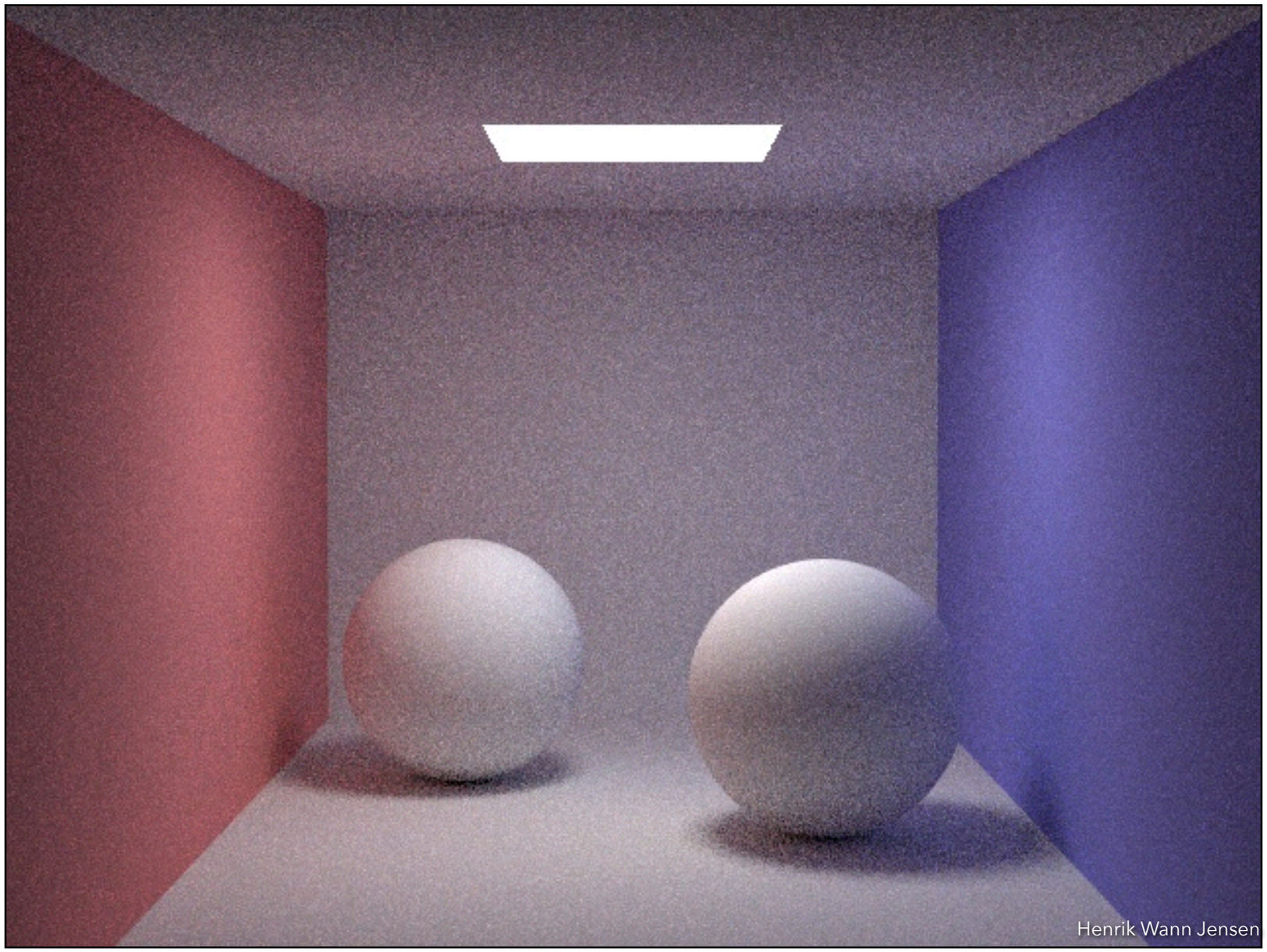
+ Glossy Materials



+ Inconveniently placed light source

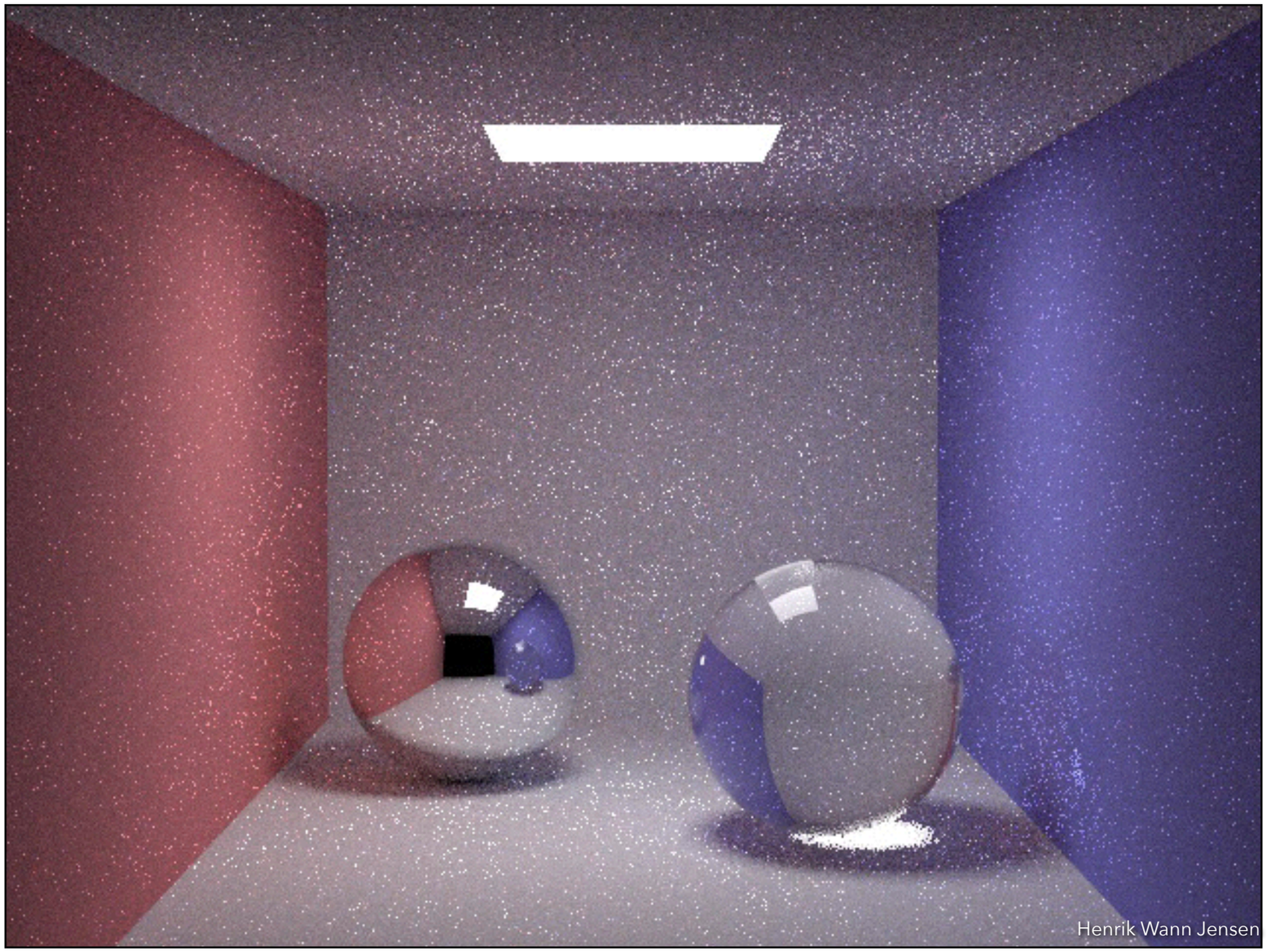


Another Simple Scene



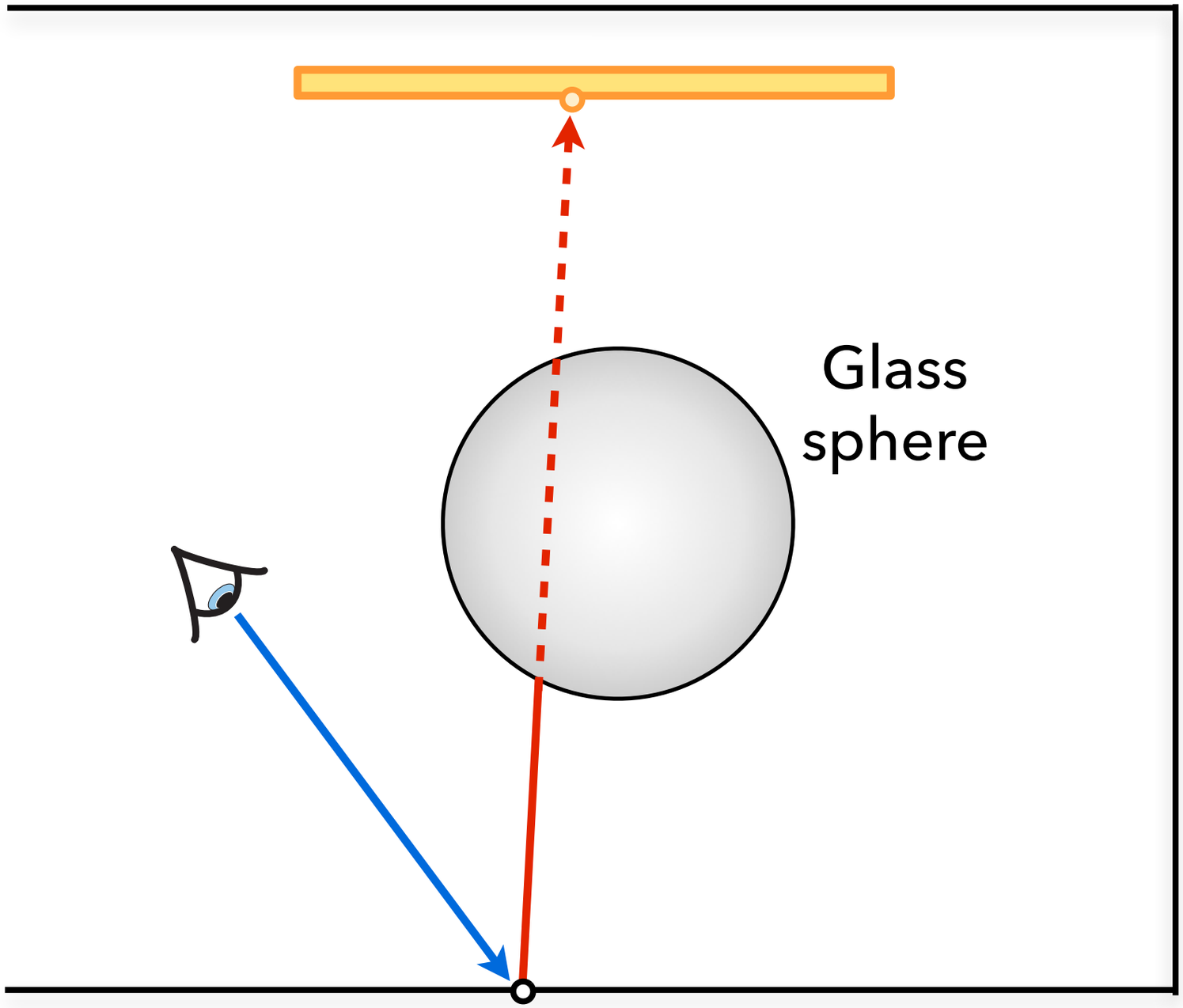
10 paths/pixel

+ Glass/Mirror Material

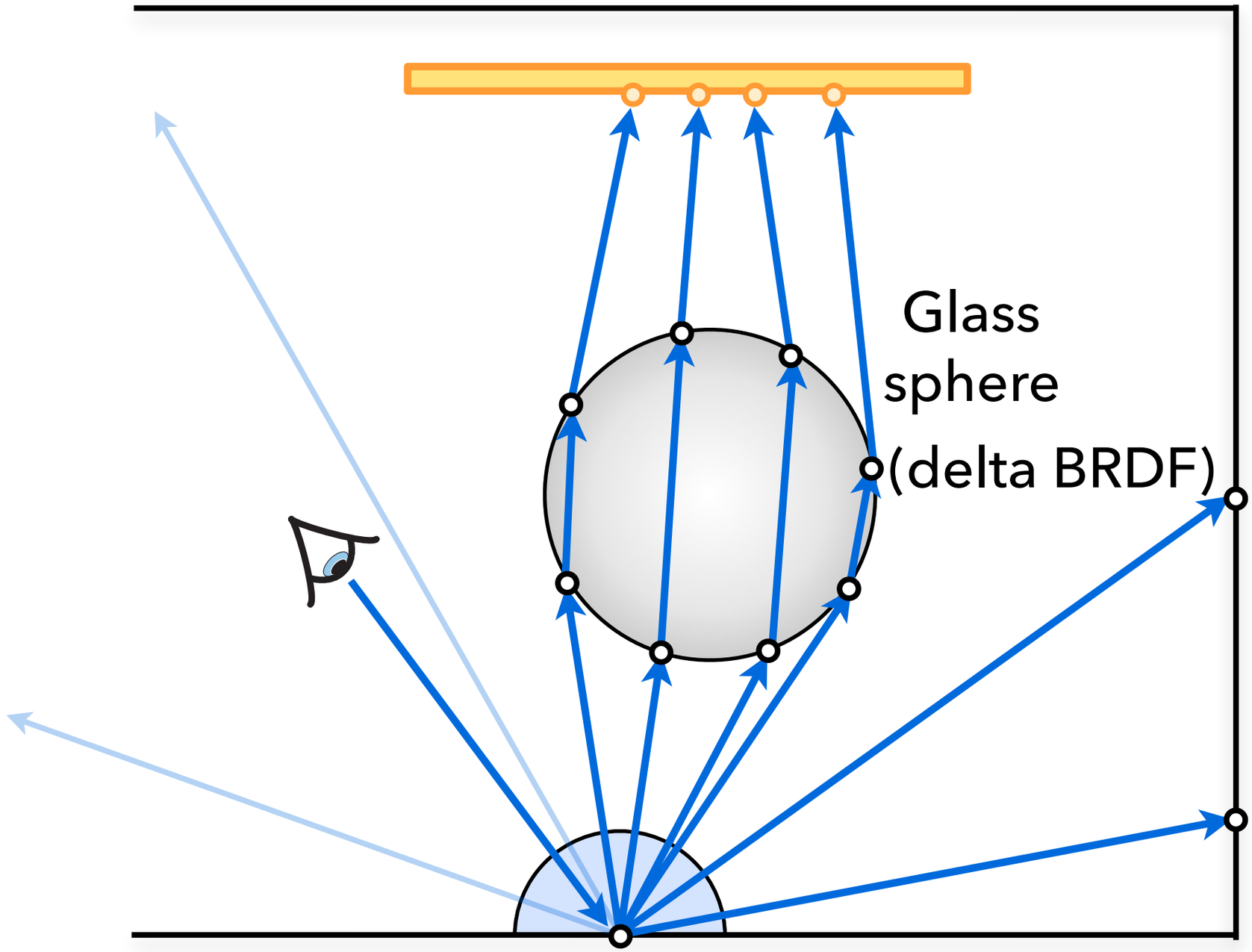


10 paths/pixel

Path Tracing Caustics

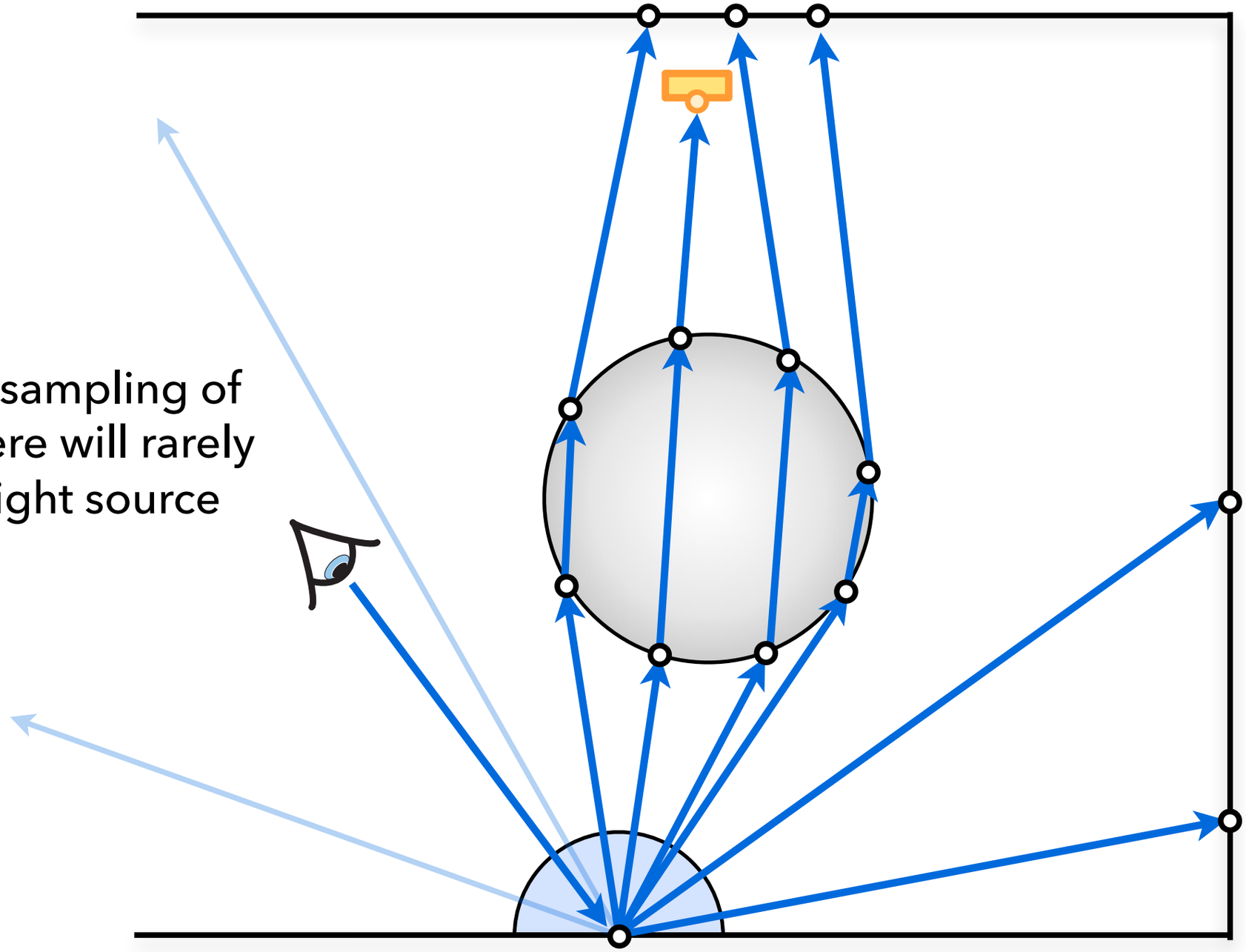


Path Tracing Caustics



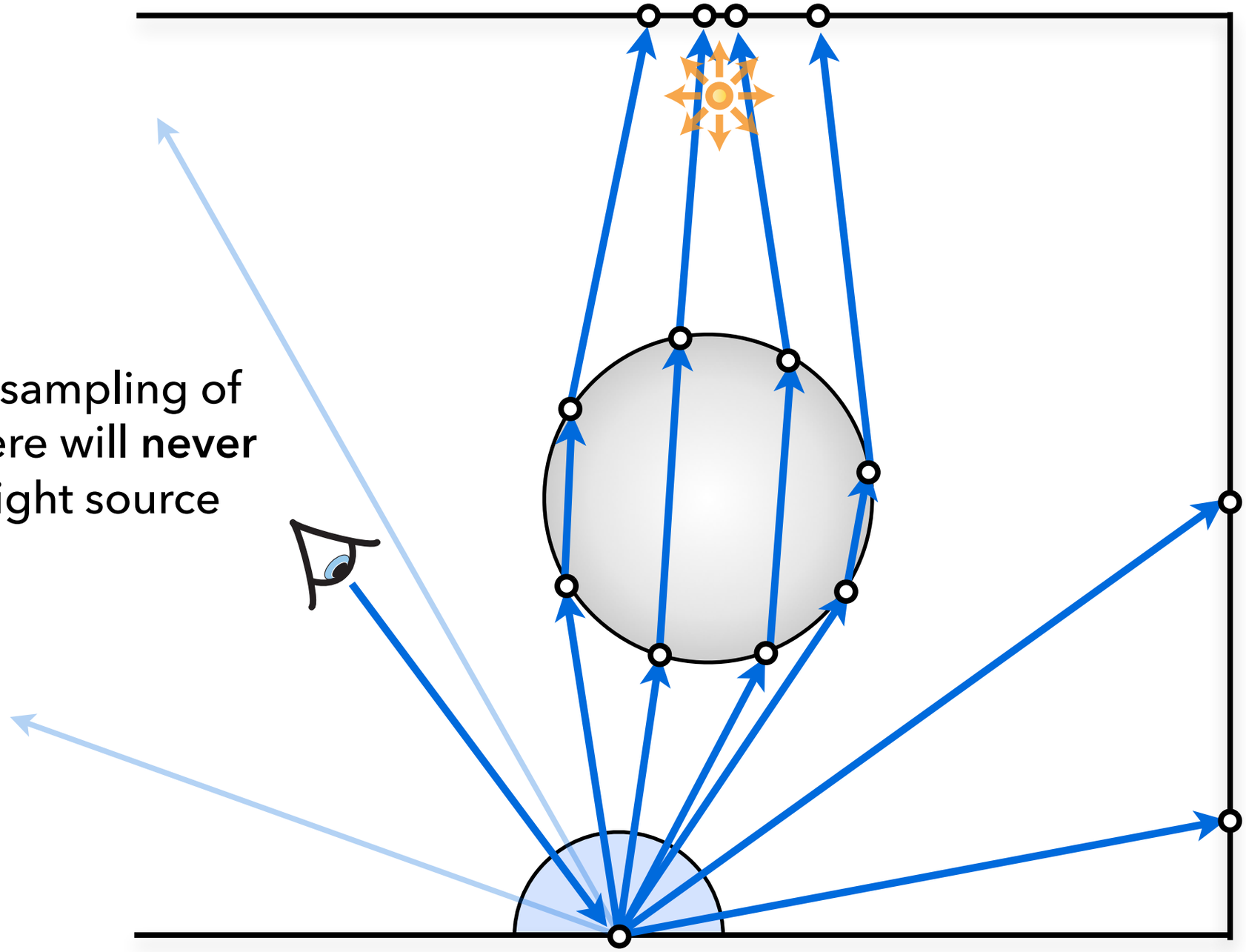
Path Tracing Caustics

Random sampling of hemisphere will rarely hit the light source



Path Tracing Caustics

Random sampling of hemisphere will **never** hit the light source

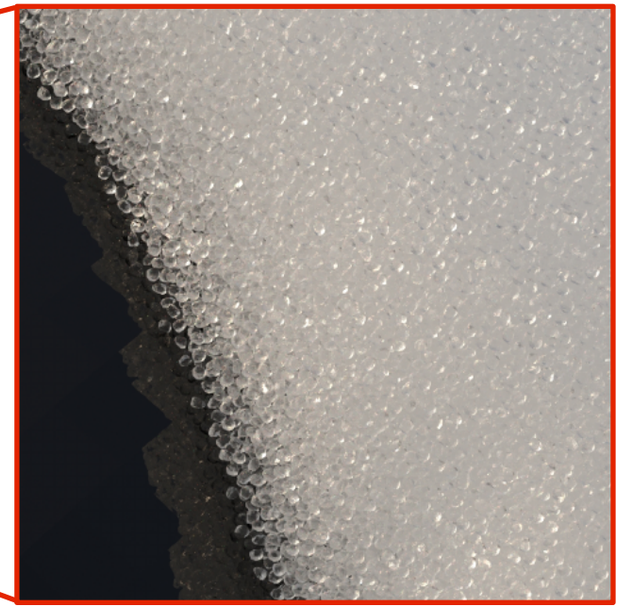


Let’s just give it more time...

- Nature ~ 2×10^{33} / second
- Fastest GPU ray tracer ~ 2×10^{8} / second

Tim Webber, Gravity VFX supervisor

Let’s just give it more time…



1 image ~ 8 core years
(parallelized on a cluster)

Path Tracing - Summary

✓ Full solution to the rendering equation

✓ Simple to implement

✘ Slow convergence

- requires 4x more samples to half the error

✘ Robustness issues

- does not handle some light paths well (or not at all), e.g. caustics ($LS+DE$)

✘ No reuse or caching of computation

✘ General sampling issue

- makes only locally good decisions