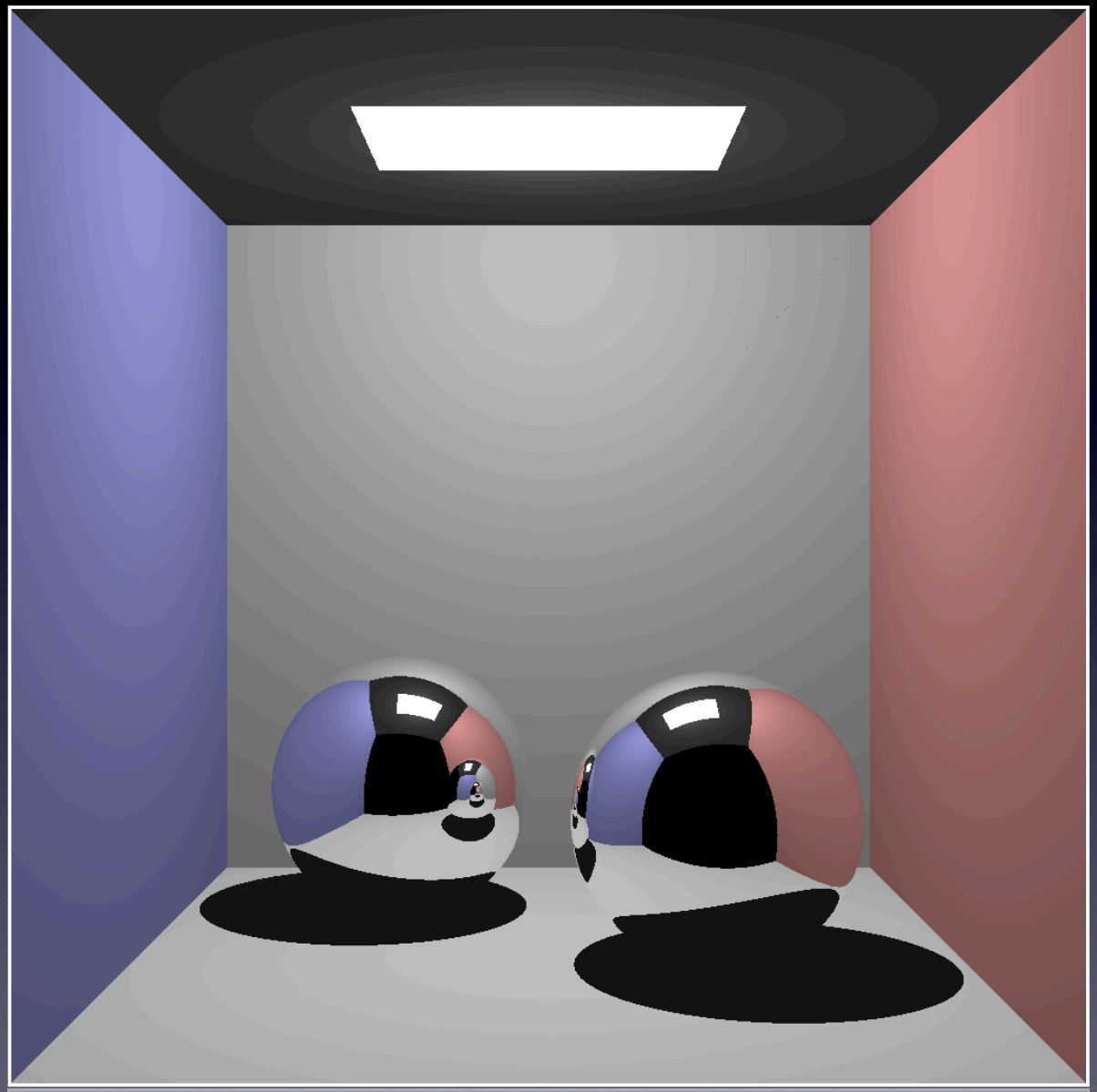
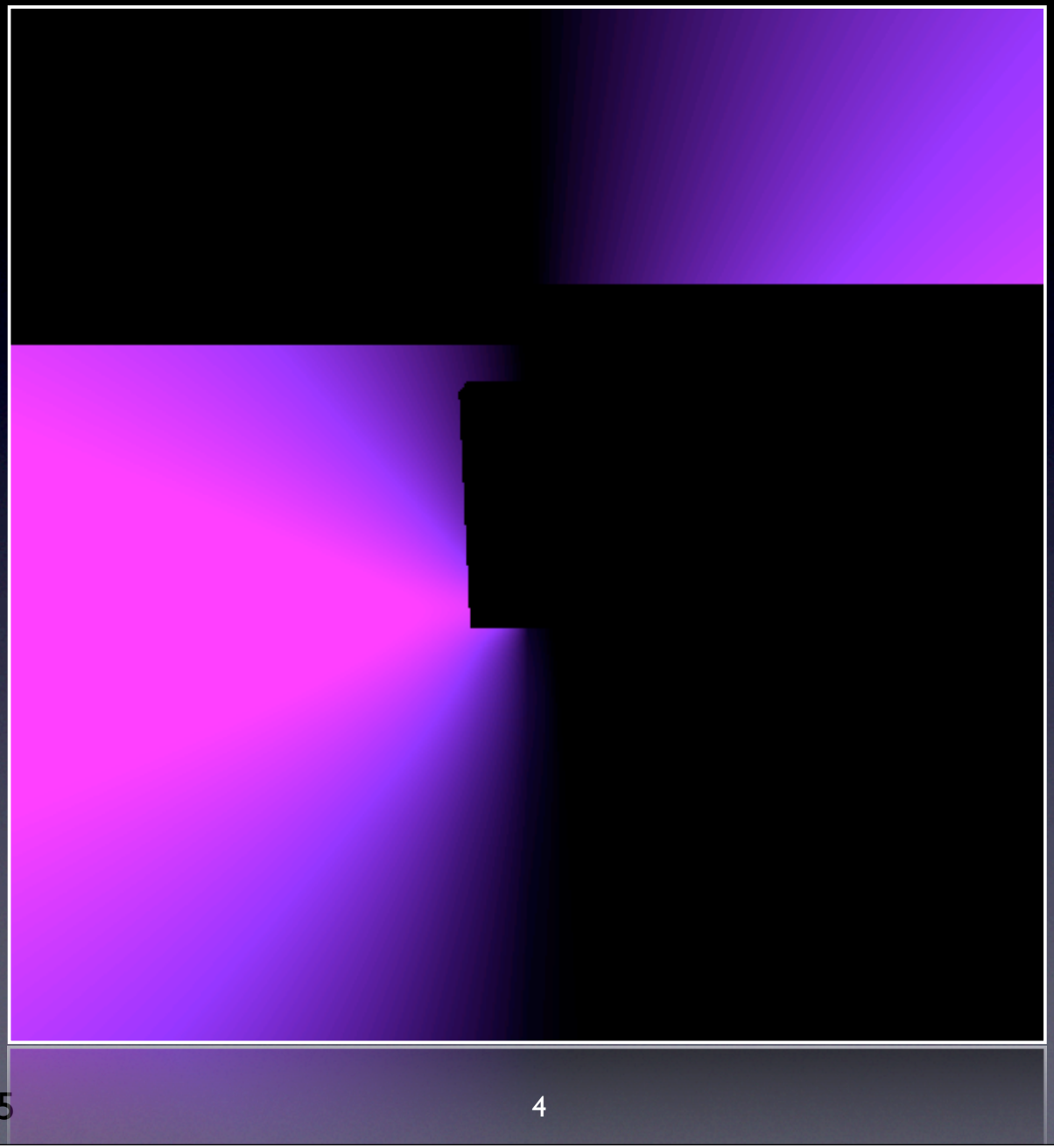
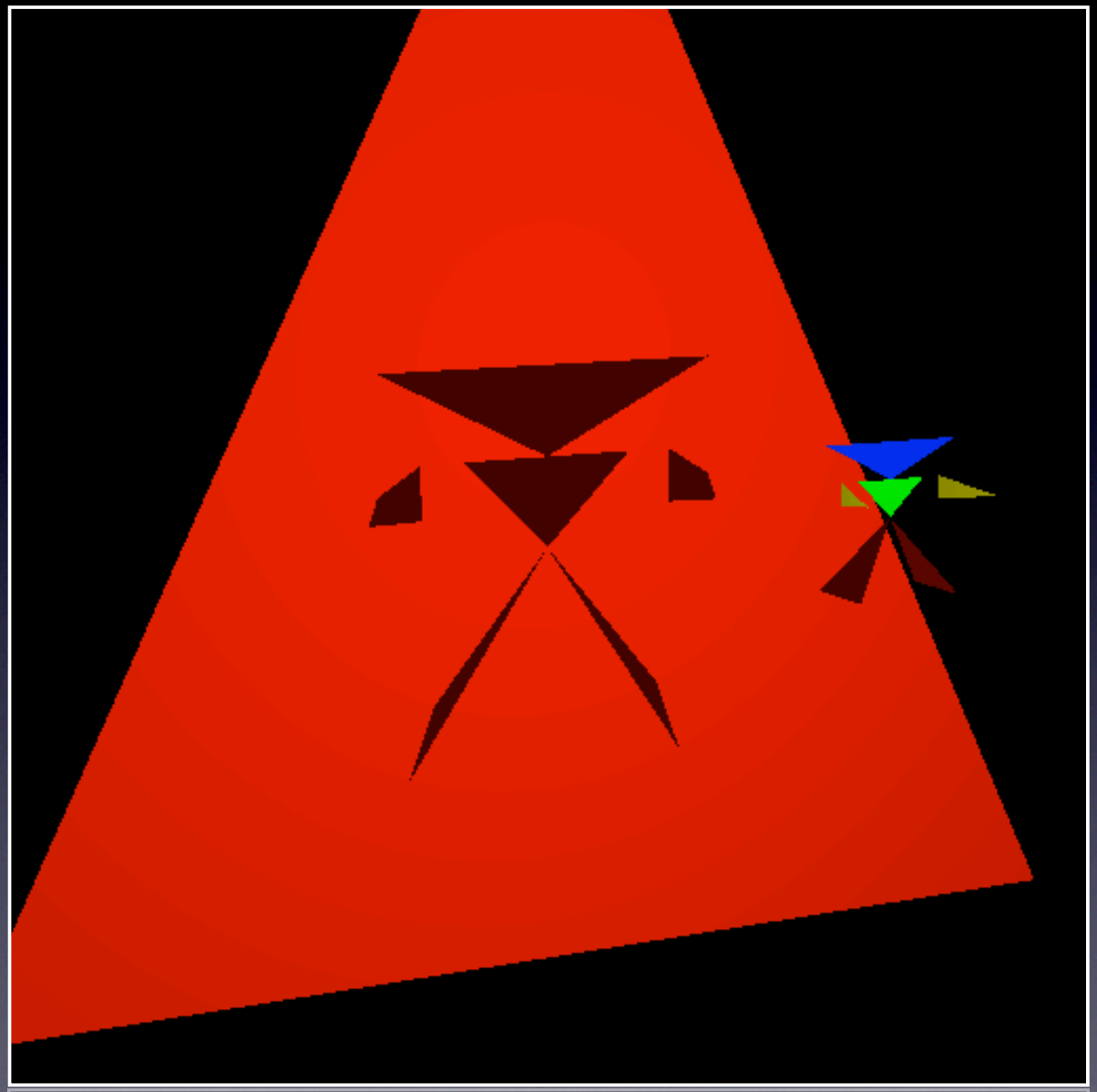
Sampling

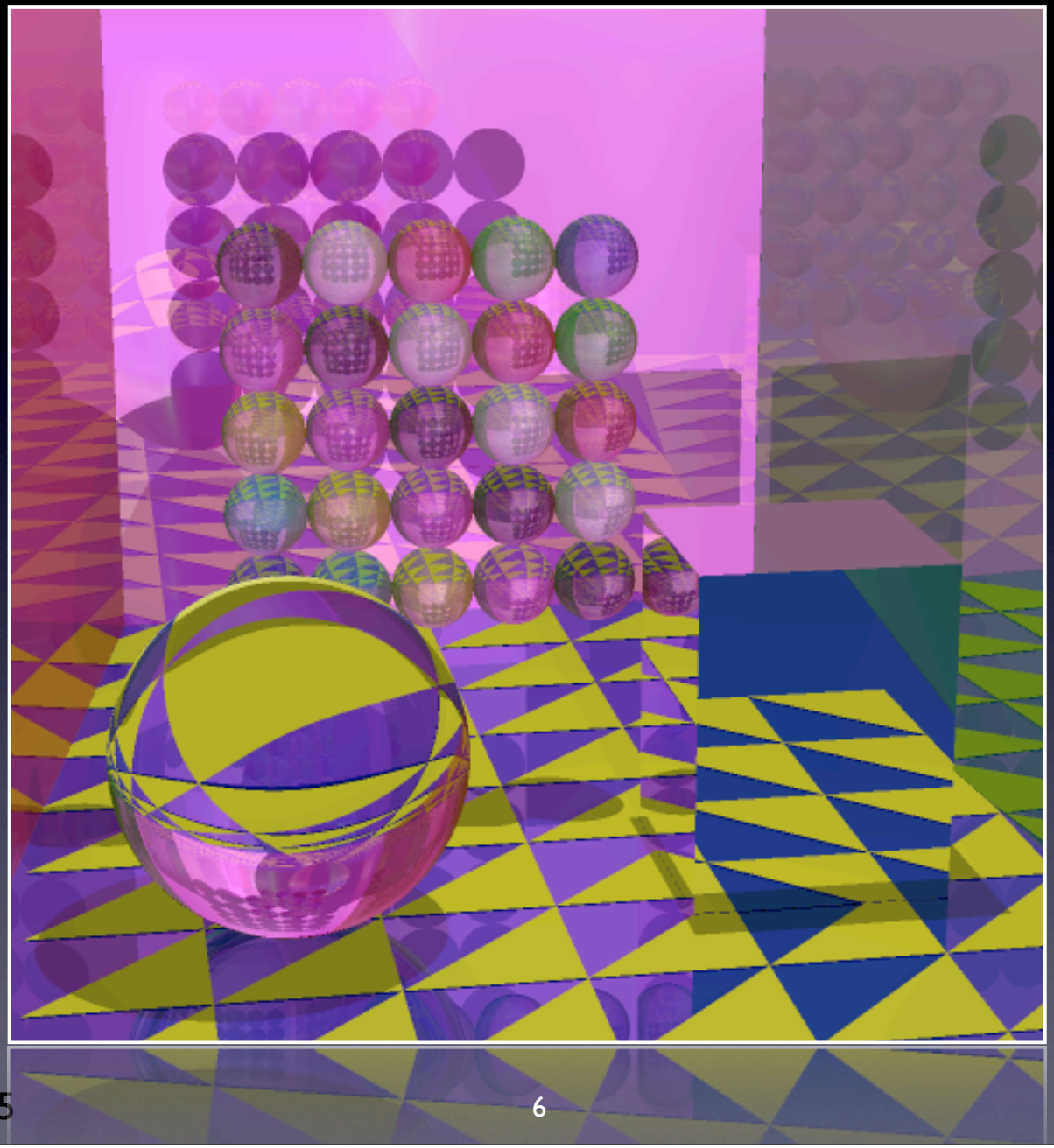
CS 6965 Fall 2011

Creative Program 3









Sampling Theory

- World: continuous
- Image: discrete samples
- Display: continuous function
- Ray tracing is a sampling proces
- Result: jaggies!

- Let's understand why

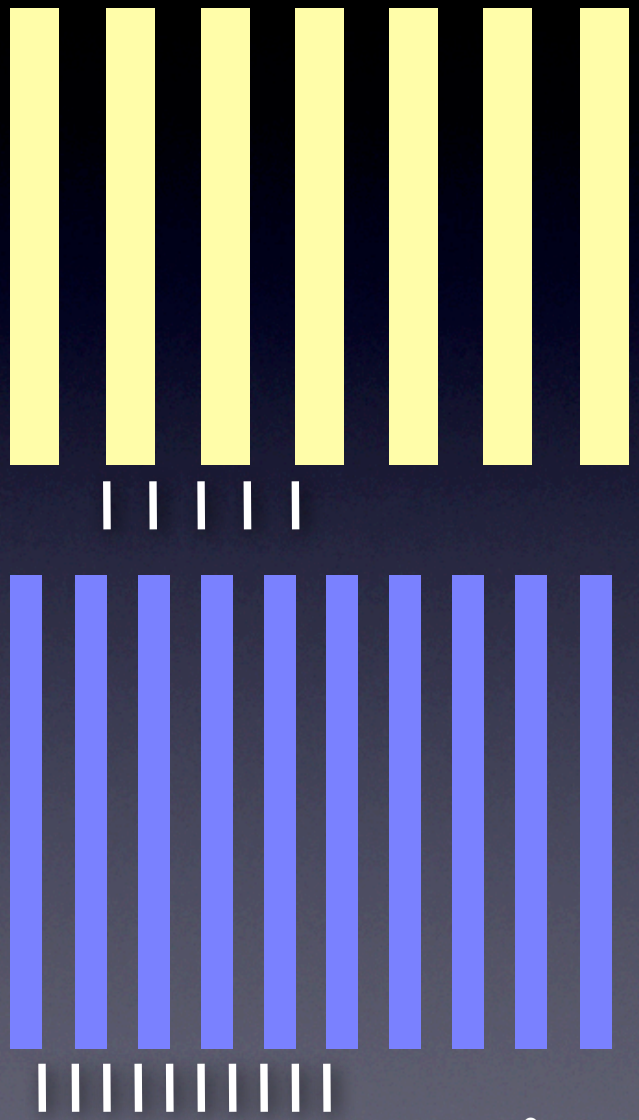
Aliasing

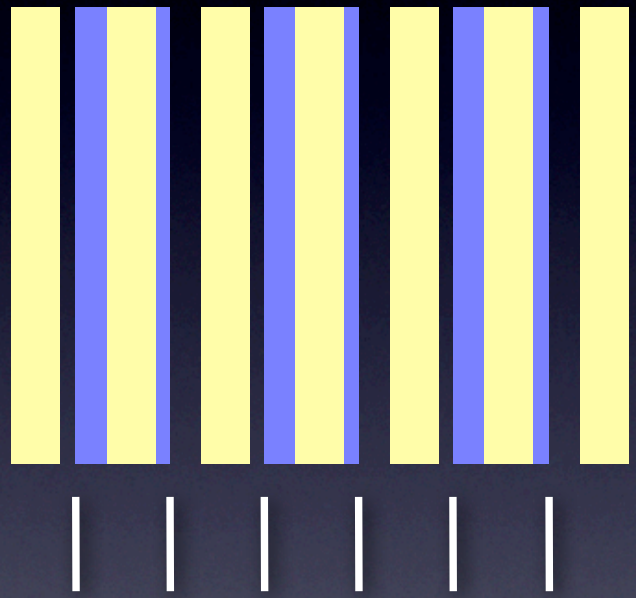
- Aliasing appears in several forms:
  - Jagged edges
  - Moire patterns
  - Strobe effect

Picket fences

Fence 1:
6 inch boards

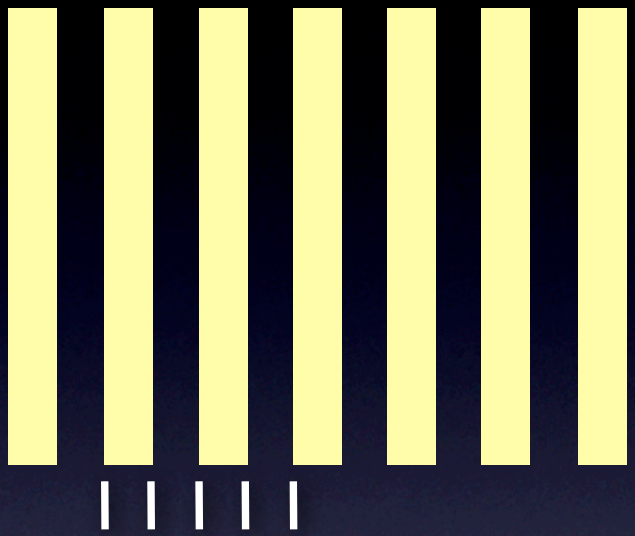
Fence 2:
4 inch boards

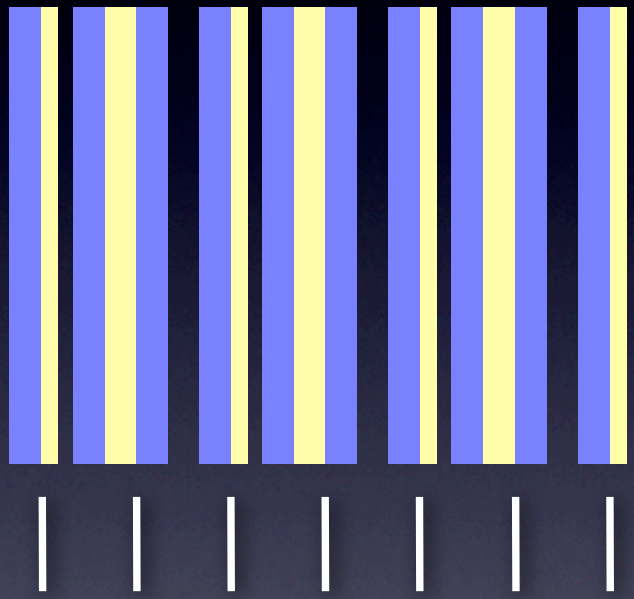




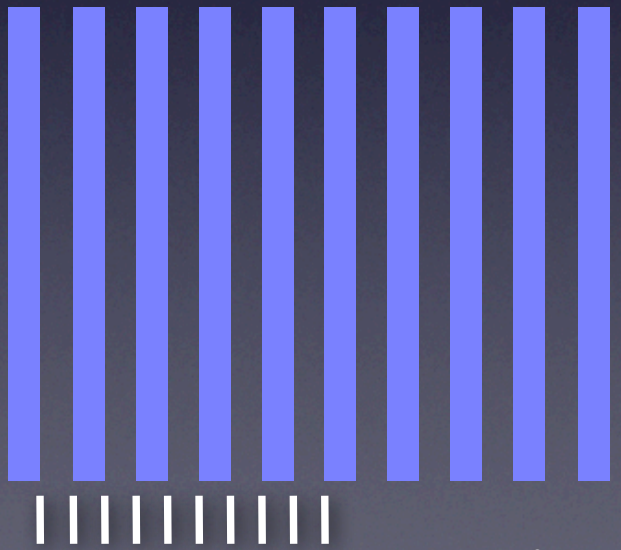
Picket fences

Fence 1:
6 inch boards





Fence 2:
4 inch boards



Aliasing

- Aliasing is just high frequencies disguised as low frequencies

High frequencies

- We need to understand where the high frequencies come from and what we can do about them

Low frequencies

Mathematical tools

- Fourier transform
- Sampling
- Shannon theorem
- Convolution
- Filtering

Fourier transform

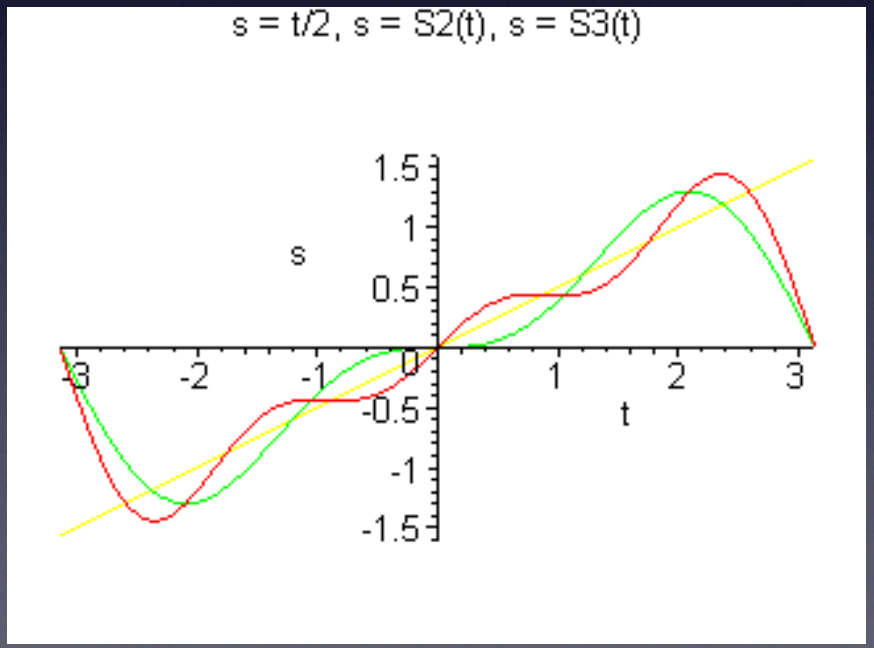
Given a function h(t) :

$$F\left[h(t)\right] = H(\omega) = \int_{-\infty}^{\infty} h(t) e^{-i\omega t} dt$$

$$F^{-1}\left[H(\omega)\right] = h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega) e^{i\omega t} dt$$

$$i = \sqrt{-1}$$

$$e^{ix} = \cos(x) + i\sin(x)$$



Discrete Fourier Transform

Continuous Fourier Transform(FT):

$$F\left[h(t)\right] = H(\omega) = \int_{-\infty}^{\infty} h(t) e^{-i\omega t} dt$$

$$F^{-1}\left[H(\omega)\right] = f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega) e^{i\omega t} dt$$

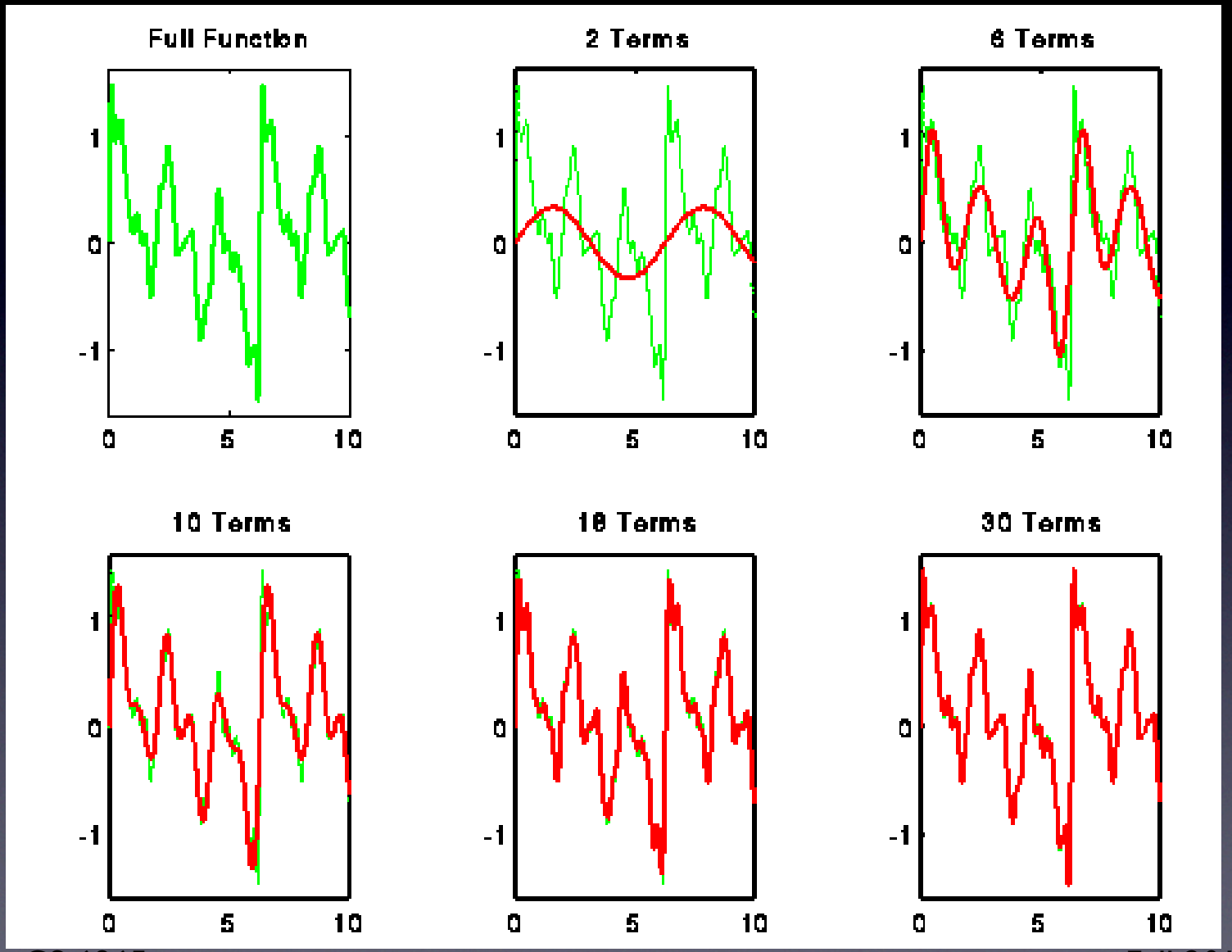
Discrete Fourier Transform (DFT):

$$F_n = \sum_{k=0}^{N-1} f_k e^{-2\pi i k/N}$$

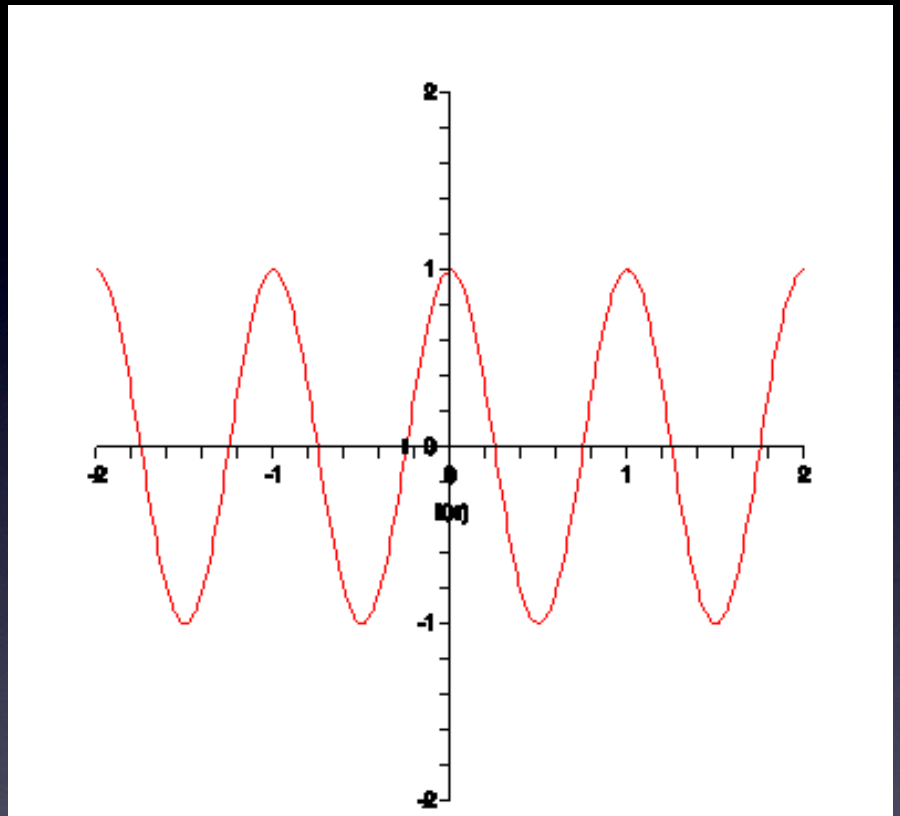
$$f_k = \frac{1}{N} \sum_{n=0}^{N-1} f_k e^{2\pi i n n/N}$$

Fast Fourier Transform (FFT) is just an optimized version of DFT (using half-angle identities)

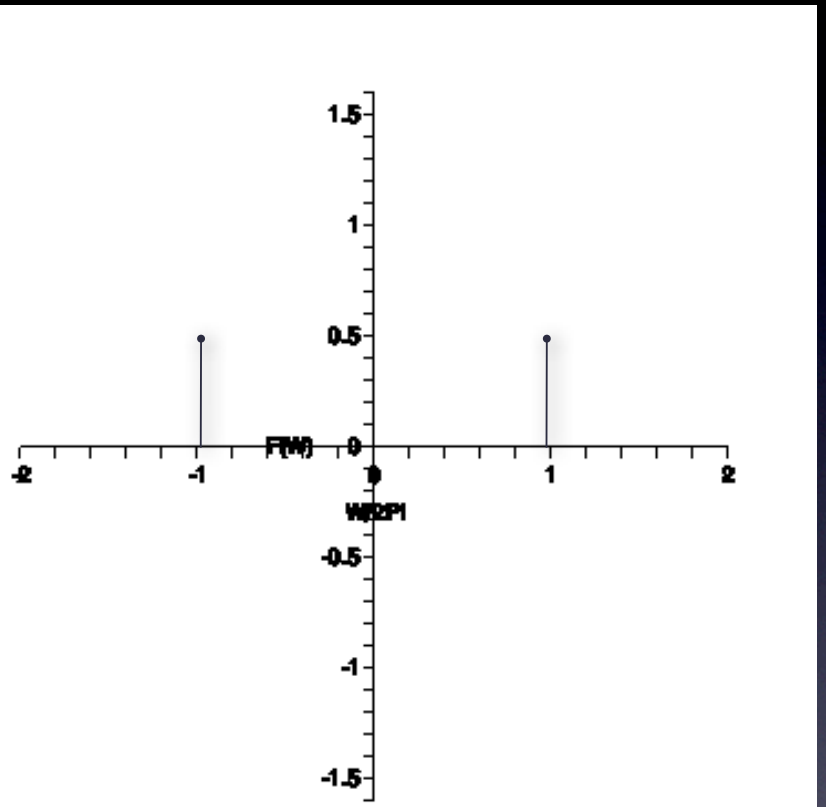
Frequencies



Frequency space

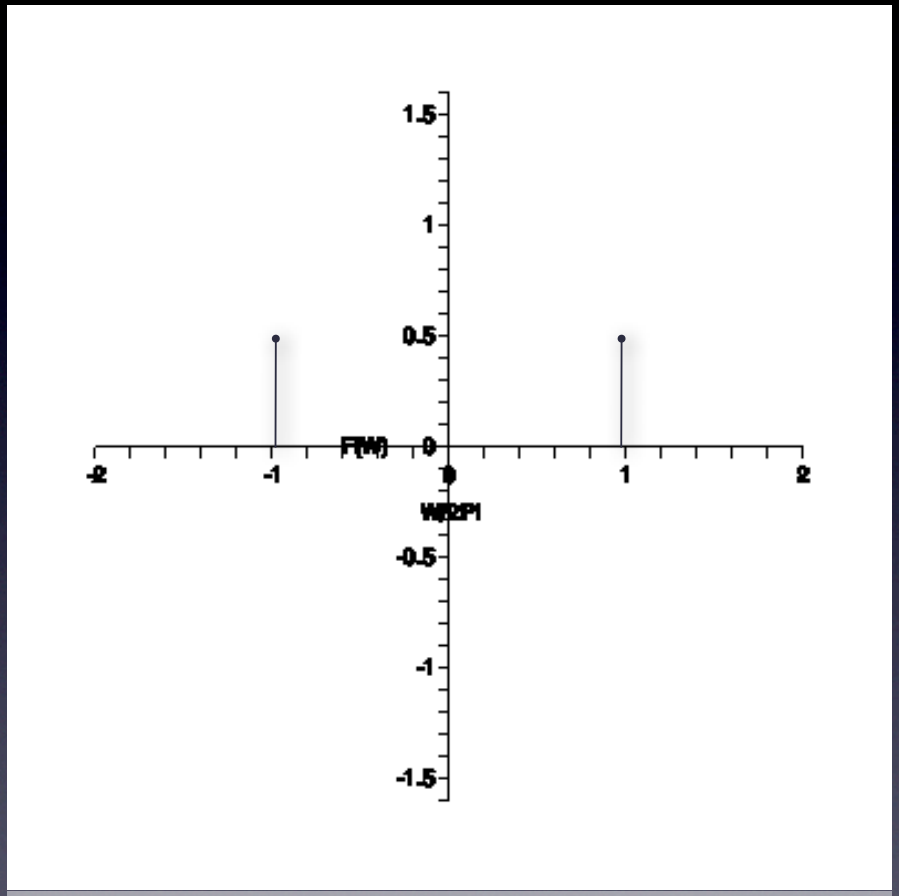


Time domain

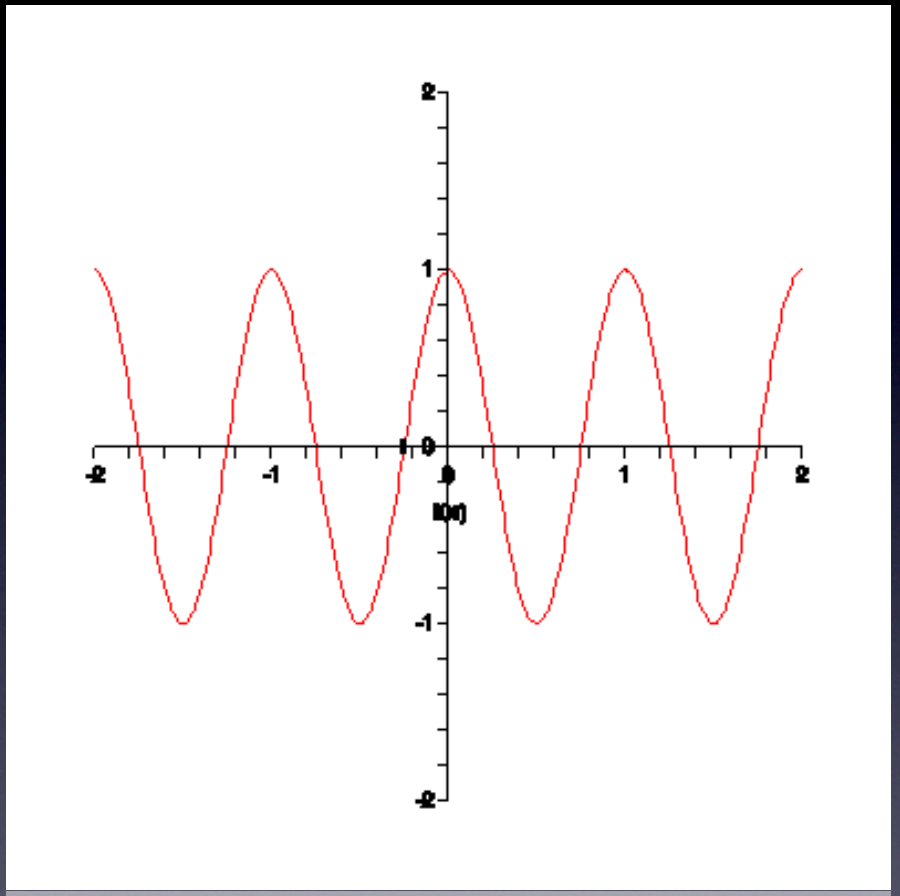


Frequency domain

Frequency space



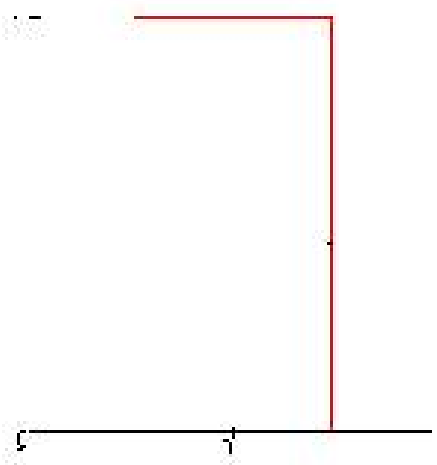
Time domain



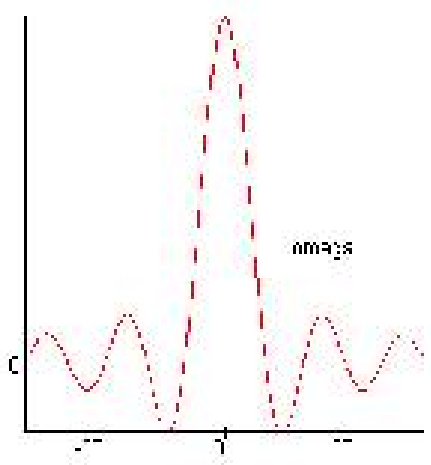
Frequency domain

Duality

Examples of FT pairs



box

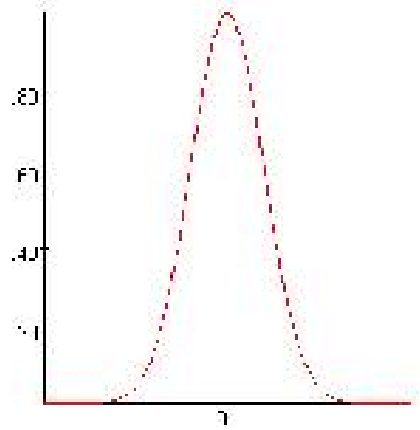


sinc

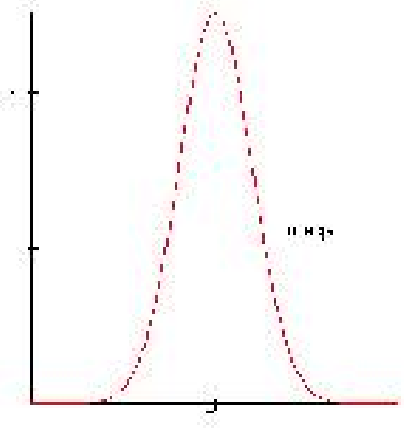
© 1999, Denis Zorin

source: http://mrl.nyu.edu/~dzorin/intro-graphics/lectures/lecture2/sld013.htm

Examples of FT pairs



Gaussian



Gaussian

© 1999, Denis Zorin

Dirac delta function

$$\delta(t) = \begin{cases} \infty & t = 0 \\ 0 & t \neq 0 \end{cases}$$

$$\delta(t - t_0) = \begin{cases} \infty & t = t_0 \\ 0 & t \neq t_0 \end{cases}$$

$$\int_{-\infty}^{\infty} \delta(t)\,dt = 1$$

Sampling

$$\int_{-\infty}^{\infty} \delta(t) f(t) dt = f(0)$$

$$\int_{-\infty}^{\infty} \delta(t) f(t - t_0) dt = f(t_0)$$

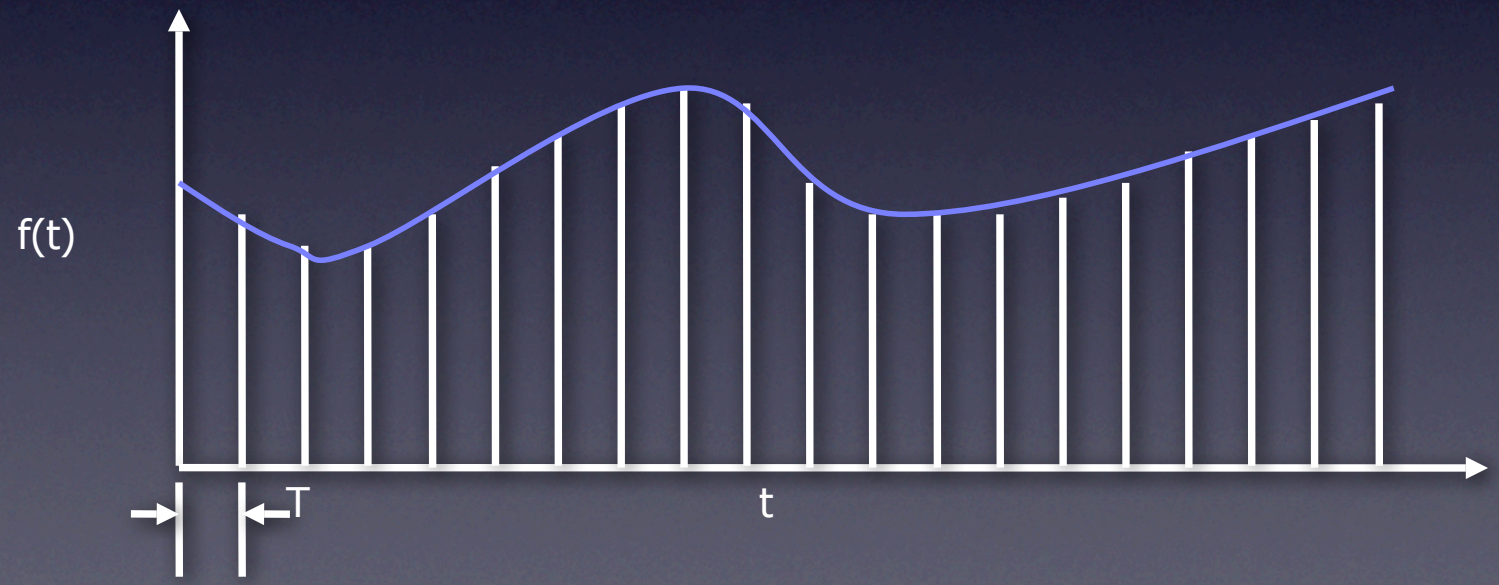
The delta function allows us to sample the function f

Sampling

More samples: more delta functions

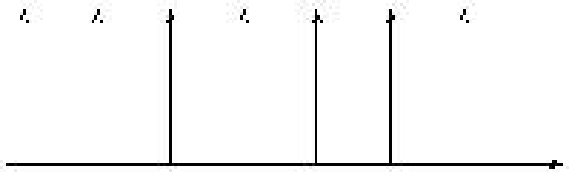
$$comb(t) = \sum_{-\infty}^{\infty} \delta(t - nT)$$

$$\int_{-\infty}^{\infty} comb(t) f(t) dt = ..., f(-2T), f(-T), f(0), f(T), f(2T), ...$$



Comb function

$$comb = \sum_{n=-\infty}^{\infty} \delta(t - nT)$$



FT of a comb is also a comb

© 1999, Denis Zorin

Convolution

g, h : *functions*

$$g * h = c(t) = \int_{-\infty}^{\infty} g(t)h(t-\tau)d\tau$$

$$F(g(t)) * F(h(t)) = F(g(t)h(t))$$

$$F(g(t)h(t)) = F(g(t) * h(t))$$

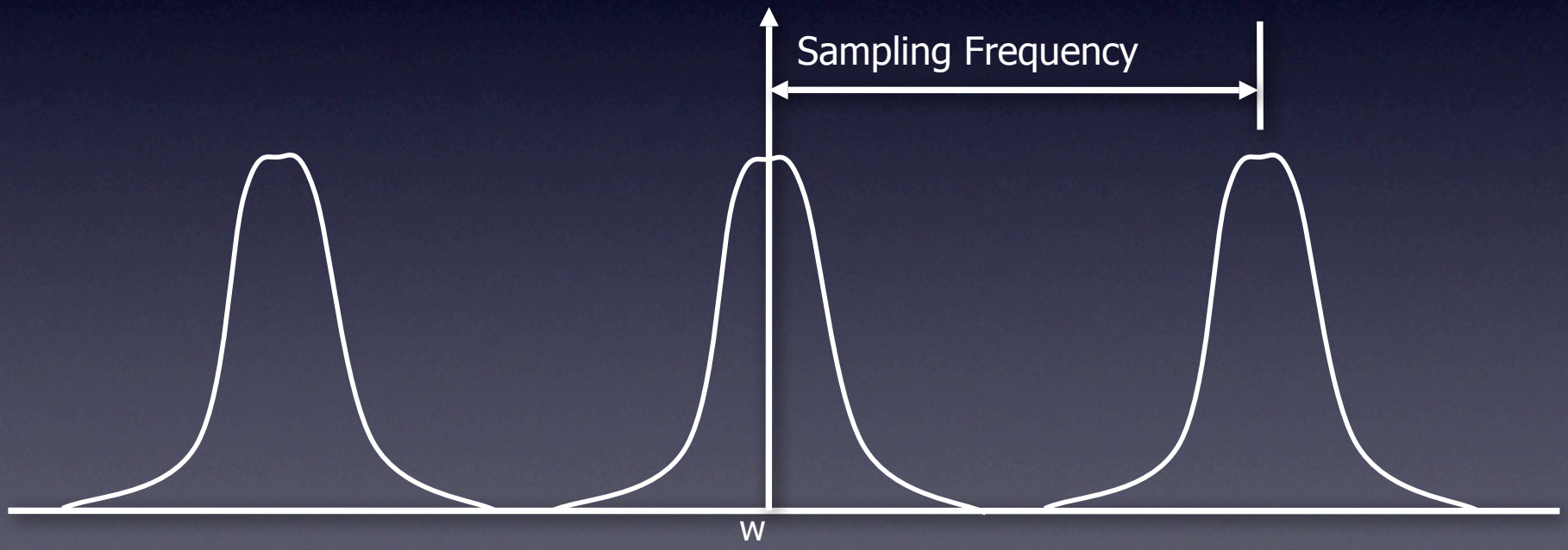
Multiplication in frequency space is convolution in time and vice-versa

Sampling

given g(t):

$$F\big(g(t)comb(t)\big) = F\big(g(t)\big) * F\big(comb(t)\big)$$

$$= F\big(g(t)\big) * comb(t)$$

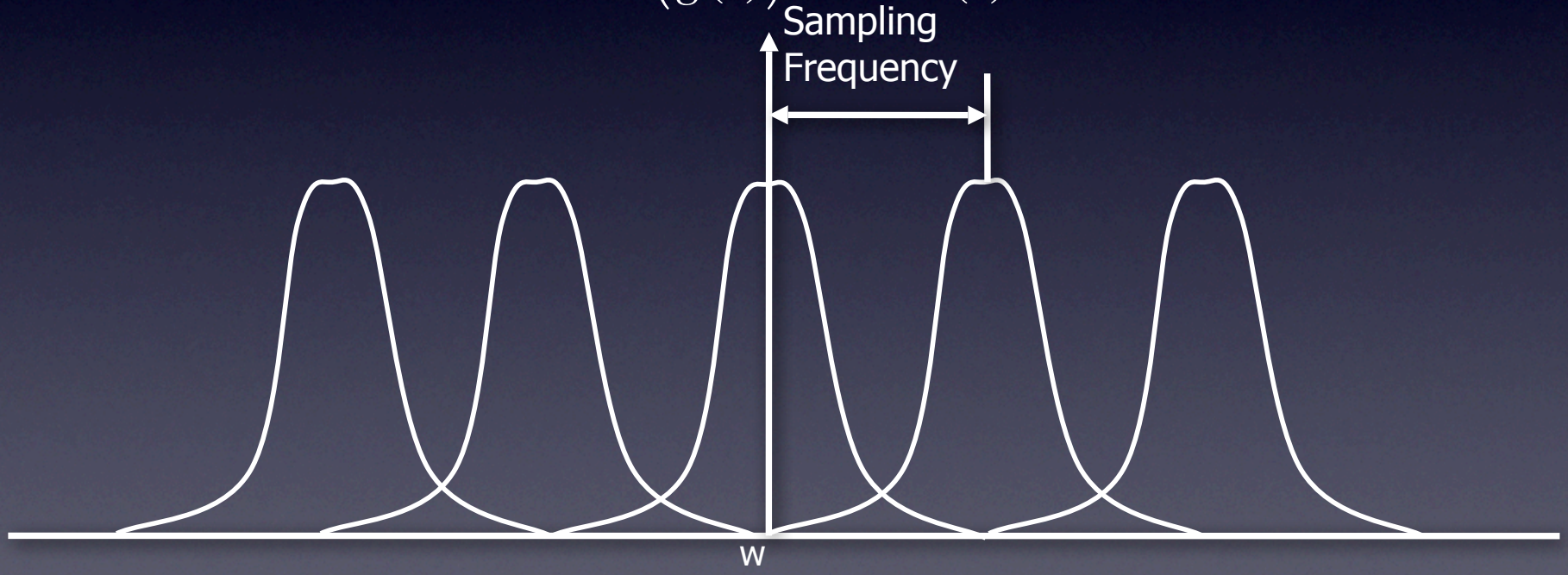


Sampling

given g(t):

$$F\big(g(t)comb(t)\big) = F\big(g(t)\big) * F\big(comb(t)\big)$$

$$= F\big(g(t)\big) * comb(t)$$

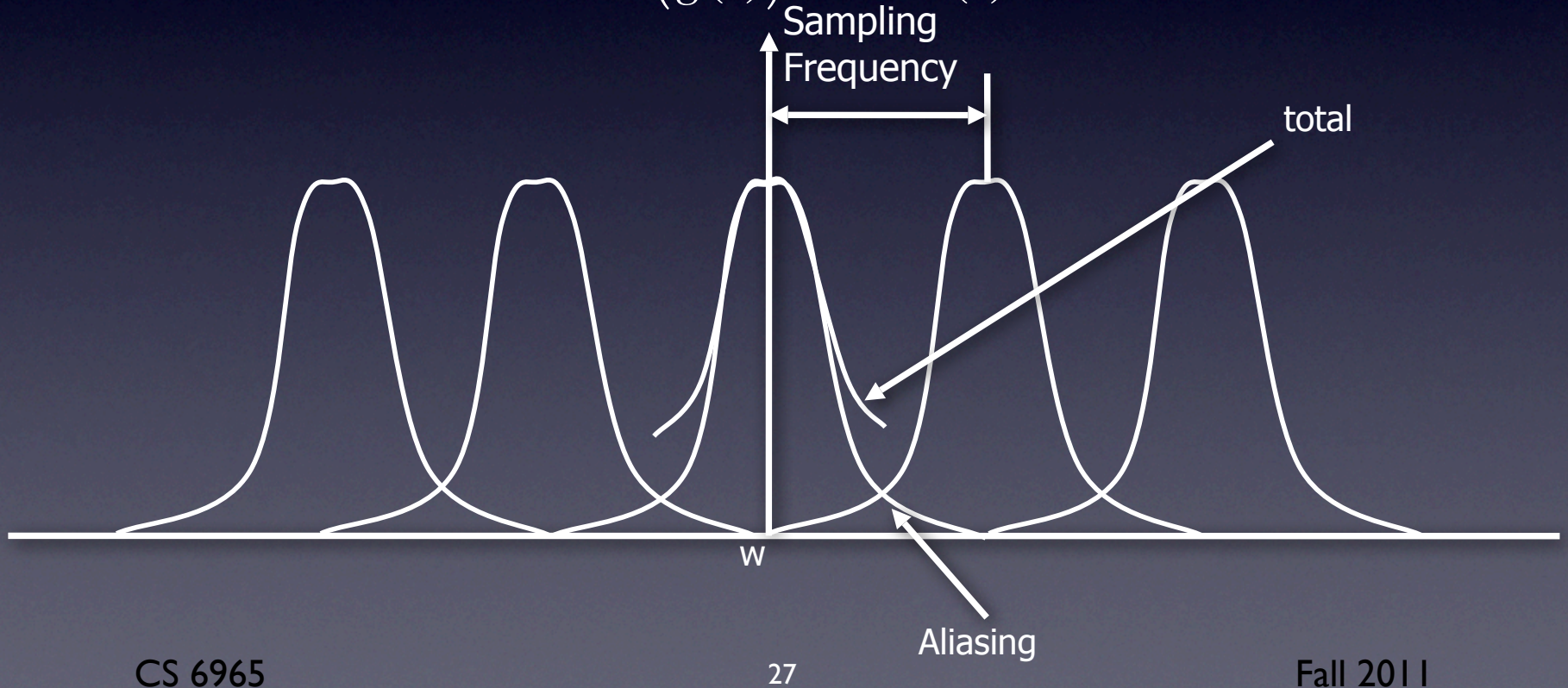


Sampling

given g(t):

$$F\big(g(t)comb(t)\big) = F\big(g(t)\big) * F\big(comb(t)\big)$$

$$= F\big(g(t)\big) * comb(t)$$

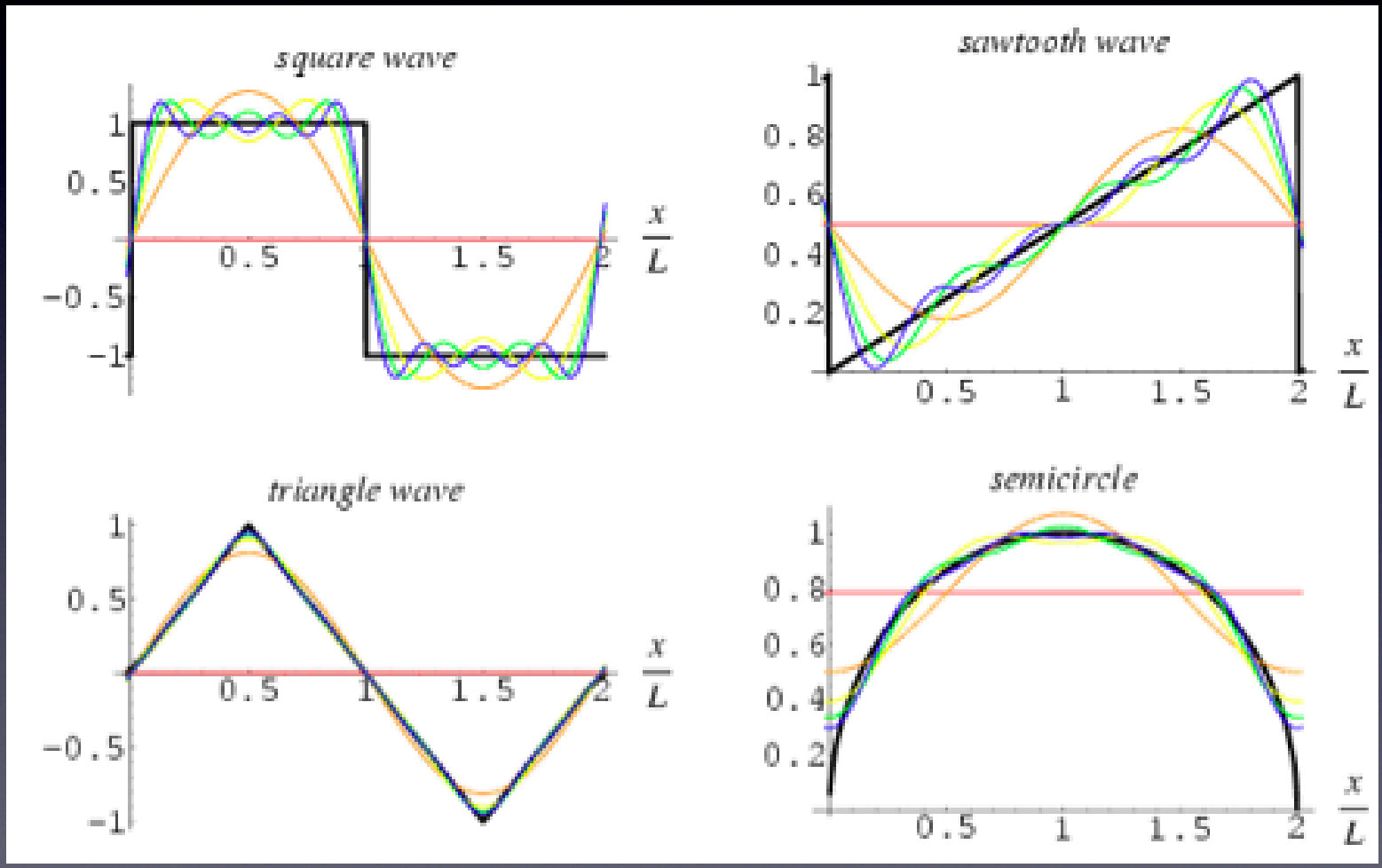


Nyquist theorem (Shannon theorem, sampling theorem)

- Any function g can be represented by samples of frequency F iff the function g is band-limited to a frequency of F/2

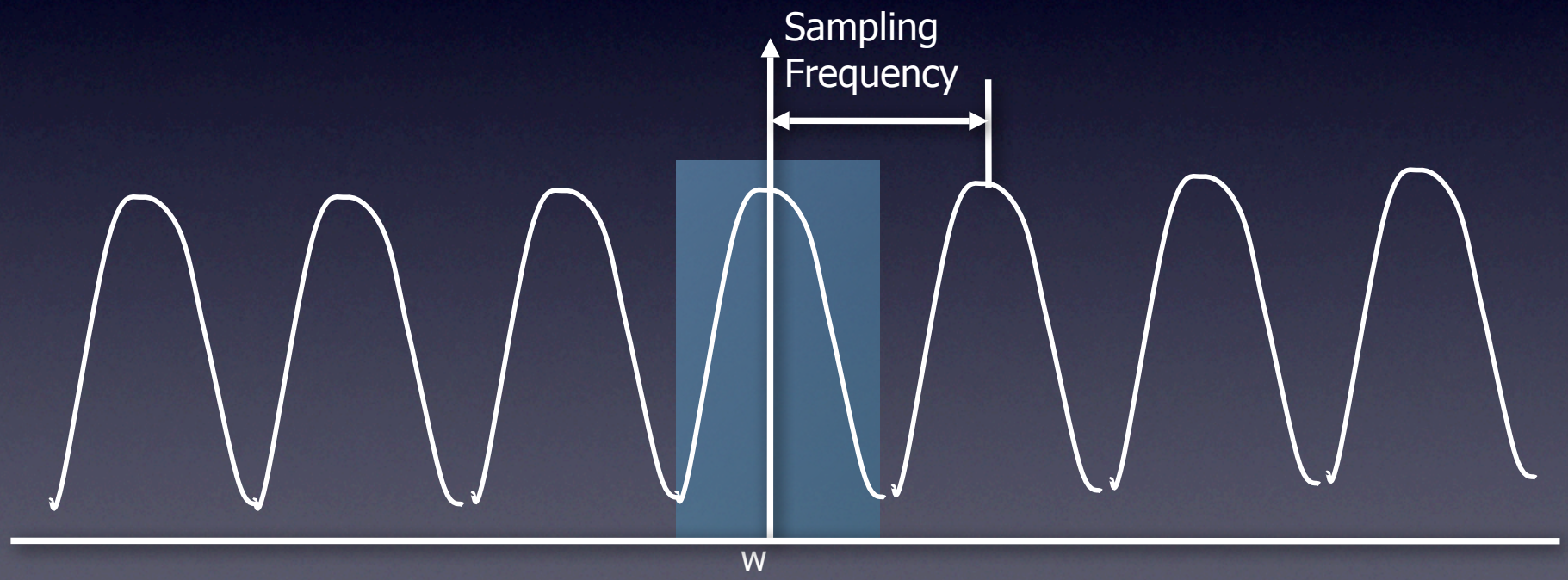
Fourier series

$$f(t) = a_0 + \sum_{n=1}^{\infty} \left(a_0 \cos \pi n + b_0 \sin \pi n \right)$$



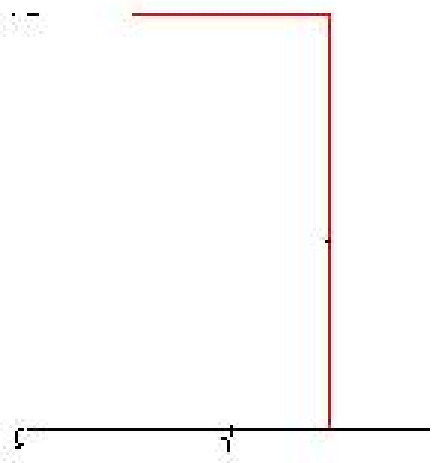
Band-limiting

- How do we prevent aliasing?
- Band-limited functions

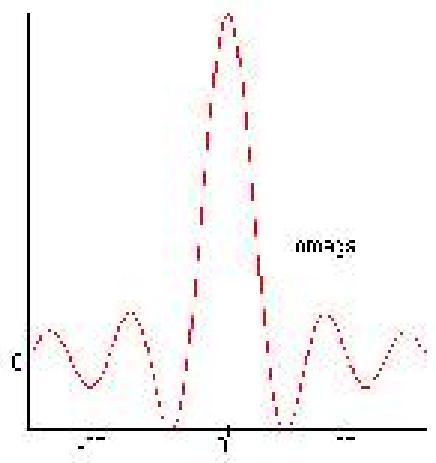


Fix 1: Band-limiting

Examples of FT pairs



box



sinc

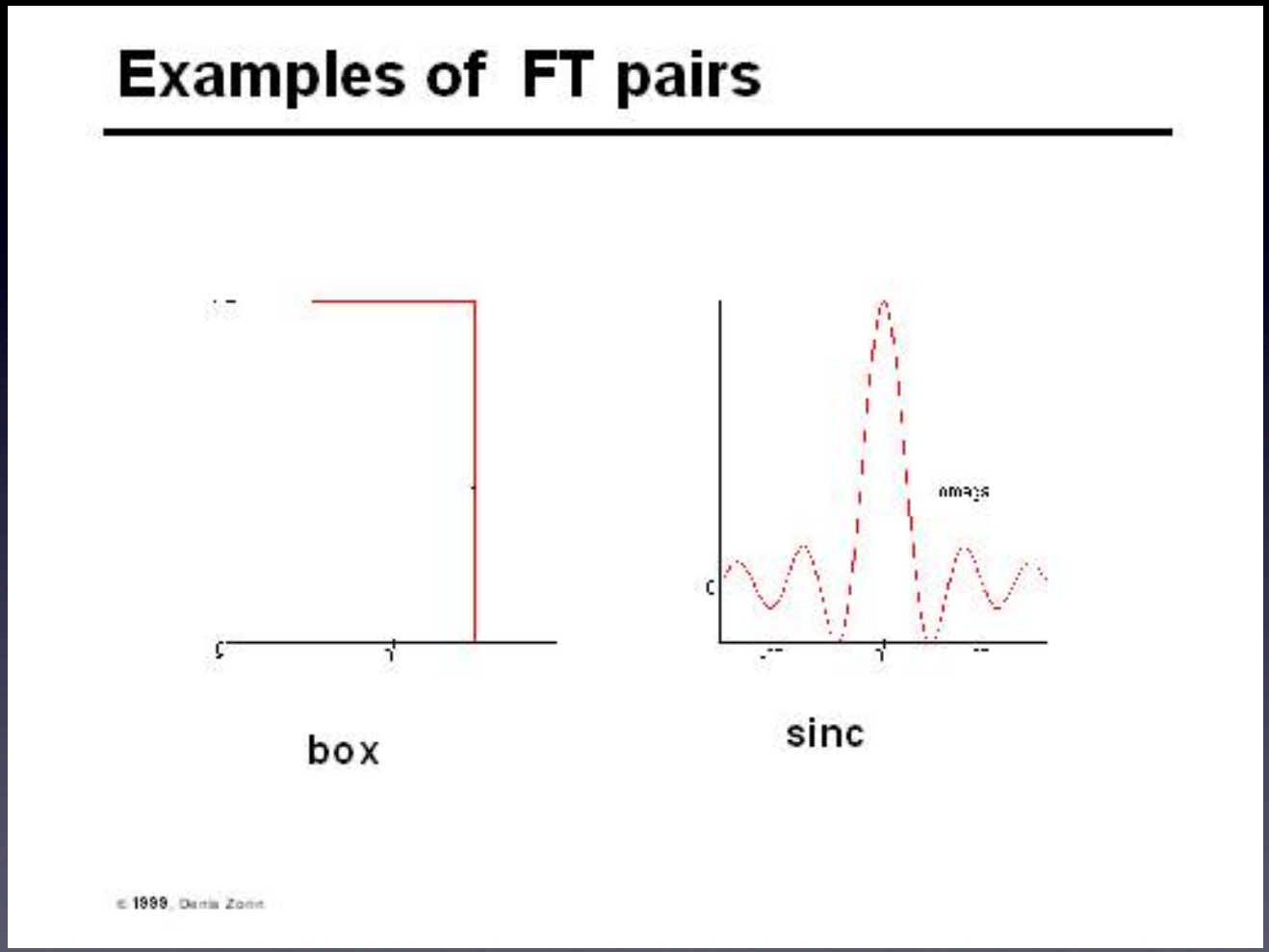
© 1998, Denis Zorin

source: http://mrl.nyu.edu/~dzorin/intro-graphics/lectures/lecture2/sld013.htm

Band limiting in graphics

- No silver bullet:
  - Lines, polygon edges: analytical formulations (doesn't handle lines crossing)
  - Textures: mip-maps, other filters
  - Objects: level of detail
  - Fonts?

Fix 2: Increase sampling frequency



Fix 3: Non-delta function samples

- Beam tracing
- Cone tracing
- Not always possible

Fix 4: Non-uniform samples

- Idea 1: Use a few samples to estimate the frequencies to decide the sampling rate
  - May miss some high frequencies
  - Works pretty well for edges
- Idea 2: Use random instead of uniform samples
  - F(white noise)=white noise
  - Can potentially recover all frequencies
  - May not always recover some frequencies
  - Lost frequencies "masked" by visual system

Reconstruction

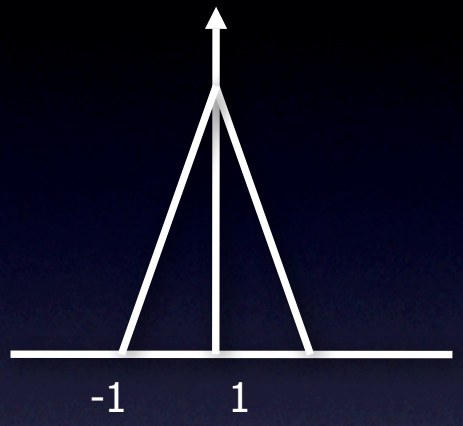
- Nyquist/Shannon: Any function g can be represented by samples of frequency F iff the function g is band-limited to a frequency of F/2
- How?

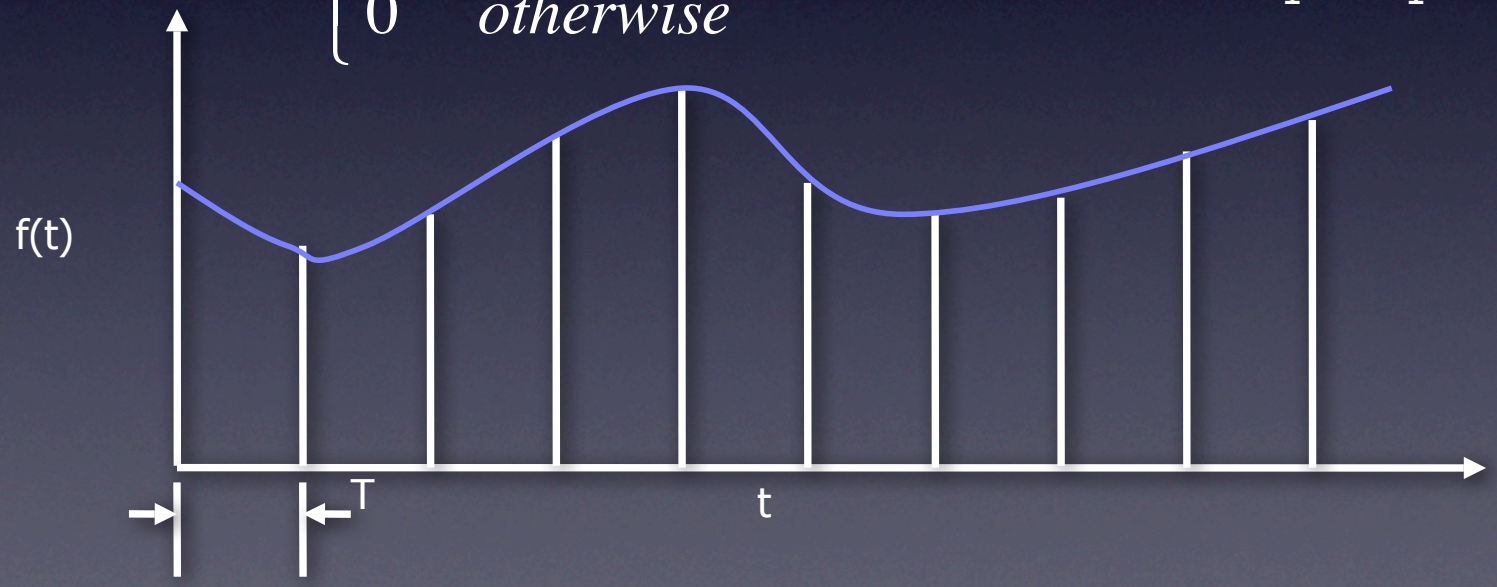
Reconstruction

Connect the dots?

$$f(t)=\sum_{s=1}^{n} a_s T(t)$$

$$T(t) = \begin{cases} -t & -1 \le t < 0 \\ t & 0 \le t \le 1 \\ 0 & otherwise \end{cases}$$

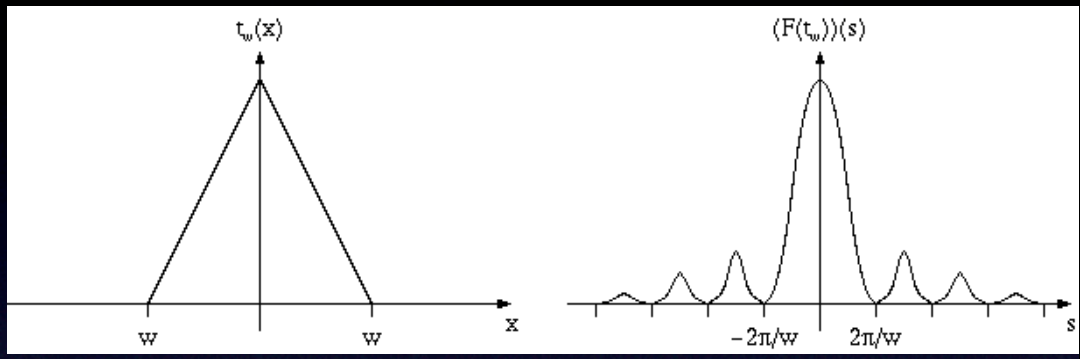


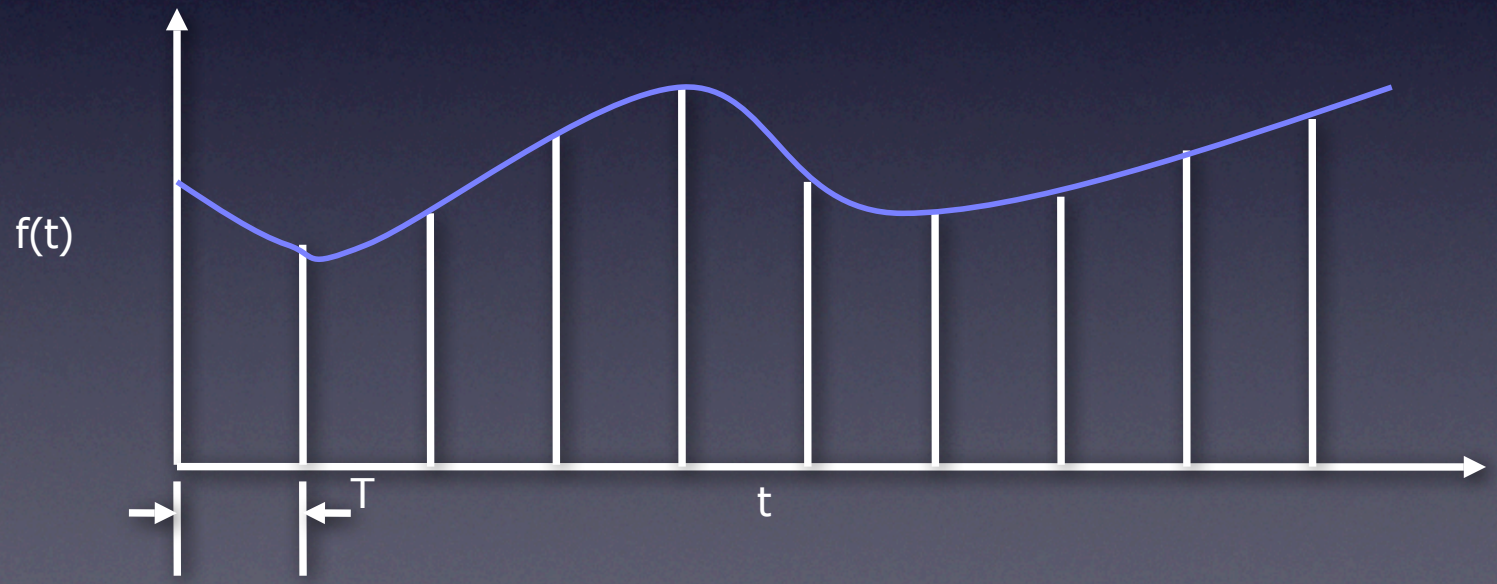


Reconstruction

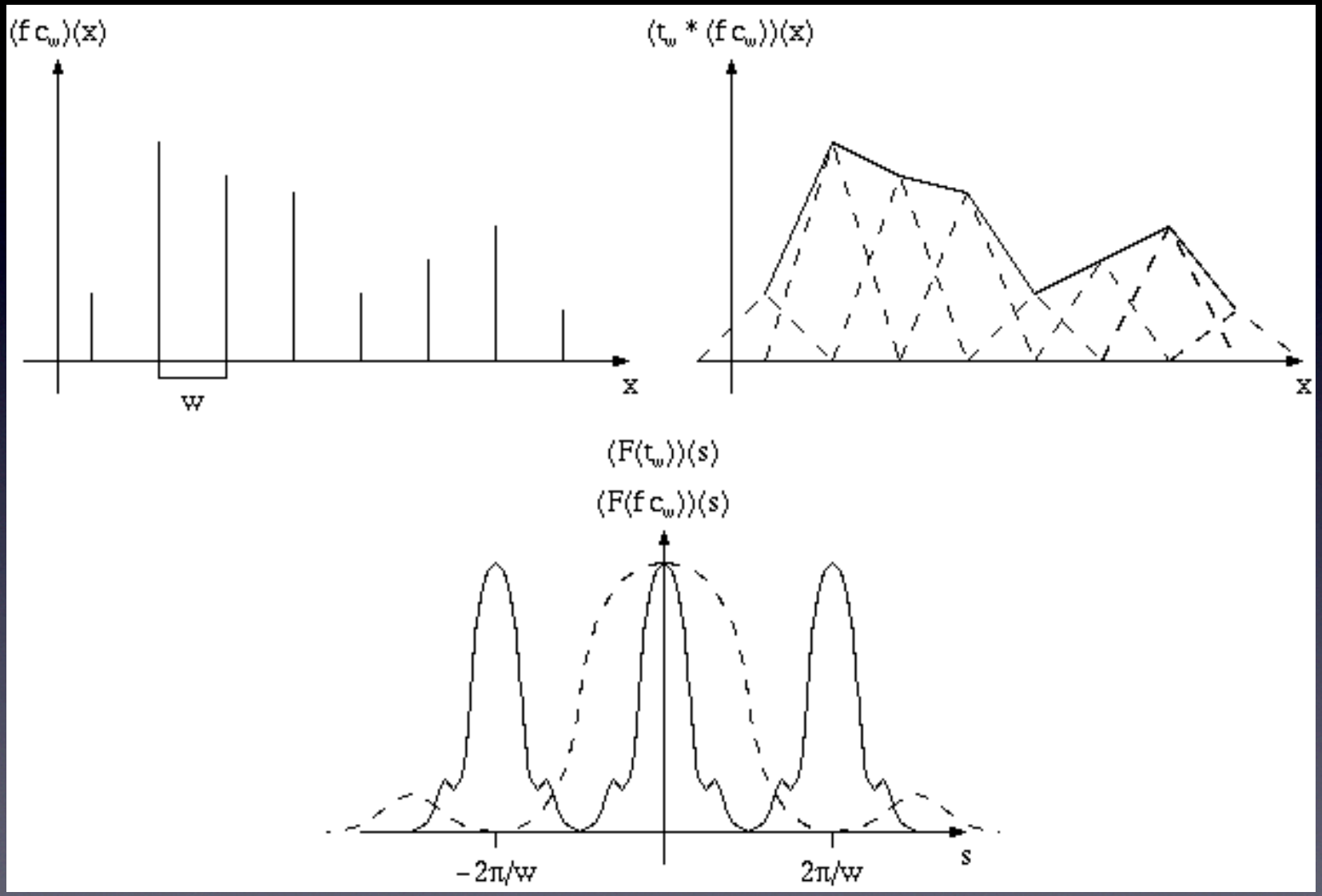
Connect the dots?

$$F(f(t)) = \sum_{s=1}^{n} a_s F\left(T\left(t\right)\right)$$





Triangle reconstruction

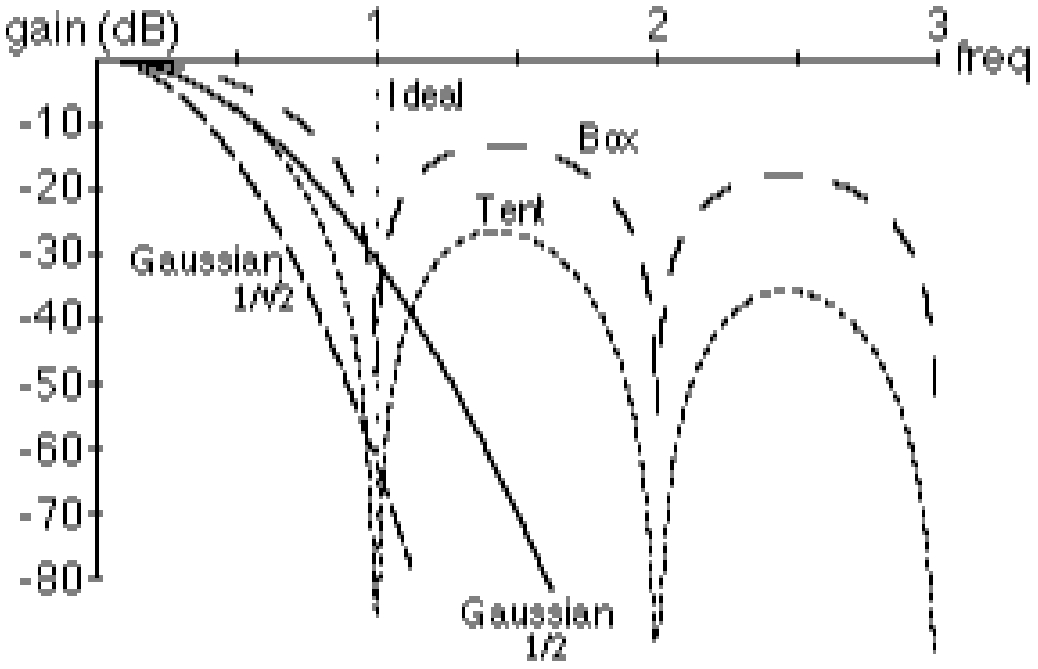


Better reconstruction

- Ideal reconstruction filter: sinc function (infinite in time)
- Bad news: ALL filters that have finite frequency response have infinite time response and vice-versa

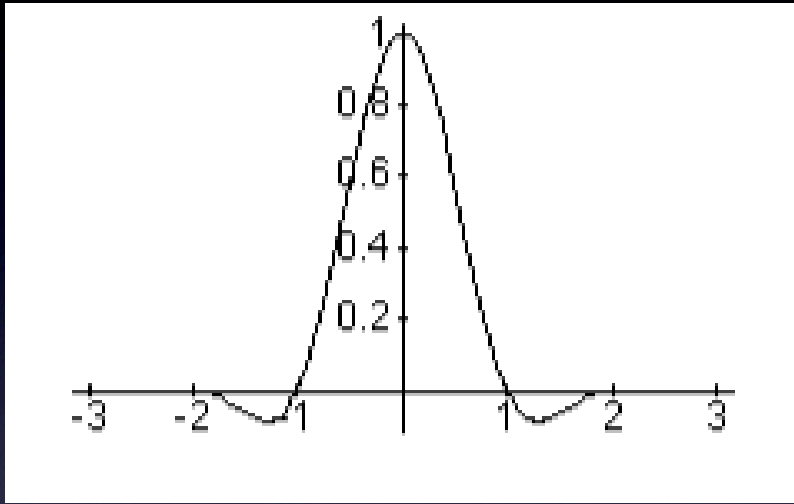
- For a detailed discussion of some of the tradeoffs:
  - Marschner, S.R. and Lobb, R.J., An Evaluation of Reconstruction Filters for Volume Rendering, in Proceedings of Visualization'94, Washington, DC (1994), pp. 100-107

Filter responses

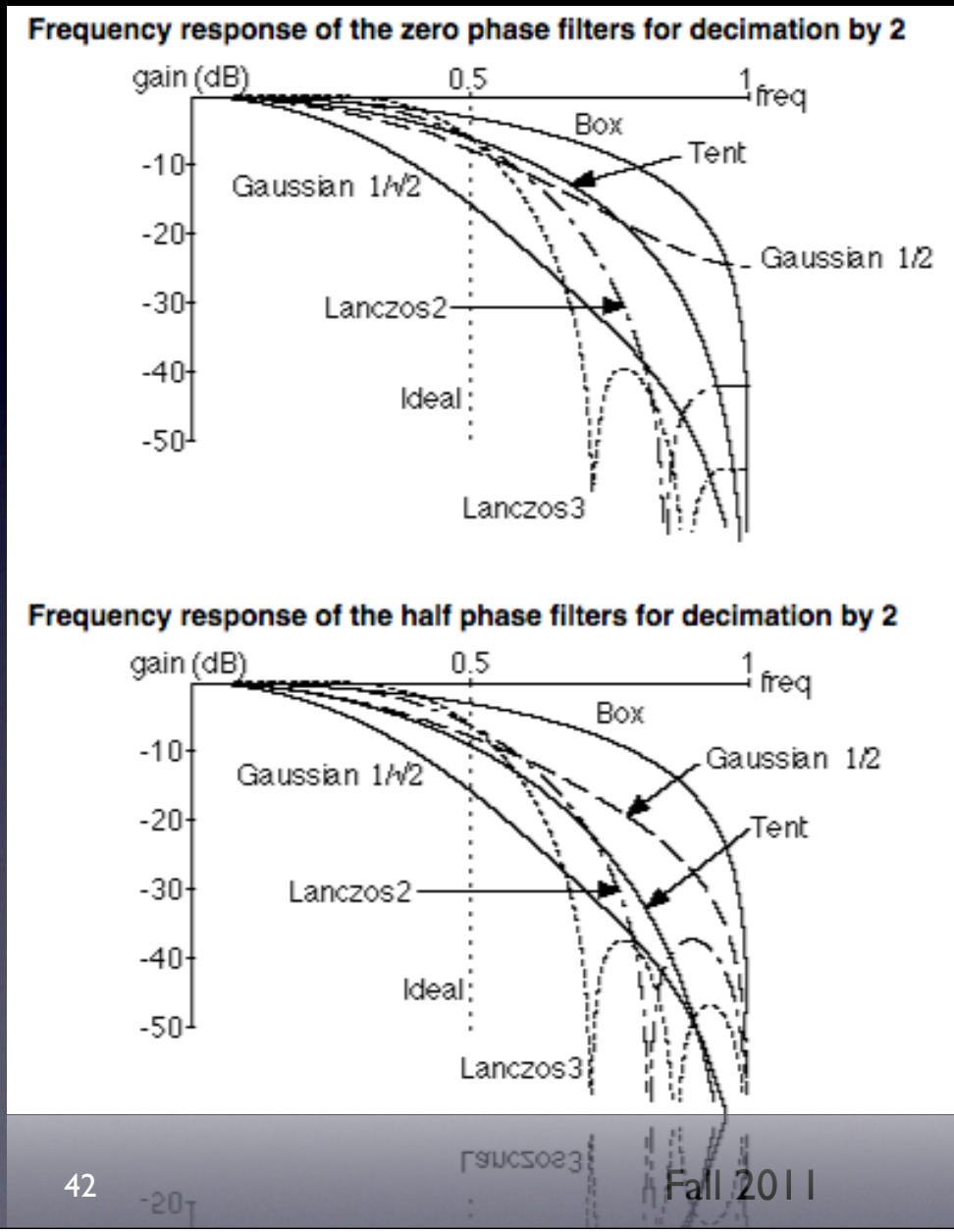


Ken Turkowski, Apple Computer,
"Filters for Common Resampling Tasks"

Windowed Sinc

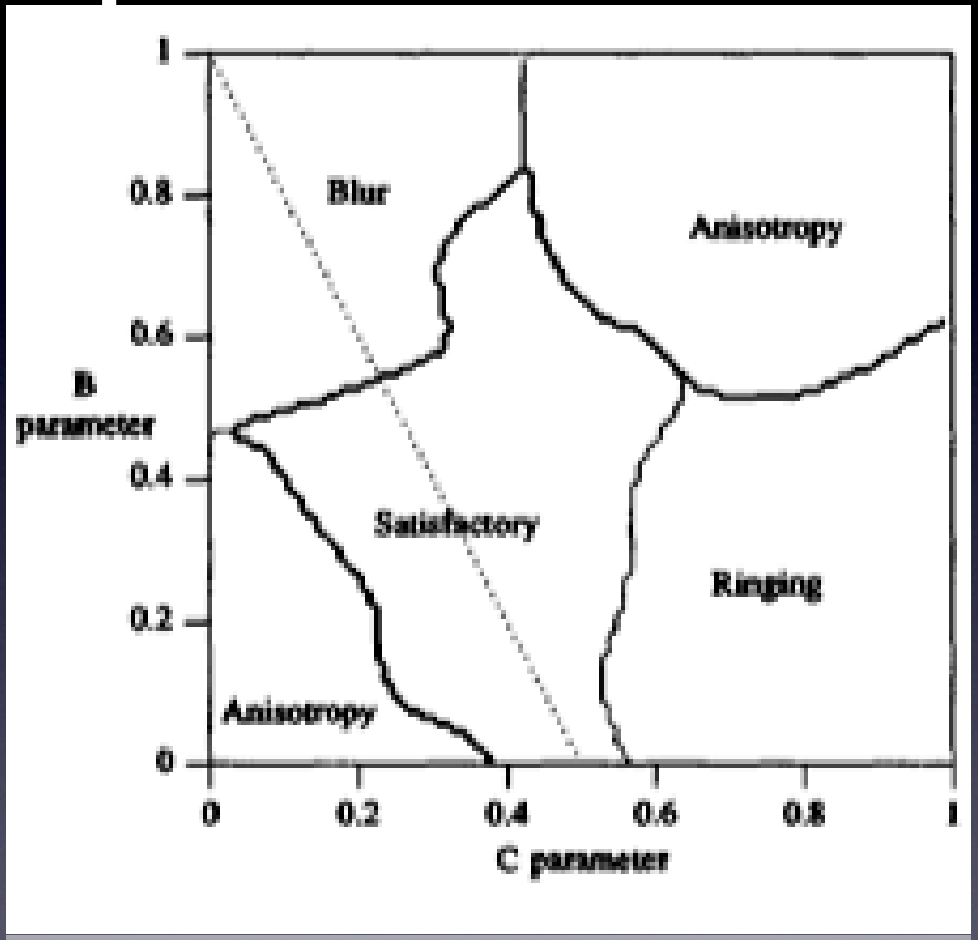


Lanczos-windowed Sinc function



Filter compromises

- Don P. Mitchell, Arun N. Netravali "Reconstruction filters in computer-graphics" Computer Graphics, Volume 22, Number 4, 1988.
- User study with over 500 subjects
- Identified a new filter with good properties (now called the Mitchell filter)



Sampling 2

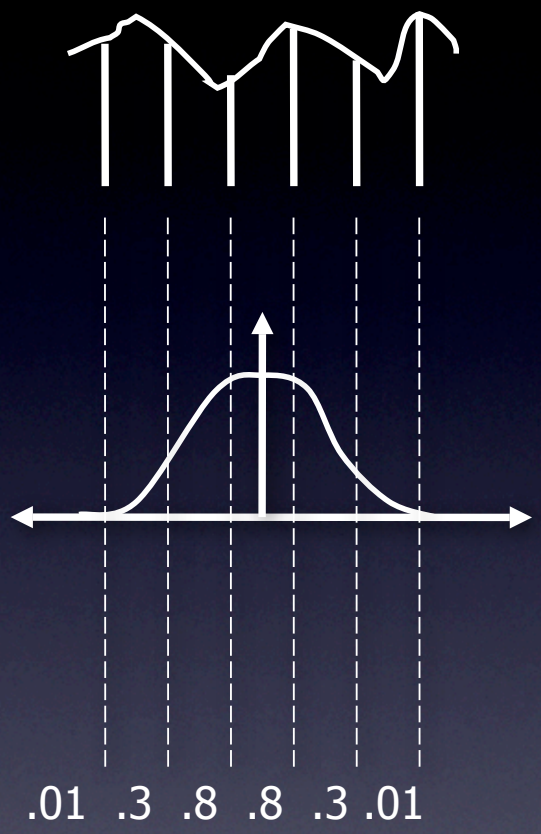
Supersampling pipeline

- World (continuous)
- Fine Samples (discrete)
- Reconstructed world (continuous)
- Image Samples (discrete)
- Reconstructed image (continuous)

Resampling

- To move from one sample rate to another, we use the same filters
- Interpolation: increasing resolution
- Decimation: decreasing resolution
- Pre-aliasing: undersampling
- Post-aliasing: errors in reconstruction

Filtering details (decimation)



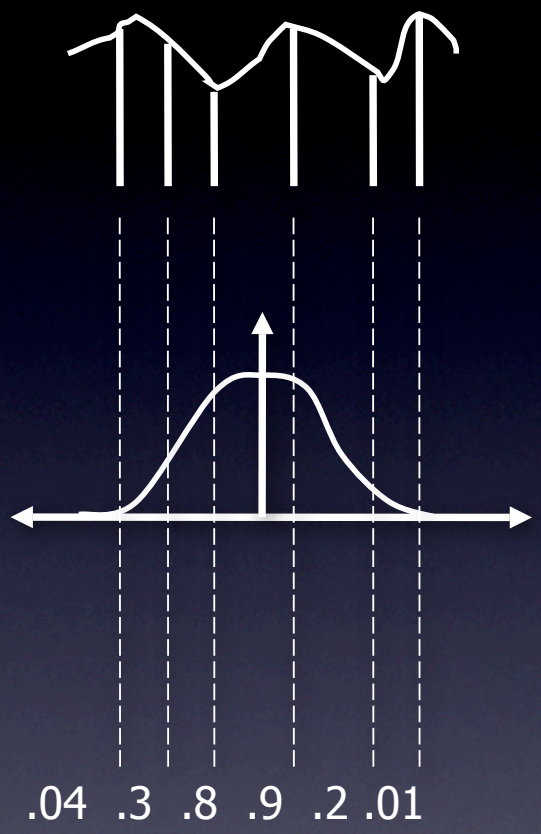
Fine samples

Fine samples

Coarse sample

$$\frac{\sum_{samples} filter(x) f(x)}{\sum_{samples} filter(x)}$$

Filtering details (non-uniform sampling)



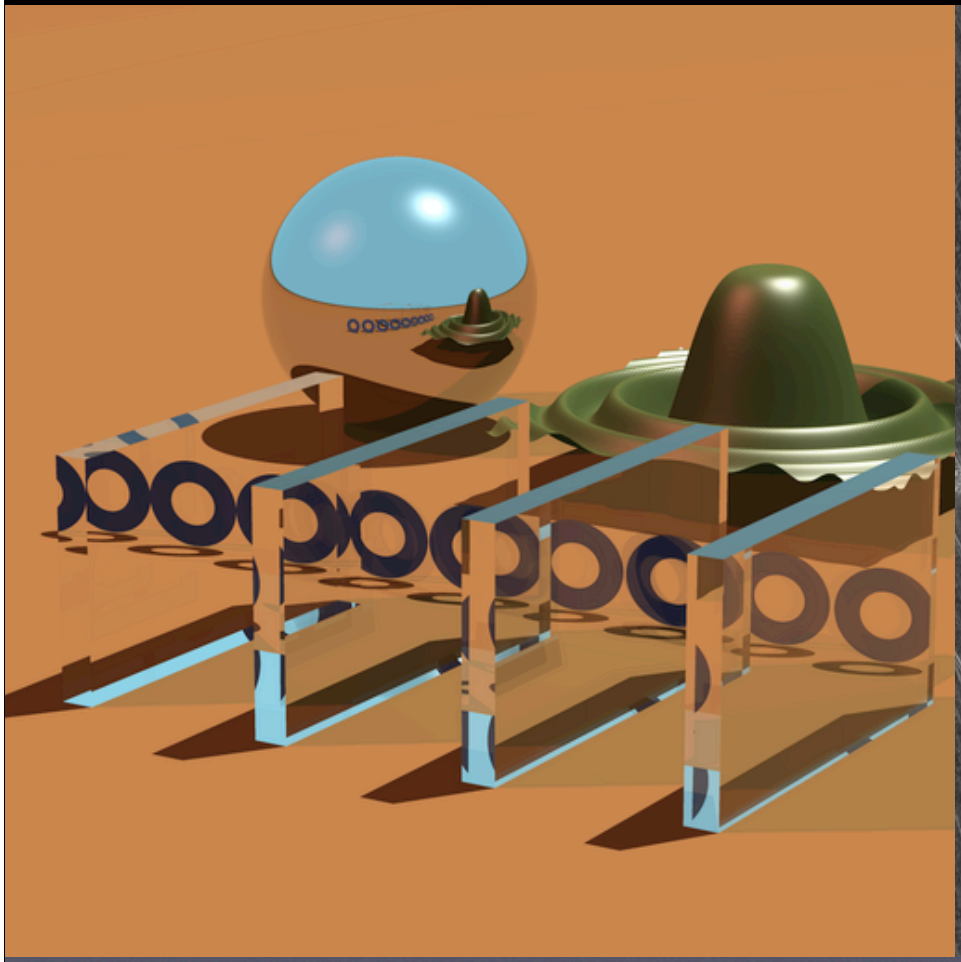
Fine samples

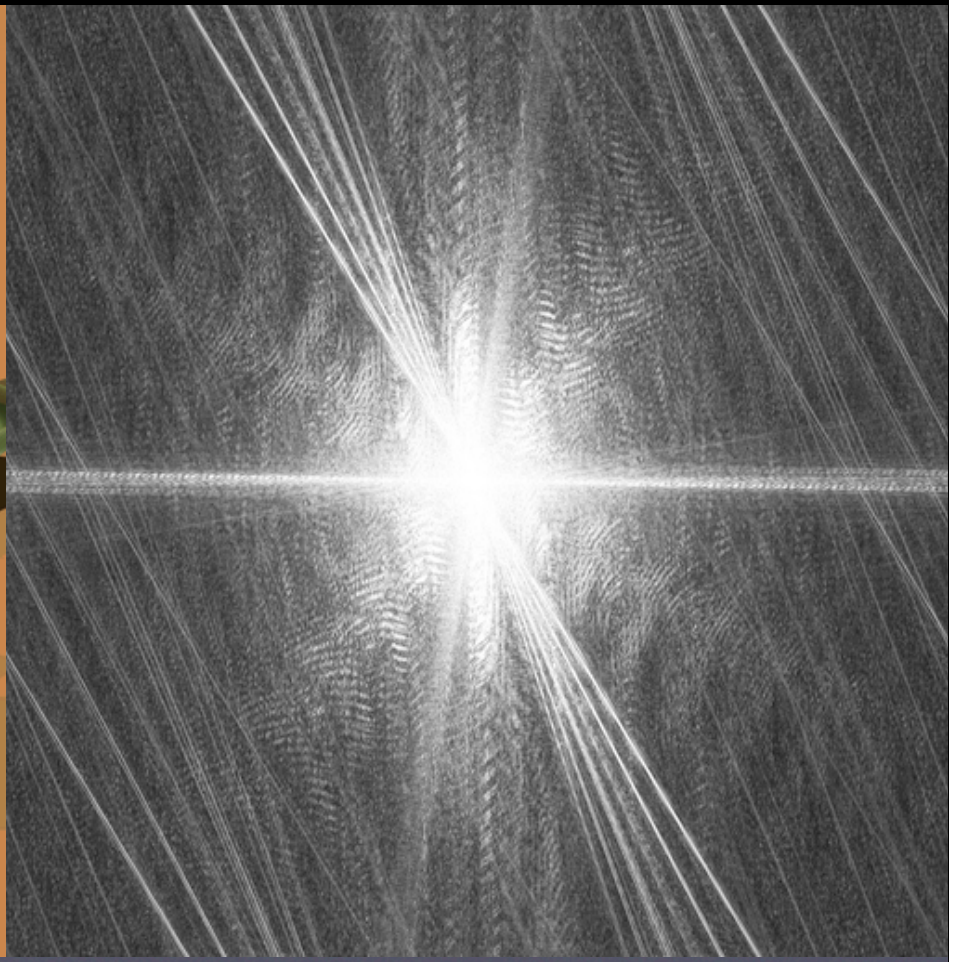
Fine samples

Coarse sample

$$\frac{\sum_{samples} filter(x) f(x)}{\sum_{samples} filter(x)}$$

2D Fourier transform





2D filtering

2D Filtering:

$$\frac{\sum_{samples} filter(x,y) f(x,y)}{\sum_{samples} filter(x,y)}$$

General

2D Separable filter:

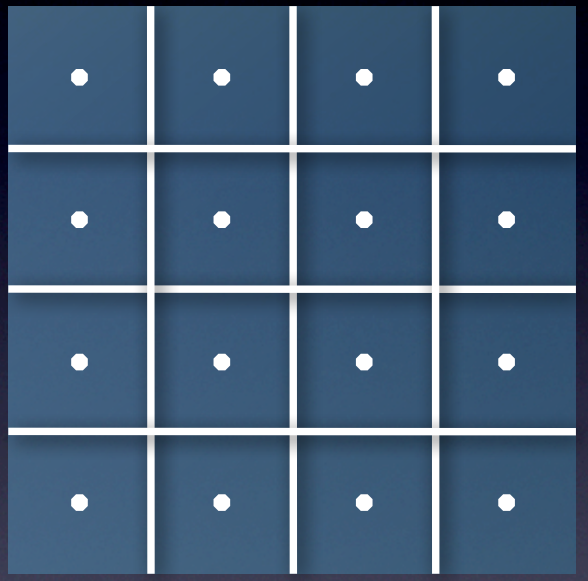
$$filter(x,y) = filter_x(x)\, filter_y(y)$$

$$\frac{\sum_{samples} filter_x(x)\, filter_y(y) f(x,y)}{\sum_{samples} filter_x(x)\, filter_y(y)}$$

Separable

Sampling patterns

Random



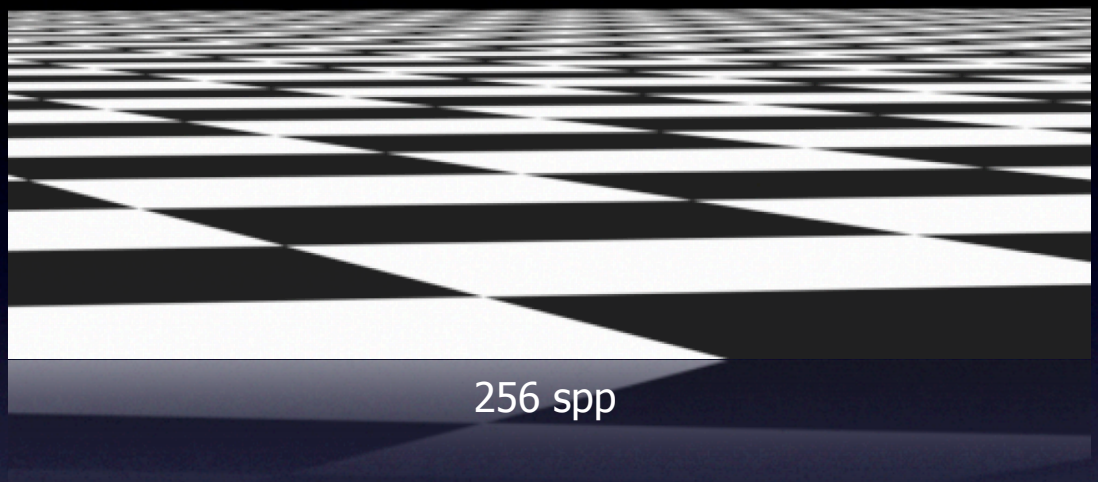
Uniform

Sampling patterns

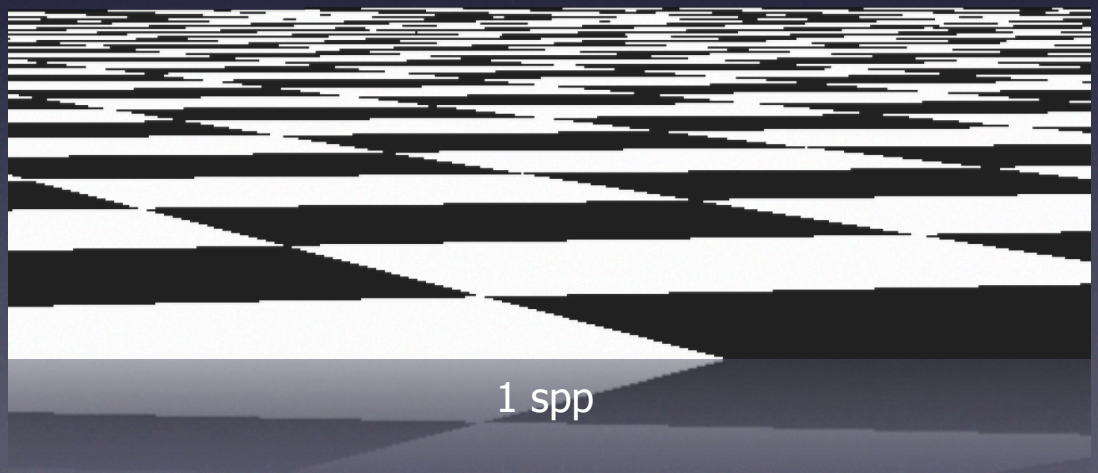
Stratified

n-rooks

Aliasing



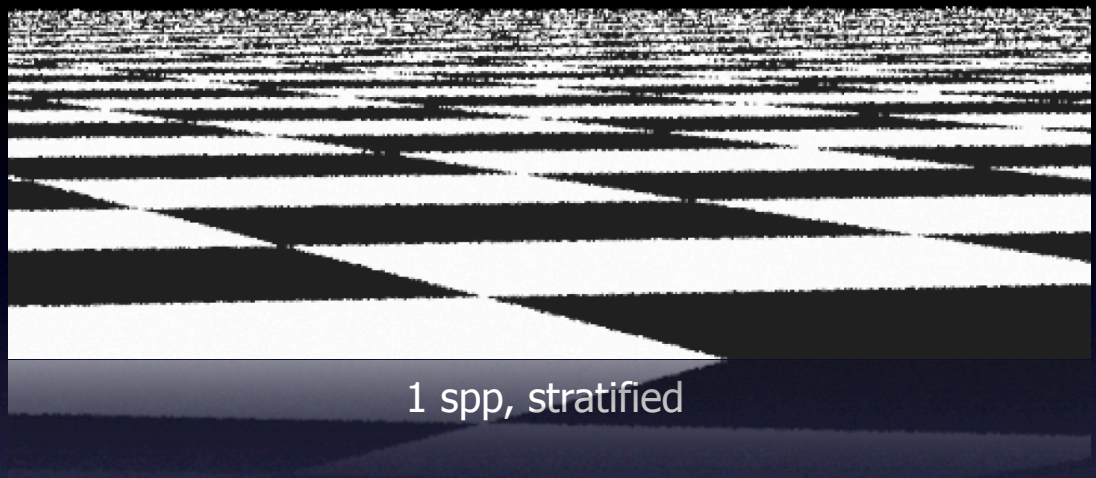
256 spp



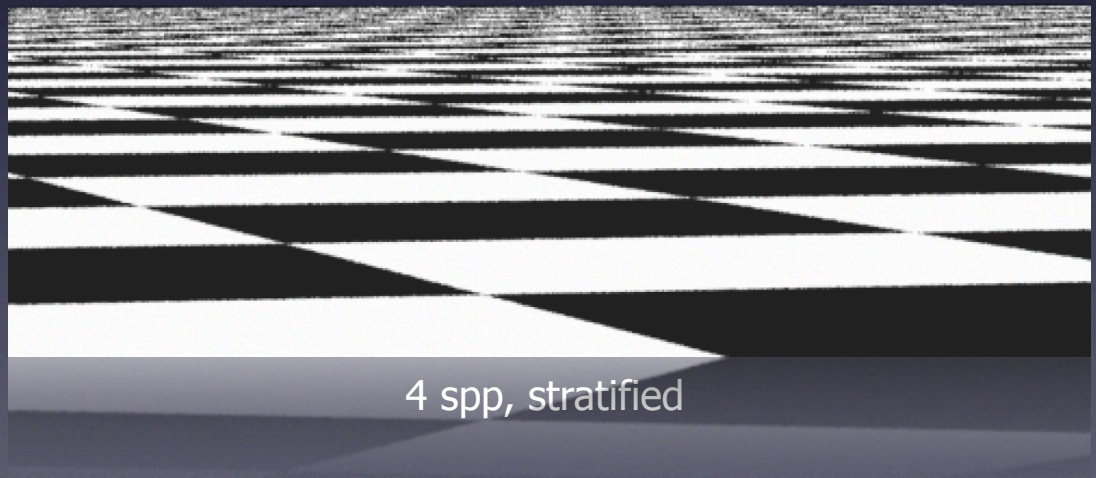
1 spp

Source: *Physically Based Rendering*, Pharr & Humphreys

Aliasing

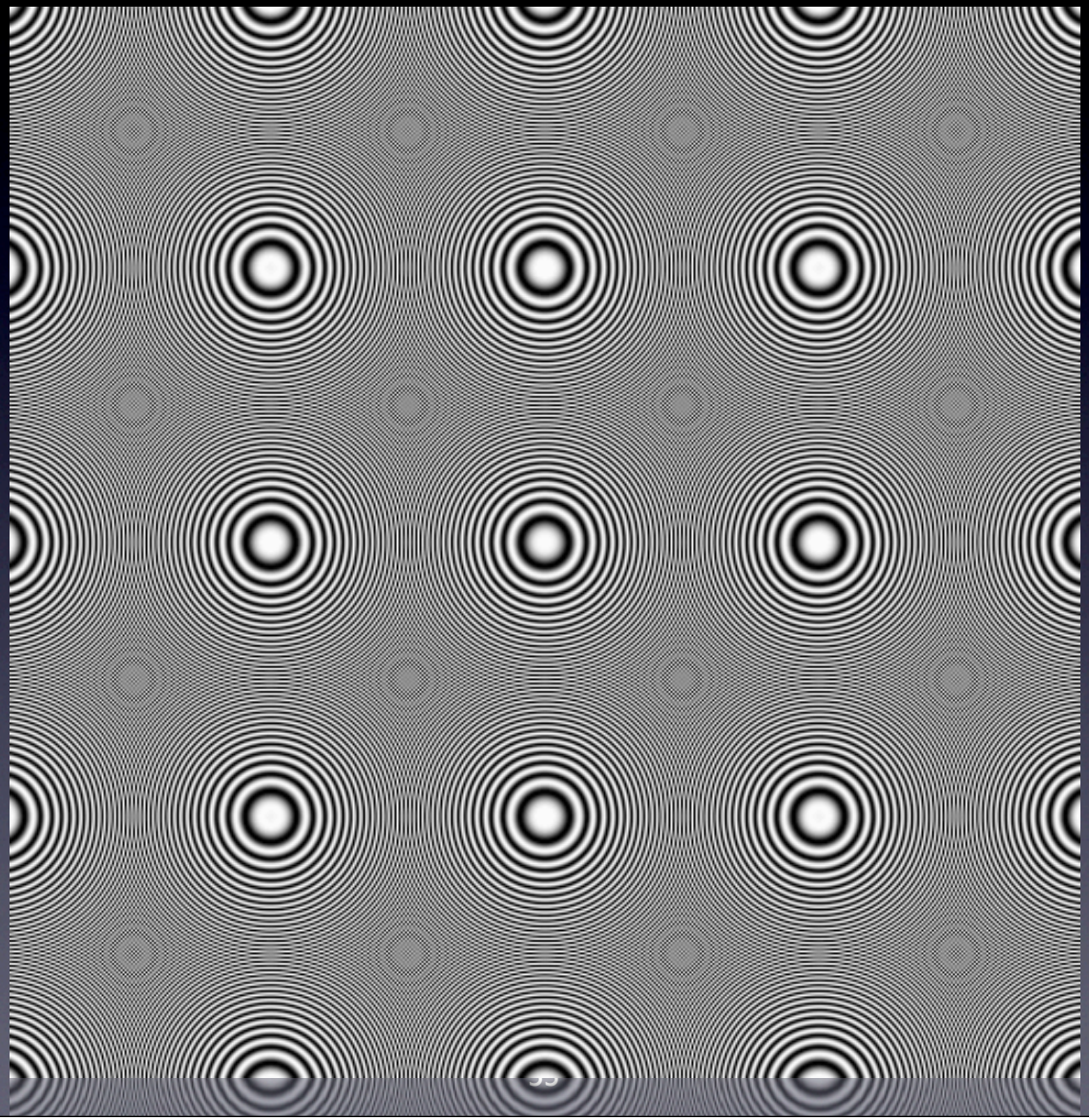


1 spp, stratified

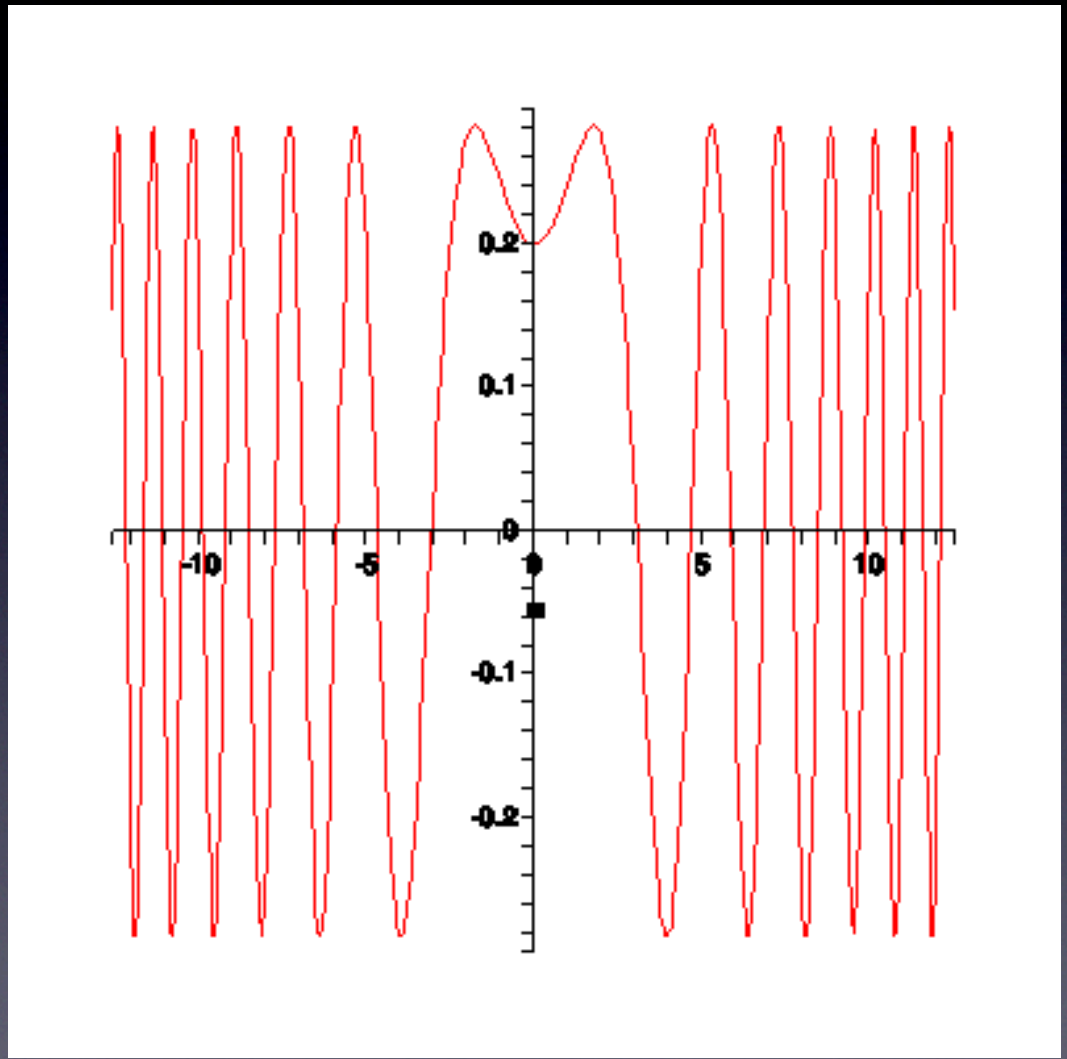


4 spp, stratified

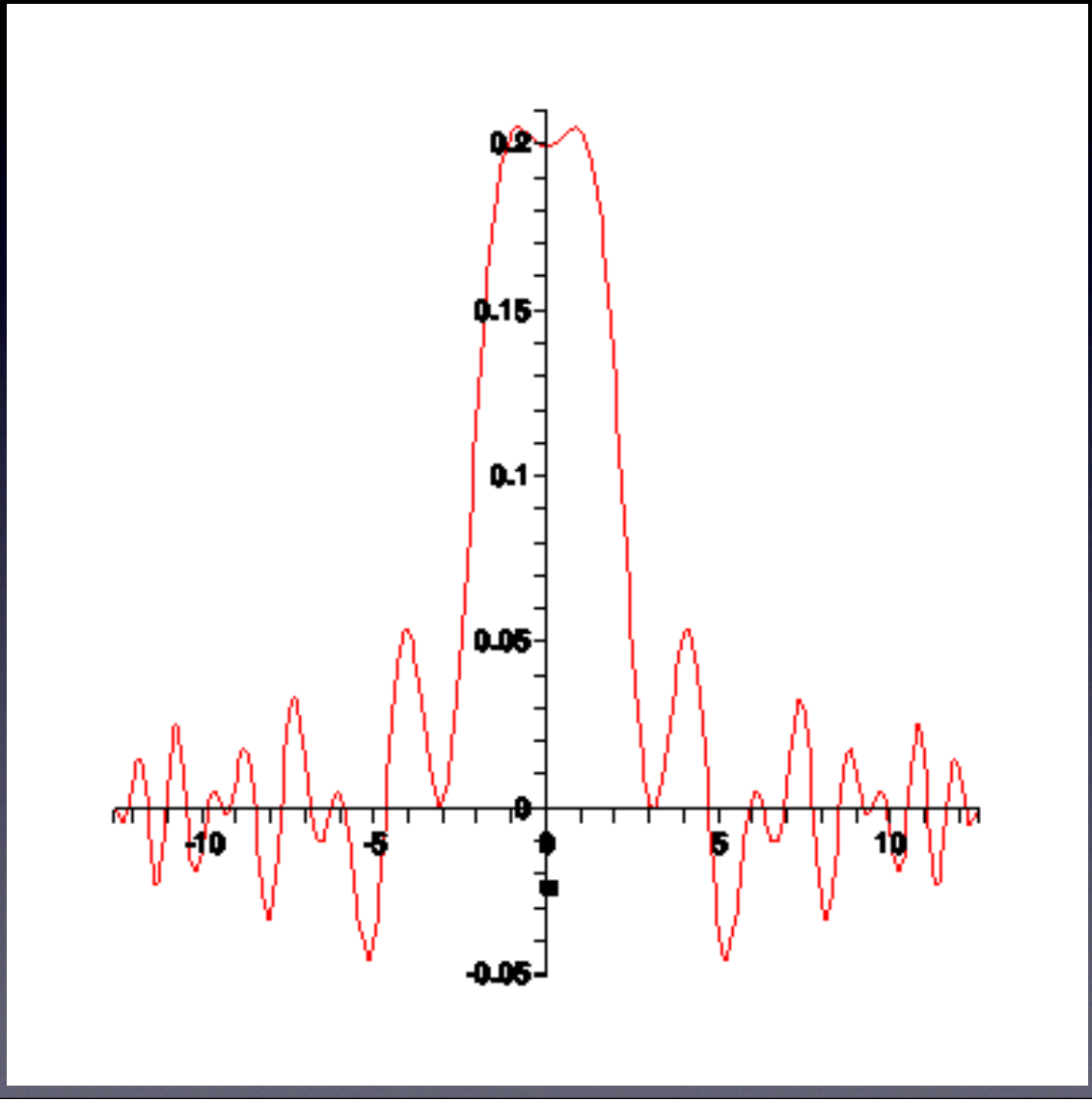
$$f(x,y) = \frac{\sin(500\pi(x^2+y^2))+1}{2}$$



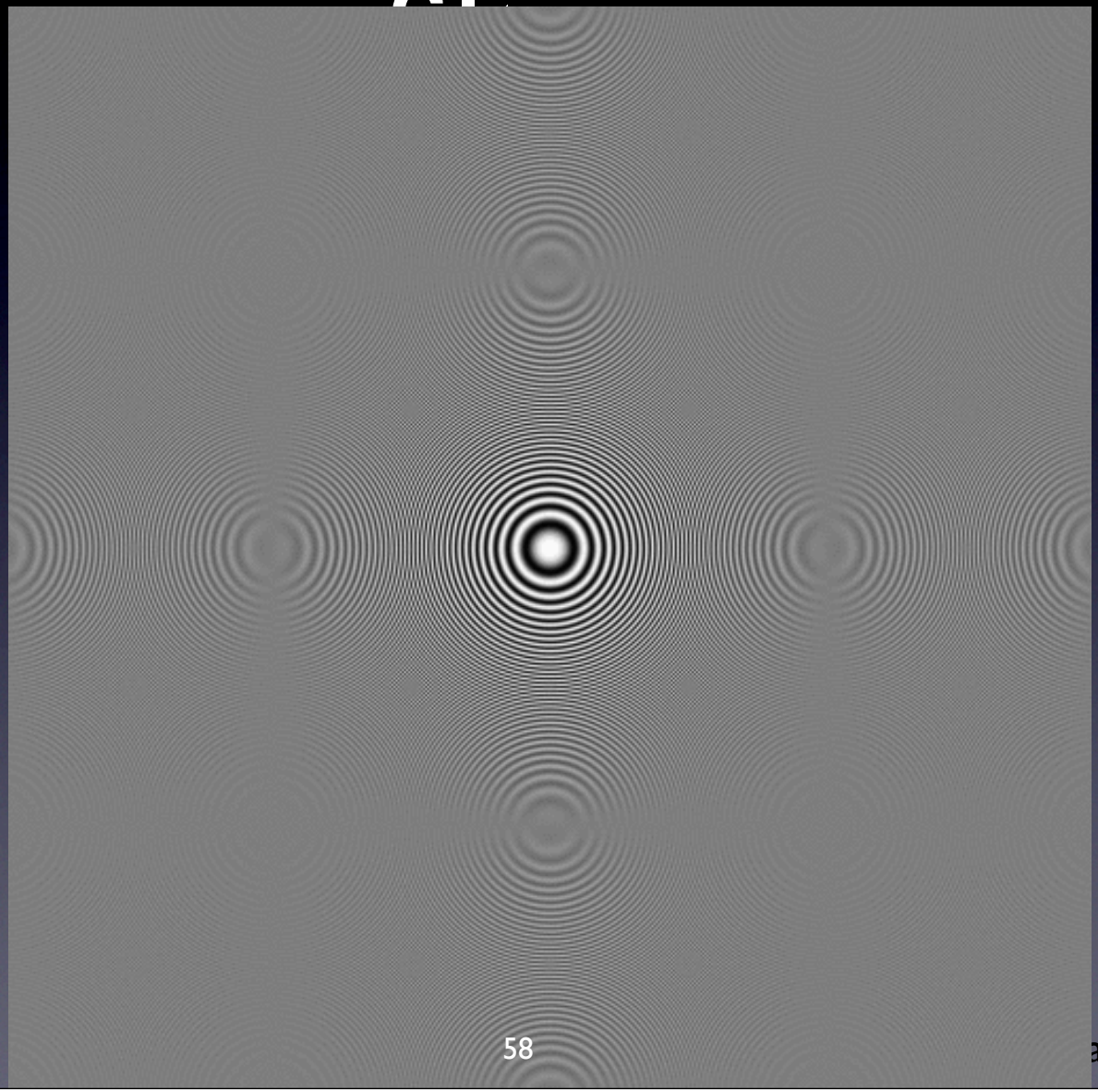
Fourier transform



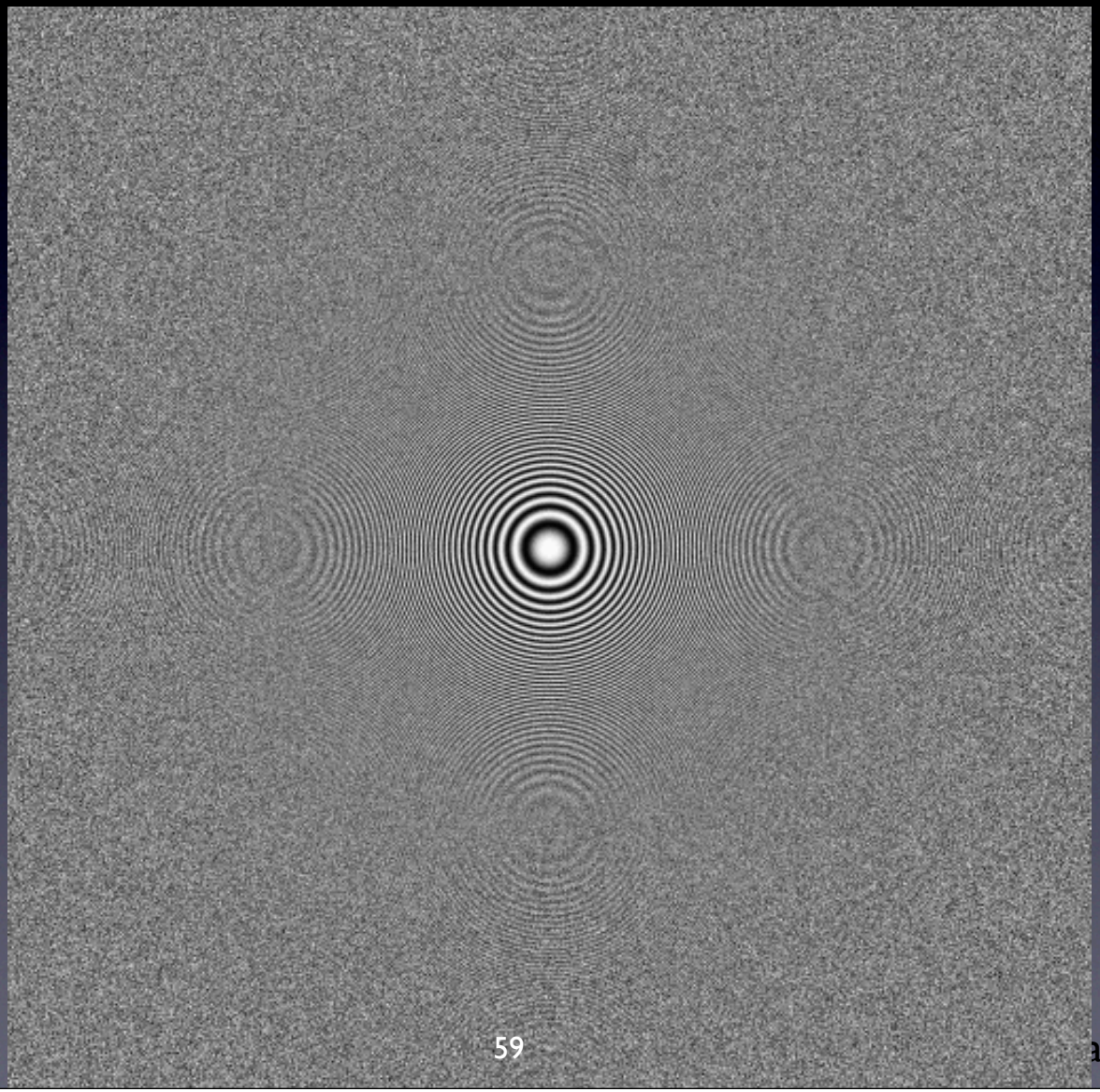
Box filter



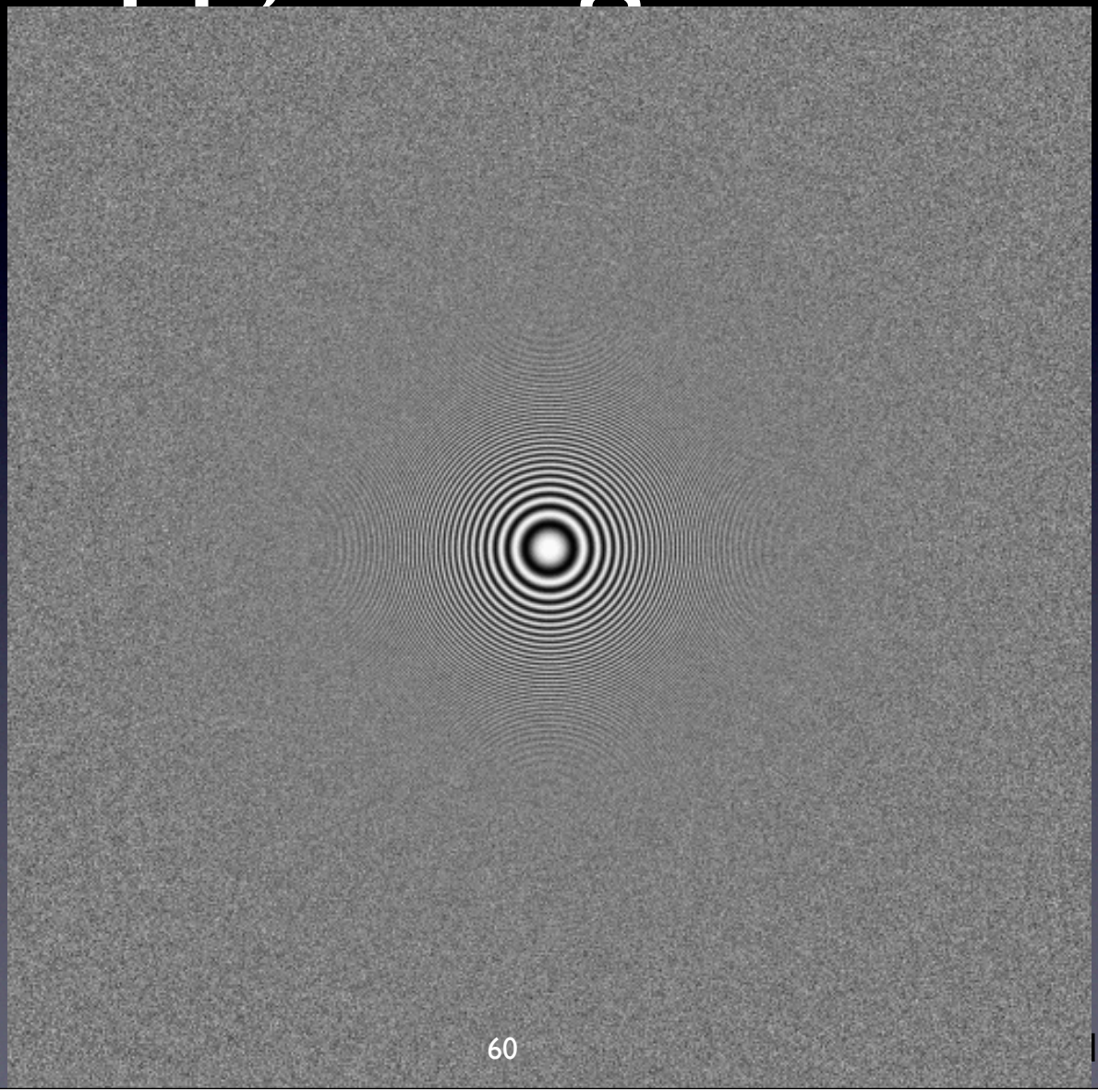
9 uniform samples, box



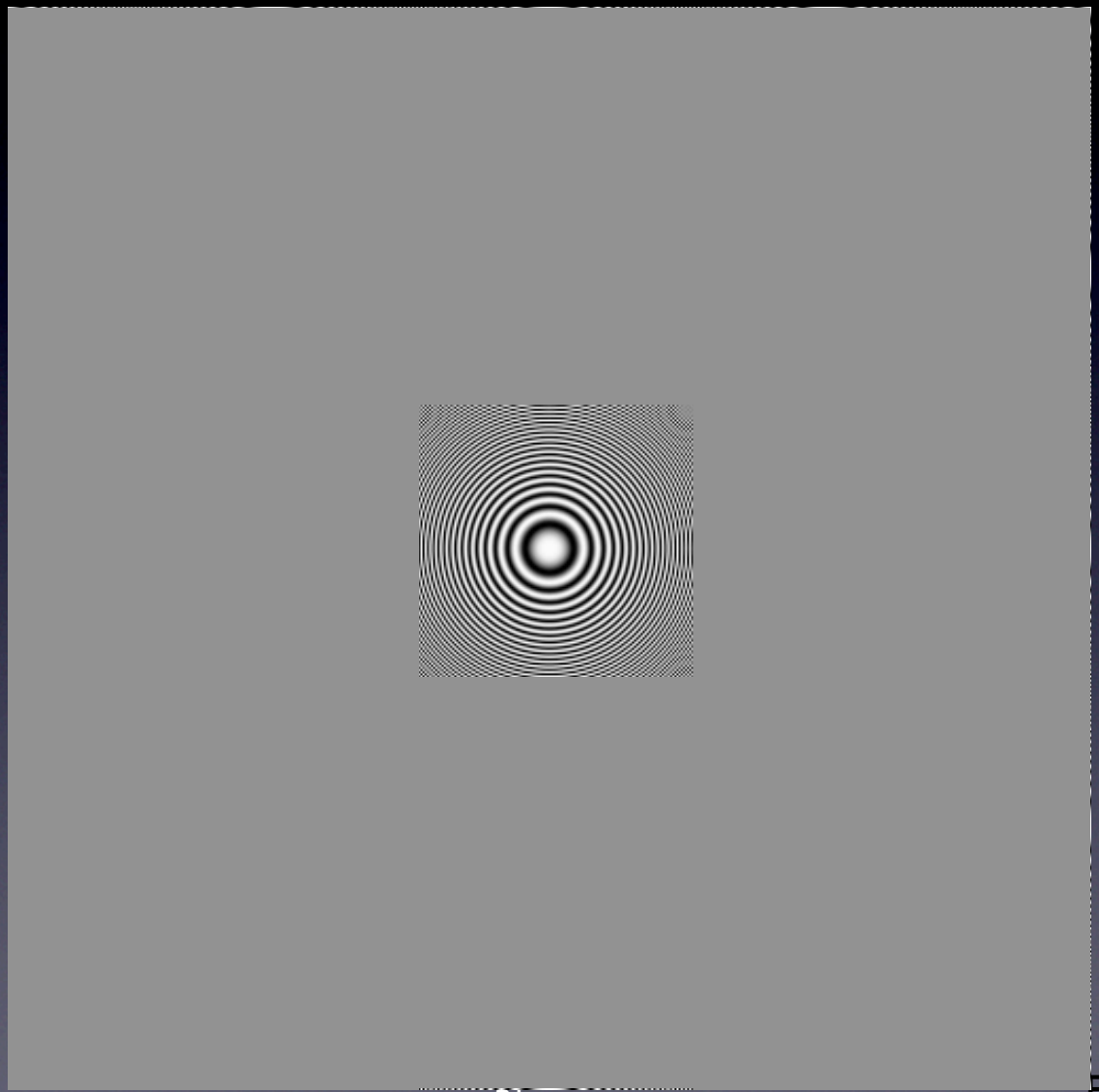
9 jittered samples, box filter



9 spp, triangle filter

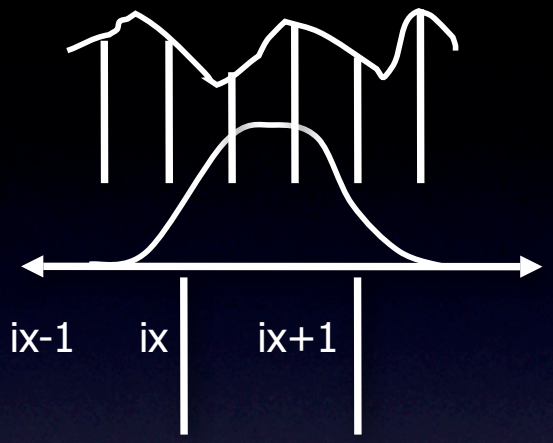


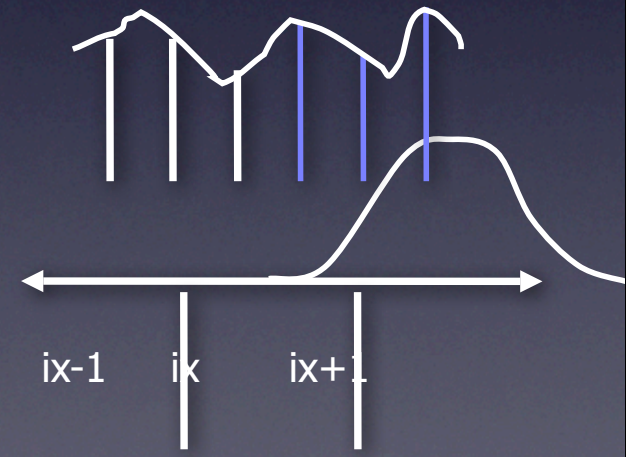
Perfect filter



Overlapping pixels

- Any filter with support > 1 overlaps multiple pixels
- You can reuse samples, but it muddies the architecture
- If you don't, it limits use of high order filters
- Alternatively: generate samples with a distribution

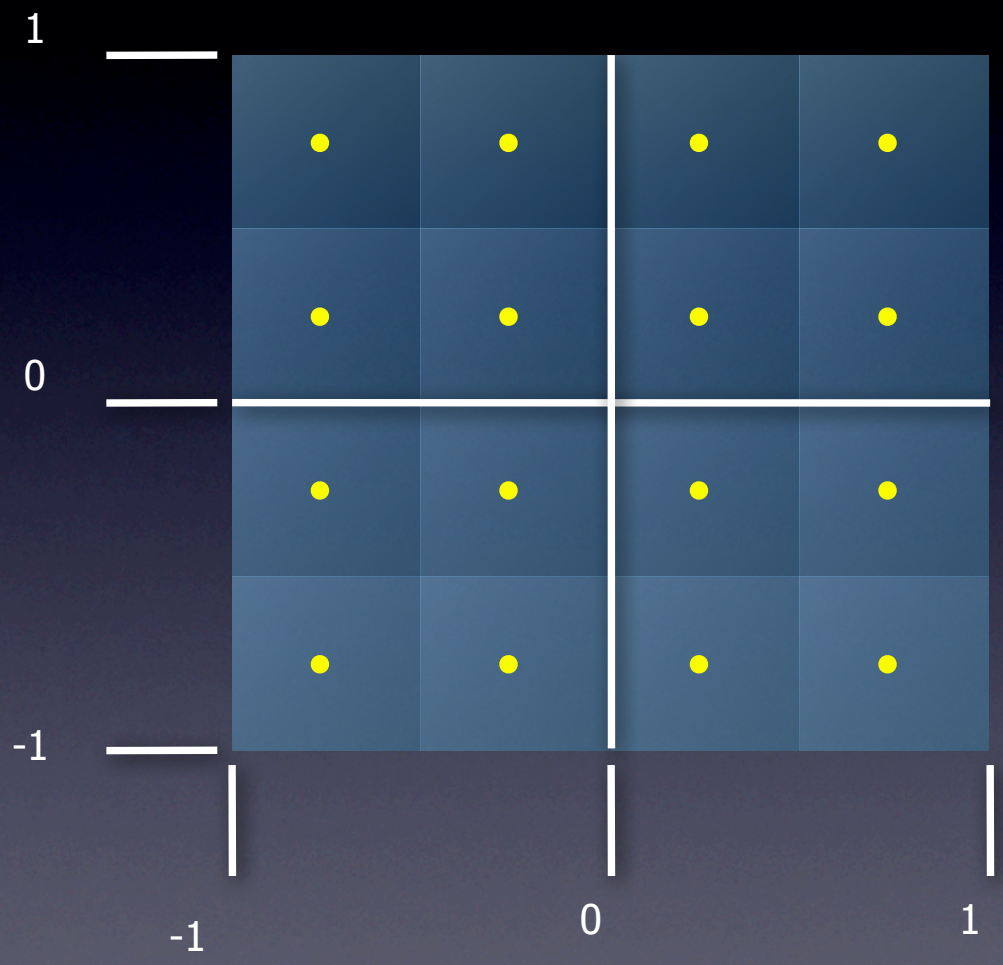




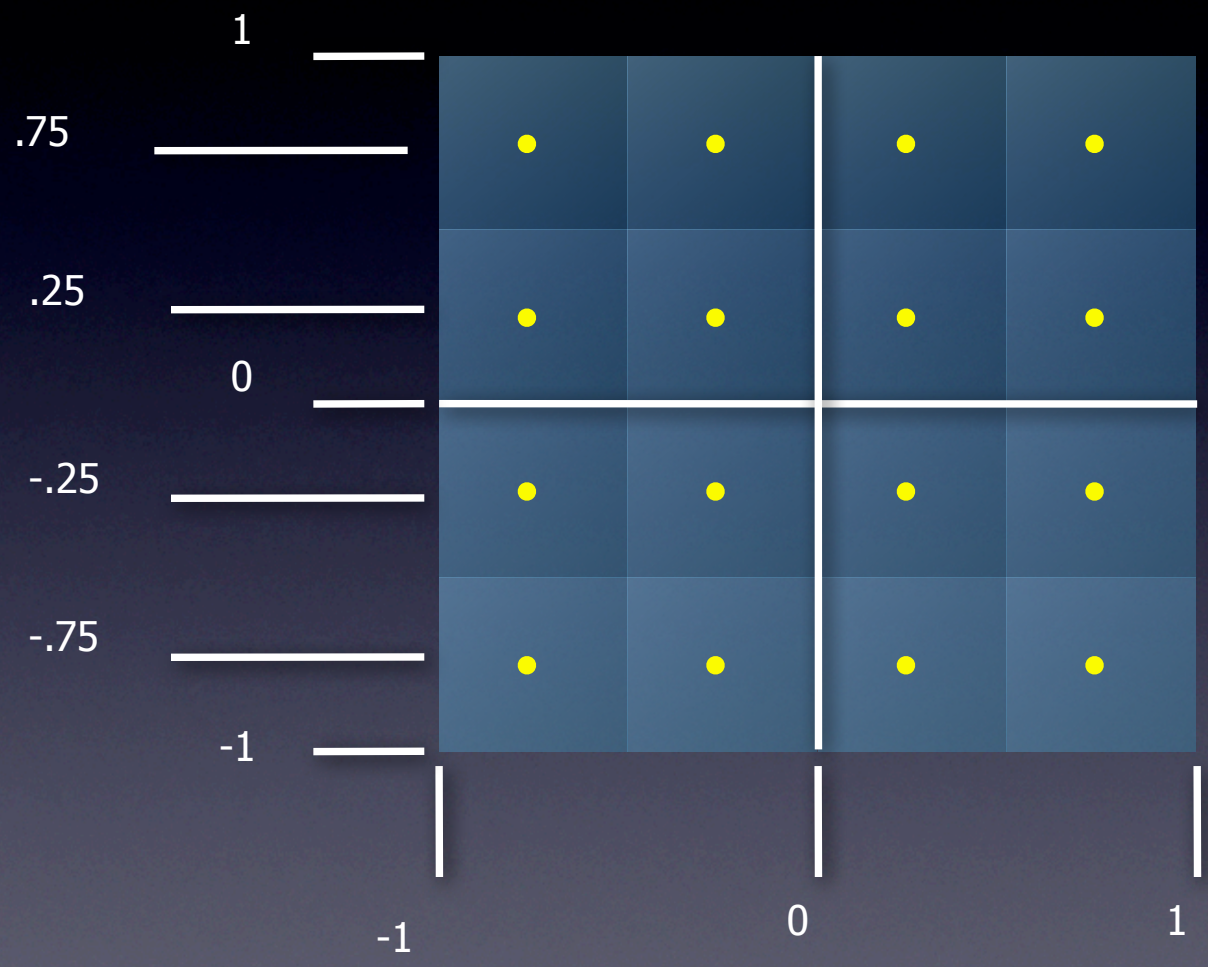
Implementation notes:

- Draw graphs of your filters before using them
- Try a box filter first
- Hardest part: get subpixel coordinates right

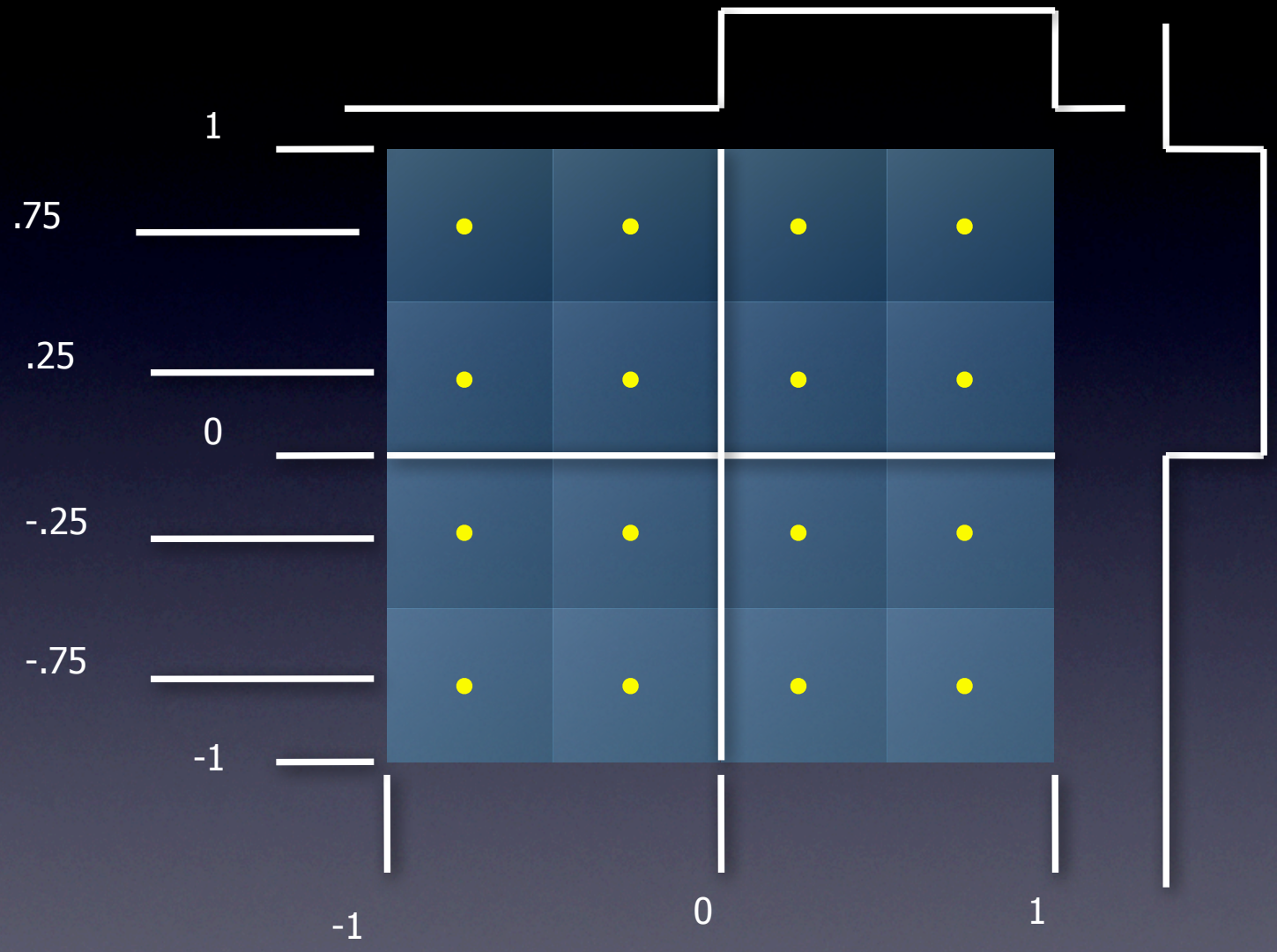
Pixel coordinate clarification



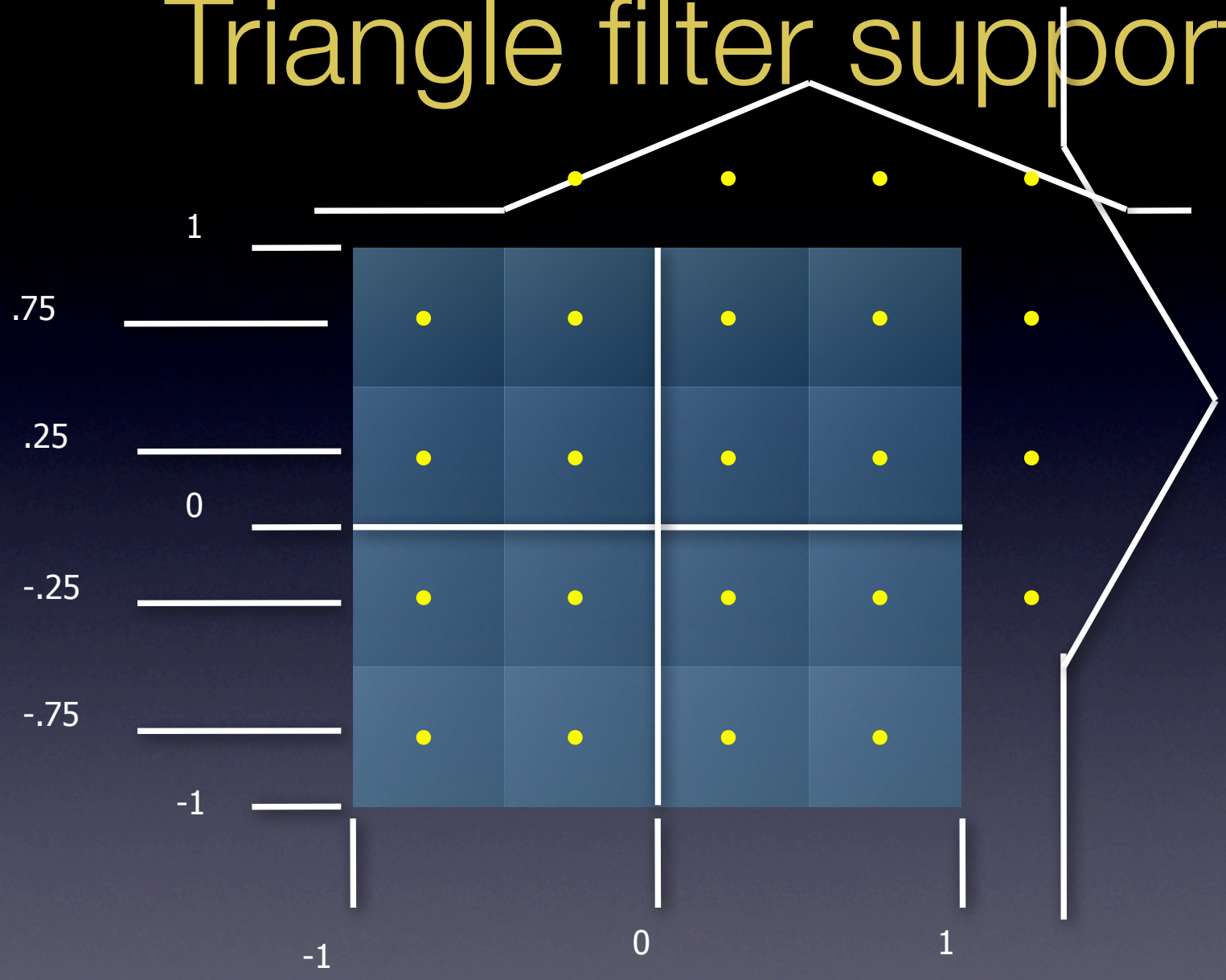
Pixel coordinate clarification



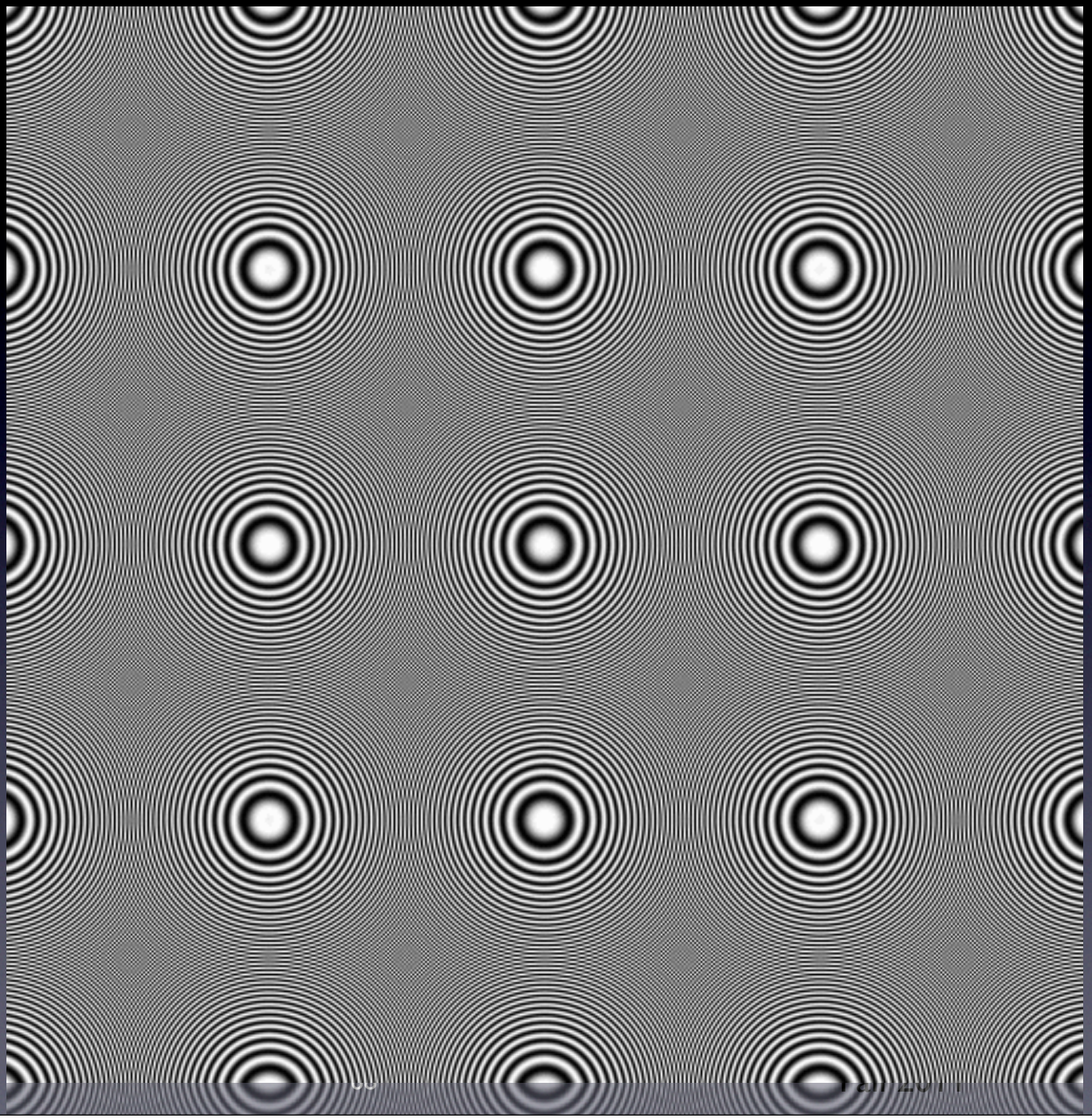
Box filter support



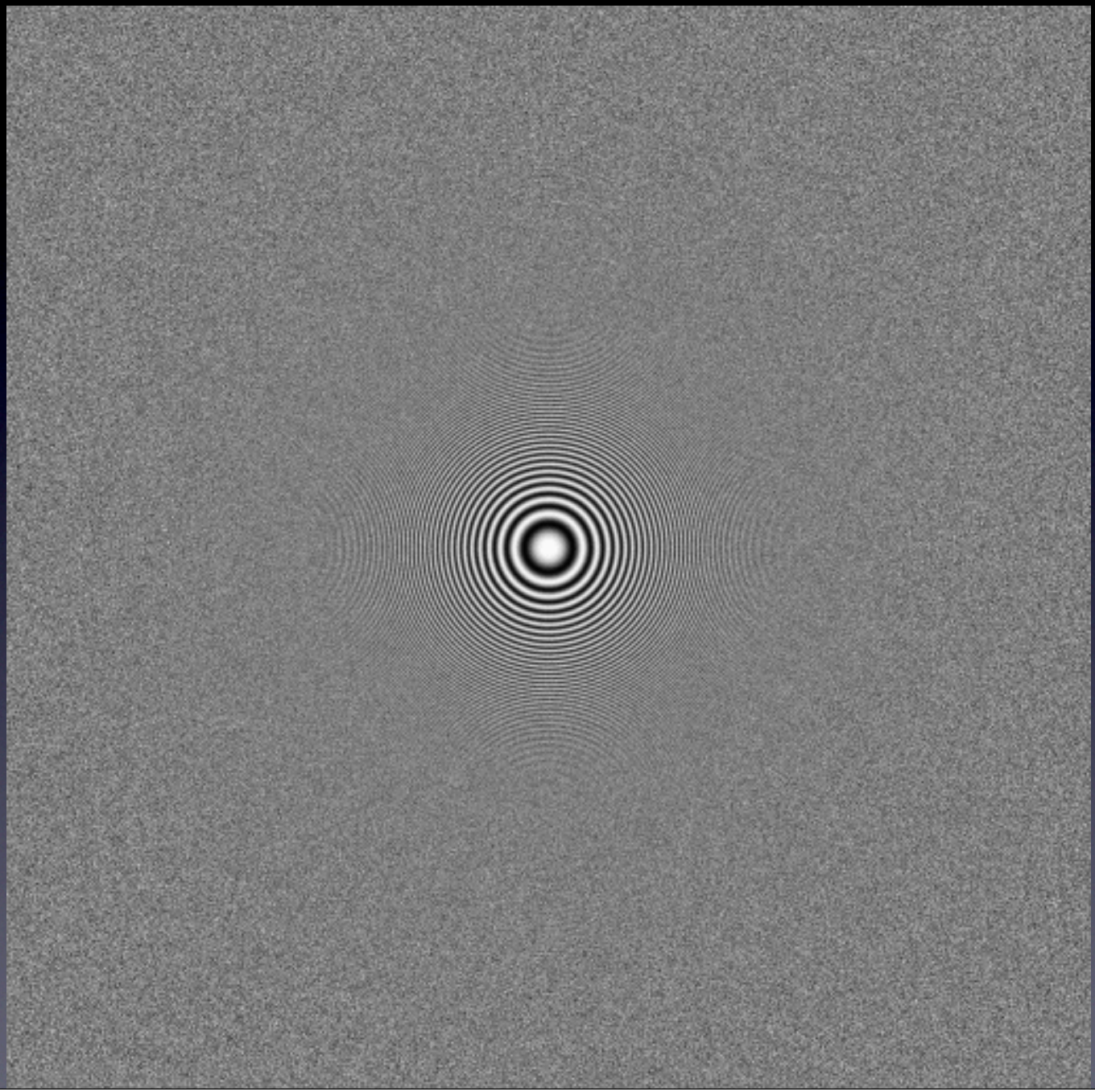
Triangle filter support



1 spp



9 spp jittered triangle



Adaptive supersampling

- Images look better with more samples
- More samples take longer
- Can we put the samples where we need them?

Statistical approach

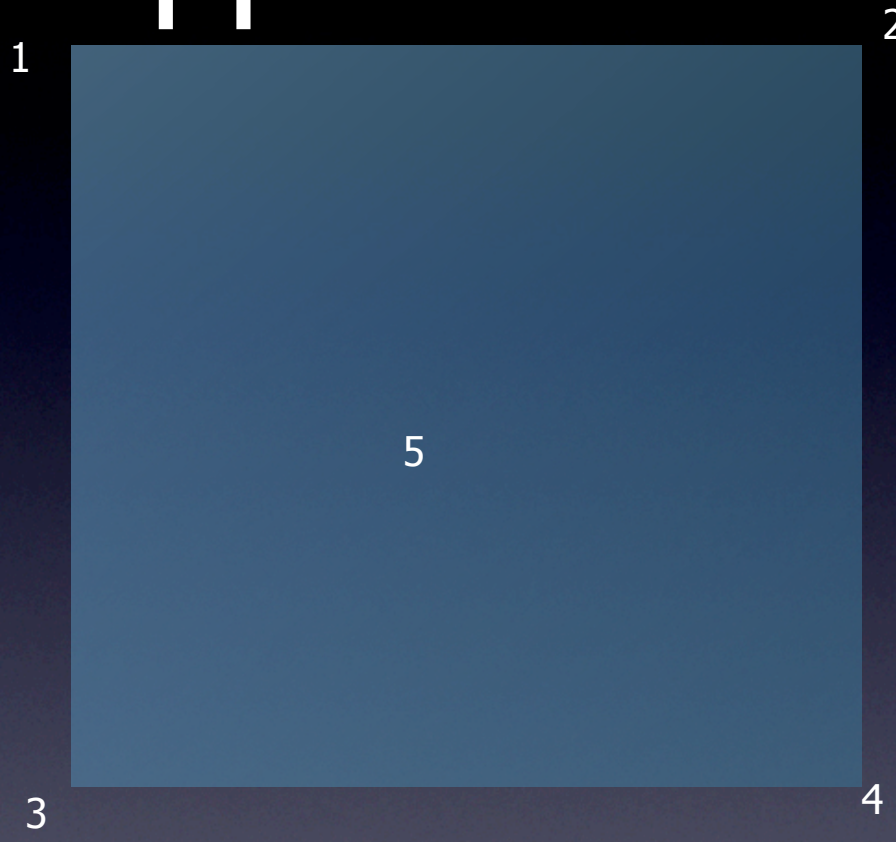
- Compute variance of samples
- Add more samples if variance is too large
- Confidence interval: 90% sure that average is within some range (+/- 1 lsb)
- Theoretically, you must throw away old samples
- You can still miss things

Improved statistical approach

- Discontinuity of sampling rate introduces artifacts
- Two pass algorithm:
  - Normal statistical supersampling
  - Go back over pixels near pixels with high sample rates
- Could repeat

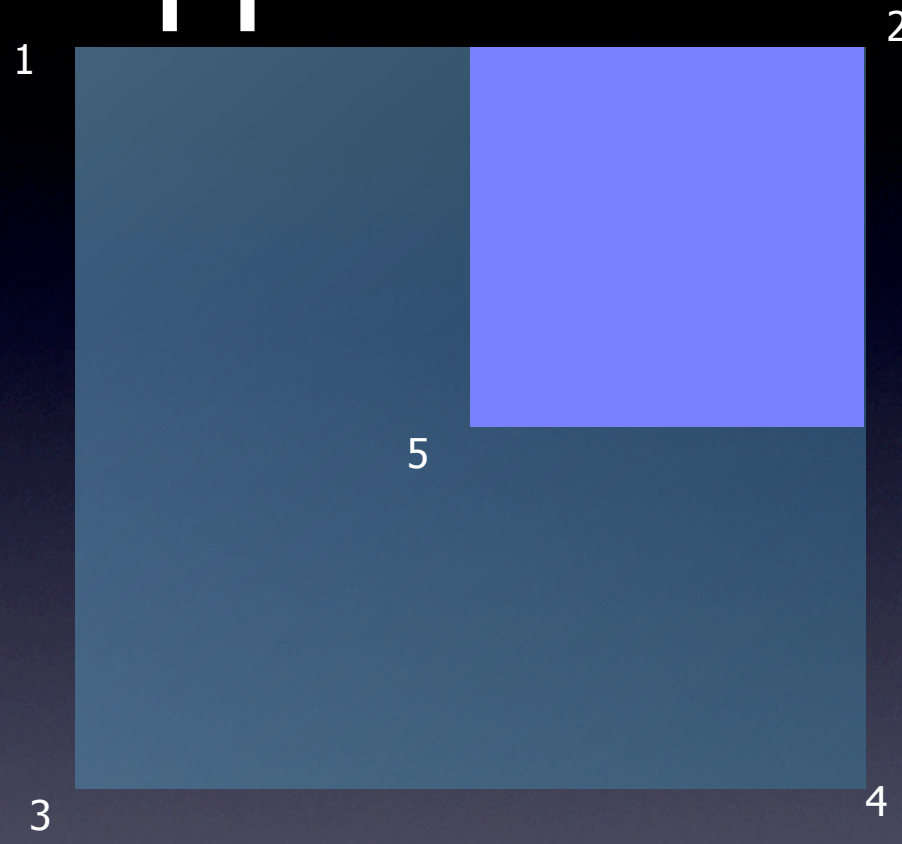
Geometric approach

- 4 samples: corners + center
- Recursively subdivide pixels: (add three more)
- Final color: area weighted average
- Duplicates on edges?



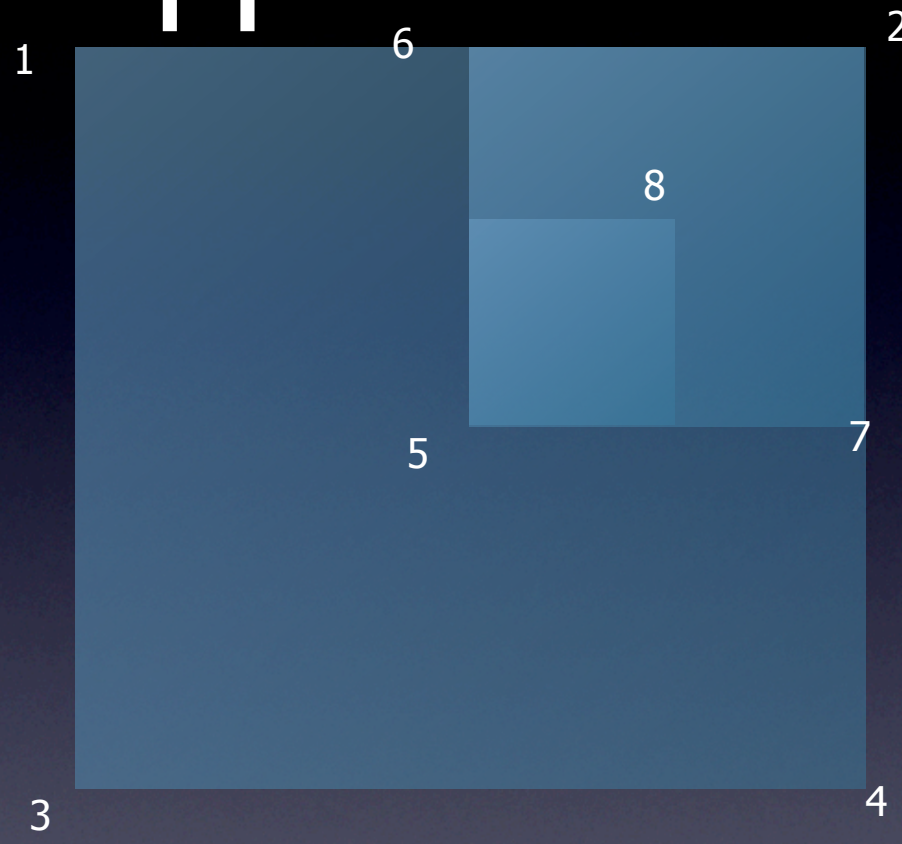
Geometric approach

- 4 samples: corners + center
- Recursively subdivide pixels: (add three more)
- Final color: area weighted average
- Duplicates on edges?

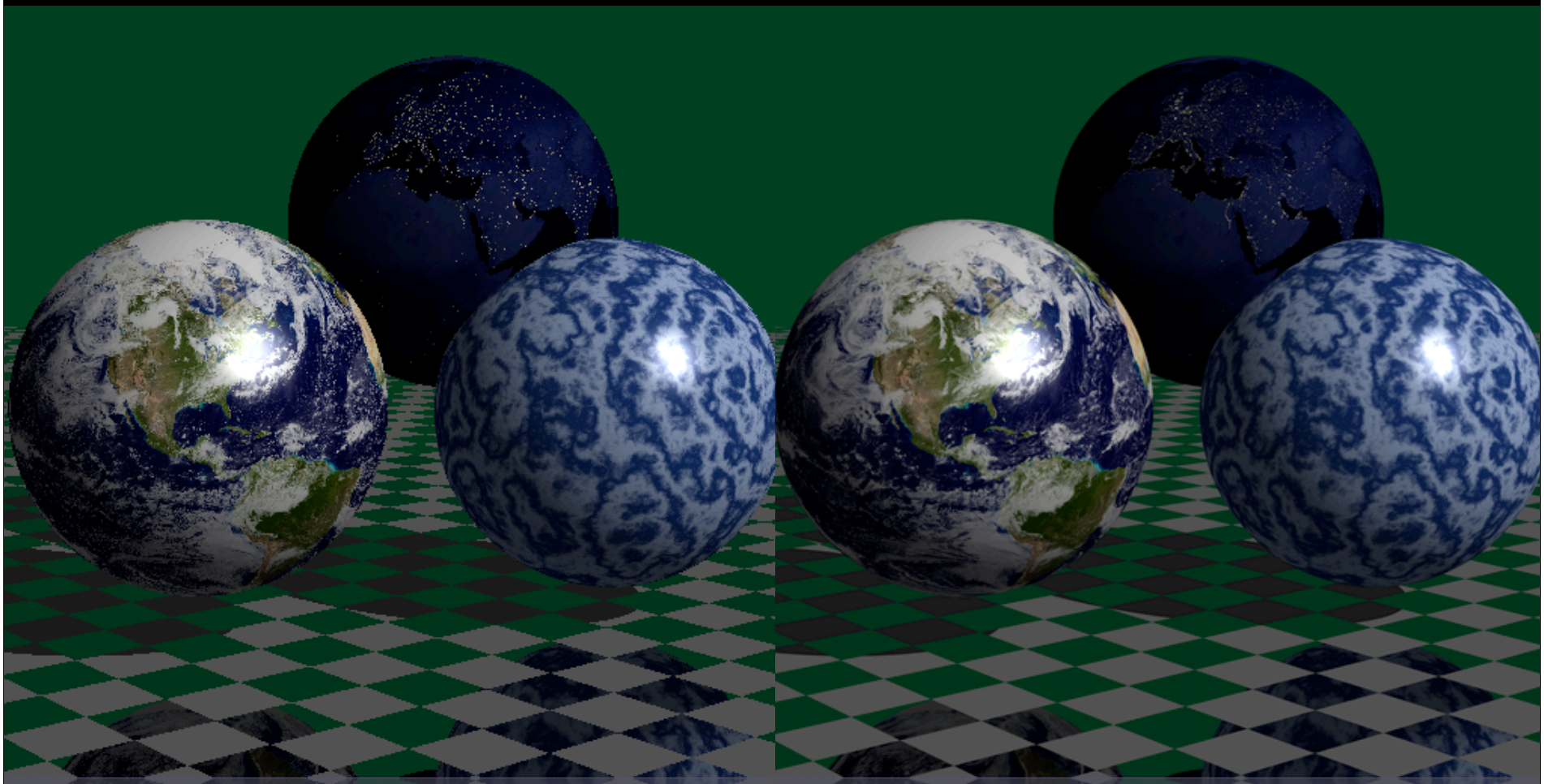


Geometric approach

- 4 samples: corners + center
- Recursively subdivide pixels: (add three more)
- Final color: area weighted average
- Duplicates on edges?



Comparison

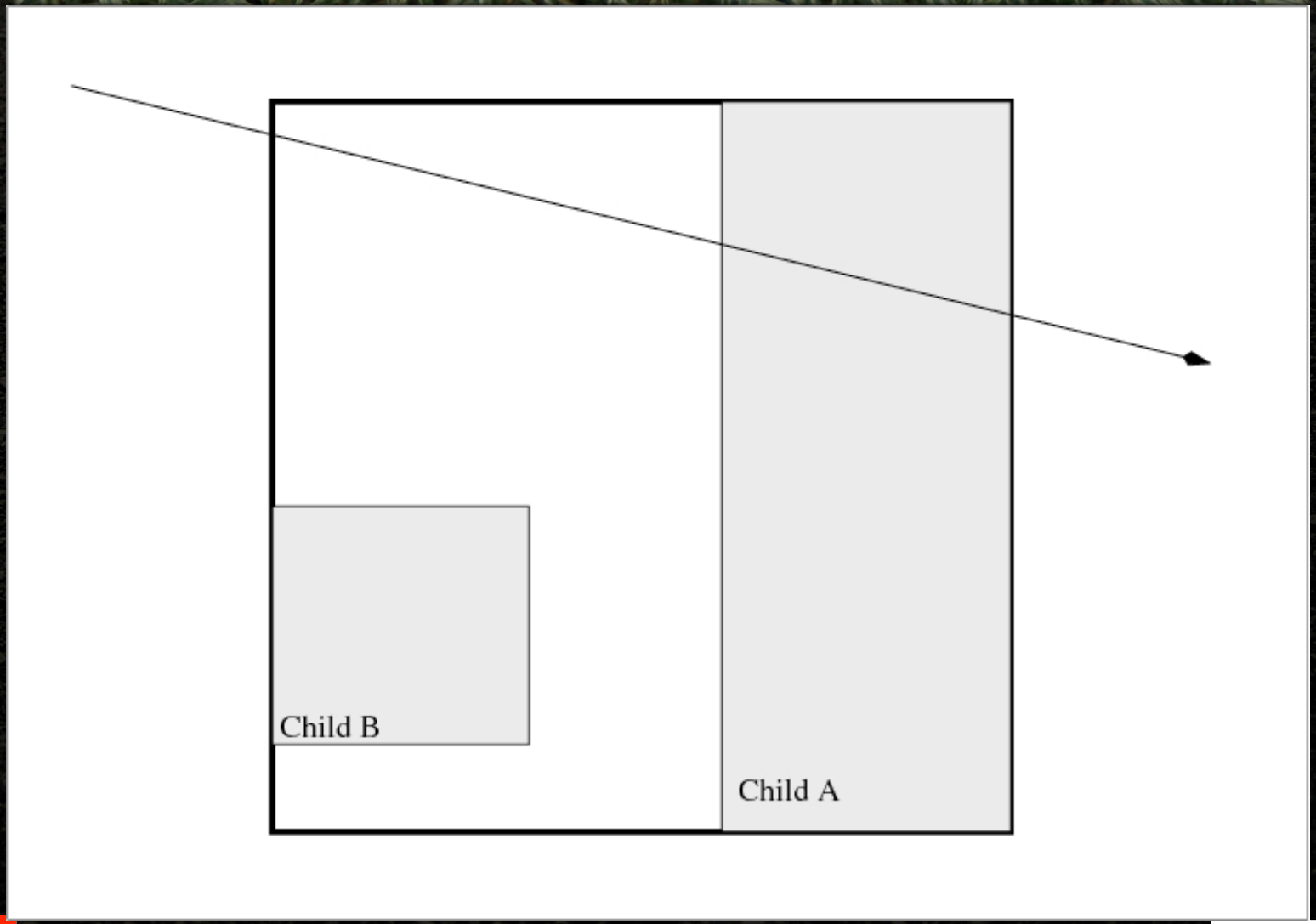


Anti-aliasing review

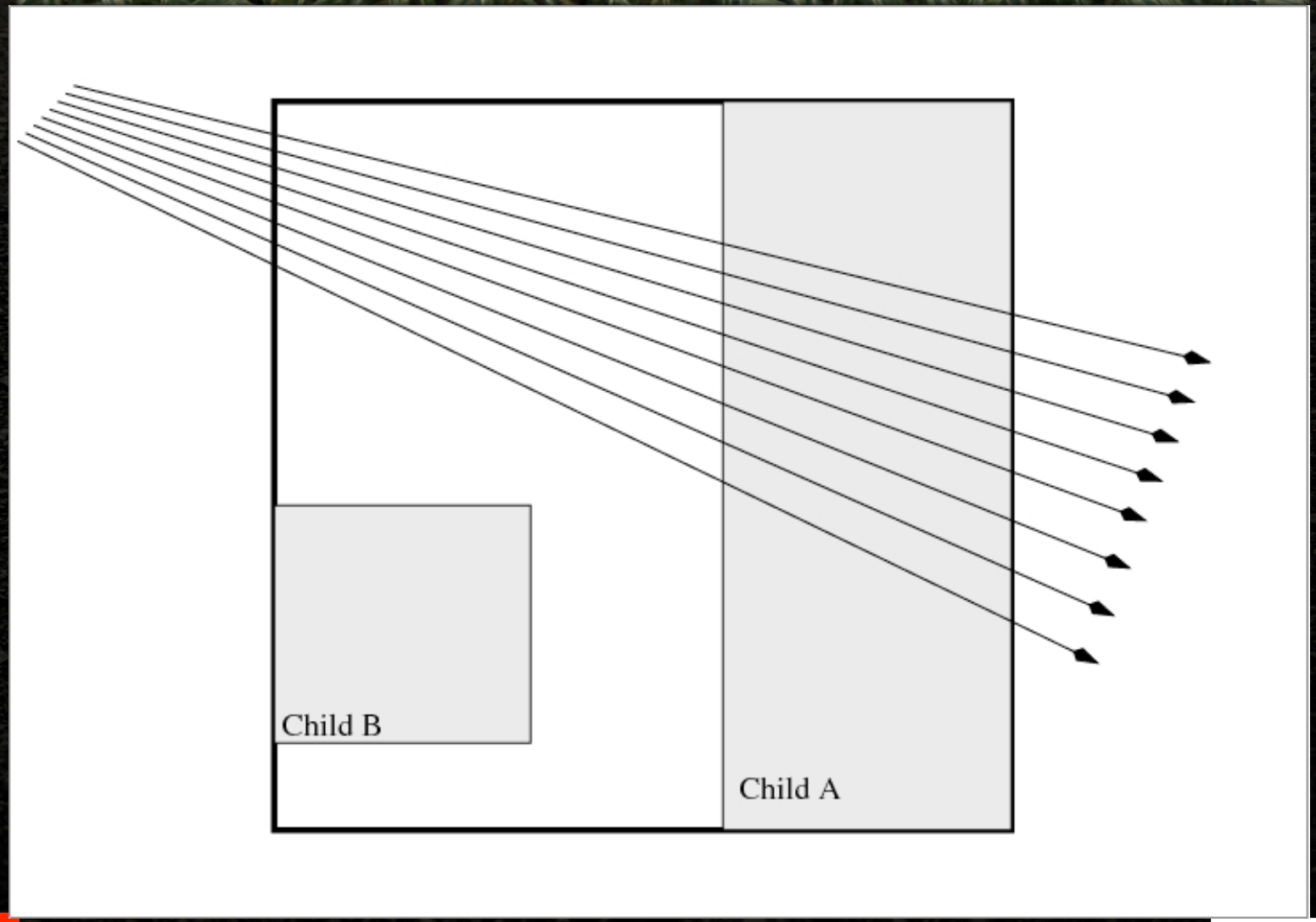
- Nyquist limit requires sampling rate of 2F
- Graphics uses infinitely large frequencies
- Can get reasonably close with good filters
- Simple filters do a poor job, even at low frequencies (just above the Nyquist limit)

Packet Traversal

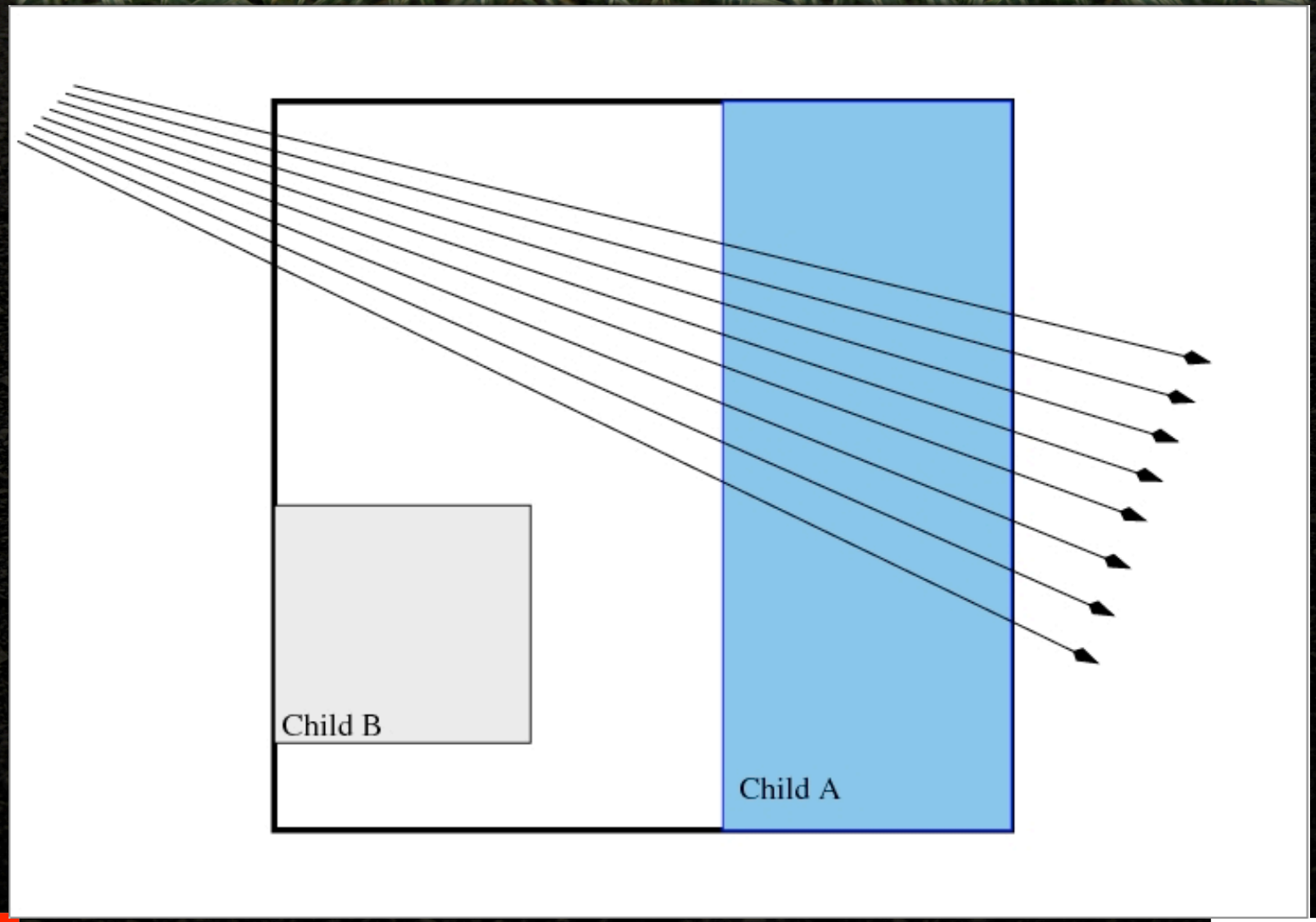
BVH traversal



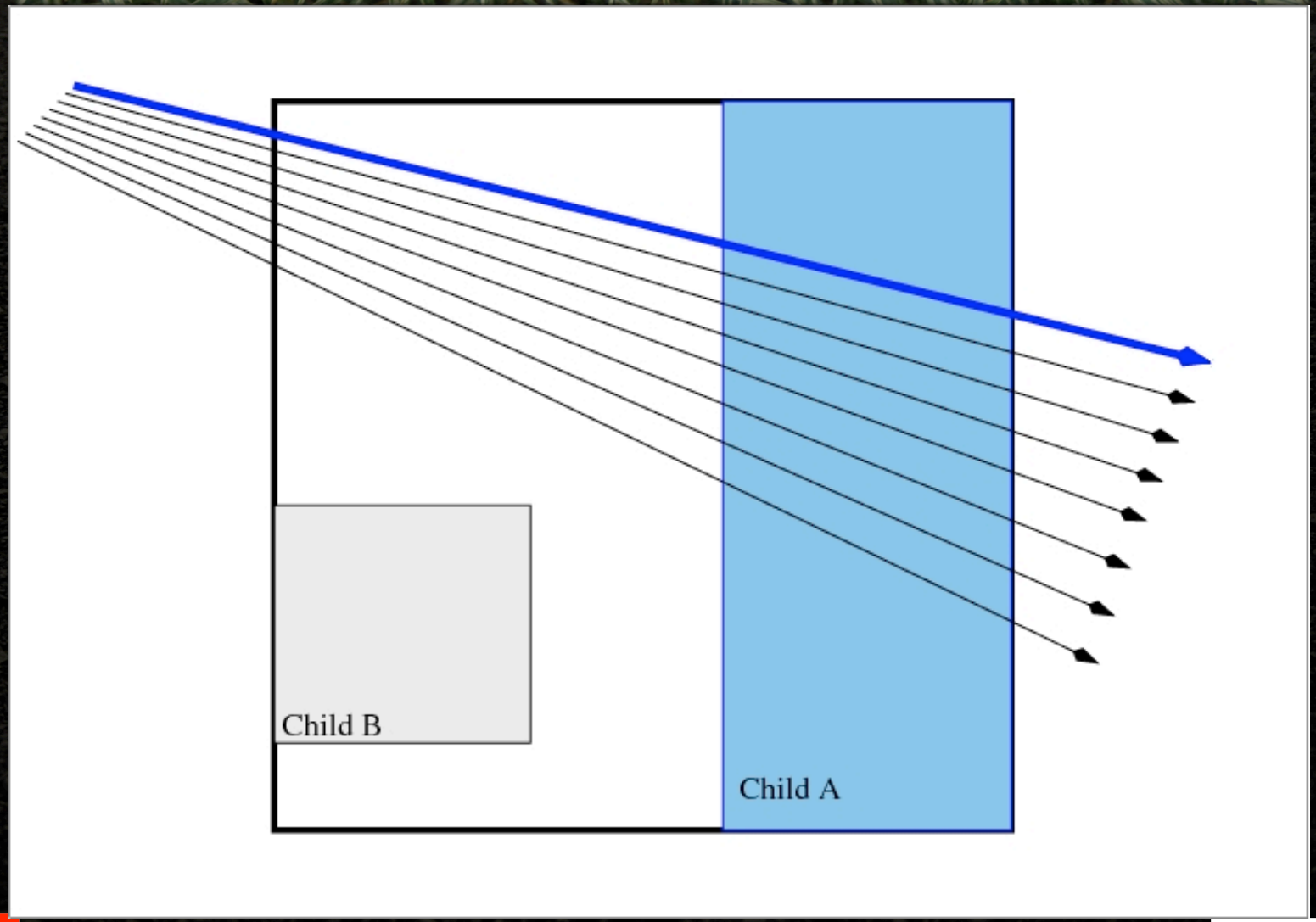
BVH packet traversal



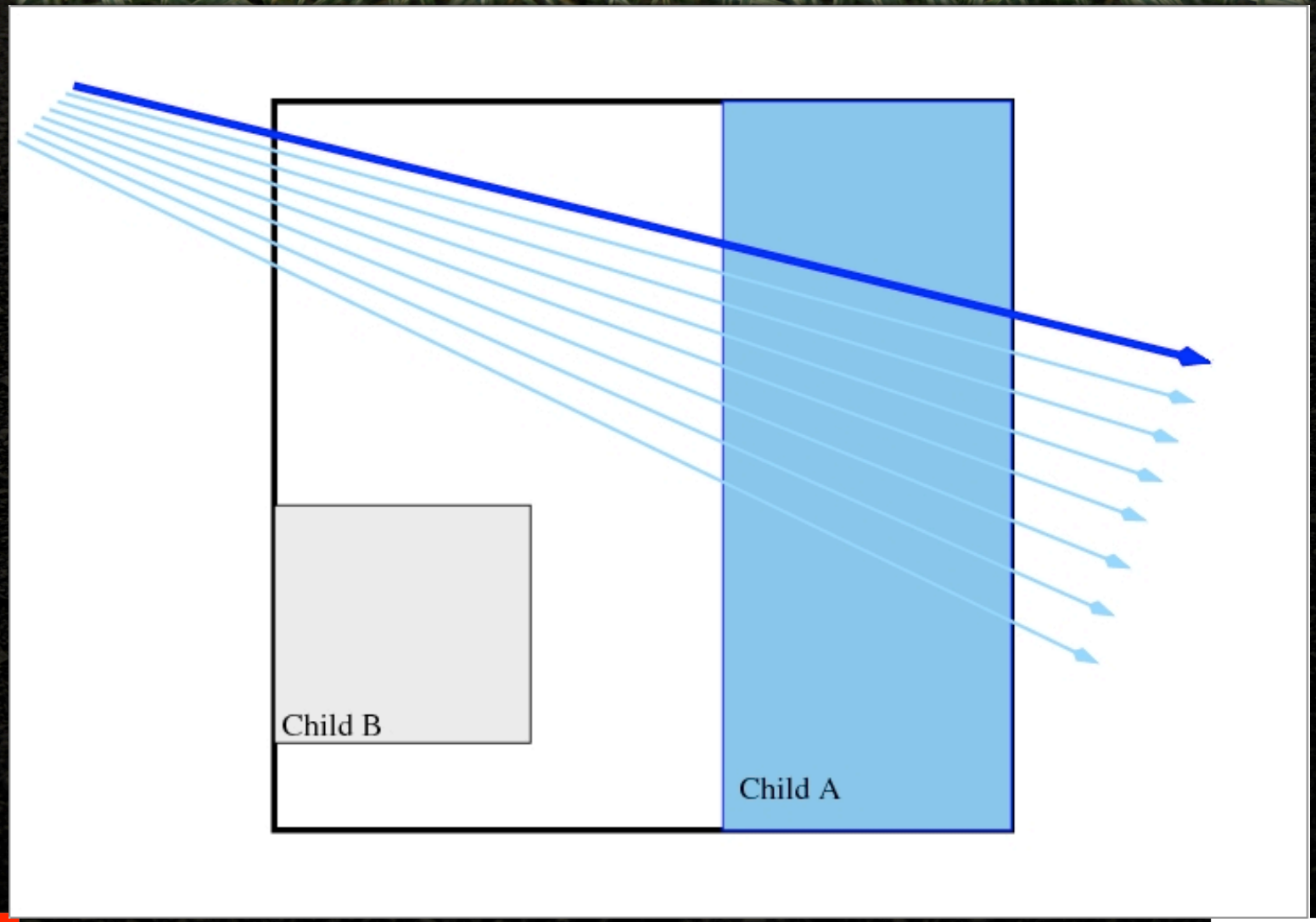
BVH packet traversal



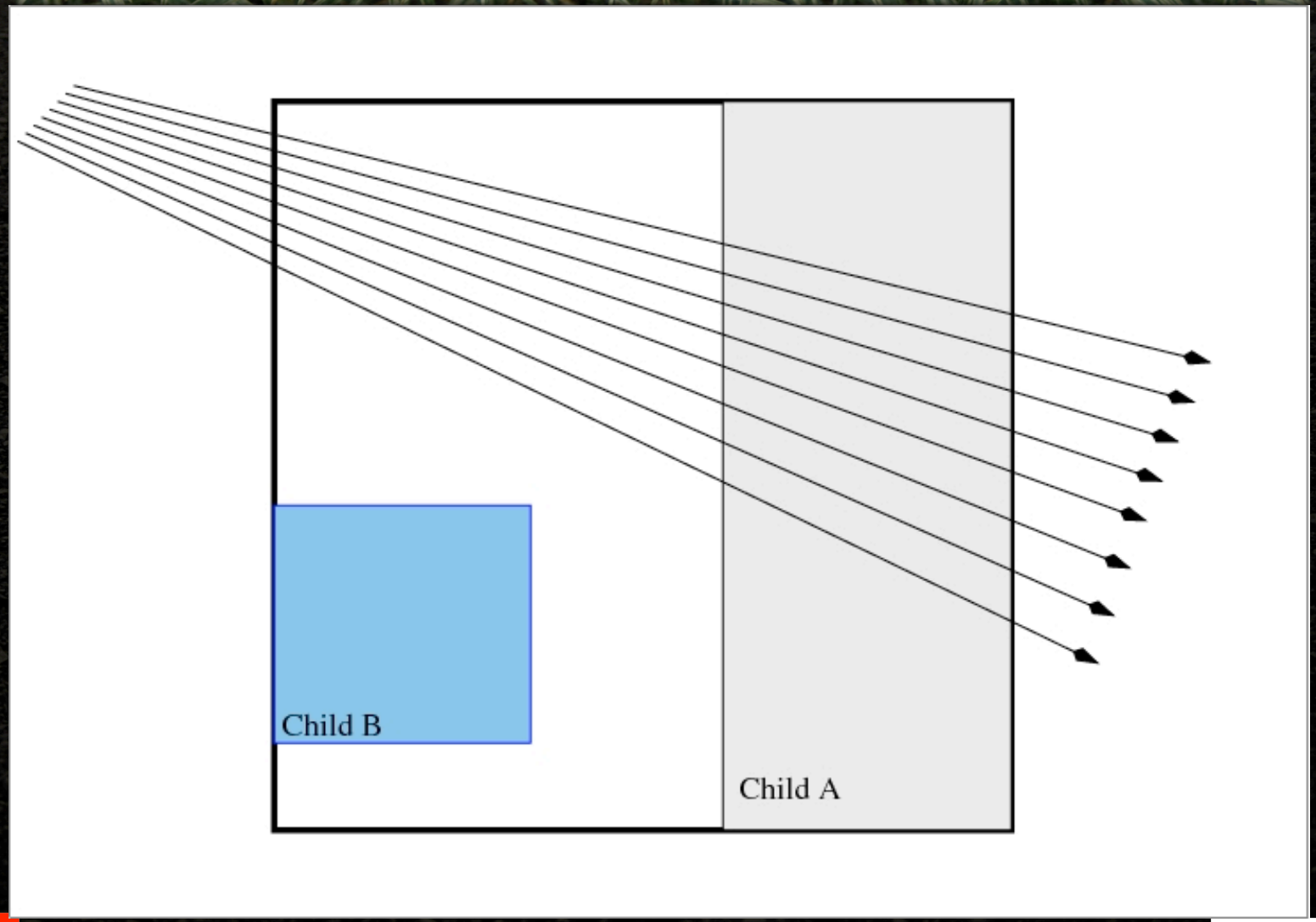
BVH packet traversal



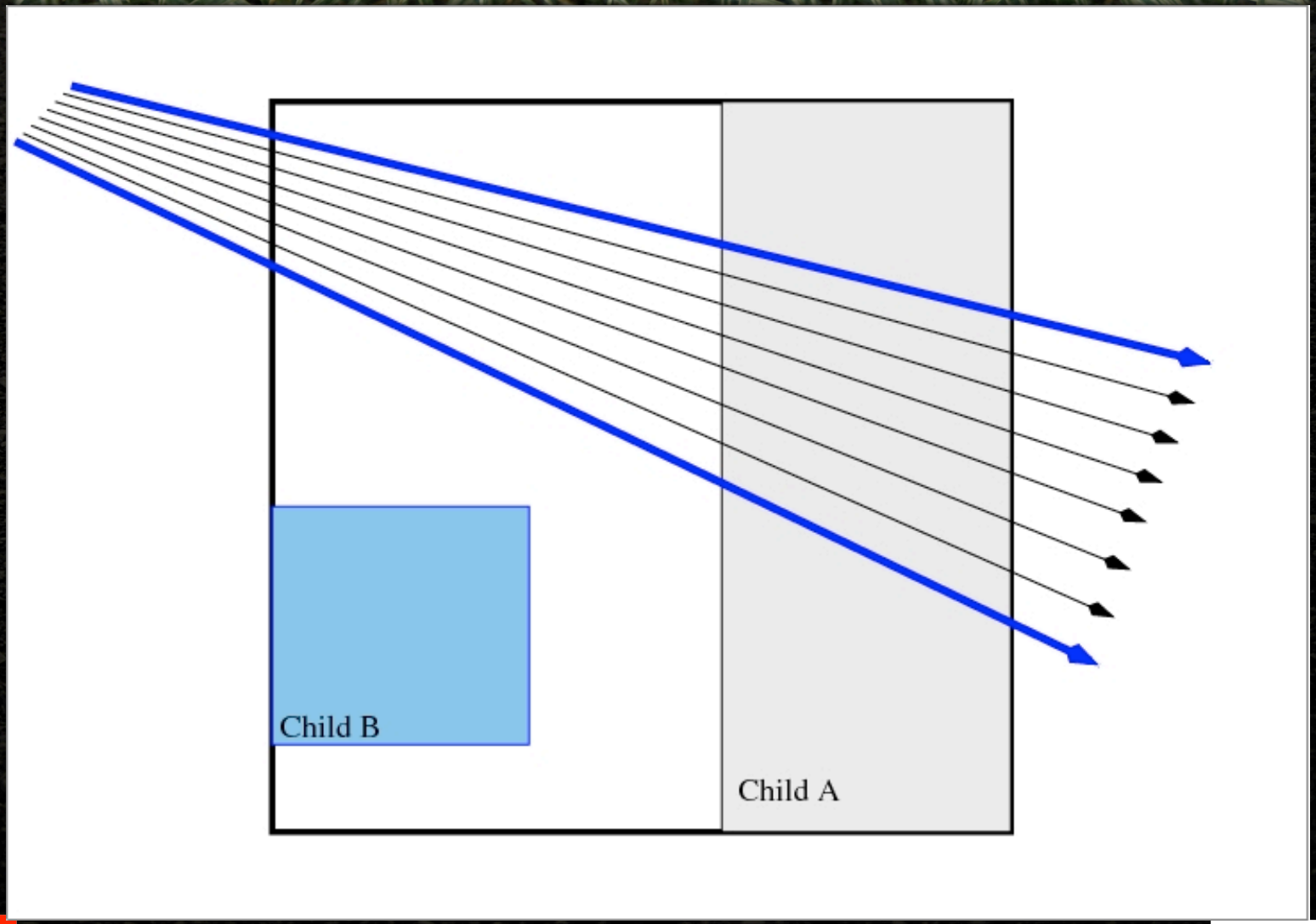
BVH packet traversal



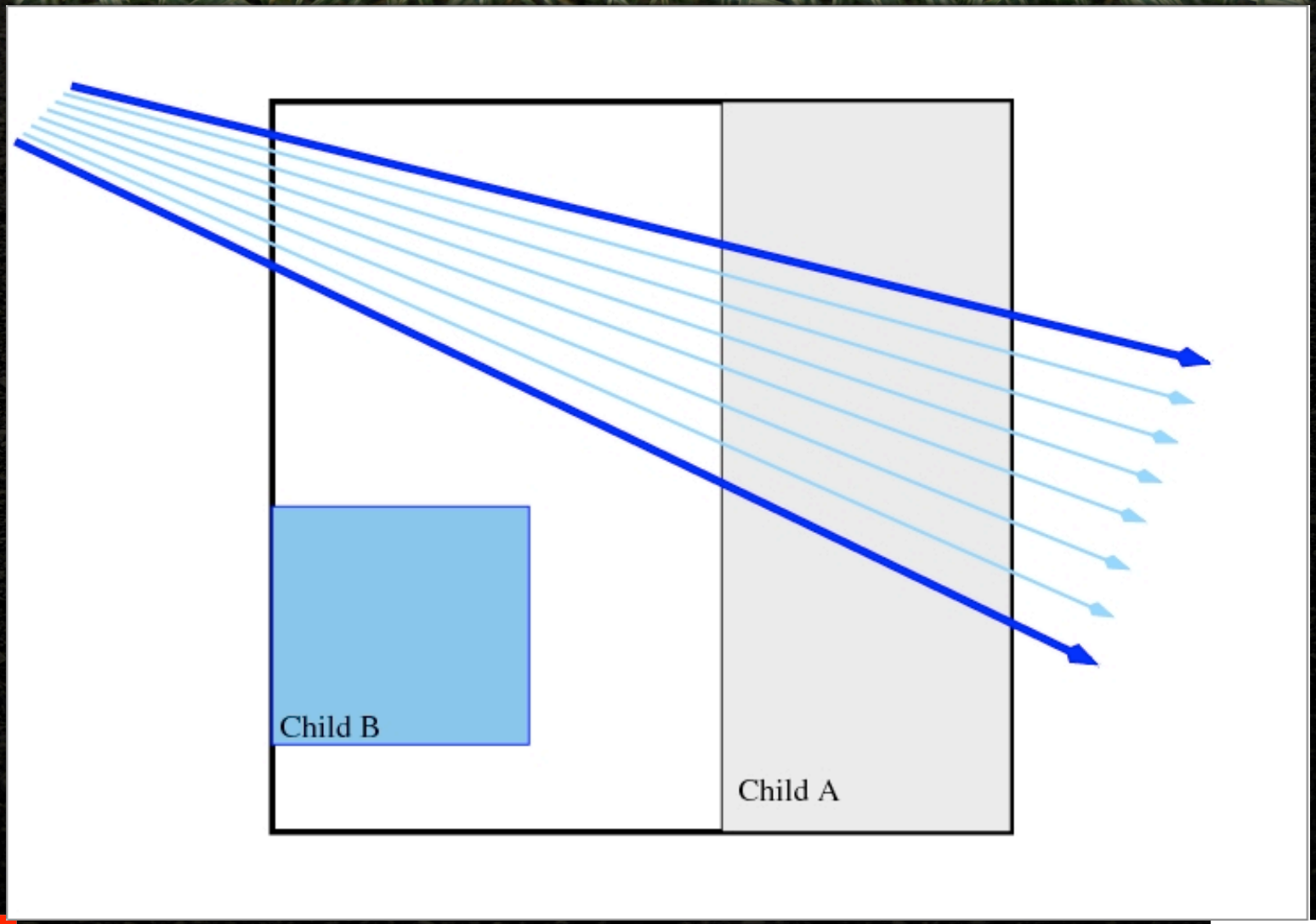
BVH packet traversal



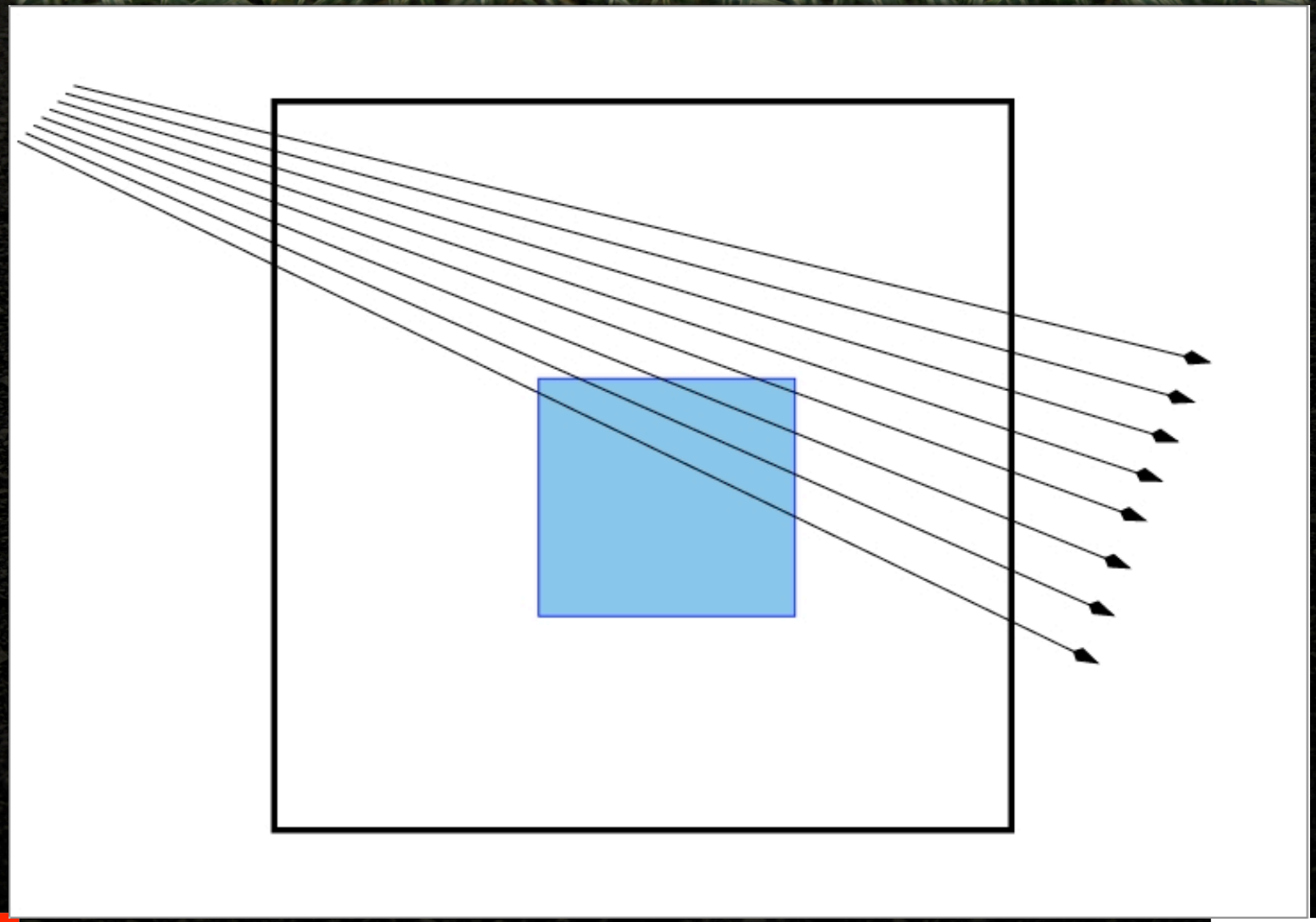
BVH packet traversal



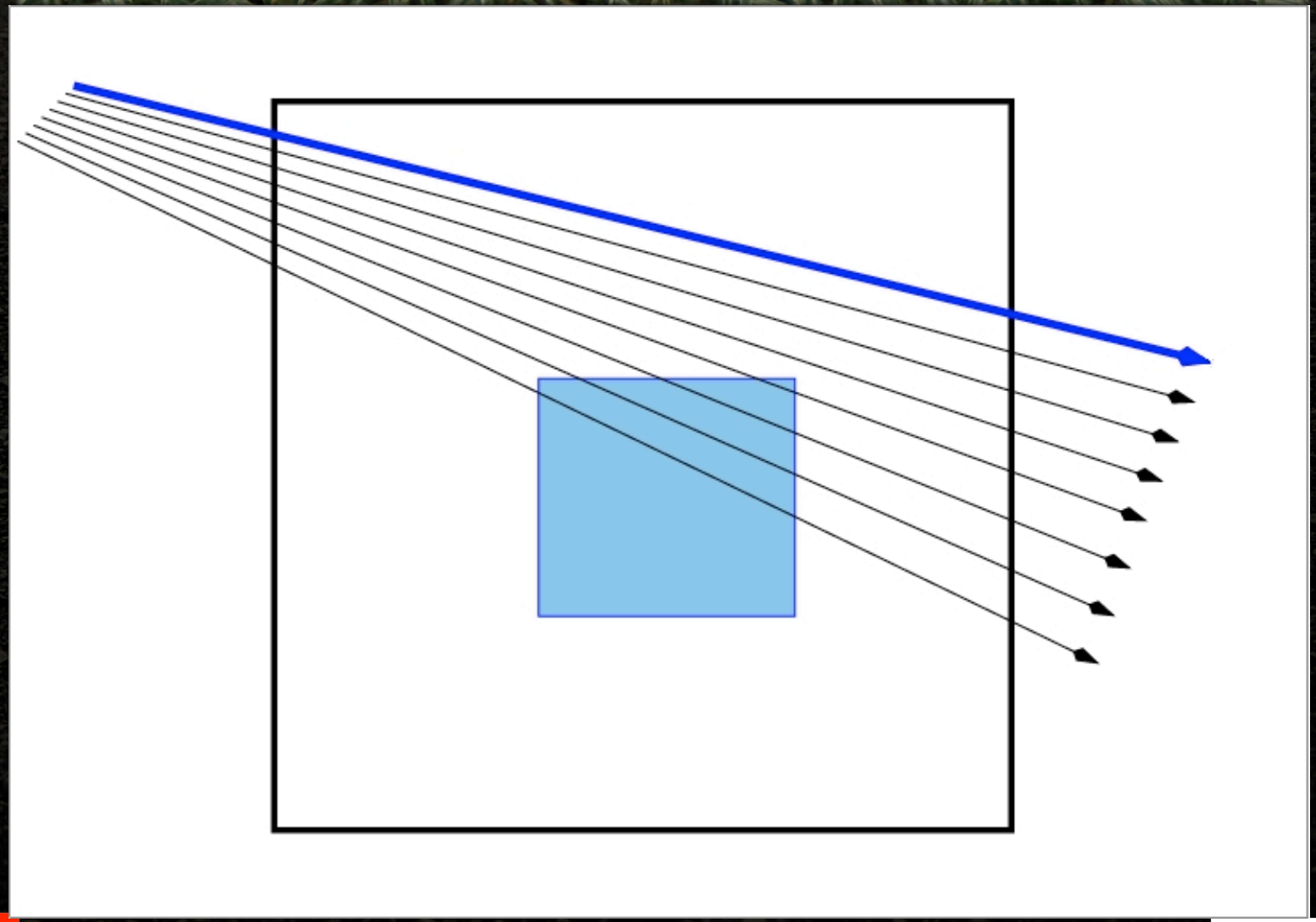
BVH packet traversal



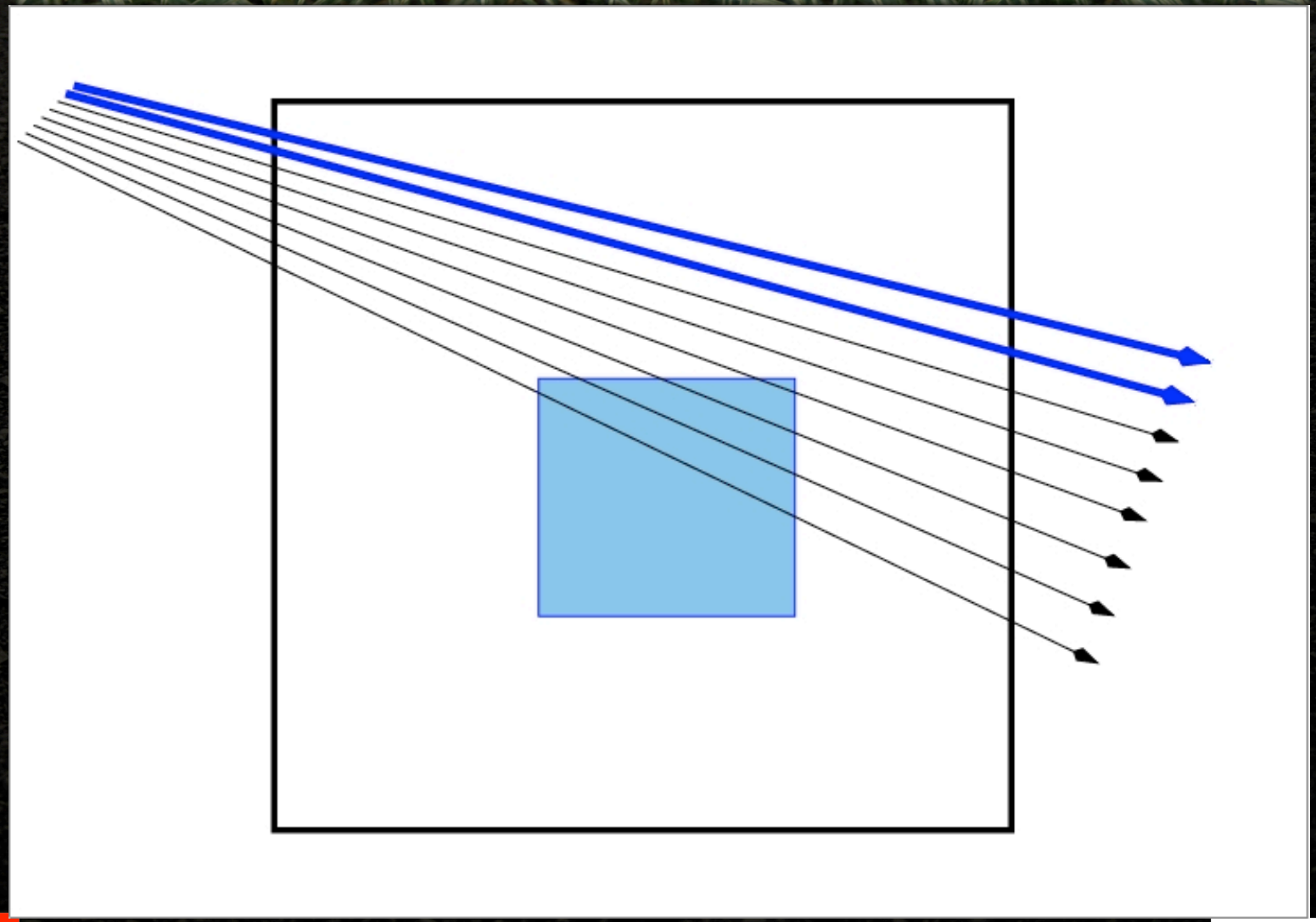
BVH packet traversal



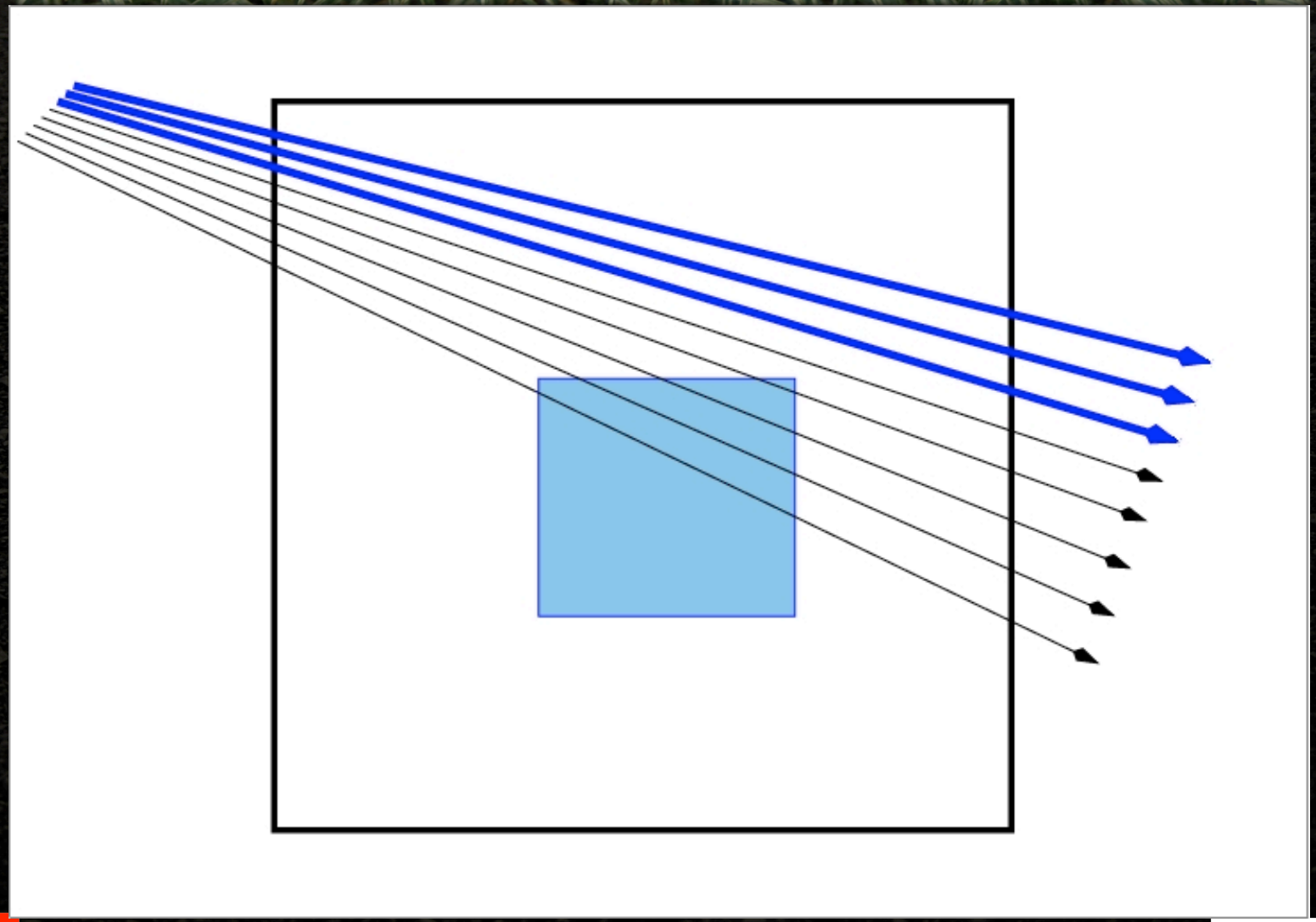
BVH packet traversal



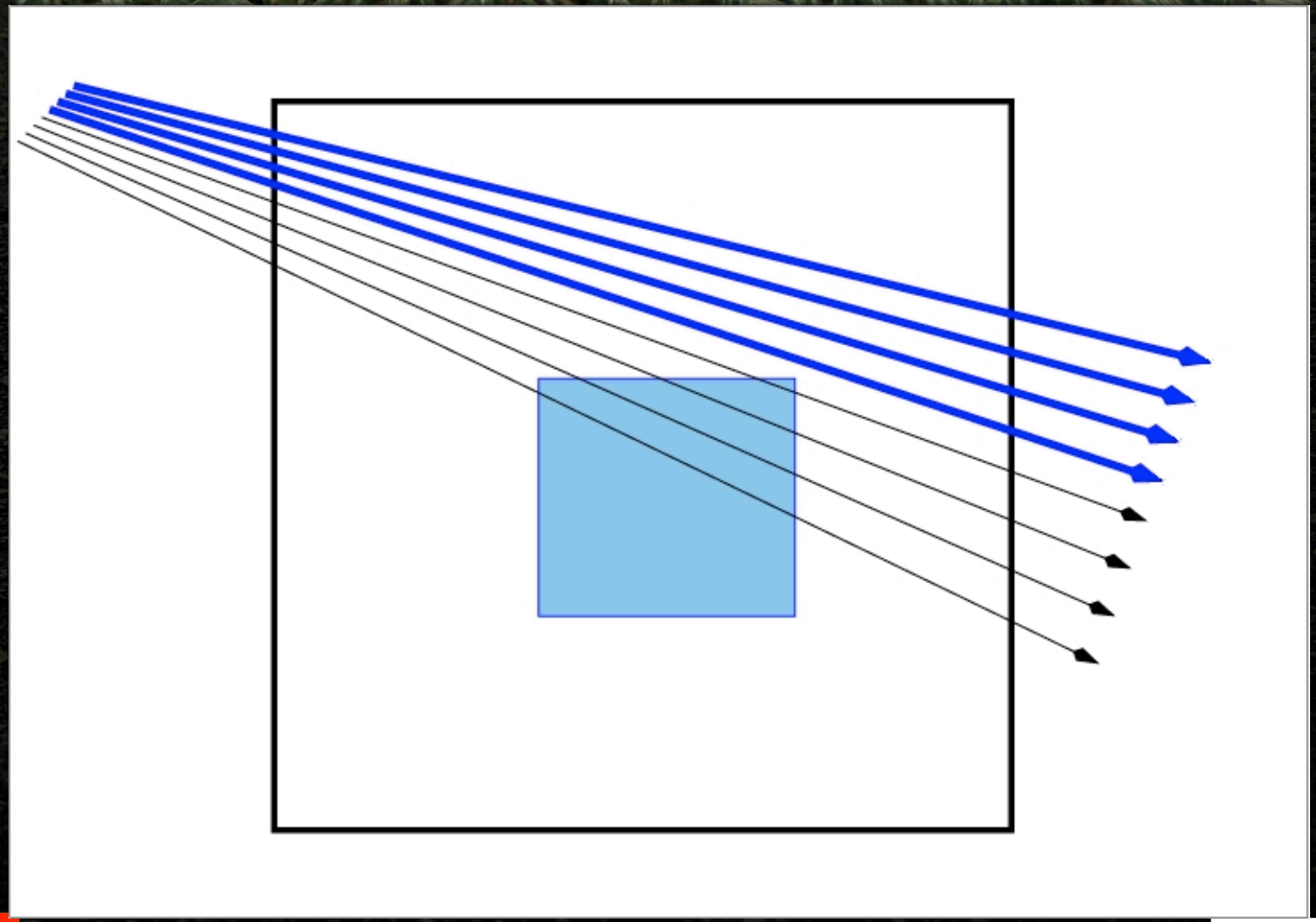
BVH packet traversal



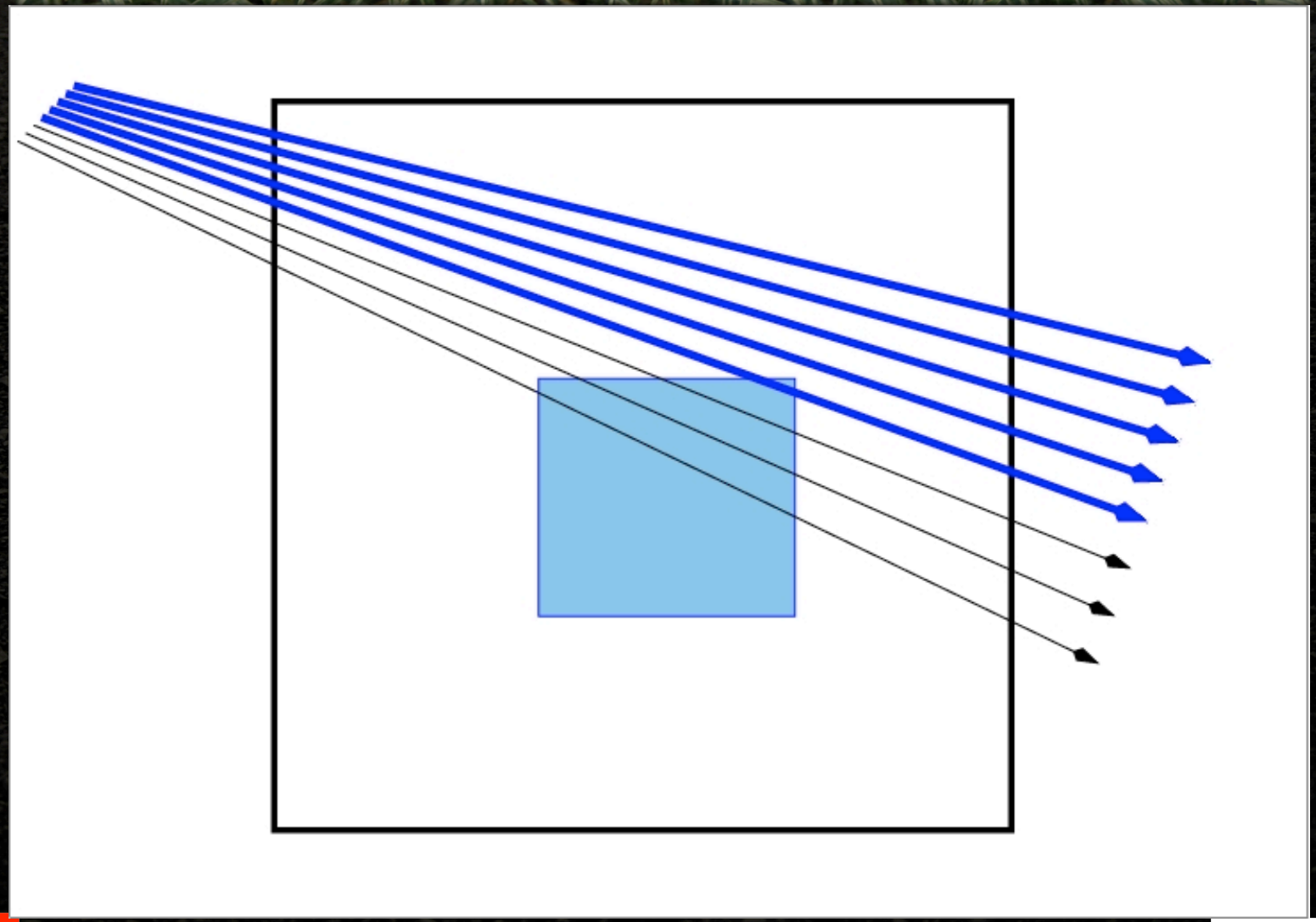
BVH packet traversal



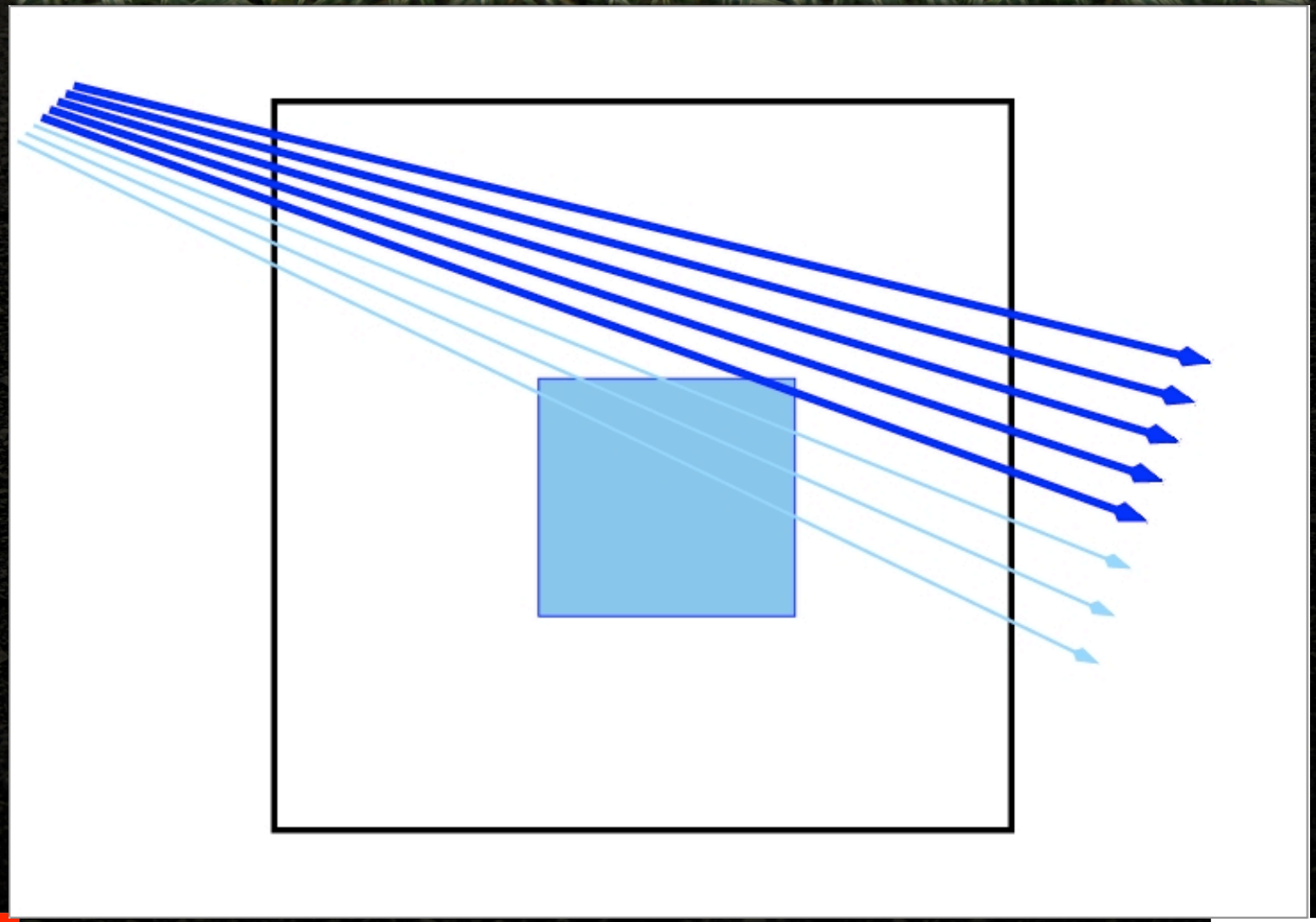
BVH packet traversal



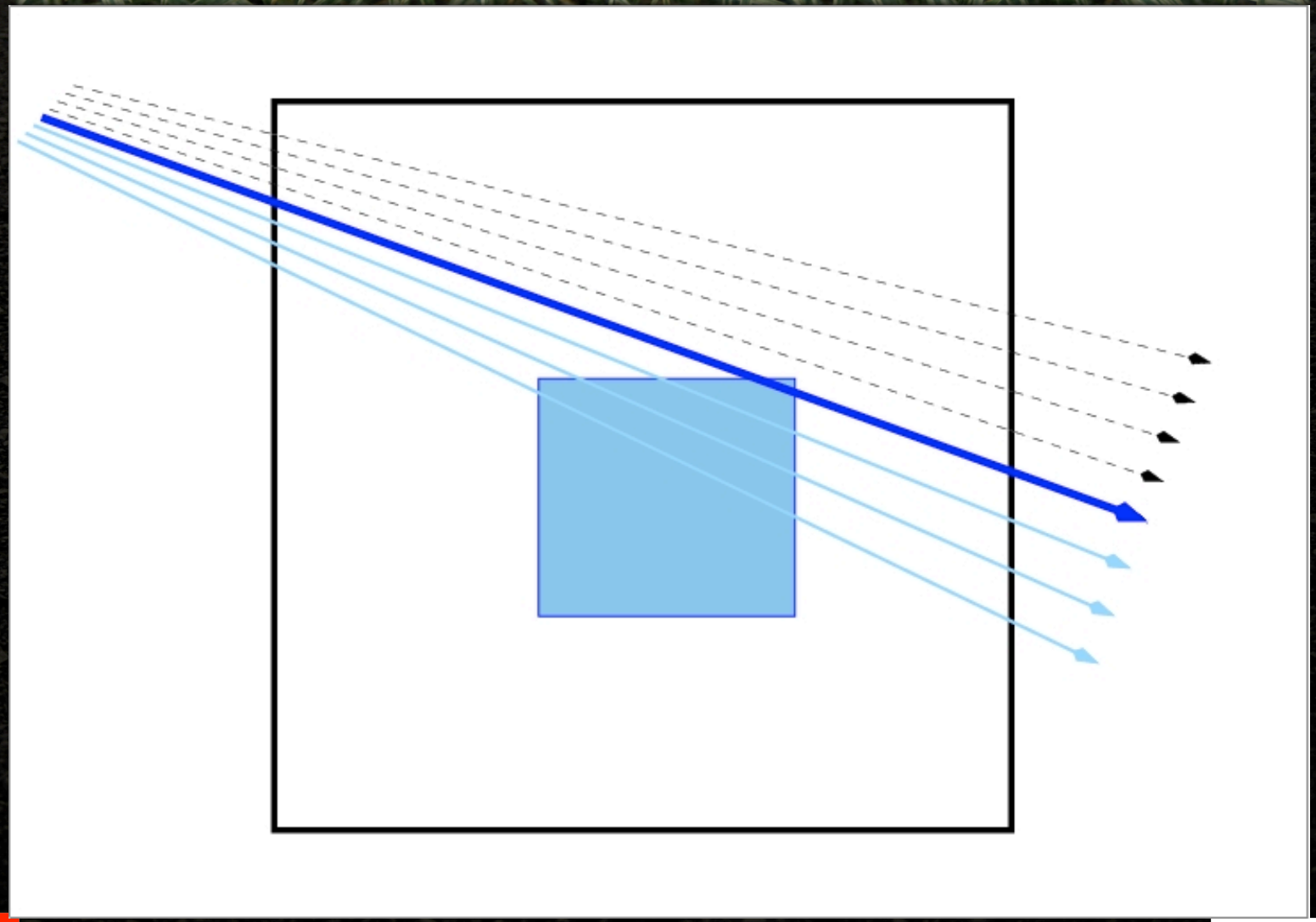
BVH packet traversal



BVH packet traversal



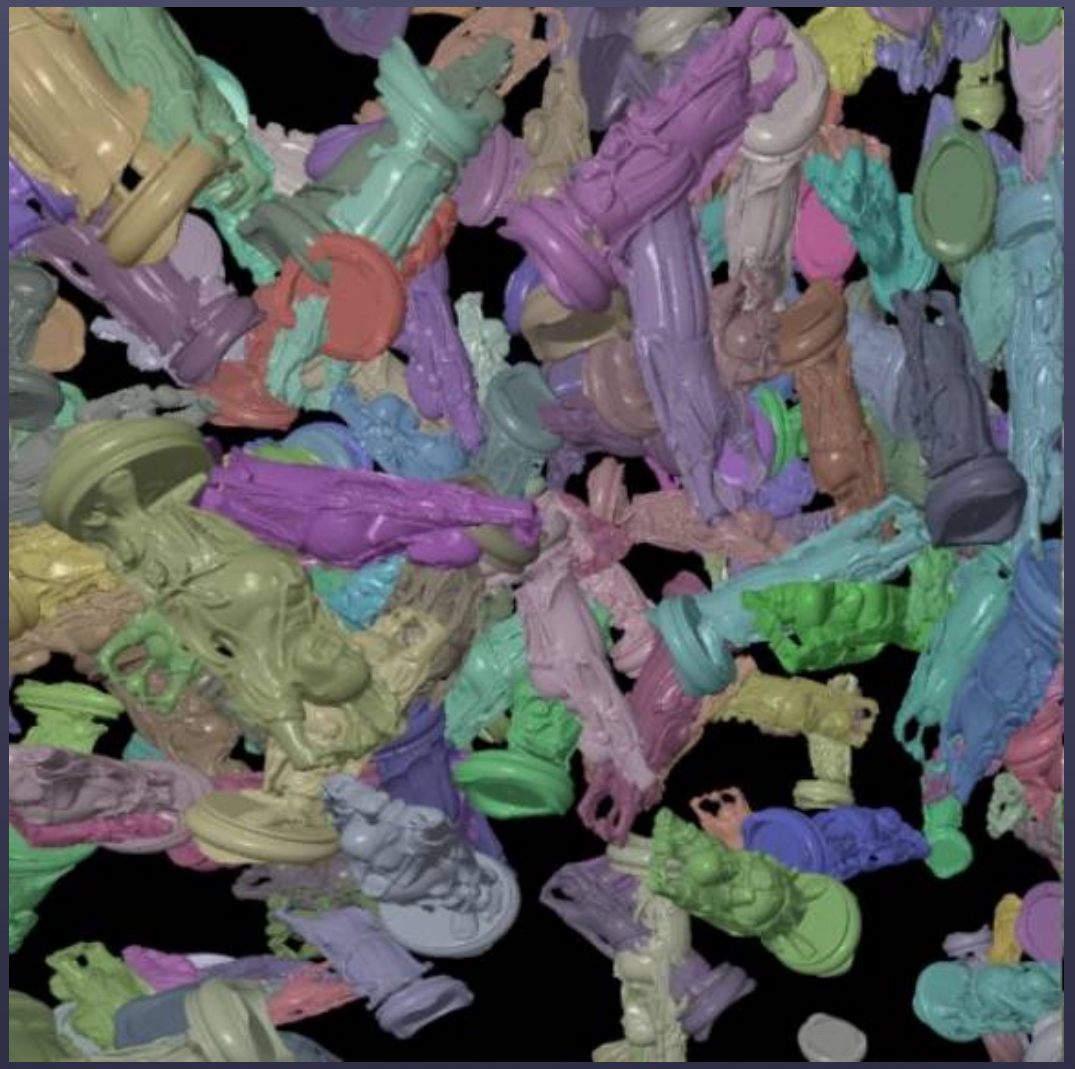
BVH packet traversal



Instancing

Instancing

- Increase geometric complexity through creating multiple copies of similar objects



CS6620 Spring 07

http://www.cs.utah.edu/~angell/cs6620/assignment6/

Instances

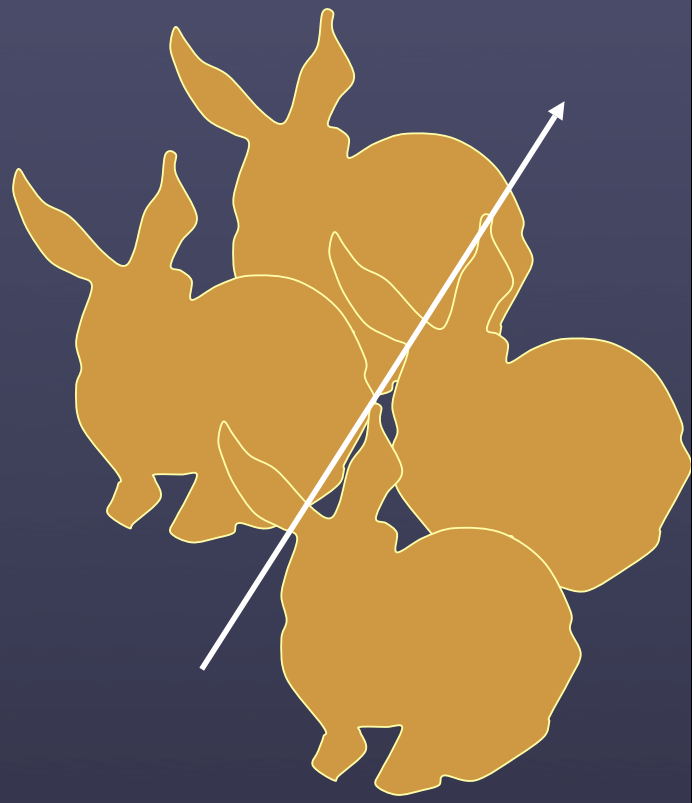
- Instances may affect position and material properties

Pre-transformation

Simplest way: transform objects as they are read in

+No additional overhead in ray tracing objects

- Requires a lot of memory for complex objects

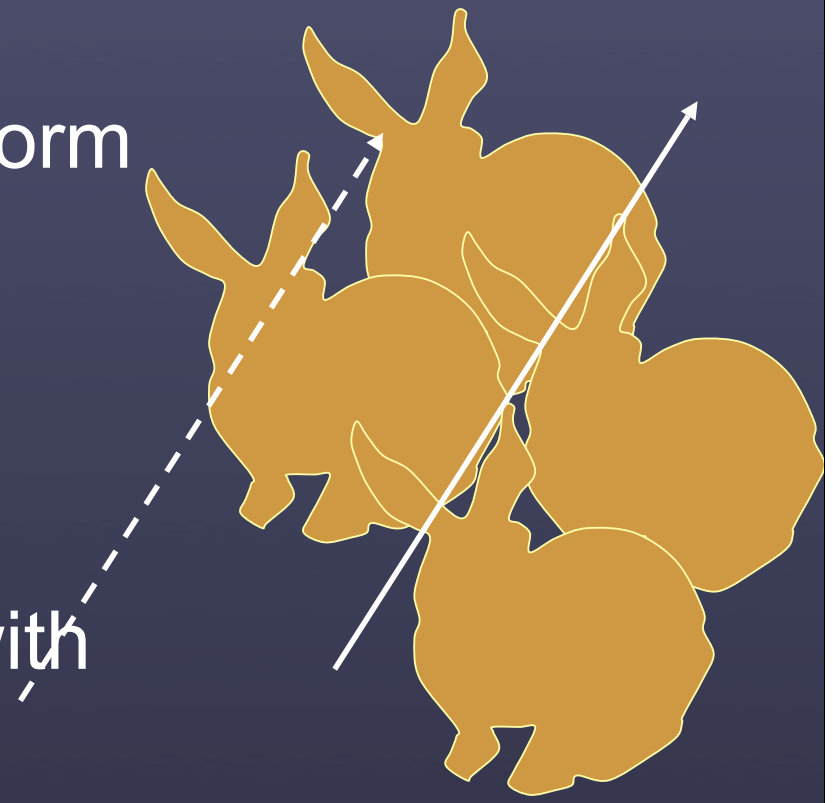


Transform

More efficient way: transform ray at intersection time

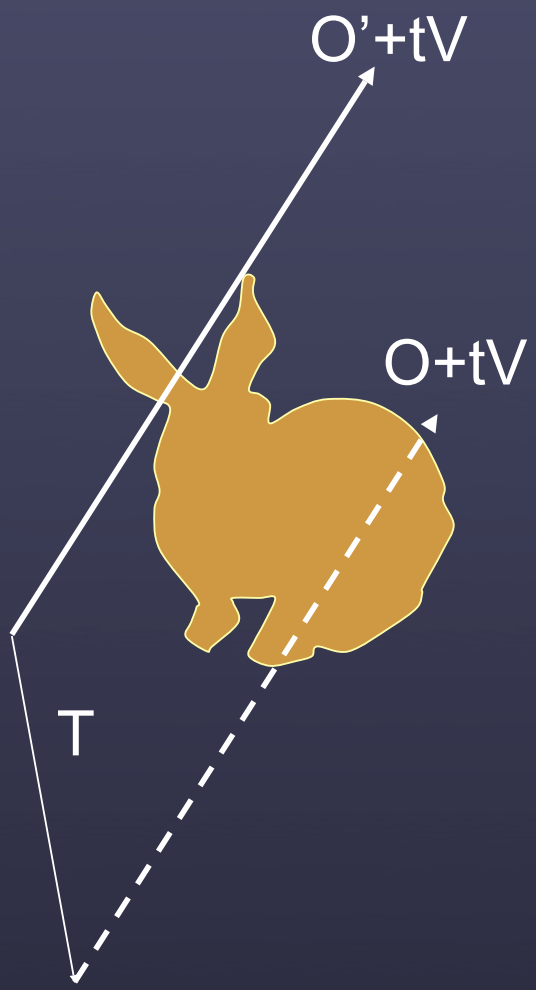
+ Transform must be computed for each ray

- Scenes with billions of polygons are possible with modest memory requirements



Translational instances

- Simplest instance: translation vector
- Create a TInstance class as a subclass of Primitive
- Contains a translation vector and a pointer to an underlying object
- Intersect method subtracts translation vector from origin and calls intersect on underlying object
- Similar for call to normal()

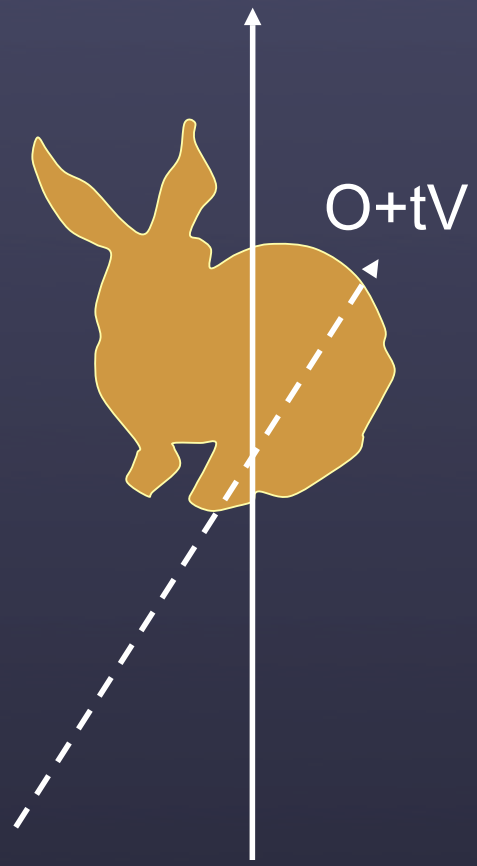


General instances

- General linear transformations are more powerful

$$\vec{P} = \vec{O} + t\vec{V}$$

$$\vec{P'} = \begin{bmatrix} M_{00} & M_{01} & M_{02} & T_x \\ M_{10} & M_{11} & M_{12} & T_y \\ M_{20} & M_{21} & M_{22} & T_z \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$



General transformations

$$\vec{P} = \vec{O} + t\vec{V}$$

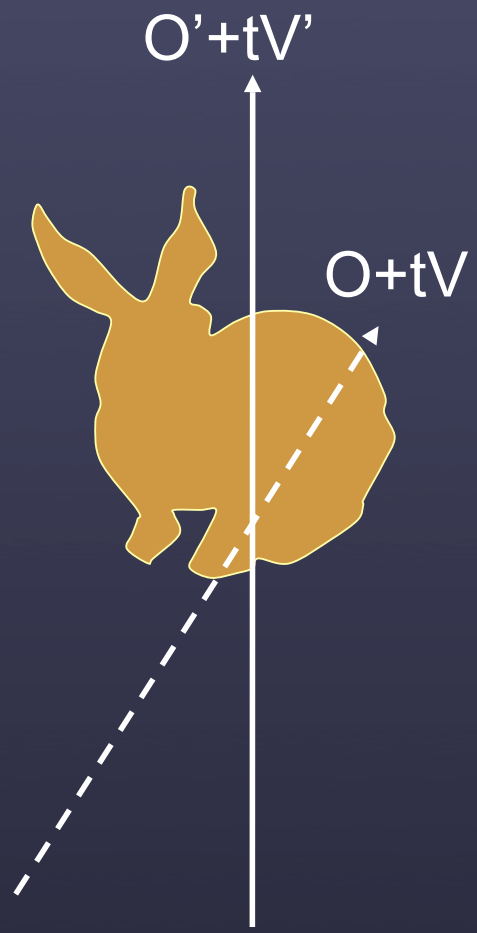
$$\vec{P'} = \begin{bmatrix} M_{00} & M_{01} & M_{02} & T_x \\ M_{10} & M_{11} & M_{12} & T_y \\ M_{20} & M_{21} & M_{22} & T_z \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

$$\begin{aligned} \vec{P'} &= M\vec{P} + \vec{T} \\ &= M\vec{O} + M\vec{V} + \vec{T} \\ &= \vec{O'} + t\vec{V'} \end{aligned}$$

where :

$$\vec{O'} = M\vec{O} + \vec{T}$$

$$\vec{V'} = M\vec{V}$$

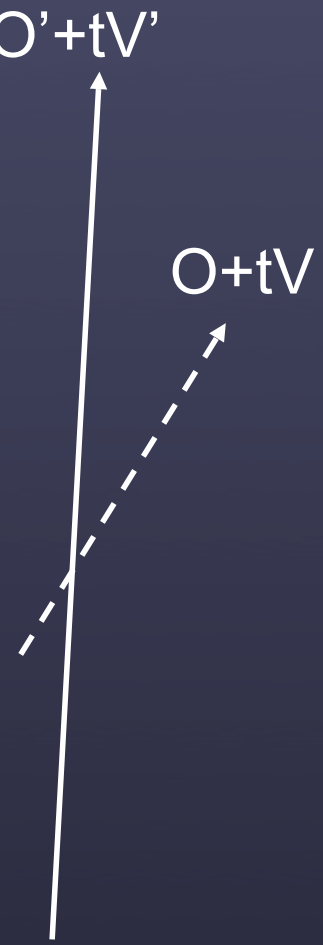


Instances

- Create an Instance class as a subclass of Primitive
- Contains a transform object and a pointer to an underlying object
- Intersect method transforms origin and direction and then calls intersect on underlying object with the transformed ray
- Two gotchas:
  - Unit length ray directions
  - Normals

Unit length ray directions

- Transformation may change length of a unit vector
- If your implementation relies on normalized ray directions, then ray must be “re-normalized”
- This involves a square root, in general
  - Can be optimized for uniform scale factors



$$l = v.normalize()$$

intersect child

$$t_{parent} = t_{child} * l$$

Normal transformation

Consider a plane:

$\overrightarrow{N} \cdot \overrightarrow{P} = 0$

and a transform:

$\overrightarrow{P'} = M\overrightarrow{P}$

Find a plane with normal $\overrightarrow{N'}$ such that $\overrightarrow{N'} \cdot \overrightarrow{P'} = 0$

$\overrightarrow{N} \cdot \overrightarrow{P} = 0$

$NP^T = 0$

$N\left(M^{-1}M\right)P^T = 0$

$\left(NM^{-1}\right)\left(MP^T\right) = 0$

$\left(NM^{-1}\right)P' = 0$

$\overrightarrow{N'} = \left(NM^{-1}\right) = \left(M^{-1}\right)^T N = \left(M^T\right)^{-1} N$

If M is a rotation matrix, $M^{-1} = M^T$, so $\left(M^T\right)^{-1} = M$

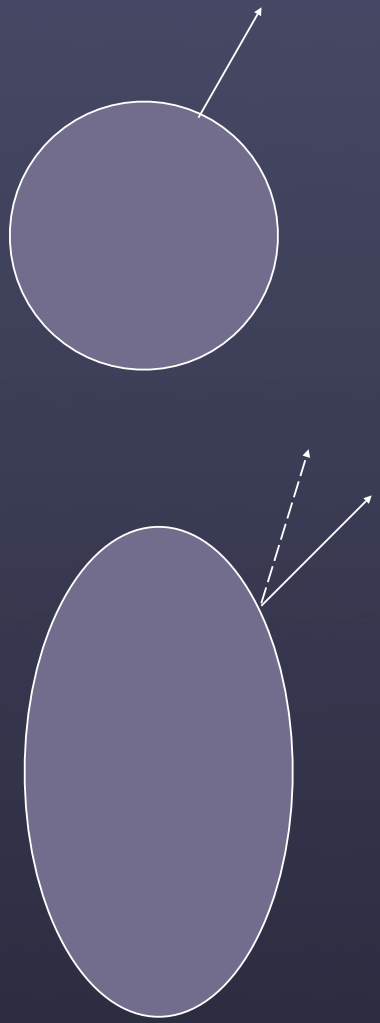
Transformed normals

- Normal must be transformed

$$\overrightarrow{N'} = \left(M^{-1}\right)^T \overrightarrow{N}$$

$$\overrightarrow{N'} = \frac{\overrightarrow{N'}}{\left\|\overrightarrow{N'}\right\|}$$

- Easy with normals computed at intersection time
- Deferred normals (scratchpad) require a little more thought



Transformed deferred normals

- Method 1: Use scratchpad to defer transformation but compute untransformed normal at intersection time

```
intersect:
  HitInfo subhit
  transform ray
  length = normalize ray direction
  intersect child with subhit
  if(subhit.wasHit()
      && hit.hit(t*length, this, subhit.matl)
    Put normal in scratchpad
```

Transformed deferred normals

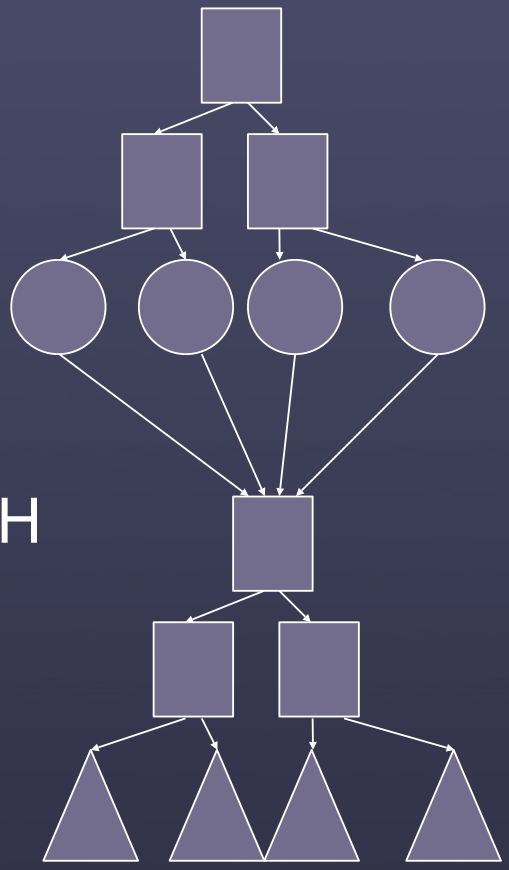
- Method 2: Extend scratchpad to allow a “stack” of objects
- Normal is transformed only if object is actually closest
- Good for nested transformations

Nested transformations

- Multiple transformations can be combined by nesting instance objects
- Must be careful with deferred computation (i.e. scratchpad) mechanisms
  - All of the above solutions work fine, others may not

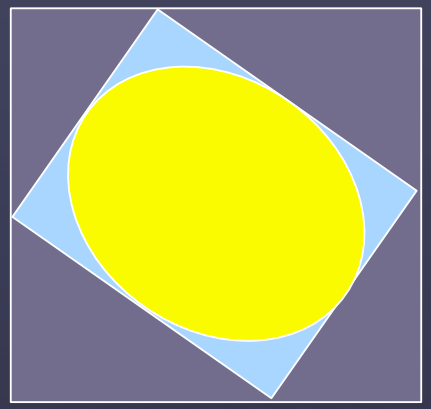
Nested acceleration structures

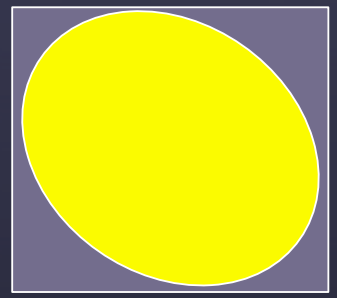
- Using nested structures you can create enormously complex scenes (millions of bunnies)
- Acceleration structures can be mixed/matched (BVH on top, Grid on bottom, etc.)



Bounding box computation

- To use acceleration structures you will need the bounding box of the transformed object
- Simplest method:
  - Transform each vertex of the child bounding box (8 of them)
  - Compute the min/max of each of these vertices
  - Doesn't always result in tight bound
- Complex method:
  - Transform underlying model
  - Easy for polygonal objects
  - Not always possible for general objects





Instance optimizations

- Precompute matrix inverse
  - Perhaps store only inverse
- Specialized methods and/or storage for special cases:
  - Pure translation
  - Pure rotation
  - Rotation + translation
  - Uniform scale
  - Uniform scale plus translation, rotation, or both
  - General case

Changing materials

intersect:

```
  HitInfo subhit
  transform ray
  length = normalize ray direction
  intersect child with subhit
  if(subhit.wasHit()
      && hit.hit(t*length, this, overide_matl)
    Put normal in scratchpad
```

Similar for other mechanisms

Other instancing tricks

- Implicitly defined fractals
- Uses a graph structure instead of an instance tree

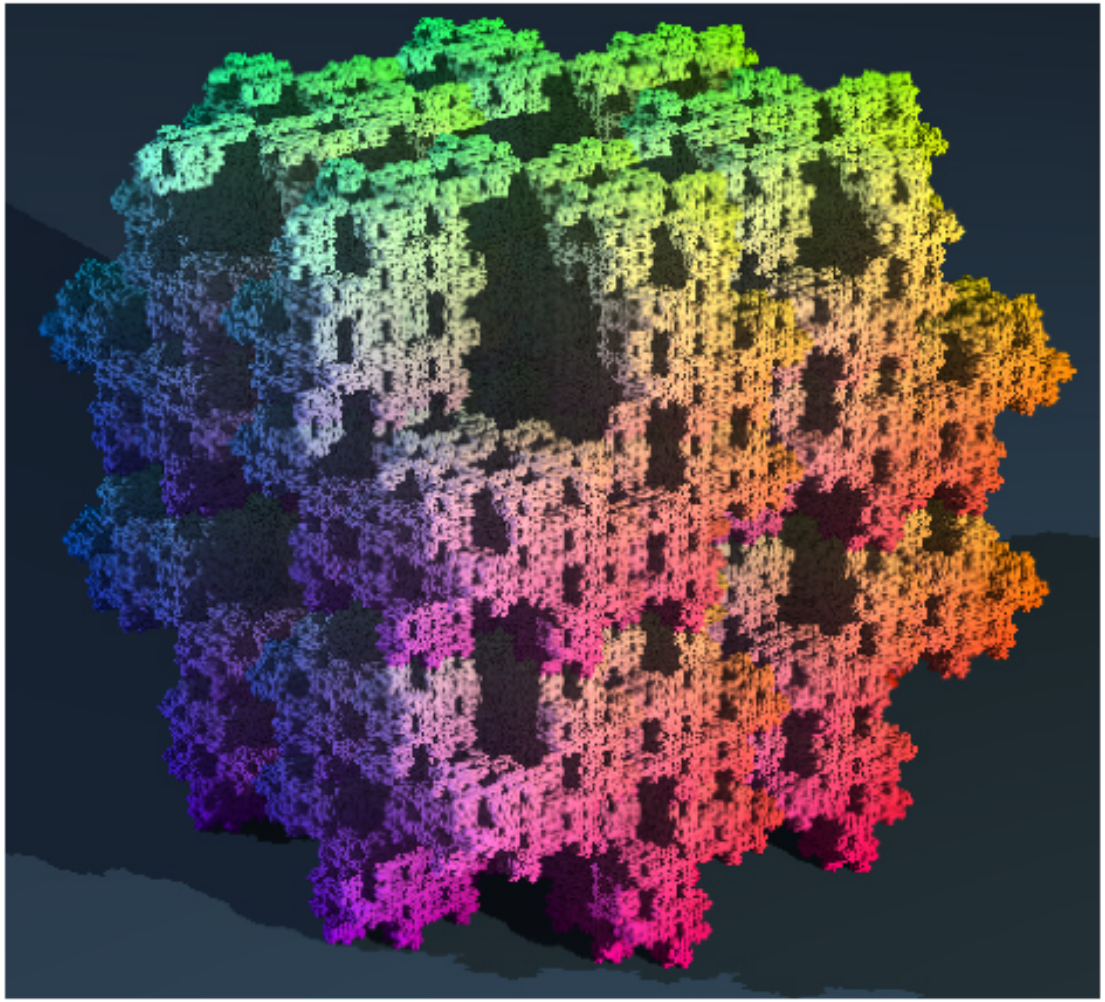


Figure 5: The 3-D fractal pound sign.

graphics.cs.uiuc.edu/~jch/papers/paradigm.pdf

Hart instancing paper

- Trees and other complex objects can be specified with a few bytes

Figure 6: Autumn --- an L-system tree with IFS leaves on instanced grass.