Advanced 3D graphics for movies and games (NPGR010)

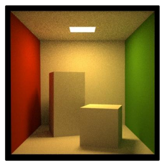
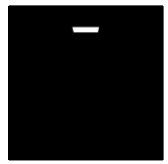
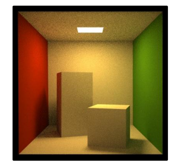
– Path tracing

Jiří Vorba, MFF UK/Weta Digital
jirka@cgg.mff.cuni.cz

Slides by prof. Jaroslav Křivánek, extended by Jiří Vorba

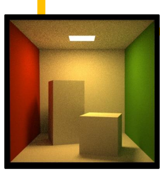
Rendering equation

$$L = L_{\mathrm{e}} + T \circ L$$

 $=$  $+ \; T \circ$ 

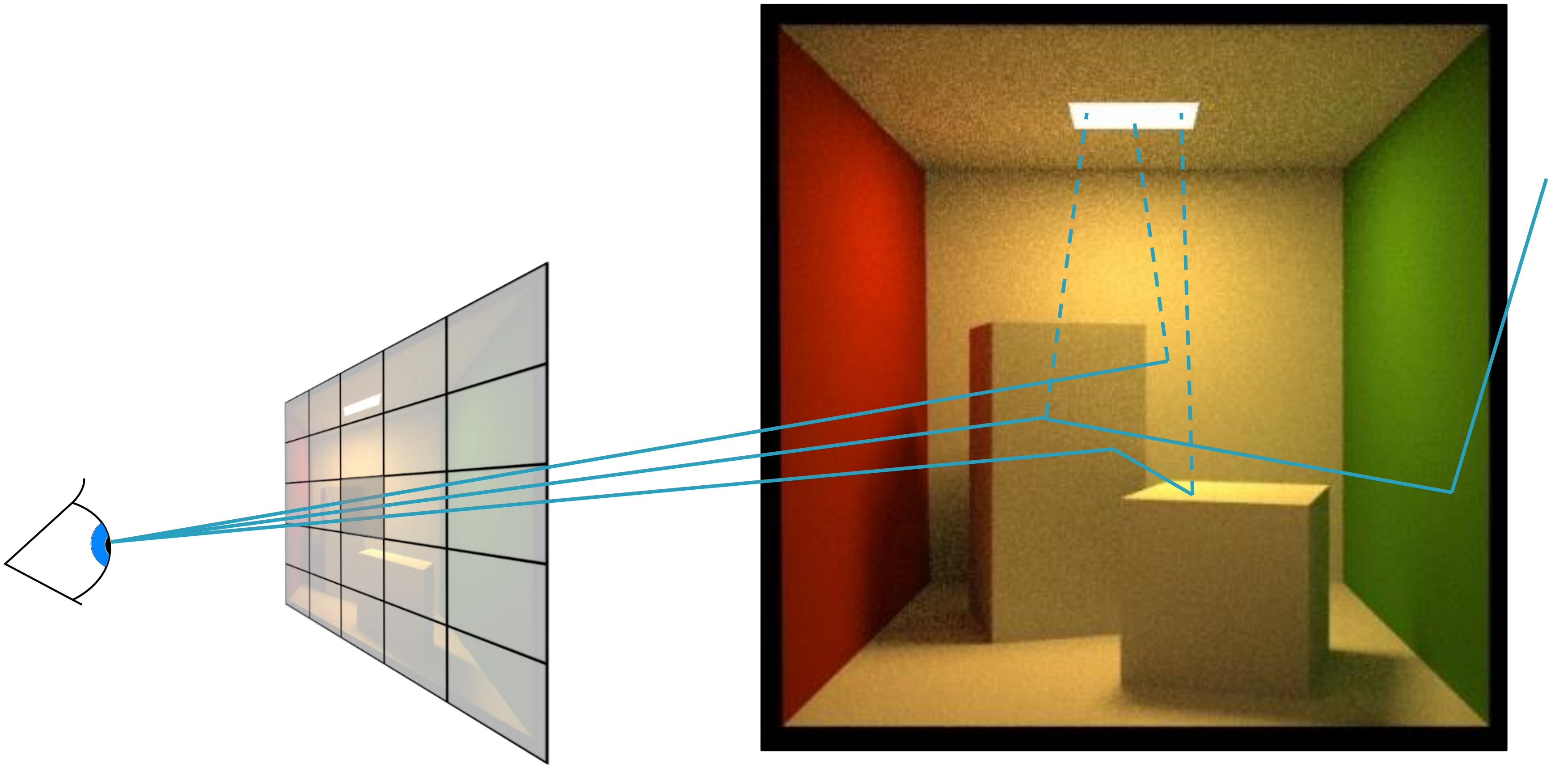
- **Solution**: Neumann series

$$L = L_{\mathrm{e}} + TL_{\mathrm{e}} + T^2L_{\mathrm{e}} + T^3L_{\mathrm{e}} + \ldots$$

Path tracing

Transport over many paths



Tracing paths from the (pinhole) camera

```
renderImage()
{
  for all pixels
  {
    Spectrum pixelColor = (0,0,0);
    for k = 1 to Np
    {
      ωk := random direction through the pixel
      pixelColor += estimateLin(cameraPosition, ωk)
    }
    pixelColor /= Np;
    writePixel(k, pixelColor);
  }
}
```

Tracing paths from the (pinhole) camera

- For progressive rendering, swap the loop nesting:

```
renderImage()
{
  for k = 1 to Np  // rendering "passes"
  {
    for all pixels
    {
     Spectrum pixelColor = (0,0,0);
     ...
```

Path tracing, v. 0.1

estimateLin (x, ω): // radiance incident at x from direction ω

y = findNearestIntersection(x, ω)

if (no intersection)

return backgroud.getLe (–ω) // emitted radiance from envmap

else

return getLe (y, –ω) + // emitted radiance (if y is on a light)

estimateLrefl (y, –ω)// reflected radiance

estimateLrefl(x, ω_{out}):

[ω_{in} , pdf] = genRandomDir(x, ω_{out}); // e.g. BRDF imp. sampling

return estimateLin(x, ω_{in}) * brdf(x, ω_{in}, ω_{out}) * dot($\mathbf{n}_x$, ω_{in}) / pdf

Path Tracing – Loop version

- Path tracing only has tail recursion
  - Can be unrolled into a loop for better efficiency

- New feature: “Russian Roulette” for unbiased path termination

$$L = \sum_{i=0}^{\boxed{M}} T^i L_\mathrm{e} \quad \Longrightarrow \quad L = \sum_{i=0}^{\boxed{\infty}} T^i L_\mathrm{e}$$

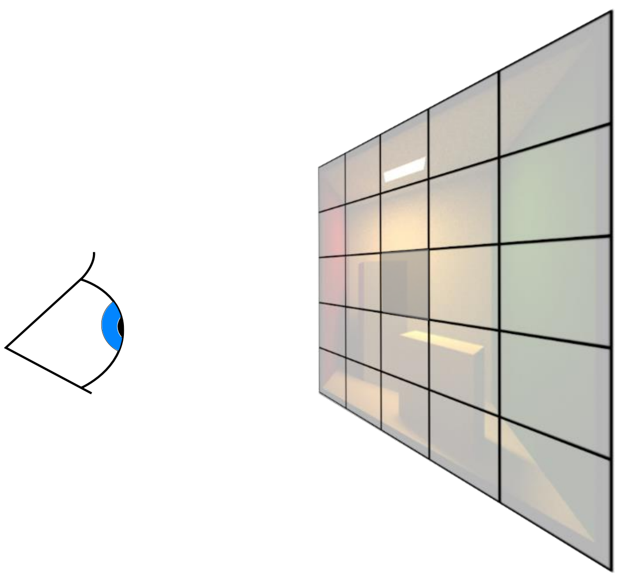
```
estimateLin(x, omegaInAtX)  // radiance incident at “x” from the direction “omegaInAtX”
{                           //  (“omegaInAtX” is pointing *away* from “x”)
  Spectrum throughput = (1,1,1)
  Spectrum accum  = (0,0,0)
  for i := 1 to maxLength // solve only first “maxLength” terms of Neumann series
  {
    hit = findNearestIntersection(x, omegaInAtX)

    if noIntersection(hit)            // ray leaves the scene – it “hits” the background
      return accum + throughput * bkgLight.getLe(x, - omegaInAtX)

    omegaOut := -omegaInAtX           // omegaOut at hit.pos
    if isOnLightSource(hit)           // ray happened to directly hit a light source
      accum += throughput * getLe(hit.pos, omegaOut)          // “pick up” emission
    // now estimate the reflected radiance
    [omegaIn, pdfIn] := generateRandomDir(hit)                // omegaIn at hit.pos
    throughput *= 1/pdfIn * brdf(hit.pos, omegaIn, omegaOut) * dot(hit.n, omegaIn)



     x := hit.pos                     // “recursion”
     omegaInAtX := omegaIn            // “recursion”


    }
    return accum;
}
```

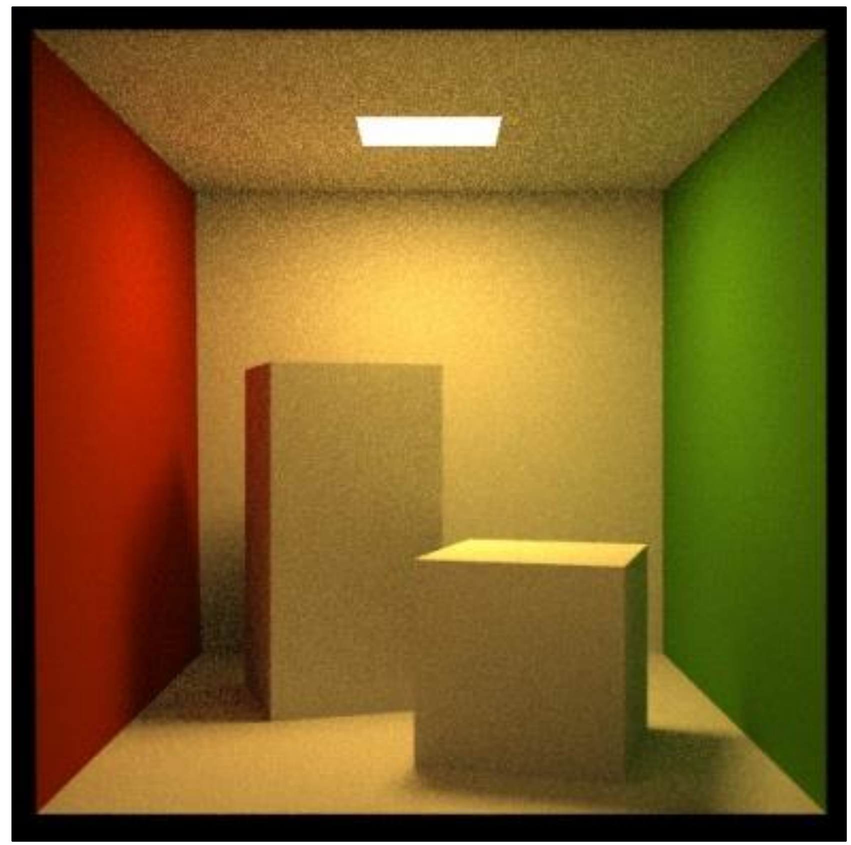
```
estimateLin(x, omegaInAtX)  // radiance incident at “x” from the direction “omegaInAtX”
{                           //  (“omegaInAtX” is pointing *away* from “x”)
  Spectrum throughput = (1,1,1)
  Spectrum accum  = (0,0,0)
  while(1)                  // we don’t cut off the path length now
  {
    hit = findNearestIntersection(x, omegaInAtX)

    if noIntersection(hit)             // ray leaves the scene – it “hits” the background
      return accum + throughput * bkgLight.getLe(x, - omegaInAtX)

    omegaOut := -omegaInAtX            // omegaOut at hit.pos
    if isOnLightSource(hit)            // ray happened to directly hit a light source
      accum += throughput * getLe(hit.pos, omegaOut)          // “pick up” emission
    // now estimate the reflected radiance
    [omegaIn, pdfIn] := generateRandomDir(hit)                // omegaIn at hit.pos
    throughput *= 1/pdfIn * brdf(hit.pos, omegaIn, omegaOut) * dot(hit.n, omegaIn)




      x := hit.pos                     // “recursion”
      omegaInAtX := omegaIn            // “recursion”



    }
    return accum;
}
```

```
estimateLin(x, omegaInAtX)  // radiance incident at "x" from the direction "omegaInAtX"
{                           //  ("omegaInAtX" is pointing *away* from "x")
  Spectrum throughput = (1,1,1)
  Spectrum accum  = (0,0,0)
  while(1)                  // we don't cut off the path length now
  {
    hit = findNearestIntersection(x, omegaInAtX)

    if noIntersection(hit)            // ray leaves the scene - it "hits" the background
      return accum + throughput * bkgLight.getLe(x, - omegaInAtX)

    omegaOut := -omegaInAtX           // omegaOut at hit.pos
    if isOnLightSource(hit)           // ray happened to directly hit a light source
      accum += throughput * getLe(hit.pos, omegaOut)          // "pick up" emission
    // now estimate the reflected radiance
    [omegaIn, pdfIn] := generateRandomDir(hit)                 // omegaIn at hit.pos
    throughput *= 1/pdfIn * brdf(hit.pos, omegaIn, omegaOut) * dot(hit.n, omegaIn)

    survivalProb = min(1, throughput.maxComponent())
    if rand() < survivalProb          // Russian Roulette - survive (reflect)
      throughput /= survivalProb
      x := hit.pos                    // "recursion"
      omegaInAtX := omegaIn           // "recursion"


    }
    return accum;
}
```

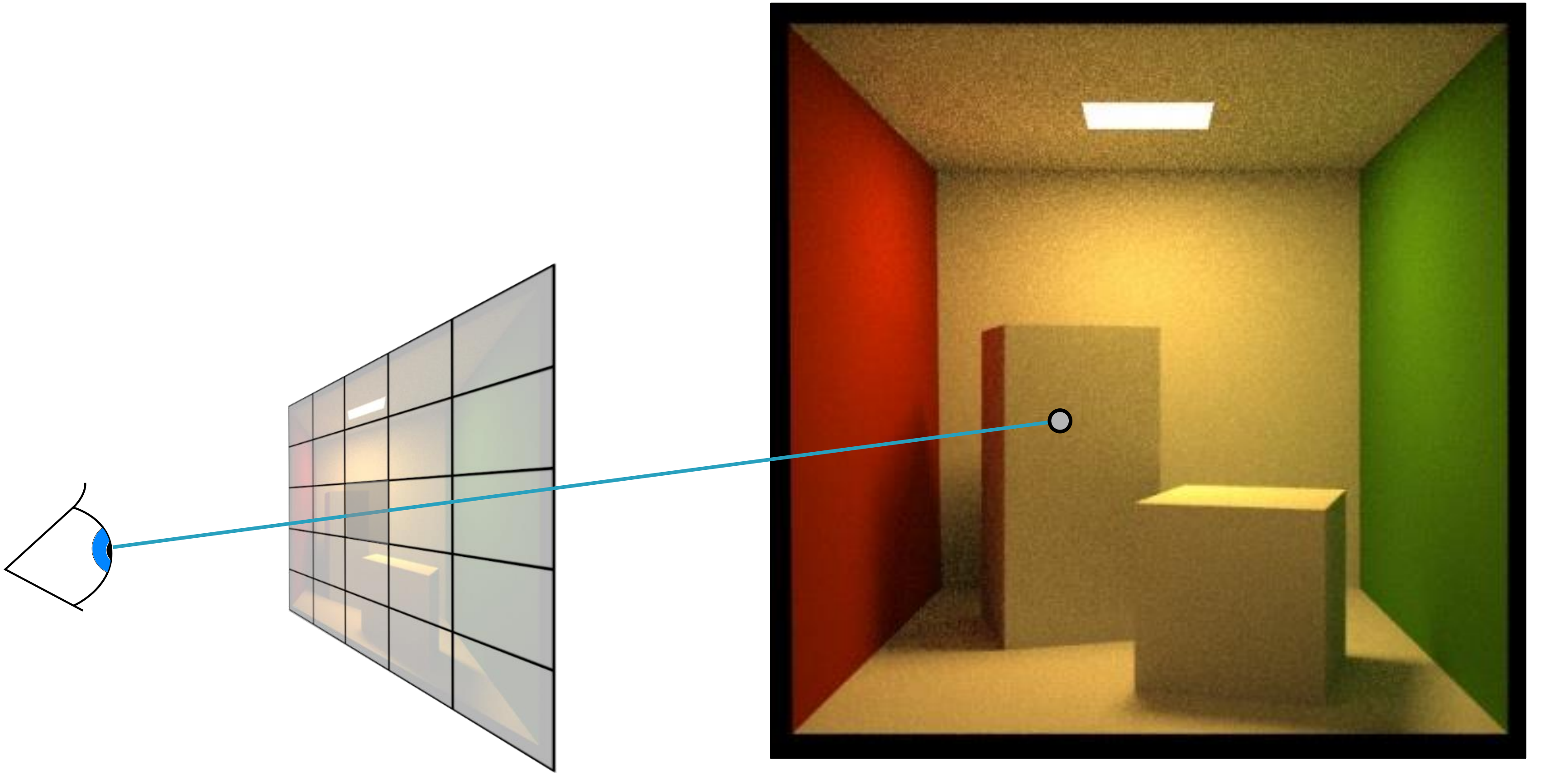
```
estimateLin(x, omegaInAtX)  // radiance incident at "x" from the direction "omegaInAtX"
{                           //  ("omegaInAtX" is pointing *away* from "x")
  Spectrum throughput = (1,1,1)
  Spectrum accum  = (0,0,0)
  while(1)                  // we don't cut off the path length now
  {
    hit = findNearestIntersection(x, omegaInAtX)

    if noIntersection(hit)           // ray leaves the scene - it "hits" the background
      return accum + throughput * bkgLight.getLe(x, - omegaInAtX)

    omegaOut := -omegaInAtX          // omegaOut at hit.pos
    if isOnLightSource(hit)          // ray happened to directly hit a light source
      accum += throughput * getLe(hit.pos, omegaOut)          // "pick up" emission
    // now estimate the reflected radiance
    [omegaIn, pdfIn] := generateRandomDir(hit)                 // omegaIn at hit.pos
    throughput *= 1/pdfIn * brdf(hit.pos, omegaIn, omegaOut) * dot(hit.n, omegaIn)

    survivalProb = min(1, throughput.maxComponent())
    if rand() < survivalProb          // Russian Roulette - survive (reflect)
      throughput /= survivalProb
      x := hit.pos                    // "recursion"
      omegaInAtX := omegaIn           // "recursion"
    else
      break;                          // terminate path
  }
  return accum;
}
```

Loop with Russian roulette

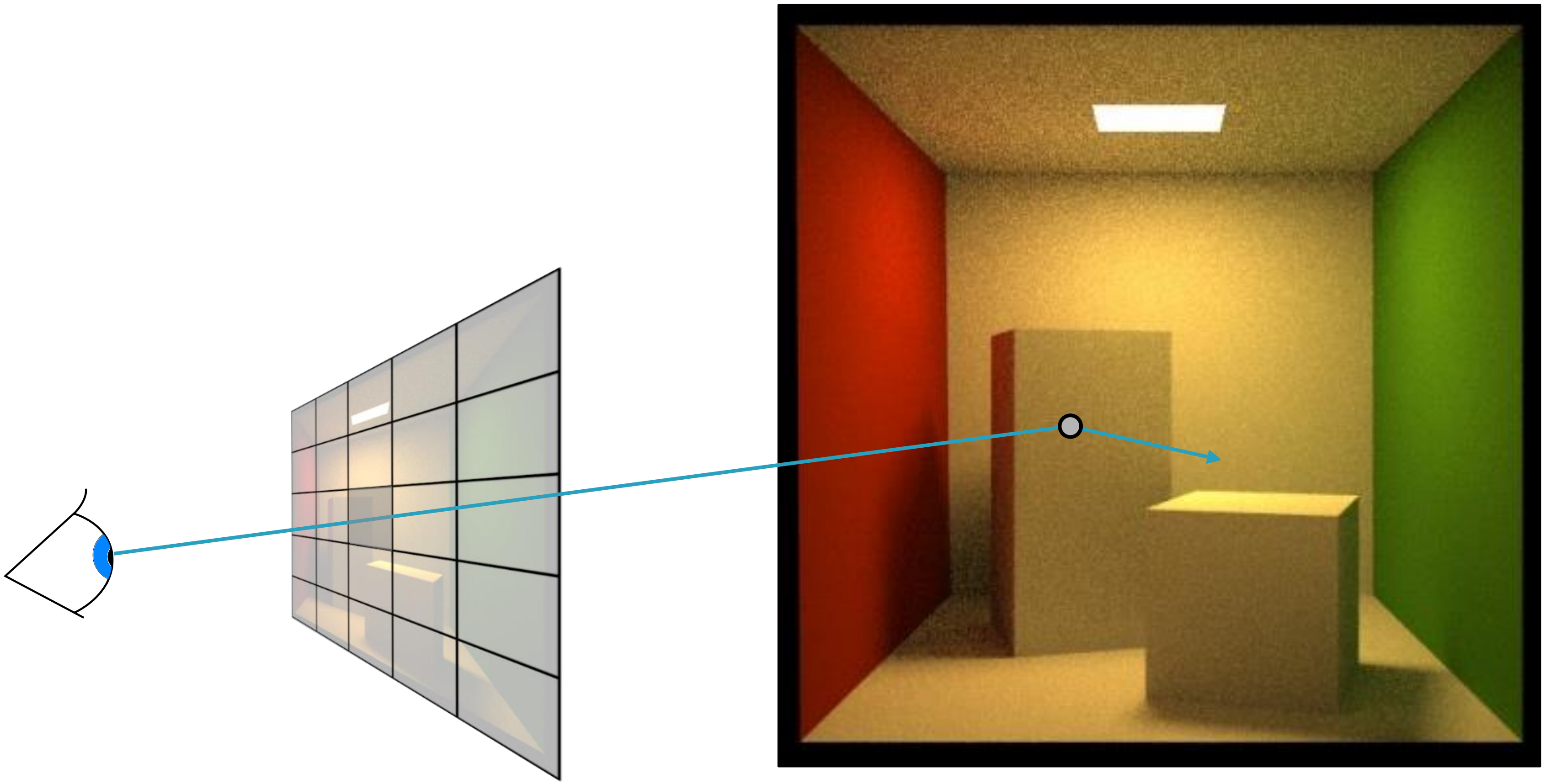




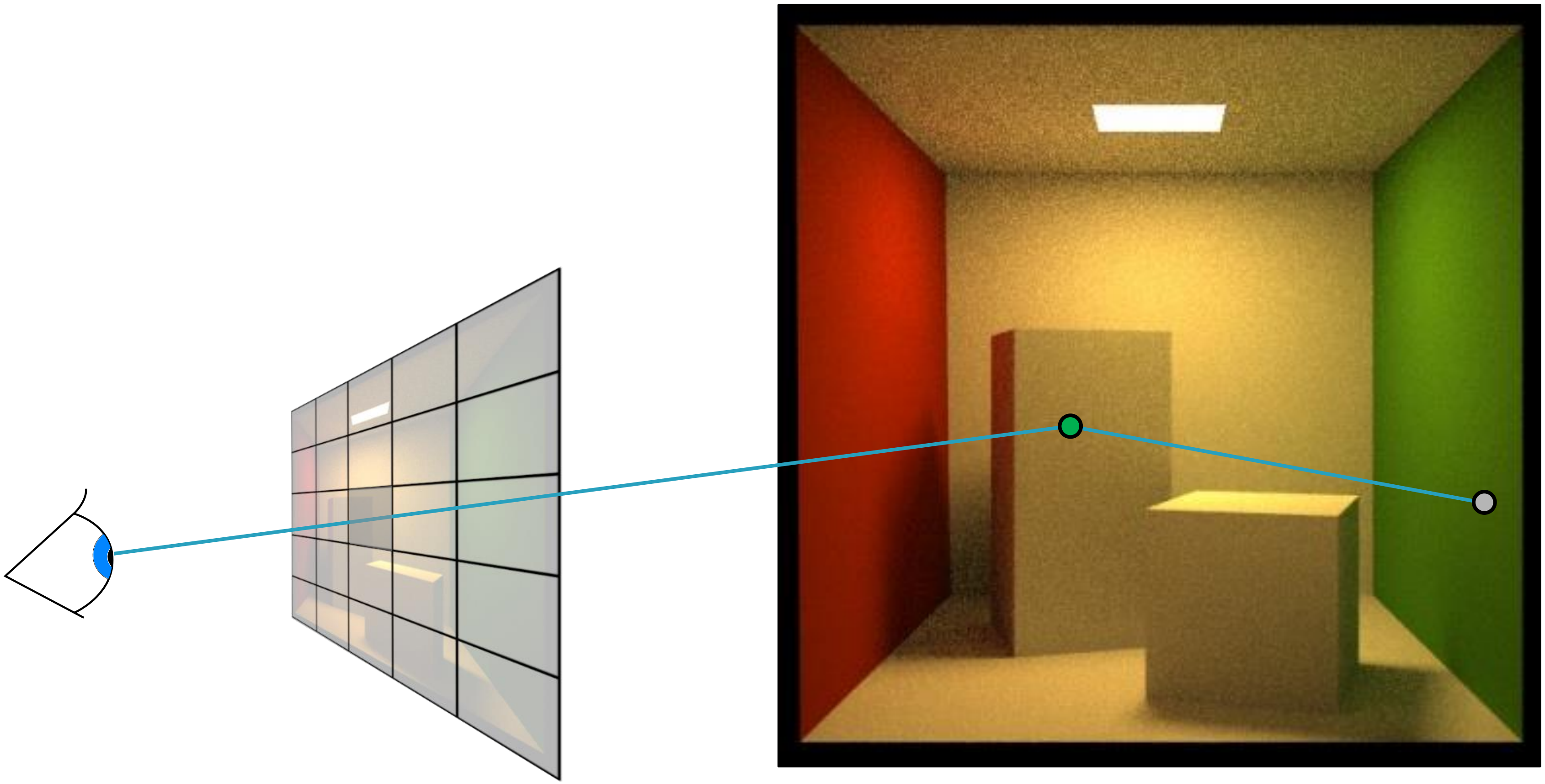
Loop with Russian roulette



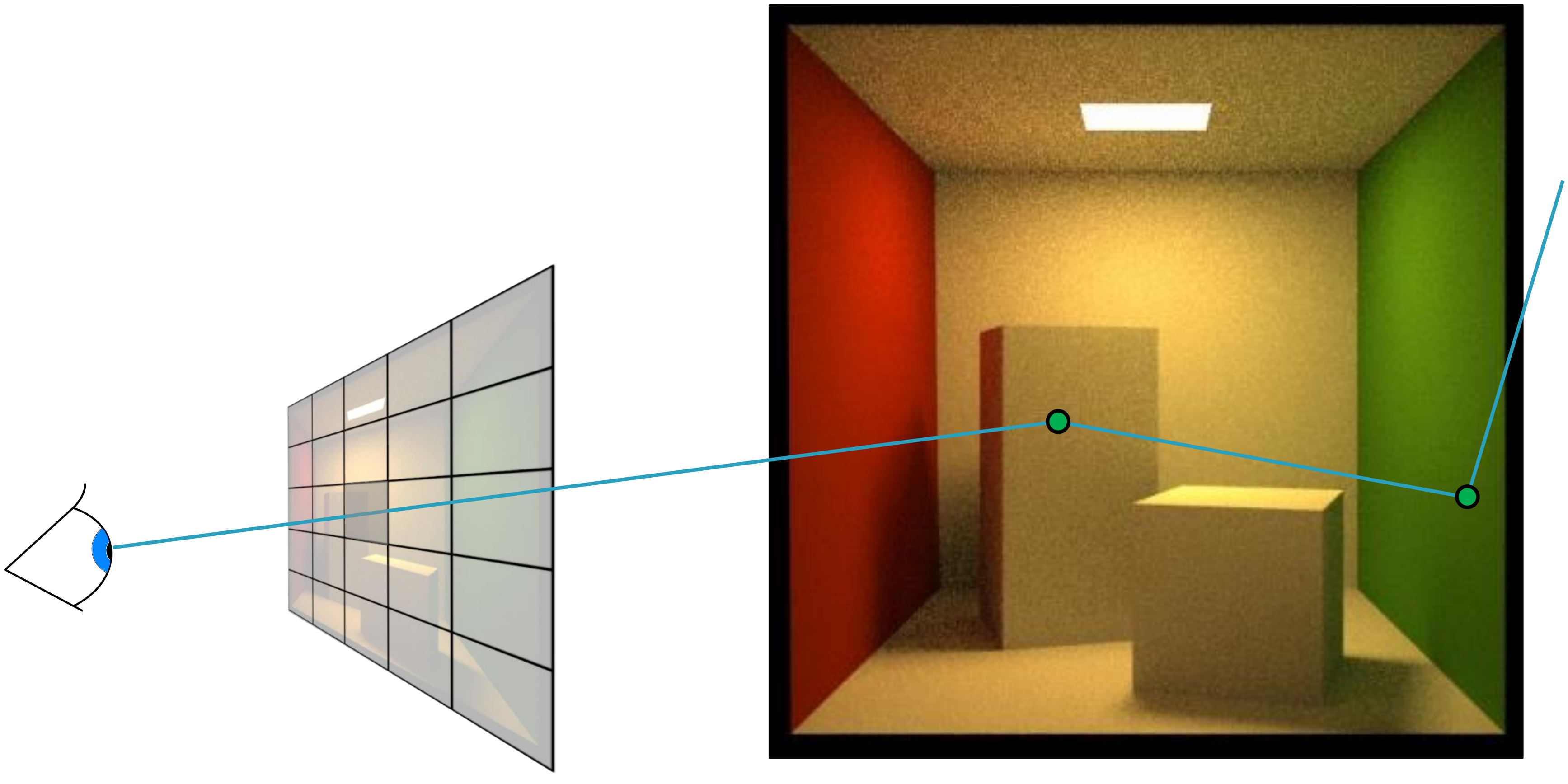
Loop with Russian roulette



Loop with Russian roulette

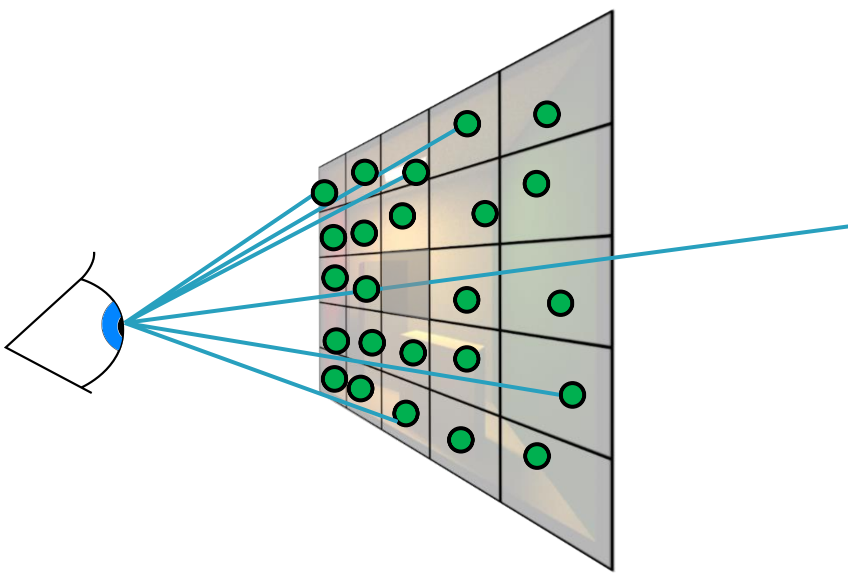


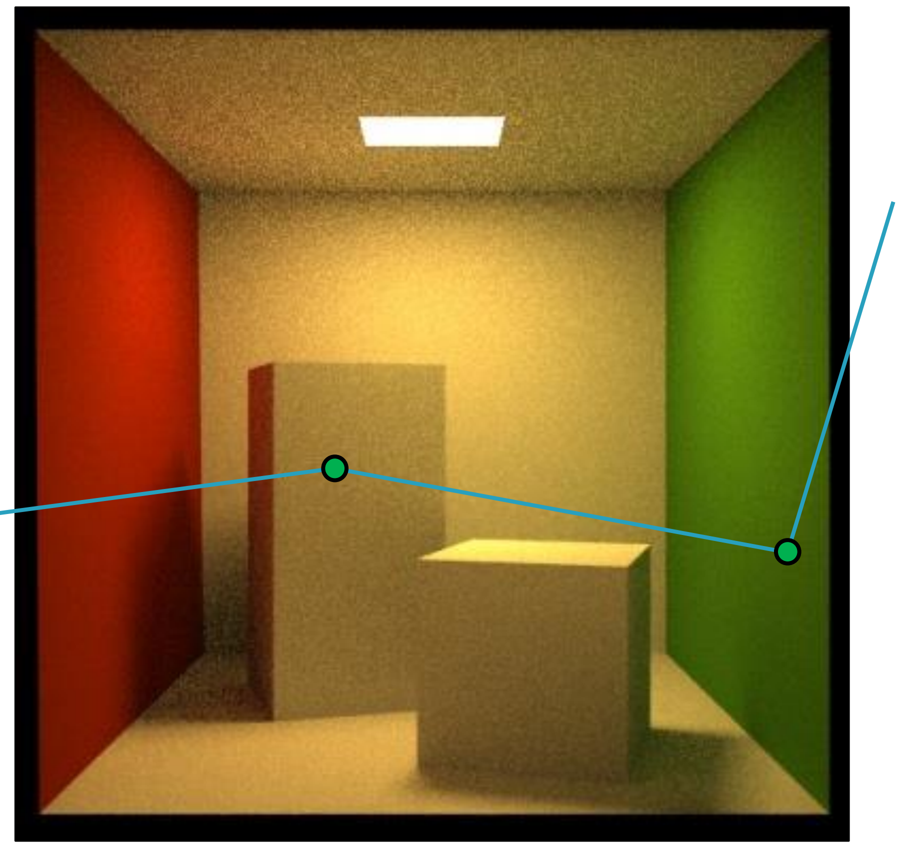
Loop with Russian roulette



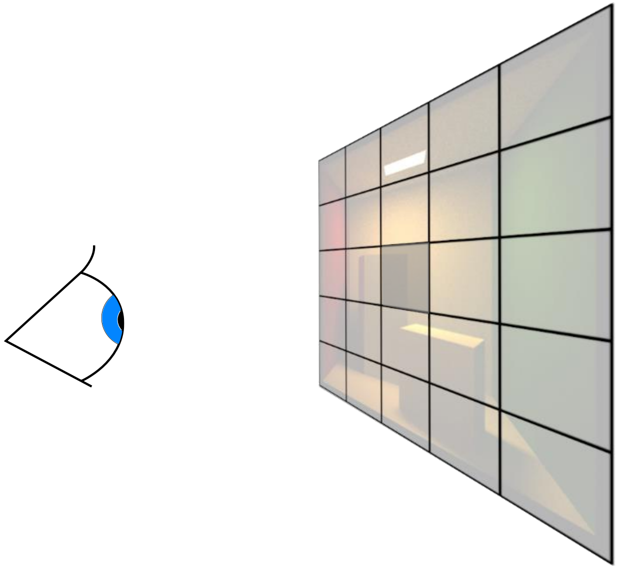
Finish progression

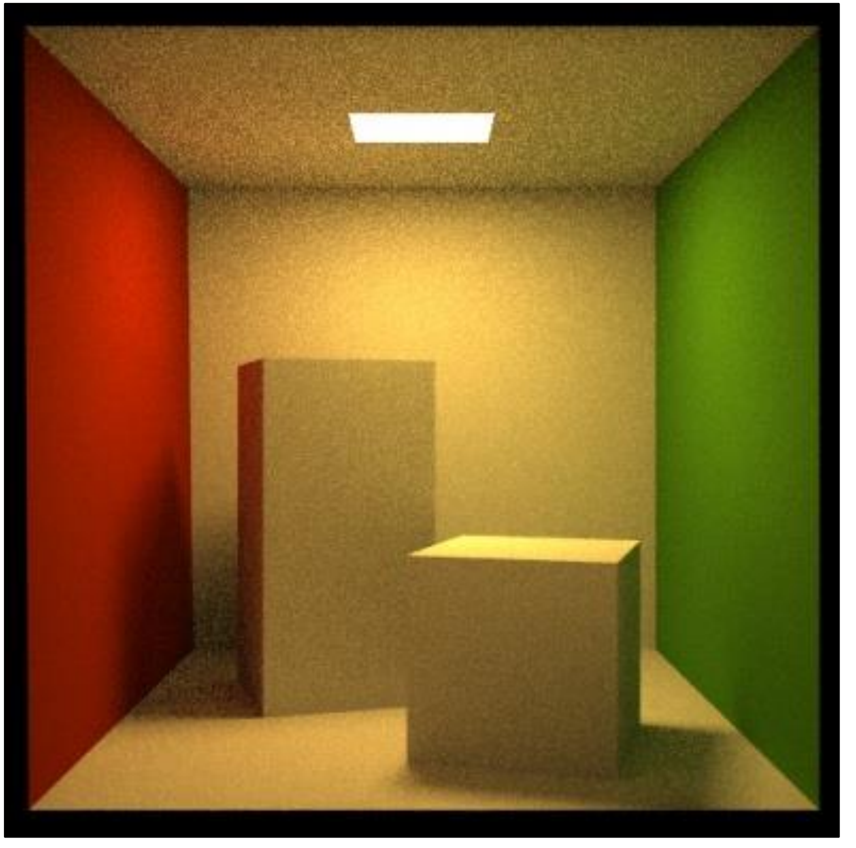
```
renderImage()
{
    for k = 1 to N_p  // rendering "passes"
    {
        for all pixels
        {
            estimateLin(x,omegaInAtX)
            …
```



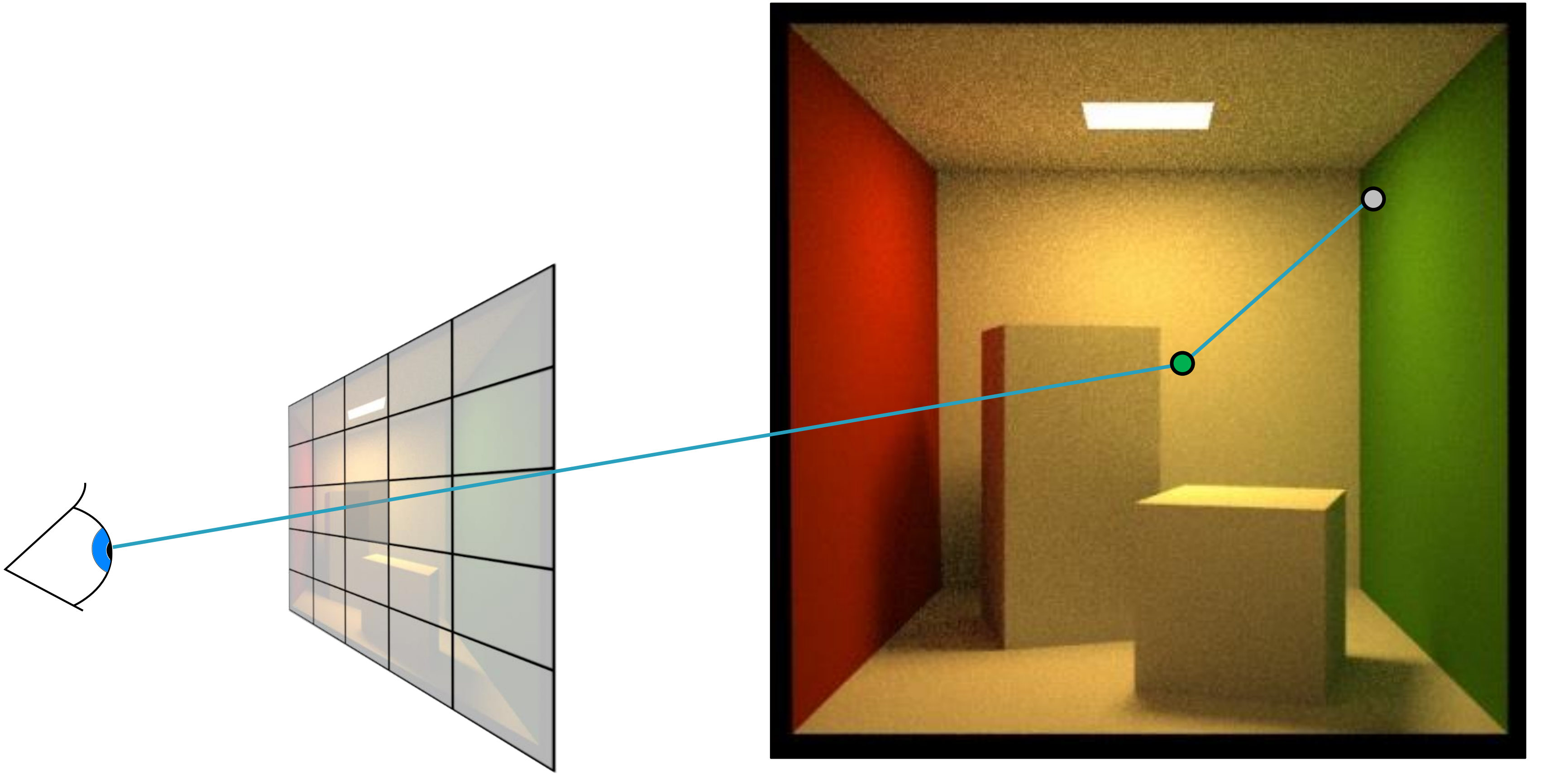


Loop with Russian roulette – path #2

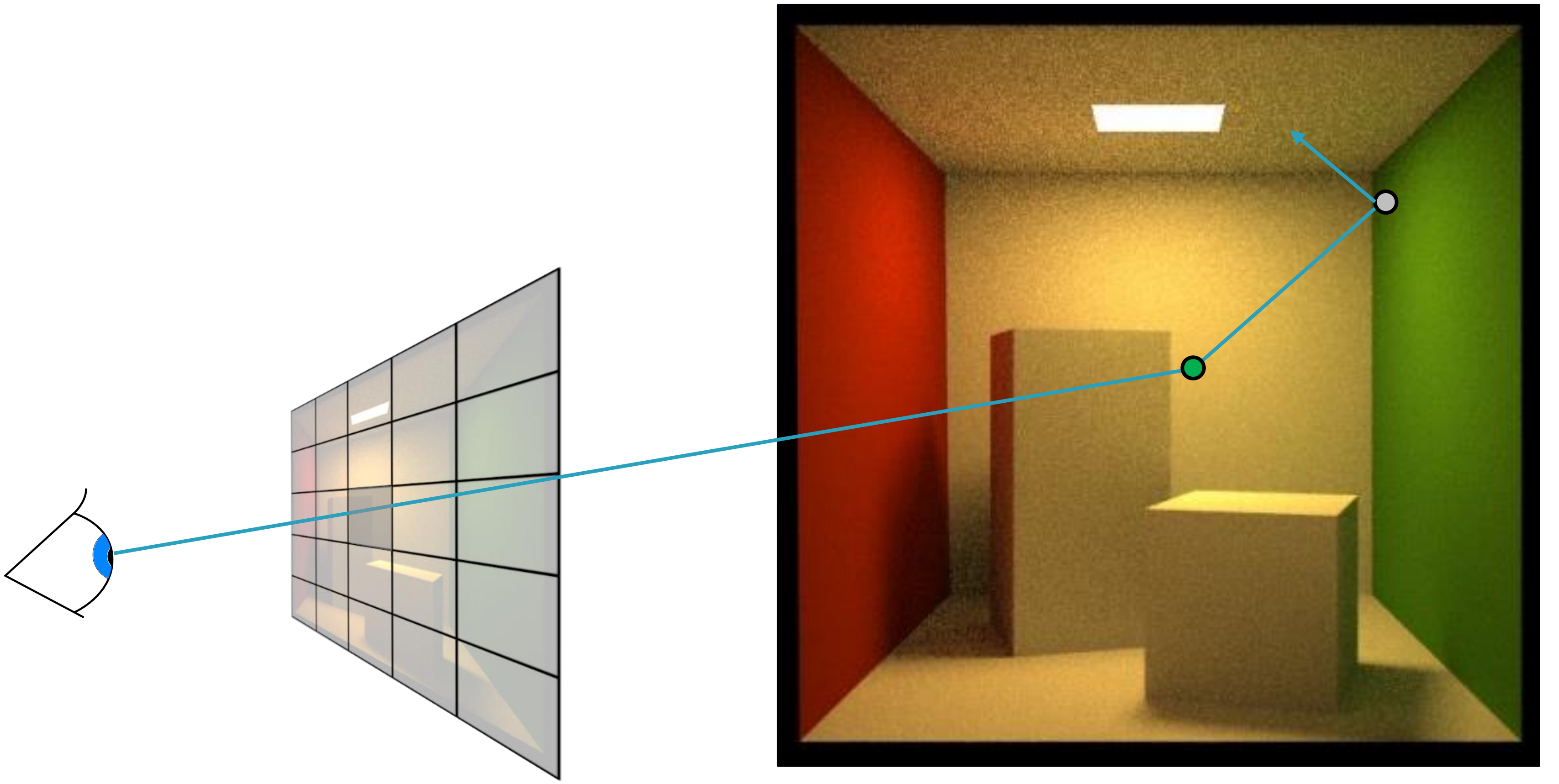




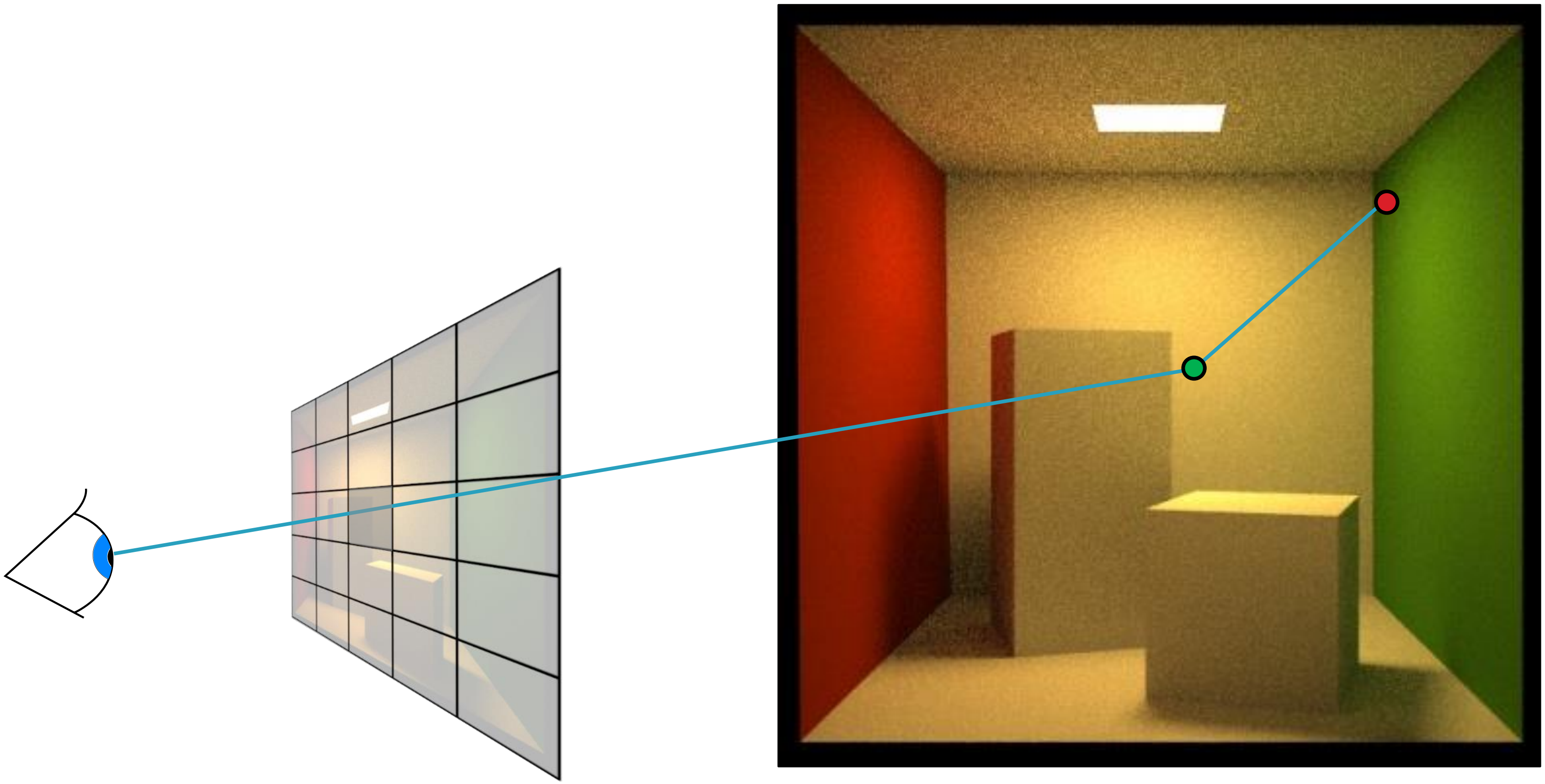
Loop with Russian roulette – path #2



Loop with Russian roulette – path #2



Loop with Russian roulette – path #2



Terminating paths – Russian roulette

- Continue the path with probability q
- Multiply weight (throughput) of surviving paths by 1 / q

$$Z = \begin{cases} Y / q & \text{if } \xi < q \\ 0 & \text{otherwise} \end{cases}$$

- RR is unbiased!

$$E[Z] = \frac{E[Y]}{q} \cdot q + 0 \cdot \frac{1}{q-1} = E[Y]$$

Survival probability

- It makes sense to use the surface reflectance ρ as the survival probability
  - If the surface reflects only 30% of light energy, we continue with the probability of 30%. That's how light behaves in physical reality.

Survival probability

- What if we cannot calculate ρ? Then there's a convenient alternative, which in fact works even better:
  1. First sample a random direction ω_{in} according to *pdf*(ω_{in})
  2. Update the path throughput
  3. Use the updated throughput as the survival probability

- If direction sampling *pdf*(ω_{in}) is exactly proportional to BRDF*cos, the above strategy turns out to be exactly equivalent to setting survival probability to the surface reflectance (*prove this*).

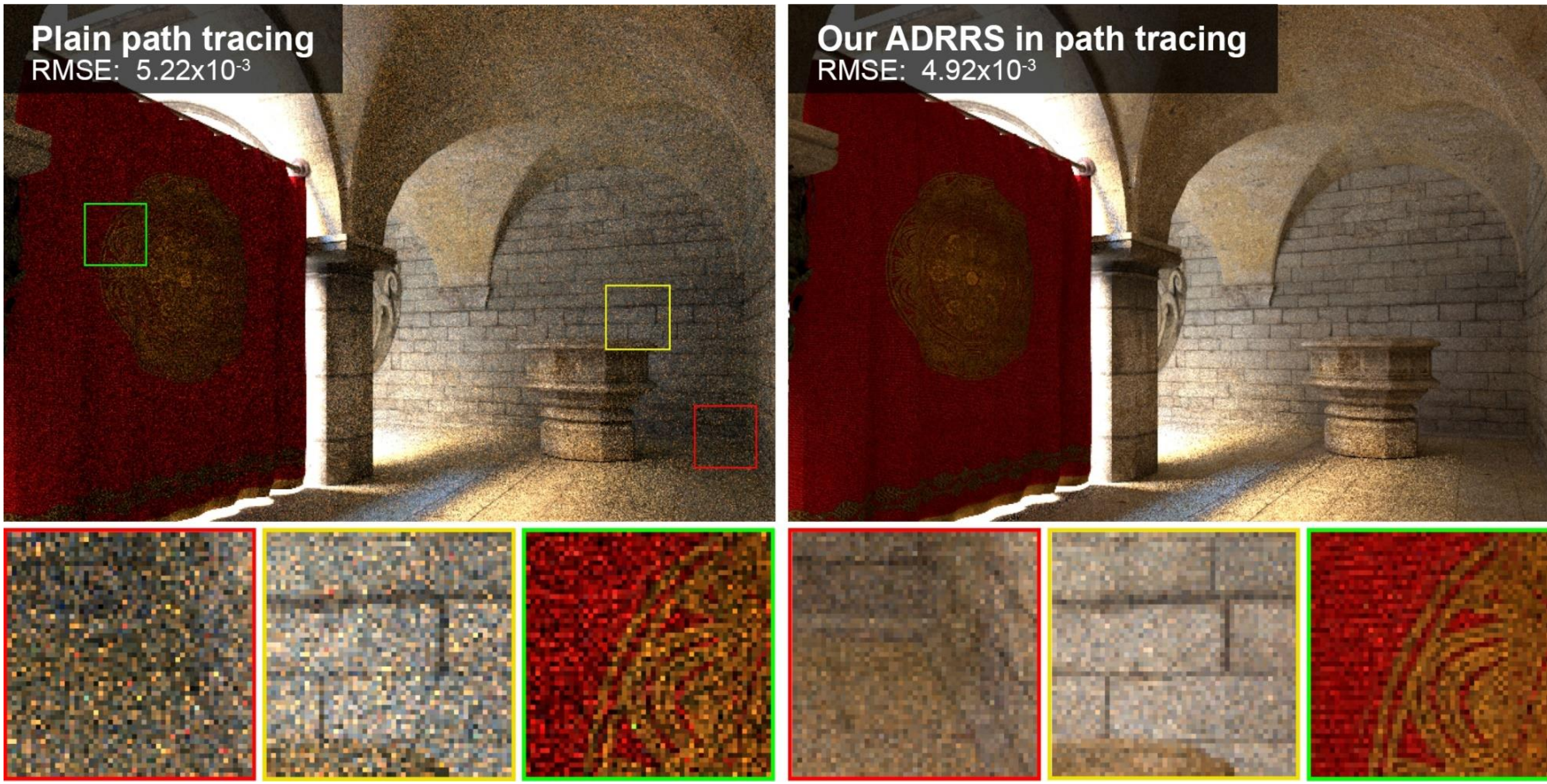
Survival probability

- Our work: Adjoint-driven Russian Roulette & Splititng [Vorba & Křivánek 2016]

  - Weight the survival probability by the expected path contribution
    - If we enter a bright region, continue path even if throughput might be low
    - If we enter a dark region, kill the path even if throughput might be high

  - If the "survival probability" ends up > 1, split the path

Adjoint-driven RR and splitting



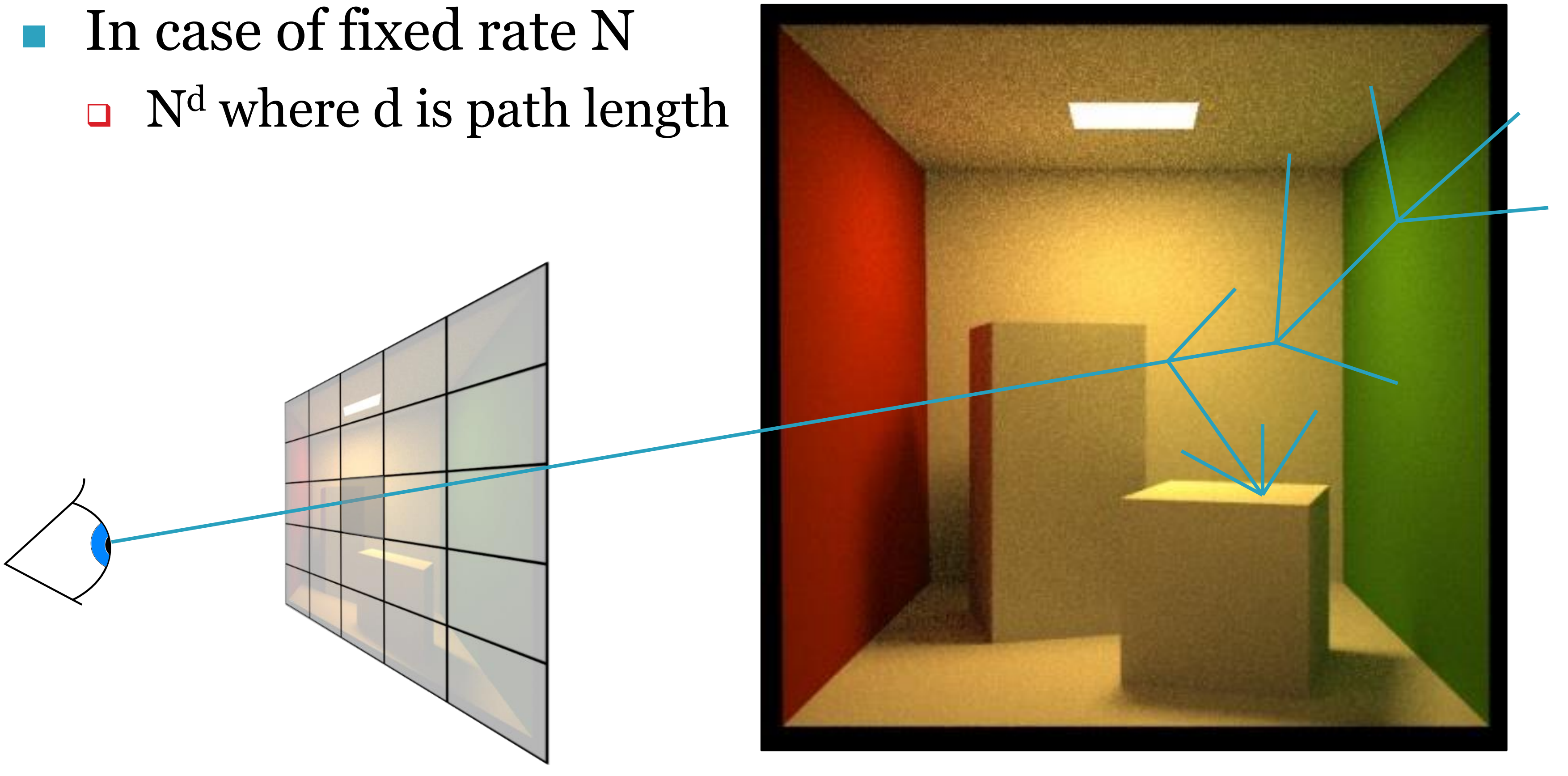
Vorba and Křivánek. **Adjoint-Driven Russian Roulette and Splitting in Light Transport Simulation**. *ACM SIGGRAPH 2016*

Path tracing, v. 0.1 – with splitting

```
estimateLin (x, ω):                          // radiance incident at x from direction ω
    y = findNearestIntersection(x, ω)
    if (no intersection)
        return backgroud.getLe (–ω) // emitted radiance from envmap
    else
        return    getLe (y, –ω) +            // emitted radiance (if y is on a light)
                  estimateLrefl (y, –ω)// reflected radiance

estimateLrefl(x, ω_out):
    accum := 0
    N := computeSplit(x);                    // number of “new“ split paths
    for i = 1 to N
        [ω_in , pdf] = genRandomDir(x, ω_out);   // e.g. BRDF imp. sampling
        accum += estimateLin(x, ω_in) * brdf(x, ω_in, ω_out) * dot(n_x, ω_in) / pdf;
    return accum / N;                        // average the split contributions
```

Beware of exponential branching!

- In case of fixed rate N
  - N^d where d is path length



```
estimateLin(x, omegaInAtX)  // radiance incident at "x" from the direction "omegaInAtX"
{                           //  ("omegaInAtX" is pointing *away* from "x")
  Spectrum throughput = (1,1,1)
  Spectrum accum  = (0,0,0)
  while(1)
  {
    hit = findNearestIntersection(x, omegaInAtX)

    if noIntersection(hit)           // ray leaves the scene - it "hits" the background
      return accum + throughput * bkgLight.getLe(x, - omegaInAtX)

    omegaOut := -omegaInAtX          // omegaOut at hit.pos
    if isOnLightSource(hit)          // ray happened to directly hit a light source
      accum += throughput * getLe(hit.pos, omegaOut)         // "pick up" emission
    // now estimate the reflected radiance
    [omegaIn, pdfIn] := generateRandomDir(hit)                // omegaIn at hit.pos
    throughput *= 1/pdfIn * brdf(hit.pos, omegaIn, omegaOut) * dot(hit.n, omegaIn)

    survivalProb = min(1, throughput.maxComponent())
    if rand() < survivalProb          // Russian Roulette - survive (reflect)
      throughput /= survivalProb
      x := hit.pos                    // "recursion"
      omegaInAtX := omegaIn           // "recursion"
    else                              // terminate the path - break the while loop
      break;
  }
  return accum;
}
```

Direction sampling – genRandomDir()

- We usually sample the direction ω_{in} from a pdf similar to

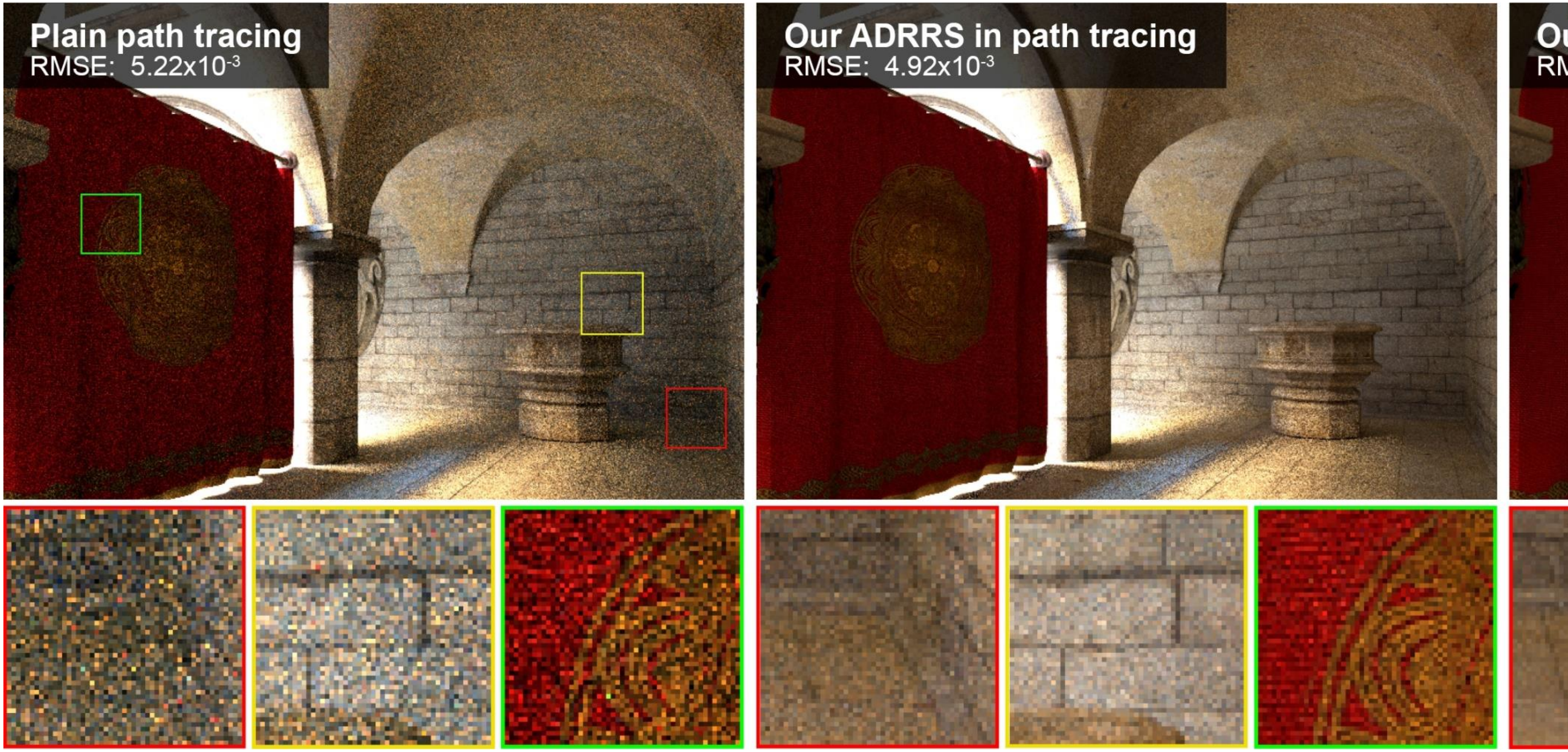
$$f_r(x, \omega_{\text{in}} \to \omega_{\text{out}}) \cos \theta_{\text{in}}$$

- Ideally, we would want to sample proportionally to the integrand itself

$$L_{\text{in}}(x, \omega_{\text{in}})\, f_r(x, \omega_{\text{in}} \to \omega_{\text{out}}) \cos \theta_{\text{i}},$$

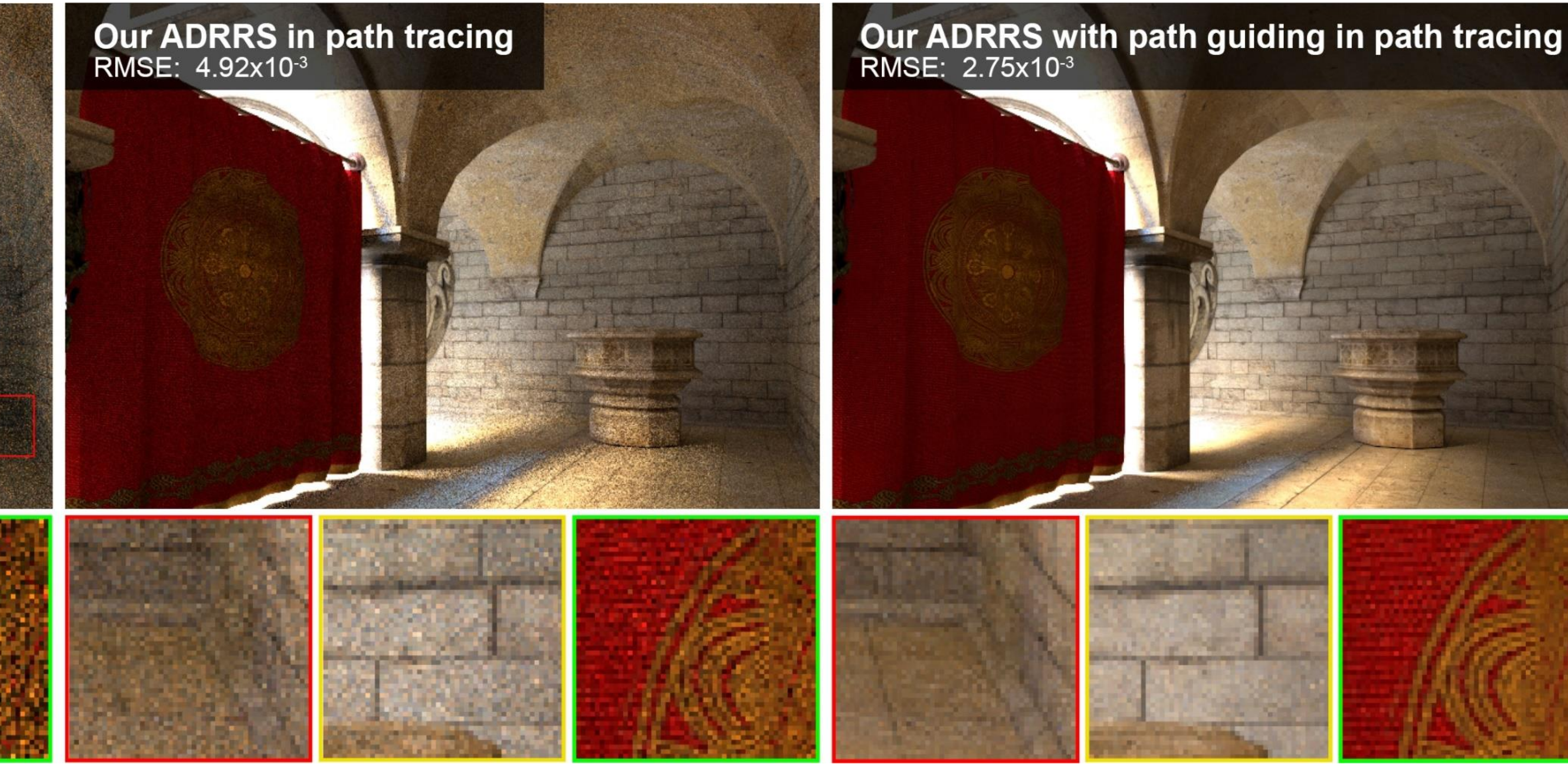
  but this is difficult, because we do not know L_{in} upfront. With some precomputation, it is possible to use a rough estimate of L_{in} for sampling [Jensen 95, Vorba et al. 2014]. This is called "path guiding".

Path guiding



Vorba, Karlík, Šik, Ritschel, and Křivánek. **On-line Learning of Parametric Mixture Models for Light Transport Simulation**. *ACM SIGGRAPH 2014*

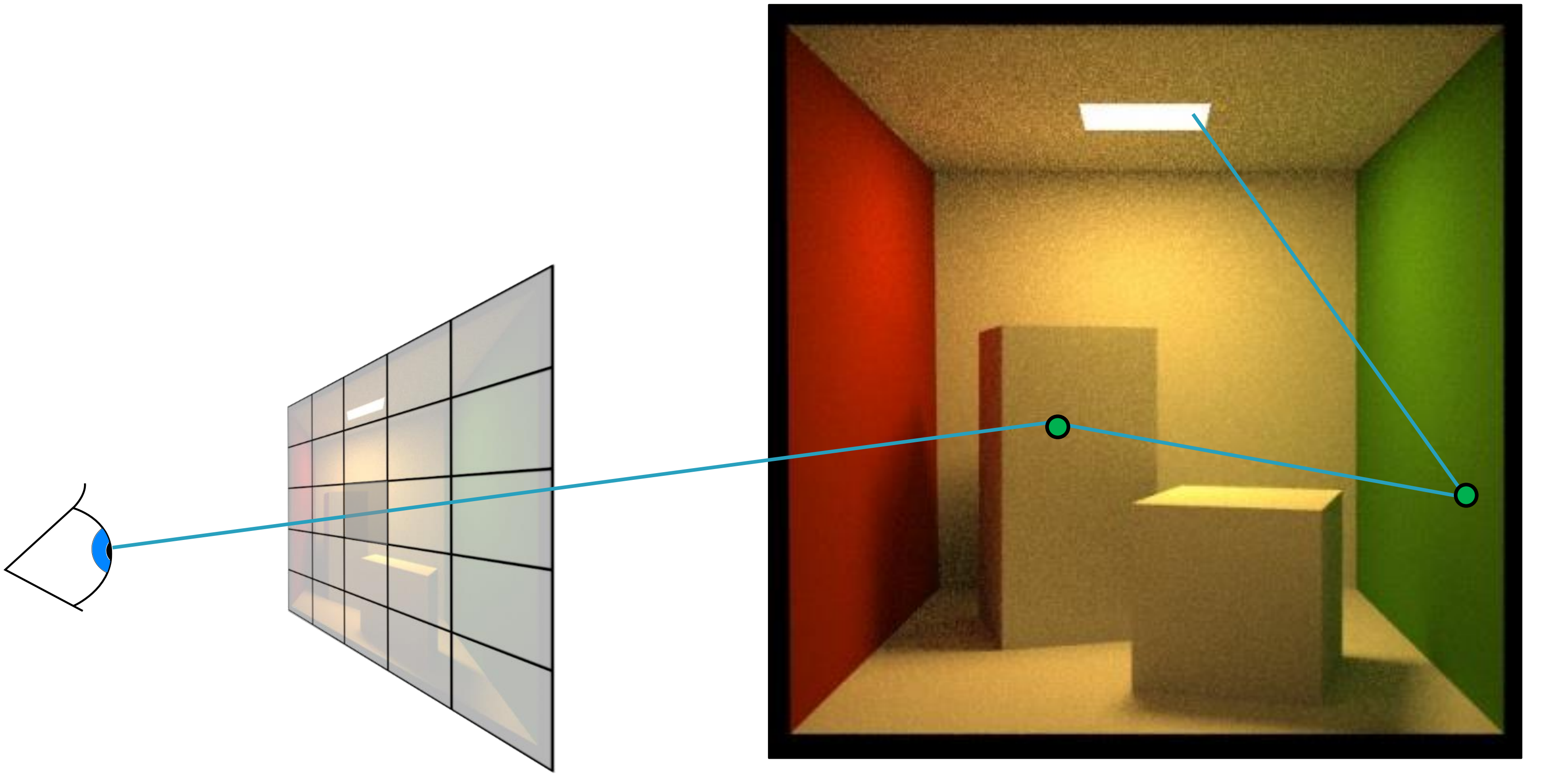
Path guiding



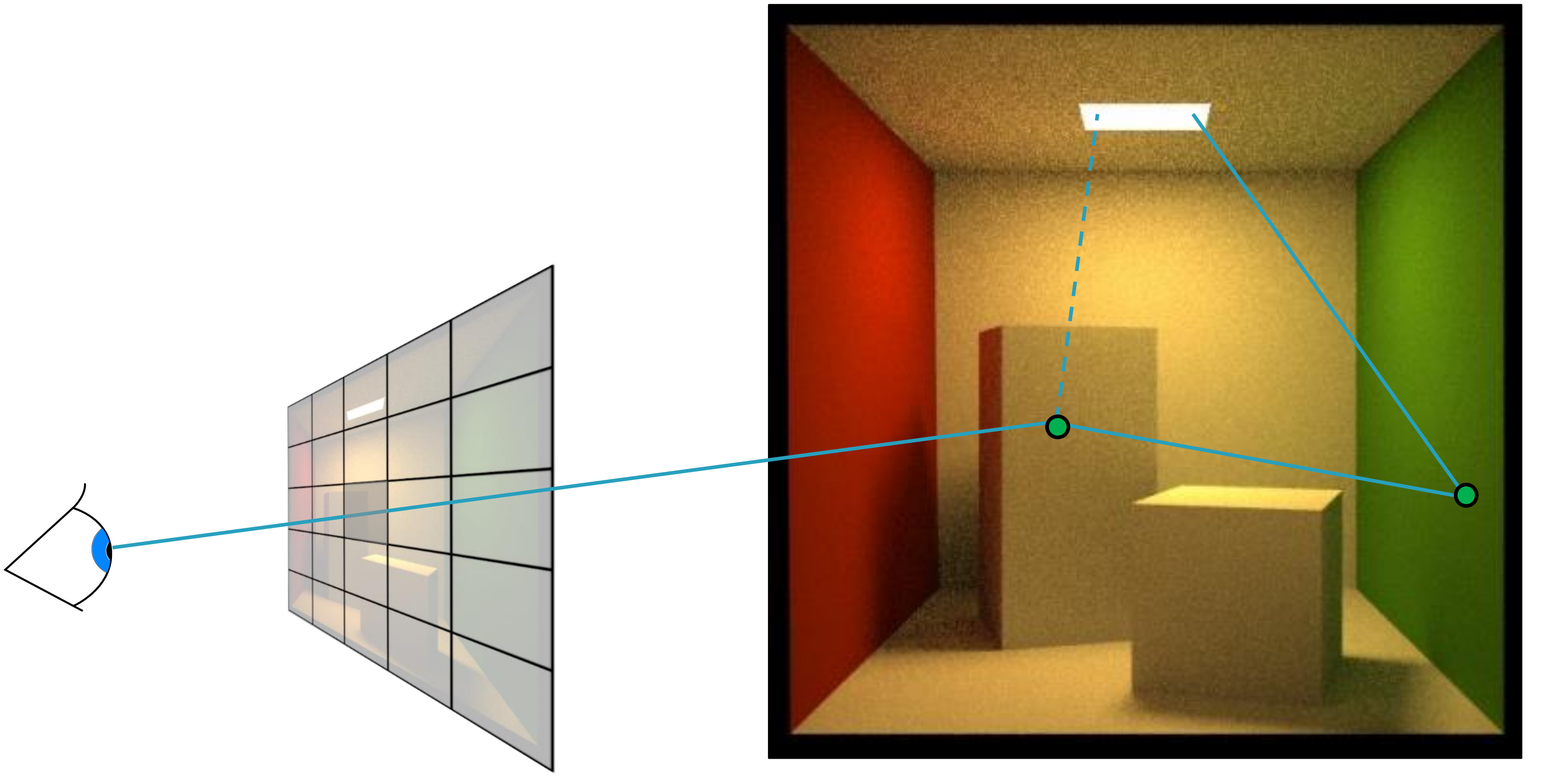
Vorba, Karlík, Šik, Ritschel, and Křivánek. **On-line Learning of Parametric Mixture Models for Light Transport Simulation**. *ACM SIGGRAPH 2014*

Direct illumination calculation in a path tracer

So far: accumulate $T^i L_e$



Now: at every vertex



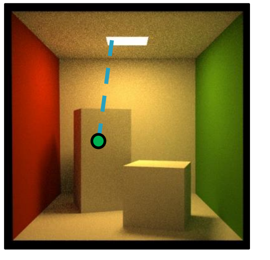
Direct illumination

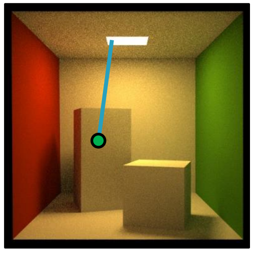
- At each path vertex, we are calculating **direct illumination**
  - i.e. radiance reflected from the surface point exclusively due to the light coming *directly* from the light sources

Direct illumination: Two strategies

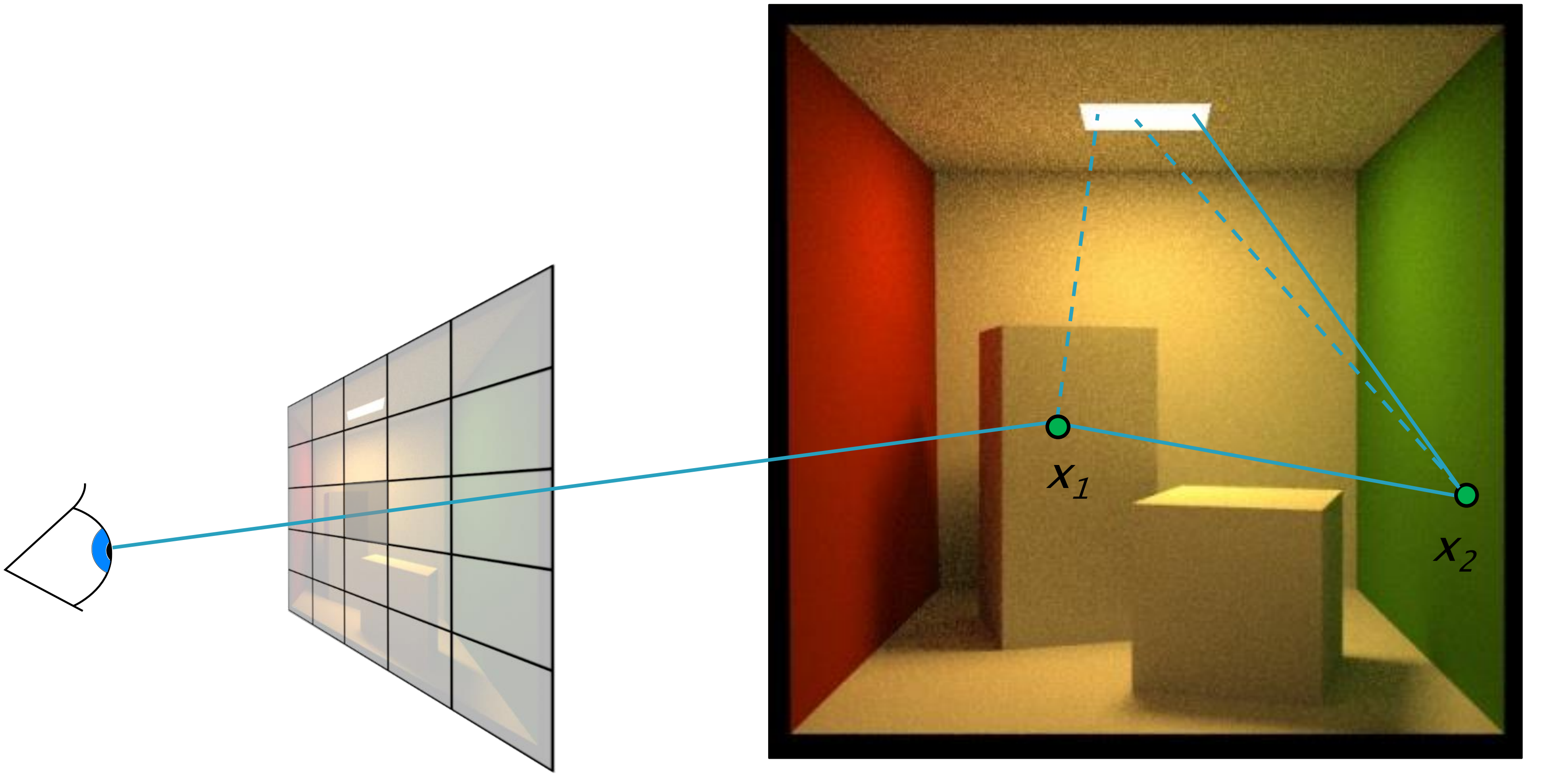
- At each path vertex, we are calculating **direct illumination**
  - i.e. radiance reflected from the surface point exclusively due to the light coming *directly* from the light sources

- Two sampling strategies
  1. **Explicit light source sampling (NEE)**
     ("next event estimation")
  2. **BRDF-proportional sampling**
     (already in the above code)





Two strategies at every vertex



The use of MIS in a path tracer

- At each path vertex do both:
  - **Explicit light source sampling**
    - Generate point on light source & cast shadow ray
  - **BRDF-proportional sampling**
    - One ray can be shared for the calculation of both **direct** and **indirect** illumination
    - But the MIS weight is applied only on the direct term (indirect illumination is added unweighted because there is no alternative technique to calculate it)

Direct illumination: formulas

- Look into the previous lecture on MIS.
  - MIS, two forms of reflection equation and its estimators

NEE - multiple light sources

- Option 1:
  1. Loop over all sources and send a shadow ray to each one
- Option 2:
  1. Choose one source at random
  2. Sample illumination only on the chosen light, divide the result by the prob of picking that light
  - (Scales better with many sources but has higher variance per path)
- Beware: The probability of choosing a light influences the sampling pds and therefore also the MIS weights.

Option 1 – Sum all contributions

```
Spectrum accum(0);

// Loop over all "Nl" lights
for i := 1 to Nl
{
    // Estimate reflected radiance at "x" into "omegaOut"
    // due to light i
    accum += estimateReflLoDirect(x, omegaOut, lights[i]);
}
```

Option 2 – Stochastic sampling

```
// Construct probability mass function (aka discrete pdf)
// over all lights with respect to shading position "x"
// and outgoing direction "omegaOut"
pmf := constructPmf(x,omegaOut,lights);

// Select light "i" with probability "pmf[i]"
i := pmf.sampleIndex();

// Estimate reflected radiance at "x" into "omegaOut"
// due to light i
Spectrum accum =
   estimateReflLoDirect(x, omegaOut, lights[i]) / pmf[i];
```

Option 2 – Choice of PMF

- Naive
  - Uniform
  - Proportional to power
- Example: Equal area, same orientation, same power
  - Q: *Why are the above not optimal?*

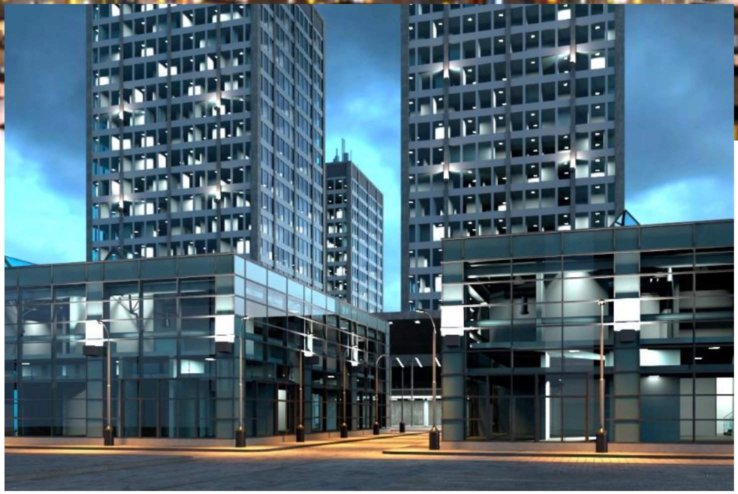
Option 2 – Choice of PMF

- Naive
  - Uniform
  - Proportional to power
- Ideal
  - Proportional to light contribution with respect to x (and ω_o)

Efficient many-light methods

- Problem: thousands of lights

Source: Estevez et Kulla 2018



Source: Vevoda et al. 2018

Efficient many-light methods

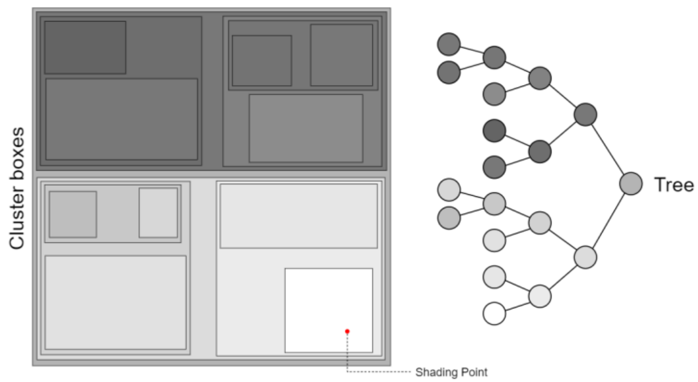
- Problem: thousands of lights
- Ideal pmf depends on
  - Position
  - Orientation
  - Distance
  - Power
  - Visibility

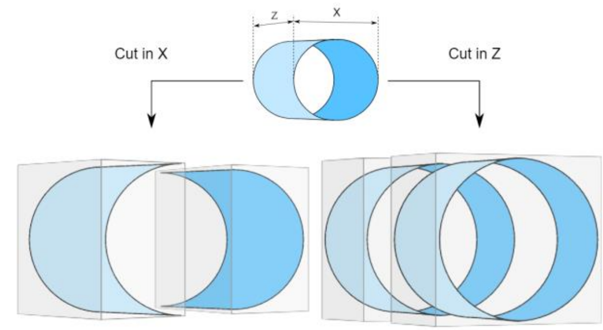
Efficient many-light methods

- Typical approach
  1. Build a light hierarchy (preprocess)
  2. Construct a tree cut given position x (rendering)

Build a light hierarchy

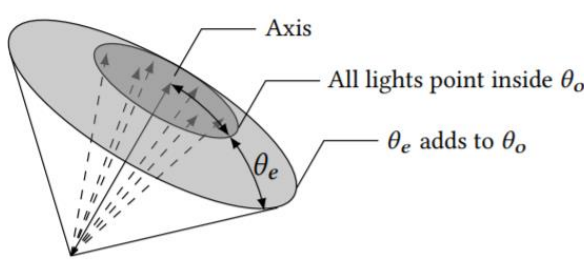
- Cluster based on position, orientation

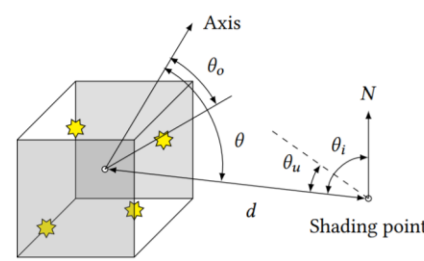




Left split reduces the orientation bounds

- Compute bounds for each node (spatial, directional)

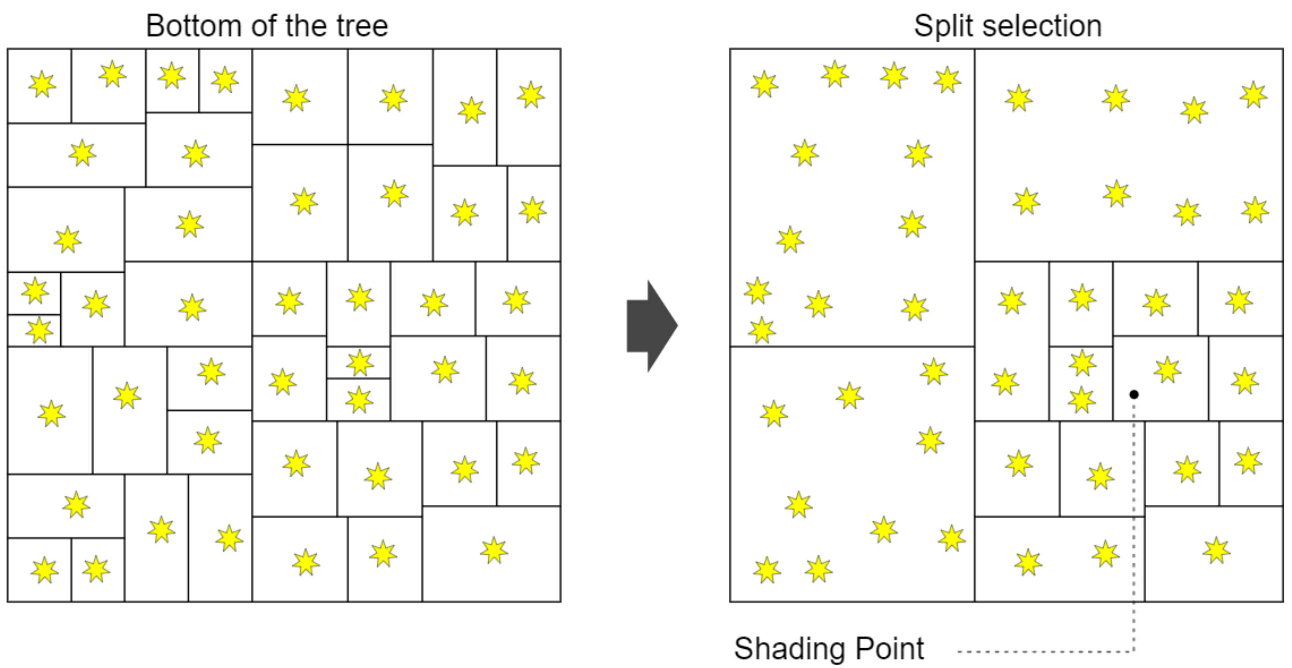




Source: Estevez et Kulla 2018

Construct a tree cut given position x

- Usually based on un-occluded contributions
- Gives us our pmf



Source: Estevez et Kulla 2018

Learning the lights' contributions

Védová, Kondapaneni, Křivánek. **Bayesian online regression for adaptive direct illumination sampling**. *ACM SIGGRAPH 2018*

Learning the lights' contributions

Vévoda, Kondapaneni, Křivánek. **Bayesian online regression for adaptive direct illumination sampling**. *ACM SIGGRAPH 2018*

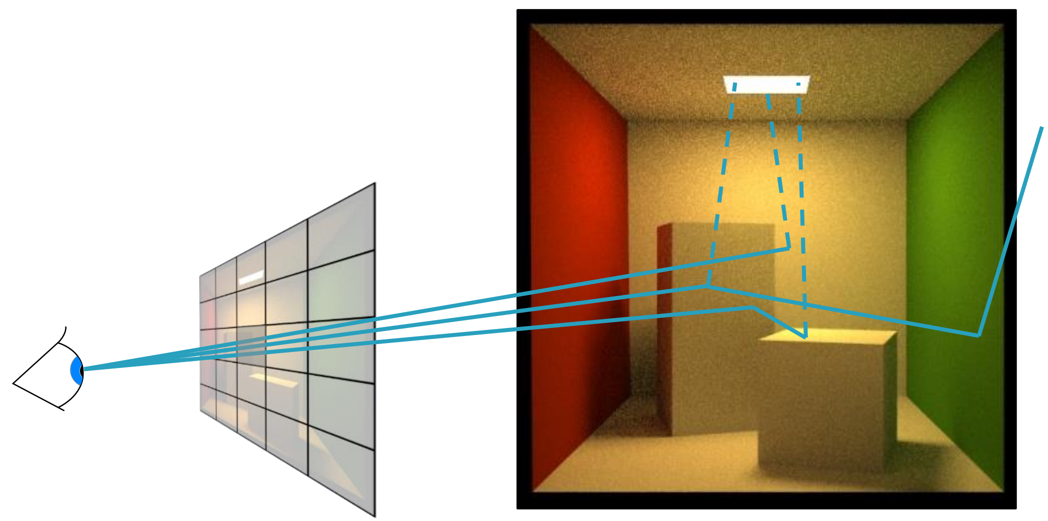
Misc

Typical numbers of rays cast in the scene

- Consider 2K image (2M pixels)
- Per progression
  - 2M primary rays
  - (2M shadow rays + 2M) * "average path length"
  - NEE shadow rays (cheaper - early exit, unless we consider transparent surfaces)
- Progressions (aka samples or paths per pixel)
  - Typically a few hundreds (with good importance sampling)
- Highly scene dependent!!
- Depends also on the algorithm (Russian roulette, splitting…)

Summary

- **Pathtracer with next-event estimation**
- Core of the most production renderers



What we have not covered

- Adaptive image plane sampling
  - Equalizes error over pixels
  - Essential for movie production (and offline rendering)
- Denoising
  - Essential for both movies and ray-traced games