Sampling and Reconstruction

Digital Image Synthesis

Yung-Yu Chuang

10/26/2005

with slides by Pat Hanrahan, Torsten Moller 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

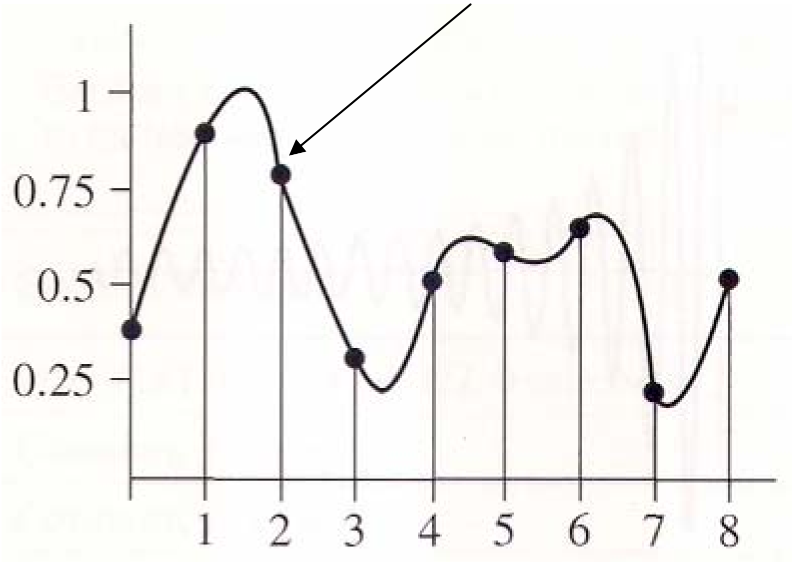
Aliasing

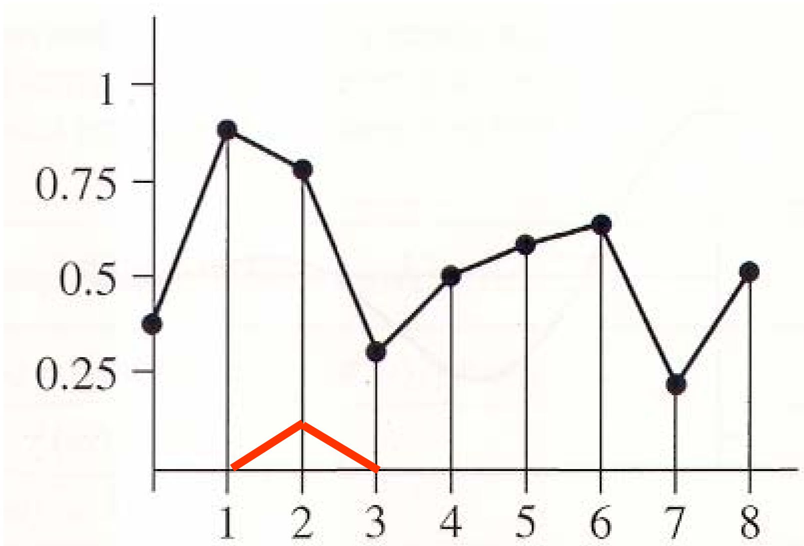
- Reconstruction generates an approximation to the original function. Error is called aliasing.

sampling

reconstruction

sample value





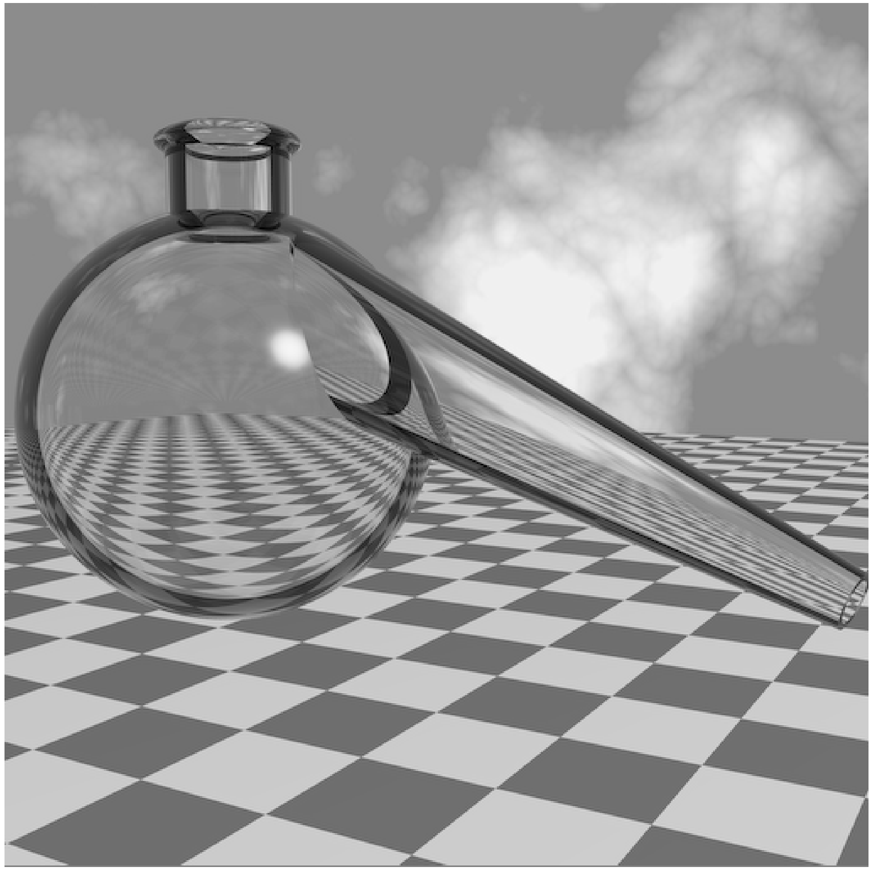
sample position

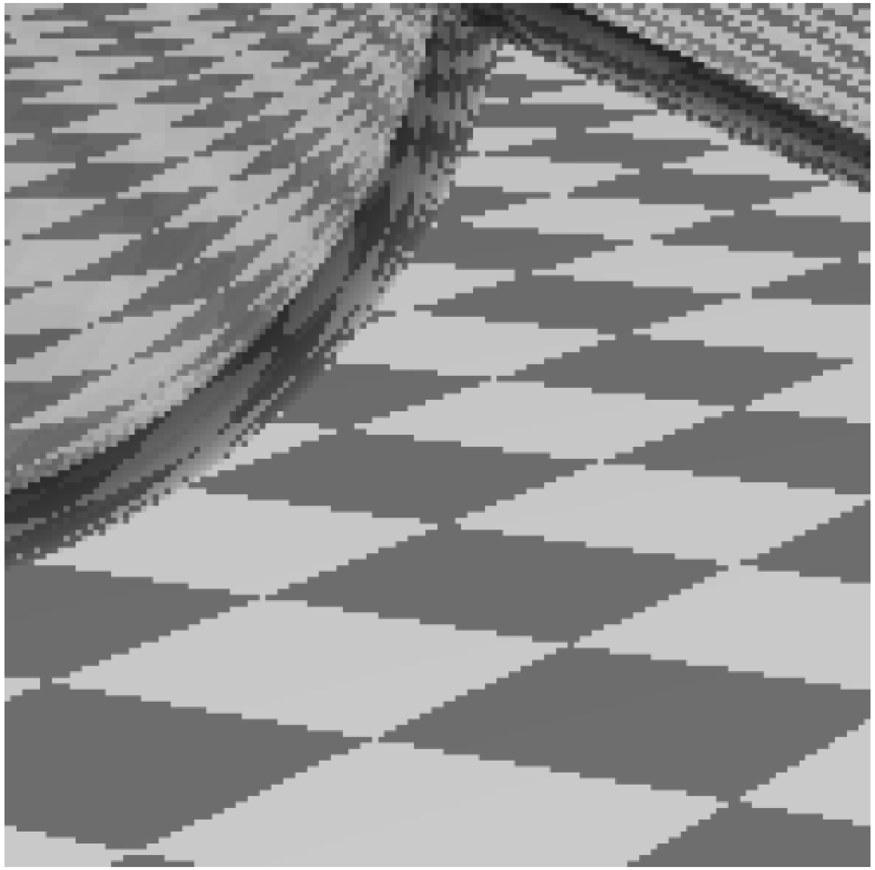
Sampling in computer graphics

- Artifacts due to sampling - Aliasing
  - Jaggies
  - Moire
  - Flickering small objects
  - Sparkling highlights
  - Temporal strobing (such as Wagon-wheel effect)
- Preventing these artifacts - Antialiasing

Jaggies

Retort sequence by Don Mitchell



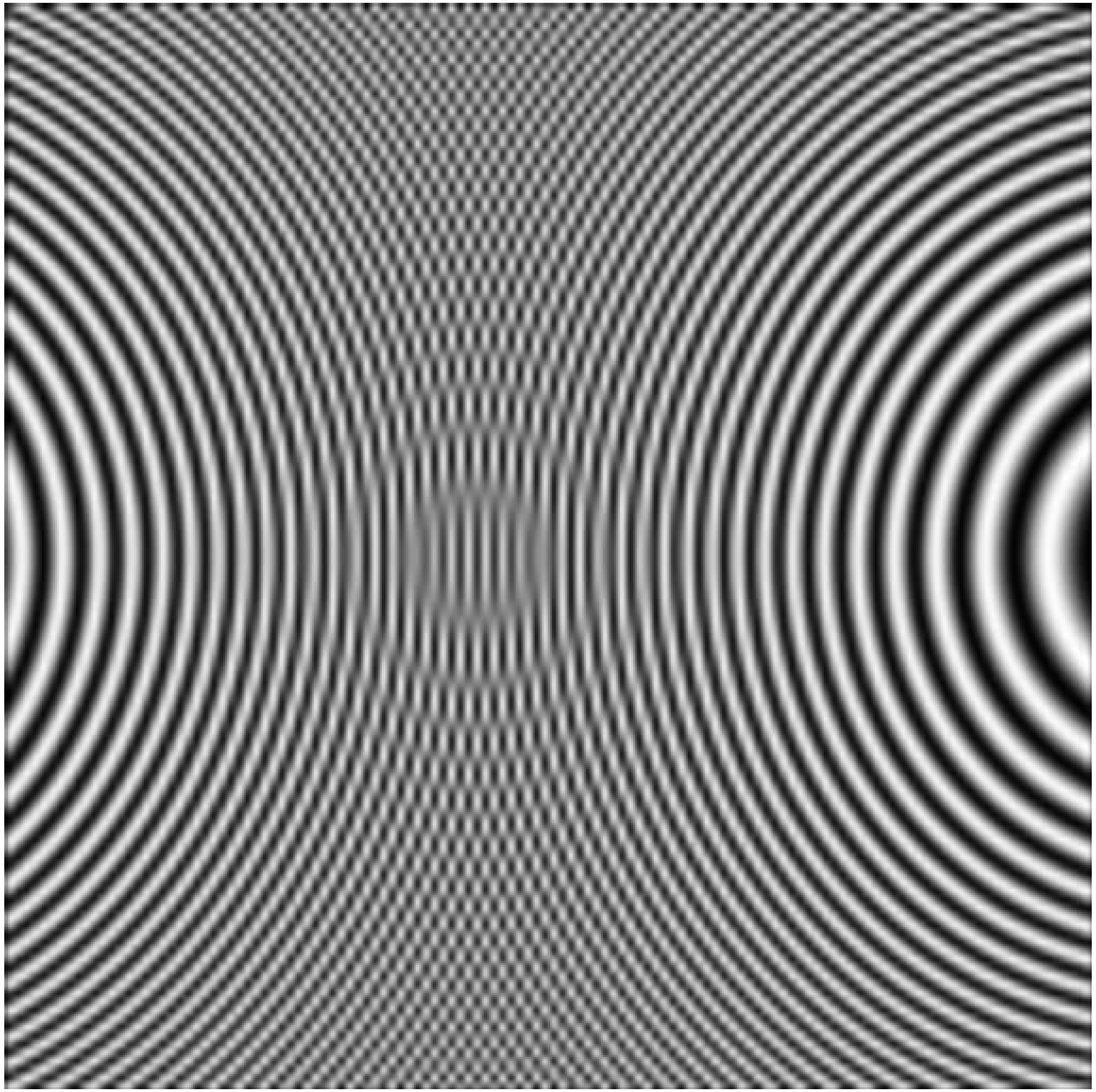


Staircase pattern or jaggies

Moire pattern

- Sampling the equation

$$\sin(x^2 + y^2)$$



Fourier transforms

- Most functions can be decomposed into a weighted sum of shifted sinusoids.
- Each function has two representations
  – Spatial domain - normal representation
  – Frequency domain - spectral representation
- The *Fourier transform* converts between the spatial and frequency domain

Spatial Domain
$f(x)$

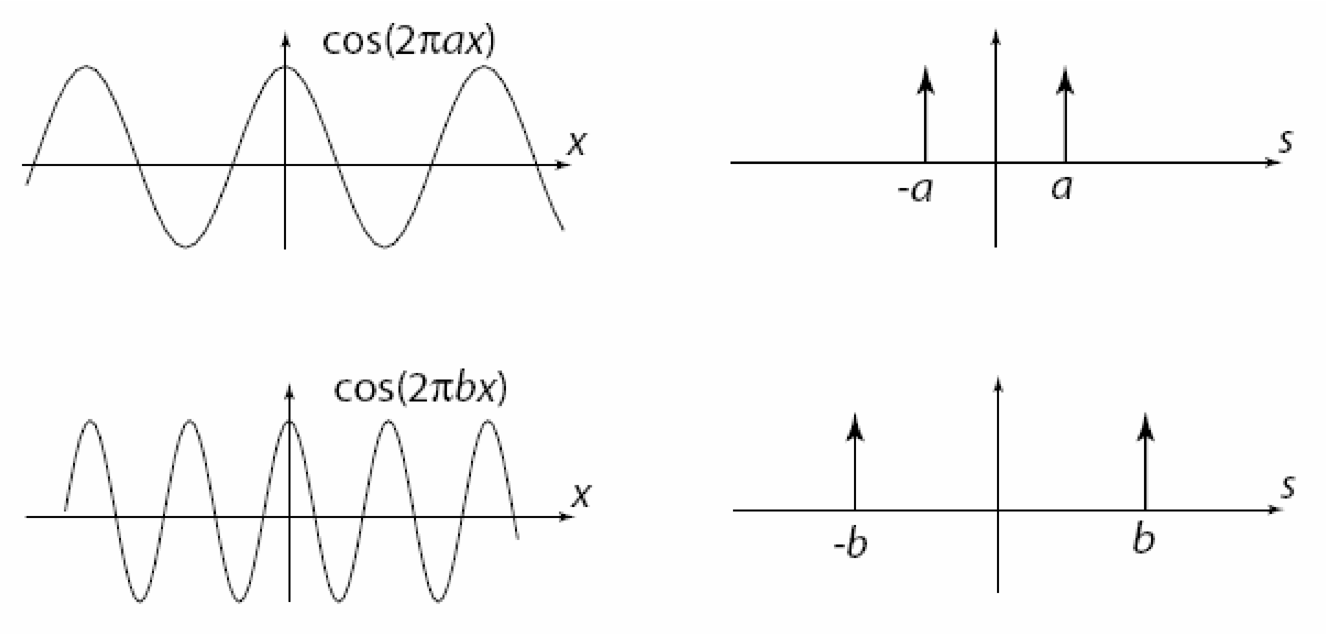
$$\Rightarrow \quad F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-i\omega x}\, dx \quad \Rightarrow$$

$$\Leftarrow \quad f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{i\omega x}\, d\omega \quad \Leftarrow$$

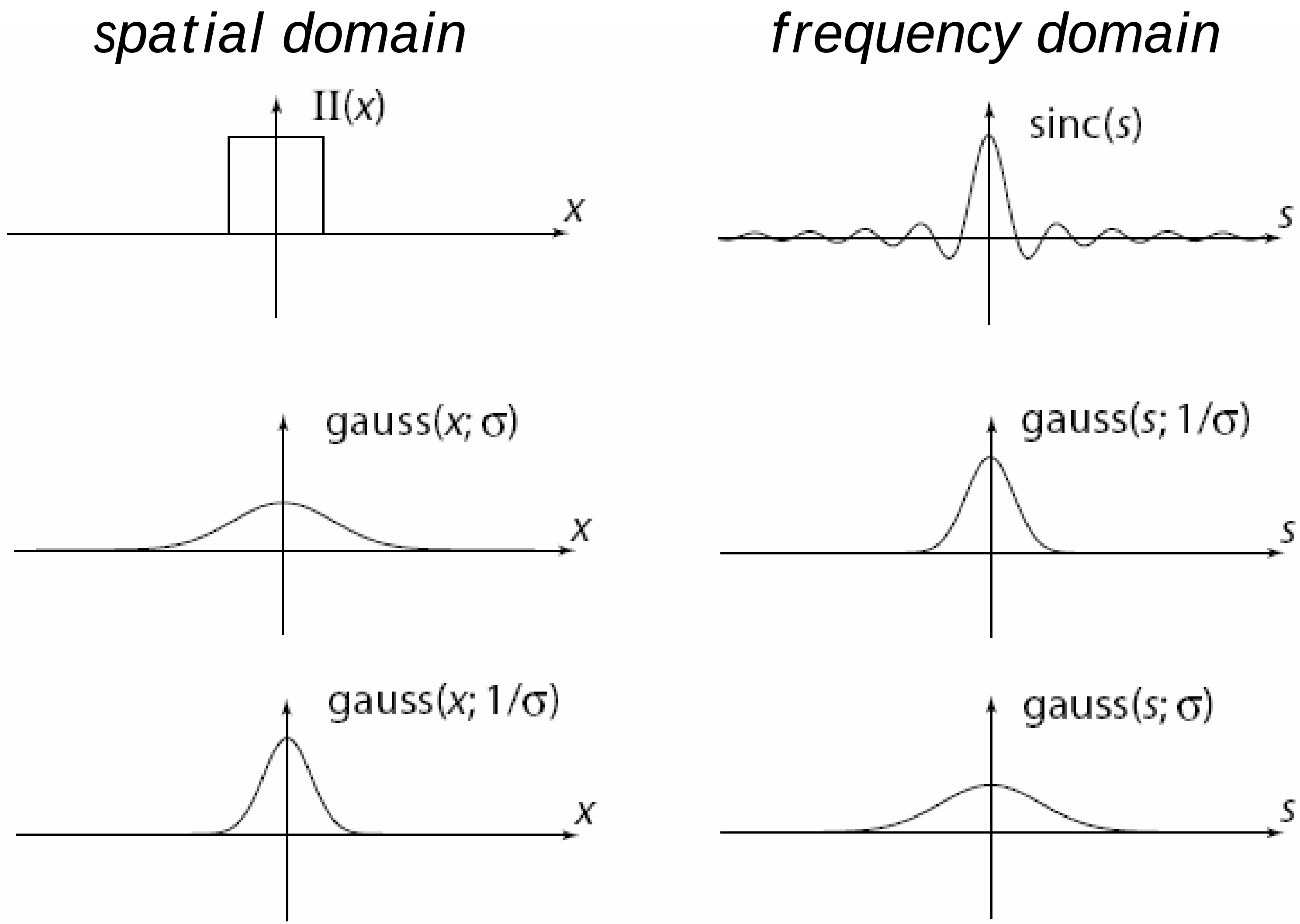
Frequency Domain
$F(\omega)$

Fourier analysis

spatial domain *frequency domain*

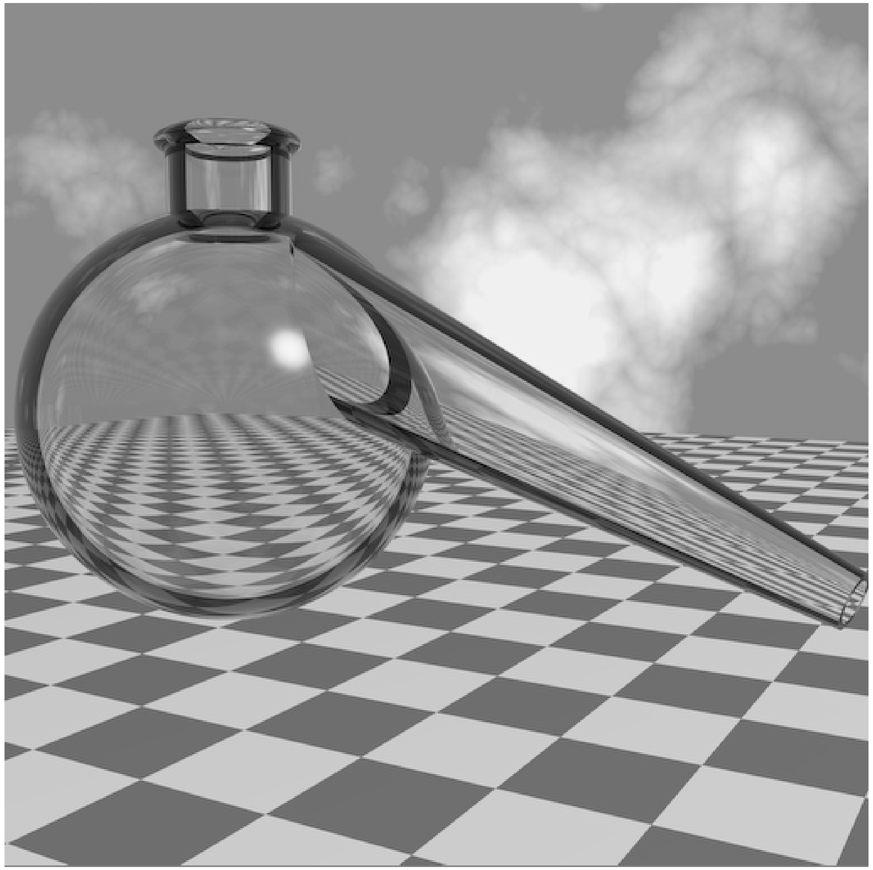


Fourier analysis

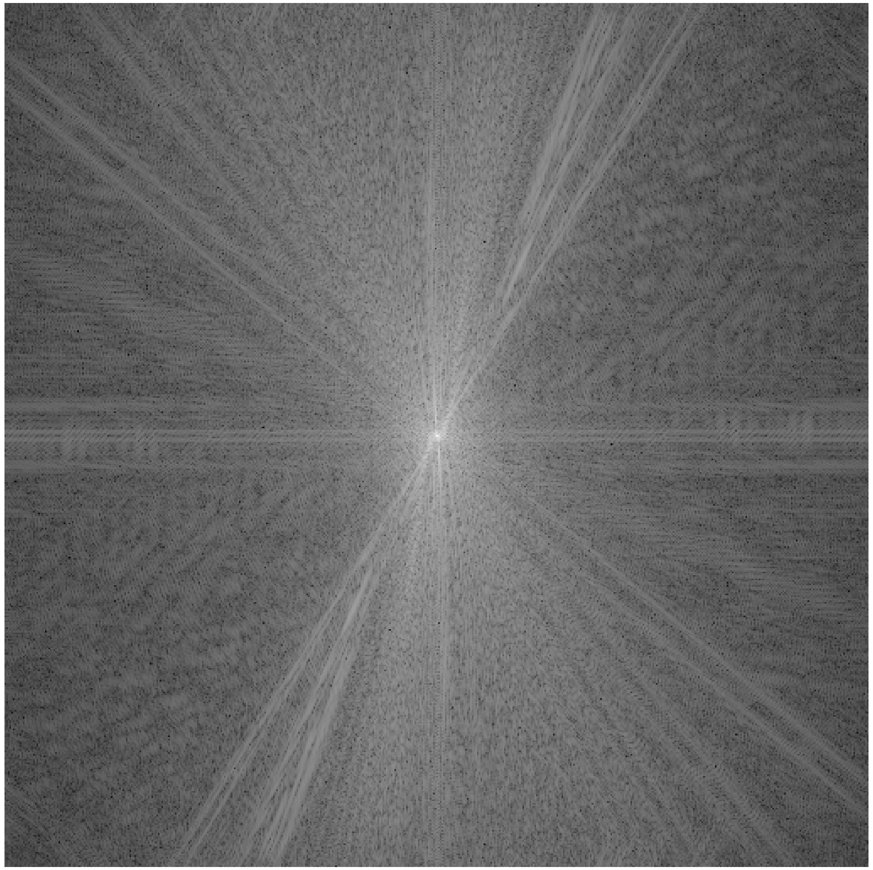


Fourier analysis

spatial domain



frequency domain



Convolution

- *Definition*

$$h(x) = f \otimes g = \int f(x')g(x - x')\,dx'$$

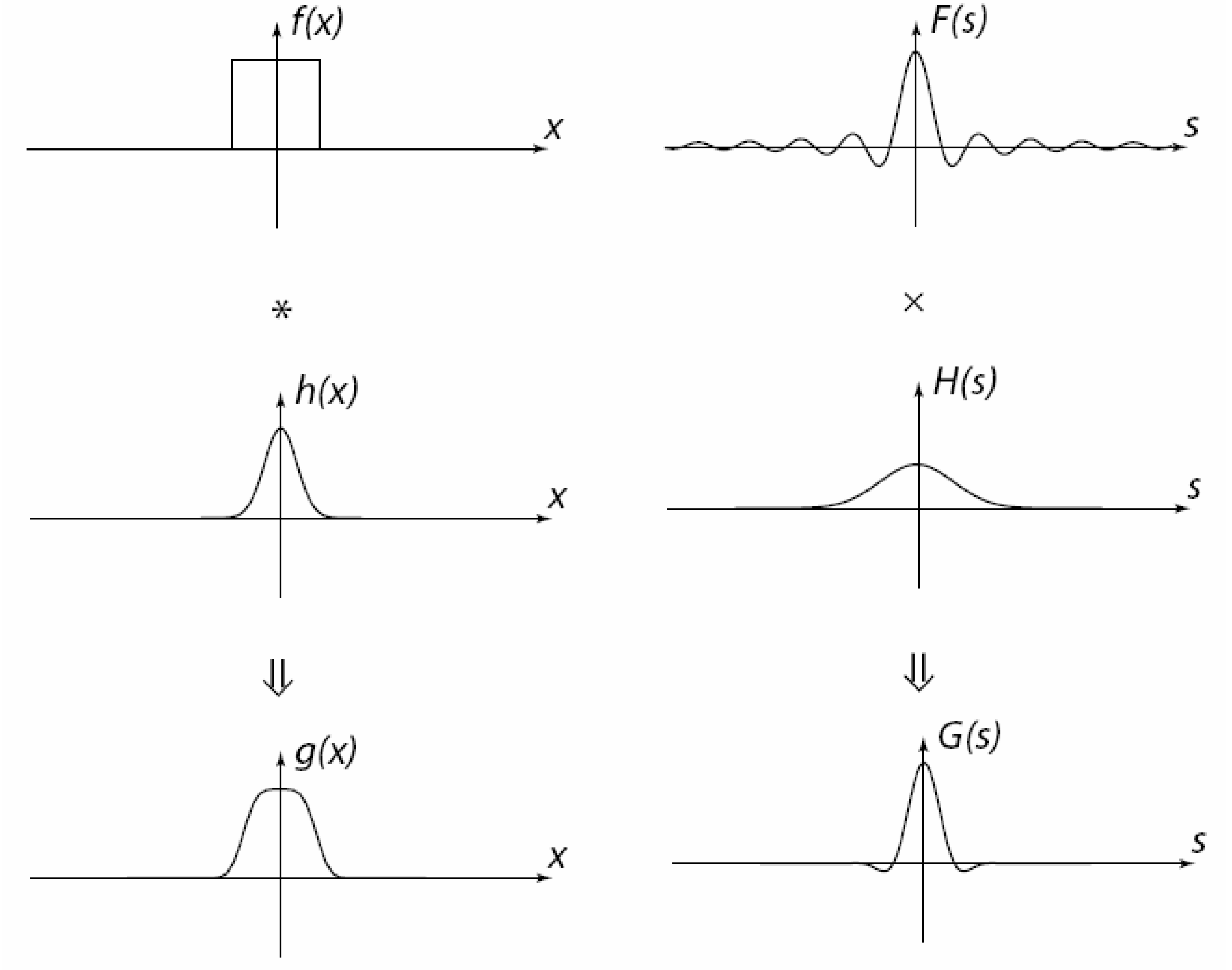
- *Convolution Theorem*: Multiplication in the frequency domain is equivalent to convolution in the space domain.

$$f \otimes g \leftrightarrow F \times G$$

- *Symmetric Theorem*: Multiplication in the space domain is equivalent to convolution in the frequency domain.

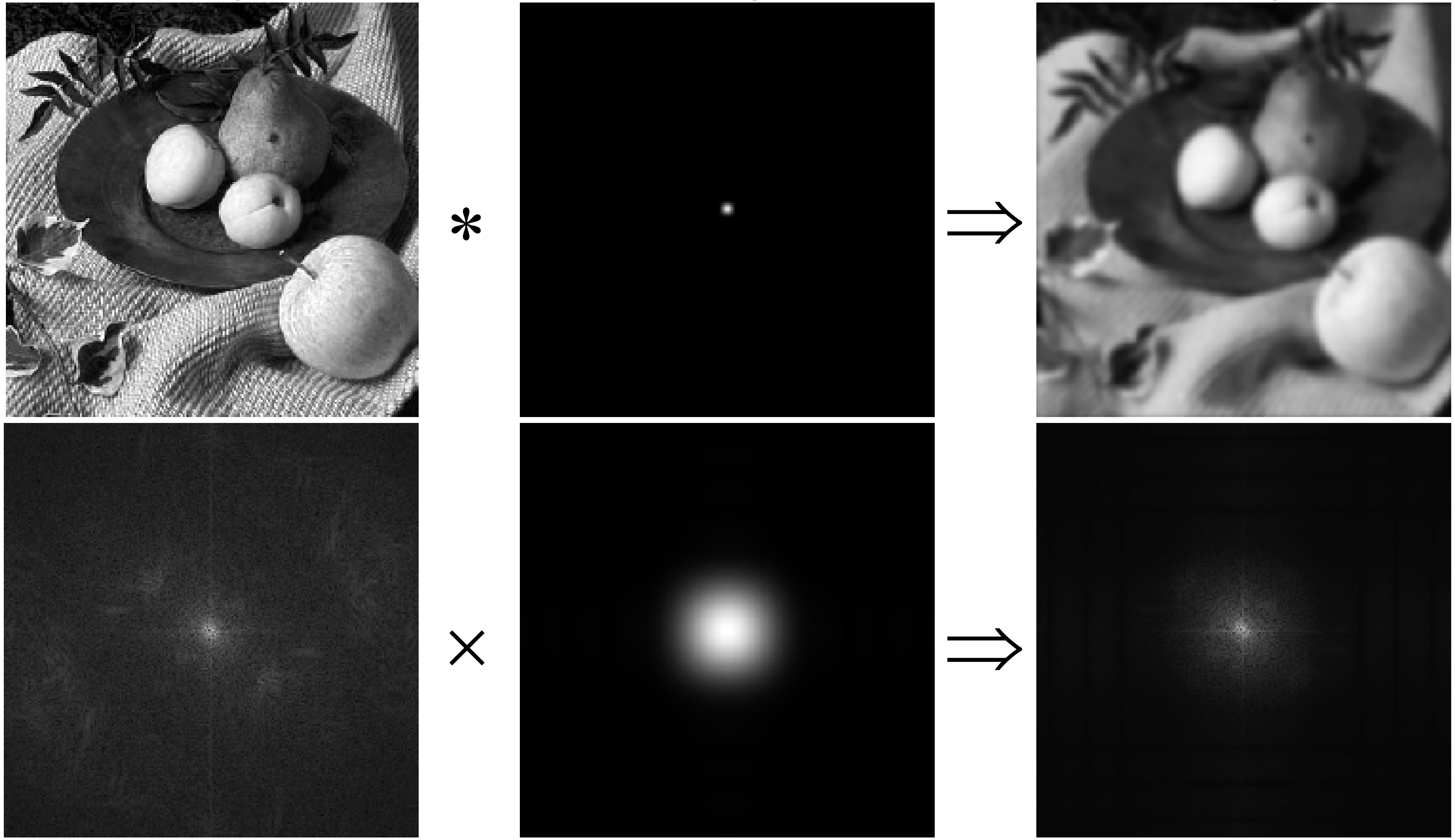
$$f \times g \leftrightarrow F \otimes G$$

1D convolution theorem example



2D convolution theorem example

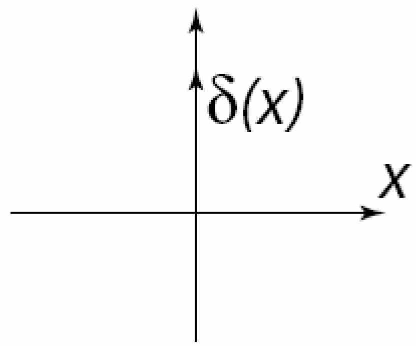
$f(x,y)$ $h(x,y)$ $g(x,y)$



$F(s_x,s_y)$ $H(s_x,s_y)$ $G(s_x,s_y)$

The delta function

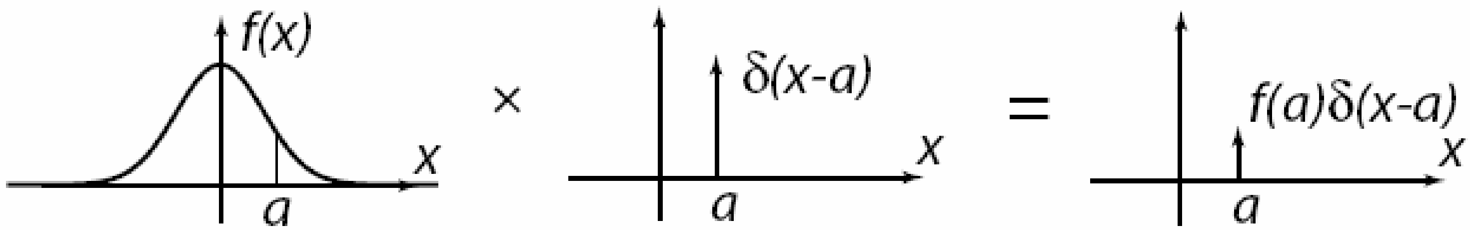
- Dirac delta function, zero width, infinite height and unit area



Sifting and shifting

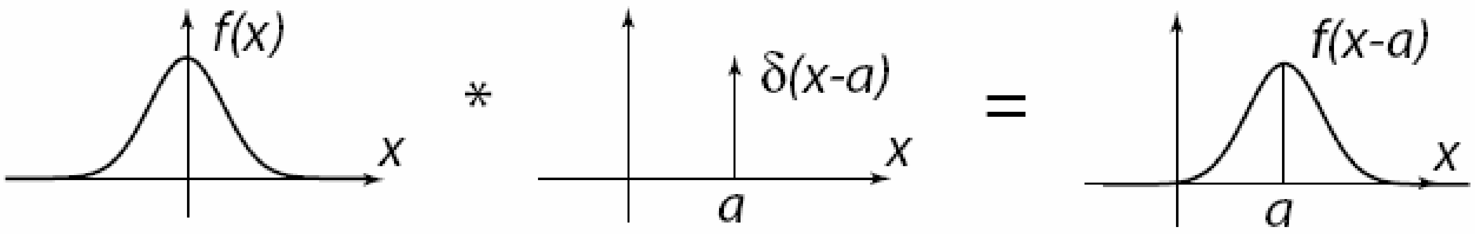
Sifting:

$$f(x)\delta(x-a) = f(a)\delta(x-a)$$



Shifting:

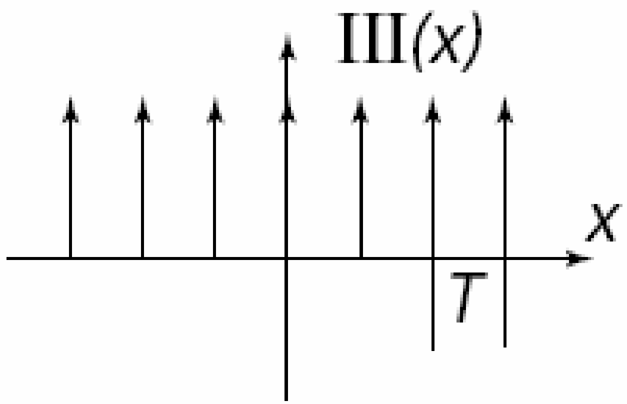
$$f(x) * \delta(x-a) = f(x-a)$$



Shah/impulse train function

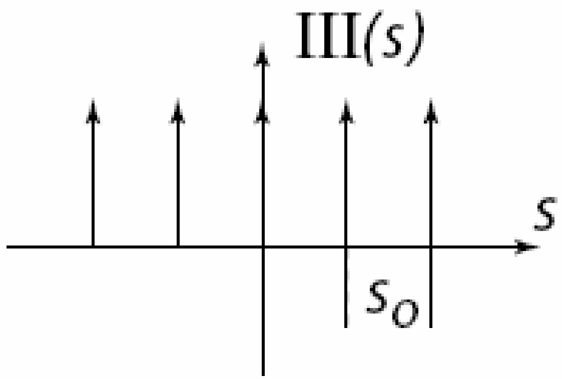
spatial domain

$$\mathrm{III}(x) = \sum_{n=-\infty}^{\infty} \delta(x - nT)$$

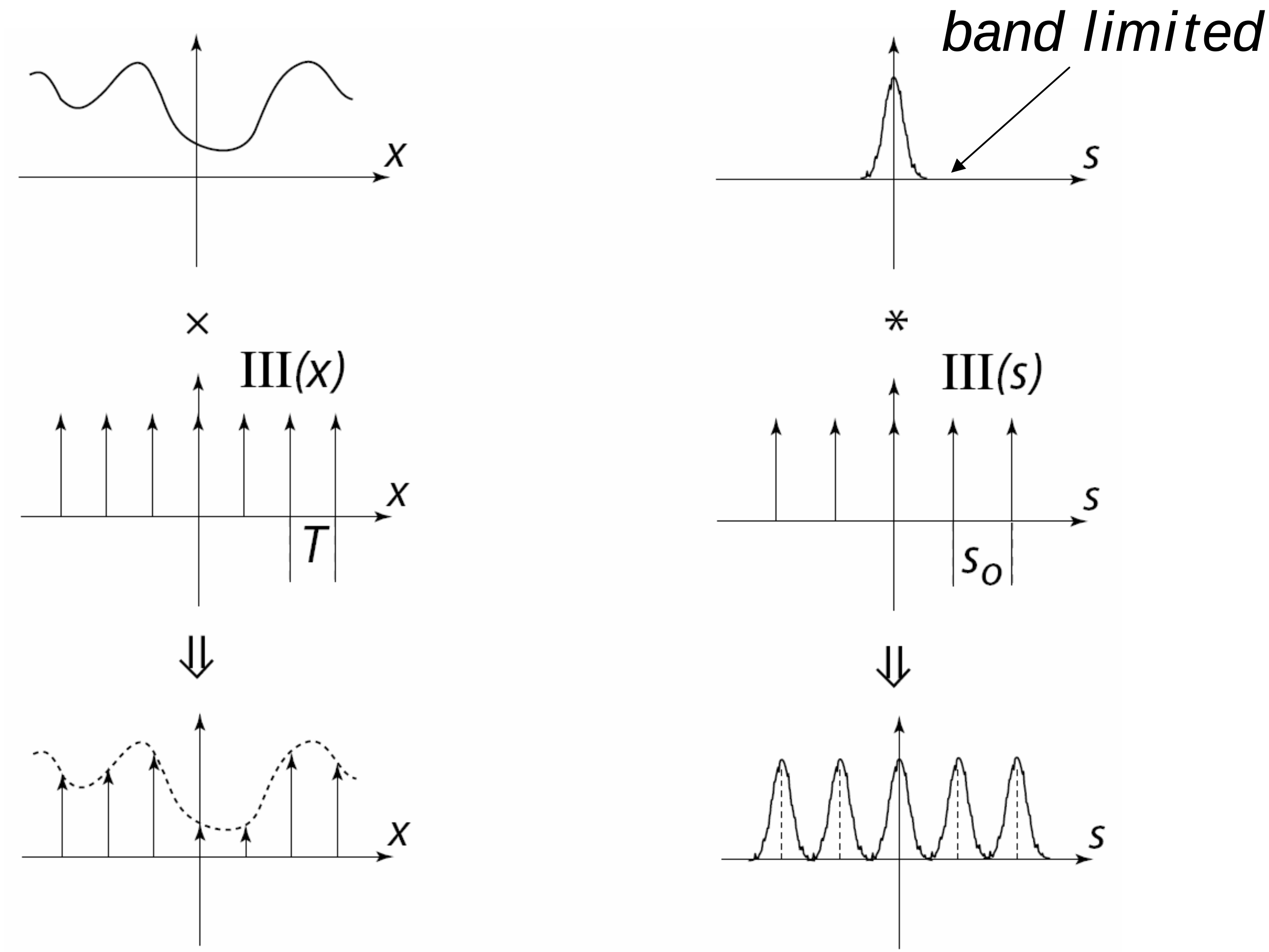


frequency domain

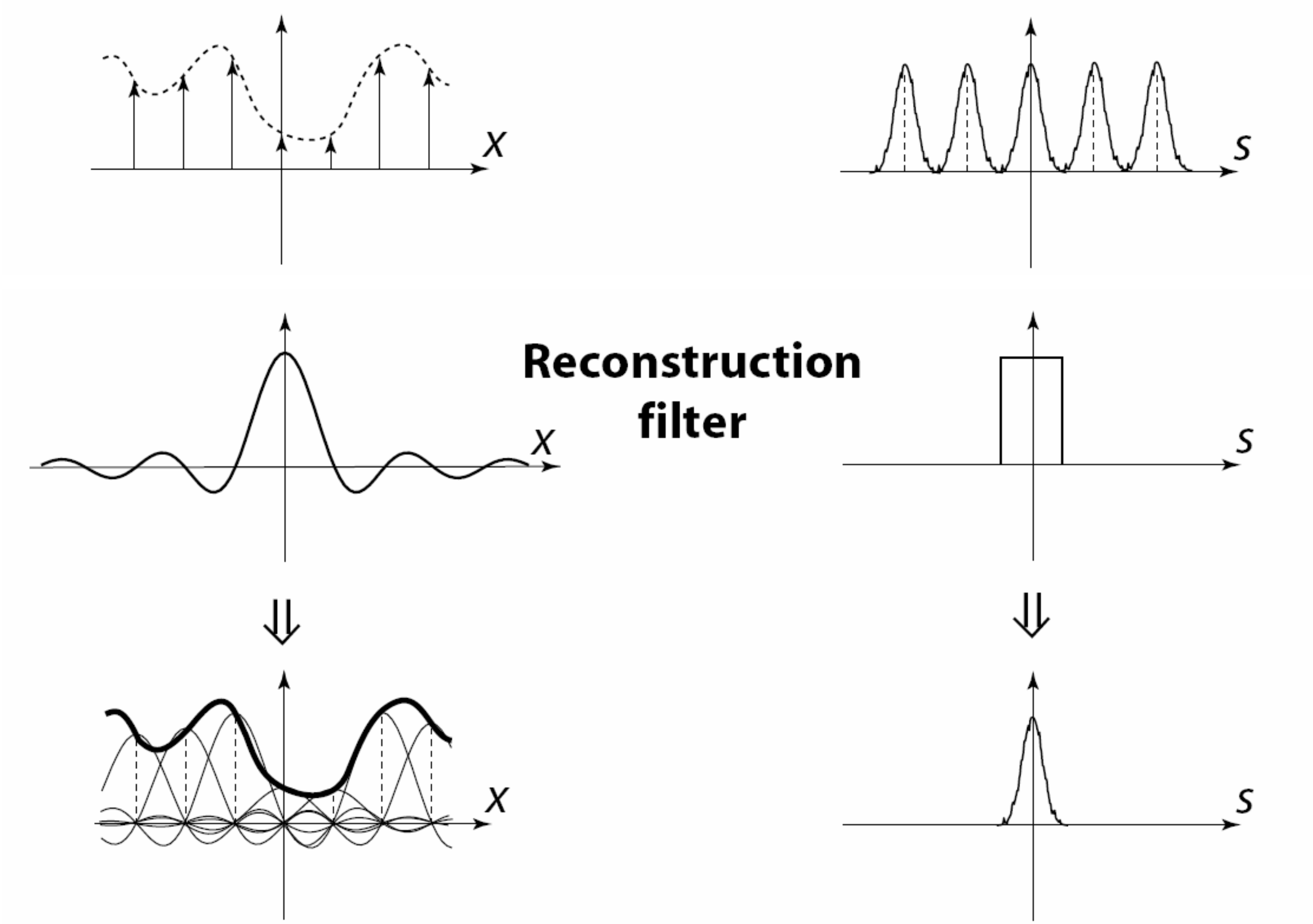
$$\mathrm{III}(s) = \sum_{n=-\infty}^{\infty} \delta(s - ns_o), \quad s_o = 1/T$$



Sampling



Reconstruction



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

In math forms

$$\tilde{F} = (F(s) * \mathrm{III}(s)) \times \Pi(s)$$

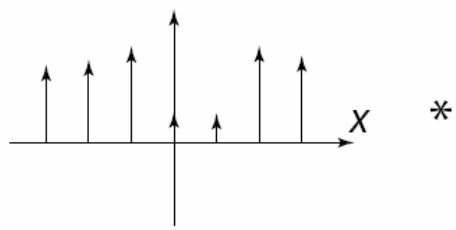
$$\tilde{f} = (f(x) \times \mathrm{III}(x)) * \mathrm{sinc}(x)$$

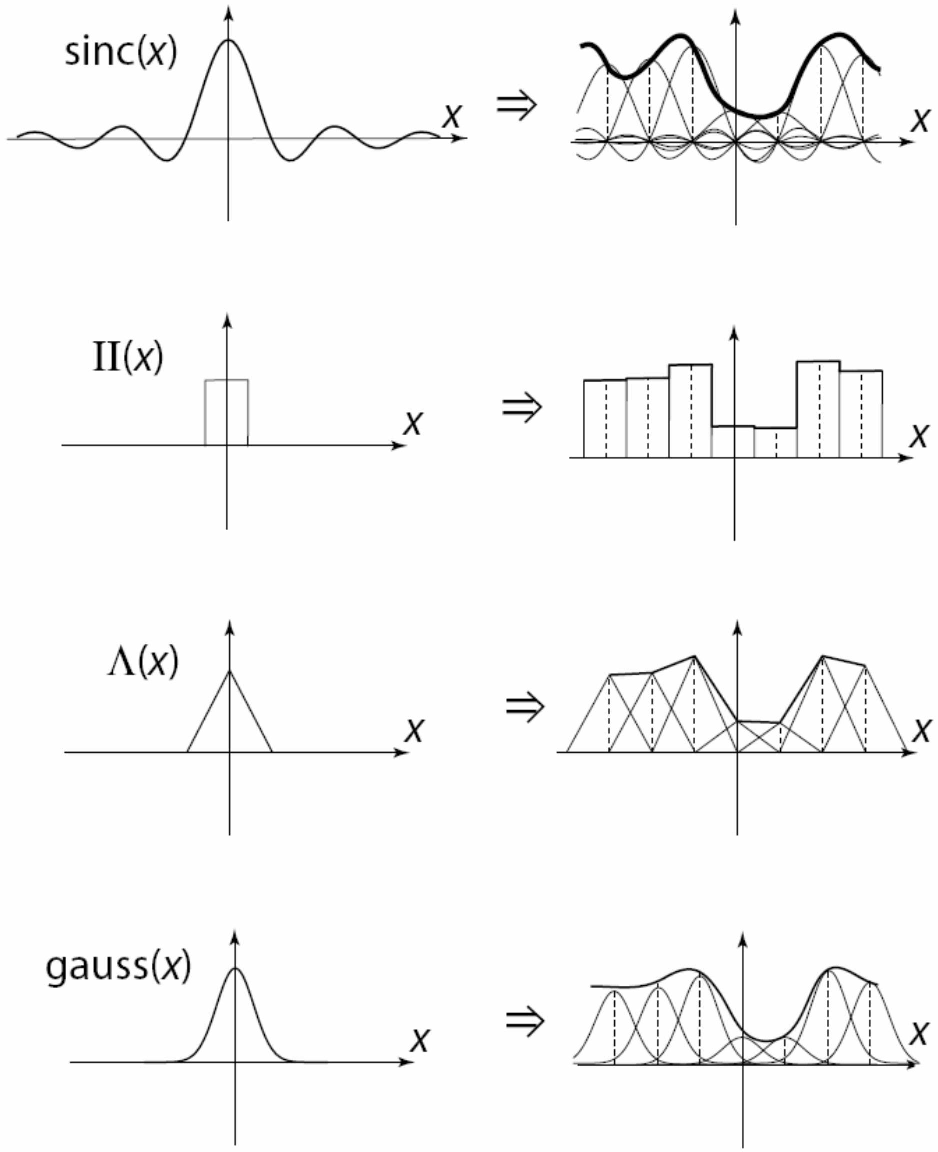
$$\tilde{f}(x) = \sum_{i=-\infty}^{\infty} \mathrm{sinc}(x - i) f(i)$$

Reconstruction filters

The sinc filter, while ideal, has two drawbacks:

- It has large support (slow to compute)
- It introduces ringing in practice



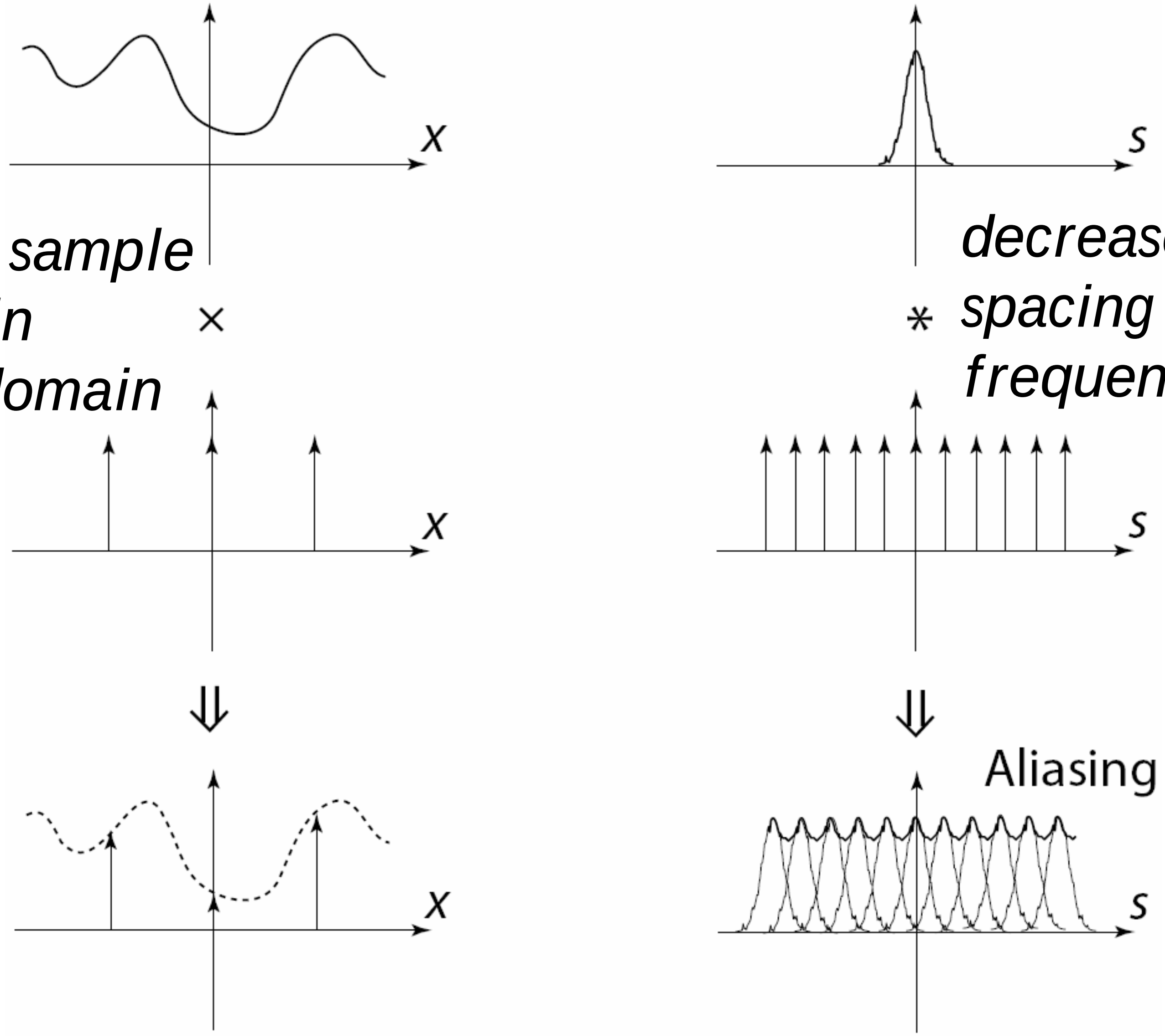


The box filter is bad because its Fourier transform is a sinc filter which includes high frequency contribution from the infinite series of other copies.

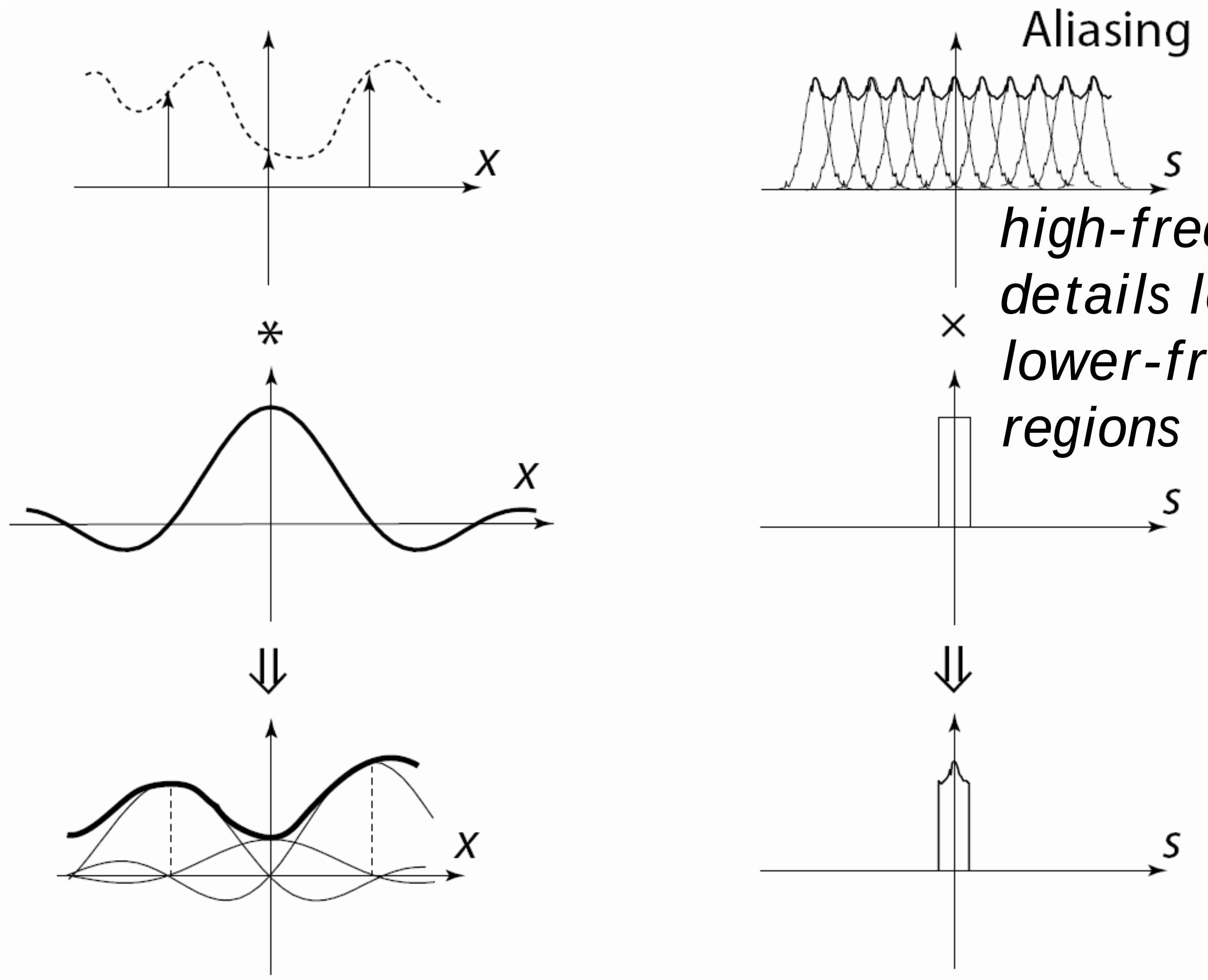
Aliasing

increase sample spacing in spatial domain

decrease sample spacing in frequency domain



Aliasing



high-frequency details leak into lower-frequency regions

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 ½ the sampling frequency.

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

Sampling theorem

- 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 about this discontinuity.

Aliasing

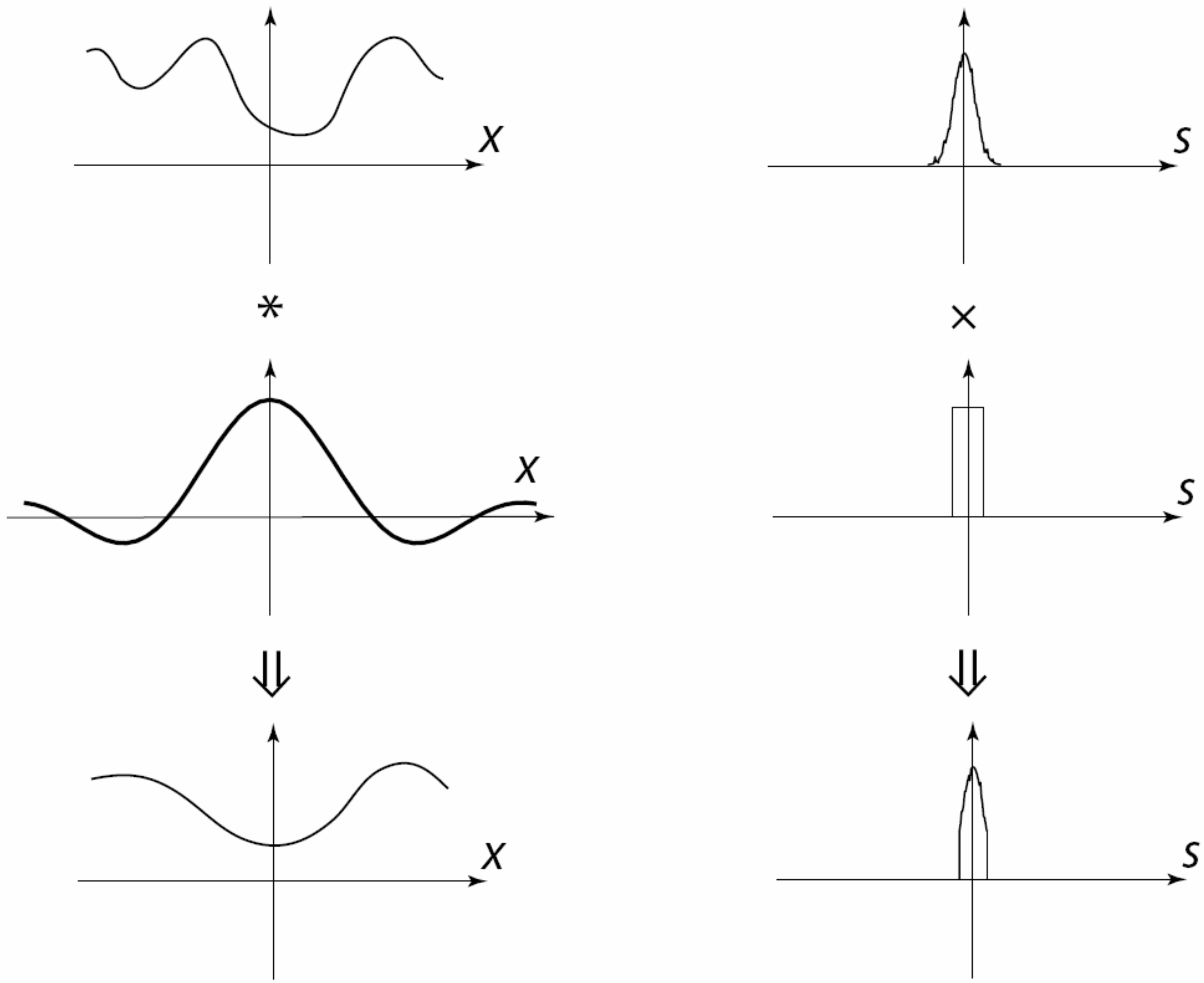
- Prealiasing: due to sampling under Nyquist rate
- Postaliasing: due to use of imperfect reconstruction filter

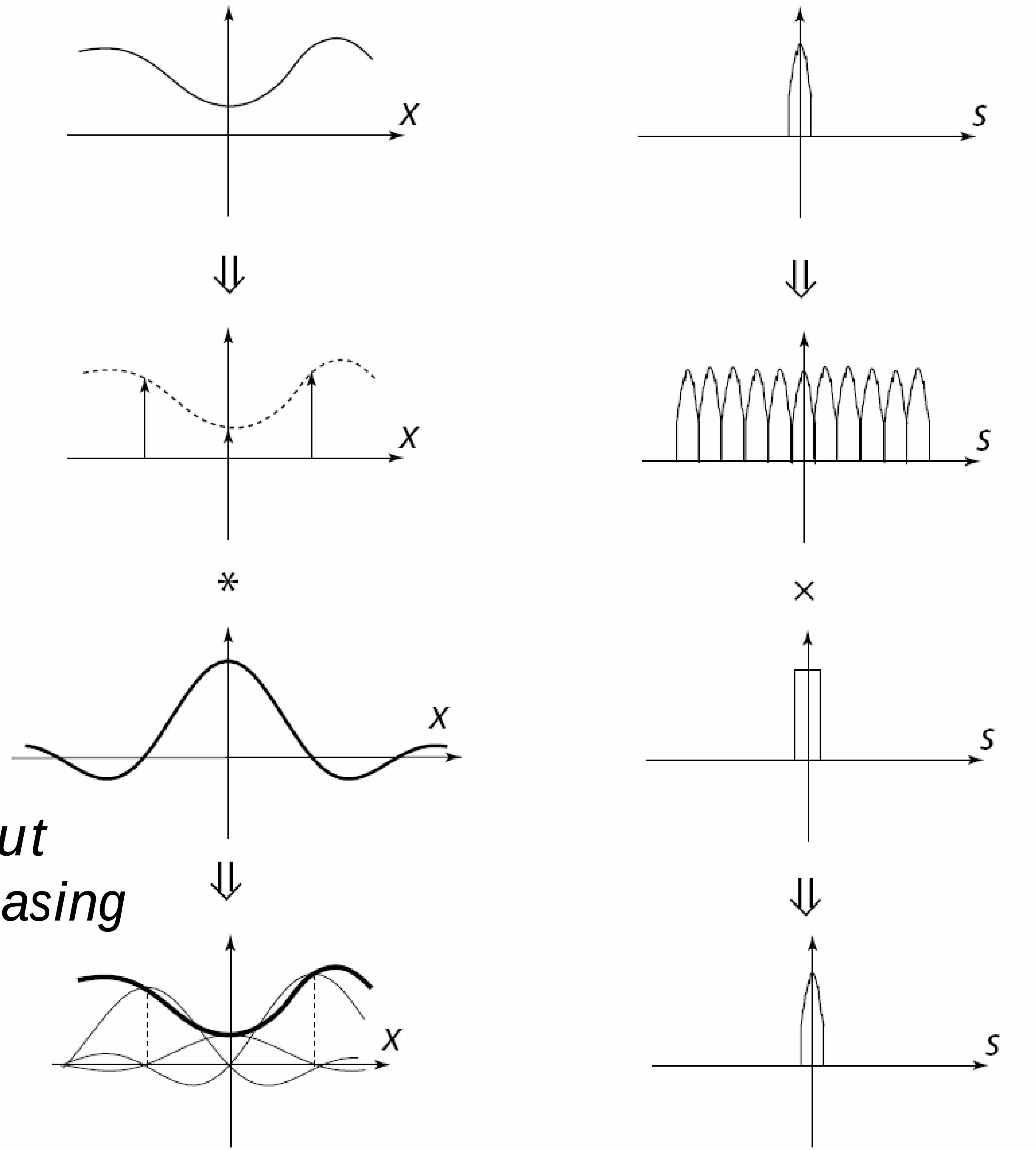
Antialiasing

- Antialiasing = Preventing aliasing

1. Analytically prefilter the signal
   - Not solvable in general
2. Uniform supersampling and resample
3. Nonuniform or stochastic sampling

Antialiasing (Prefiltering)

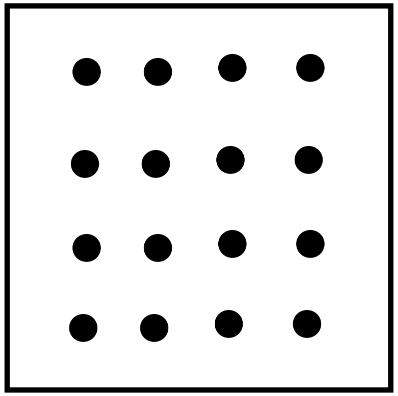




It is blurred, but better than aliasing

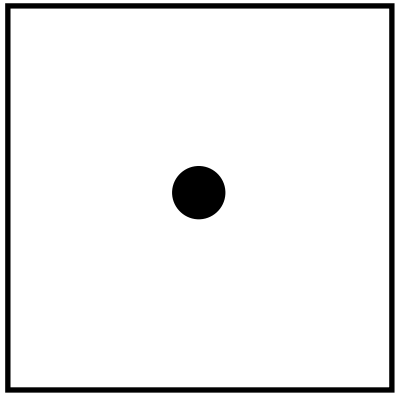
Uniform Supersampling

- Increasing the sampling rate moves each copy of the spectra further apart, potentially reducing the overlap and thus aliasing
- Resulting samples must be resampled (filtered) to image sampling rate



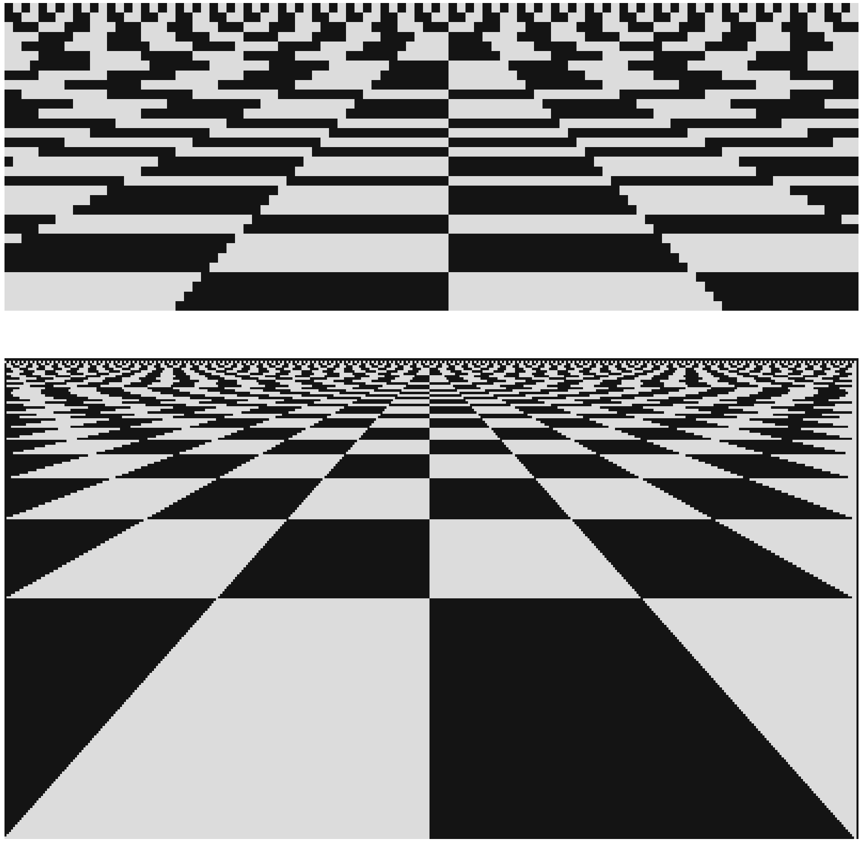
Samples

$$Pixel = \sum_s w_s \cdot Sample_s$$

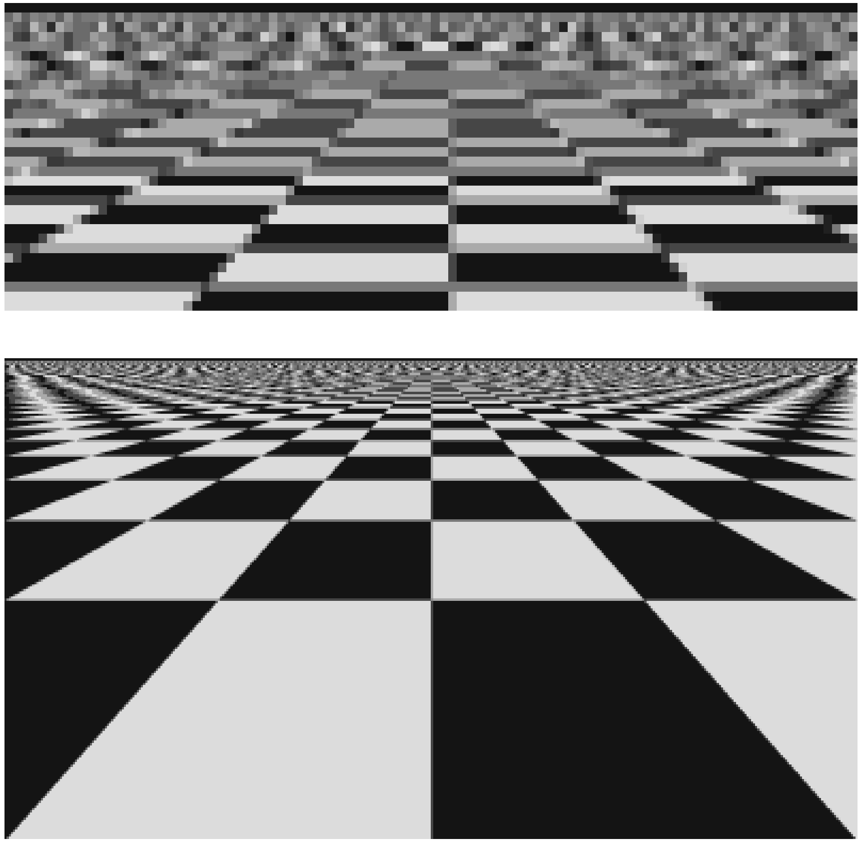


Pixel

Point vs. Supersampled



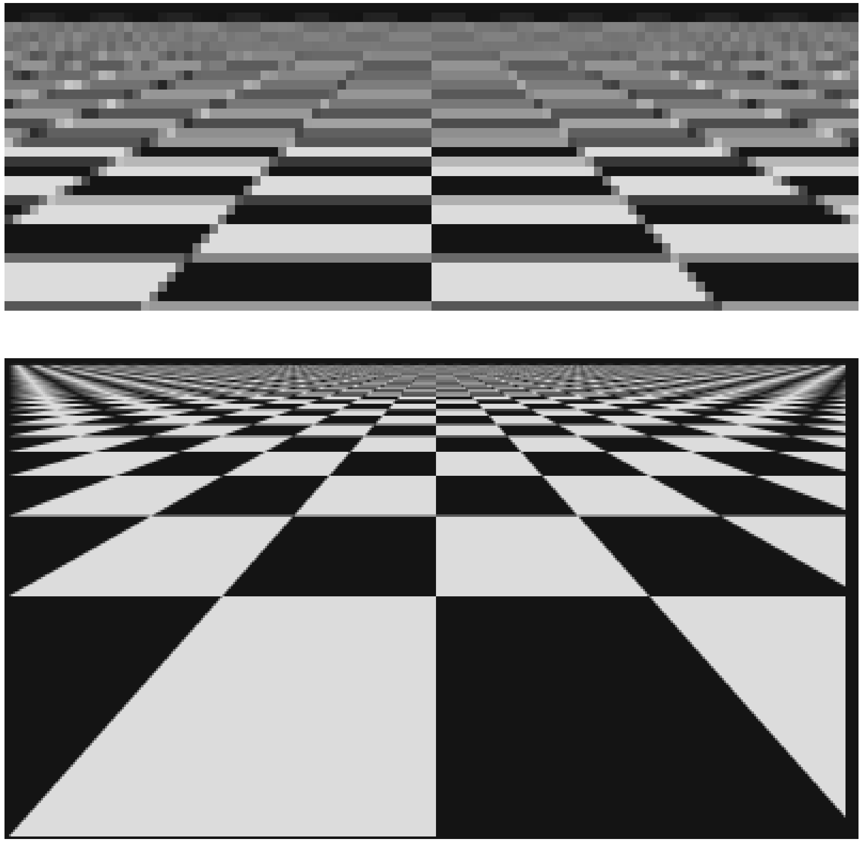
Point



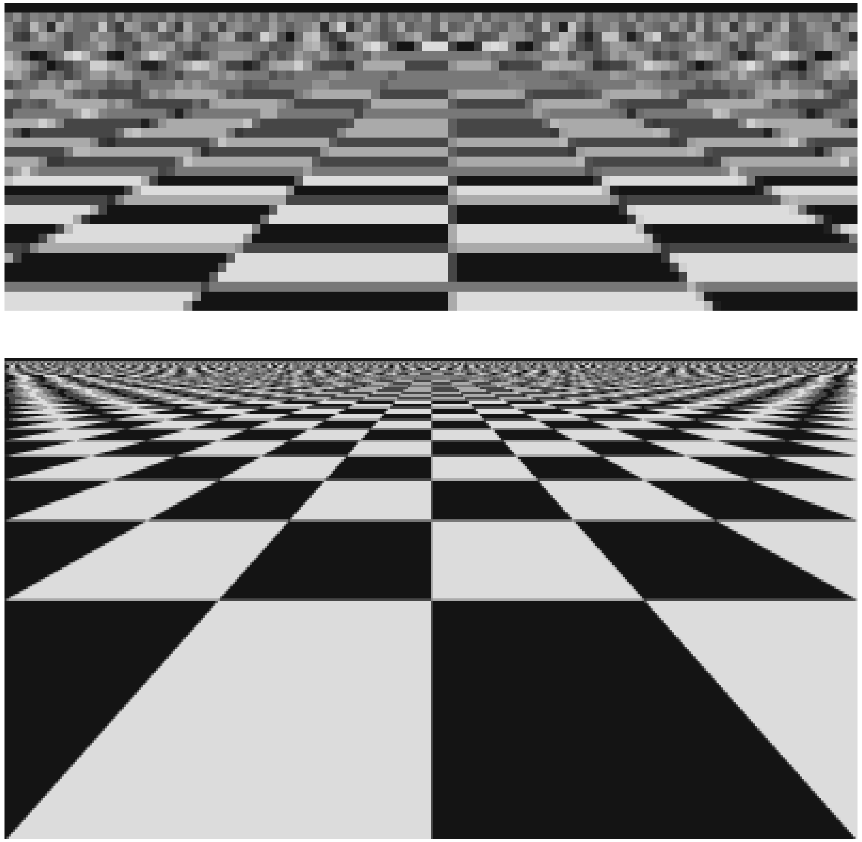
4x4 Supersampled

Checkerboard sequence by Tom Duff

Analytic vs. Supersampled

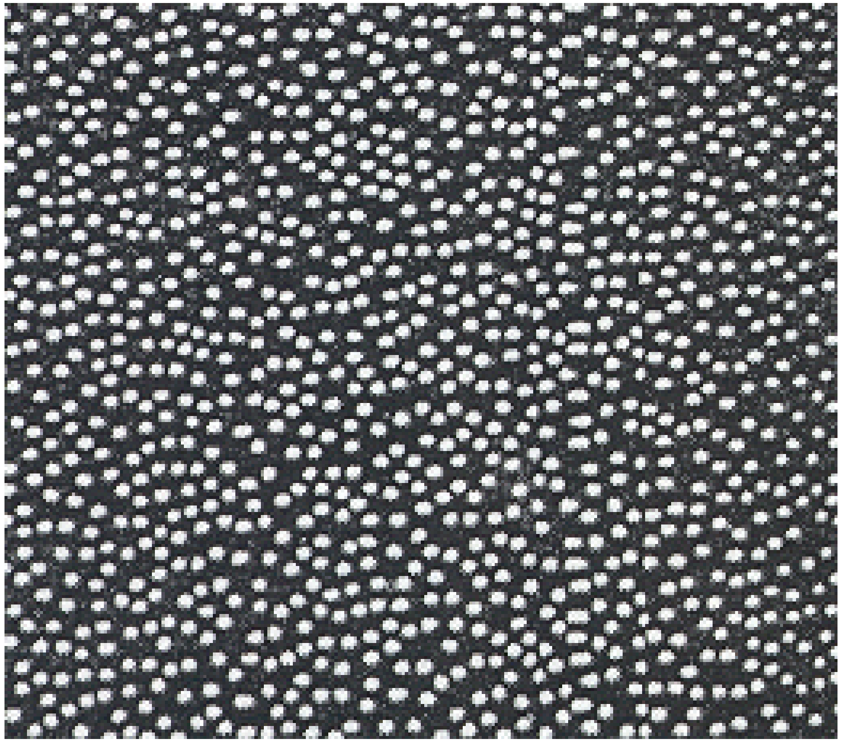


Exact Area



4x4 Supersampled

Distribution of Extrafoveal Cones



Monkey eye
cone distribution

Fourier transform

Yellot theory

- **Aliases replaced by noise**
- **Visual system less sensitive to high freq noise**

Non-uniform Sampling

- Intuition
- Uniform sampling
  - The spectrum of uniformly spaced samples is also a set of uniformly spaced spikes
  - Multiplying the signal by the sampling pattern corresponds to placing a copy of the spectrum at each spike (in freq. space)
  - Aliases are coherent, and very noticable
- Non-uniform sampling
  - Samples at non-uniform locations have a different spectrum; a single spike plus noise
  - Sampling a signal in this way converts aliases into broadband noise
  - Noise is incoherent, and much less objectionable

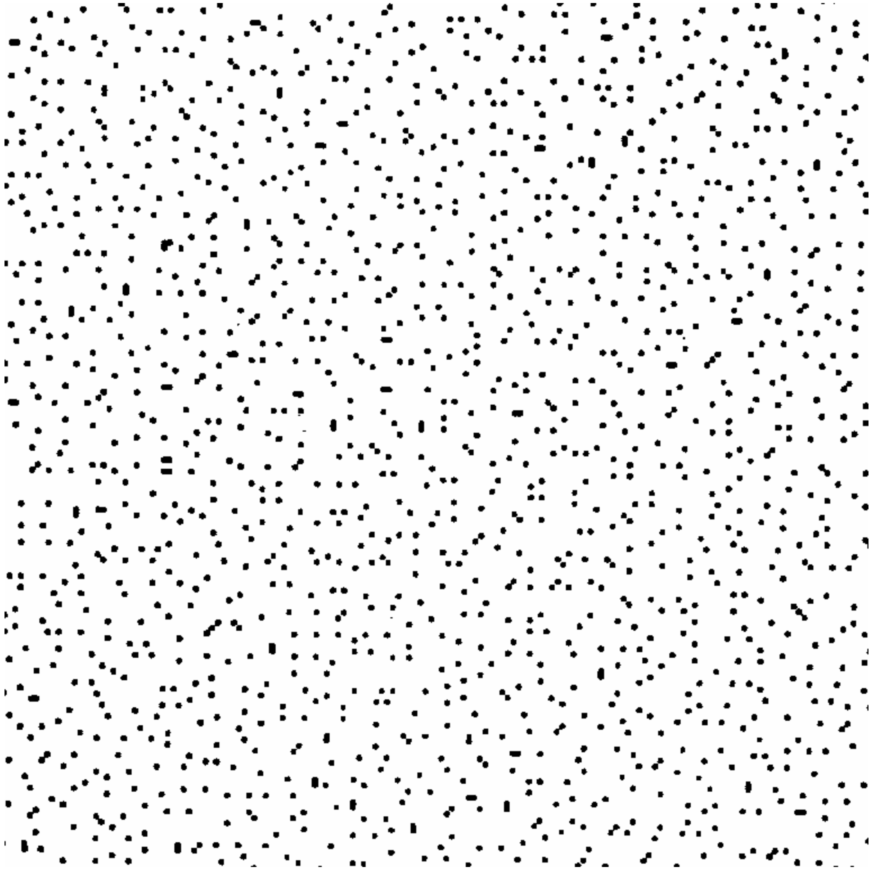
Antialiasing (nonuniform sampling)

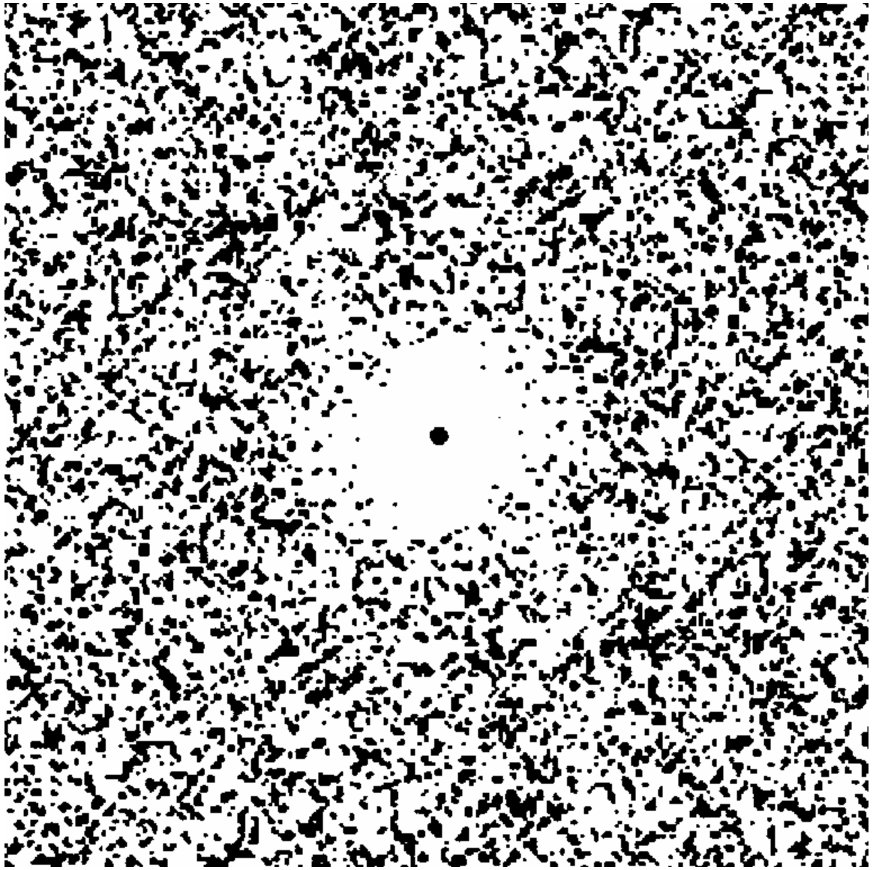
- The impulse train is modified as

$$\sum_{i=-\infty}^{\infty} \delta\left(x - \left(iT + \frac{1}{2} - \xi \right) \right)$$

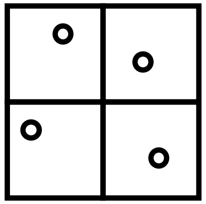
- It turns regular aliasing into noise. But random noise is less distracting than coherent aliasing.

Jittered Sampling

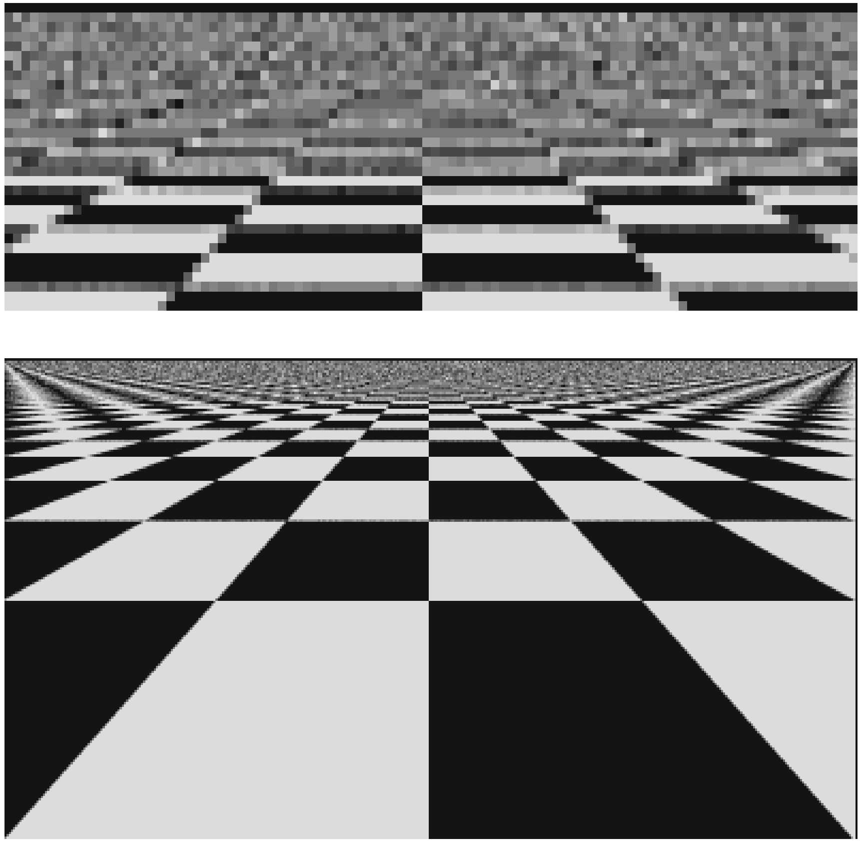


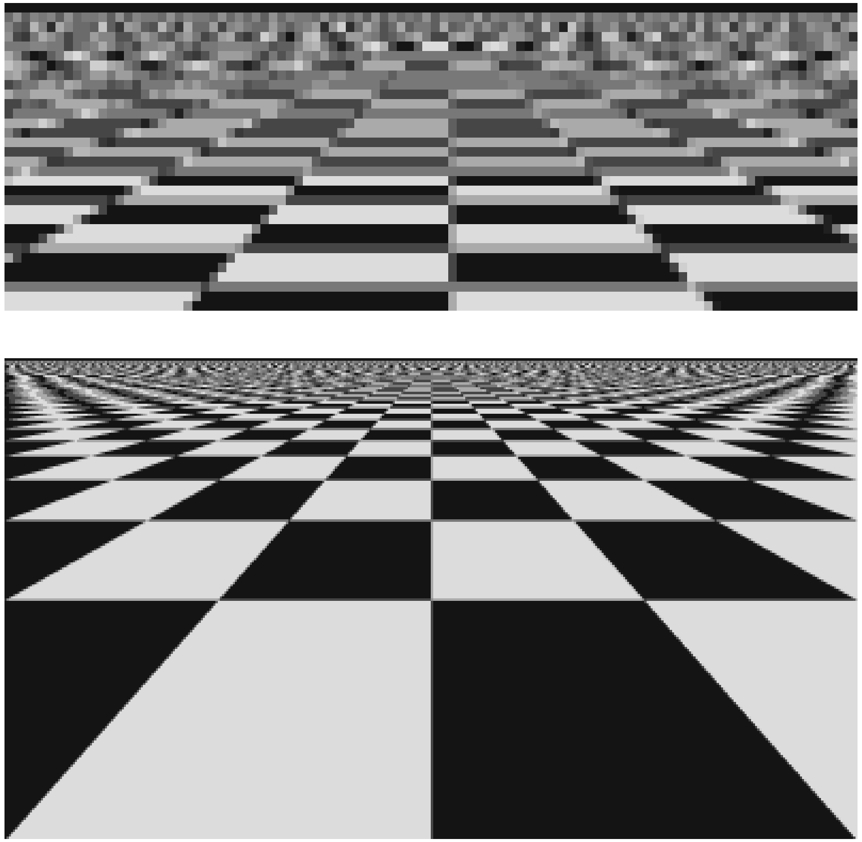


Add uniform random jitter to each sample



Jittered vs. Uniform Supersampling





4x4 Jittered Sampling **4x4 Uniform**

Antialiasing (adaptive sampling)

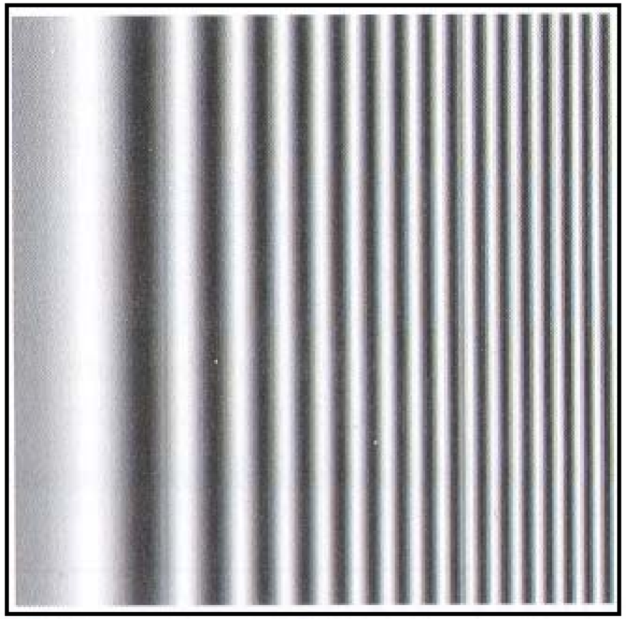
- Take more samples only when necessary. However, in practice, it is hard to know where we need supersampling. Some heuristics could be used.
- It makes a less aliased image, but may not be more efficient than simple supersampling particular for complex scenes.

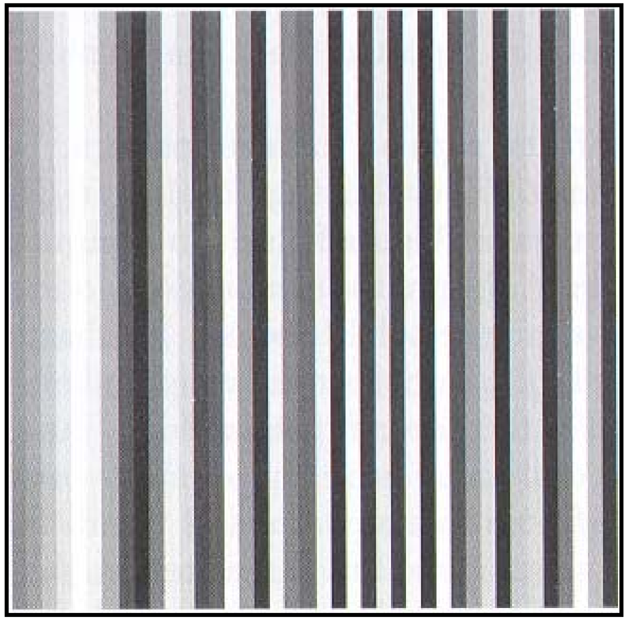
Application to ray tracing

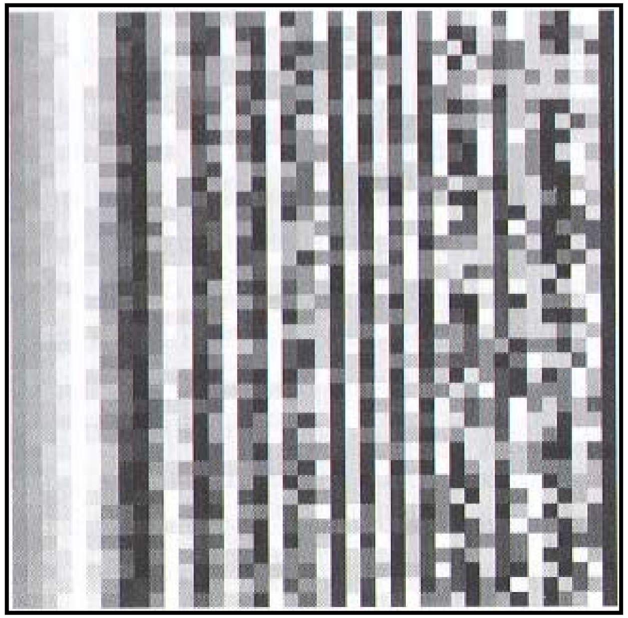
- Sources of aliasing: object boundary, small objects, textures and materials
- Good news: we can do sampling easily
- Bad news: we can't do prefiltering
- Key insight: we can never remove all aliasing, so we develop techniques to mitigate its impact on the quality of the final image.

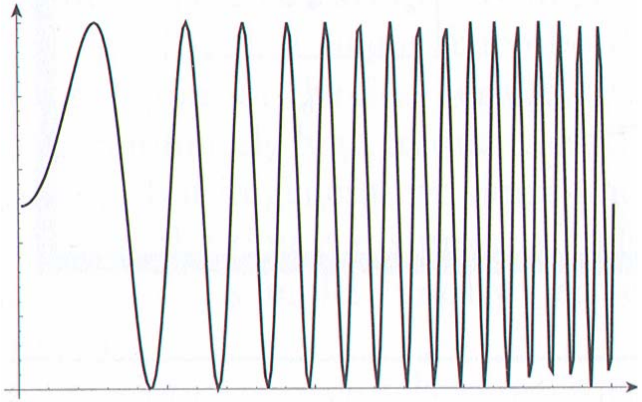
Prefer noise over aliasing

reference *aliasing* *noise*





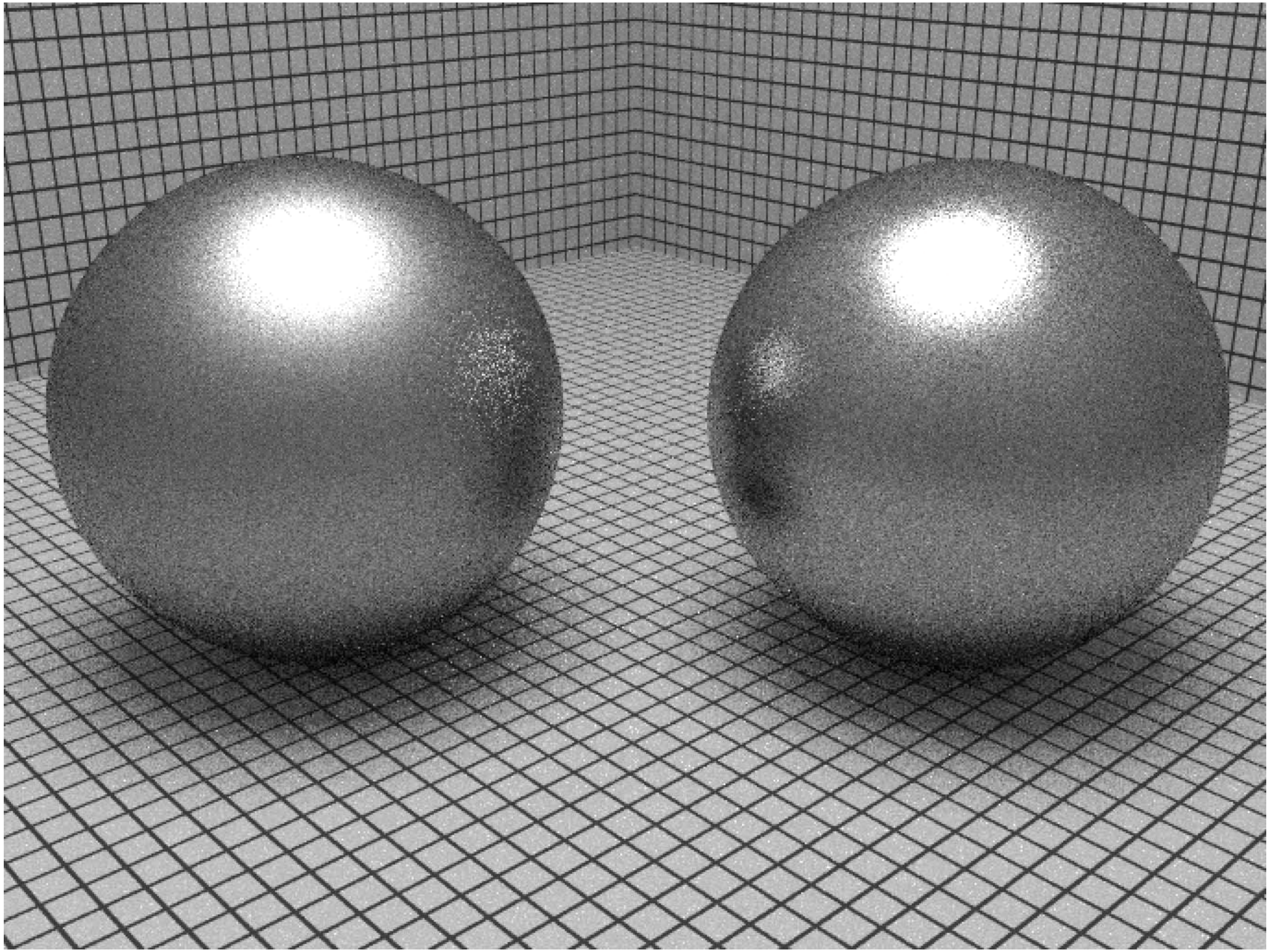




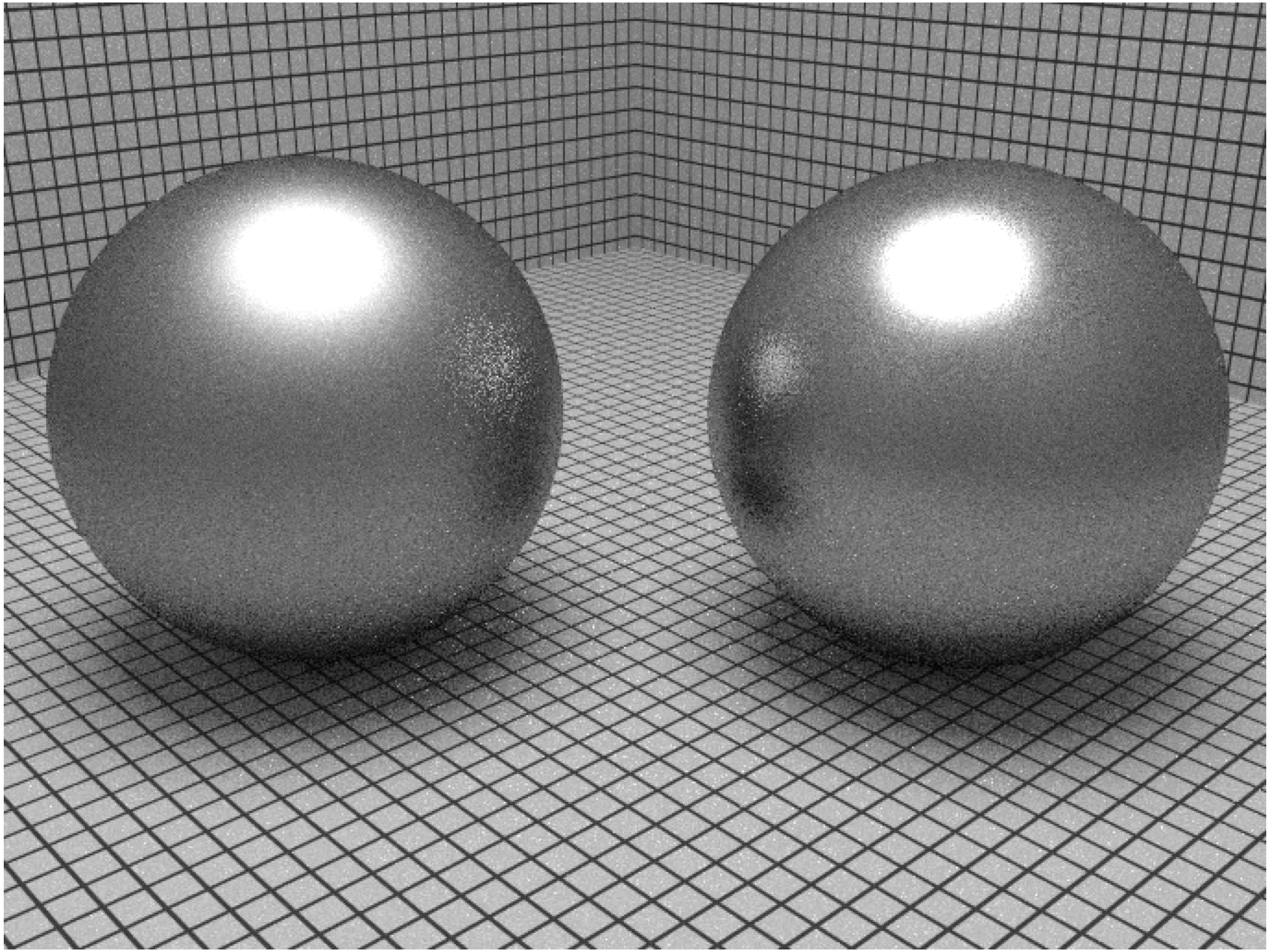
`pbrt` sampling interface

- Creating good sample patterns can substantially improve a ray tracer's efficiency, allowing it to create a high-quality image with fewer rays.
- Because evaluating radiance is costly, it pays to spend time on generating better sampling.
- **`core/sampling.*, samplers/*`**
- **`random.cpp, stratified.cpp, bestcandidate.cpp, lowdiscrepancy.cpp,`**

An ineffective sampler



A more effective sampler



Sampler

```
Sampler(int xstart, int xend,
        int ystart, int yend, int spp);
bool GetNextSample(Sample *sample);
int  TotalSamples()
       samplesPerPixel *
      (xPixelEnd - xPixelStart) *
      (yPixelEnd - yPixelStart);
```

range of pixels

sample per pixel

`Render()` in `core/scene.cpp`,

used for generating eye rays

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

Sample

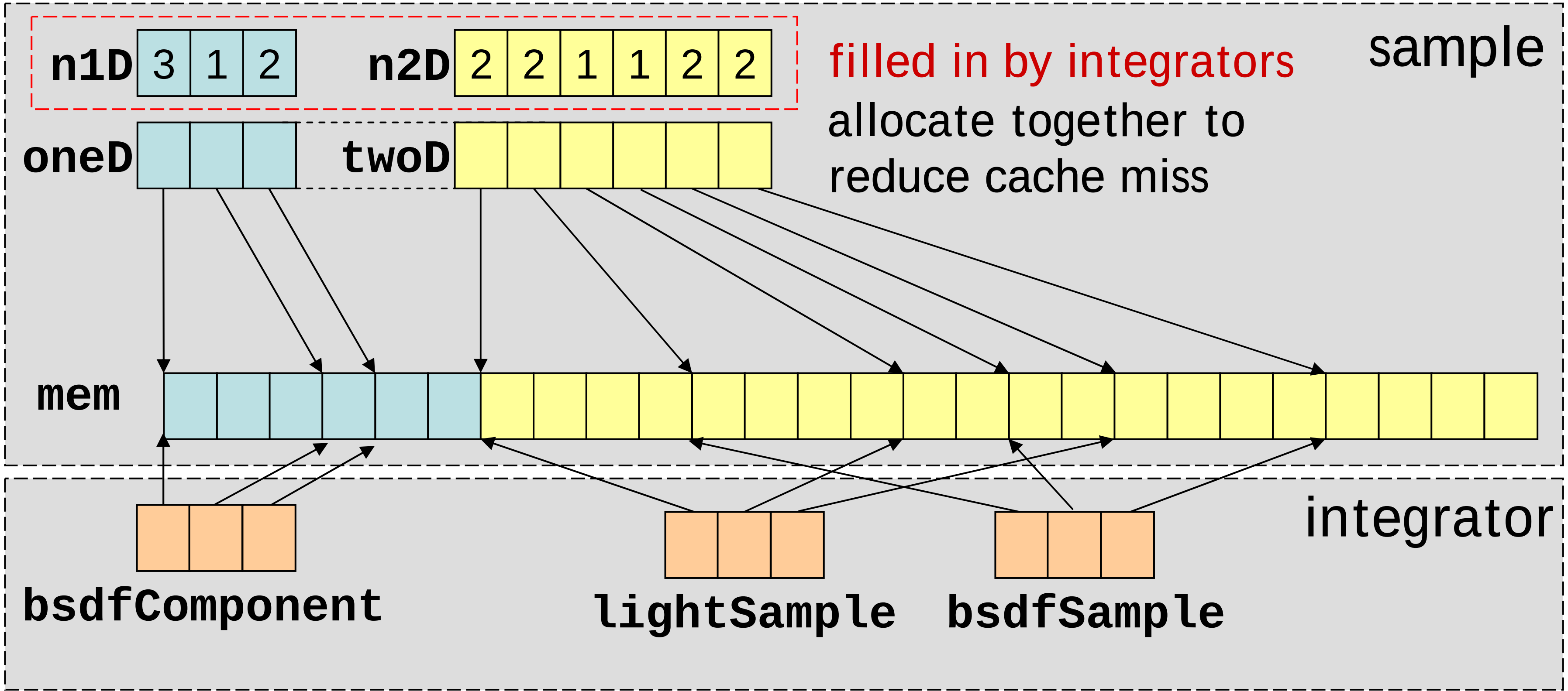
```
Struct Sample {
  Sample(SurfaceIntegrator *surf,
         VolumeIntegrator *vol,
         const Scene *scene);
   ...
   float imageX, imageY;
   float lensU, lensV;
   float time;
   // Integrator Sample Data
   vector<u_int> n1D, n2D;
   float **oneD, **twoD;
   ...
}
```

store required information for one eye ray sample

Sample is allocated once in Render(). Sampler is called to fill in the information for each eye ray. The integrator can ask for multiple 1D and/or 2D samples, each with an arbitrary number of entries, e.g. depending on #lights.

Date structure

- Different types of lights require different number of samples, usually 2D samples.
- Sampling BRDF requires 2D samples.
- Selection of BRDF components requires 1D samples.



Sample

```
Sample::Sample(SurfaceIntegrator *surf,
      VolumeIntegrator *vol, const Scene *scene) {
  // calculate required number of samples
  // according to integration strategy
  surf->RequestSamples(this, scene);
  vol->RequestSamples(this, scene);

  // Allocate storage for sample pointers
  int nPtrs = n1D.size() + n2D.size();
  if (!nPtrs) {
      oneD = twoD = NULL;
      return;
  }
  oneD=(float **)AllocAligned(nPtrs*sizeof(float *));
  twoD = oneD + n1D.size();
```

Sample

```
    // Compute total number of sample values needed
    int totSamples = 0;
    for (u_int i = 0; i < n1D.size(); ++i)
        totSamples += n1D[i];
    for (u_int i = 0; i < n2D.size(); ++i)
        totSamples += 2 * n2D[i];
    // Allocate storage for sample values
    float *mem = (float *)AllocAligned(totSamples *
        sizeof(float));
    for (u_int i = 0; i < n1D.size(); ++i) {
        oneD[i] = mem;
        mem += n1D[i];
    }
    for (u_int i = 0; i < n2D.size(); ++i) {
        twoD[i] = mem;
        mem += 2 * n2D[i];
    }
}
```

Random sampler

Just for illustration; does not work well in practice

```
RandomSampler::RandomSampler(…) {
  ...
  // Get storage for a pixel's worth of stratified
  samples imageSamples = (float *)AllocAligned(5 *
      xPixelSamples * yPixelSamples * sizeof(float));
  lensSamples = imageSamples +
                2 * xPixelSamples * yPixelSamples;
  timeSamples = lensSamples +
                2 * xPixelSamples * yPixelSamples;

  // prepare samples for the first pixel
  for (i=0; i<5*xPixelSamples*yPixelSamples; ++i)
      imageSamples[i] = RandomFloat();
  // Shift image samples to pixel coordinates
  for (o=0; o<2*xPixelSamples*yPixelSamples; o+=2) {
      imageSamples[o]   += xPos;
      imageSamples[o+1] += yPos; }
  samplePos = 0;
}
```

Random sampler

```
bool RandomSampler::GetNextSample(Sample *sample) {
  if (samplePos == xPixelSamples * yPixelSamples) {
    // Advance to next pixel for sampling
    if (++xPos == xPixelEnd) {
      xPos = xPixelStart;
      ++yPos; }
    if (yPos == yPixelEnd)
      return false;

    for (i=0; i < 5*xPixelSamples*yPixelSamples; ++i)
      imageSamples[i] = RandomFloat();

    // Shift image samples to pixel coordinates
    for (o=0; o<2*xPixelSamples*yPixelSamples; o+=2)
    { imageSamples[o]   += xPos;
      imageSamples[o+1] += yPos; }
    samplePos = 0;
  }
```

number of generated samples in this pixel

generate all samples for one pixel at once

Random sampler

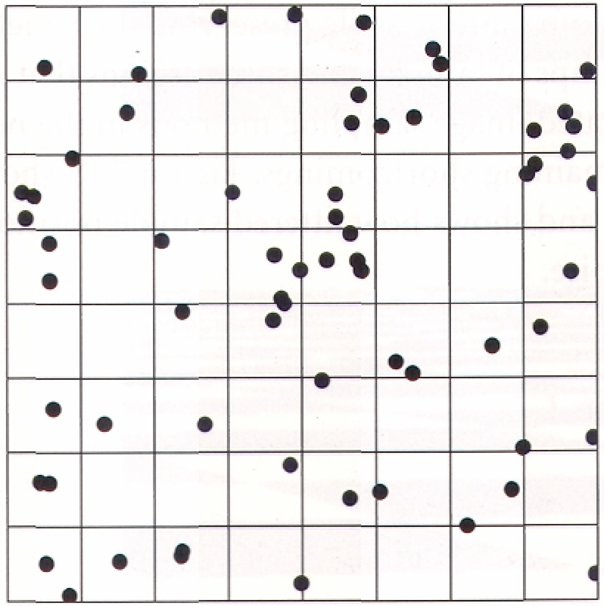
```
    // Return next sample point according to samplePos
    sample->imageX = imageSamples[2*samplePos];
    sample->imageY = imageSamples[2*samplePos+1];
    sample->lensU = lensSamples[2*samplePos];
    sample->lensV = lensSamples[2*samplePos+1];
    sample->time = timeSamples[samplePos];

    // Generate samples for integrators
    for (u_int i = 0; i < sample->n1D.size(); ++i)
        for (u_int j = 0; j < sample->n1D[i]; ++j)
              sample->oneD[i][j] = RandomFloat();
    for (u_int i = 0; i < sample->n2D.size(); ++i)
        for (u_int j = 0; j < 2*sample->n2D[i]; ++j)
              sample->twoD[i][j] = RandomFloat();

    ++samplePos;
    return true;
}
```

Random sampling

a pixel

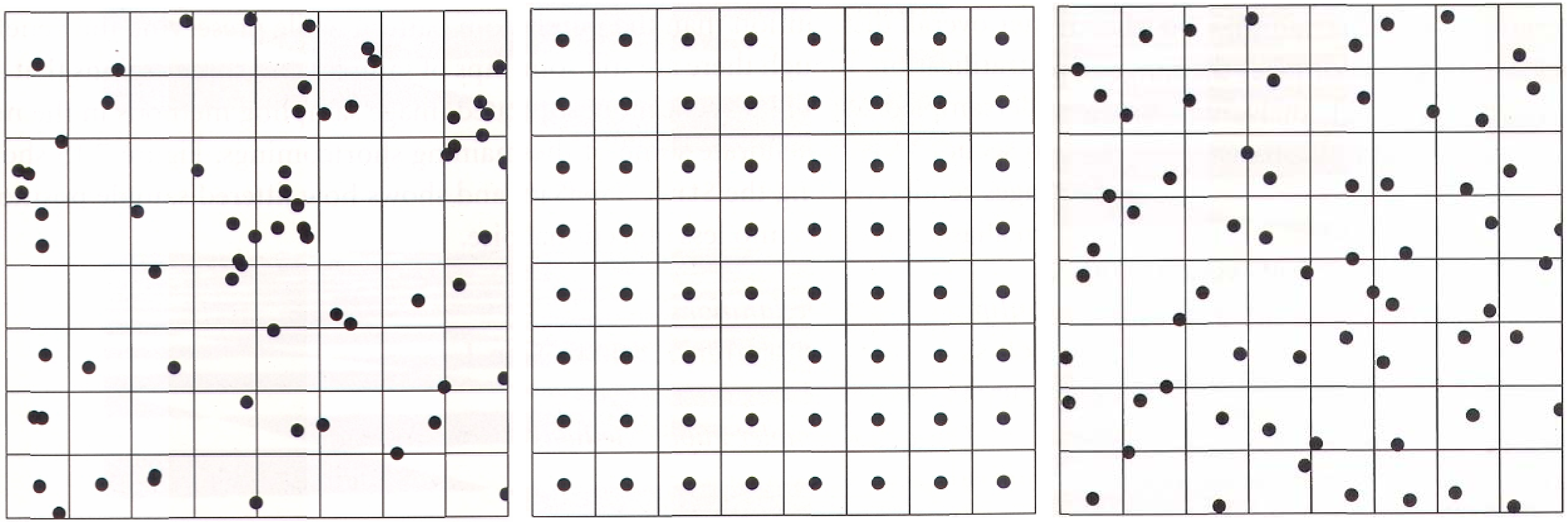


completely random

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

turn aliasing into noise

Comparison of sampling methods

256 samples per pixel as reference

1 sample per pixel (no jitter)

Comparison of sampling methods

1 sample per pixel (jittered)

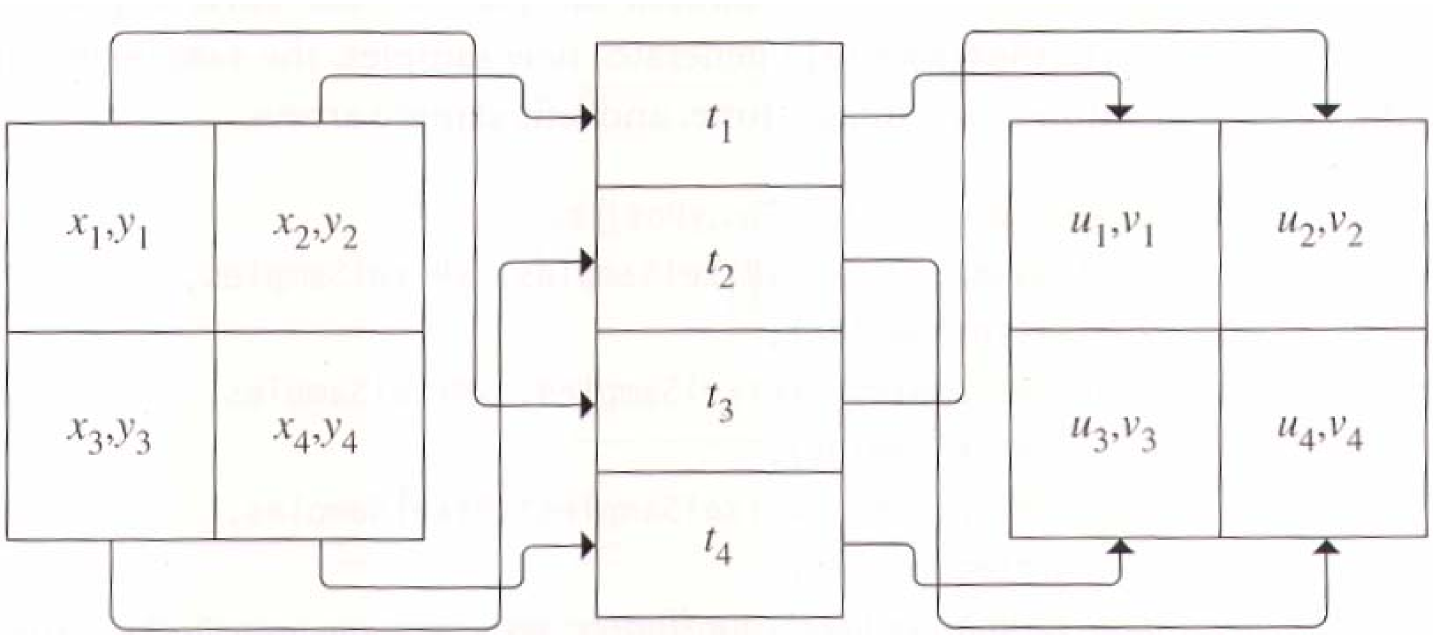
4 samples per pixel (jittered)

Stratified sampling

reference *random* *stratified jittered*

High dimension

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



Stratified sampler

```
if (samplePos == xPixelSamples * yPixelSamples) {
  // Advance to next pixel for stratified sampling
  ...
  // Generate stratified samples for (xPos, yPos)
  StratifiedSample2D(imageSamples,
    xPixelSamples, yPixelSamples, jitterSamples);
  StratifiedSample2D(lensSamples,
    xPixelSamples, yPixelSamples, jitterSamples);
  StratifiedSample1D(timeSamples,
    xPixelSamples*yPixelSamples, jitterSamples);

  // Shift stratified samples to pixel coordinates
  ...
  // Decorrelate sample dimensions
  Shuffle(lensSamples,xPixelSamples*yPixelSamples,2);
  Shuffle(timeSamples,xPixelSamples*yPixelSamples,1);
  samplePos = 0;
}
```

Stratified sampling

n stratified samples within [0..1]

```
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;
   }
}
```

nx*ny stratified samples within [0..1]X[0..1]

```
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;
       }
}
```

Shuffle

```
void Shuffle(float *samp, int count, int dims) {
  for (int i = 0; i < count; ++i) {
    u_int other = RandomUInt() % count;
    for (int j = 0; j < dims; ++j)
      swap(samp[dims*i + j], samp[dims*other + j]);
  }
}
```

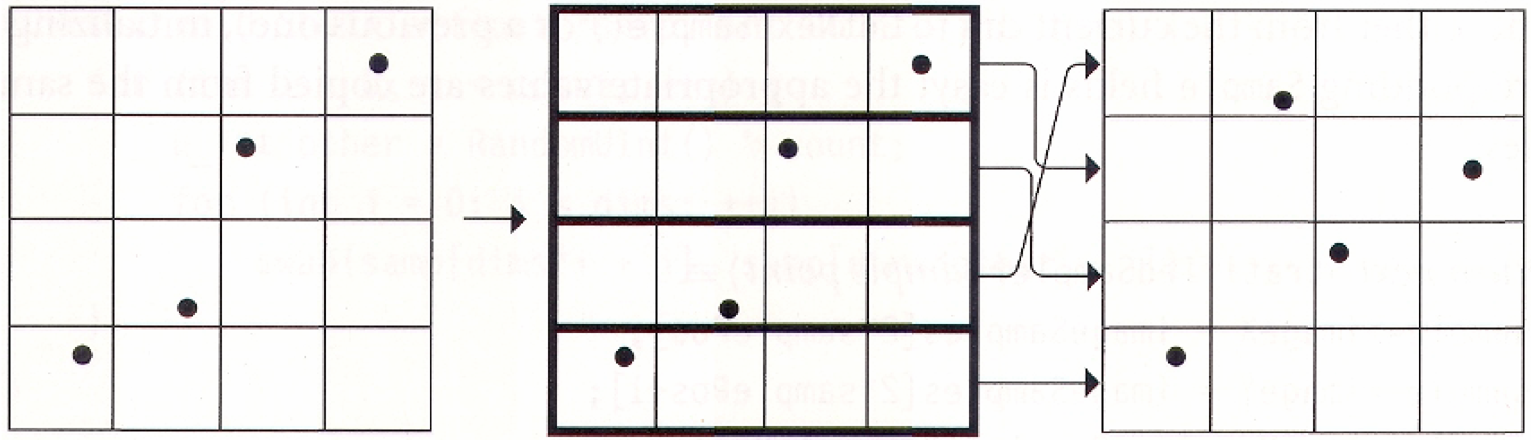
Stratified sampler

```
// Return next _StratifiedSampler_ sample point
sample->imageX = imageSamples[2*samplePos];
sample->imageY = imageSamples[2*samplePos+1];
sample->lensU = lensSamples[2*samplePos];
sample->lensV = lensSamples[2*samplePos+1];
sample->time = timeSamples[samplePos];
// what if integrator asks for 7 stratified 2D samples
// Generate stratified samples for integrators
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;
```

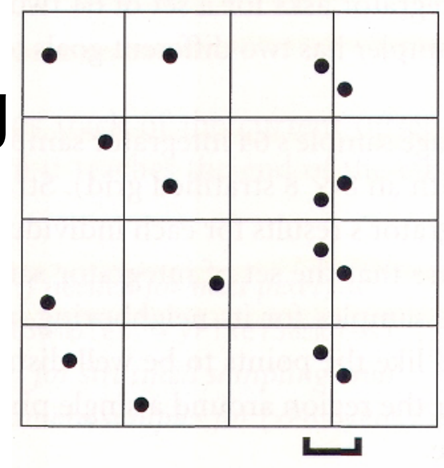
Latin hypercube sampling

- Integrators could request an arbitrary n samples. nx1 or 1xn doesn't give a good



A worst case for stratified sampling

LHS can prevent this to happen



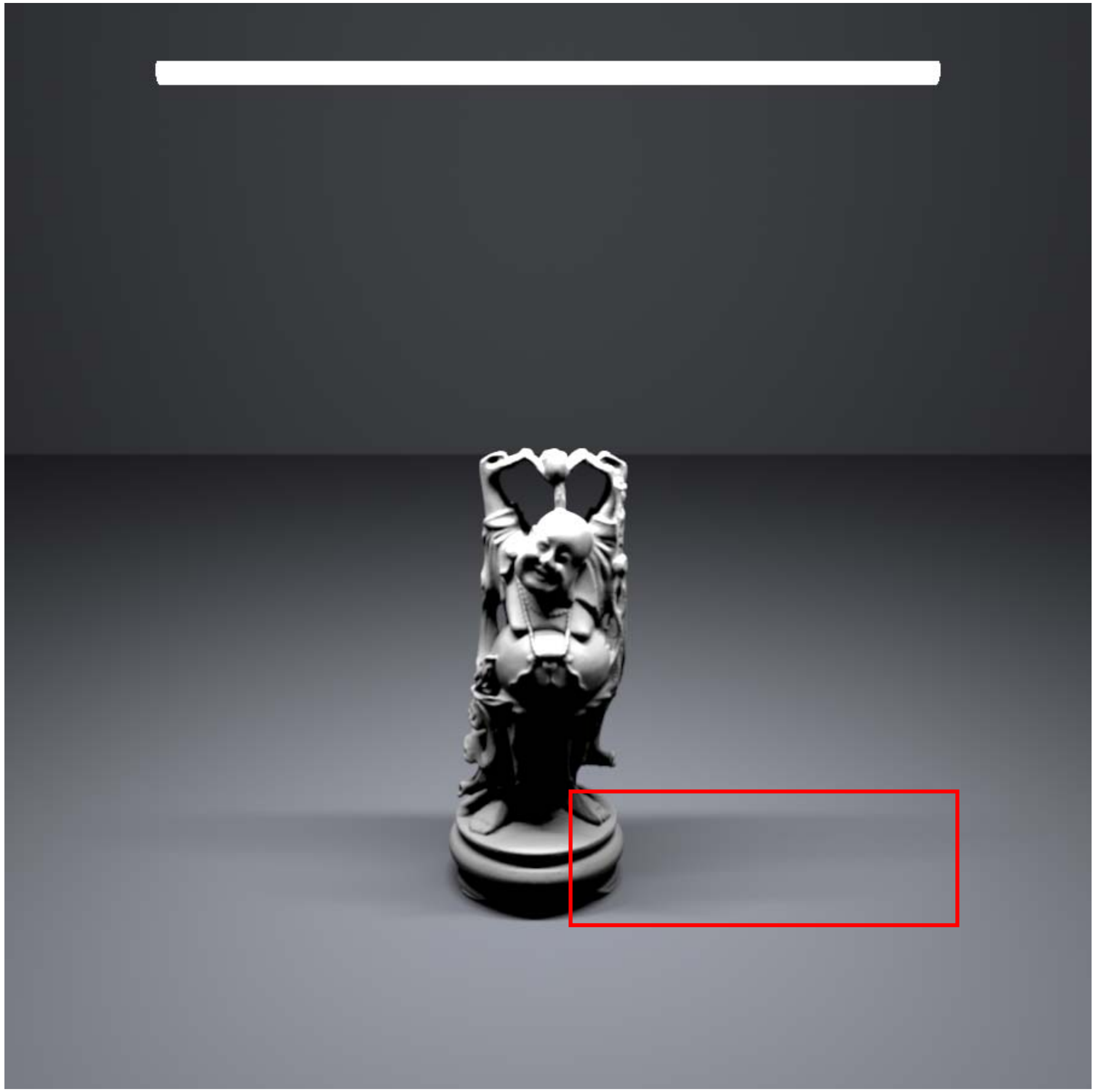
Latin Hypercube

```
void LatinHypercube(float *samples,
                    int nSamples, int nDim)
{
  // Generate LHS samples along diagonal
  float delta = 1.f / nSamples;
  for (int i = 0; i < nSamples; ++i)
    for (int j = 0; j < nDim; ++j)
      samples[nDim*i+j] = (i+RandomFloat())*delta;
```

note the difference with **`shuffle`**

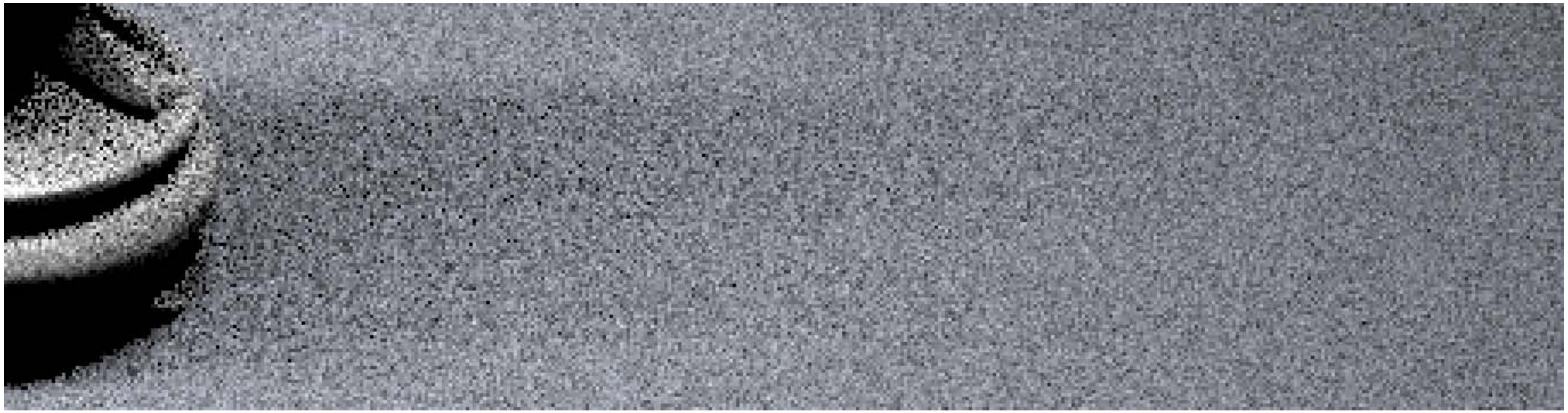
```
  // Permute LHS samples in each dimension
  for (int i = 0; i < nDim; ++i) {
    for (int j = 0; j < nSamples; ++j) {
      u_int other = RandomUInt() % nSamples;
      swap(samples[nDim * j + i],
           samples[nDim * other + i]);
      }
  }
}
```

Stratified sampling



Stratified sampling

1 camera sample and 16 shadow samples per pixel



16 camera samples and each with 1 shadow sample per pixel

Low discrepancy sampling

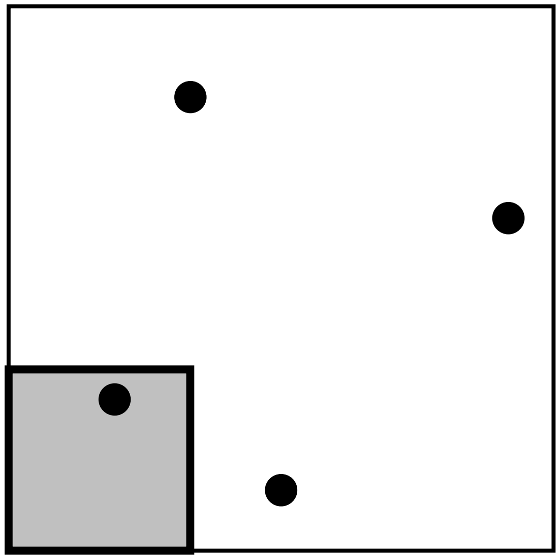
a family of shapes

set of N sample points

$$D_N(B,P) = \sup_{b\in B} \left| \boxed{\frac{\#\{x_i \in b\}}{N}} - \boxed{Vol(b)} \right|$$

volume estimated by sample number

real volume



When B is the set of AABBs with a corner at the origin, this is called star discrepancy

$D_N^*(P)$

1D discrepancy

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

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

$$x_i = general \quad \Rightarrow \quad D_N^*(x_1,\ldots,x_n) = \frac{1}{2N} + \max_{1\le i\le N}\left| x_i - \frac{2i-1}{2N} \right|$$

Uniform is optimal! However, we have learnt that Irregular patterns are perceptually superior to uniform samples. Fortunately, for higher dimension, the low-discrepancy patterns are less uniform and works reasonably well as sample patterns in practice.

Radical inverse

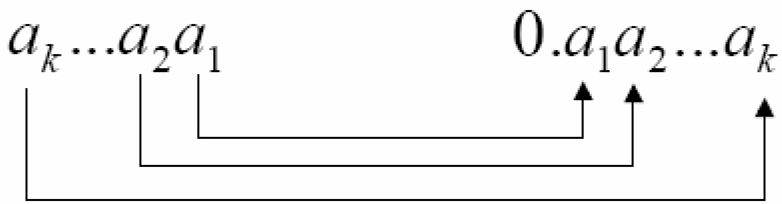
- A positive number n can be expressed in a base b as

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

- A radical inverse function in base b converts a nonnegative integer n to a floating-point number in *[0,1)*

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

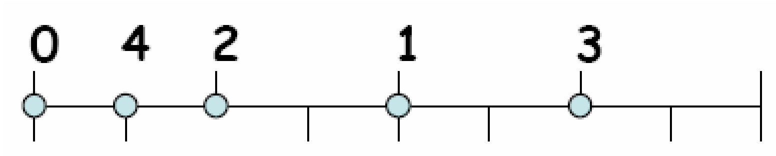
```
inline double RadicalInverse(int n, int base) {
  double val = 0;
  double invBase = 1. / base, invBi = invBase;
  while (n > 0) {
    int d_i = (n % base);
    val += d_i * invBi;
    n /= base;
    invBi *= invBase;
  }
  return val;
}
```



van der Corput sequence

- The simplest sequence $x_i = \Phi_2(i)$
- Recursively split 1D line in half, sample centers
- Achieve minimal possible discrepancy

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



i	binary form of i	radical inverse	x_i
0	0	0.0	0
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

Halton sequence

- Use relatively prime numbers as bases for each dimension

recursively split the dimension into p_d parts, sample centers

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

- Achieve best possible discrepancy for N-D

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

- Can be used if N is not known in advance
- All prefixes of a sequence are well distributed so as additional samples are added to the sequence, low discrepancy will be maintained

Hammersley sequence

- Similar to Halton sequence.
- Slightly better discrepancy than Halton.
- Needs to know N in advance.

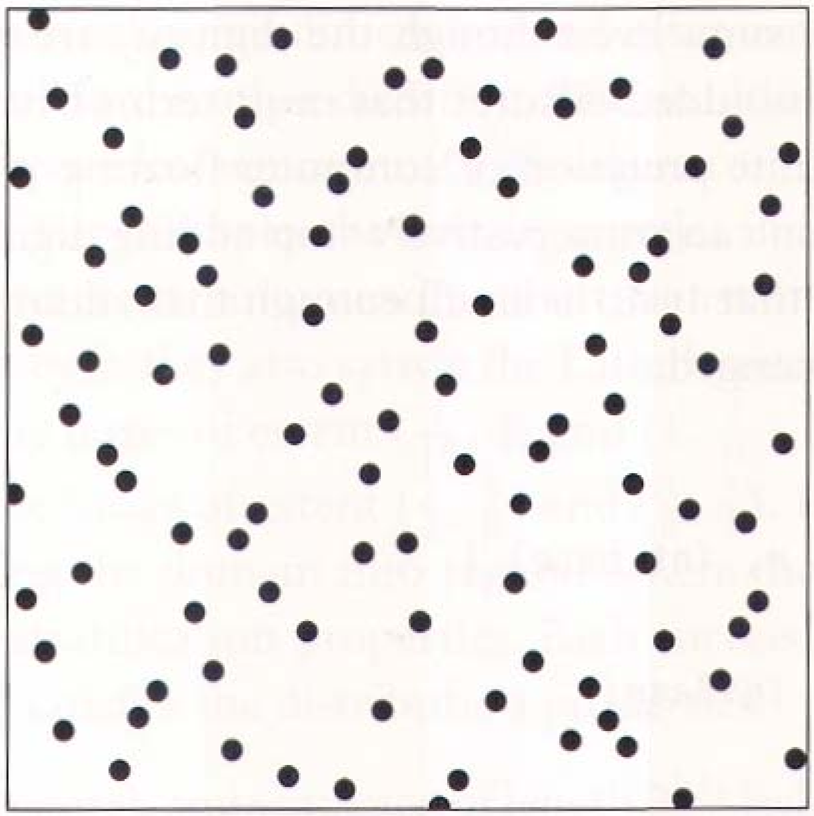
$$x_i = \left(\frac{i-1/2}{N}, \Phi_{b_1}(i), \Phi_{b_2}(i), \ldots, \Phi_{b_{d-1}}(i)\right)$$

Folded radical inverse

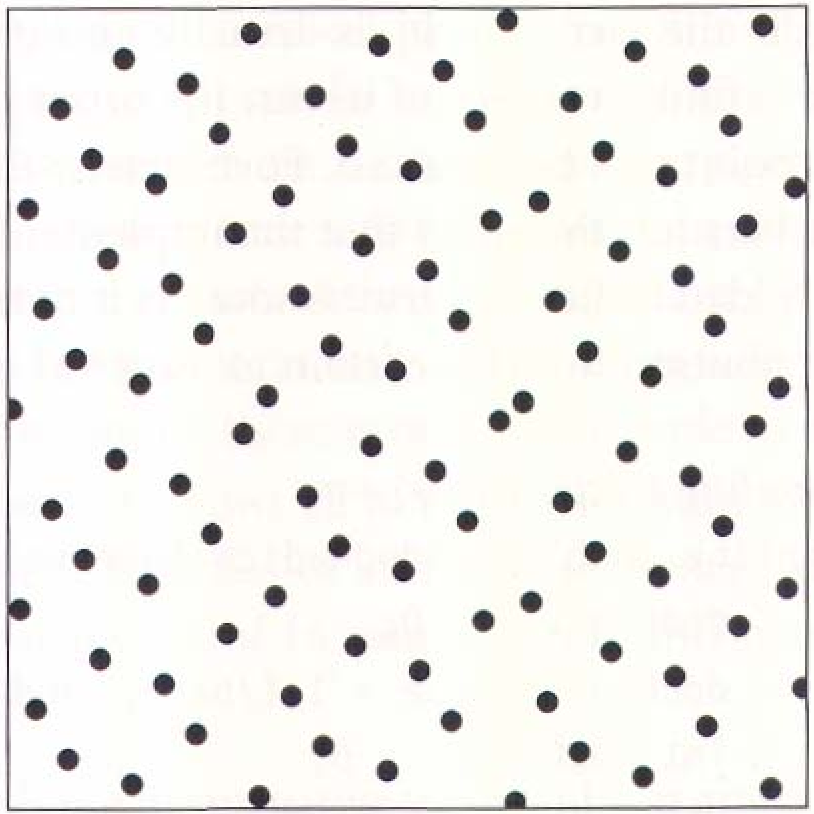
- It can be used to improve Hammersley and Halton, called Hammersley-Zaremba and Halton-Zaremba.

$$\Phi_b(n) = \sum_{i=1}^{\infty} ((a_i + i - 1) \bmod b) \frac{1}{b^i}$$

Radial inverse



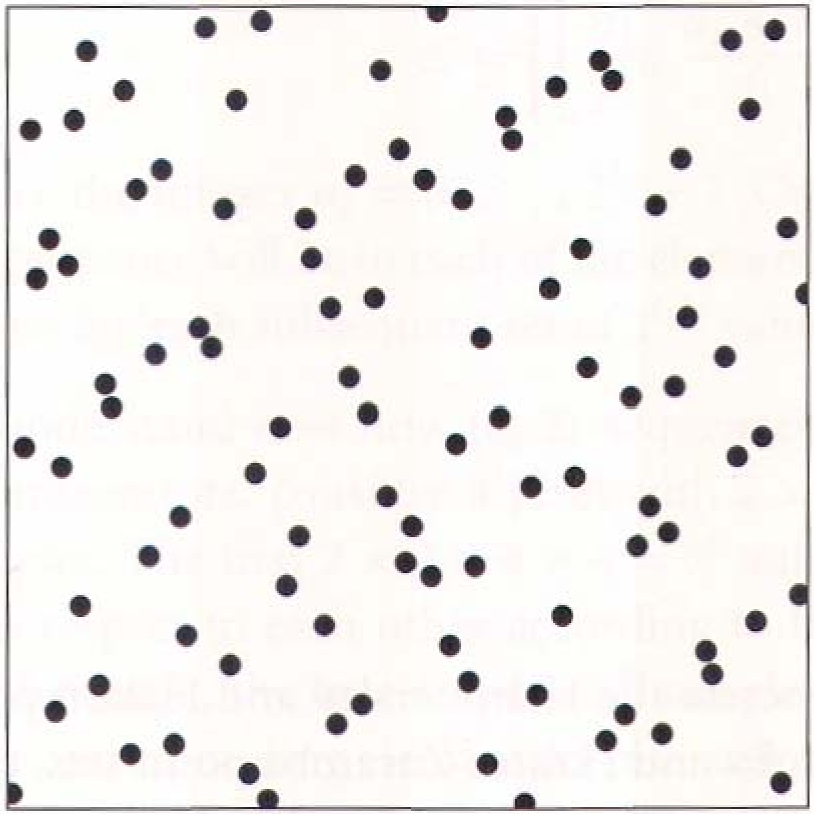
Halton



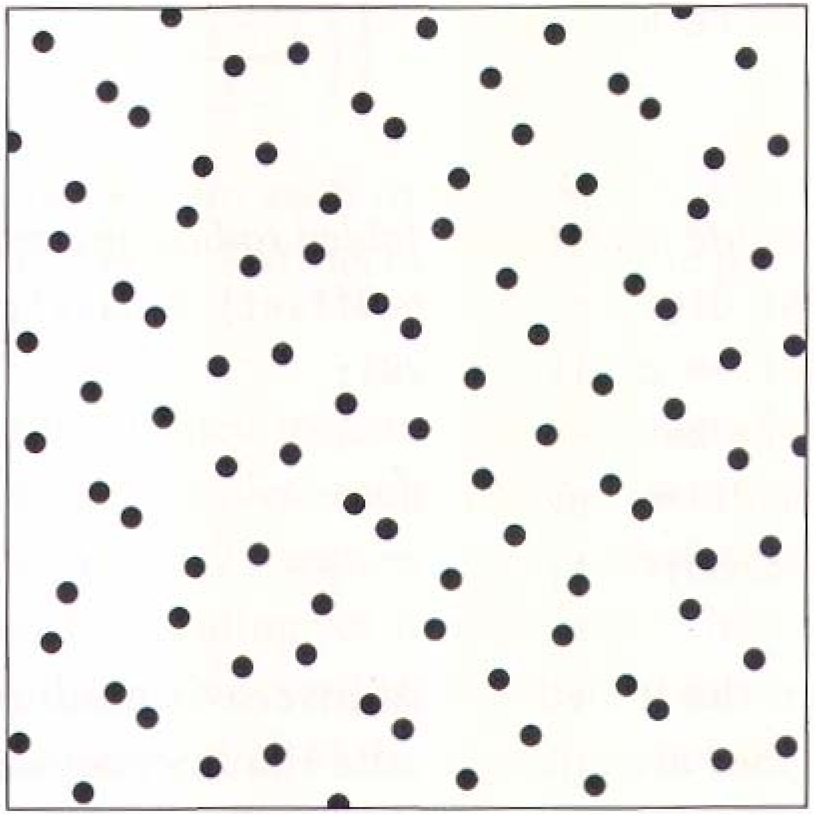
Hammersley

Better for that there are fewer clumps.

Folded radial inverse



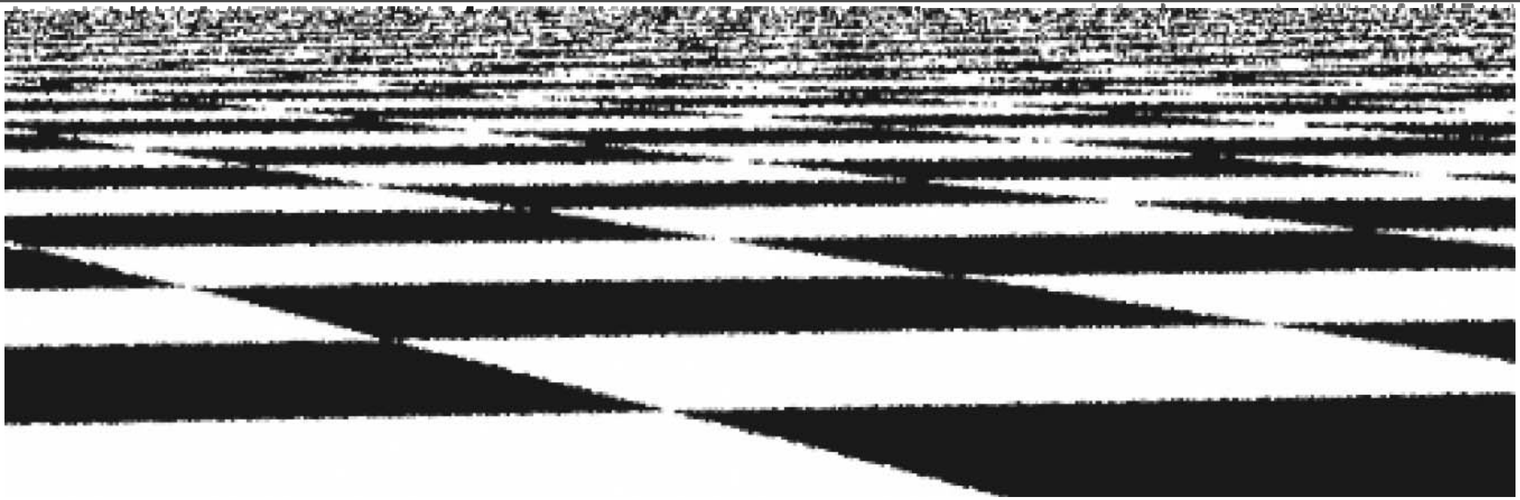
Halton



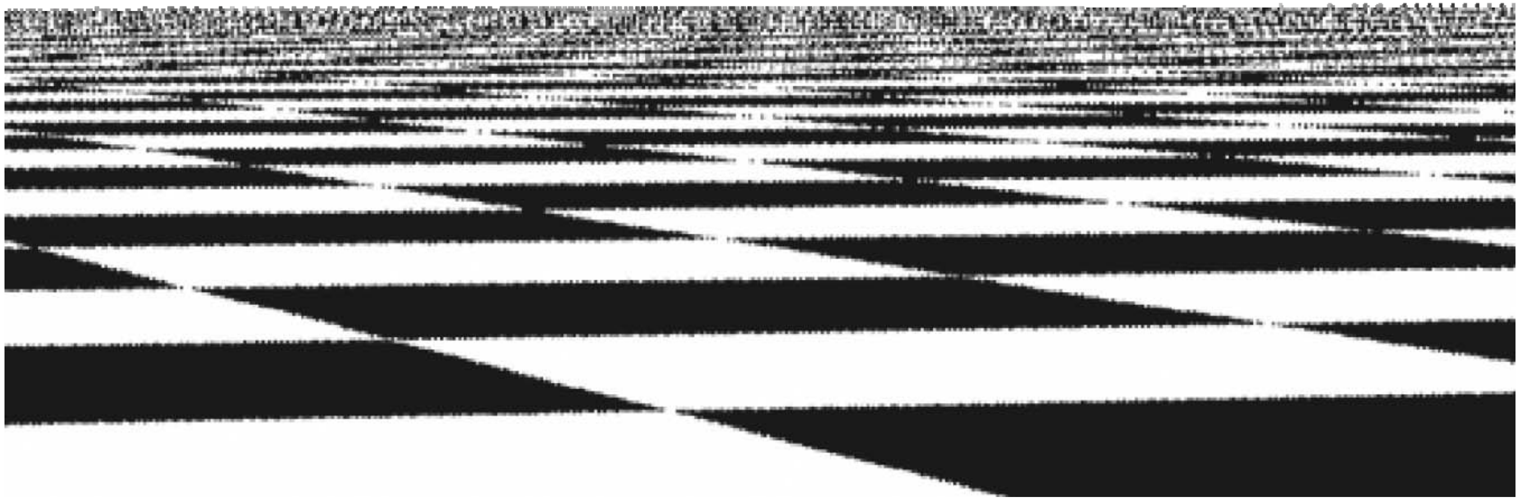
Hammersley

The improvement is more obvious

Low discrepancy sampling



stratified jittered, 1 sample/pixel

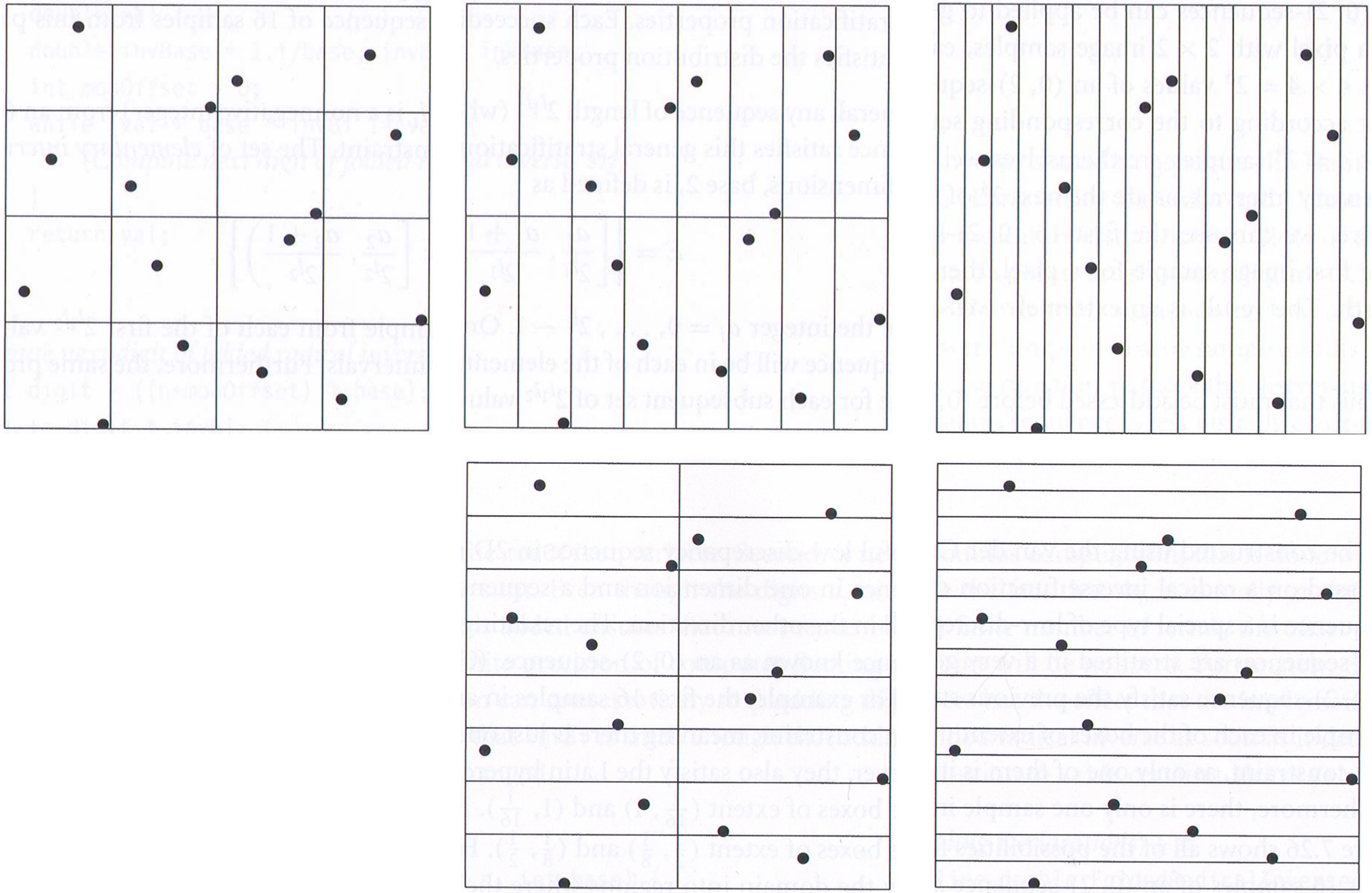


Hammersley sequence, 1 sample/pixel

(0,2)-sequences

- A useful low-discrepancy sequence in 2D is to use the van der Corput sequence in one dimension and a Sobol sequence in the other.
- It is stratified in a very general way.
- To generate different sequences for different pixels, pbrt scrambles the (0,2)-sequence by permuting the original sequence.
- Divide the square into half, swap two halves with 50% probability. Repeat until below numerical precision.

(0,2)-sequences



Implementation of (0,2)-sequences

- We use binary base; scramble equals XOR
- Assume the same scramble decision for the same level

(0,2)-sequences

```
void Sample02(u_int n, u_int scramble[2],
                       float sample[2]) {
  sample[0] = VanDerCorput(n, scramble[0]);
  sample[1] = Sobol2(n, scramble[1]);
}
float VanDerCorput(u_int n, u_int scramble) {
  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 (float)n / (float)0x100000000LL;
}
float Sobol2(u_int n, u_int scramble) {
  for (u_int v = 1<<31; n != 0; n >>= 1, v ^= v >> 1)
      if (n & 0x1) scramble ^= v;
  return (float)scramble / (float)0x100000000LL;
}
```

LDSampler

- pbrt uses (0,2)-sequence instead of Hammersley because it is prone to aliasing.
- LDSampler uses (0,2)-sequences for position and lens, van der Corput with scramble for time.

```
// Generate low-discrepancy samples for pixel
LDShuffleScrambled2D(1, pixelSamples, imageSamples);
LDShuffleScrambled2D(1, pixelSamples, lensSamples);
LDShuffleScrambled1D(1, pixelSamples, timeSamples);
for (u_int i = 0; i < sample->n1D.size(); ++i)
  LDShuffleScrambled1D(sample->n1D[i], pixelSamples,
                                   oneDSamples[i]);
for (u_int i = 0; i < sample->n2D.size(); ++i)
  LDShuffleScrambled2D(sample->n2D[i], pixelSamples,
                                   twoDSamples[i]);
```

copy to oneD and twoD of Sample

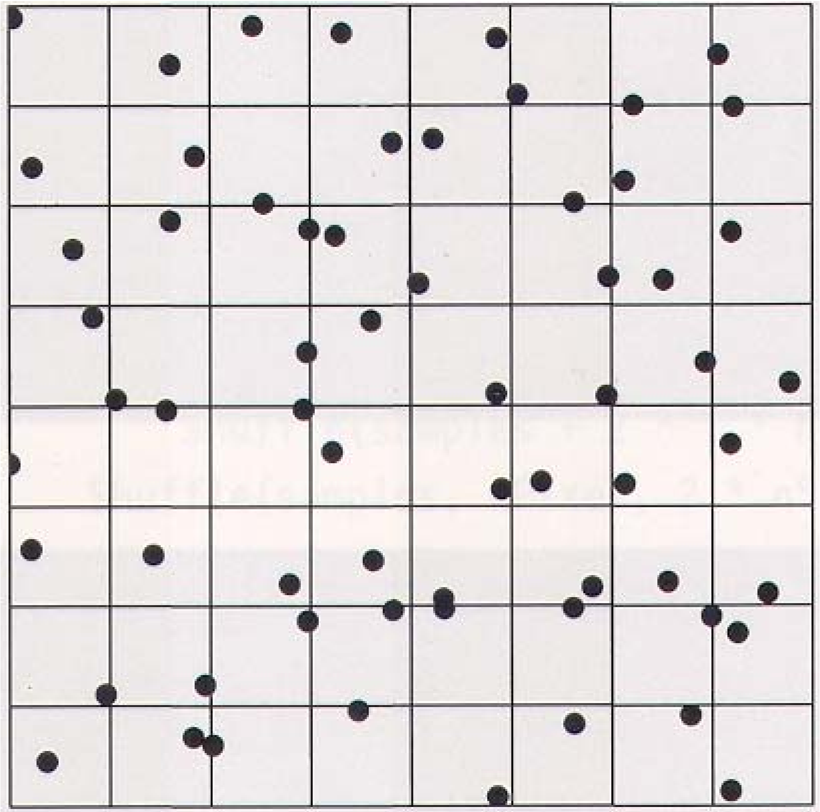
LDSampler

```
void LDShuffleScrambled1D(int nSamples, int nPixel,
                          float *samples) {
  u_int scramble = RandomUInt();
  for (int i = 0; i < nSamples * nPixel; ++i)
    samples[i] = VanDerCorput(i, scramble);
  for (int i = 0; i < nPixel; ++i)
    Shuffle(samples + i * nSamples, nSamples, 1);
  Shuffle(samples, nPixel, nSamples);
}
void LDShuffleScrambled2D(int nSamples, int nPixel,
                          float *samples) {
  u_int scramble[2] = { RandomUInt(), RandomUInt() };
  for (int i = 0; i < nSamples * nPixel; ++i)
    Sample02(i, scramble, &samples[2*i]);
  for (int i = 0; i < nPixel; ++i)
    Shuffle(samples + 2 * i * nSamples, nSamples, 2);
  Shuffle(samples, nPixel, 2 * nSamples);
}
```

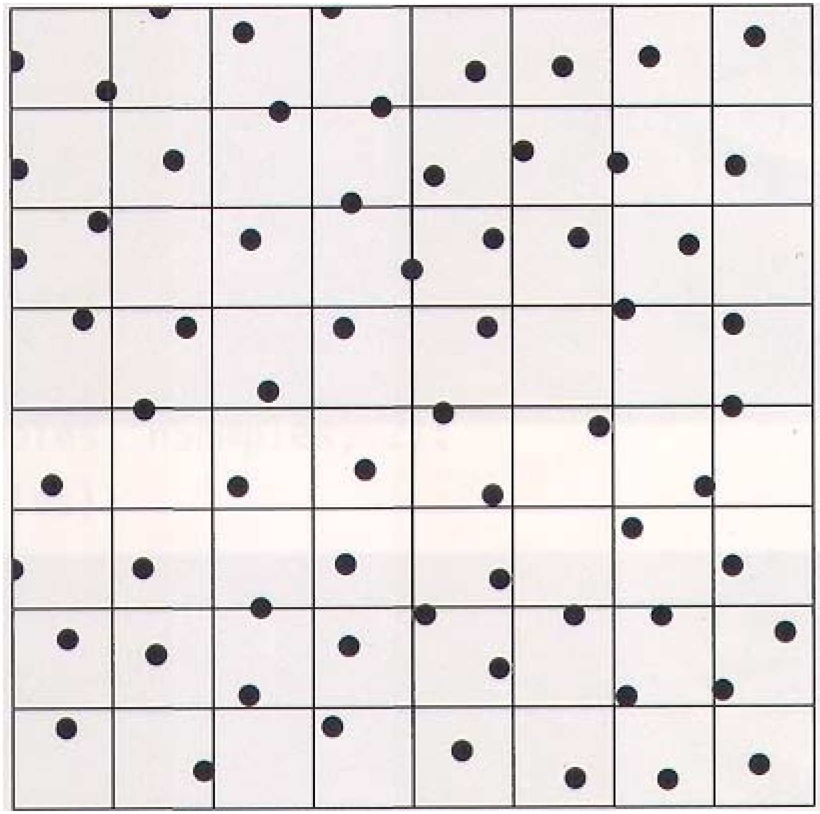
Best candidate sampling

- 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 randomly generates many candidates but only inserts the one farthest to all previous samples.

Best candidate sampling



stratified jittered



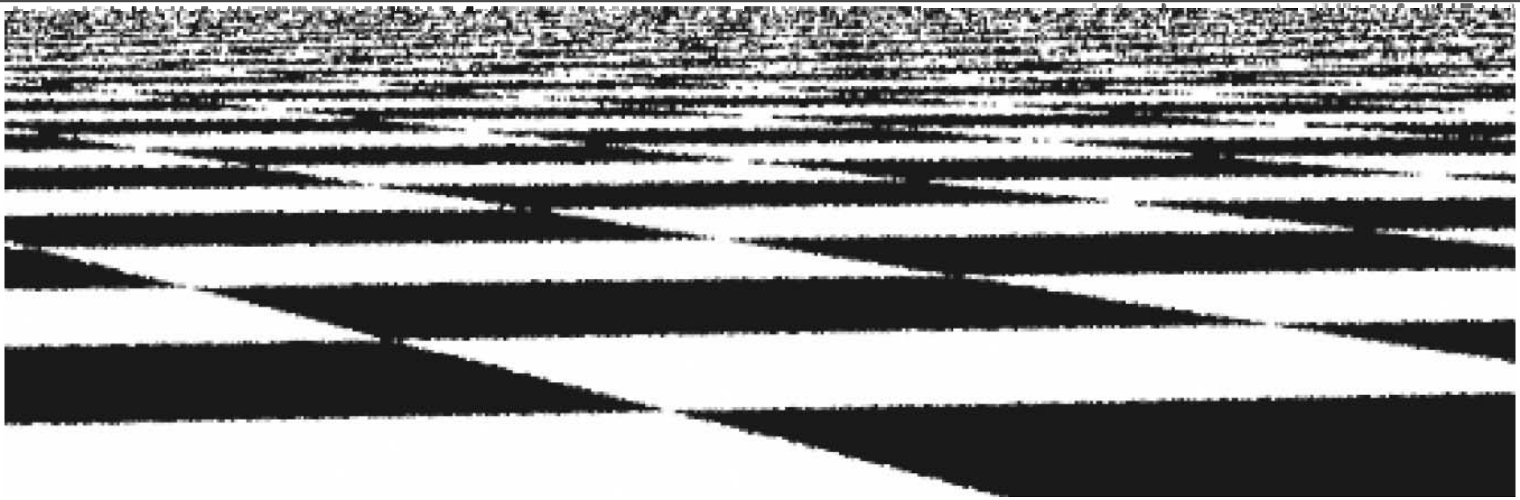
best candidate

It avoids holes and clusters.

Best candidate sampling

- Because of it is costly to generate best candidate pattern, pbrt computes a “tilable pattern” offline (by treating the square as a rolled torus).
- **`tools/samplepat.cpp`**$\rightarrow$**`sampler/sampledata.cpp`**

Best candidate sampling



stratified jittered, 1 sample/pixel

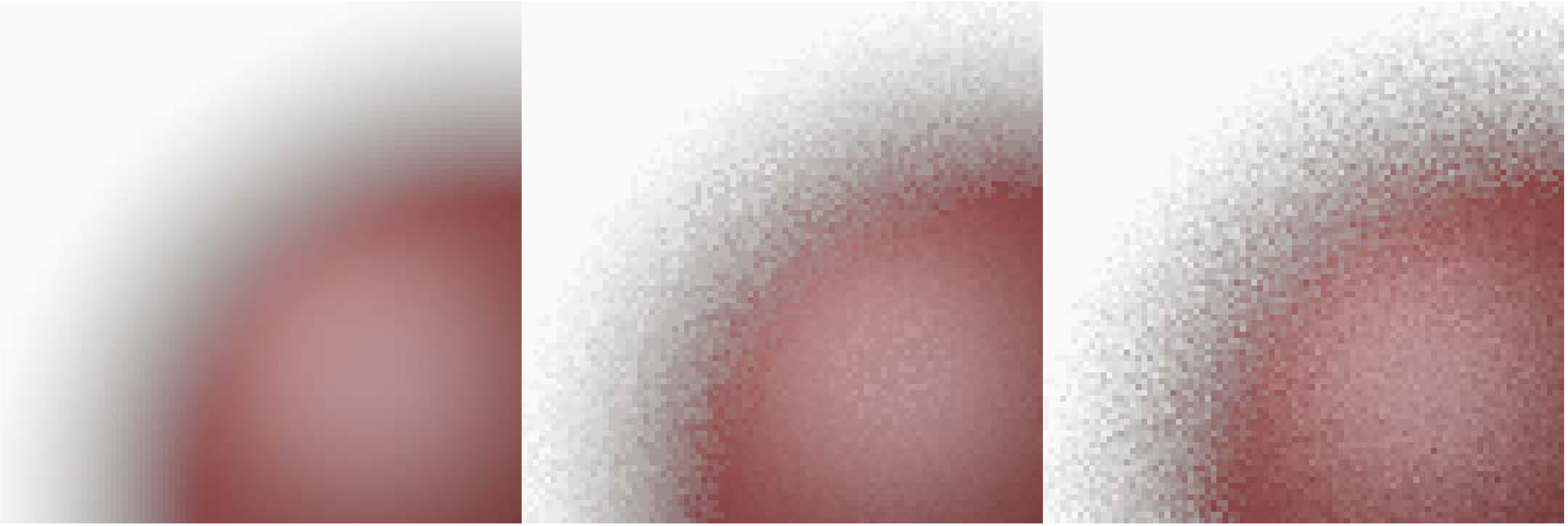
best candidate, 1 sample/pixel

Best candidate sampling

stratified jittered, 4 sample/pixel

best candidate, 4 sample/pixel

Comparisons



reference *low-discrepancy* *best candidate*

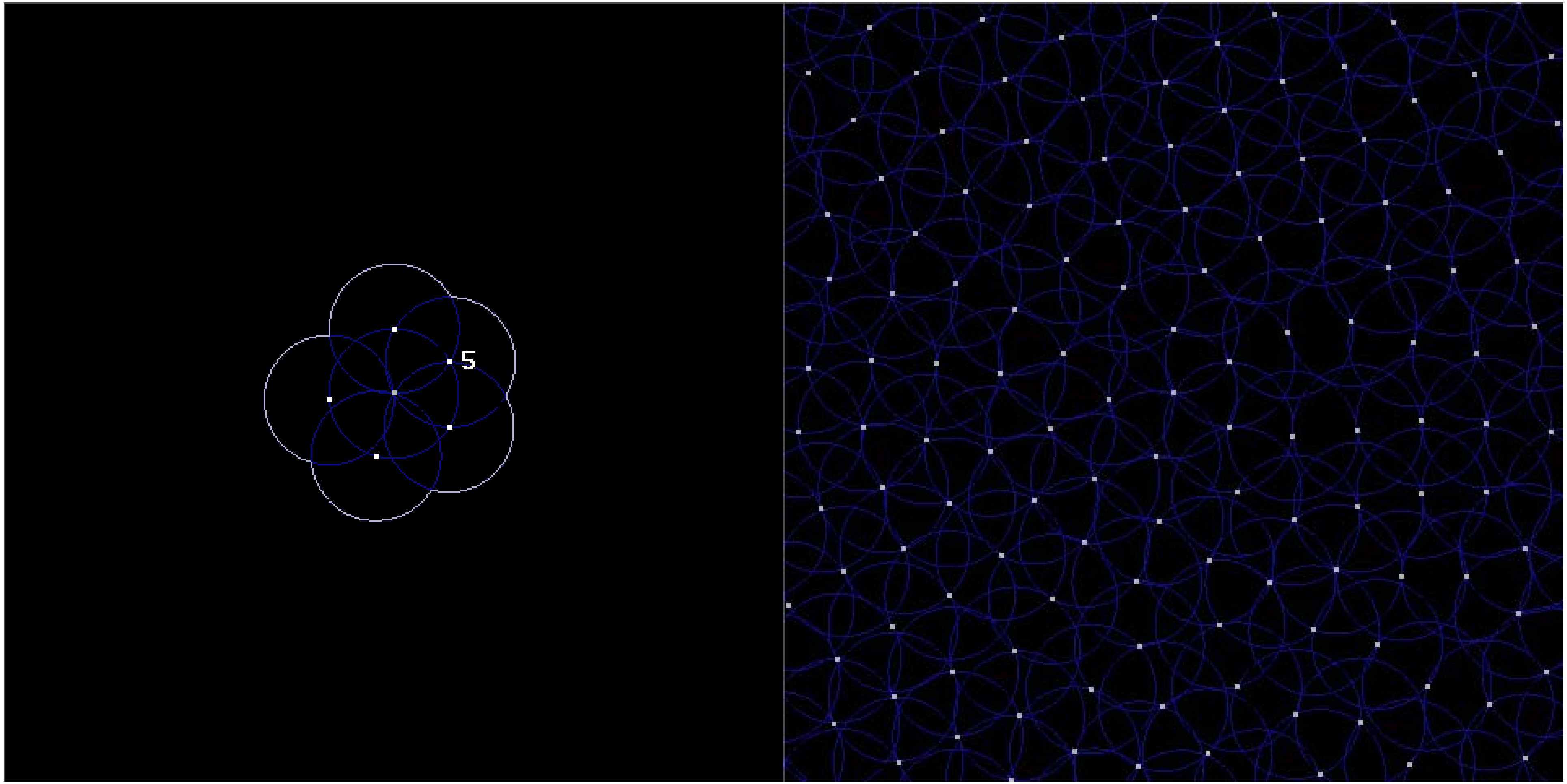
Some recent progresses

- Fast Poisson Disk Sampling
- Recursive Wang Tiles for Real-Time Blue Noise

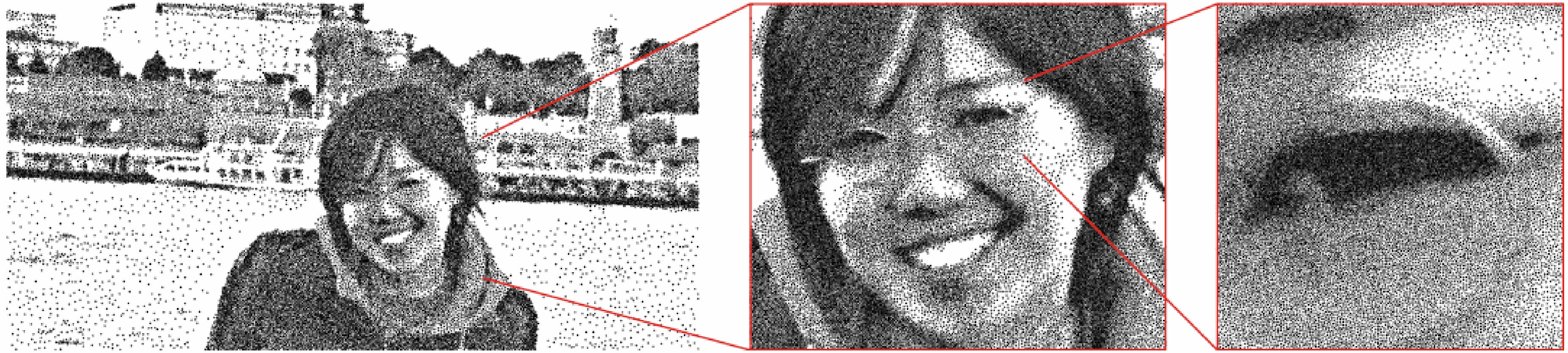
- Good topic for your final project

Fast Poisson-Disk Sampling

Fast Poisson-Disk Sampling



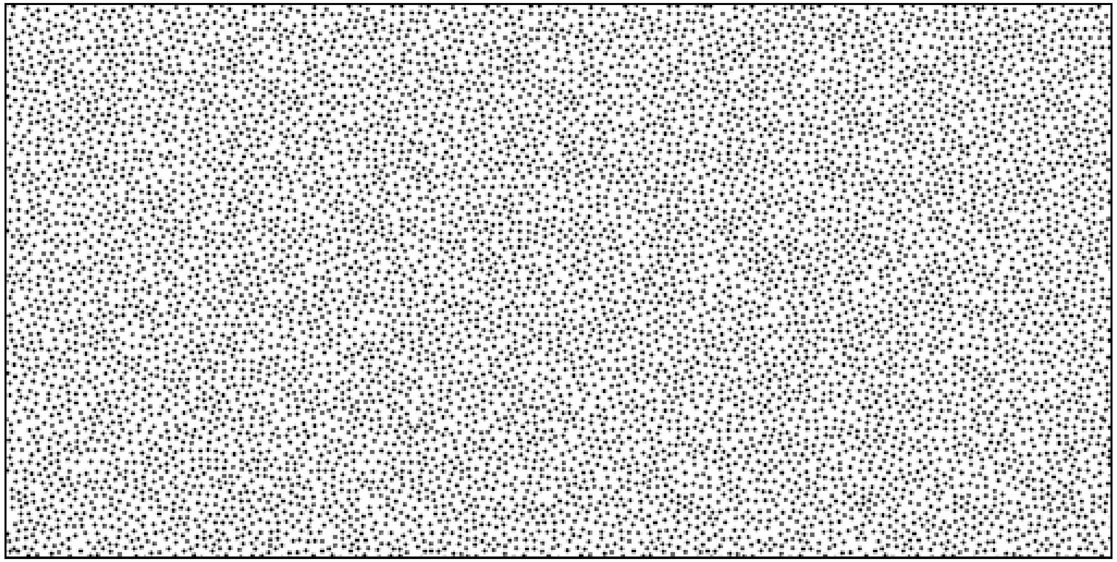
Recursive Wang Tiles for Blue Noise

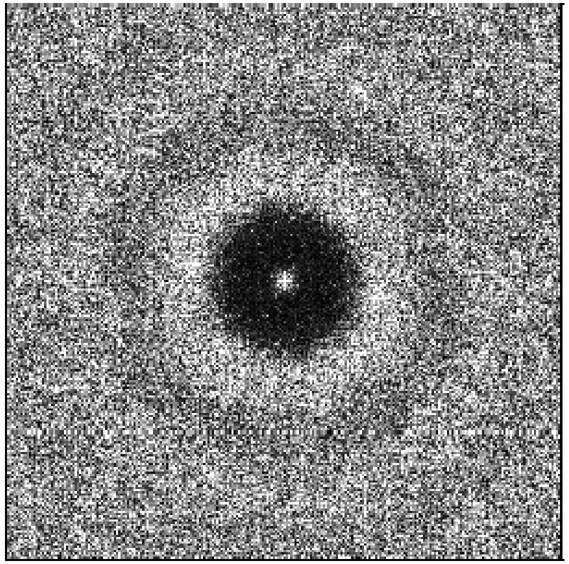


32,965 points, 22.75ms
1,449,011 points per second

34,897 points, 15.7ms
2,222,739 points per second

22,748 points, 11.67ms
1,949,272 points per second





Reconstruction filters

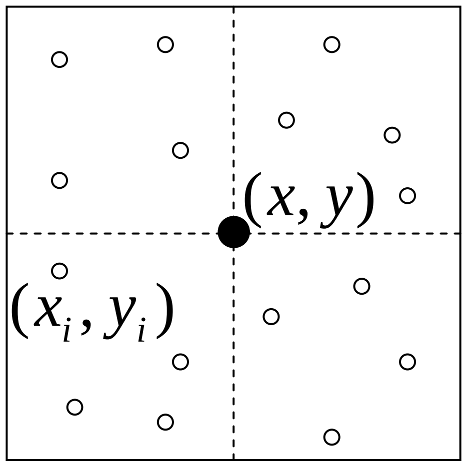
- Given image samples, we can do the following to compute pixel values.
  1. reconstruct a continuous function L’ from samples
  2. prefilter L’ to remove frequency higher than Nyquist limit
  3. sample L’ at pixel locations
- Because we will only sample L’ at pixel locations, we do not need to explicitly reconstruct L’s. Instead, we combine the first two steps.

Reconstruction filters

- Ideal reconstruction filters do not exist because of discontinuity in rendering. We choose nonuniform sampling, trading off noise for aliasing. There is no theory about ideal reconstruction for nonuniform sampling yet.
- Instead, we consider an interpolation problem

filter sampled radiance

$$\underbrace{I(x,y)}_{\text{final value}} = \frac{\sum_i \overbrace{f(x-x_i, y-y_i)}^{\text{filter}} \overbrace{L(x_i, y_i)}^{\text{sampled radiance}}}{\sum_i f(x-x_i, y-y_i)}$$



`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.

width, half of support

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

x, y is guaranteed to be within the range;
range checking is not necessary

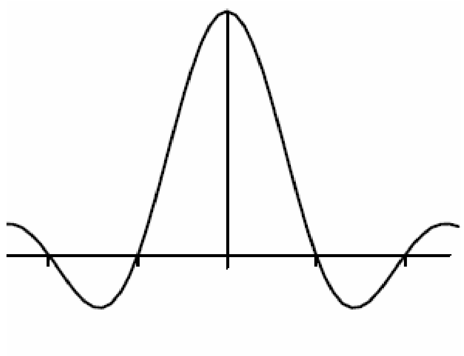
- **`filters/* (box, gaussian, mitchell, sinc, triangle)`**

Box filter

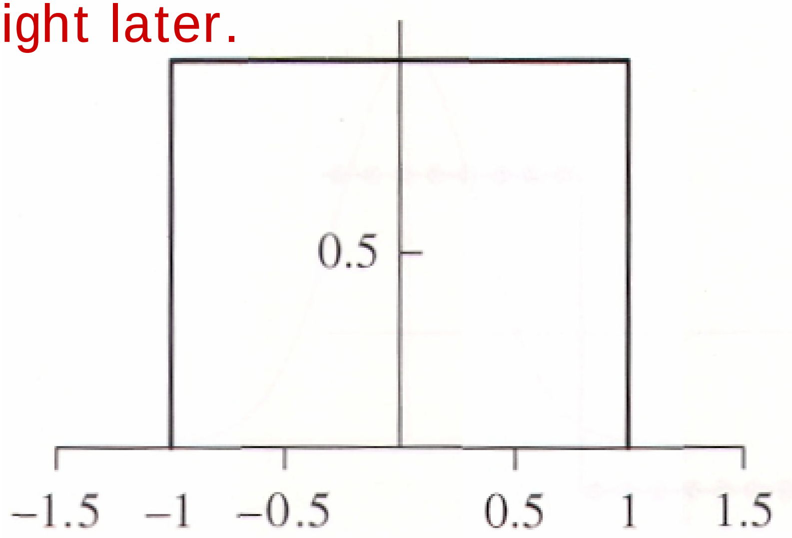
- Most commonly used in graphics. It's just about the worst filter possible, incurring postaliasing by high-frequency leakage.

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

no need to normalize since the weighted sum is divided by the total weight later.

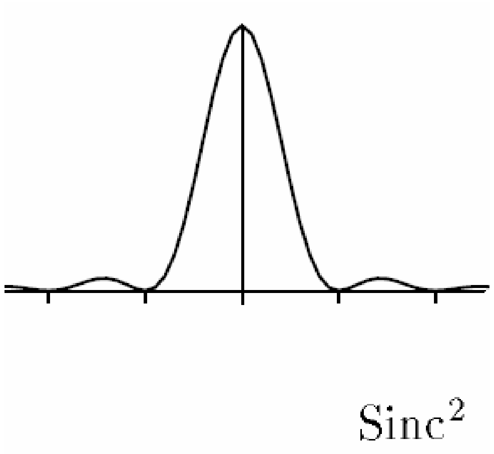


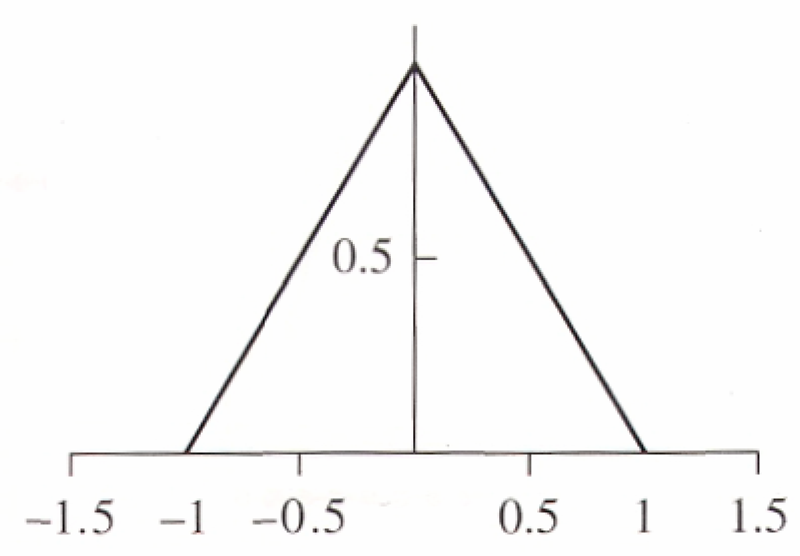
Sinc



Triangle filter

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



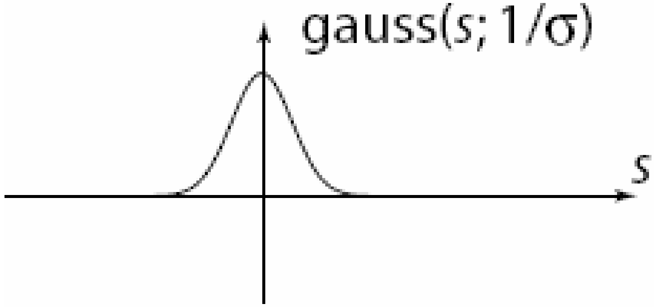


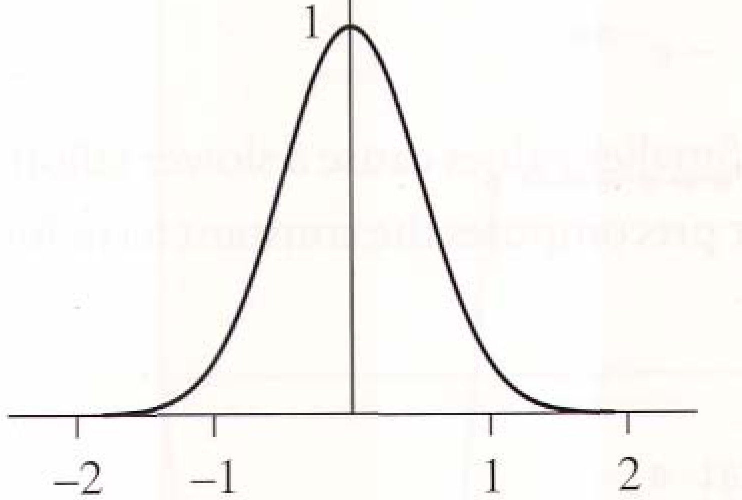
Gaussian filter

- Gives reasonably good results in practice

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

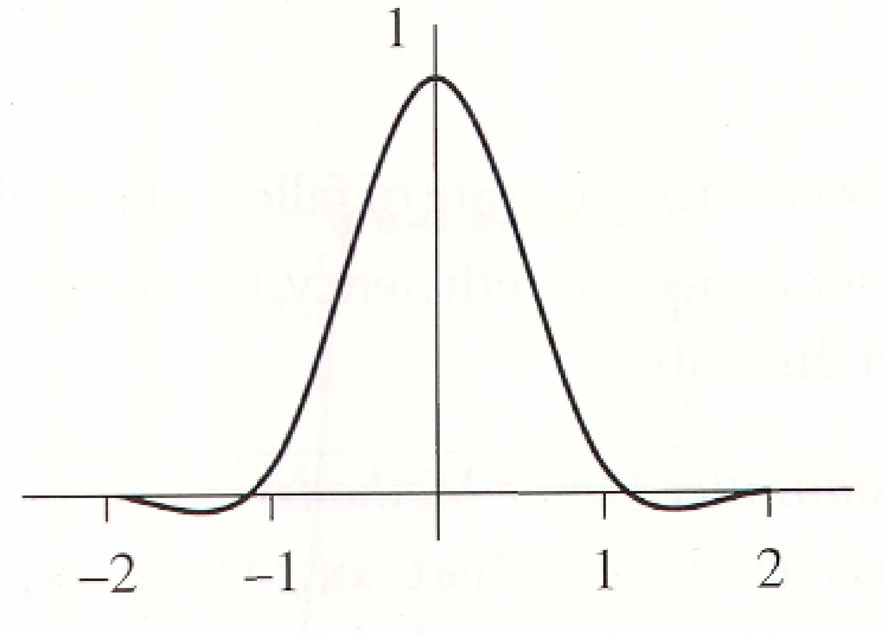
Gaussian essentially has a infinite support; to compensate this, the value at the end is calculated and subtracted.





Mitchell filter

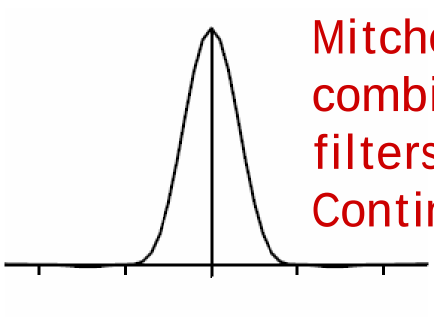
- parametric filters, tradeoff between ringing and blurring
- Negative lobes improve sharpness; ringing starts to enter the image if they become large.



Mitchell filter

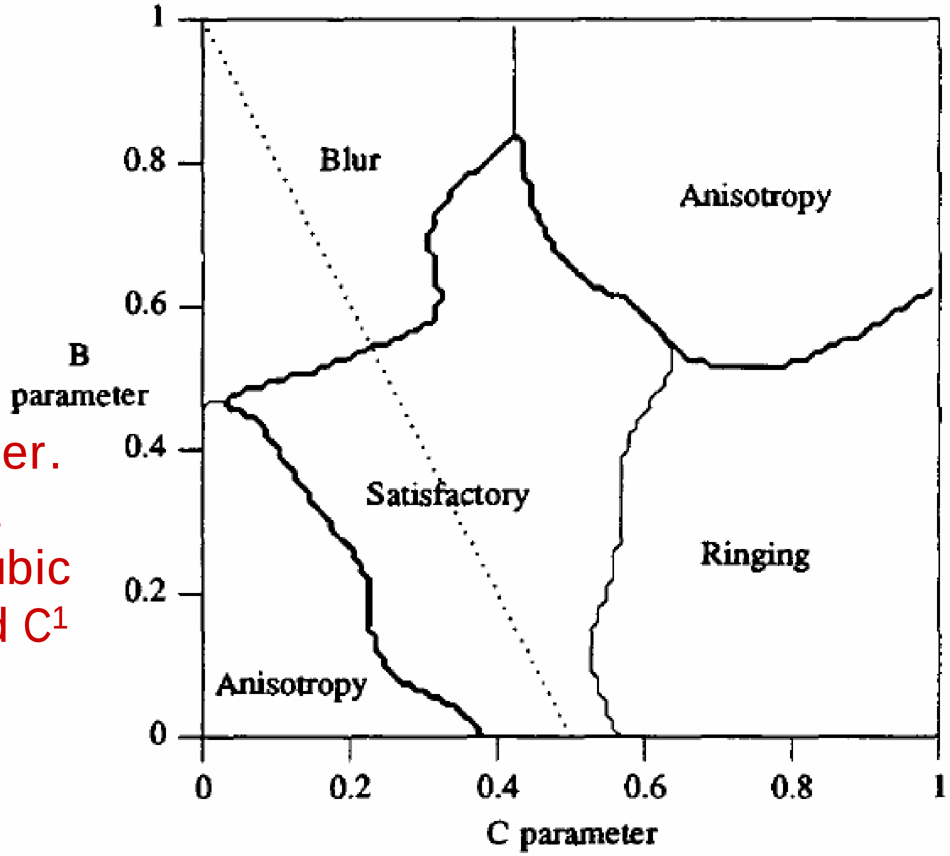
$$h(x)=\frac{1}{6}\begin{cases}(12-9B-6C)x^3+(-18+12B+6C)x^2+(6-2B) & |x|<1\\ (-B-6C)x^3+(6B+30C)x^2+(-12B-48C)x+(8B+24C) & 1<|x|<2\\ 0 & otherwise\end{cases}$$

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



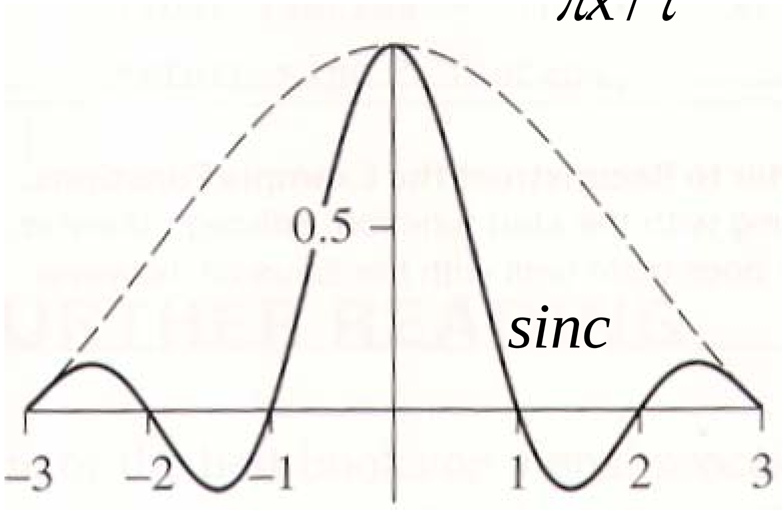
Sinc^4

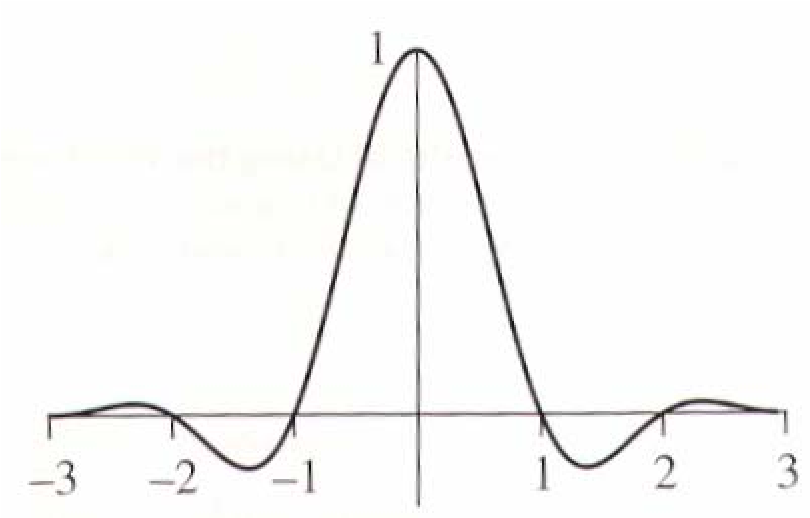
FFT of a cubic filter. Mitchell filter is a combination of cubic filters with C^0 and C^1 Continuity.



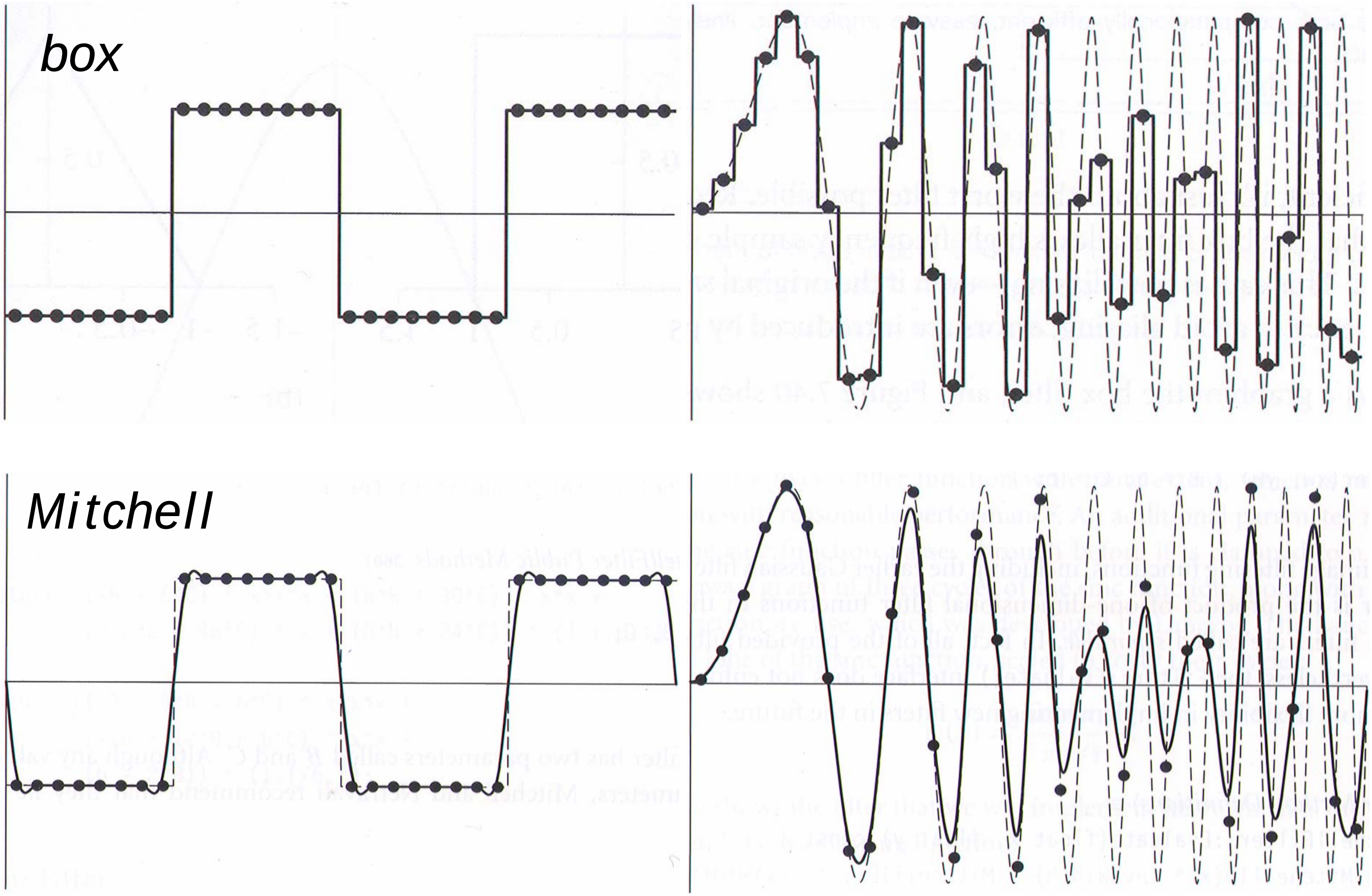
Windowed sinc filter

$$Lanczos \quad w(x) = \frac{\sin \pi x / \tau}{\pi x / \tau}$$

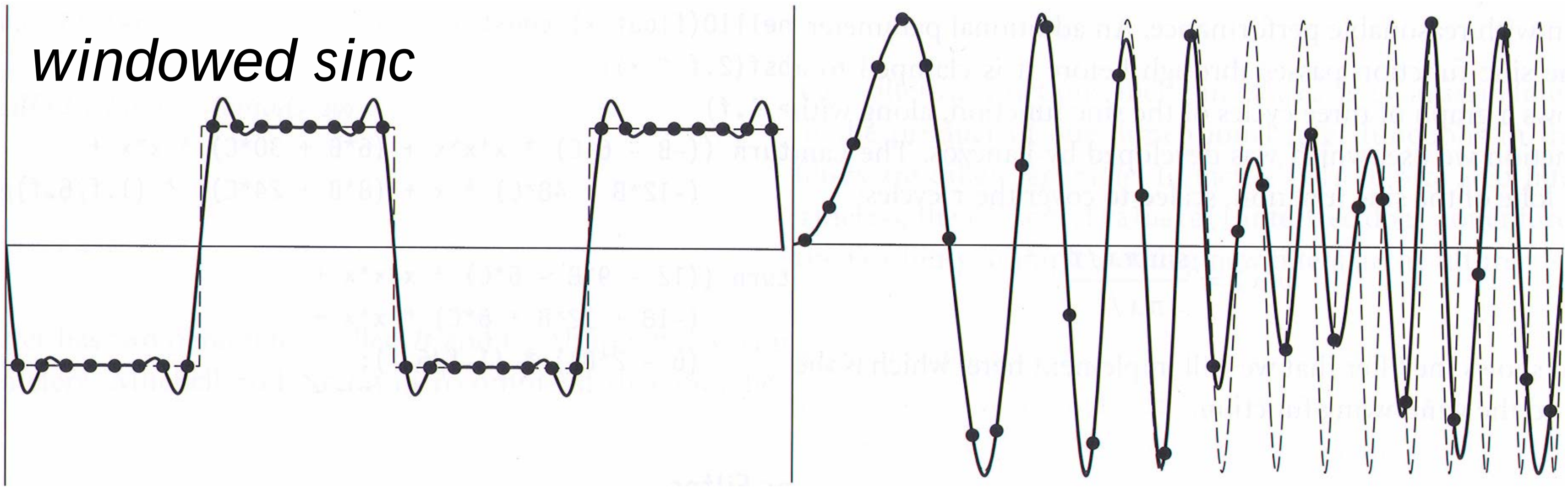


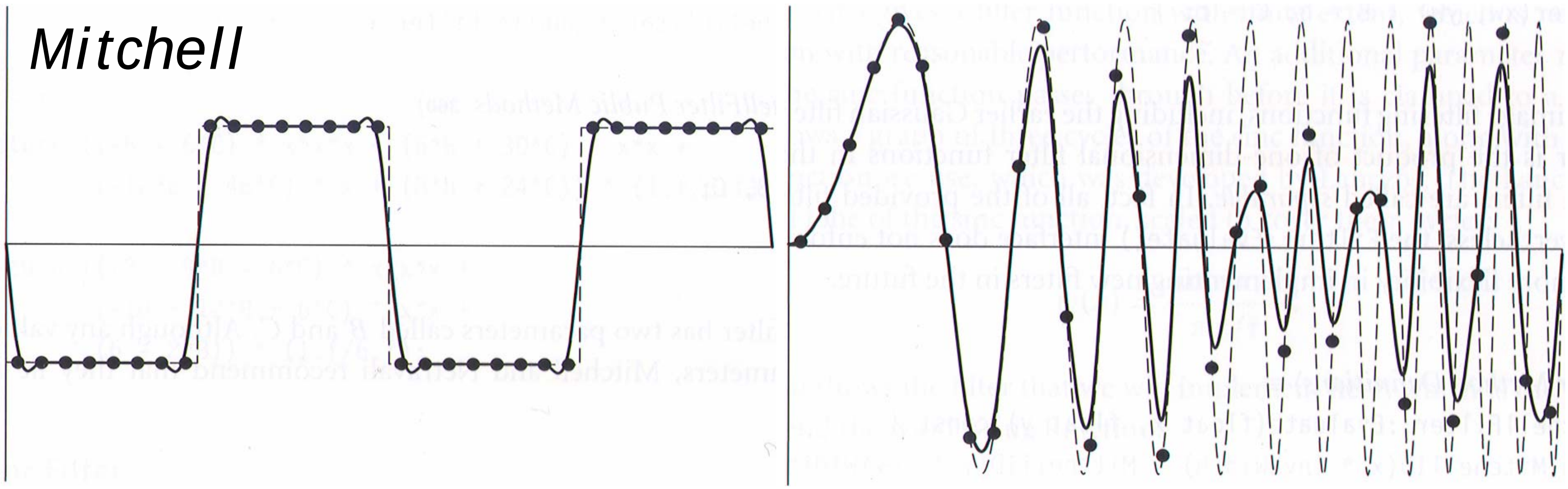


Comparisons

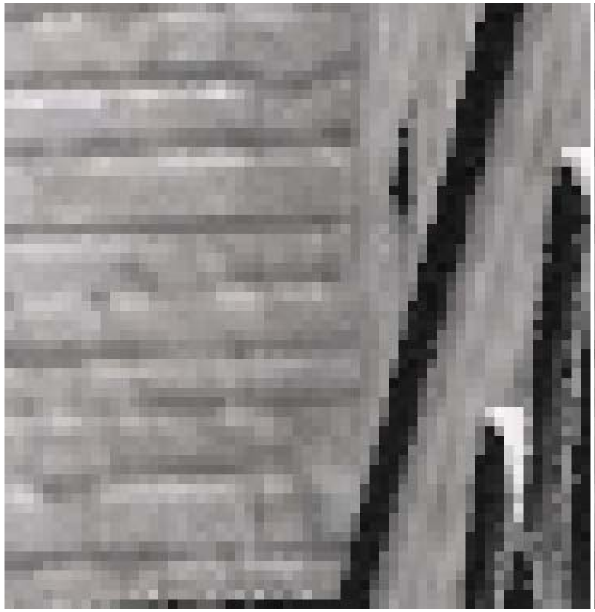


Comparisons

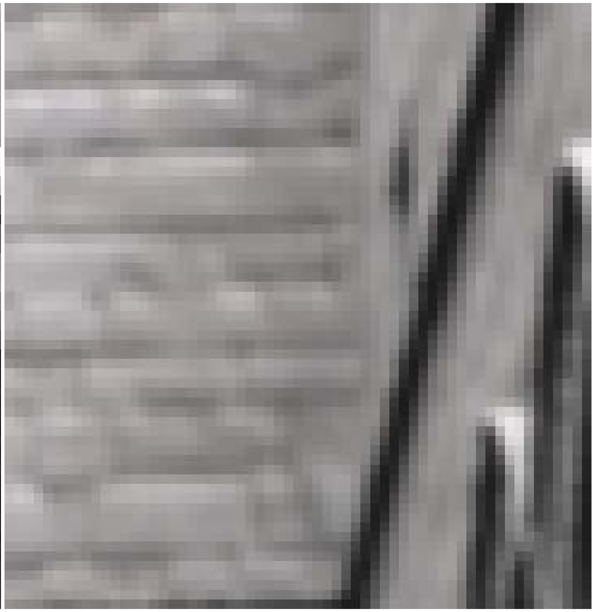




Comparisons



box



Gaussian

Mitchell