Improved sampling and quasi-Monte Carlo
Course announcements

• Programming assignment 2 posted, due Friday 2/25 at 23:59.
  - How many of you have looked at/started/finished it?
  - Any questions?

• Take-home quiz 3 posted, due tonight.

• Take-home quiz 4 will be posted tonight, will be due next Tuesday.

• Propose topics for second reading on Friday 2/18, 4-6 pm.
Neural rendering talk today

- **Speaker:** Christian Theobalt (MPI)
- **Title:** Neural Methods for Reconstruction and Rendering of Real World Scenes
- **Date:** Tuesday, February 15th
- **Time:** Noon - 1 pm ET

**Abstract:** In this presentation, I will talk about some of the recent work we did on new methods for reconstructing computer graphics models of real world scenes from sparse or even monocular video data. These methods are based on bringing together neural network-based and explicit model-based approaches. I will also talk about new neural rendering approaches that combine explicit model-based and neural network based concepts for image formation in new ways. They enable new means to synthesize highly realistic imagery and videos of real work scenes under user control.
Overview of today’s lecture

• Stratified sampling.

• Uncorrelated jitter.

• N-rooks.

• Multi-jittered sampling.

• Poisson disk sampling.

• Discrepancy.

• Quasi-Monte Carlo.

• Low-discrepancy sequences.
Slide credits

Most of these slides were directly adapted from:

• Wojciech Jarosz (Dartmouth).
Strategies for Reducing Variance

\[ \sigma \left[ \langle F^N \rangle \right] = \frac{1}{\sqrt{N}} \sigma [Y] \]

Reduce the variance of \( Y \)

- Importance sampling

Relax assumption of uncorrelated samples
Independent Random Sampling

for (int k = 0; k < num; k++)
{
    samples(k).x = randf();
    samples(k).y = randf();
}

✔ Trivially extends to higher dimensions
✔ Trivially progressive and memory-less
✗ Big gaps
✗ Clumping
Regular Sampling

for (uint i = 0; i < numX; i++)
    for (uint j = 0; j < numY; j++)
    {
        samples(i,j).x = (i + 0.5)/numX;
        samples(i,j).y = (j + 0.5)/numY;
    }

✔ Extends to higher dimensions, but...

✗ Curse of dimensionality

✗ Aliasing
Jittered/Stratified Sampling

for (uint i = 0; i < numX; i++)
    for (uint j = 0; j < numY; j++)
    {
        samples(i,j).x = (i + randf())/numX;
        samples(i,j).y = (j + randf())/numY;
    }

✔ Provably cannot increase variance
✔ Extends to higher dimensions, but...
✗ Curse of dimensionality
✗ Not progressive
Monte Carlo (16 random samples)
Monte Carlo (16 jittered samples)
Stratifying in Higher Dimensions

Stratification requires $O(N^d)$ samples

- e.g. pixel (2D) + lens (2D) + time (1D) = 5D
  - splitting 2 times in 5D = $2^5 = 32$ samples
  - splitting 3 times in 5D = $3^5 = 243$ samples!

Inconvenient for large $d$

- cannot select sample count with fine granularity
“Padding” 2D points (Uncorrelated Jitter)

[Cook 86]

\[
\begin{align*}
&\text{2D} & & \text{2D} \\
&(x_1, y_1) & & (u_1, v_1) \\
&(x_2, y_2) & & (u_2, v_2) \\
&(x_3, y_3) & & (u_3, v_3) \\
&(x_4, y_4) & & (u_4, v_4) \\
&\vdots & & \vdots \\
&(x_1, y_1, u_3, v_3) & & \vdots \\
&(x_2, y_2, u_1, v_1) & & \vdots \\
&(x_3, y_3, u_4, v_4) & & \vdots \\
&(x_4, y_4, u_2, v_2) & & \vdots \\
\end{align*}
\]

Slide after Gurprit Singh
Depth of Field (4D)

Reference

Random Sampling

Uncorrelated Jitter

Image source: PBRTe2 [Pharr & Humphreys 2010]
Uncorrelated Jitter → Latin Hypercube

Like uncorrelated jitter, but using 1D point sets

- for 5D: 5 separate 1D jittered point sets
- combine dimensions in random order
Uncorrelated Jitter ➔ Latin Hypercube

Like uncorrelated jitter, but using 1D point sets

- for 5D: 5 separate 1D jittered point sets

- combine dimensions in random order

Shuffle order
N-Rooks = 2D Latin Hypercube [Shirley 91]

Like uncorrelated jitter, but using 1D point sets

- for 2D: 2 separate 1D jittered point sets

- combine dimensions in random order

```
x
x1  x2  x3  x4
y
y4  y2  y1  y3
```
Latin Hypercube (N-Rooks) Sampling

[Shirley 91]
Latin Hypercube (N-Rooks) Sampling

// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
  for (uint i = 0; i < numS; i++)
    samples(d,i) = (i + randf())/numS;

// shuffle each dimension independently
for (uint d = 0; d < numDimensions; d++)
  shuffle(samples(d,:));
Latin Hypercube (N-Rooks) Sampling

// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

// shuffle each dimension independently
for (uint d = 0; d < numDimensions; d++)
    shuffle(samples(d,:));
// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

// shuffle each dimension independently
for (uint d = 0; d < numDimensions; d++)
    shuffle(samples(d,:));

Latin Hypercube (N-Rooks) Sampling

Shuffle rows
Latin Hypercube (N-Rooks) Sampling

// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

// shuffle each dimension independently
for (uint d = 0; d < numDimensions; d++)
    shuffle(samples(d,:));

Shuffle columns
Latin Hypercube (N-Rooks) Sampling

// initialize the diagonal
for (uint d = 0; d < numDimensions; d++)
    for (uint i = 0; i < numS; i++)
        samples(d,i) = (i + randf())/numS;

// shuffle each dimension independently
for (uint d = 0; d < numDimensions; d++)
    shuffle(samples(d,:));
Latin Hypercube (N-Rooks) Sampling
Latin Hypercube (N-Rooks) Sampling
Latin Hypercube (N-Rooks) Sampling

Unevenly distributed in n-dimensions

Evenly distributed in each individual dimension
Multi-Jittered Sampling


– combine N-Rooks and Jittered stratification constraints
Multi-Jittered Sampling
Multi-Jittered Sampling

// initialize
float cellSize = 1.0 / (resX*resY);
for (uint i = 0; i < resX; i++)
    for (uint j = 0; j < resY; j++)
    {
        samples(i,j).x = i/resX + (j+randf()) / (resX*resY);
        samples(i,j).y = j/resY + (i+randf()) / (resX*resY);
    }

// shuffle x coordinates within each column of cells
for (uint i = 0; i < resX; i++)
    for (uint j = resY-1; j >= 1; j--)
        swap(samples(i, j).x, samples(i, randi(0, j)).x);

// shuffle y coordinates within each row of cells
for (unsigned j = 0; j < resY; j++)
    for (unsigned i = resX-1; i >= 1; i--)
        swap(samples(i, j).y, samples(randi(0, i), j).y);
Multi-Jittered Sampling

Shuffle x-coords Initialize
Multi-Jittered Sampling

Shuffle x-coords
Multi-Jittered Sampling

Shuffle x-coords
Multi-Jittered Sampling

Shuffle x-coords
Multi-Jittered Sampling

Shuffle x-coords
Shuffle y-coords
Multi-Jittered Sampling

Shuffle y-coords
Multi-Jittered Sampling

Shuffle y-coords
Multi-Jittered Sampling

Shuffle y-coords
Multi-Jittered Sampling

Shuffle y-coords
Multi-Jittered Sampling (Projections)
Multi-Jittered Sampling (Projections)
Multi-Jittered Sampling (Projections)
Multi-Jittered Sampling (Projections)

Evenly distributed in 2D!

Evenly distributed in each individual dimension
Multi-Jittered Sampling (Sudoku)

[Boulos et al. 2006]
Poisson-Disk/Blue-Noise Sampling

Enforce a minimum distance between points

Poisson-Disk Sampling:


Random Dart Throwing
Random Dart Throwing
Random Dart Throwing
Stratified Sampling
“Best Candidate” Dart Throwing
Blue-Noise Sampling (Relaxation-based)

1. Initialize sample positions (e.g. random)
2. Use an iterative relaxation to move samples away from each other.
Discrepancy

Previous stratified approaches try to minimize “clumping”

“Discrepancy” is another possible formal definition of clumping: $D^*(x_1,\ldots,x_n)$

- for every possible subregion compute the maximum absolute difference between:
  
  - fraction of points in the subregion
  - volume of containing subregion
Discrepancy
Discrepancy
Discrepancy
Discrepancy
Discrepancy
Koksma-Hlawka inequality

\[ \left| \frac{1}{n} \sum_{i=1}^{n} f(x_i) - \int f(u) \, du \right| \leq V(f)D^*(x_1, \ldots, x_n) \]
Low-Discrepancy Sampling

*Deterministic* sets of points specially crafted to be evenly distributed (have low discrepancy).

Entire field of study called Quasi-Monte Carlo (QMC)
The Radical Inverse

A positive integer value $n$ can be expressed in a base $b$ with a sequence of digits $d_m...d_2d_1$

The radical inverse function $\Phi_b$ in base $b$ converts a nonnegative integer $n$ to a floating-point value in $[0, 1)$ by reflecting these digits about the decimal point:

$$\Phi_b(n) = 0.d_1d_2...d_m$$

Subsequent points “fall into biggest holes”
The Van der Corput Sequence

Radical Inverse $\Phi_b$ in base 2

Subsequent points “fall into biggest holes”

<table>
<thead>
<tr>
<th>$k$</th>
<th>Base 2</th>
<th>$\Phi_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>.1 = 1/2</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>.01 = 1/4</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>.11 = 3/4</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>.001 = 1/8</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>.101 = 5/8</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>.011 = 3/8</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>.111 = 7/8</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
The Radical Inverse

float radicalInverse(int n, int base, float inv) {
    float v = 0.0f;
    for (float p = inv; n != 0; p *= inv, n /= base)
        v += (n % base) * p;
    return v;
}

float radicalInverse(int n, int base) {
    return radicalInverse(n, base, 1.0f / base);
}

More efficient version available for base 2
The Radical Inverse (Base 2)

```c
float vanDerCorputRIU(uint n)
{
    n = (n << 16) | (n >> 16);
    n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);
    n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);
    n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);
    n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
    return n / float (0x100000000LL);
}
```
Halton and Hammersley Points

**Halton:** Radical inverse with different base for each dimension:

\[ \vec{x}_k = (\Phi_2(k), \Phi_3(k), \Phi_5(k), \ldots, \Phi_{p_n}(k)) \]

- The bases should all be relatively prime.
- Incremental/progressive generation of samples

**Hammersley:** Same as Halton, but first dimension is \( k/N \):

\[ \vec{x}_k = \left( \frac{k}{N}, \Phi_2(k), \Phi_3(k), \Phi_5(k), \ldots, \Phi_{p_n}(k) \right) \]

- Not incremental, need to know sample count, \( N \), in advance
The Hammersley Sequence

1 sample in each “elementary interval”
The Hammersley Sequence

1 sample in each “elementary interval”
The Hammersley Sequence

1 sample in each “elementary interval”
The Hammersley Sequence

1 sample in each “elementary interval”
The Hammersley Sequence

| Sample in each “elementary interval” |
The Hammersley Sequence

1 sample in each “elementary interval”
(0,2)-Sequences

1 sample in each “elementary interval”
(0,2)-Sequences

1 sample in each “elementary interval”
(0,2)-Sequences

1 sample in each “elementary interval”
(0,2)-Sequences

1 sample in each “elementary interval”
(0,2)-Sequences

1 sample in each “elementary interval”
(0,2)-Sequences

1 sample in each “elementary interval”
More info on QMC in Rendering

S. Premoze, A. Keller, and M. Raab. 
*Advanced (Quasi-) Monte Carlo Methods for Image Synthesis.* In SIGGRAPH 2012 courses.
Many more...

Sobol

Faure

Larcher-Pillichshammer

Folded Radical Inverse

(t,s)-sequences & (t,m,s)-nets

Scrambling/randomization

much more...
Challenges

LD sequence identical for multiple runs
- cannot average independent images!
- no “random” seed

Quality decreases in higher dimensions
Randomized/Scrambled Sequences

Random permutations: compute a permutation table for the order of the digits and use it when computing the radical inverse

\[ \Phi_b(n) = 0. \pi(d_1) \pi(d_2) \ldots \pi(d_m) \]

Without scrambling

With scrambling
Randomized/Scrambled Sequences

Random permutations: compute a permutation table for the order of the digits and use it when computing the radical inverse

- Can be done very efficiently for base 2 with XOR operation

See PBRe2 Ch7 for details
Scrambled Radical Inverse (Base 2)

float vanDerCorputRIU(uint n, uint scramble = 0) {
    n = (n << 16) | (n >> 16);
    n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);
    n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);
    n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);
    n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
    n ^= scramble;
    return n / float (0x100000000LL);
}
Monte Carlo (16 random samples)
Monte Carlo (16 stratified samples)
Quasi-Monte Carlo (16 Halton samples)
Scrambled Quasi-Monte Carlo

scrambled Larcher-Pillichshammer sequence
Implementation tips

Using QMC can often lead to unintuitive, difficult-to-debug problems.

- Always code up MC algorithms first, using random numbers, to ensure correctness
- Only after confirming correctness, slowly incorporate QMC into the mix
How do you add this to your renderer?

Lots of details in the book

Read about the Sampler interface

- Basic idea: replace global randf with a Sampler class that produces random (or stratified/quasi-random) numbers

- Also better for multi-threading
How can we predict error from these?
N-Rooks Sampling

Samples

Expected power spectrum

Radial mean
Multi-Jittered Sampling

Samples

Expected power spectrum

Radial mean

Power

Frequency
Jittered Sampling

Samples

Expected power spectrum

Radial mean

[Image showing samples, expected power spectrum, and radial mean graph]

[Graph with x-axis labeled Frequency, y-axis labeled Power, showing a power spectrum with a peak at zero frequency and decreasing power with increasing frequency]
Poisson Disk Sampling

Samples

Expected power spectrum

Radial mean

[Graph showing samples, expected power spectrum, and radial mean]