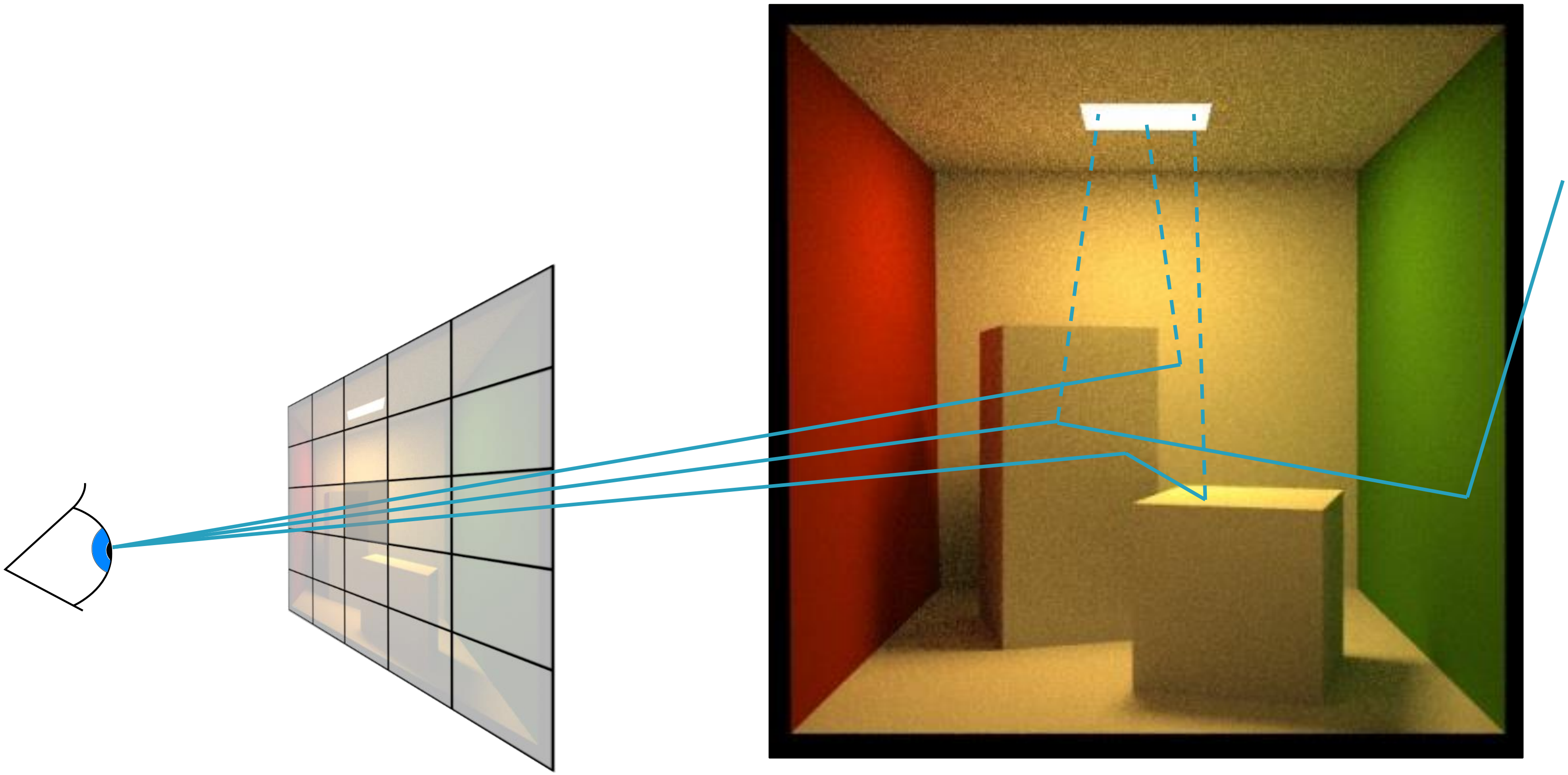
Advanced 3D graphics for movies and games (NPGR010)

– Low-discrepancy sequences and quasi-Monte Carlo methods

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

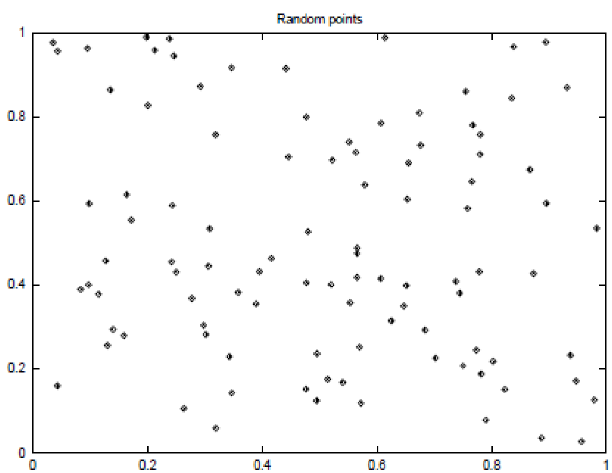
Slides by prof. Jaroslav Křivánek, extended by Jiří Vorba

Path tracing

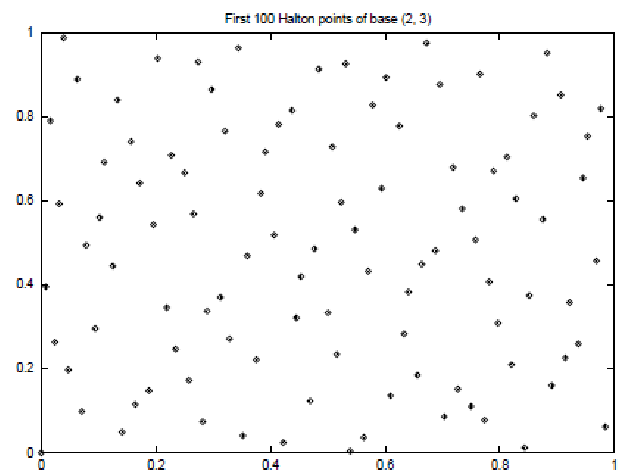


Quasi-Monte Carlo

- Goal: Use point sequences that cover the integration domain as uniformly as possible, while keeping a ‘randomized’ look of the point set



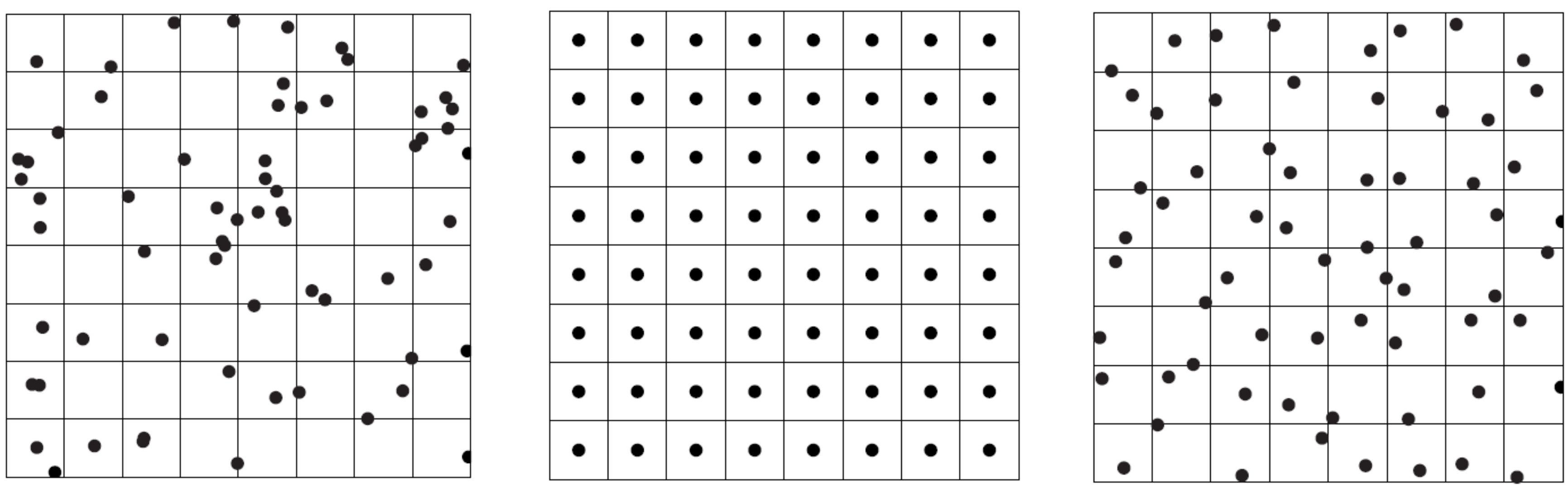
High Discrepancy
(clusters of points)



Low Discrepancy
(more uniform)

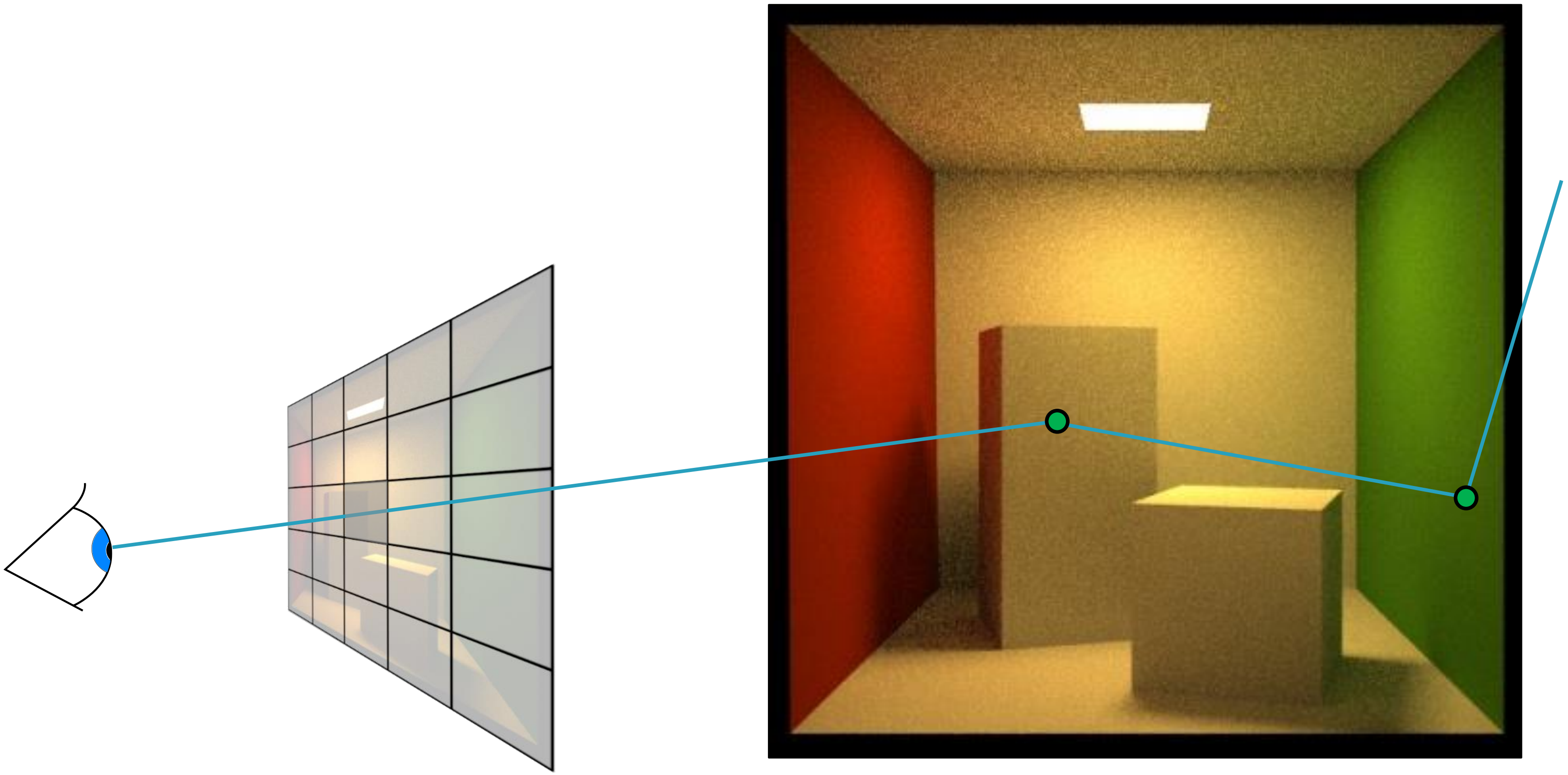
Recall: Stratified samples

- Samples can still form clumps at borders
- Suffers from course of dimensionality



Random   One sample per stratum   Jittered stratified sampling

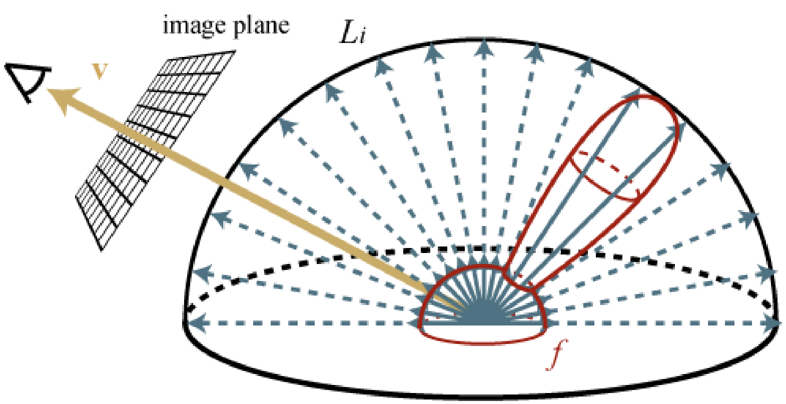
Path tracing



```
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;
}
```

BSDF sampling

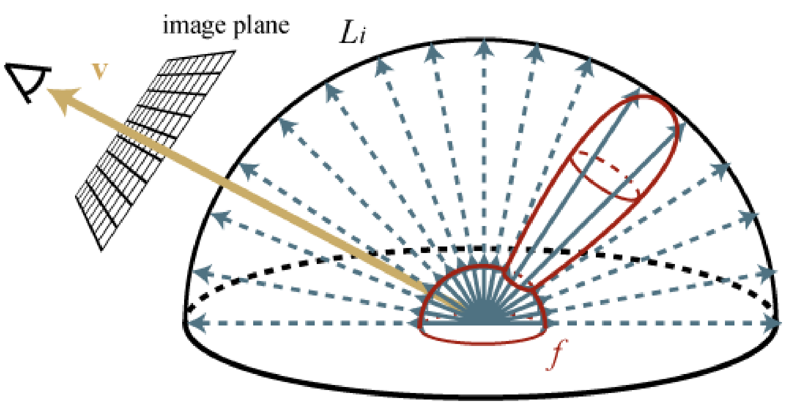
- We need 2 random samples to cover hemisphere



$$L_{\text{refl}}(x, \omega_{\text{out}}) = \int_{H(\mathbf{x})} L_{\text{in}}(x, \omega_{\text{in}}) \cdot f_r(x, \omega_{\text{in}} \to \omega_{\text{out}}) \cdot \cos\theta_{\text{in}} \;\, \mathrm{d}\omega_{\text{in}}$$

BSDF sampling

- We need 2 random samples to cover hemisphere
- + 1 to choose a component

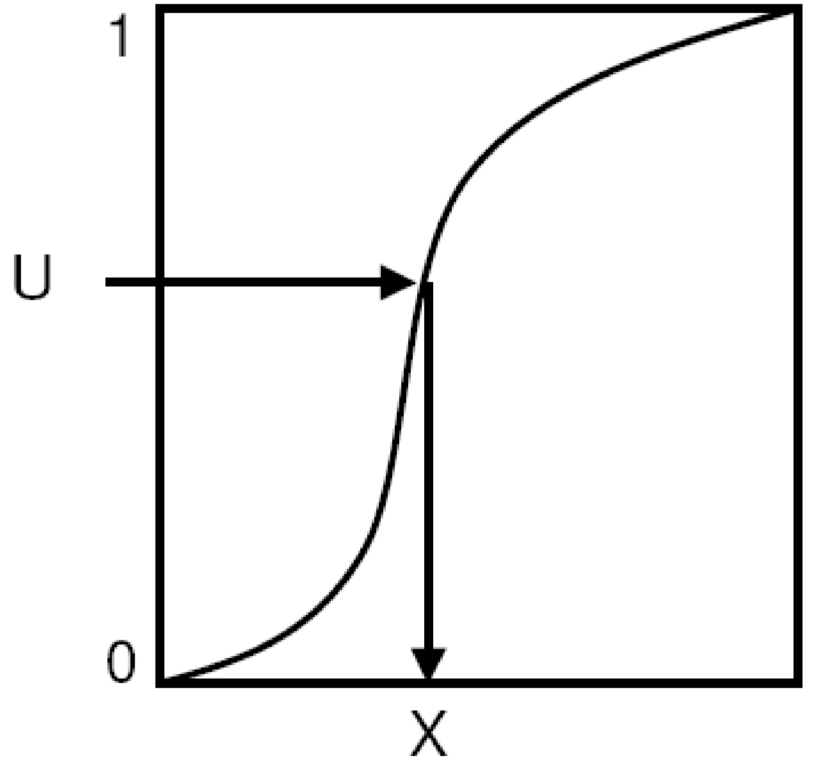


$$f_r^{\text{Phong modif}} = \frac{\rho_d}{\pi} + \frac{n+2}{2\pi} \rho_s \cos^n \theta_r$$

$$L_{\text{refl}}(x, \omega_{\text{out}}) = \int_{H(\mathbf{x})} L_{\text{in}}(x, \omega_{\text{in}}) \cdot f_r(x, \omega_{\text{in}} \to \omega_{\text{out}}) \cdot \cos\theta_{\text{in}} \; \mathrm{d}\omega_{\text{in}}$$

Transformation method – cdf inversion

- U is uniformly distributed in $[0,1]^n$



Transformation of point sets

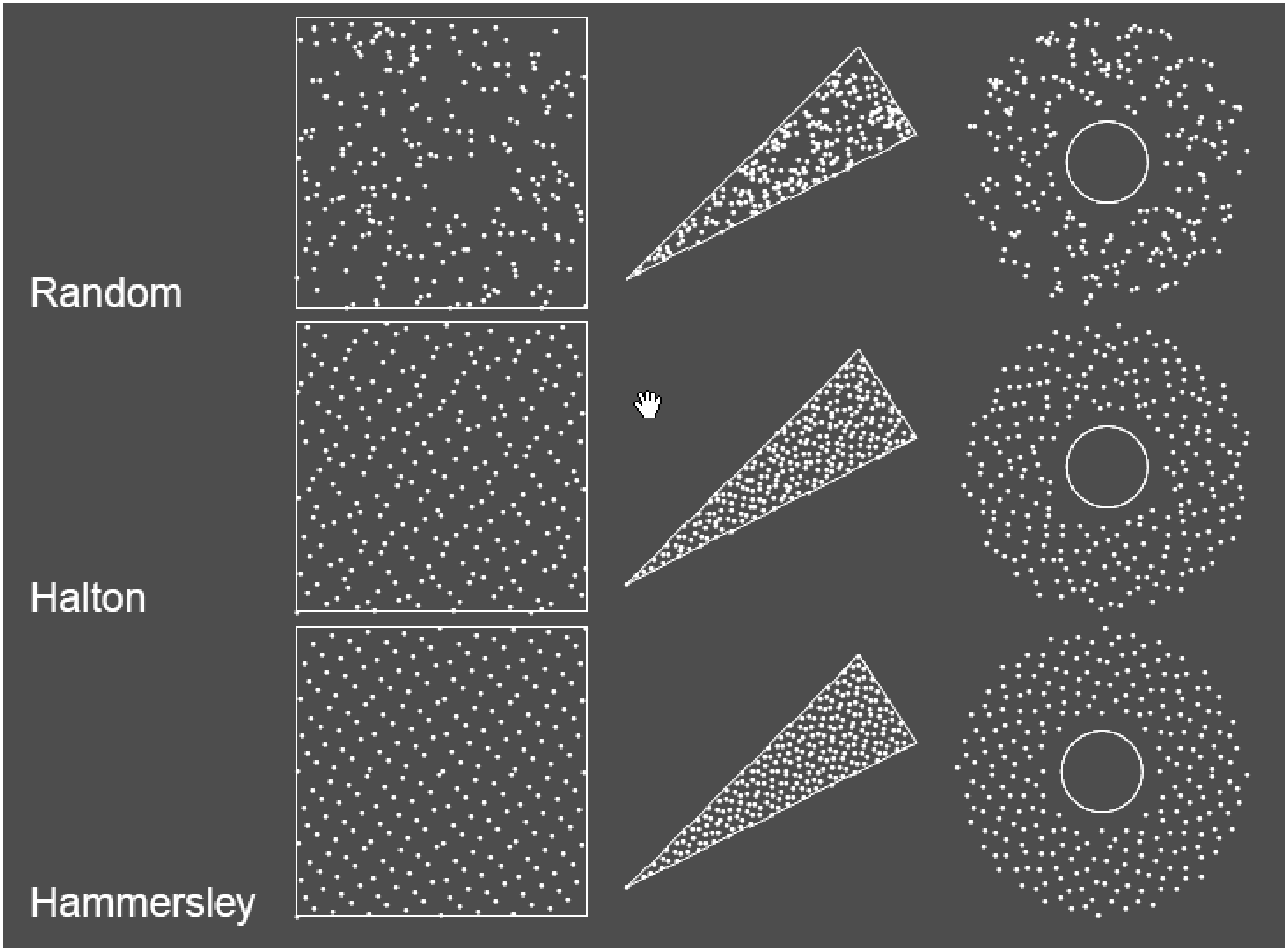
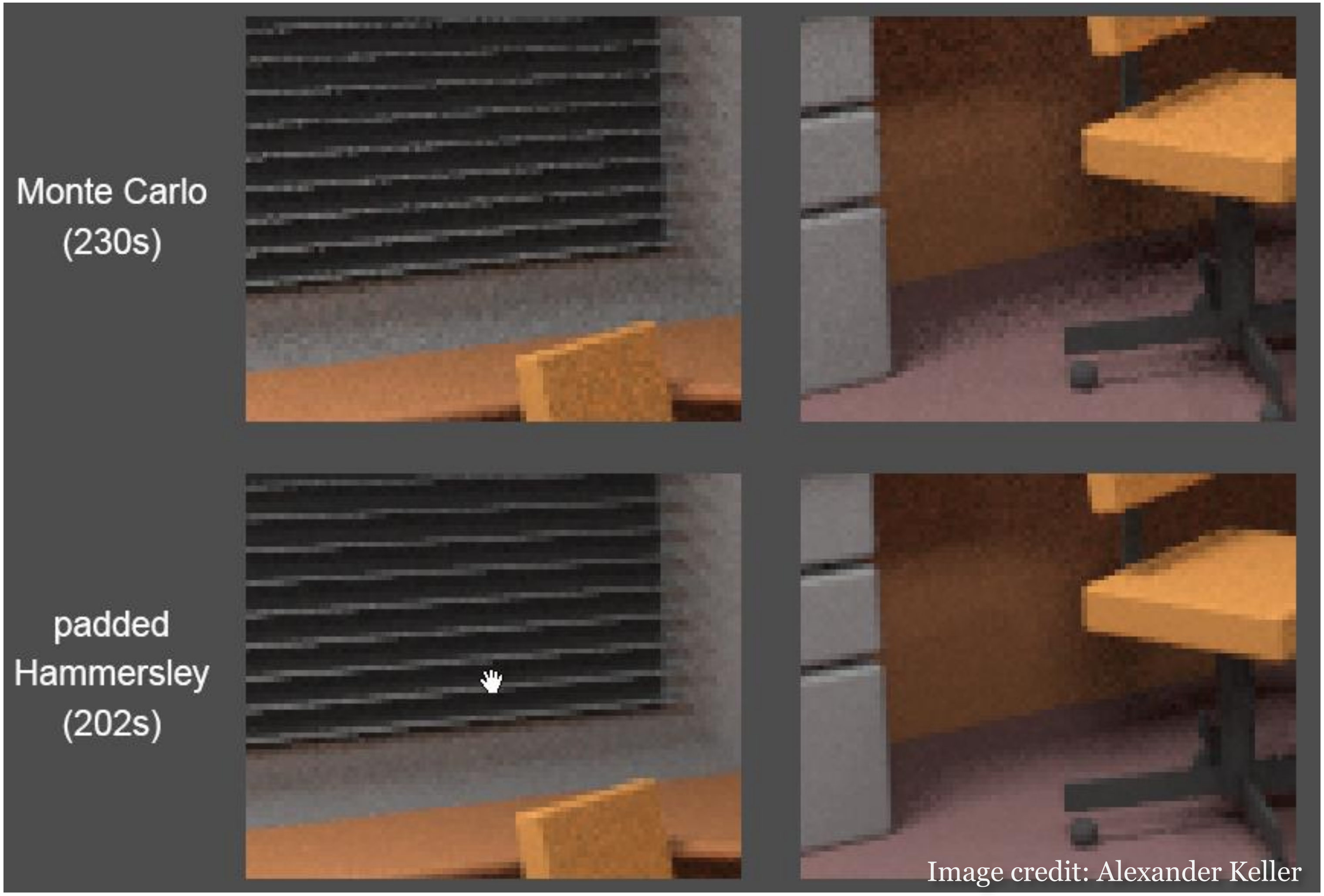


Image credit: Alexander Keller

MC vs. QMC


Monte Carlo (230s)
padded Hammersley (202s)
Image credit: Alexander Keller

Random

- Mersenne twister
- Pseudorandom number generator
- Available in C++11
- 32-bit wide (period)

Large state – 2.5KB

```
#include <random>

const uint32_t seed = 123;
std::mt19937 generator (seed);
std::uniform_real_distribution<float> dis(0.0, 1.0);
float ksi = dis(generator);
```

Random - seeding

Truly random – seed by current time

But we want deterministic renders!

Why?

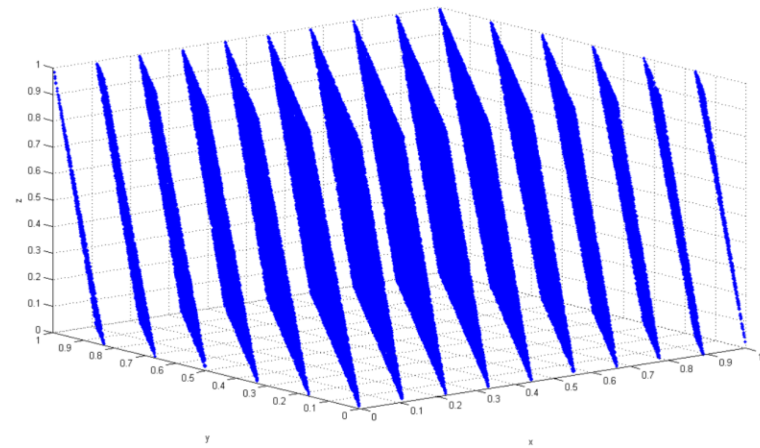
Debugging!
Imagine chasing a source of firefly

Stay away from `srand()` and `rand()`

- Bad statistical properties
- Short period (only 16-bit)

```
#include <stdlib.h>
#include <time.h>

srand(time(NULL));
float ksi = rand() / (float) RAND_MAX;
```

Bad idea!



Source: Wikipedia

- More details
  - https://channel9.msdn.com/Events/GoingNative/2013/rand-Considered-Harmful

Random

- Xorshift (by George Marsaglia)
- Pseudorandom generator
- Fast, tiny state
- 32-bit, (64-bit, 128-bit)
- Xorshift+ (statistically as good as Mersenne twister)

```c
#include <stdint.h>

struct xorshift32_state {
  uint32_t a;
};

/* The state word must be initialized to non-zero */
uint32_t xorshift32(struct xorshift32_state *state)
{
    /* Algorithm "xor" from p. 4 of Marsaglia, "Xorshift RNGs" */
    uint32_t x = state->a;
    x ^= x << 13;
    x ^= x >> 17;
    x ^= x << 5;
    return state->a = x;
}
```

Source: wikipedia

Quasi Monte Carlo (QMC) methods

- Use of strictly deterministic sequences instead of random numbers
  - Also true for pseudo-random numbers
- All formulas as in MC, just the underlying proofs cannot reply on the probability theory (nothing is random)
- Based on **low-discrepancy sequences**

Defining discrepancy

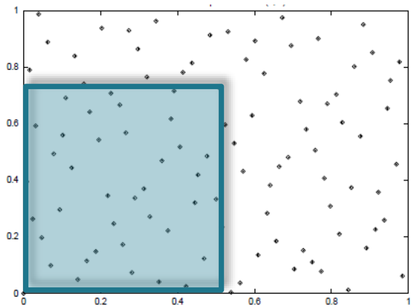
- *s*-dimensional "brick" function:

$$\mathcal{L}(\mathbf{z}) = \begin{cases} 1 \text{ if } 0 \le \mathbf{z}|_1 \le v_1, 0 \le \mathbf{z}|_2 \le v_2, \ldots, 0 \le \mathbf{z}|_s \le v_s \\ 0 \text{ otherwise.} \end{cases}$$

- True volume of the "brick" function:

$$V(A) = \prod_{j=1}^{s} v_j$$

- MC estimate of the volume of the "brick":



$$\frac{1}{N}\sum_{i=1}^{N} f(\mathbf{z}_i) = \frac{m(A)}{N}$$

total number of sample points

number of sample points that actually fell inside the "brick"

Discrepancy

- Discrepancy (of a point sequence) is the maximum possible error of the MC quadrature of the "brick" function over all possible brick shapes:

$$\mathcal{D}^*(\mathbf{z}_1, \mathbf{z}_2, \ldots \mathbf{z}_N) = \sup_A \left| \frac{m(A)}{N} - V(A) \right|.$$

  - serves as a measure of the uniformity of a point set
  - must converge to zero as N -> infty
  - the lower the better (cf. **Koksma-Hlawka Inequality**)

Koksma-Hlawka inequality

- Koksma-Hlawka inequality

„variation“ of *f*

$$\left| \int_{\mathbf{z}\in[0,1]^s} f(\mathbf{z})\, d\mathbf{z} - \frac{1}{N}\sum_{i=1}^{N} f(\mathbf{z}_i)\right| \leq \mathcal{V}_{\mathrm{HK}} \cdot \mathcal{D}^*(\mathbf{z}_1, \mathbf{z}_2, \ldots \mathbf{z}_N)$$

- the KH inequality only applies to *f* with finite variation
- QMC can still be applied even if the variation of f is infinite

Van der Corput Sequence (base 2)

i	binary form of i	radical inverse	H_i
1	1	0.1	0.5
2	10	0.01	0.25
3	11	0.11	0.75
4	100	0.001	0.125
5	101	0.101	0.625
6	110	0.011	0.375
7	111	0.111	0.875

Table credit: Laszlo Szirmay-Kalos

- point placed in the middle of the interval
- then the interval is divided in half
- has low-discrepancy

Van der Corput Sequence (base 2)

i	binary form of i	radical inverse	H_i
1	1	0.1	0.5
2	10	0.01	0.25
3	11	0.11	0.75
4	100	0.001	0.125
5	101	0.101	0.625
6	110	0.011	0.375
7	111	0.111	0.875

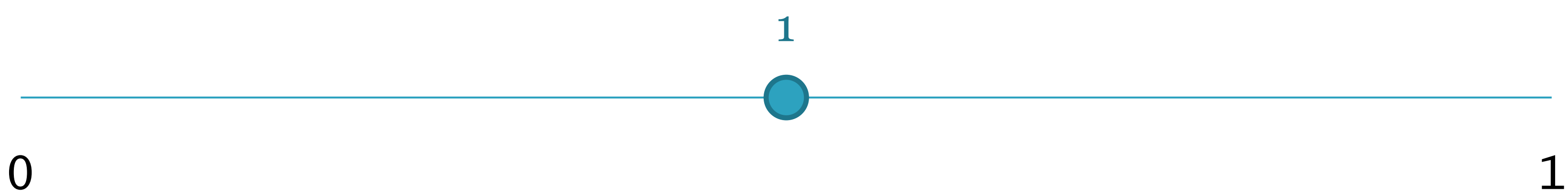
Table credit: Laszlo Szirmay-Kalos

0 1

Van der Corput Sequence (base 2)

i	binary form of i	radical inverse	H_i
1	1	0.1	0.5
2	10	0.01	0.25
3	11	0.11	0.75
4	100	0.001	0.125
5	101	0.101	0.625
6	110	0.011	0.375
7	111	0.111	0.875

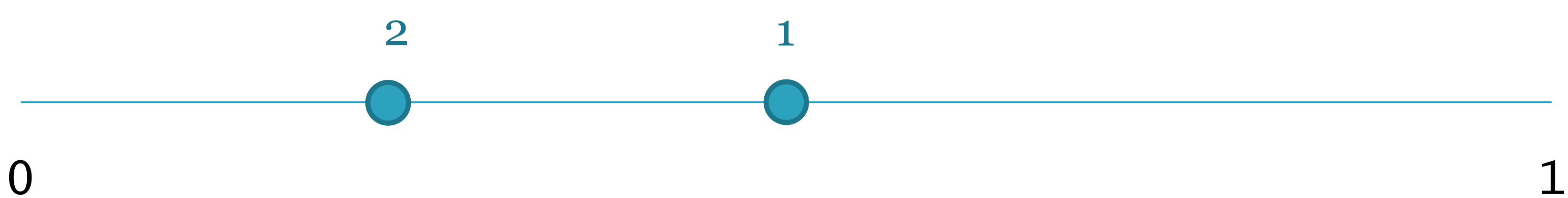
Table credit: Laszlo Szirmay-Kalos



Van der Corput Sequence (base 2)

i	binary form of i	radical inverse	H_i
1	1	0.1	0.5
2	10	0.01	0.25
3	11	0.11	0.75
4	100	0.001	0.125
5	101	0.101	0.625
6	110	0.011	0.375
7	111	0.111	0.875

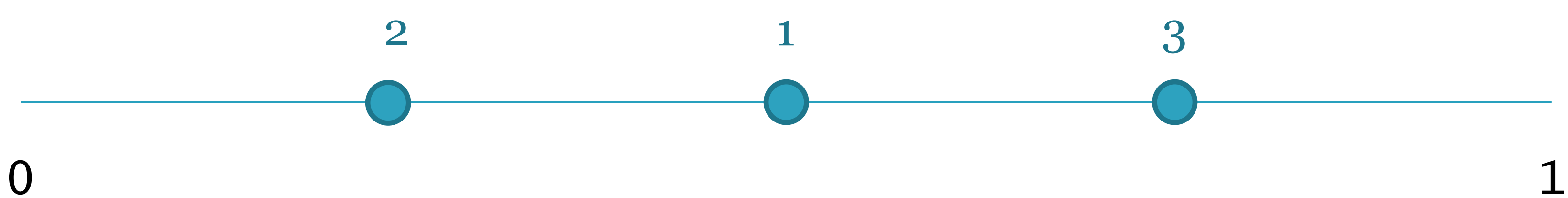
Table credit: Laszlo Szirmay-Kalos



Van der Corput Sequence (base 2)

i	binary form of i	radical inverse	H_i
1	1	0.1	0.5
2	10	0.01	0.25
3	11	0.11	0.75
4	100	0.001	0.125
5	101	0.101	0.625
6	110	0.011	0.375
7	111	0.111	0.875

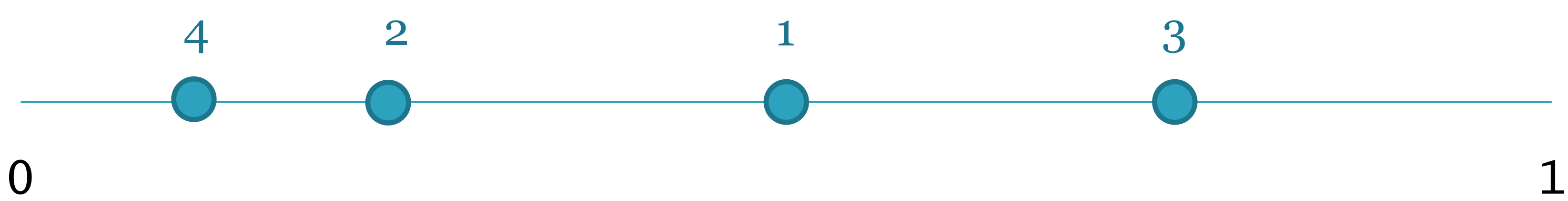
Table credit: Laszlo Szirmay-Kalos



Van der Corput Sequence (base 2)

i	binary form of i	radical inverse	H_i
1	1	0.1	0.5
2	10	0.01	0.25
3	11	0.11	0.75
4	100	0.001	0.125
5	101	0.101	0.625
6	110	0.011	0.375
7	111	0.111	0.875

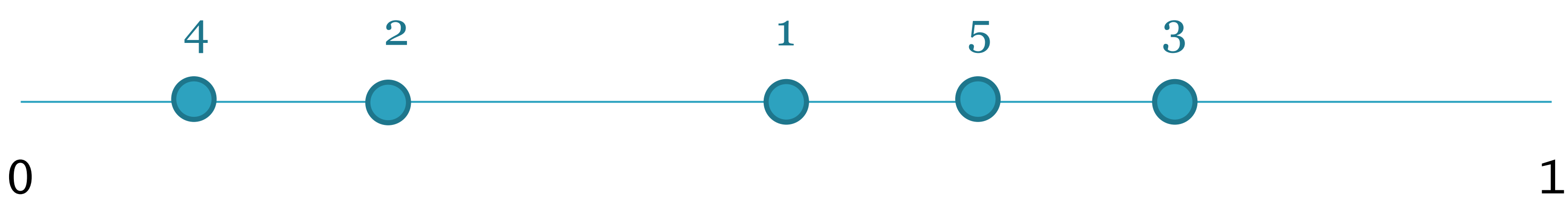
Table credit: Laszlo Szirmay-Kalos



Van der Corput Sequence (base 2)

i	binary form of i	radical inverse	H_i
1	1	0.1	0.5
2	10	0.01	0.25
3	11	0.11	0.75
4	100	0.001	0.125
5	101	0.101	0.625
6	110	0.011	0.375
7	111	0.111	0.875

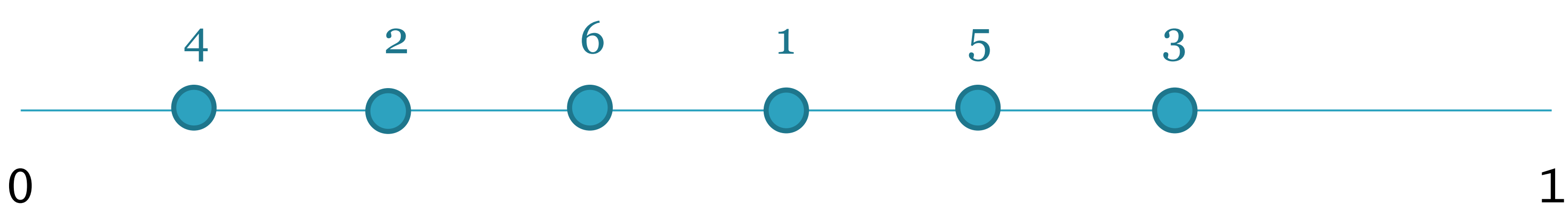
Table credit: Laszlo Szirmay-Kalos



Van der Corput Sequence (base 2)

i	binary form of i	radical inverse	H_i
1	1	0.1	0.5
2	10	0.01	0.25
3	11	0.11	0.75
4	100	0.001	0.125
5	101	0.101	0.625
6	110	0.011	0.375
7	111	0.111	0.875

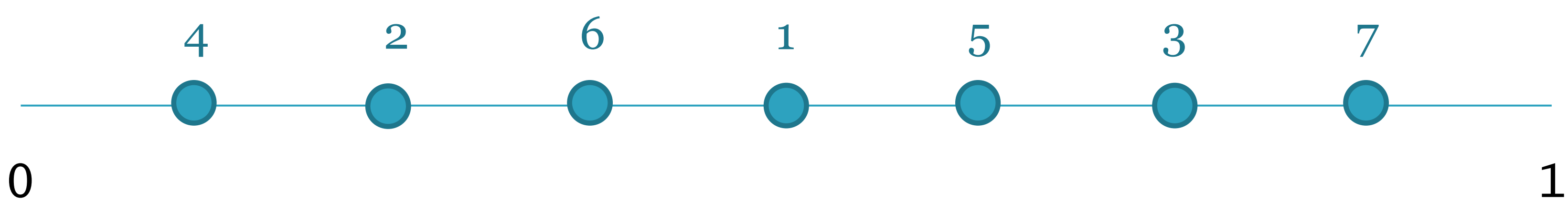
Table credit: Laszlo Szirmay-Kalos



Van der Corput Sequence (base 2)

i	binary form of i	radical inverse	H_i
1	1	0.1	0.5
2	10	0.01	0.25
3	11	0.11	0.75
4	100	0.001	0.125
5	101	0.101	0.625
6	110	0.011	0.375
7	111	0.111	0.875

Table credit: Laszlo Szirmay-Kalos



Van der Corput Sequence

- b … **Base**
- radical inverse

$$i = \sum_{j=0}^{\infty} a_j(i) b^j \quad \mapsto \quad \Phi_b(i) := \sum_{j=0}^{\infty} a_j(i) b^{-j-1}$$

Van der Corput Sequence

- b … **Base**

- radical inverse

$$i = \sum_{j=0}^{\infty} a_j(i) b^j \quad \mapsto \quad \Phi_b(i) := \sum_{j=0}^{\infty} a_j(i) b^{-j-1}$$

- Example:

$i = 123$

$b = 10$

$$123 = 3 * 10^0 + 2 * 10^1 + 1 * 10^2$$

$$\rightarrow$$

$$\Phi_{10}(123) = \frac{3}{10} + \frac{2}{10^2} + \frac{1}{10^3} = 0.321$$

Van der Corput Sequence (base *b*)

```
double radicalInverse(const int base, int i)
{
        double digit, radical;
        digit = radical = 1.0 / (double)base;
        double inverse = 0.0;
        while(i)
        {
                inverse += digit * (double)(i % base);
                digit *= radical;
                i /= base;
        }
        return inverse;
}
```

Van der Corput Sequence (base *b*)

- Discrepancy of sequence P

$$D_N^*(P) = O\left(\frac{\log N}{N}\right)$$

Sequences in higher dimension

Halton sequence $x_i := (\Phi_{b_1}(i), \ldots, \Phi_{b_s}(i))$ where b_i is the i-th prime number

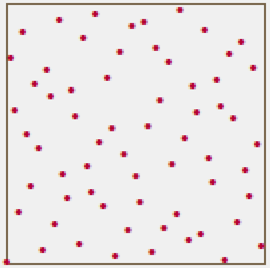
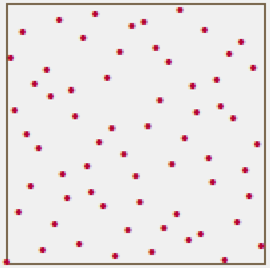


Image credit: Alexander Keller

Sequences in higher dimension

Halton sequence $x_i := \left(\Phi_{b_1}(i), \ldots, \Phi_{b_s}(i)\right)$ where b_i is the i-th prime number



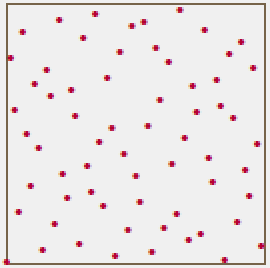
- Discrepancy

$$D_N^*(x_i) = O\left(\frac{(\log N)^s}{N}\right)$$

Image credit: Alexander Keller

Sequences in higher dimension

Halton sequence $x_i := (\Phi_{b_1}(i), \ldots, \Phi_{b_s}(i))$ where b_i is the i-th prime number



Hammersley point set $x_i := \left(\frac{i}{n}, \Phi_{b_1}(i), \ldots, \Phi_{b_{s-1}}(i)\right)$

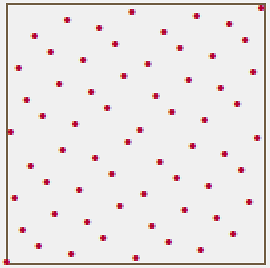


Image credit: Alexander Keller

Sequences vs sets

- Set
  - Number of samples need to be known a-priory
  - Hammersley, stratified samples
  - Usually needs to be recomputed if we change N

- Sequence
  - We don't need to know number of samples beforehand
  - Points added without recomputing of previous points
  - Halton

Progressive sequences/sets

- Any prefix preserves low-discrepancy
- Suitable for progressive rendering

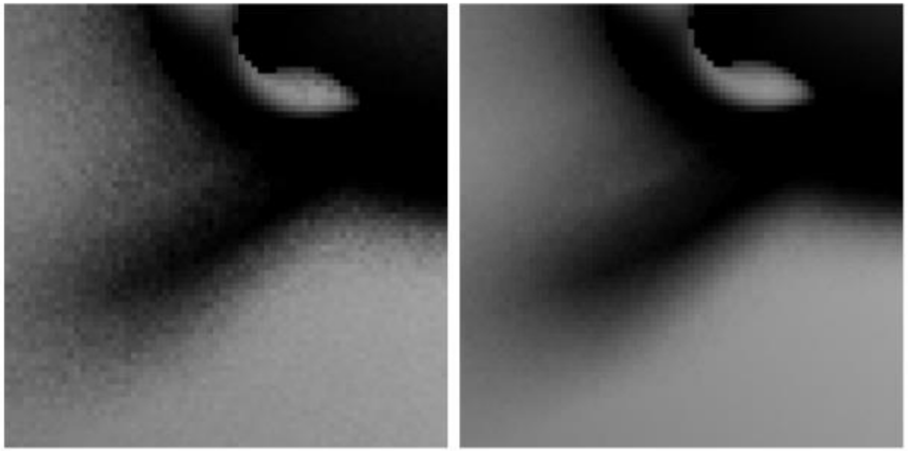
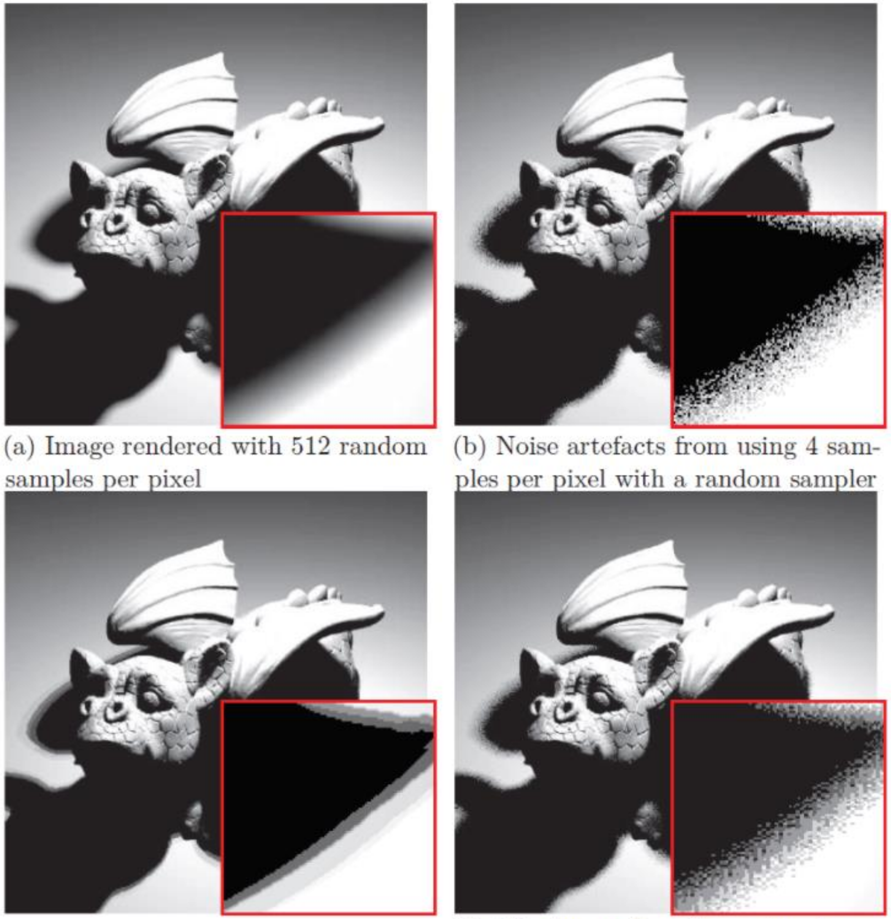


Figure 2: *Penumbra region with 100 samples per pixel. Left: non-progressive sample set. Right: progressive sample sequence.*

Source: Christensen[2018]

Quasi-Monte Carlo (QMC) Methods

- Disadvantages of QMC:
  - Regular patterns can appear in the images (instead of the more acceptable noise in purely random MC)
  - Random scrambling can be used to suppress it



(a) Image rendered with 512 random samples per pixel

(b) Noise artefacts from using 4 samples per pixel with a random sampler

(c) Aliasing artefacts from using the same 4 samples for each pixel

(d) Aliasing artefacts from using 4 samples per pixel with a structured sampler

Source: http://extremelearning.com.au/

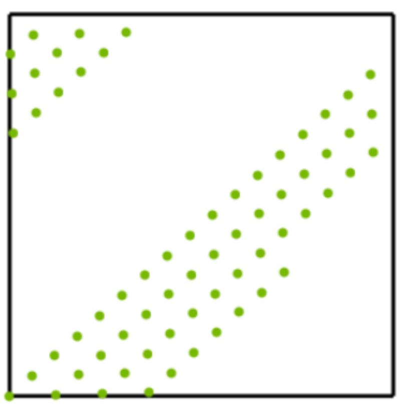
Scrambling

- Low-dimensional projections show visible patterns
- Mitigated by scrambling

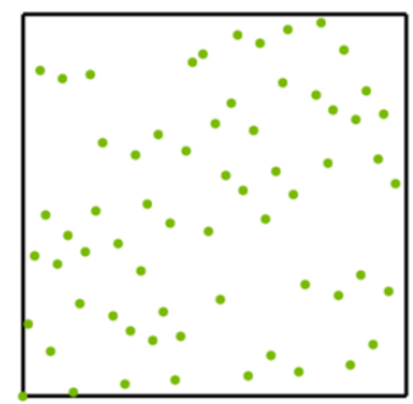
$(\Phi_{17}(i), \Phi_{19}(i)), i = 0 \dots 63$

Scrambling - permutations

- Low-dimensional projections show visible patterns
- Mitigated by scrambling



$$\rightarrow \quad \begin{aligned} \sigma_2 &= (0,1) \\ \sigma_3 &= (0,1,2) \\ \sigma_4 &= (0,2,1,3) \\ \sigma_5 &= (0,3,2,1,4) \\ \sigma_6 &= (0,2,4,1,3,5) \\ &\vdots \end{aligned} \quad \rightarrow$$

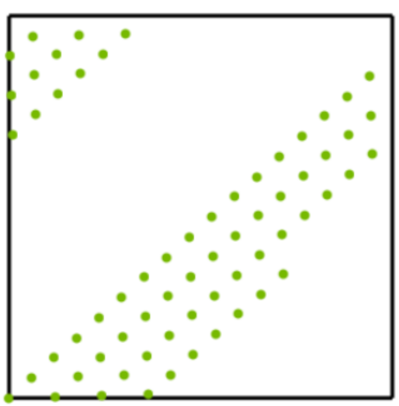


$(\Phi_{17}(i), \Phi_{19}(i)), i = 0 \dots 63$

Scrambling - permutations

- Apply the same permutation at every digit

$$\Phi_{\sigma_b}(i) = \sum_{j=0}^{\infty} \sigma_b\left(a_j(i)\right) b^{-j-1}$$

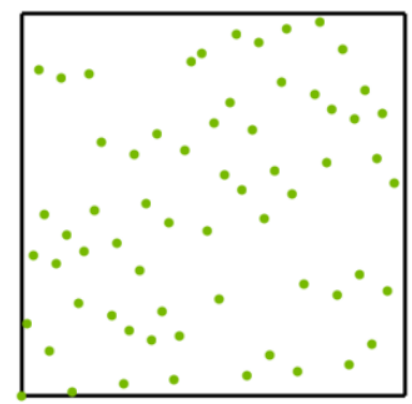


$(\Phi_{17}(i), \Phi_{19}(i)), i = 0 \ldots 63$

$\rightarrow$

$$\begin{aligned}
\sigma_2 &= (0,1) \\
\sigma_3 &= (0,1,2) \\
\sigma_4 &= (0,2,1,3) \\
\sigma_5 &= (0,3,2,1,4) \\
\sigma_6 &= (0,2,4,1,3,5) \\
&\vdots
\end{aligned}$$

$\rightarrow$

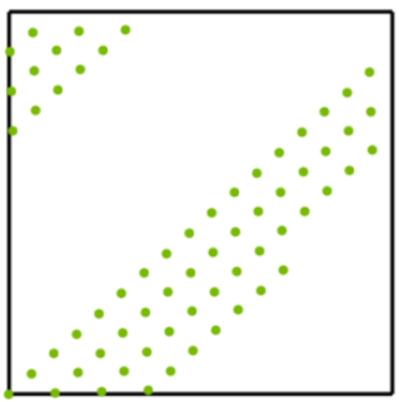


$(\Phi_{\sigma_{17}}(i), \Phi_{\sigma_{19}}(i)), i = 0 \ldots 63$

Scrambling – permutations by Faure

- In general, permutations for each base can be arbitrary
- Deterministic perms. by Faure

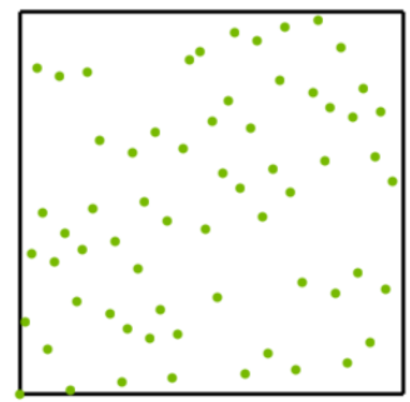
- When b is even: Take $2\sigma_{\frac{b}{2}}$ and append $2\sigma_{\frac{b}{2}} + 1$



$\rightarrow$

$$\begin{aligned}
\sigma_2 &= (0,1) \\
\sigma_3 &= (0,1,2) \\
\sigma_4 &= (0,2,1,3) \\
\sigma_5 &= (0,3,2,1,4) \\
\sigma_6 &= (0,2,4,1,3,5) \\
&\vdots
\end{aligned}$$

$\rightarrow$



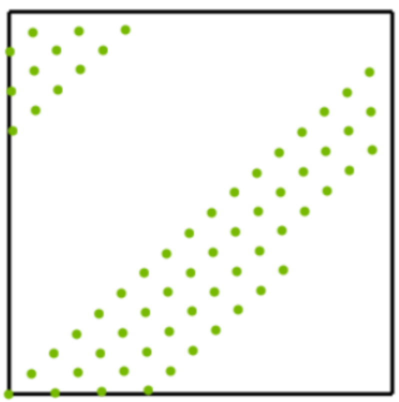
$(\Phi_{17}(i), \Phi_{19}(i)), i = 0 \dots 63$

$(\Phi_{\sigma_{17}}(i), \Phi_{\sigma_{19}}(i)), i = 0 \dots 63$

Scrambling – permutations by Faure

- In general, permutations for each base can be arbitrary
- Deterministic perms. by Faure

- When b is odd: Take σ_{b-1}, increment each value $\geq \frac{b-1}{2}$, insert $\frac{b-1}{2}$ in the middle

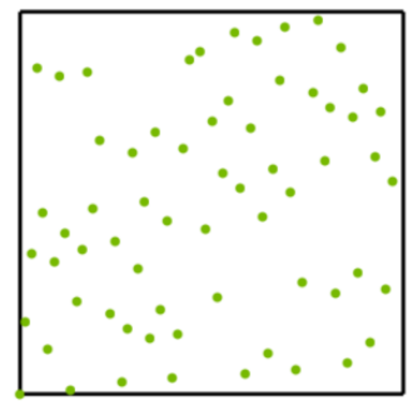


$(\Phi_{17}(i), \Phi_{19}(i)), i = 0 \dots 63$

$\rightarrow$

$$\begin{aligned}
\sigma_2 &= (0,1) \\
\sigma_3 &= (0,1,2) \\
\sigma_4 &= (0,2,1,3) \\
\sigma_5 &= (0,3,2,1,4) \\
\sigma_6 &= (0,2,4,1,3,5) \\
&\vdots
\end{aligned}$$

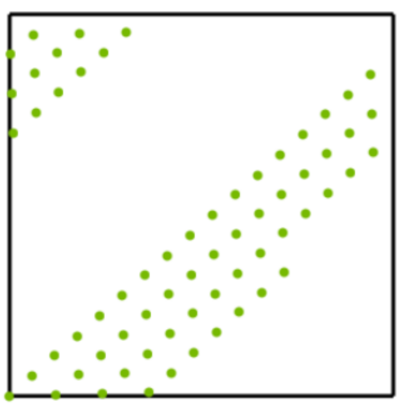
$\rightarrow$



$(\Phi_{\sigma_{17}}(i), \Phi_{\sigma_{19}}(i)), i = 0 \dots 63$

Scrambling – permutations by Faure

- Faure permutations can be implemented efficiently without branching

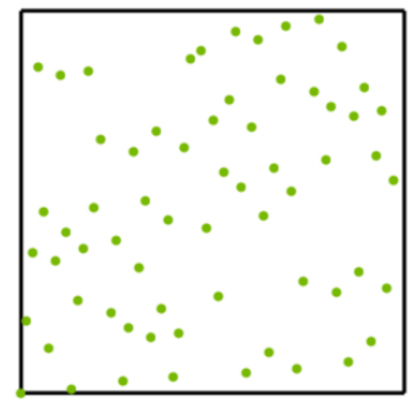


$(\Phi_{17}(i), \Phi_{19}(i)), i = 0 \ldots 63$

$$\rightarrow$$

$$\begin{aligned} \sigma_2 &= (0,1) \\ \sigma_3 &= (0,1,2) \\ \sigma_4 &= (0,2,1,3) \\ \sigma_5 &= (0,3,2,1,4) \\ \sigma_6 &= (0,2,4,1,3,5) \\ &\vdots \end{aligned}$$

$$\rightarrow$$



$(\Phi_{\sigma_{17}}(i), \Phi_{\sigma_{19}}(i)), i = 0 \ldots 63$

Use in path tracing

- **Objective**: Generated paths should cover the entire high-dimensional path space uniformly

- **Approach**:
  - Paths are interpreted as "points" in a high-dimensional path space

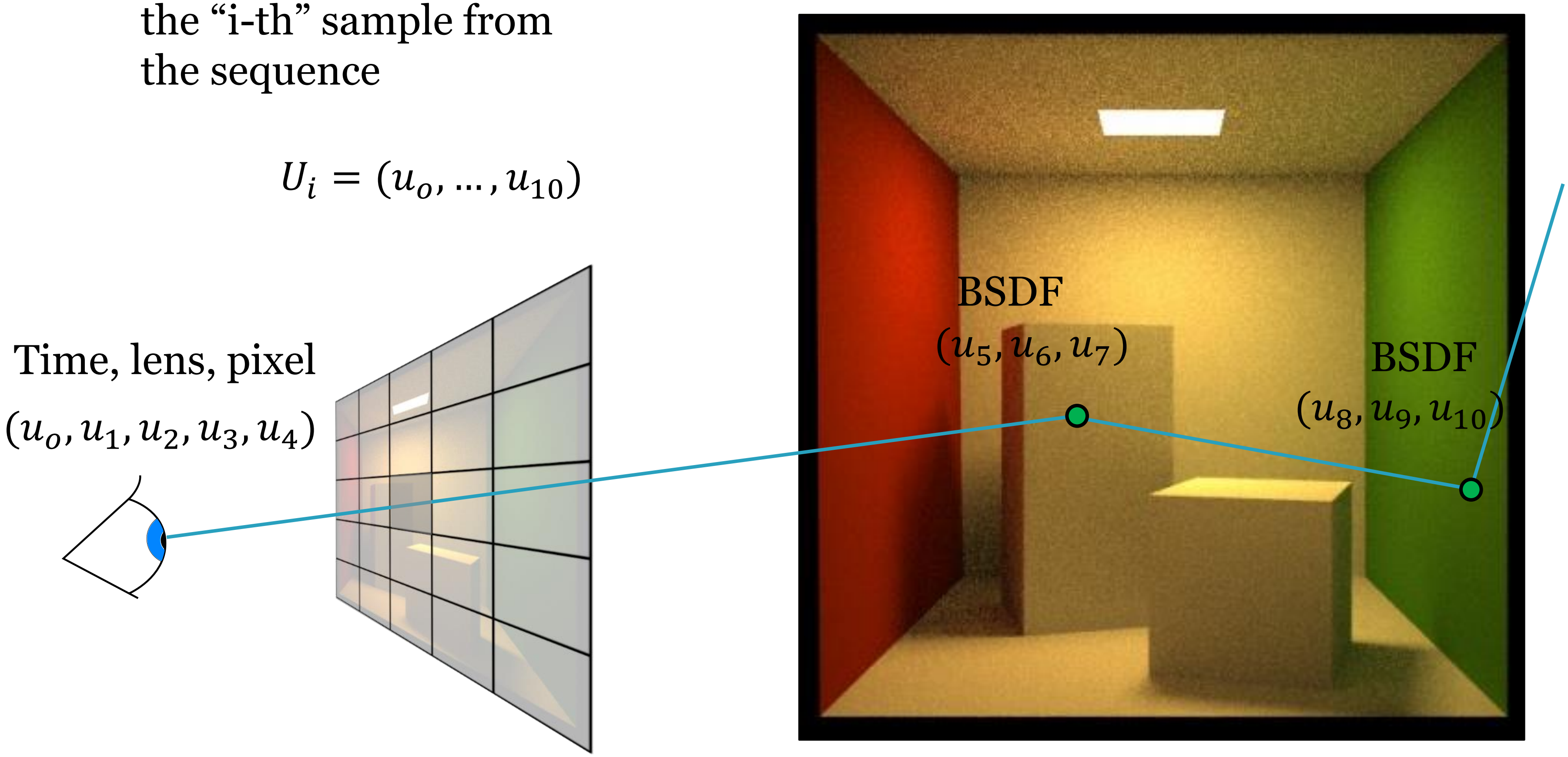
  - Each path is defined by a long vector of "random numbers"
    - **Subsequent random events** along a single path use **subsequent components** of the **same** vector

  - Only when tracing the next path, we switch to a brand new "random vector" (e.g. next vector from a Halton sequence)

Path tracing

- Path "i" sampled based on the "i-th" sample from the sequence

$$U_i = (u_o, \dots, u_{10})$$



Progressive rendering with Halton

- Option 1:
  - Halton sequence per pixel, use different scrambling
  - Loose low-discrepancy properties in the image plane
  - In fact results into stratified sampling in the image plane
- Option 2:
  - Use nearest power of two to your resolution
  - It is possible to compute index of n-th sample in the Halton sequence given the pixels coordinates
  - Skip samples falling outside your actual pixels
  - Details: PBRT, 3rd edition – Chapter 7.4.2.

Further study material

- (t,s)-sequences
  - (0,2)- sequence: Sobol sequence generator matrices
- Rank-1 lattice sequences
- Multi-jittered samples
- References
  - https://sites.google.com/view/myfavoritesamples
  - Christiane Lemieux: Monte Carlo and Quasi-Monte Carlo Sampling [2008]