Path Tracing

Priv. Doz. Dr. Ing. Martin Lambers

Repetition

Accelerating our Path Tracer

Currently we call `Surface::hit()` for **all surfaces** in the scene

- For every pixel
- For every sample per pixel
- For every path segment

Cornell Box	Utah Teapot	Crytek Sponza	San Miguel
36 triangles	172721 (HiRes)	262267	9980699

. . . we have to be smarter!

Repetition

Accelerating our Path Tracer

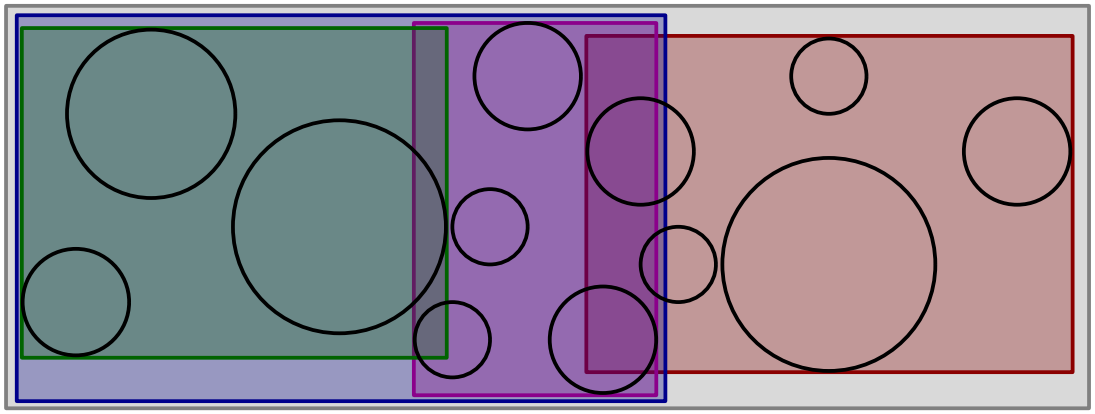
Currently we call `Surface::hit()` for **all surfaces** in the scene

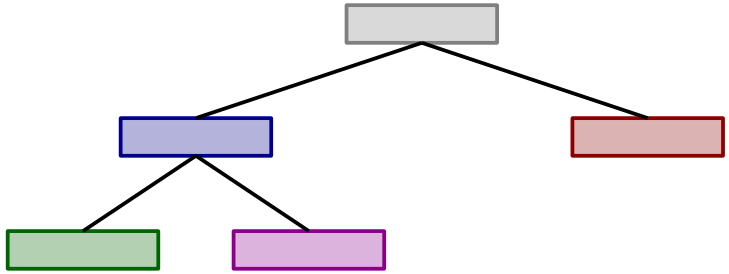
- For every pixel
  Nothing we can do other than reducing resolution
- For every sample per pixel
  $\rightarrow$ Monte Carlo techniques: Importance Sampling, Stratified Sampling, Quasi Monte Carlo Sampling
- For every path segment
  $\rightarrow$ Monte Carlo technique: Russian Roulette

Today we focus on not having to intersect our ray with **all surfaces**.

Repetition

Idea: Hierarchy of Bounding Volumes





- Sublinear complexity requires a hierarchy
- Choose a bounding volume type that allows fast ray intersection
- Ray does not intersect volume
  $\Rightarrow$ ray does not intersect anything in the volume
- Quick rejection of large parts of a scene

Repetition

Axis-Aligned Bounding Boxes

- All edges are parallel to the main axes x, y, z
- All sides are parallel to the main planes x/y, x/z, y/z
- Along each axis, the box defines an interval:
  $[x_{\min}, x_{\max}]$ $[y_{\min}, y_{\max}]$ $[z_{\min}, z_{\max}]$
- AABB can be represented by lower corner lo $= (x_{\min}, y_{\min}, z_{\min})$ and higher corner hi $= (x_{\max}, y_{\max}, z_{\max})$
- We only need to know *if* there is an intersection, not *where*

```
class AABB {
    vec3 lo, hi;
    bool hit(const Ray& ray, float amin, float amax) const
    { ... }
};
```

Repetition

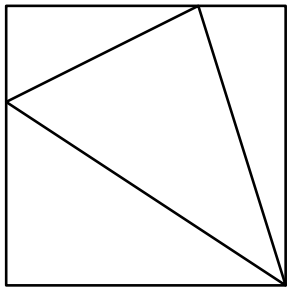
Axis-Aligned Bounding Boxes: Ray Intersection

- Since we need $\frac{1}{\vec{d}}$ for every Ray/AABB intersection test, we can precompute it and store it with the ray:

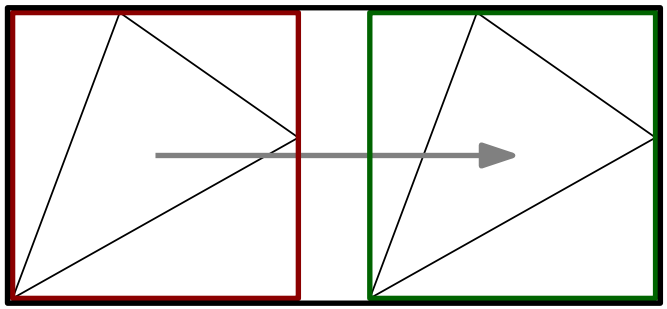
```
class Ray
{
    vec3 origin;
    vec3 direction; // must have unit length!
    vec3 invDirection;
    float time;

    Ray(const vec3& o, const vec3& d, float t) :
        origin(o),
        direction(d),
        invDirection(vec3(1.0f) / d),
        time(t)
    {
    }
```

Repetition

Generating Axis-Aligned Bounding Boxes



- For a static triangle (A, B, C):
  $\text{lo} = \min(A, B, C)$
  $\text{hi} = \max(A, B, C)$
- For an animated triangle:
  - assume linear motion in exposure interval $[t_0, t_1]$
  - merge the AABBs of triangle at t_0 and triangle at t_1



- If the linear motion assumption does not hold:
  Sample n time steps in $[t_0, t_1]$ and merge the resulting AABBs
  This still assumes linear motion in every time step!
- For more information, see PBR3 Sec. 2.9.3

Repetition

Bounding Volume Hierarchy

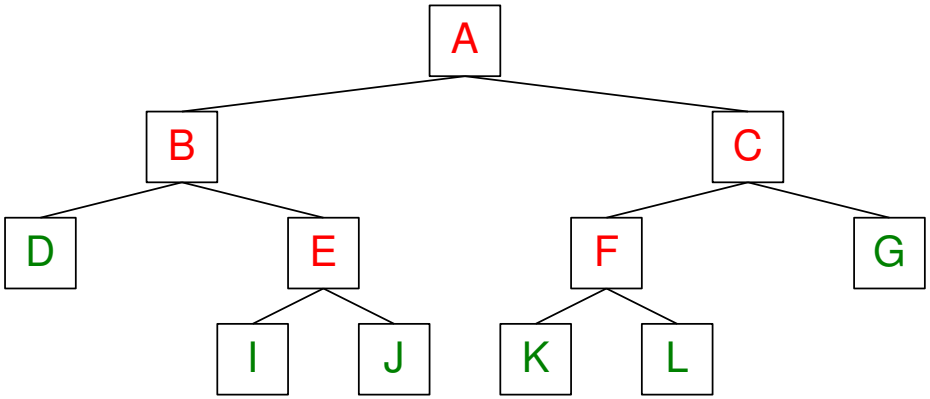
- A BVH tree is build recursively top-down:

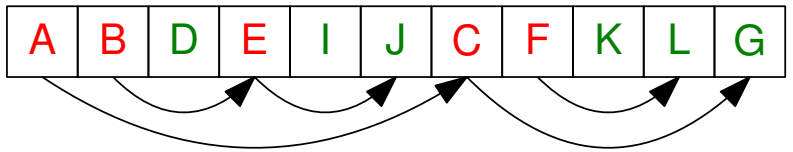
  Given N surfaces
  - If $N = 1$, this is a leaf node:
    - Store the surface
    - Set AABB to the surface AABB
    - End recursion
  - If $N > 1$:
    - Set AABB to the merged AABBs of all surfaces
    - Find the longest axis of the AABB
    - Sort the surfaces according to their center along this axis
    - Split the set of surfaces at index $i = \frac{N}{2}$
      (Note: we will look at a better strategy later)
    - Recursively build child nodes for
      surface set $\{0, \ldots, i-1\}$ and
      surface set $\{i, \ldots, N-1\}$

Repetition

Bounding Volume Hierarchy

- Tree traversal is expensive (“pointer chasing”)
- Optimization: linearize the BVH into an array
- If a node is a leaf node:
  - The node stores a pointer to the surface
- Otherwise:
  - The first child is the next node in the array
  - The node stores the index to the second child





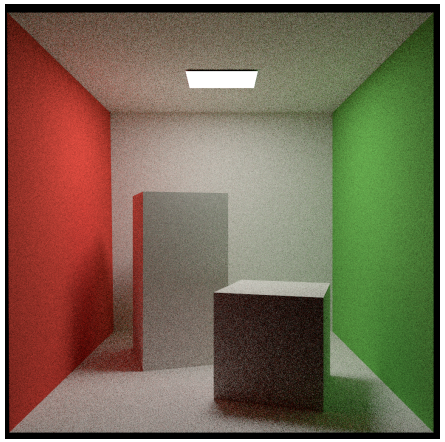
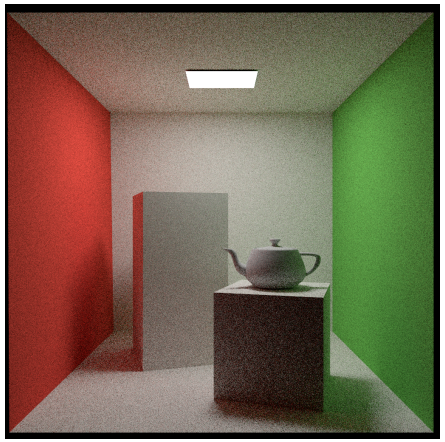
Repetition

Bounding Volume Hierarchy

- Revisit splitting criterion:
  - Find the longest axis of the AABB
  - Sort the surfaces according to their center along this axis
  - Split the set of surfaces at index $i = \frac{N}{2}$
- Any split criterion will result in a *correct* BVH
- But which split criterion will result in a *good* BVH?
- No universal solution for general scenes

- The Surface Area Heuristic (SAH)
  - Assume that the probability for ray/box intersection is proportional to the surface area of the box
  - Favor AABBs that
    - Contain many surfaces
    - Have a low probability for ray intersection
- Results in very efficient BVHs in practice!

Repetition

Scene		
Triangles	36	172756
Relative duration without BVH	1	ca. 8300
Relative duration with BVH	0.29	0.35

- Without a BVH, the computational costs increase at least linearly with the number of triangles
- Rendering even moderately complex geometry *requires* a BVH

Repetition

Pixel-level parallelization

- Static parallelization on pixel level is not optimal
- Paths take different amounts of time to compute!
- Example: some paths don't hit any geometry at all, others bounce around many times before hitting a light source
- Some CPU threads finish early, others still have to compute, the CPU is under-utilized
- Better: dynamic parallelization
  Each idle CPU thread takes the next unprocessed pixel

```
#pragma omp parallel for schedule(dynamic)
for (int pixel = 0; pixel < width * height; pixel++) {
    ...;
}
```

- This does not result in reduced costs, only in better scheduling!
- Example result: 101 seconds instead of 123 seconds wall clock time for the Cornell Box with teapot scene on my machine

Repetition

Tile-level parallelization

- Useful when computing a full frame takes too long on one machine
- Subdivide frame into $R \times C$ tiles of size $N \times M$
- Each process computes just one tile
- Recompose complete frame from the tiles

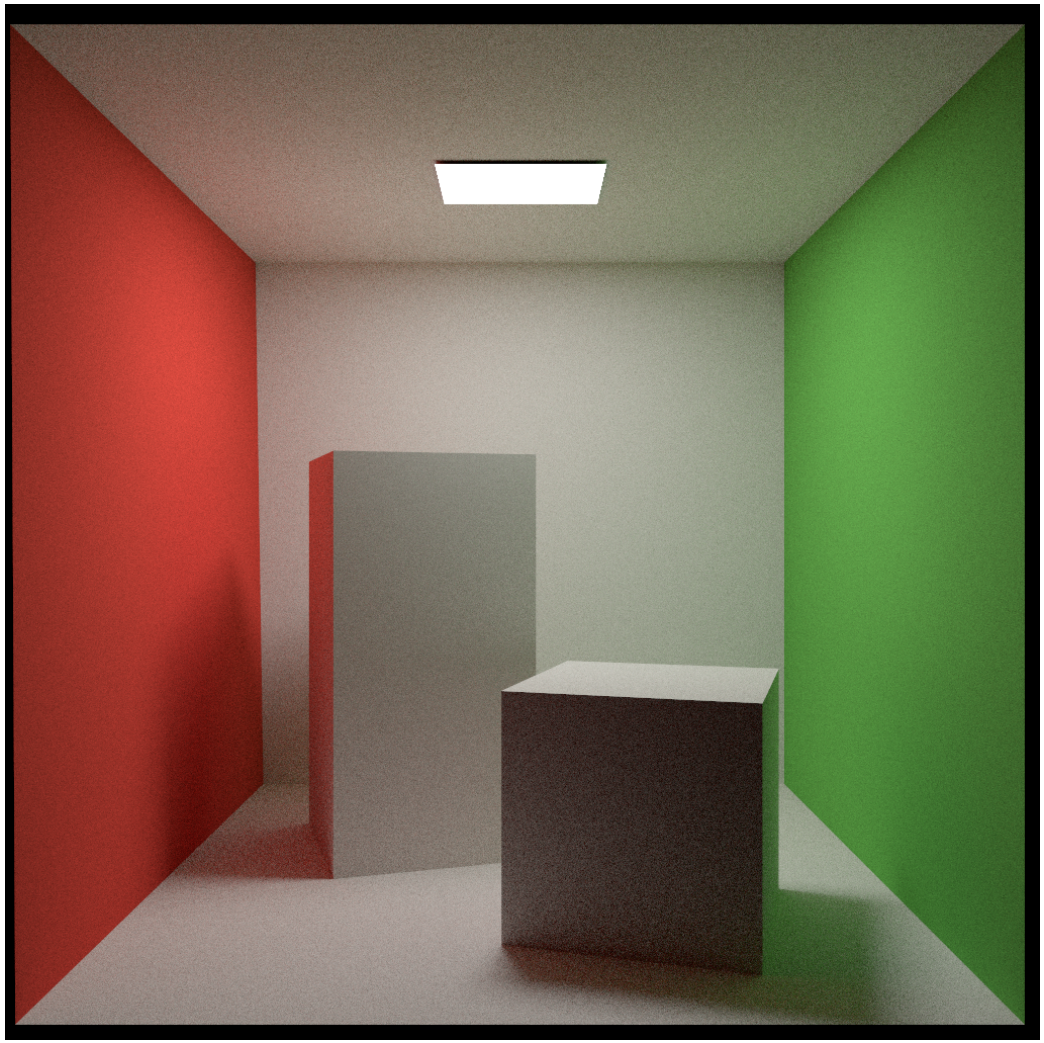
Frame-level parallelization

- Useful when computing a sequence of N frames (for movies etc.)
- Each process computes one frame
- Encode video once all frames are complete

Tile-level and Frame-level parallelization target multi-CPU systems, e.g. a cluster for High Performance Computing (HPC)

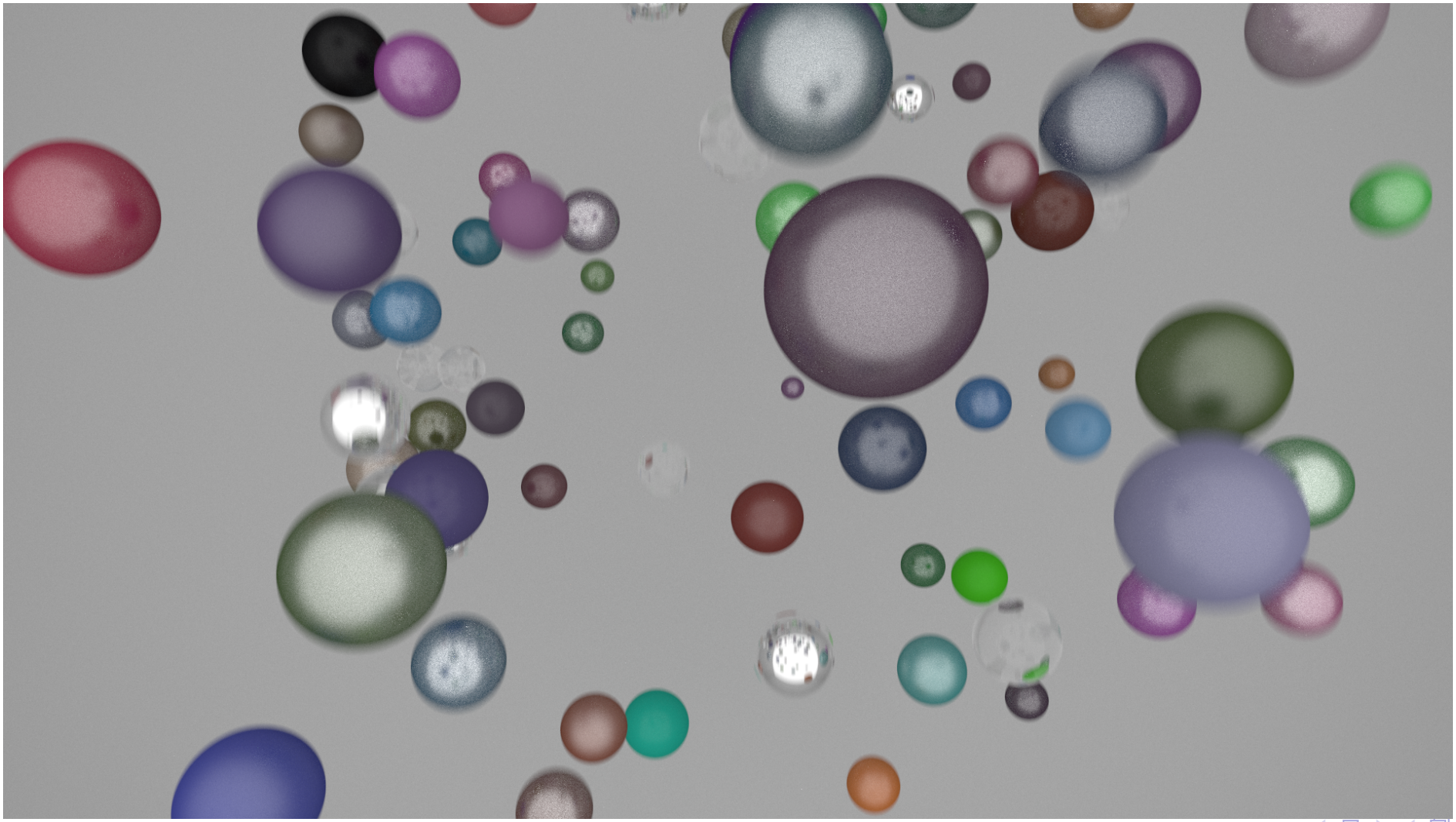
Repetition

Using a HPC cluster (tiling, 8192 spp): Tadaa!



Repetition

Using a HPC cluster (250 frames, 2048 spp): Tadaa!



Monte-Carlo Path Tracing

Accelerating our Path Tracer

Currently we call `Surface::hit()` for **all surfaces** in the scene

- For every pixel
- For every sample per pixel
- For every path segment

Cornell Box

36 triangles

Utah Teapot

172721 (HiRes)

Crytek Sponza

262267

San Miguel

9980699

. . . we have to be smarter!

Monte-Carlo Path Tracing

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.

Monte-Carlo Path Tracing

Basic Stochastics

- Discrete random variables, e.g. rolling a perfect dice: $x \in \{1, 2, 3, 4, 5, 6\}$
  - Probability $p(x)$ of a result x: $0 \leq p(x) \leq 1 \quad \forall x$
  - Certain result: $p(x) = 1$
  - Impossible result: $p(x) = 0$
  - There's always a result: $\sum_x p(x) = 1$
  - Perfect dice: $p(x) = \frac{1}{6} \quad \forall x$

Monte-Carlo Path Tracing

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:
    $V(x) = \sigma^2$

Monte-Carlo Path Tracing

Basic Stochastics

- Functions $f(x)$ of random variables x
  - Expected value:
    $E(f(x)) = \sum_x f(x)p(x)$
  - Rules:
    $E(f(x) + g(y)) = E(f(x)) + E(g(y))$
    $E(a \cdot f(x)) = a \cdot E(f(x))$

Monte-Carlo Path Tracing

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:

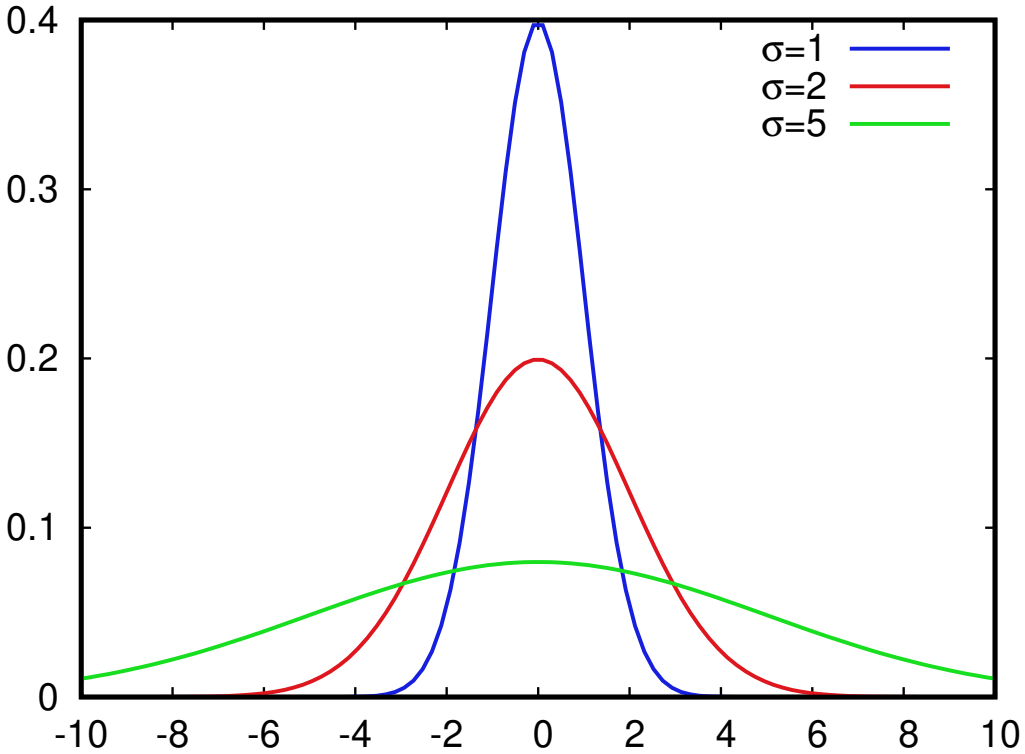
    $$E(x) \approx \frac{1}{N} \sum_{i=1}^{N} x_i$$

Monte-Carlo Path Tracing

Basic Stochastics

- Example: Normal distribution (Gaussian distribution)
  Expected value μ, standard deviation σ

$$p_f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$



Monte-Carlo Path Tracing

Monte-Carlo Estimation of an Integral

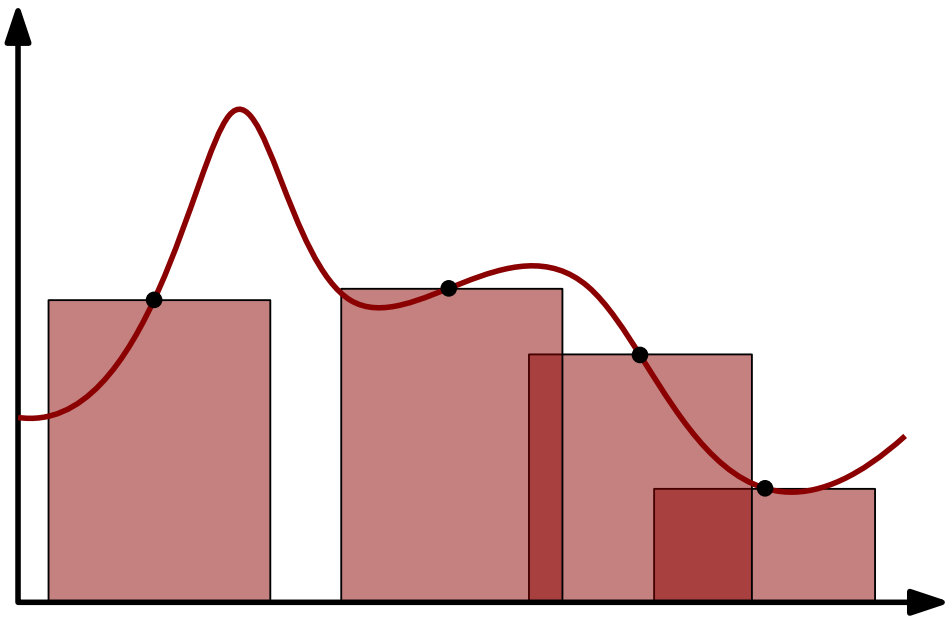
- $I = \int_a^b f(x)dx$
- Compute samples $f(x_i)$ at discrete sample points x_i
- An **estimator** J uses these samples to estimate I
- For an **unbiased** estimator: $E(J) = I$
  This is what we want in physically based rendering!
  (However that does not mean that biased estimators are useless.)
-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

Monte-Carlo Path Tracing

Monte-Carlo Estimation of an Integral

- Stochastic sampling with uniform distribution

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

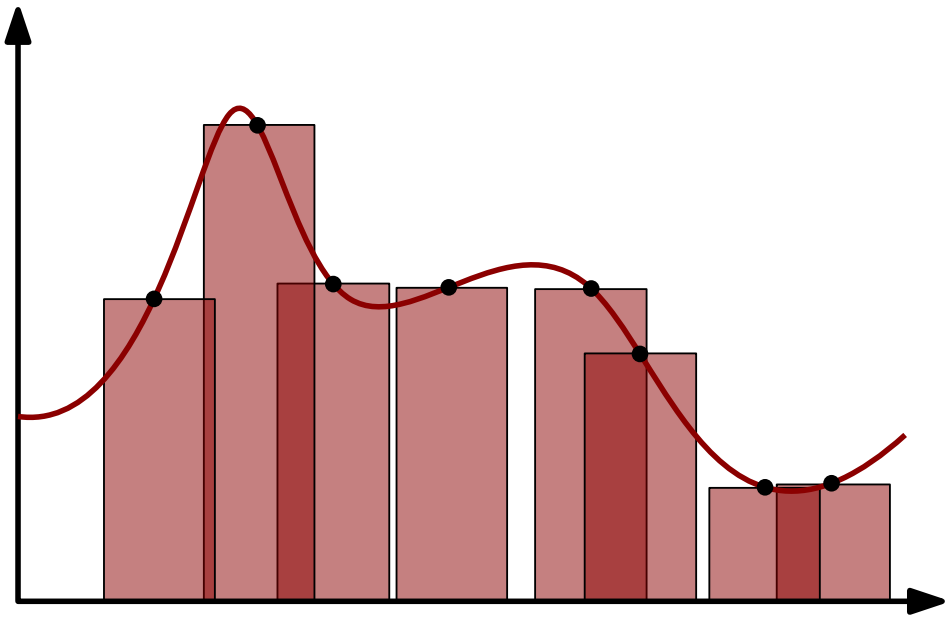


Monte-Carlo Path Tracing

Monte-Carlo Estimation of an Integral

- Stochastic sampling with uniform distribution

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

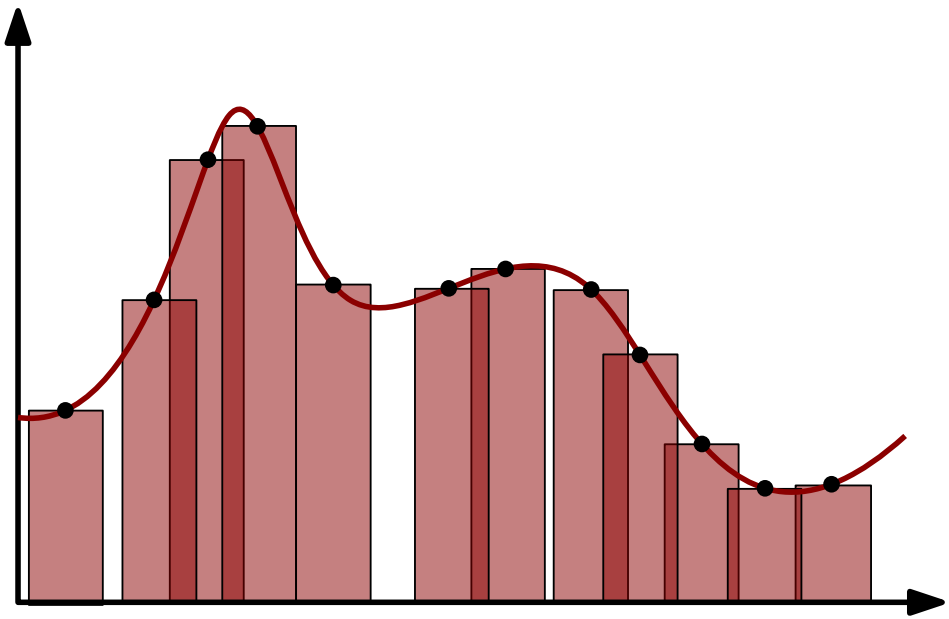


Monte-Carlo Path Tracing

Monte-Carlo Estimation of an Integral

- Stochastic sampling with uniform distribution

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

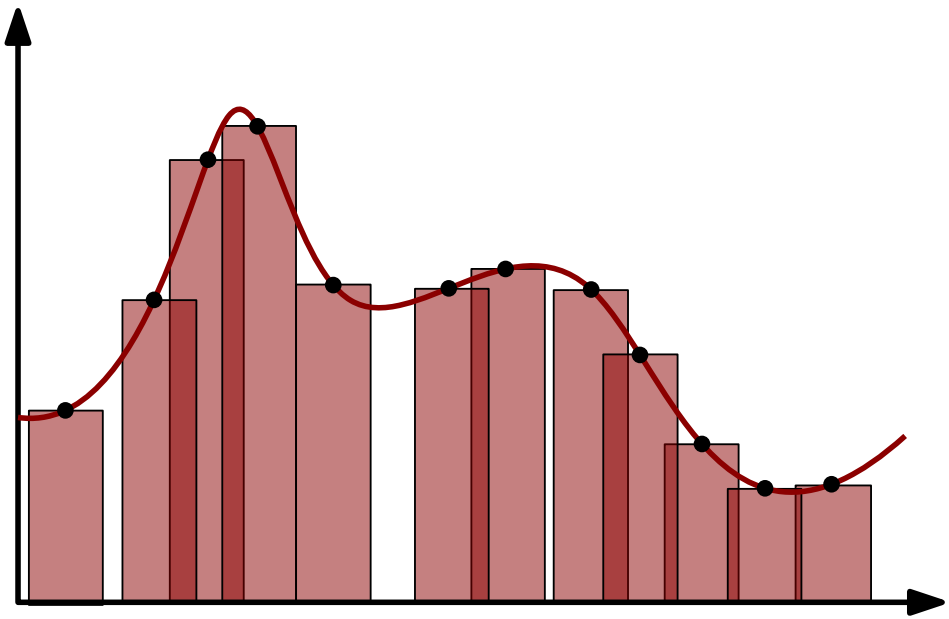


Monte-Carlo Path Tracing

Monte-Carlo Estimation of an Integral

- Stochastic sampling with uniform distribution

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



- This is how we currently estimate the Rendering Equation!

Monte-Carlo Path Tracing

Monte-Carlo Estimation of an Integral

- Is this an *unbiased estimator*?

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

- Yes!

Monte-Carlo Path Tracing

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.

Monte-Carlo Path Tracing

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!

Monte-Carlo Path Tracing

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: $q \approx 1 -$ throughput
  Lower throughput will lead to higher probability to cancel the path

Monte-Carlo Path Tracing

Path Termination via Russian Roulette

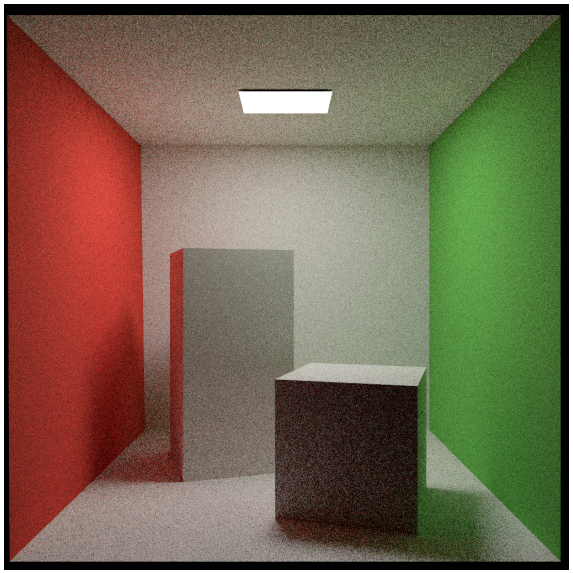
- Practical implementation (based on PBRT3 and Mitsuba2):

```
for (int segment = 0; segment < maxSegments; segment++) {
    ...;

    float maxThroughput = std::max(throughput.x(),
        std::max(throughput.y(), throughput.z()));
    if (maxThroughput < 1.0f && segment >= 5) {
        float q = 1.0f - maxThroughput;
        if (q > 0.95f)
            q = 0.95f;
        if (prng.in01() < q)
            break;
        throughput *= 1.0f / (1.0f - q);
    }
}
```

Monte-Carlo Path Tracing

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!

Monte-Carlo Path Tracing

Accelerating our Path Tracer

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

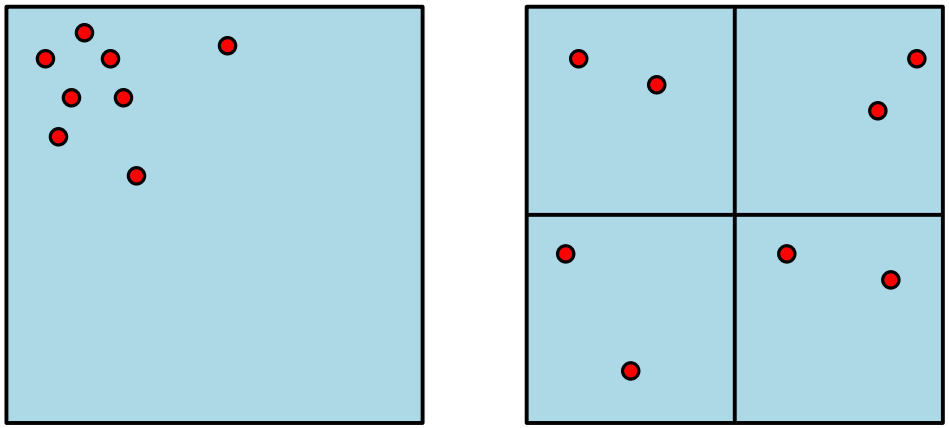
- For every pixel
  Nothing we can do other than reducing resolution
- For every sample per pixel
  $\rightarrow$ Monte Carlo techniques: Importance Sampling, Stratified Sampling, Quasi Monte Carlo Sampling
- For every path segment
  $\rightarrow$ Monte Carlo technique: Russian Roulette

Now we focus on reducing the number of samples per pixel to get a visually good result.

Monte-Carlo Path Tracing

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

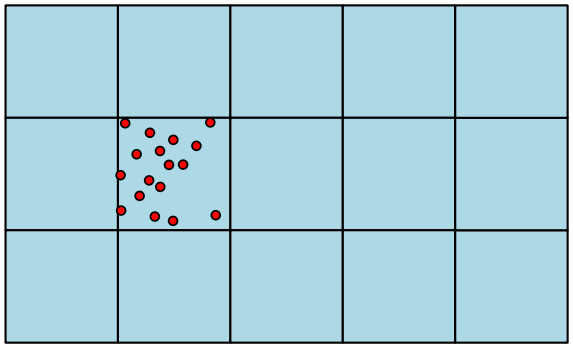


- Variance with stratified sampling is never larger than variance without stratified sampling!

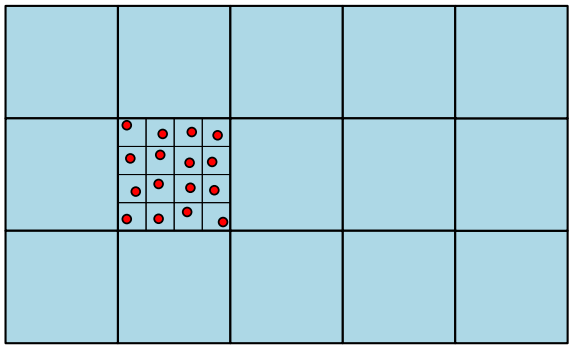
Monte-Carlo Path Tracing

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

Monte-Carlo Path Tracing

Stratified Sampling

- Each use of the pseudo random number generator can be seen as one dimension of our Monte-Carlo integration
  - Dimensions 0 and 1: pixel sampling
  - Dimension 2: time sampling in Camera::getRay()
  - 8 dimensions at each intersection i:
    - Dimensions 3+8i – 3+8i+2: new direction
    - Dimensions 3+8i+2 – 3+8i+4: material specifics
    - Dimensions 3+8i+5 – 3+8i+6: other purposes
    - Dimension 3+8i+7 : Russian Roulette
- We would like stratified sampling in all dimensions, not just the first two!

Monte-Carlo Path Tracing

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).

Monte-Carlo Path Tracing

Quasi-Random Sampling

- Example, similar to an existing renderer:

```
virtual ScatterRecord Material::scatter(ray, hitRecord,
    const Sampler& s, /* l-d sequence generator */
    uint sIndex,      /* index of sample per pixel */
    uint sDim,        /* sample dimension */
    Prng& prng)       /* for fallback at high dims */
{
    float u0 = s.in01(sIndex, sDim, prng);
    float u1 = s.in01(sIndex, sDim + 1, prng);
    // u0, u1 for new ray direction
    float u2 = s.in01(sIndex, sDim + 2, prng);
    float u3 = s.in01(sIndex, sDim + 3, prng);
    // u2, u3 for other purposes
    ...;
}
```

Monte-Carlo Path Tracing

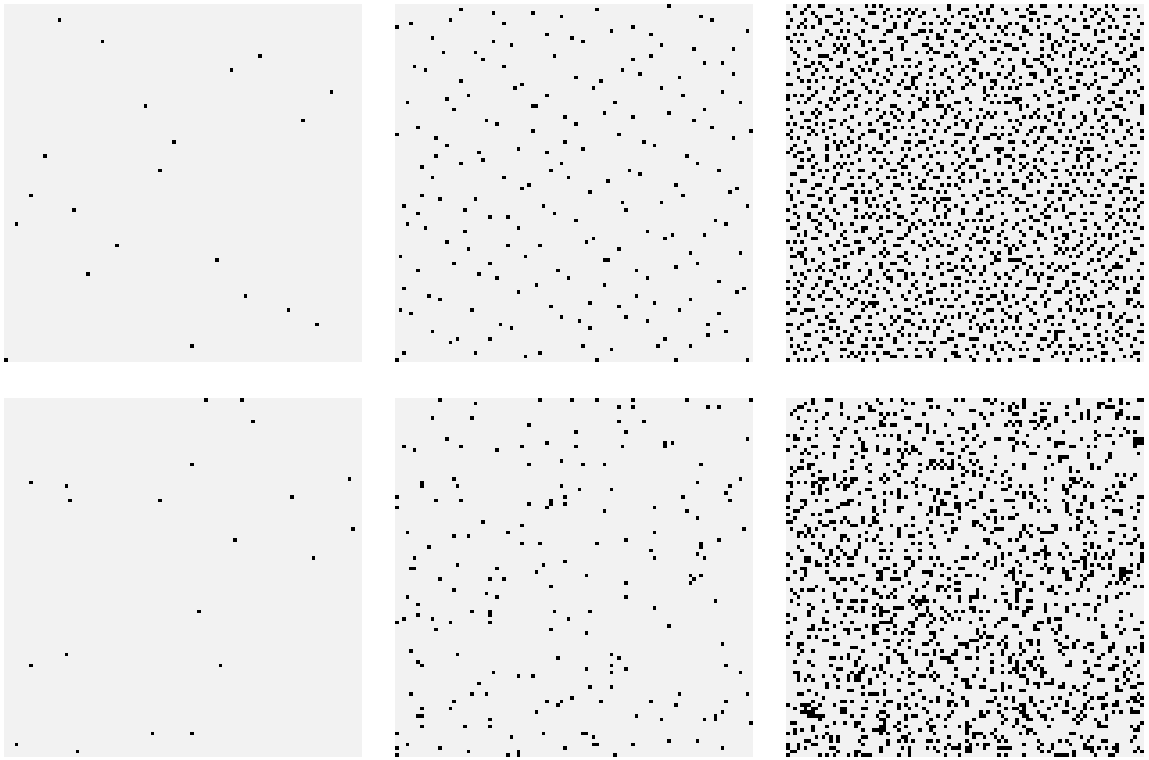
Quasi-Random Sampling

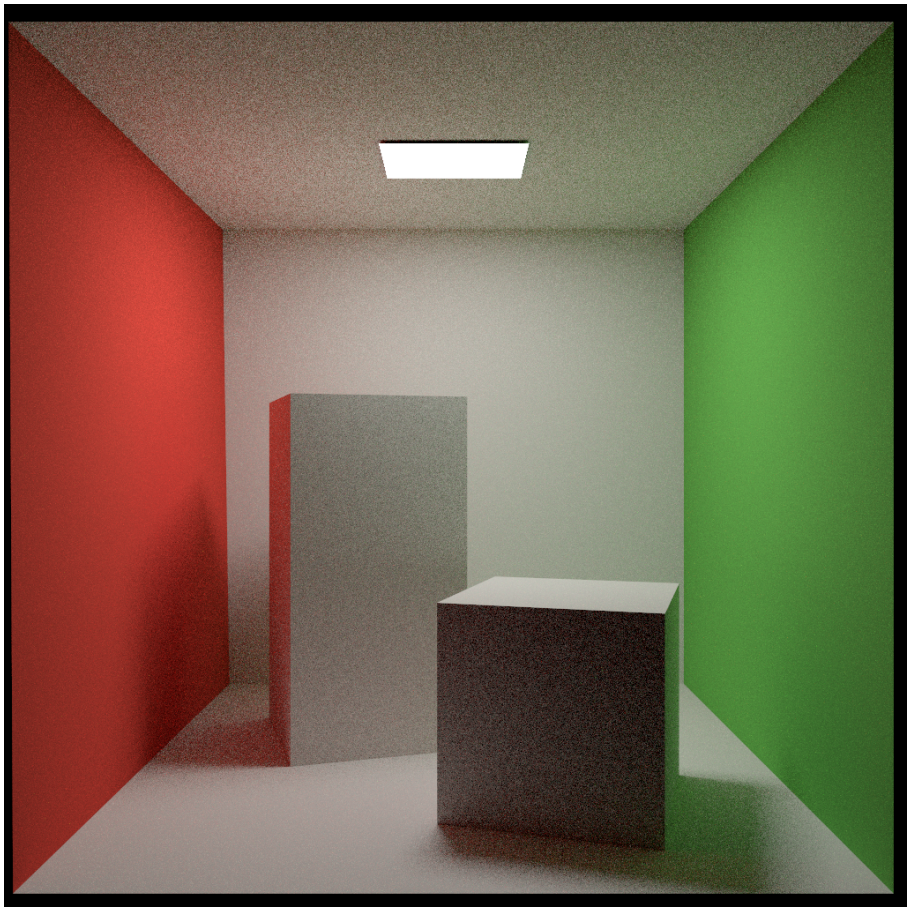
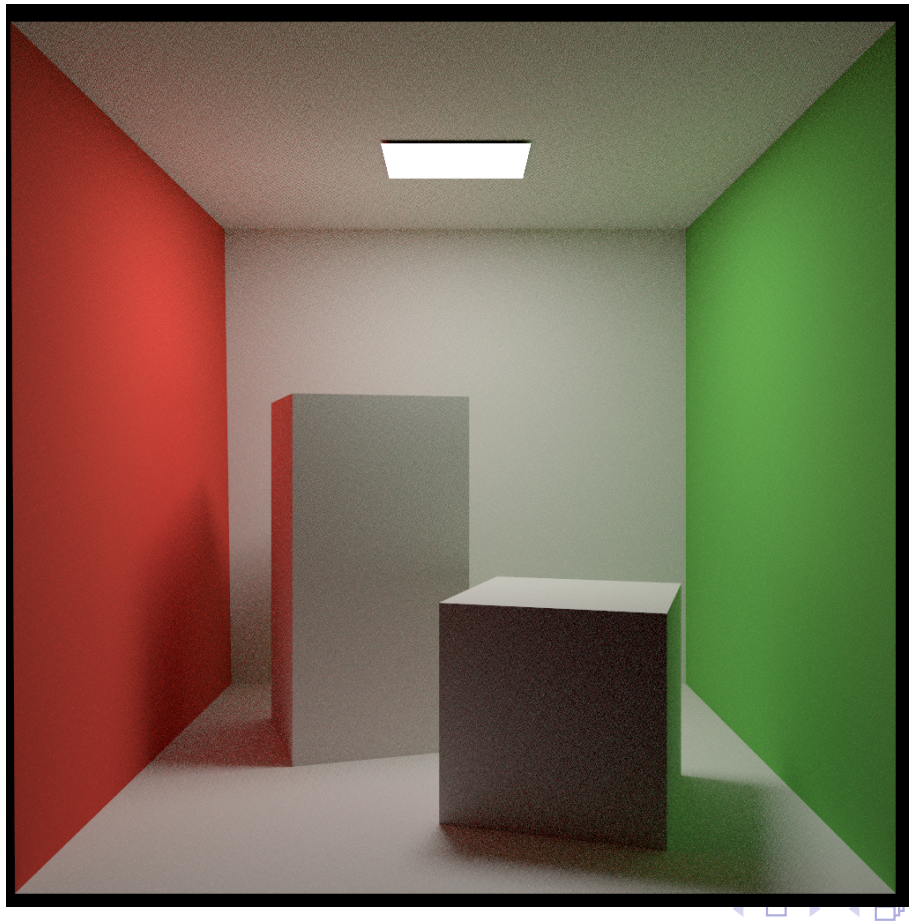
- Note that the benefit will diminish in high dimensions: “curse of dimensionality”
  - To sample the interval $[0, 1]$, you can place 100 samples so that their distances do not exceed 0.01.
  - To cover $[0, 1]^2$ with the same distances, you need $100 \times 100 = 10000$ samples.
  - For $[0, 1]^3$: 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

Monte-Carlo Path Tracing

Quasi-Random Sampling

- Low-discrepancy sequence generators relevant in rendering:
  - Halton Sequence
  - Sobol Sequence
- For more information: Sec. 7 in “Physically Based Rendering”
- Example for Halton sequence vs pseudo-random numbers in 2D:



Monte-Carlo Path Tracing

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