Path Tracing

Priv. Doz. Dr. Ing. Martin Lambers

Repetition

Accelerating our Path Tracer

We traverse a Bounding Volume Hierarchy and call `AABB::hit()` and `Surface::hit()`

- For every pixel
  Nothing we can do other than reducing resolution
- For every sample per pixel
  $\rightarrow$ Monte Carlo techniques: Importance Sampling, Stratified Sampling, Quasi Monte Carlo Sampling
- For every path segment
  $\rightarrow$ Monte Carlo technique: Russian Roulette

Today we focus on Monte Carlo techniques.
The good news: We already do Monte-Carlo Path Tracing.
We just need some background to apply optimizations.

Repetition

Basic Stochastics

- **Expected value** E: What is the expected mean result when repeating the experiment often?
  - $E(x) = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} x$
    $= \sum_{x} x \cdot p(x)$
  - Perfect dice: $E(x) = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = \frac{21}{6} = 3.5$
- **Variance** V: How much do results divert from the expected value?
  - $V(x) = E\left((x - E(x))^2\right)$
    $= E(x^2) - E^2(x)$
  - Standard deviation σ:
    $V(x) = \sigma^2$

Repetition

Basic Stochastics

- Continuous random variables x, e.g. $x \in [0, 1) \subseteq \mathbb{R}$
  - The *absolute* probability $p(x)$ for one outcome x is zero since there are infinitely many outcomes!
  - The **probability density function** $p_f(x)$ gives a *relative* probability for outcome x:

    $$p(x \in [a, b]) = \int_a^b x \cdot p_f(x) dx$$

    "The probability for x falling in the range $[a, b]$ is given by the integral of $p_f(x)$ over $[a, b]$"
  - Expected value:

    $$E(x) = \int_{-\infty}^{\infty} x \cdot p_f(x) dx$$

  - Estimate expected value from N samples:

    $$E(x) \approx \frac{1}{N} \sum_{i=1}^{N} x_i$$

Repetition

Monte-Carlo Estimation of an Integral

- $I = \int_a^b f(x)dx$
- Compute samples $f(x_i)$ at discrete sample points x_i
- An **estimator** J uses these samples to estimate I
- For an **unbiased** estimator: $E(J) = I$
  This is what we want in physically based rendering!
  (However that does not mean that biased estimators are useless.)
- Desirable property of J:
  $V(J)$ min.

Repetition

Path Termination via Russian Roulette

- Our paths can contain many segments.
  In theory: unlimited. In practice: currently capped at 128.
- Rough estimate: how many segments can a path have before its contribution is too low to notice?
  - Assume a Lambertian material with $k_d = \frac{1}{2}$
  - Assume an average polar angle of $\theta = 60°$ so that $\cos\theta_i = \frac{1}{2}$
  - Attenuation factor at each surface: $\frac{k_d}{\pi}\cos\theta_i = \frac{1}{2\pi}\frac{1}{2} \approx 0.08$
  - Path throughput for each new segment:
    1, 0.08, 0.064, 0.000512, 0.00004096, 0.0000032768, ...
- In scenes where paths can bounce often between surfaces, many long paths will not contribute any noticable radiance.

Repetition

Path Termination via Russian Roulette

- Bad idea: cancel a path when the throughput falls below a threshold
  - Reason 1: even paths with low throughput might contribute noticable radiance if the path hits a bright light – the threshold is scene dependent!
  - Reason 2: cancelling paths changes the expected value – our estimator is not unbiased anymore!
- Idea: at each intersection, decide randomly whether to terminate a ray
  - Let J be our unbiased Monte-Carlo estimator for the rendering equation
  - Define the new estimator $K = \begin{cases} 0 & \text{if } p_f(K) \leq \alpha \\ \frac{J}{1-\alpha} & \text{if } p_f(K) > \alpha \end{cases}$
  - $E(K) = \int K p_f(K) dK = 0 \cdot \alpha + \frac{J}{1-\alpha}(1-\alpha) = J$
  - The expected value does not change!

Repetition

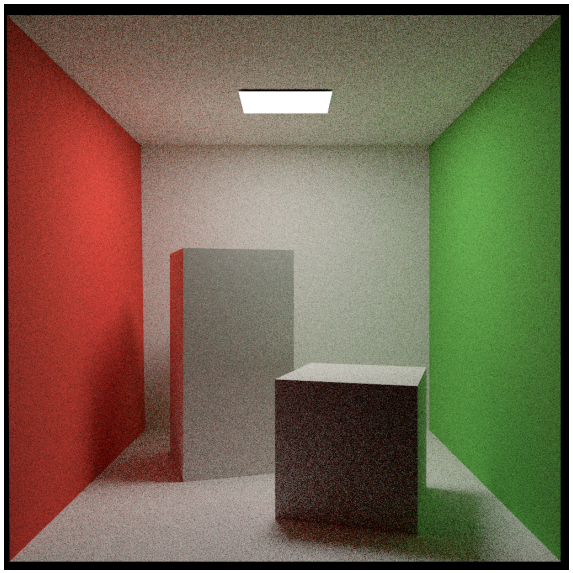
Path Termination via Russian Roulette

- The new estimator is unbiased
- Variance is *not* reduced – in fact, it will increase!
- But efficiency is increased by skipping unnecessary computations
- Implementation:
  - At each intersection, determine probability $q \in [0, 1]$ to cancel the path
  - Generate uniformly distributed pseudo random number $u \in [0, 1)$
  - If $u < q$: cancel the path
  - Otherwise: weight the path with factor $\frac{1}{1-q}$
- The usefulness of Russian Roulette depends on the choice of q!
- Common heuristic: $q \approx 1 - \text{throughput}$
  Lower throughput will lead to higher probability to cancel the path

Repetition

Path Termination via Russian Roulette

- Example result for Cornell Box scene, 1024×1024 with 1024 spp:



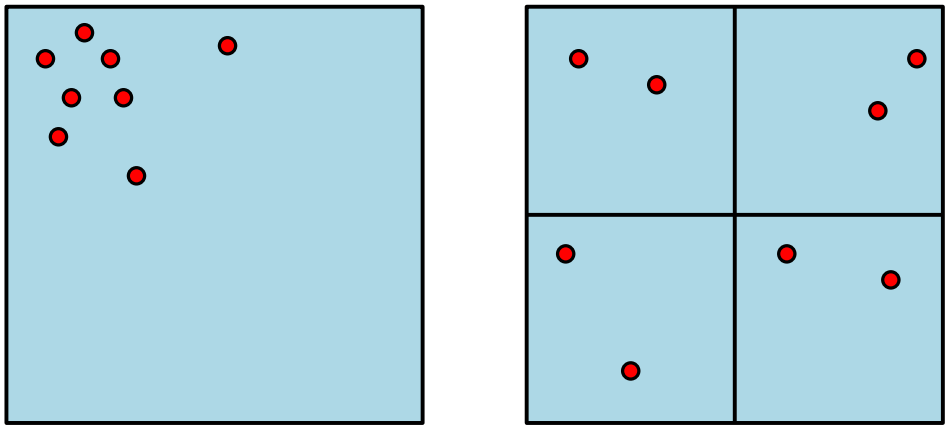
CPU time without Russian Roulette: 2816s
CPU time with Russian Roulette: 1622s
Reduction by more than 40%
No noticable difference in the rendering result

- Other scenes with fewer opportunities for a path to leave the scene or hit a light source will deliver even better results!

Repetition

Stratified Sampling

- Uniformly distributed random numbers can form clusters and leave wide areas of the domain uncovered
  (think of bad luck in Yahtzee/Kniffel)
- Solution: combine uniform and stochastic sampling
  - Subdivide your domain into equally-sized patches
  - Sample each patch with uniformly distributed random numbers

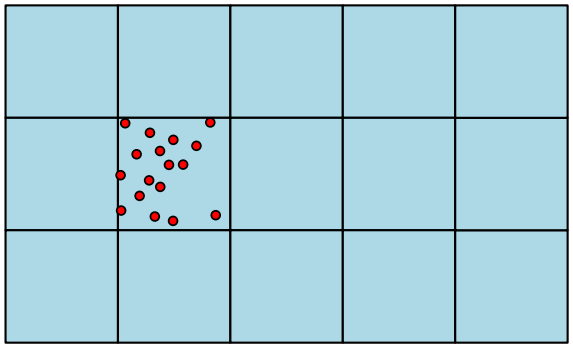


- Variance with stratified sampling is never larger than variance without stratified sampling!

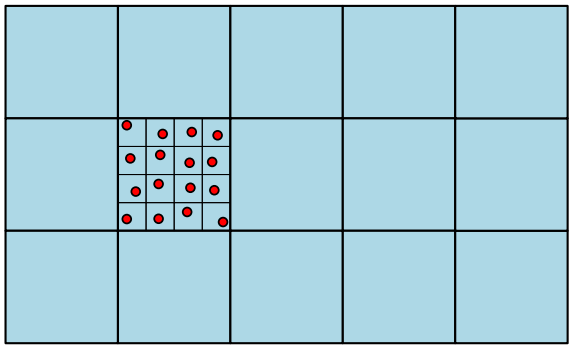
Repetition

Stratified Sampling

- Straightforward application: Stratified Sampling of image space
- We already subdivide into the pixel raster!



- We can take that one step further by subdividing the pixels.



- Better variance (with varying effects), basically for free

Repetition

Quasi-Random Sampling

- *Low-discrepancy sequences*:
  - Sequences of numbers
  - Properties similar to random numbers
  - Low discrepancy $\rightarrow$ even coverage of the domain
- Not (pseudo-)random numbers, but quasi-random
- In Monte Carlo rendering, such sequences can be used instead of pseudo-random numbers ($\rightarrow$ Quasi Monte Carlo rendering)
- To get the benefits of evenly distributed sample points, the sequence dimensions must be used consistently!
- Many implementations of low-discrepancy sequences have a maximum number of dimensions. Fall back to pseudo-random numbers for higher dimensions (i.e. very long paths).

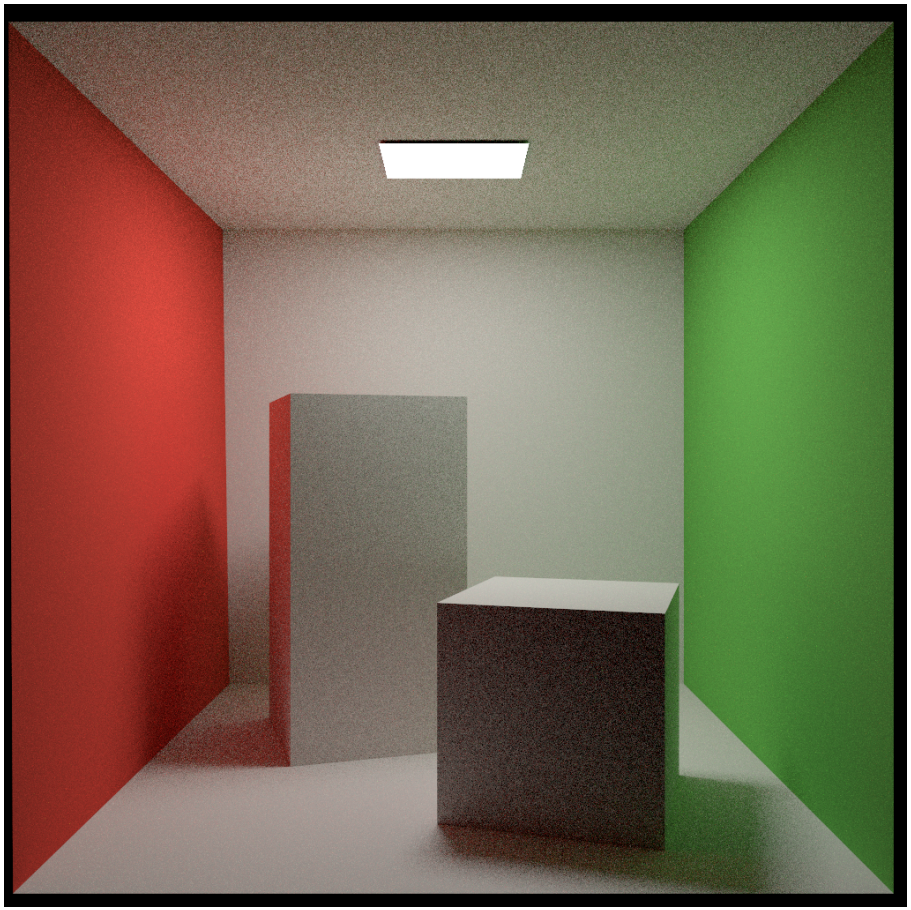
Repetition

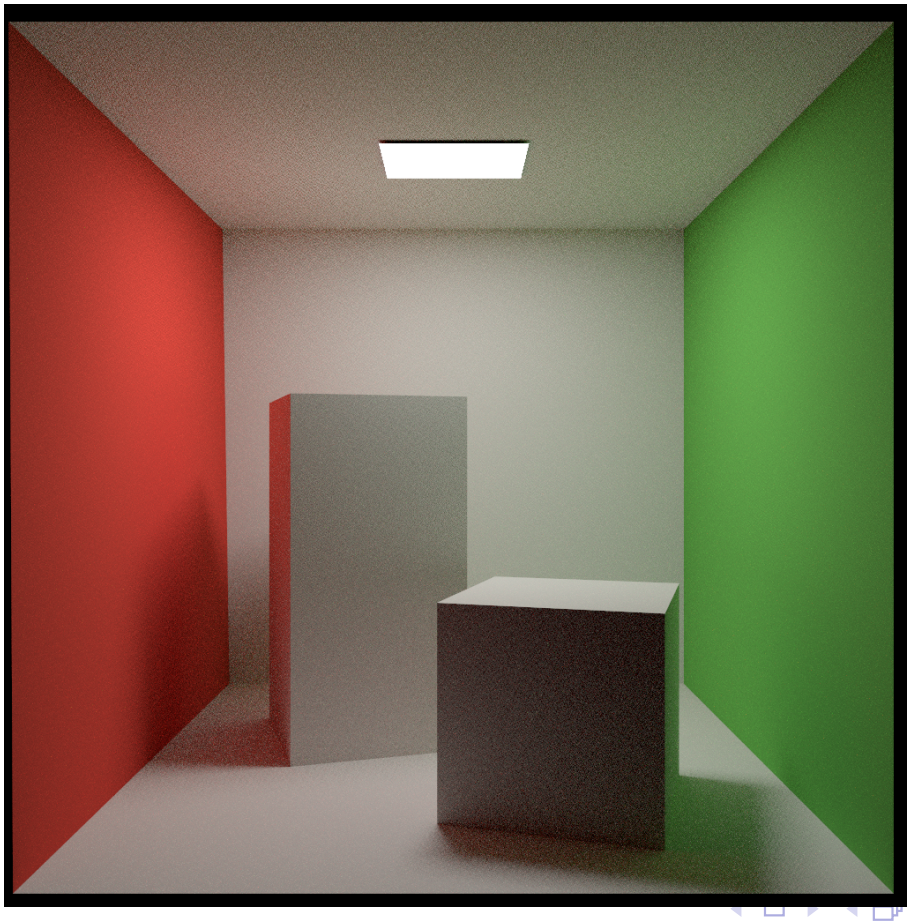
Quasi-Random Sampling

- Note that the benefit will diminish in high dimensions: “curse of dimensionality”
  - To sample the interval $[0, 1]$, you can place 100 samples so that their distances do not exceed 0.01.
  - To cover $[0, 1]^2$ with the same distances, you need $100 \times 100 = 10000$ samples.
  - For $[0, 1]^3$: 1 million
  - For $[0, 1]^4$: 100 million
- Even if we start with many samples per pixel, in higher sampling dimensions along our path our samples will be sparse!
- That is why it’s ok to fall back to pseudo-random numbers in very high dimensions

Repetition

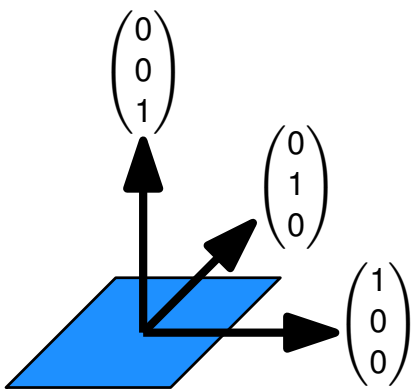
Quasi-Random Sampling: Example Result
Renderer with Multiple Importance Sampling,
Cornell Box with 1024×1024 pixels, 25 samples per pixel,
LCG pseudo-random numbers vs Halton sequence

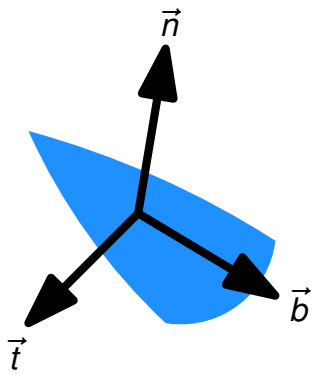




Tangent Space

Tangent Space is a normalized local space around a surface point x





The tangent space for a world space surface point x is given by

- Normal vector $\vec{n}$
- Tangent vector $\vec{t}$
- Bitangent vector $\vec{b}$

- Transformation from tangent space to world space:
  $\vec{v}_w = \begin{pmatrix}\vec{t} & \vec{b} & \vec{n}\end{pmatrix} \vec{v}_t$
- In the following, we will create and manipulate vectors in tangent space and then transform to world space

Tangent Space

How do we create the tangent space for a surface point x?

- Normal vector $\vec{n}$: interpolated mesh normal (fallback: face normal)
- Bitangent vector $\vec{b} = \vec{n} \times \vec{t}$
- Tangent vector $\vec{t}$
  - Ideally: precomputed vertex attribute depending on
    - Mesh normal vectors
    - Mesh texture coordinates

    The tangent points in the direction of the horizontal texture coordinate, the bitangent in the direction of the vertical texture coordinate.
    More on that later!
  - As a fallback:
    We don't really care where the tangent and bitangent point, they only need to be purpendicular to the normal and each other!

Tangent Space

Create a tangent: Hughes-Möller method

- Given: normal $\vec{n} = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}$
- Consider the three candidate vectors
  $\vec{t}_1 = \begin{pmatrix} -n_y \\ n_x \\ 0 \end{pmatrix} \quad \vec{t}_2 = \begin{pmatrix} 0 \\ -n_z \\ n_y \end{pmatrix} \quad \vec{t}_3 = \begin{pmatrix} -n_z \\ 0 \\ n_x \end{pmatrix}$
- All are perpendicular to $\vec{n}$ per construction:
  $\vec{n}\vec{t}_1 = -n_x n_y + n_x n_y + 0 = 0$
  $\vec{n}\vec{t}_2 = 0 - n_y n_z + n_y n_z = 0$
  $\vec{n}\vec{t}_3 = -n_x n_z + 0 + n_x n_z = 0$
- Choose the longest one, to avoid vectors with (near-)zero length
- Equivalent: choose the one where the component closest to 0 is set to 0

Tangent Space

```
class TangentSpace {
  vec3 normal, tangent, bitangent;

  TangentSpace(const vec3& n, const vec3& t) :
    normal(n), tangent(t), bitangent(cross(n, t)) { }

  TangentSpace(const vec3& n) {
    normal = n;
    vec3 w;
    if (abs(n.x()) > abs(n.z()) && abs(n.y()) > abs(n.z()))
        w = vec3(-n.y(), n.x(), 0.0f);
    else if (abs(n.y()) > abs(n.x()))
        w = vec3(0.0f, -n.z(), n.y());
    else
        w = vec3(-n.z(), 0.0f, n.x());
    tangent = normalize(w);
    bitangent = cross(normal, tangent);
  }
}
```

Tangent Space

Create a tangent: State of the Art

- T. Duff, J. Burgess, P. Christensen, C. Hery, A. Kensler, M. Liani, R. Villemin, Building an Orthonormal Basis, Revisited, Journal of Computer Graphics Techniques (JCGT), vol. 6, no. 1, 1-8, 2017

```
float sign = std::copysignf(1.0f, normal.z());
float a = -1.0f / (sign + normal.z());
float b = normal.x() * normal.y() * a;
tangent = vec3(1.0f + sign * normal.x() * normal.x() * a,
               sign * b, -sign * normal.x());
bitangent = vec3(b, sign + normal.y() * normal.y() * a,
               -normal.y());
```

- No branching! Ca. 2x faster than Hughes-Möller.
- Extensive precision analysis
- Developed and used at Pixar

Tangent Space

- Clean up our `Prng` class
  - We only need the `in01()` method
  - Remove `inUnitSphere()`, `onUnitSphere()`
- Clean up our `Sampler` class
  - All functions will transform uniformly distributed pseudo random numbers u_0, u_1, ... to samples in a normalized local space, e.g. tangent space
  - New function `uniformOnHemisphere(u0, u1)`:
    $r = \sqrt{1 - z^2}, \quad \varphi = 2\pi u_1$
    $x = r\cos\varphi, \quad y = r\sin\varphi, \quad z = u_0$
    For proof that this creates uniformly distributed samples on the hemisphere, see PBR3 Sec. 13.6.1
- Update materials Lambertian, Phong:

```
TangentSpace ts(hr.normal);
vec3 newDirTS = Sampler::uniformOnHemisphere(u0, u1);
vec3 newDirection = ts.toWorldSpace(newDirTS);
float p = 1.0f / (2.0f * pi);
```

Monte-Carlo Path Tracing

Importance Sampling

- We checked correctness and quality with *uniform* stochastic sampling

$$\int_a^b f(x)dx \quad \approx \quad J = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p_f(x_i)} \quad \text{with} \quad p_f(x_i) \equiv \frac{1}{b-a}$$

  - "Blind" Monte-Carlo integration
  - No assumptions about function $f(x)$
  - High variance $\rightarrow$ slow convergence
- Uniform stochastic sampling likely uses many samples that are useless for f!
- Solution: *Importance Sampling*

Monte-Carlo Path Tracing

Importance Sampling

- $J = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p_f(x_i)}$ with non-uniform $p_f(x_i)$
- In fact, *any* strategy $p_f(x_i)$ to choose samples x_i is correct (as long as all directions are covered):

$$
\begin{aligned}
E(J) &= E\left(\frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p_f(x_i)}\right) \\
&= \frac{1}{N} \sum_{i=1}^{N} E\left(\frac{f(x_i)}{p_f(x_i)}\right) \\
&= \frac{1}{N} \sum_{i=1}^{N} \int_a^b \frac{f(x)}{p_f(x)} p_f(x) dx \\
&= \int_a^b f(x) dx
\end{aligned}
$$

Monte-Carlo Path Tracing

Importance Sampling

- $J = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p_f(x_i)}$ with non-uniform $p_f(x_i)$
- Any strategy $p_f(x_i)$ to choose samples x_i is correct
- But when is it *better* than uniform sampling?
  I.e. when is the variance smaller?
- $$\begin{aligned} V(J) &= E\left((J - E(J))^2\right) \\ &= E(J^2) - (E(J))^2 \\ &= \frac{1}{N} \sum_{i=1}^{N} \left(\frac{f(x_i)}{p_f(x_i)}\right)^2 - \left(\frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p_f(x_i)}\right)^2 \end{aligned}$$

Monte-Carlo Path Tracing

Importance Sampling

- Which choice of $p_f(x_i)$ is *better* than uniform sampling?
- $$V(J) = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{f(x_i)}{p_f(x_i)}\right)^2 - \left(\frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p_f(x_i)}\right)^2$$
- Now $p_f(x) = \lambda f(x) \Rightarrow V(J) = 0$ if $\int_D p_f(x)dx = 1 \Rightarrow \lambda = \frac{1}{\int_D f(x)dx}$
- We cannot determine the *optimal* $p_f(x)$ because we would have to know the solution of the integral!
- But we do know *something* about the integral:

  $$L_o(\vec{x}, \omega_o) = \int_\Omega f(\vec{x}, \omega_i, \omega_o)\ L_i(\vec{x}, \omega_i)\ \cos\theta_i\ d\omega_i$$

  - Unknown: $L_o(\vec{x}, \omega_o)$, $L_i(\vec{x}, \omega_i)$
  - Known: $f(\vec{x}, \omega_i, \omega_o)$, $\cos\theta_i$
- Choosing p_f similar to the known components reduces the variance considerably!

Monte-Carlo Path Tracing

Material Importance Sampling

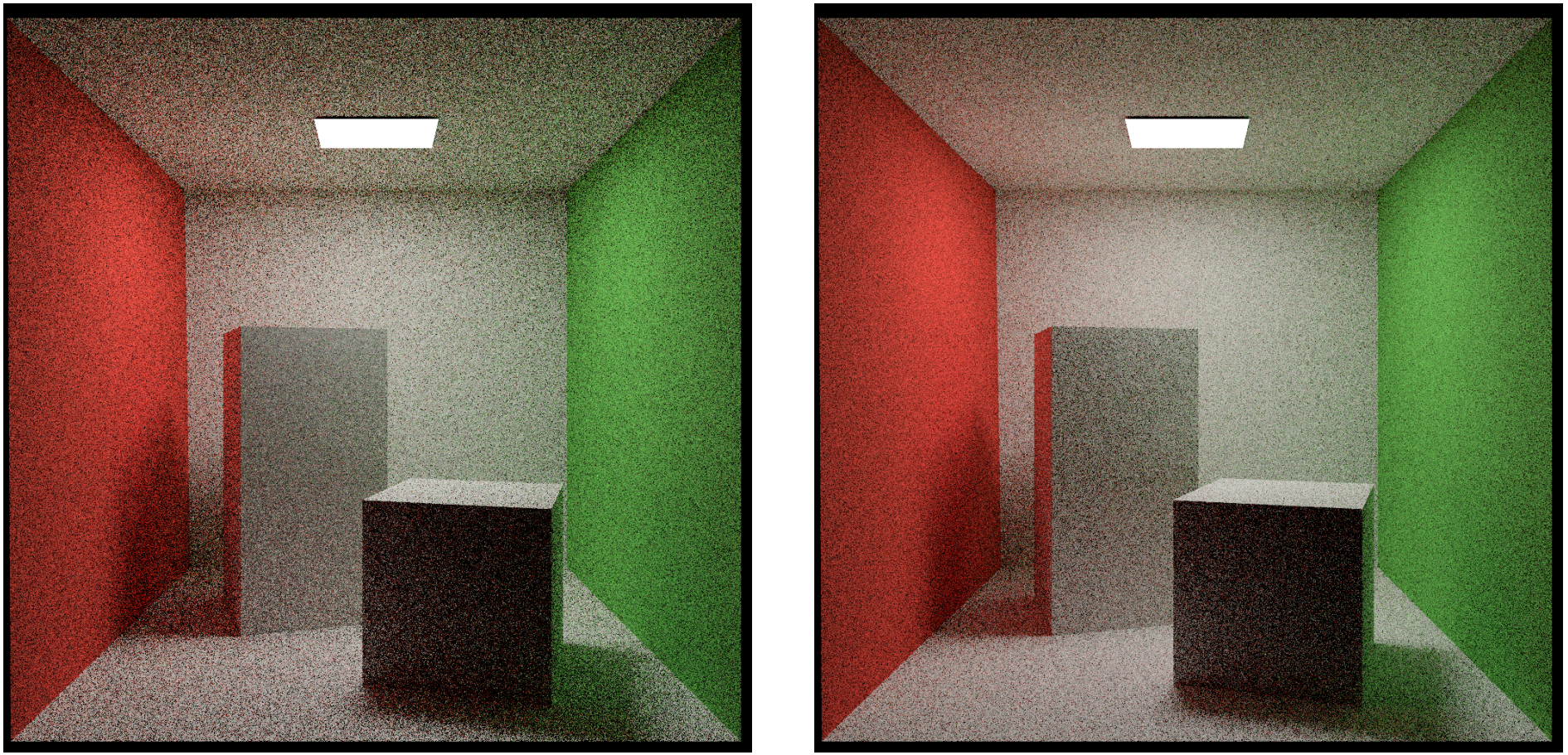
- Choose $p_f(\omega_i) \propto f(\vec{x}, \omega_i, \omega_o) \cos\theta_i$
- For Lambertian materials, the BRDF is constant $\rightarrow p_f(\omega_i) \propto \cos\theta_i$
- Cosine-weighted samples on the hemisphere: $p_f(\omega_i) = \frac{\cos\theta_i}{\pi}$
- Two-step approach:
  - Create uniformly distributed sample $d = (d_x, d_y)$ in the unit disk
  - Project d onto the unit hemisphere: $(d_x, d_y, \sqrt{1 - d \cdot d})$
  - Beware: Make sure that $1 - d \cdot d \geq 0$, otherwise z becomes NaN!

  For details, see PBR3 Sec. 13.6.2 and 13.6.3.
- Update `MaterialLambertian`:

```
vec3 tmp = Sampler::cosineWeightedOnHemisphere(u0, u1);
vec3 newDirection = ts.toWorldSpace(tmp);
float p = dot(hr.normal, newDirection) / pi;
```

Monte-Carlo Path Tracing

Material Importance Sampling in our Path Tracer: Tadaa!

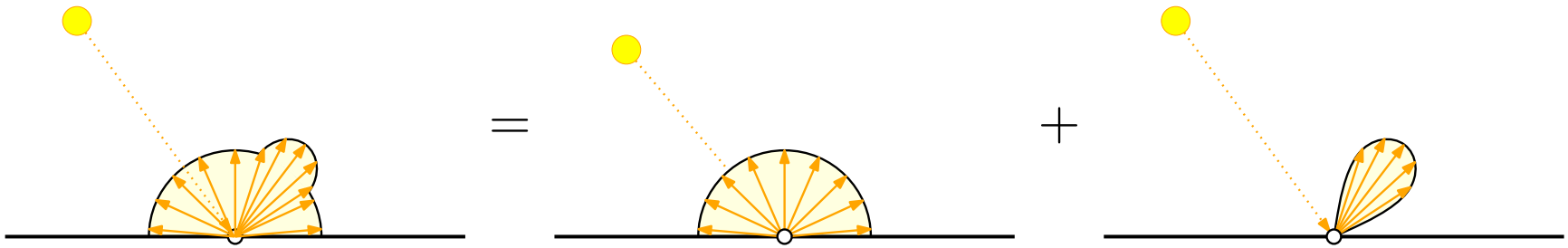


Cornell Box with 256 spp, without (left) and with (right) Material Importance Sampling

Monte-Carlo Path Tracing

Material Importance Sampling for Phong

- Remember that *any* $p_f(\omega_i)$ is correct
  (as long as all possible directions are covered)
- Even if you ignore the BRDF completely, it is still good to use cosine-weighted samples on the hemisphere according to the attenuation term
  $\rightarrow$ Cosine-weighted samples are a good default!
- For Phong: two components
  - Diffuse part: identical to Lambertian
  - Specular part: “Phong lobe”



- Decide *randomly* whether to act diffuse or specular!

Monte-Carlo Path Tracing

Material Importance Sampling for Phong

- Set a probability q to be specular, e.g. $q = \frac{1}{2}$
- Specular case:
  - Generate direction $\vec{l}$ around $\vec{r}$ (the perfect reflection direction)
    $$\vec{l} = \begin{pmatrix} \cos\varphi \sin\theta \\ \sin\varphi \sin\theta \\ \cos\theta \end{pmatrix}$$
    with $\theta = (1 - u_0)^{\frac{1}{1+s}}$, $\varphi = 2\pi u_1$
  - Transform $\vec{l}$ to world space
  - Compute its pdf value
    $p_s = \frac{s+1}{2\pi}(\cos\beta)^s$ where β is the angle between $\vec{l}$ and $\vec{r}$

E.P. Lafortune, Y.D. Willems, Using the modified phong reflectance model for physically based rendering. TR, K.U. Leuven, 1994.

- Diffuse case: identical to Lambertian
- Combine both cases:
  - Choose direction from either diffuse or specular case
  - Combined pdf $p = qp_s + (1 - q)p_d$

Monte-Carlo Path Tracing

Material Importance Sampling for Phong

```
if (prng.in01() < specularProbability) {
  vec3 newDirAroundR = Sampler::phongWeightedOnHemisphere(
          s, prng.in01(), prng.in01());
  TangentSpace rts = TangentSpace(r);
  l = normalize(rts.toWorldSpace(newDirectionAroundR));
} else {
  TangentSpace ts(n);
  vec3 newDirTS = Sampler::cosineWeightedOnHemisphere(...);
  l = normalize(ts.toWorldSpace(newDirTS));
}
float cosTheta = std::max(dot(l, n), 0.0f);
float diffusePdfValue = cosTheta / pi;
float specularPdfValue = 0.5f / pi * (s + 1.0f)
          * std::pow(std::max(dot(r, l), 0.0f), s);
float p = mix(diffusePdfValue, specularPdfValue,
      specularProbability);
```

Monte-Carlo Path Tracing

Material Importance Sampling for Phong

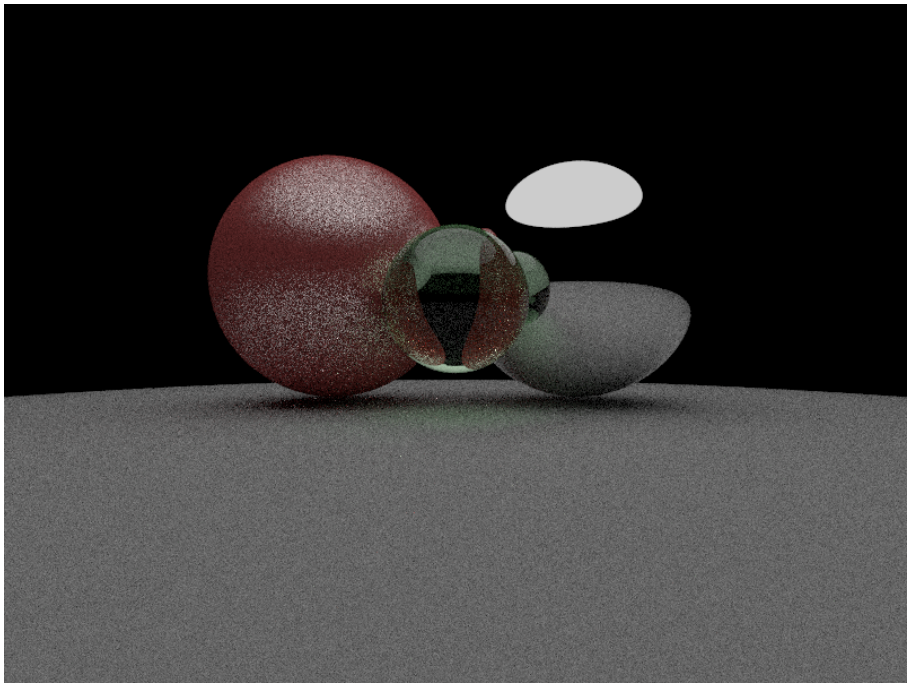
- Choosing the probability q to be specular
- Any choice of $q \in (0, 1)$ will do
- But we would like to favor the diffuse case if k_s is low, and the specular case if k_d is low
- Heuristic:

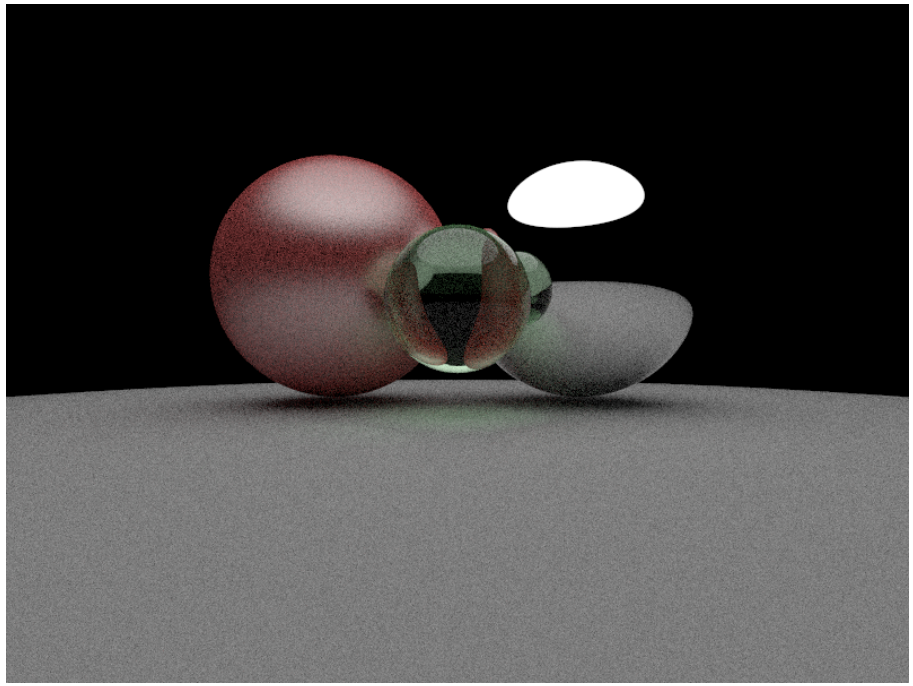
$$q \approx \frac{\sum_{R,G,B} k_s}{\sum_{R,G,B} k_d + \sum_{R,G,B} k_s}$$

```
float sumKd = kd.x() + kd.y() + kd.z();
float sumKs = ks.x() + ks.y() + ks.z();
float sum = sumKd + sumKs + 1e-4f /* so that sum > 0 */;
float specularProbability = sumKs / sum;
if (specularProbability < 0.1f)
    specularProbability = 0.1f;
else if (specularProbability > 0.9f)
    specularProbability = 0.9f;
```

Monte-Carlo Path Tracing

Phong Importance Sampling in our Path Tracer: Tadaa!





Example scene from lecture 4 with 64 spp,
without (left) and with (right) Phong Importance Sampling

Monte-Carlo Path Tracing

Light Importance Sampling

- Choose $p_f(\omega_i)$ so that ω_i goes towards a light source
  - Given: Current intersection point x and one surface that acts as a light source
  - Choose a random point P on that surface
  - New direction ω_i points from x to P
  - Compute $p_f(\omega_i)$, depending on the solid angle that the surface subtends when viewed from x
- Typically more than one surface acts as a light source, e.g. multiple triangles that form an area light
  - Given: N surfaces that act as light sources
  - Choose one surface randomly (uniformly distributed)
  - Create direction ω_i for that surface
  - Compute $p_f(\omega_i) = \frac{1}{N} \sum_{i-1}^{N} p_f(i, \omega_i)$

    where $p_f(i, \omega_i) = p_f(\omega_i)$ for surface i

Monte-Carlo Path Tracing

Light Importance Sampling: Implementation

- We need to tell a material to scatter with a *given* new direction ω_i (towards a light source)
- However that only works when the material can scatter into that direction, i.e. not for perfect reflection or refraction!
- Only apply light importance sampling if the scatter type at the current intersection is `ScatterRandom`
- That is currently the case for Lambertian and Phong materials, and not for Mirror and Glass materials
- Add a new function to the `Material` class:

```
virtual ScatterRecord scatterToDirection(const Ray& ray,
  const HitRecord& hr, const vec3& direction) const
{ return ScatterRecord(); }
```

Monte-Carlo Path Tracing

Light Importance Sampling: Implementation

- Implementation for Lambertian material:

```
virtual ScatterRecord scatterToDirection(...)
{
    float cosTheta = dot(hr.normal, direction);
    if (cosTheta <= 0.0f)
        return ScatterRecord();

    float p = cosTheta / pi;
    vec3 attenuation = brdf(hr, ray.time) * cosTheta;
    return ScatterRecord(direction, p, attenuation);
}
```

- Similarly for Phong material
- Does not have to be implemented by materials Mirror, Glass

Monte-Carlo Path Tracing

Light Importance Sampling: Implementation

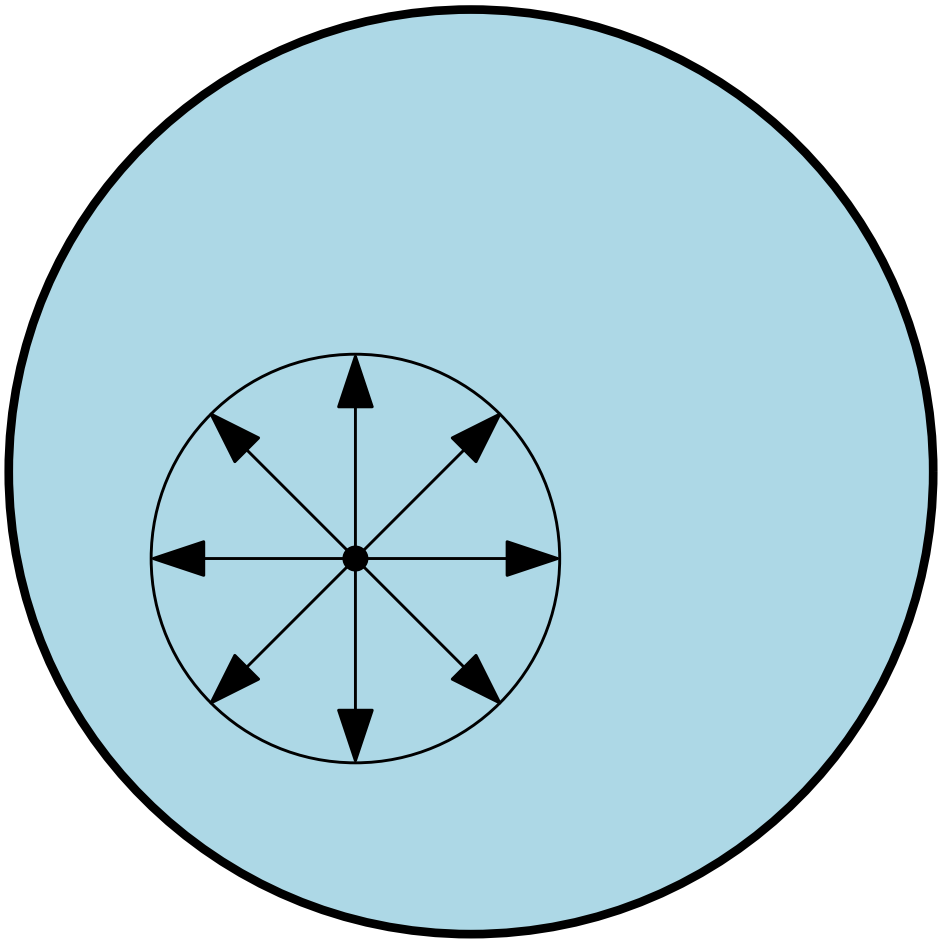
- A surface must be able to generate a direction towards itself, and to compute the pdf value for a given ray

```
class Surface {
    ...;

    virtual vec3 direction(const vec3& origin, float t,
            Prng& prng) const
    {
        return vec3(0.0f);
    }

    virtual float p(const Ray& ray) const
    {
        return 0;
    }
};
```

Monte-Carlo Path Tracing

Light Importance Sampling: Implementation

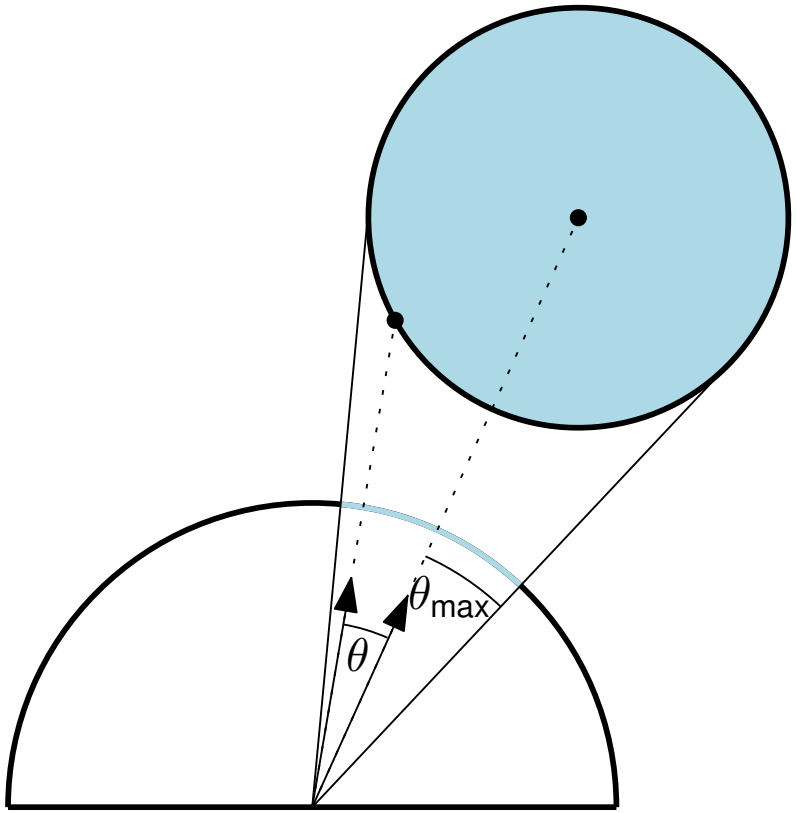
- `SurfaceSphere::direction()` and `SurfaceSphere::p()`
  - If ray starts inside the sphere:
    Sample $\vec{d}$ uniformly on sphere, $p \equiv \frac{1}{4\pi}$



Monte-Carlo Path Tracing

Light Importance Sampling: Implementation

- `SurfaceSphere::direction()` and `SurfaceSphere::p()`
  - If ray starts outside the sphere:
    Sample $\vec{d}$ uniformly across the sphere's solid angle ω, $p = \frac{1}{\omega}$



$$\omega = 2\pi(1 - \cos\theta_{max}))$$

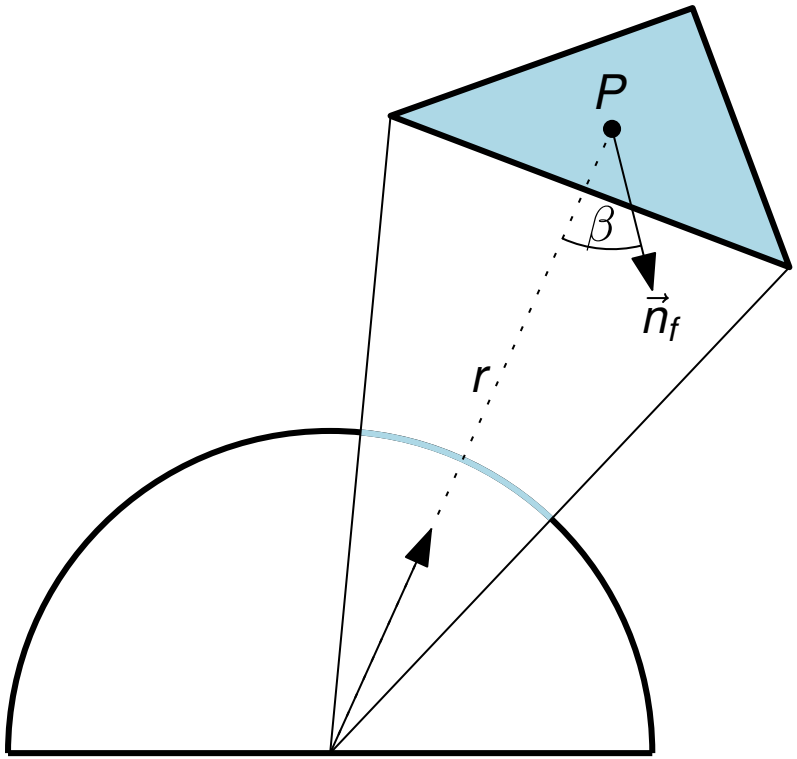
$$\theta = \text{acos}((1 - u_0) + u_0 \cos\theta_{max}))$$
$$\varphi = 2\pi u_1$$
$$\vec{d} = \begin{pmatrix} \cos\varphi \sin\theta \\ \sin\phi \sin\theta \\ \cos\theta \end{pmatrix}$$

Transform $\vec{d}$ to world space.

Monte-Carlo Path Tracing

Light Importance Sampling: Implementation

- `SurfaceTriangle::direction()` and `SurfaceTriangle::p()`
  Randomly choose point P on the triangle, uniformly distributed.
  Let $\vec{d}$ point towards P. With triangle area A: $p = \frac{r^2}{A \cos \beta}$



$\vec{n}_f$: face normal (not shading normal!)

r: distance between intersection and P

Monte-Carlo Path Tracing

Light Importance Sampling: Implementation

- `Sampler::uniformInTriangle(u0, u1)`:
  Create barycentric coordinates u,v,w from uniformly distributed random numbers $u_0, u_1 \in [0, 1)$

  $u = 1 - \sqrt{u_0}$
  $v = u_1\sqrt{u_0}$
  $w = 1 - u - v$

  See PBR3 Sec.13.6.5

Monte-Carlo Path Tracing

Light Importance Sampling: Implementation

```
vec3 direction(origin, t, prng) {
    vec3 A, B, C; // untransformed vertices
    vec3 bary = Sampler::uniformInTriangle(in01(), in01());
    vec3 P = vec3(bary.x() * A + bary.y() * B + bary.z() * C);
    if (mesh.animation) P = mesh.animation->at(t) * P;
    return normalize(P - origin);
}
float p(ray) {
    if (/* ray does not hit this triangle */) return 0.0f;
    vec3 A, B, C; // transformed vertices at time t
    vec3 edgeCross = cross(B - A, C - A);
    float edgeCrossLength = sqrt(dot(edgeCross, edgeCross));
    vec3 faceNormal = edgeCross / edgeCrossLength;
    float faceArea = 0.5f * edgeCrossLength;
    float cosine = abs(dot(faceNormal, -ray.direction));
    return hr.a * hr.a / (cosine * faceArea);
}
```

Monte-Carlo Path Tracing

Light Importance Sampling: Implementation

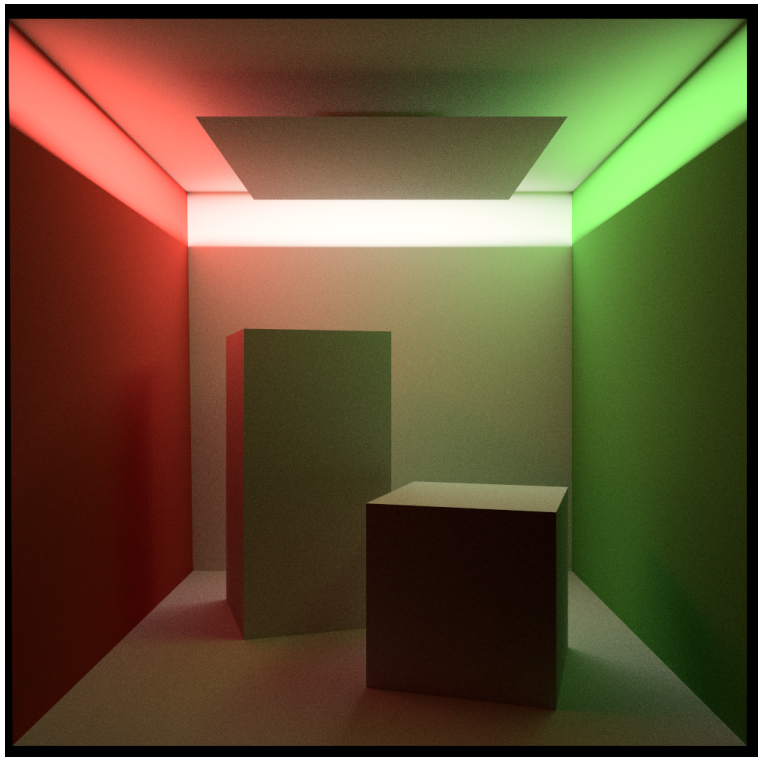
- Our scene must keep a list of surfaces that act as light sources

```
class Scene {
  ...;
  std::vector<const Surface*> lights;
  Surface* take(Surface* surf, bool isLight = false) {
    surfaces.push_back(std::unique_ptr<Surface>(surf));
    if (isLight) lights.push_back(surf);
    return surf;
  }
  Mesh* take(Mesh* mesh, bool isLight = false) {
    meshes.push_back(std::unique_ptr<Mesh>(mesh));
    for (size_t i = 0; i < mesh->surfaces(); i++)
        take(mesh->createSurface(i), isLight);
    return mesh;
  }
```

Monte-Carlo Path Tracing

Light Importance Sampling: Implementation

- Note that surfaces marked with the `isLight` flag do not have to emit light themselves, and that not all light emitters in the scene have to be marked



- For special scenes, it might make sense to use light important sampling not on the light sources themselves (“light portals”)
- Nevertheless, our OBJ import marks all surfaces that have `MaterialLight` with the `isLight` flag

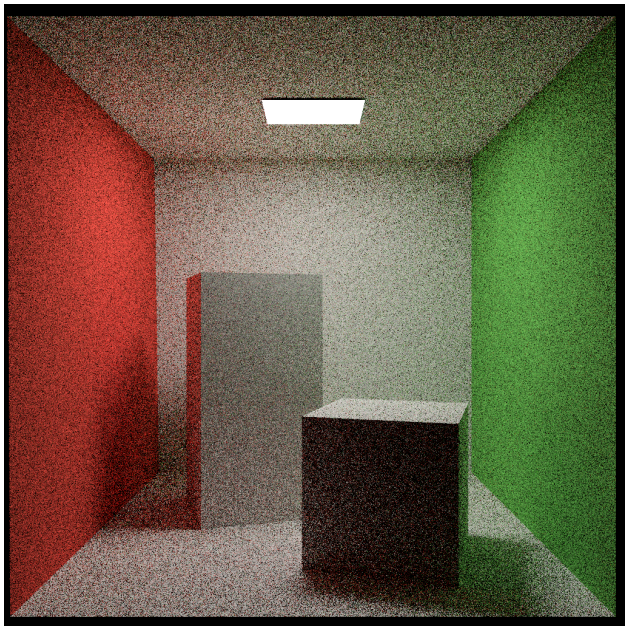
Monte-Carlo Path Tracing

Light Importance Sampling: Implementation

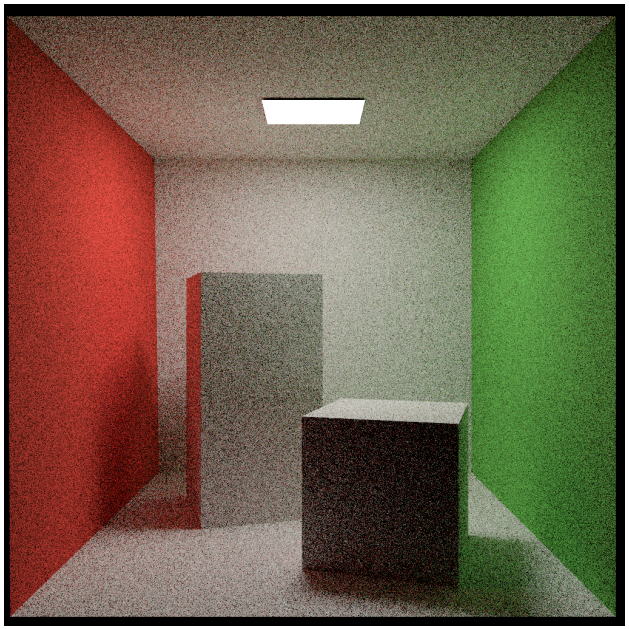
```
for (int segment = 0; segment < maxPathSegments; segment++) {
    HitRecord hr = scene.bvh.hit(ray, 0.0001f, FLT_MAX);
    if (!hr.haveHit) break;

    size_t lightIndex = prng.in01() * scene.lights.size();
    vec3 lightDir = scene.lights[lightIndex]->direction(
            hr.position, ray.time, prng);
    float lightDirP = 0.0f;
    for (size_t i = 0; i < scene.lights.size(); i++)
        lightDirP += scene.lights[i]->p(Ray(hr.position,
                lightDir, ray.time));
    lightDirP /= scene.lights.size();
    ScatterRecord sr = hr.material->scatterToDirection(
            ray, hr, lightDir);
    ...;
}
```

Monte-Carlo Path Tracing

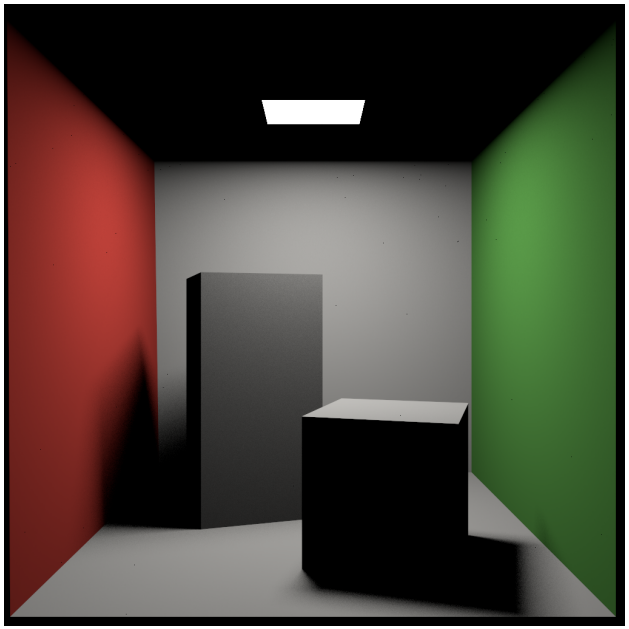
Light Importance Sampling in our Path Tracer: Tadaa!



No IS



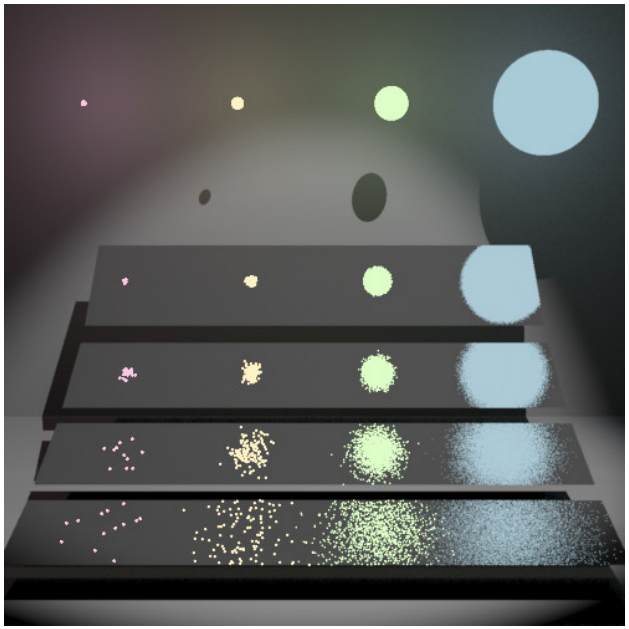
Material IS



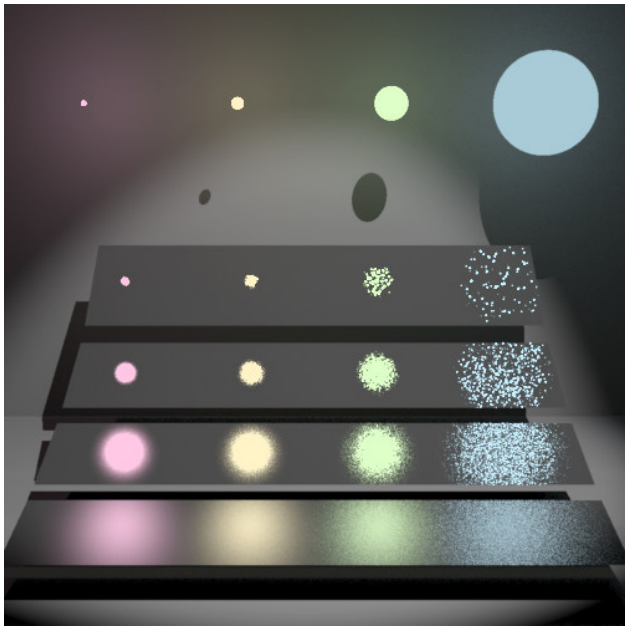
Light IS

Monte-Carlo Path Tracing

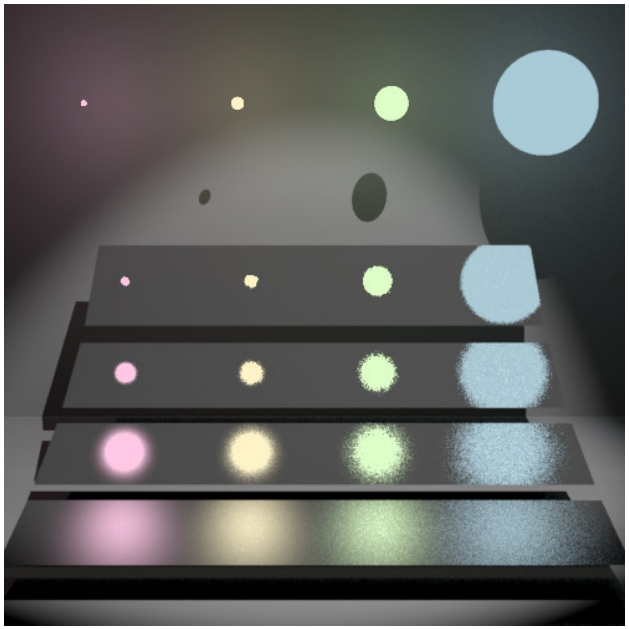
Material IS or Light IS? Both!



Material IS



Light IS



Multiple IS

Images from the PhD thesis “Robust Monte Carlo Methods for Light Transport Simulation” by Eric Veach, Dec. 1997

Monte-Carlo Path Tracing

Multiple Importance Sampling

- Previously: integral over one function f with samples chosen according to pdf p_f:

$$\int_a^b f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p_f(x_i)}$$

- Now: integral over the product of two functions f and g with samples chosen according to their pdfs p_f and p_g:

$$\int_a^b f(x)g(x)dx \approx \frac{1}{N}\sum_{i=1}^{N}\left(\frac{f(x_i)g(x_i)w_f(x_i)}{p_f(x_i)} + \frac{f(y_i)g(y_i)w_g(y_i)}{p_g(y_i)}\right)$$

- The weighting function w can be chosen freely
- A provably good (i.e. variance reducing) weighting function is the balance heuristic:

$$w_f(x_i) = \frac{p_f(x_i)}{p_f(x_i)+p_g(x_i)} \qquad w_g(x_i) = \frac{p_g(x_i)}{p_f(x_i)+p_g(x_i)}$$

Monte-Carlo Path Tracing

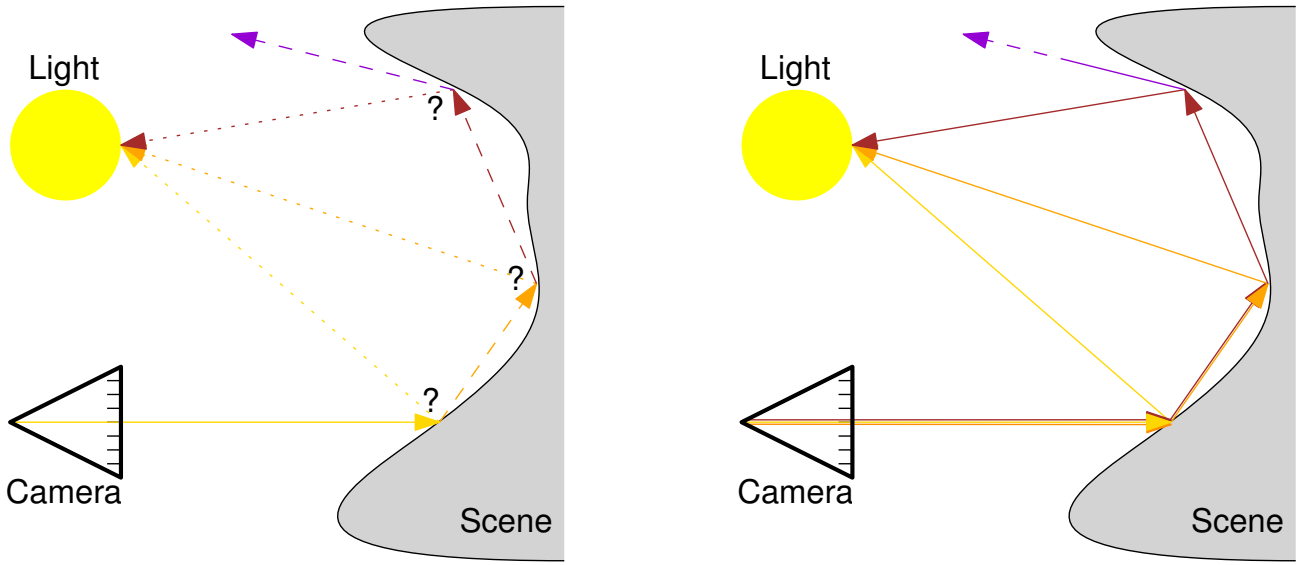
Multiple Importance Sampling

- $$\int_a^b f(x)g(x)dx \approx \frac{1}{N}\sum_{i=1}^{N}\left(\frac{f(x_i)g(x_i)w_f(x_i)}{p_f(x_i)} + \frac{f(y_i)g(y_i)w_g(y_i)}{p_g(y_i)}\right)$$
- In Monte Carlo Path Tracing:
  - The two terms in the sum are two path samples,
    one for direction x_i from material IS, and
    one for direction y_i from light IS;
    there are no distinct functions f and g
  - The power heuristic is even better than the balance heuristic in practice:
    $$w_f(x_i) = \frac{p_f^\beta(x_i)}{p_f^\beta(x_i) + p_g^\beta(x_i)} \quad w_g(x_i) = \frac{p_g^\beta(x_i)}{p_f^\beta(x_i) + p_g^\beta(x_i)}$$
  - PBRT and Mitsuba (and probably everybody else) use the power heuristic with $\beta = 2$, originally proposed by E. Veach

Monte-Carlo Path Tracing

Multiple Importance Sampling: Implementation

- Two variants:
  1. At each intersection, choose randomly whether to follow the path according to the material or the light sources
  2. At each intersection, divide the current path into two logical paths
     - The first uses material IS and continues into the scene
     - The second uses light IS and ends at the light source
     - Both logical paths contribute to the final radiance



- The second approach is much more efficient due to path reuse. PBRT, Mitsuba (and probably all others) implement this.

Monte-Carlo Path Tracing

Multiple Importance Sampling: Implementation

- We must know exactly which surface we hit, to check whether we hit the intended light source (or whether it is occluded):
  Add a pointer to the surface to `HitRecord`
- In `pathSample()`:
  - Hit the scene (no hit: path ends)
  - Scatter according to material
  - Add emitted radiance
  - End path if no further scattering occured
  - `nextThroughput = throughput * sr.attenuation / sr.p;`
  - If `ScatterRandom` and at least one light source:
    - Apply direct lighting via MIS (see next slide)
  - Update throughput and ray
  - Apply Russian Roulette

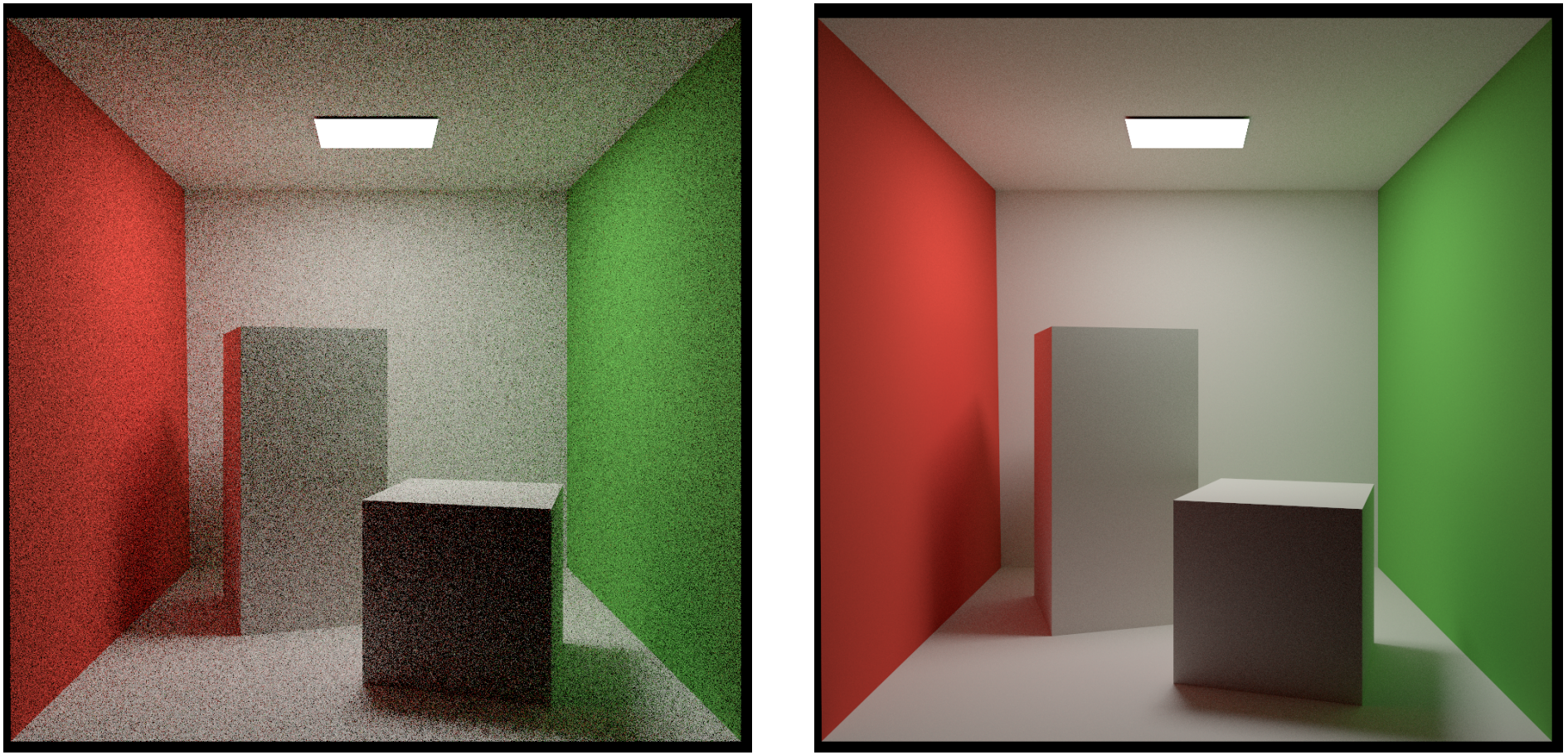
Monte-Carlo Path Tracing

Multiple Importance Sampling: Implementation

- If `ScatterRandom` and at least one light source:
  - Compute pdf for the material direction according to the light sources
  - `nextThroughput *= powerHeuristic(sr.p, lightsP);`
  - Choose light source, generate direction, compute its pdf
  - Scatter into that direction (if impossible, cancel attempt)
  - Check if ray actually hits chosen light source (otherwise cancel)
  - Add radiance with attenuation according to material and weight according to power heuristic

Monte-Carlo Path Tracing

Multiple Importance Sampling in our Path Tracer: Tadaa!



Cornell Box with 256 spp, with Material IS (left) and Multiple IS (right)

Monte-Carlo Path Tracing

Milestone reached!

All major architectural changes are done.

The rest will just be icing on the cake :)