Sampling and Reconstruction

Digital Image Synthesis
Yung-Yu Chuang
10/25/2005

with slides by Pat Hanrahan, Torsten Möller and Brian Curless

Sampling theory
- Sampling theory: the theory of taking discrete sample values (grid of color pixels) from functions defined over continuous domains (incident radiance defined over the film plane) and then using those samples to reconstruct new functions that are similar to the original (reconstruction).

Sampler: selects sample points on the image plane
Filter: blends multiple samples together

Aliasing
- Approximation errors: jagged edges or flickering in animation

Fourier analysis
- Most functions can be decomposed into a weighted sum of shifted sinusoids.
  \[ F(s) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi sx} \, dx \]
  \[ e^{ix} = \cos x + i \sin x \]
Fourier analysis

**spatial domain**

<table>
<thead>
<tr>
<th>$x$</th>
<th>$X(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\Pi(x)$</td>
</tr>
<tr>
<td></td>
<td>$\text{gauss}(x; \sigma)$</td>
</tr>
<tr>
<td></td>
<td>$\text{gauss}(s; 1/\sigma)$</td>
</tr>
<tr>
<td></td>
<td>$\text{gauss}(x; 1/\sigma)$</td>
</tr>
<tr>
<td></td>
<td>$\text{gauss}(s; \sigma)$</td>
</tr>
</tbody>
</table>

**frequency domain**

| $s$ | $\text{sinc}(s)$ |

The Fourier transform is a linear operation.

$$\mathcal{F}\{f(x)\} = F(s)$$

$$\mathcal{F}\{af(x)\} = aF\{f(x)\}$$

$$\mathcal{F}\{f(x) + g(x)\} = F\{f(x)\} + F\{g(x)\}$$

**1D convolution theorem example**

Convolution in the spatial domain is equivalent to multiplication in the frequency domain.

$$g(x) = f(x) \ast h(x)$$

$$= \int_{-\infty}^{\infty} f(x')h(x-x')dx'$$

Convolution theorem: Convolution in the spatial domain is equivalent to multiplication in the frequency domain.

$$f \ast h \longleftrightarrow \mathcal{F} \cdot \mathcal{H}$$

Symmetric theorem: Convolution in the frequency domain is equivalent to multiplication in the spatial domain.

$$f \cdot h \longleftrightarrow \mathcal{F} \ast \mathcal{H}$$

**Inverse Fourier transform**

$$f(x) = \int_{-\infty}^{\infty} F(s)e^{i2\pi sx} ds$$

$$\mathcal{F}^{-1}\{F(s)\} = f(x)$$
2D convolution theorem example

\[
f(x, y) \ast h(x, y) \rightarrow g(x, y)
\]

\[
F(s_x, s_y) \ast H(s_x, s_y) \rightarrow G(s_x, s_y)
\]

The delta function

- Dirac delta function, zero width, infinite height and unit area

\[
\delta(x)
\]

Sifting and shifting

Sifting:

\[
f(x) \delta(x-a) = f(a) \delta(x-a)
\]

Shifting:

\[
f(x) \ast \delta(x-a) = f(x-a)
\]

Shah/impulse train function

**Spatial domain**

\[
\sum_{n=-\infty}^{\infty} \delta(x - nT)
\]

**Frequency domain**

\[
\sum_{f \rightarrow \infty} \delta(s - fn) \quad s_0 = 1/T
\]
Sampling

The reconstructed function is obtained by interpolating among the samples in some manner.

Reconstruction

In math forms

\[ \tilde{F} = (F(s) \ast \Pi(s)) \times \Pi(s) \]
\[ \tilde{f} = (f(x) \ast \Pi(x)) \ast \text{sinc}(x) \]
\[ \tilde{f}(x) = \sum_{i=-\infty}^{\infty} \text{sinc}(x-i)f(i) \]

Reconstruction filters

The sinc filter, while ideal, has two drawbacks:
- It has large support (slow to compute)
- It introduces ringing in practice

The box filter is bad because its Fourier transform is a sinc filter which includes high frequency contribution from the infinite series of other copies.
Aliasing

- Increase sample spacing in spatial domain
- Decrease sample spacing in frequency domain

Sampling theorem

This result is known as the Sampling Theorem and is due to Claude Shannon who first discovered it in 1949:

A signal can be reconstructed from its samples without loss of information, if the original signal has no frequencies above \( \frac{1}{2} \) the sampling frequency.

For a given **bandlimited** function, the minimum rate at which it must be sampled is the **Nyquist frequency**.

- For band limited function, we can just increase the sampling rate
- However, few of interesting functions in computer graphics are band limited, in particular, functions with discontinuities.
- It is because the discontinuity always falls between two samples and the samples provides no information of the discontinuity.
Antialiasing (Prefiltering)

Antialiasing (nonuniform sampling)

- The impulse train is modified as
  \[ \sum_{i=-\infty}^{\infty} \delta(x - iT + \frac{1}{2} - \xi) \]

- It turns regular aliasing into noise. But random noise is less distracting than coherent aliasing.

Antialiasing (adaptive sampling)

- Take more samples only when necessary. However, in practice, it is hard to know where we need supersampling. Some heuristics could be used.

- It makes a less aliased image, but may not be more efficient than simple supersampling particular for complex scenes.
Application to ray tracing

- Sources of aliasing: object boundary, small objects, textures and materials
- Good news: we can do sampling easily
- Bad news: we can’t do prefiltering
- Key insight: we can never remove all aliasing, so we develop techniques to mitigate its impact on the quality of the final image.

pbrt sampling interface

- Creating good sample patterns can substantially improve a ray tracer’s efficiency, allowing it to create a high-quality image with fewer rays.
- core/sampling.*, samplers/*

An ineffective sampler

An effective sampler
Sampler

```cpp
Sampler(int xstart, int xend,
    int ystart, int yend, int spp);
bool GetNextSample(Sample *sample);
int TotalSamples();
```

In `core/scene.cpp`,

```cpp
while (sampler->GetNextSample(sample)) {
    ...
}
```

Sample

```cpp
Struct Sample {
    Sample(SurfaceIntegrator *surf,
        VolumeIntegrator *vol,
        const Scene *scene);

    float imageX, imageY;
    float lensU, lensV;
    float time;
    ...
}
```

Stratified sampling

- Subdivide the sampling domain into non-overlapping regions (strata) and take a single sample from each one so that it is less likely to miss important features.

Stratified sampling

- **completely random**
- **stratified uniform**
- **stratified jittered**
Comparison of sampling methods

256 samples per pixel as reference

1 sample per pixel (no jitter)

1 sample per pixel (jittered)

4 samples per pixel (jittered)

High dimension

• D dimension means $N^D$ cells.
• Solution: make strata separately and associate them randomly, also ensuring good distributions.

Stratified sampling

```c
void StratifiedSample1D(float *samp, int nSamples, bool jitter) {
  float invTot = 1.f / nSamples;
  for (int i = 0; i < nSamples; ++i) {
    float delta = jitter ? RandomFloat() : 0.5f;
    *samp++ = (i + delta) * invTot;
  }
}

void StratifiedSample2D(float *samp, int nx, int ny, bool jitter) {
  float dx = 1.f / nx, dy = 1.f / ny;
  for (int y = 0; y < ny; ++y) {
    for (int x = 0; x < nx; ++x) {
      float jx = jitter ? RandomFloat() : 0.5f;
      float jy = jitter ? RandomFloat() : 0.5f;
      *samp++ = (x + jx) * dx;
      *samp++ = (y + jy) * dy;
    }
  }
}
```
**Stratified sampling**

```cpp
StratifiedSample2D(imageSamples, xPixelSamples, yPixelSamples, jitterSamples);
StratifiedSample2D(lensSamples, xPixelSamples, yPixelSamples, jitterSamples);
StratifiedSample1D(timeSamples, xPixelSamples*yPixelSamples, jitterSamples);
for (int o=0; o<2*xPixelSamples*yPixelSamples; o+=2)
{
    imageSamples[o] += xPos;
    imageSamples[o+1] += yPos;
}
Shuffle(lensSamples, xPixelSamples*yPixelSamples, 2);
Shuffle(timeSamples, xPixelSamples*yPixelSamples, 1);
```

**Latin hypercube sampling**

- Integrators could request an arbitrary \( n \) samples.

nx1 or 1xn doesn’t give a good sampling pattern.

A worst case for stratified sampling

**Stratified sampling**

```cpp
bool StratifiedSampler::GetNextSample(Sample *sample) {
    if (samplePos == xPixelSamples*yPixelSamples) {
        <advance to next pixel for stratified sampling>
        <Generate stratified camera samples>
    }
    <fill in sample by table lookup>
    for (u_int i = 0; i < sample->n1D.size(); ++i)
        LatinHypercube(sample->oneD[i], sample->n1D[i], 1);
    for (u_int i = 0; i < sample->n2D.size(); ++i)
        LatinHypercube(sample->twoD[i], sample->n2D[i], 2);
    ++samplePos;
    return true;
}
```
Stratified sampling

- 1 camera sample and 16 shadow samples per pixel
- 16 camera samples and each with 1 shadow sample per pixel

Best candidate sampling

- Stratified sampling doesn’t guarantee good sampling across pixels.
- Poisson disk pattern addresses this issue. The Poisson disk pattern is a group of points with no two of them closer to each other than some specified distance.
- It can be generated by dart throwing. It is time-consuming.
- Best-candidate algorithm by Dan Mitchell. It generates many candidates randomly and only insert the one farthest to all previous samples.
- Compute a “tilable pattern” offline.

It avoids holes and clusters.
Best candidate sampling

stratified jittered, 1 sample/pixel

best candidate, 1 sample/pixel

Best candidate sampling

stratified jittered, 4 sample/pixel

best candidate, 4 sample/pixel

Low discrepancy sampling

- Stratified sampling could suffer when there are holes or clusters.
- Discrepancy can be used to evaluate the quality of a sampling pattern.

\[ D_N(B,P) = \sup_{b \in \mathcal{B}} \frac{\# \{ x_i \in b \}}{N} - Vol(b) \]

\[ D_N^*(P) \] for the set of AABBs with a corner at the origin

1D discrepancy

\[ x_i = \frac{i}{N} \Rightarrow D_N^*(x_1,\ldots,x_n) = \frac{1}{N} \]

\[ x_i = \frac{i-0.5}{N} \Rightarrow D_N^*(x_1,\ldots,x_n) = \frac{1}{2N} \]

\[ x_i = \text{general} \Rightarrow D_N^*(x_1,\ldots,x_n) = \frac{1}{2N} + \max_{i \in [1,N]} x_i - \frac{2i-1}{2N} \]

Uniform is optimal! Fortunately, for higher dimension, the low-discrepancy patterns are less uniform.
Radial inverse

- Building block for high-D sequences
- “inverts” an integer given in base b

\[ n = a_k \ldots a_2 a_1 = a_1 b^0 + a_2 b^1 + a_3 b^2 + \ldots \]

\[ \Phi_b(n) = 0.a_1 a_2 \ldots a_k = a_1 b^{-1} + a_2 b^{-2} + a_3 b^{-3} + \ldots \]

van der Corput sequence

- Most simple sequence
- Uses radical inverse of base 2
- Achieves minimal possible discrepancy

\[ D_i^k(P) = O \left( \frac{\log N}{N} \right) \]

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \text{binary form of } i )</th>
<th>( \text{radical inverse } x_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0.01</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>0.11</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>0.001</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>0.101</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>0.011</td>
</tr>
</tbody>
</table>

Halton sequence

- Can be used if N is not known in advance
- All prefixes of a sequence are well distributed
- Use prime number bases for each dimension
- Achieves best possible discrepancy

\[ x_i = (\Phi_2(i), \Phi_3(i), \Phi_5(i), \ldots, \Phi_{p_d}(i)) \]

\[ D_i^k(P) = O \left( \frac{(\log N)^d}{N^d} \right) \]

Hammersley sequence

- Similar to Halton
- Need to know total number of samples in advance
- Better discrepancy than Halton

\[ x_i = \left( \frac{i - 1/2}{N}, \Phi_{b_1}(i), \Phi_{b_2}(i), \ldots, \Phi_{b_{d-1}}(i) \right) \]
Folded radical inverse

- Hammersley-Zaremba
- Halton-Zaremba
- Improves discrepancy

\[ \Phi_b(n) = \sum_{i=1}^{\infty} \frac{(a_i + i - 1) \mod b \cdot 1}{b^i} \]

Radial inverse

Halton
Hammersley

Low discrepancy sampling

stratified jittered, 1 sample/pixel

Hammersley sequence, 1 sample/pixel
Reconstruction filters

• Given image samples, we can do the following to compute pixel values.
  1. reconstruct a continuous function $L'$ from samples
  2. prefilter $L'$ to remove frequency higher than Nyquist limit
  3. sample $L'$ at pixel locations

• Instead, we consider an interpolation problem

$$I(x, y) = \sum_{i} \frac{f(x - x_i, y - y_i)L(x_i, y_i)}{\sum_i f(x - x_i, y - y_i)}$$

Filter

• provides an interface to $f(x, y)$
• Film stores a pointer to a filter and use it to filter the output before writing it to disk.

```cpp
Filter::Filter(float xw, float yw)
Float Evaluate(float x, float y);
```

• filters/*

Box filter

• Most commonly used in graphics. It’s just about the worst filter possible, incurring postaliasing by high-frequency leakage.

```cpp
Float BoxFilter::Evaluate(float x, float y)
{
    return 1.;
}
```

Triangle filter

```cpp
Float TriangleFilter::Evaluate(float x, float y)
{
    return max(0.f, xWidth-fabsf(x)) * max(0.f, yWidth-fabsf(y));
}
```
**Gaussian filter**

- Gives reasonably good results in practice

```cpp
Float GaussianFilter::Evaluate(float x, float y)
{
    return Gaussian(x, expX)*Gaussian(y, expY);
}
```

**Mitchell filter**

- Parametric filters, tradeoff between ringing and blurring
- Negative lobes

**Windowed sinc filter**

- Separable filter
- Two parameters, B and C, B+2C=1 suggested

\[
 h(x) = \begin{cases} 
 (12 - 9B - 6C)x^3 + (-18 + 12B + 6C)x^2 + (6 - 2B) & |x| < 1 \\
 \frac{1}{6} (-B - 6C)x^3 + (6B + 30C)x^2 + (-12B - 48C)x + (8B + 24C) & 1 < |x| < 2 \\
 0 & \text{otherwise}
\end{cases}
\]