Domain Transforms, Warping & Morphing

Fredo Durand
MIT EECS
6.8370/6.8371

Domain operations

Domain transform

- Apply a function *f* from R^2 to R^2 to the image domain
- if *(x, y)* had color *c* in the input,
  f(x, y) has color *c* in the output

Transformation

- **Simple parametric transformations**
  - linear, affine, perspective, etc



translatio
n



rotation



aspect



affine



perspective



cylindrical

illustration by Rick Szeliski

Warping

- Imagine your image is made of rubber
- warp the rubber



No prairie dogs were armed when creating this image

Application of warping: weight loss

- **Liquify in photoshop**



© Eric Kuaimoku

figure 9.35



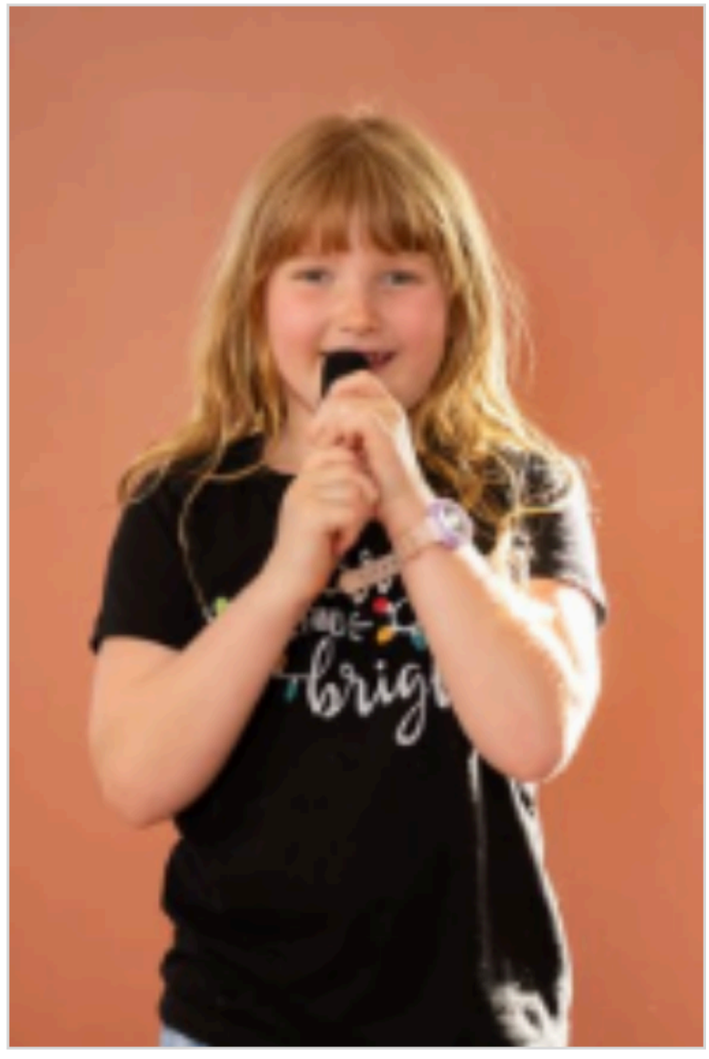
figure 9.36

Demo: Liquify demo

original



liquify: drag to warp



Push | Bloat | Pucker | Twirl | Reset



figure 9.37

Selecting the entire left side of the image avoids potential artifacts.



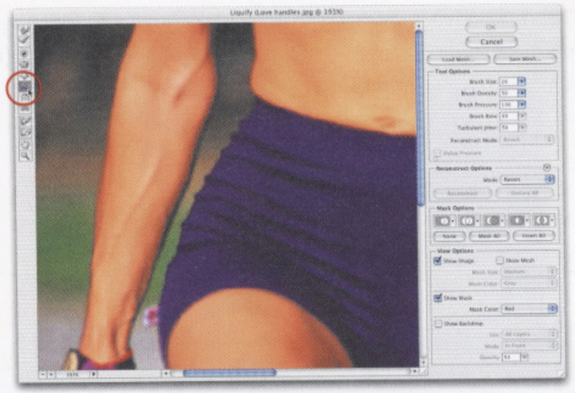
figure 9.38

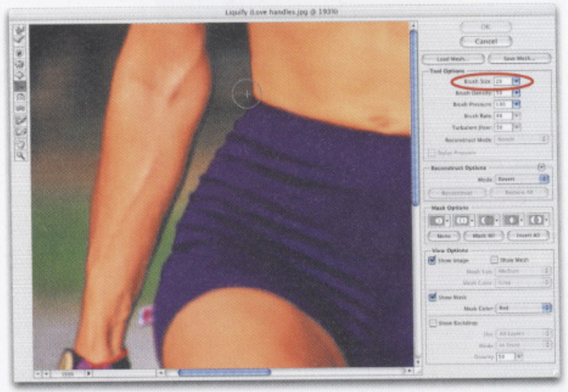
Dragging a Free Transform handle to narrow the selected area.



figure 9.39

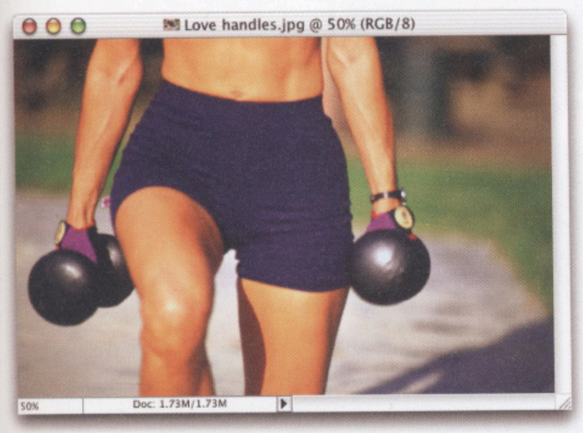
The Liquify filter's Warp tool pushes pixels forward as you drag.



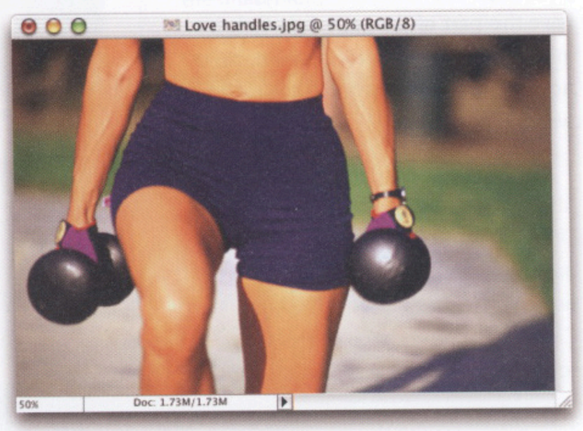


Step Three:
Get the Push Left tool from the Toolbar (as shown here). It was called the Shift Pixels tool in Photoshop 6 and 7, but Adobe realized that you were getting used to the name, so they changed it, just to keep you off balance.

Step Four:
Choose a relatively small brush size (like the one shown here) using the Brush Size field near the top-right of the Liquify dialog. With it, paint a downward stroke starting just above and outside the love handle and continuing downward. The pixels shifts back in toward the body, removing the love handle as you paint. (Note: If you need to remove love handles on the left side of the body, paint upward rather than downward. Why? That's just the way it works.) When you click OK, the love handle repair is complete.



Before.



After.

Domain transform issues

- Apply a function *f* from R2 to R2 to the image domain
- looks easy enough

- But 2.5 big issues:
  - which direction do we transform
  - how do we deal with non-integer coordinates?

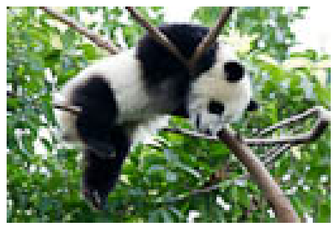
  - And for warping: how do we specify *f*?

Questions?



Basic resampling

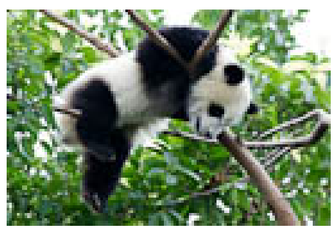
Naive scaling

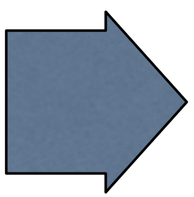


Naive scaling

- Take all the pixels in input and transform them to their output location
  - im(x, y) => out(k*x, k*y)

```
scaleBAD(im, k):
    out = Image(im.width()*k, im.height()*k)
    for each input pixel (x, y):
        out(k*x, k*y) = im(x, y)
return out
```

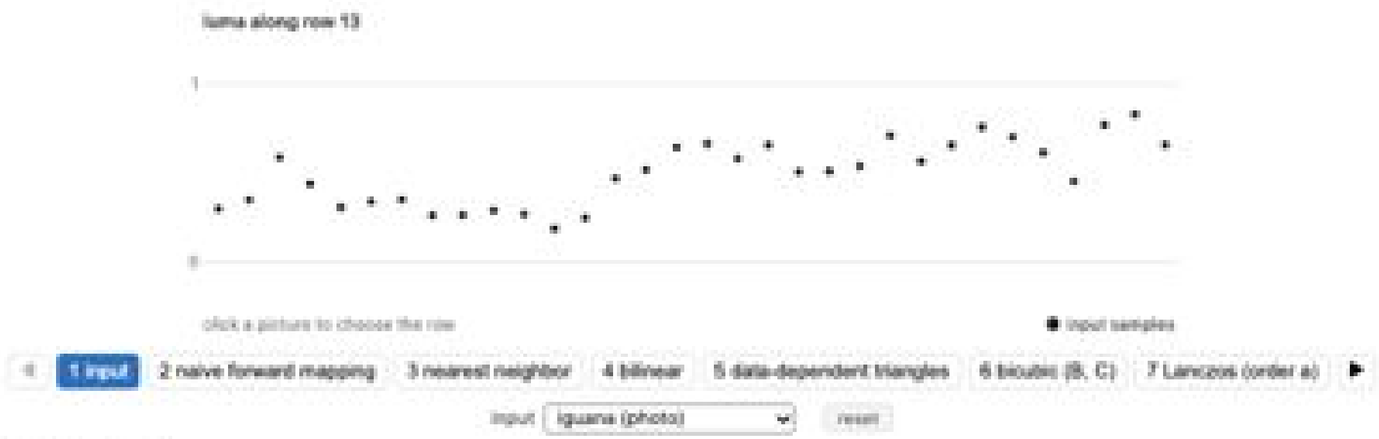






Demo: Figure 6.5.1, stage 1: input





Demo courtesy of Frédo Durand, MIT.

Demo: Figure 6.5.1, stage 3: nearest neighbor

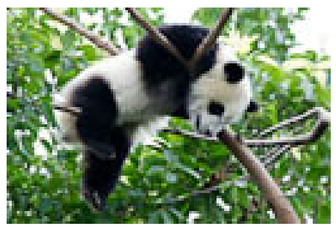


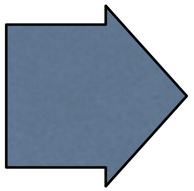
Demo courtesy of Frédo Durand, MIT.

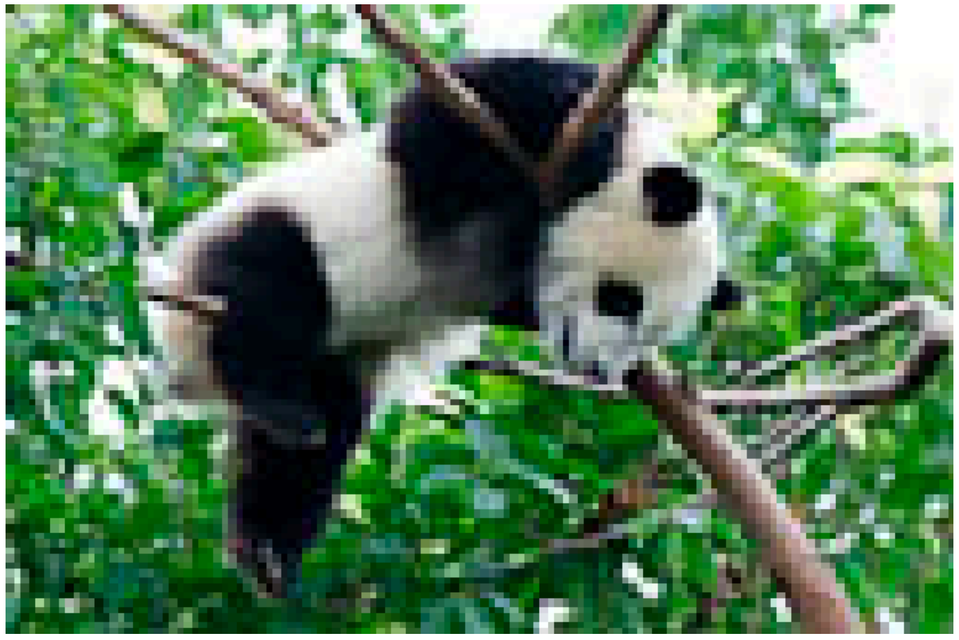
Use the inverse transform!!!!!

- **Main loop on output pixels**
  - out(x, y) <= im(x/k, y/k)

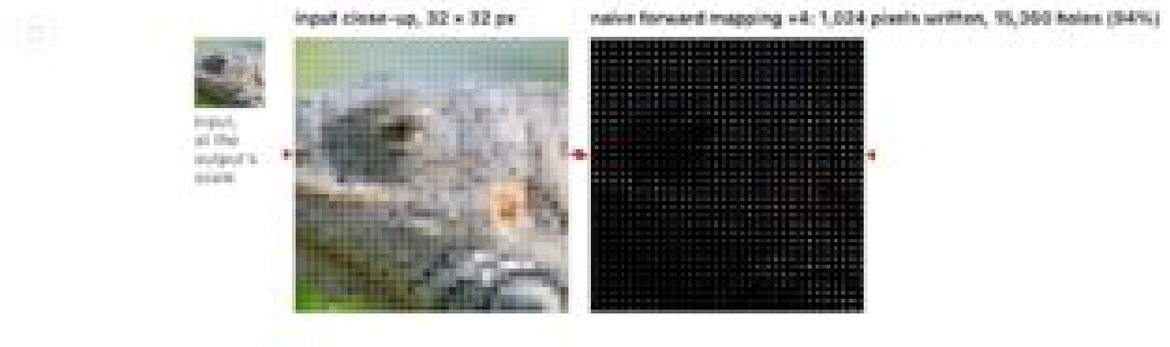
```
scaleWithInverse(im, k):
    out = Image(im.width()*k, im.height()*k)
    for each output pixel (x, y):
        out(x, y) = im(x/k, y/k)
    return out
```

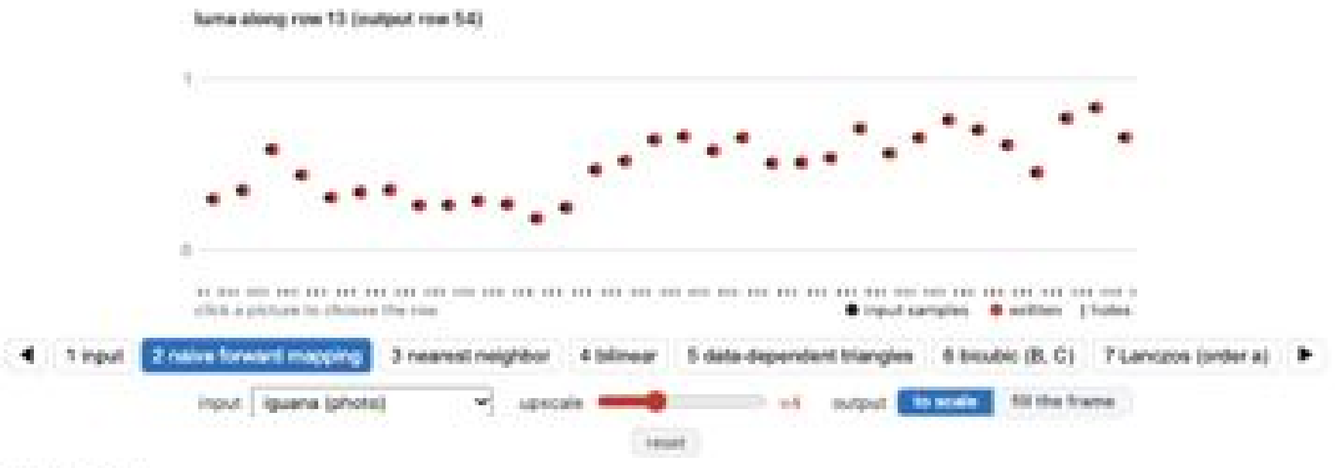






Demo: Figure 6.5.1, stage 2: naive forward mapping



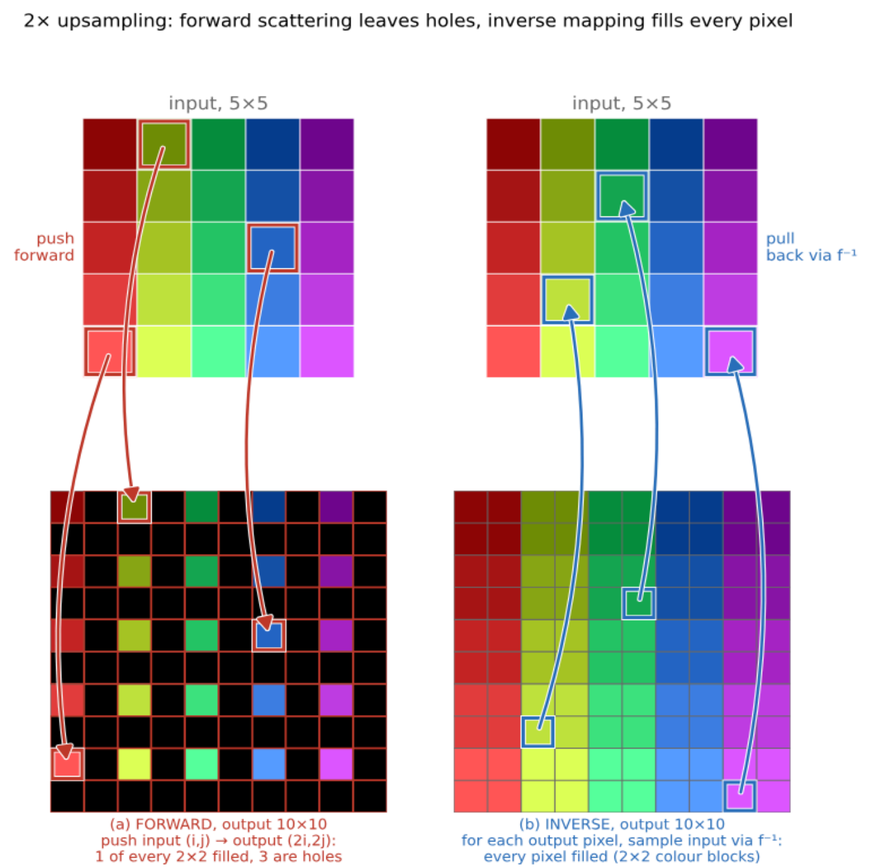


Demo courtesy of Frédo Durand, MIT.

Take-home message

- Main loop over OUTPUT pixels
- use INVERSE transform

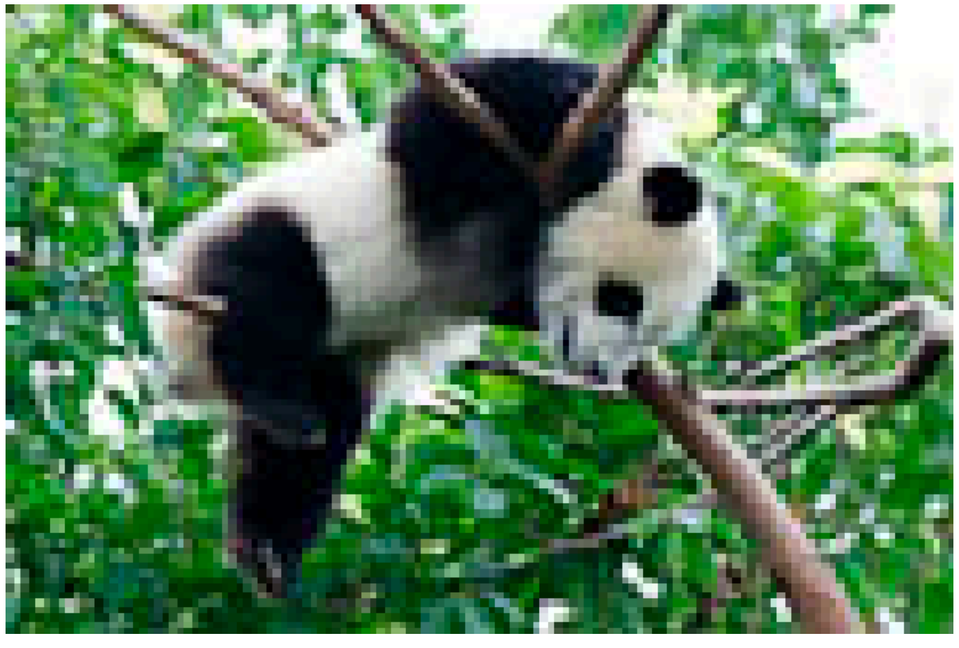
- Questions?



Basic resampling

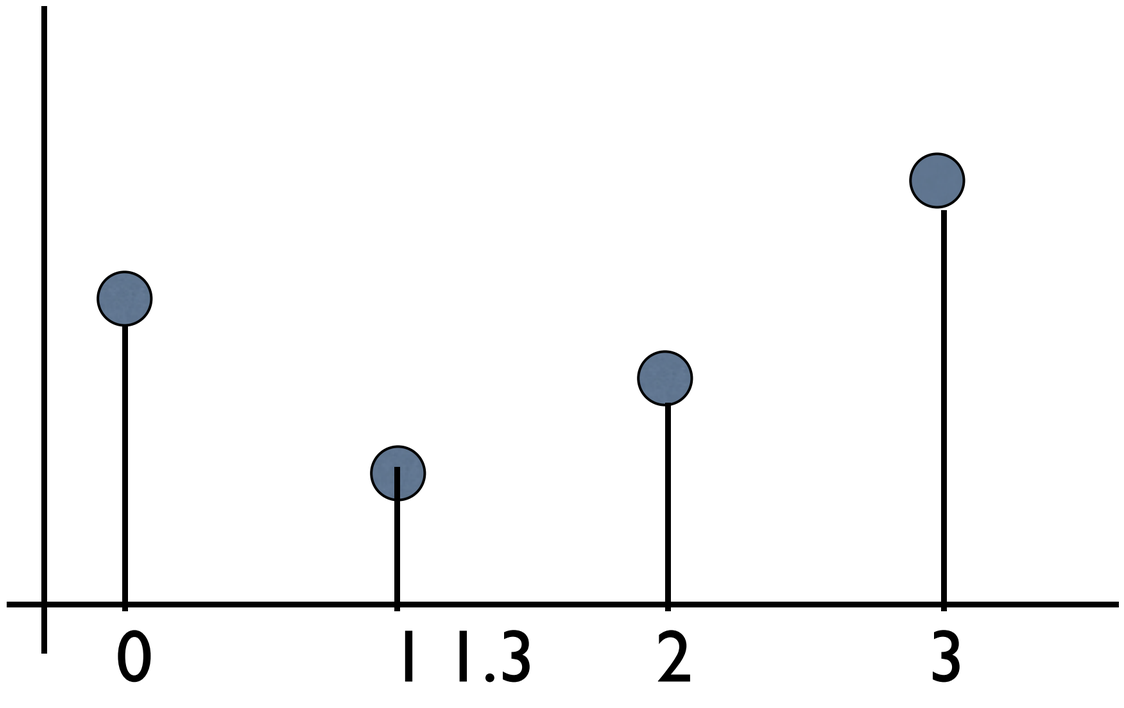
Remaining problem

- A little too “blocky”
- Because we round to the nearest integer pixel coordinates
  - called nearest neighbor reconstruction



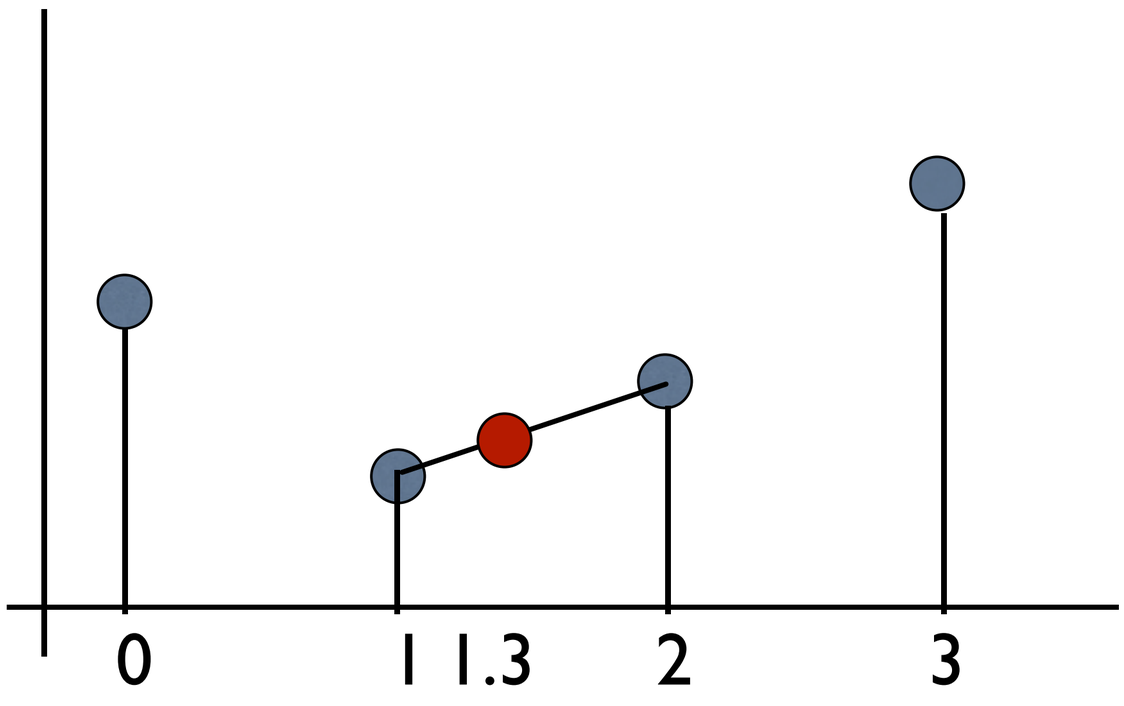
Linear reconstruction - 1D

- im is now a 1D array along x
- reconstruct im[1.3]



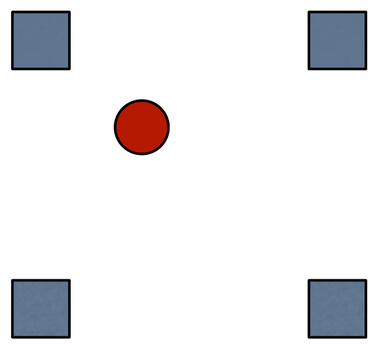
Linear reconstruction - 1D

- im is now a 1D array along x
- reconstruct im[1.3]
- =0.7*im[1]+0.3*im[2]

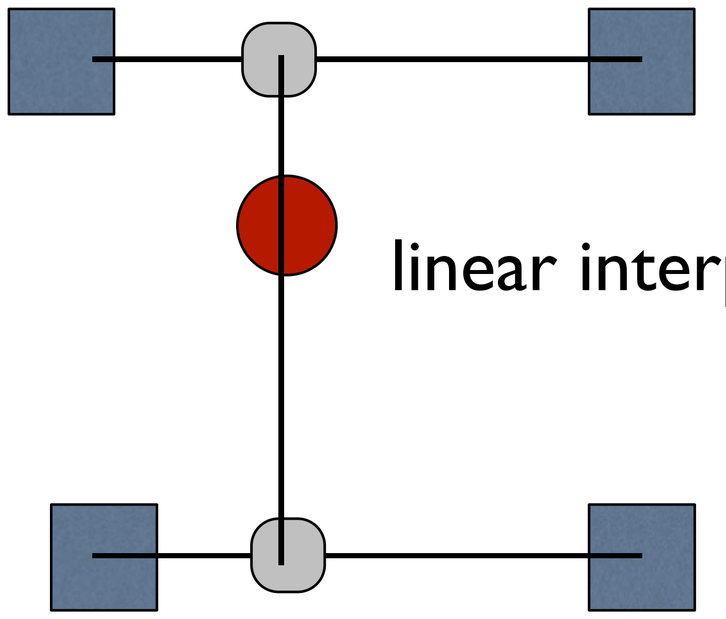


Bilinear reconstruction

- Take 4 nearest neighbors
- Weight according to x & y fractional coordinates
- Can be done using two 1D linear reconstructions along x then y (or y then x)



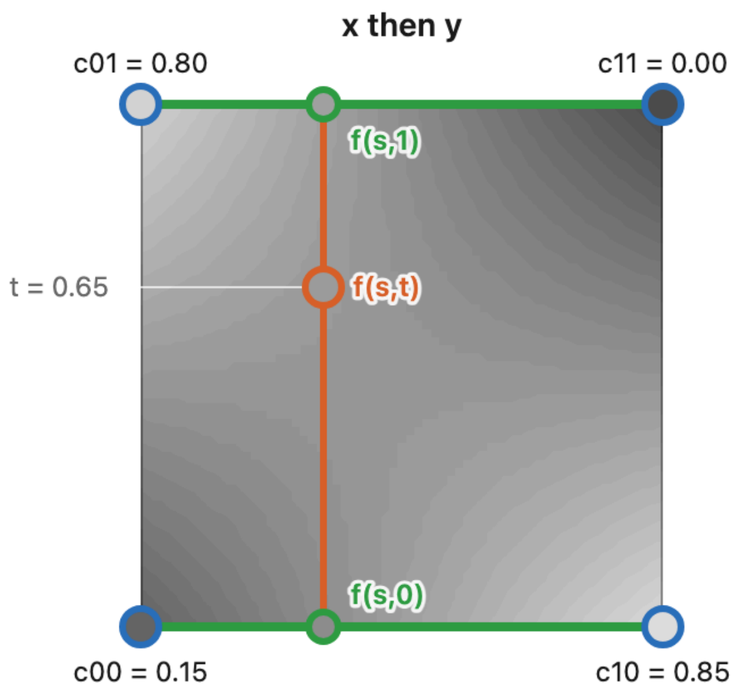
Bilinear

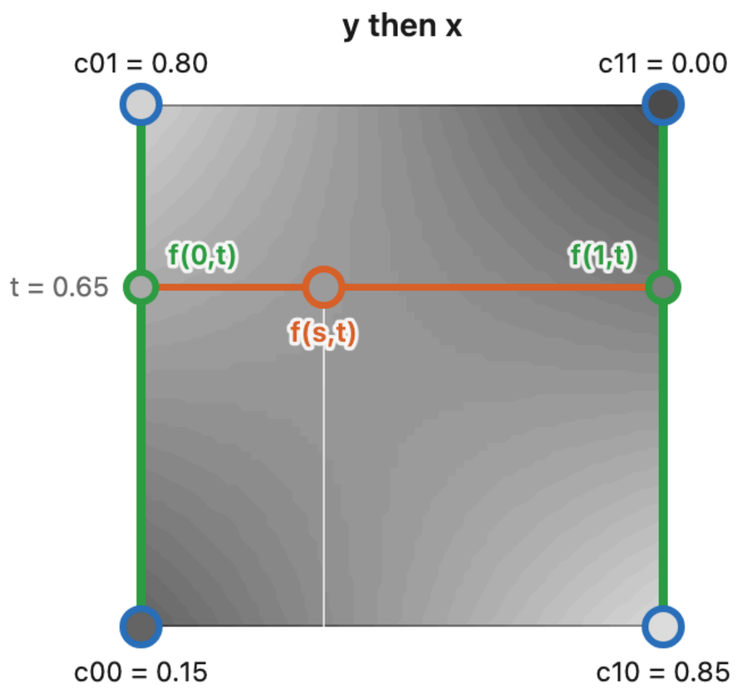


linear interpolation along x

linear interpolation along y

linear interpolation along x





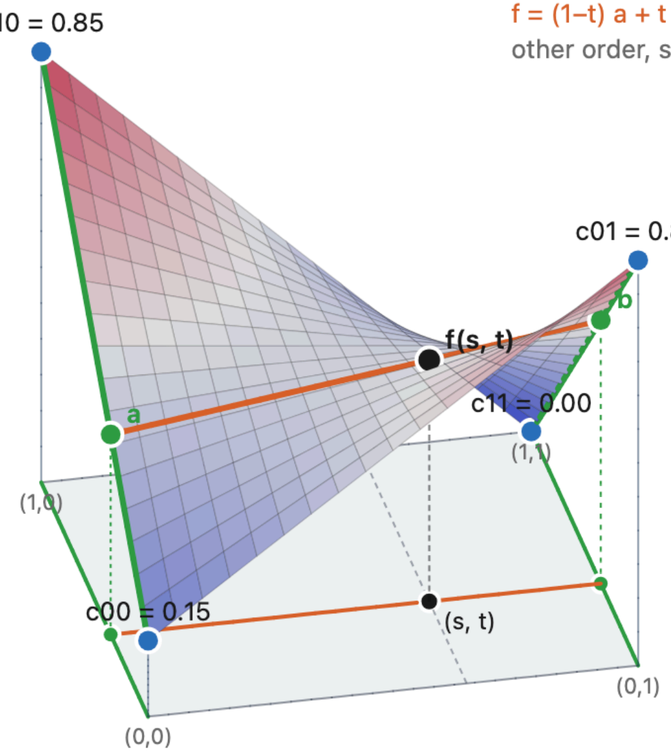
f(s,0) = (1−s) c00 + s c10 = 0.395
f(s,1) = (1−s) c01 + s c11 = 0.520
f(s,t) = (1−t) f(s,0) + t f(s,1) = 0.476

f(0,t) = (1−t) c00 + t c01 = 0.573
f(1,t) = (1−t) c10 + t c11 = 0.297
f(s,t) = (1−s) f(0,t) + s f(1,t) = 0.476

either order: f(s,t) = (1−s)(1−t) c00 + s(1−t) c10 + (1−s)t c01 + st c11 = 0.476

Demo: Bilinear height-field

a = (1–s) c00 + s c10 = 0.395
b = (1–s) c01 + s c11 = 0.520
f = (1–t) a + t b = 0.476
other order, same f = 0.476

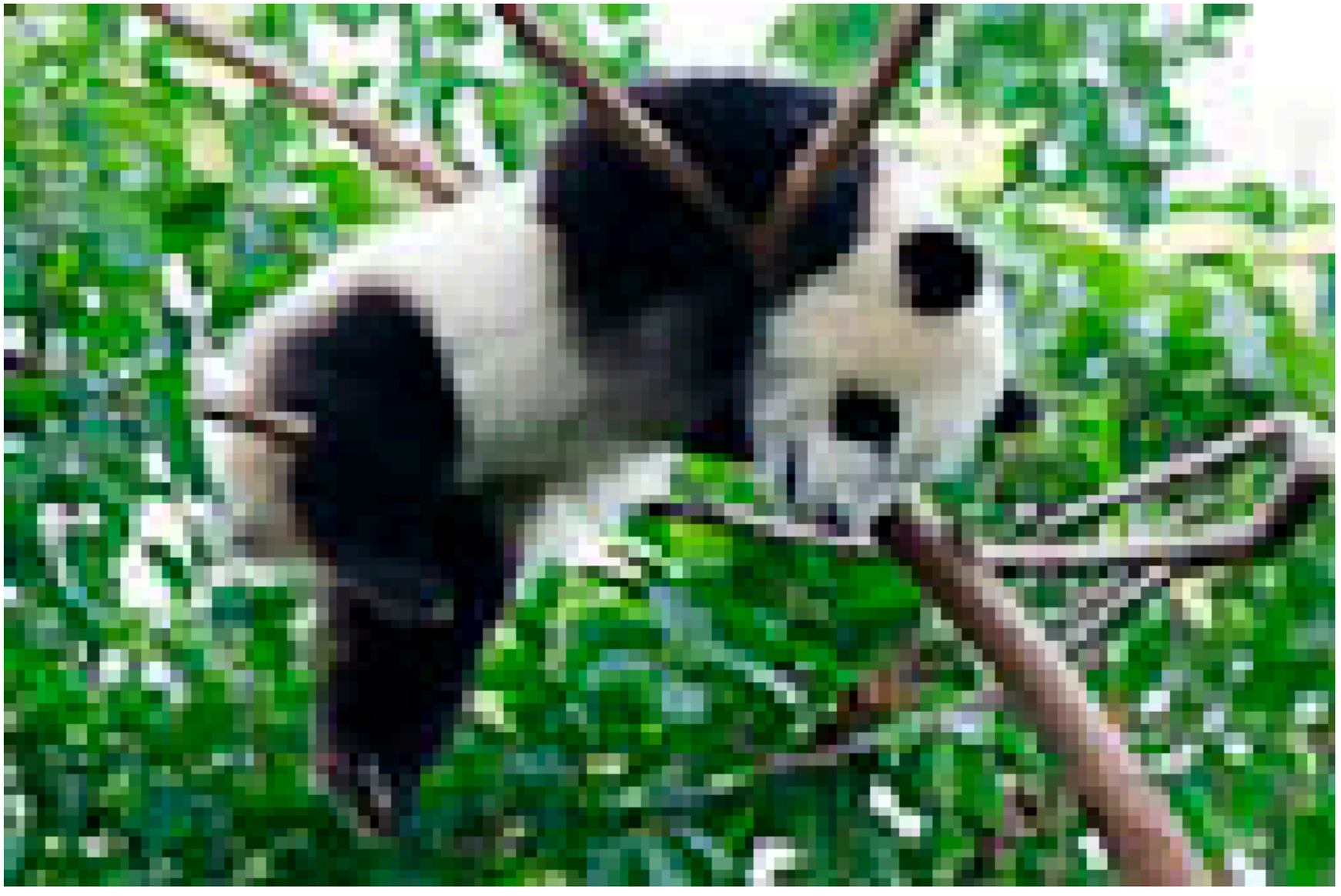


non-coplanar → saddle: straight grid lines

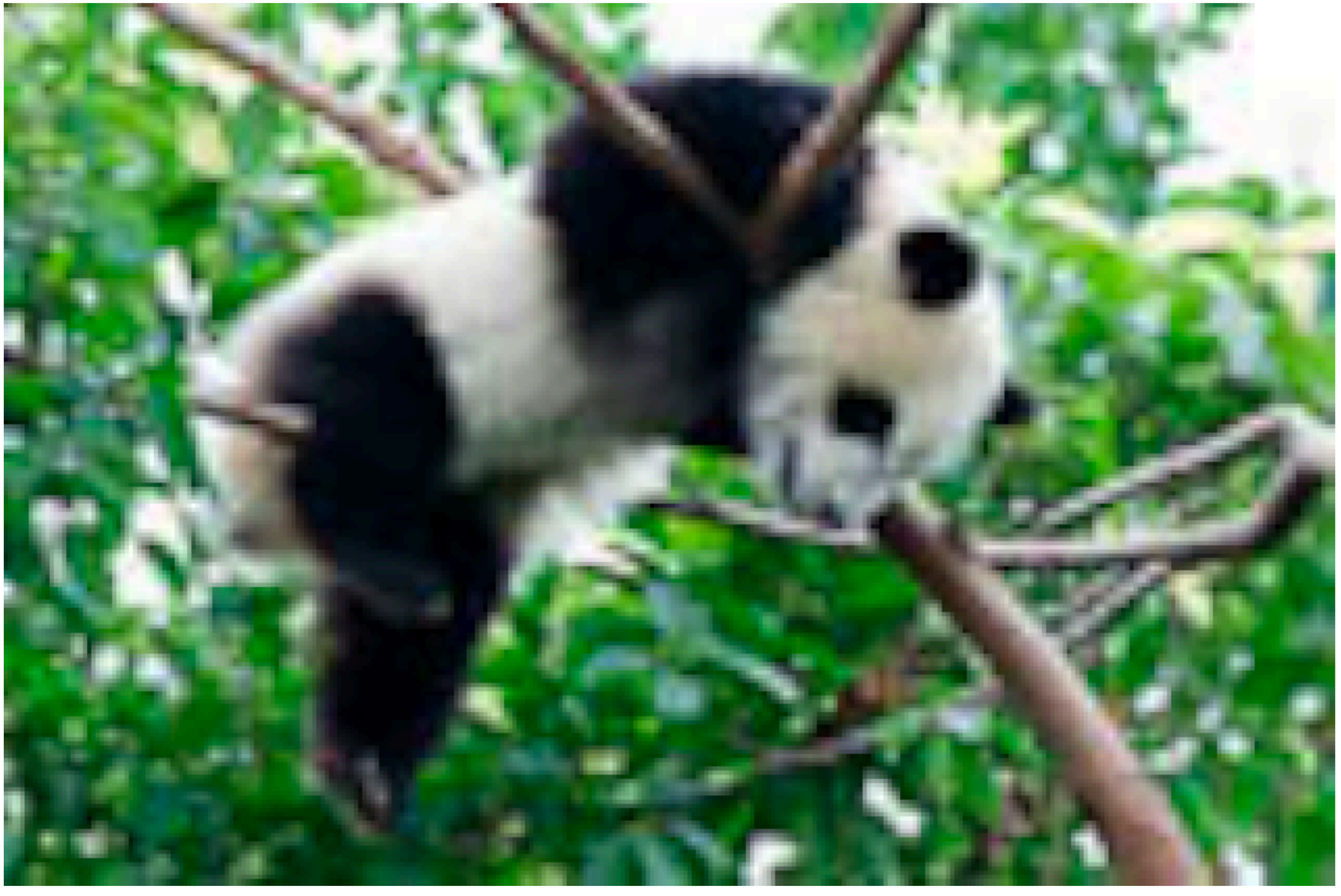
Exercise

- Prove it’s the same if you interpolate along y first

Recall nearest neighbor



Bilinear



Demo: Figure 6.5.1, stage 4: bilinear



Demo courtesy of Frédo Durand, MIT.

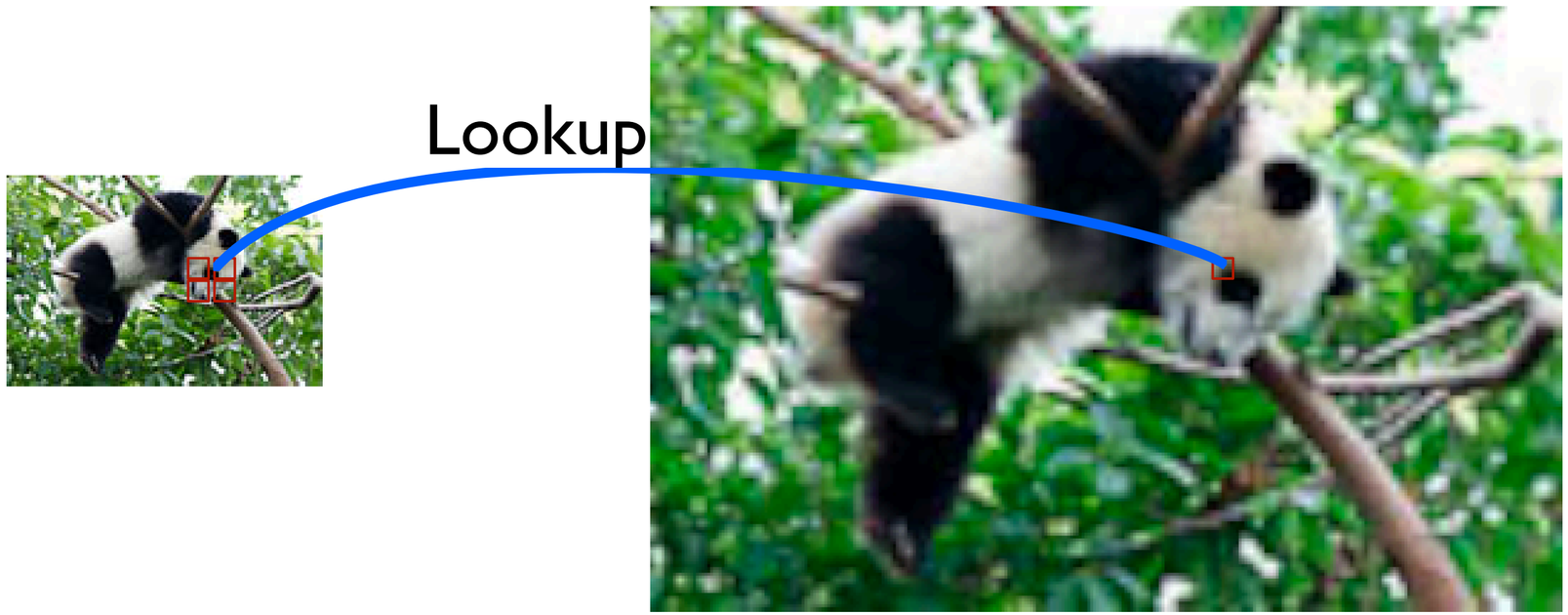
Demo: Figure 6.5.1, stage 5: data-dependent triangles



Demo courtesy of Frédo Durand, MIT.

Take home messages

- **Main loop over OUTPUT pixels**
  - Makes sure you cover all of them
- **Use inverse transform**
- **Reconstruction makes a difference**
  - Linear much better than nearest neighbor



Questions?



General & Bicubic Resampling

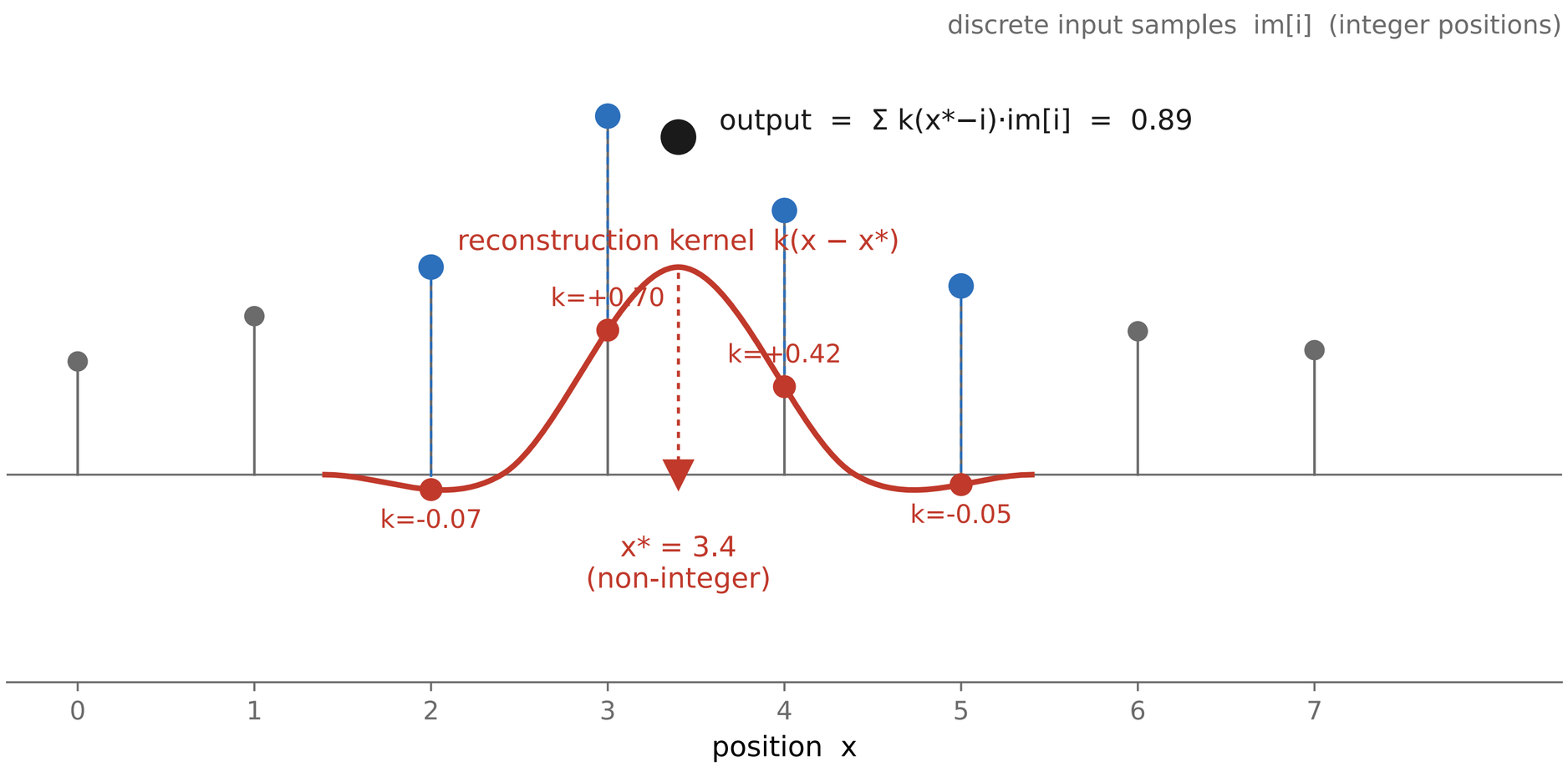
Idea

- Output is weighted average of neighbors in input
- Weights given by ANALYTICAL convolution kernel
- centered at non-integer coordinates (desired output location)

Idea in 1D

Figure 6.5.6: Why the kernel must be a continuous function

Resampling = evaluate the analytic kernel at the query x*, tap the neighbours, sum

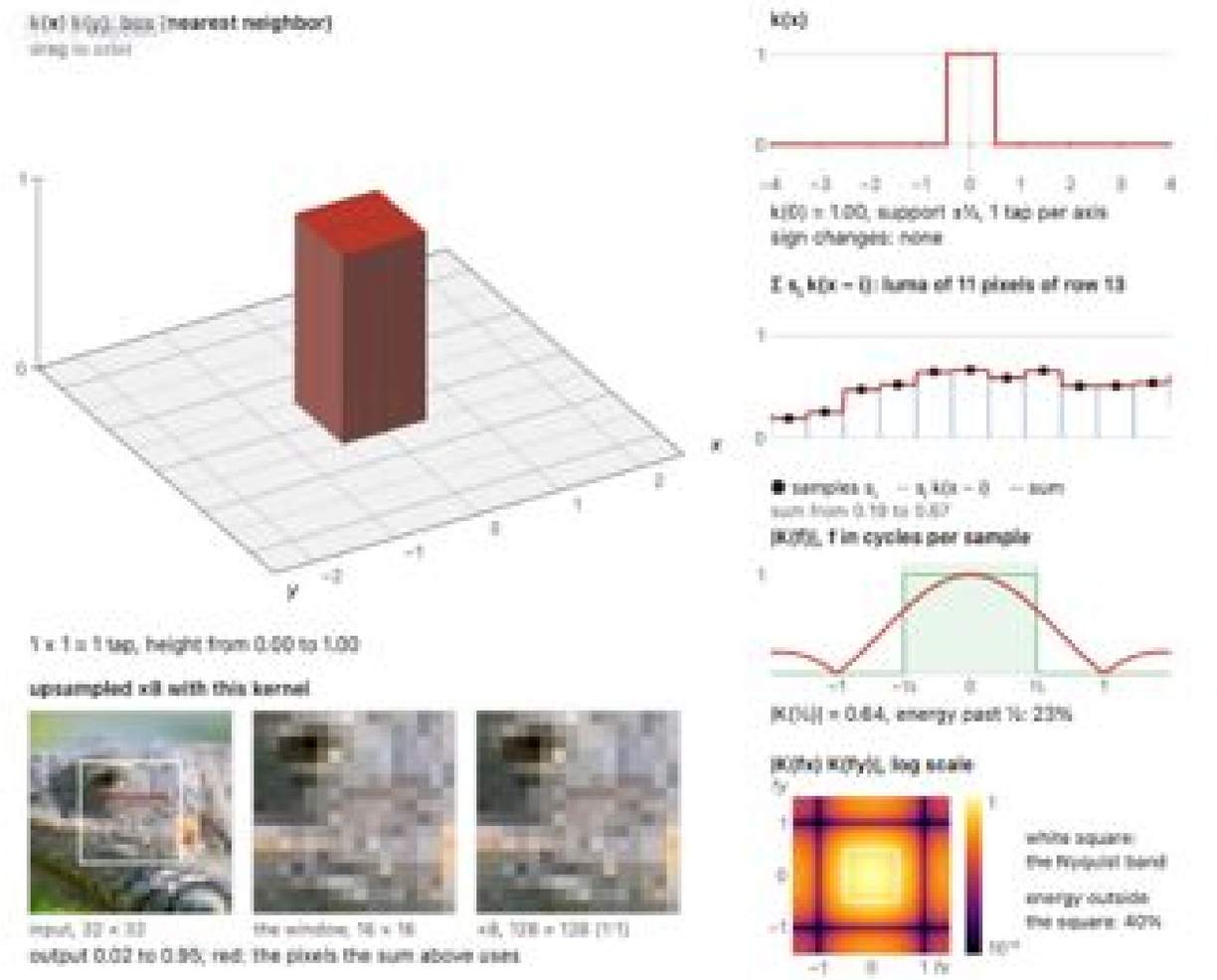


A row of discrete input samples sits on the integer grid; a smooth kernel k(x) is centered at a non-integer query location x^* between two of them, and the output…

Reinterpreting bilinear

- (Because earlier exposition does NOT generalize to bicubic)

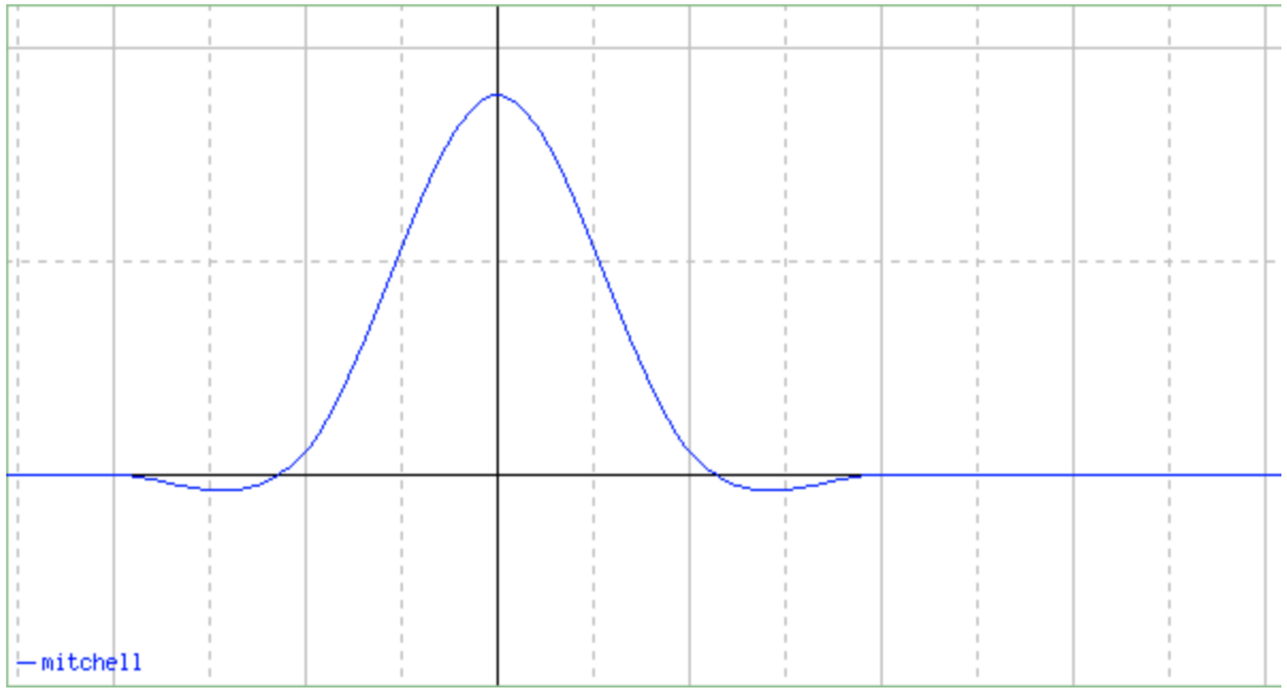
Demo: Figure 6.5.7, stage 1: nearest neighbor, the box

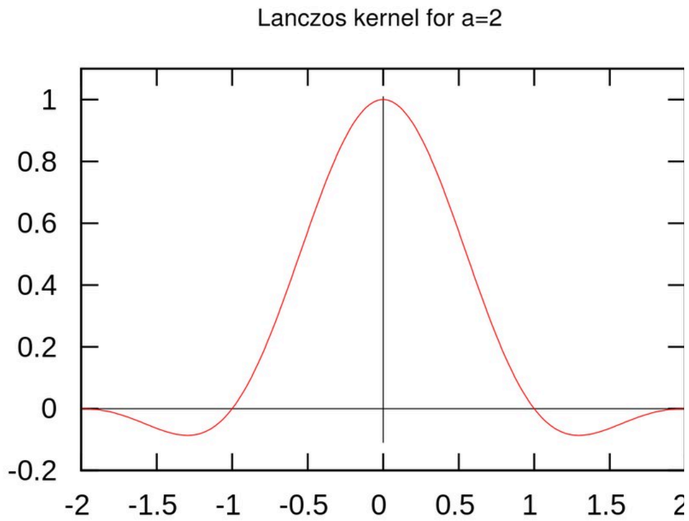


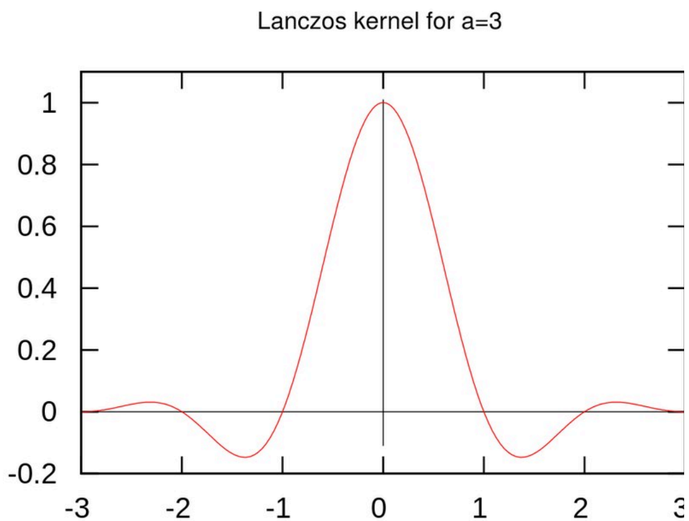
Demo courtesy of Frédo Durand, MIT.

Some kernels

- Lanczos: sinc(π x) sinc(π x/a)
- Various bicubic

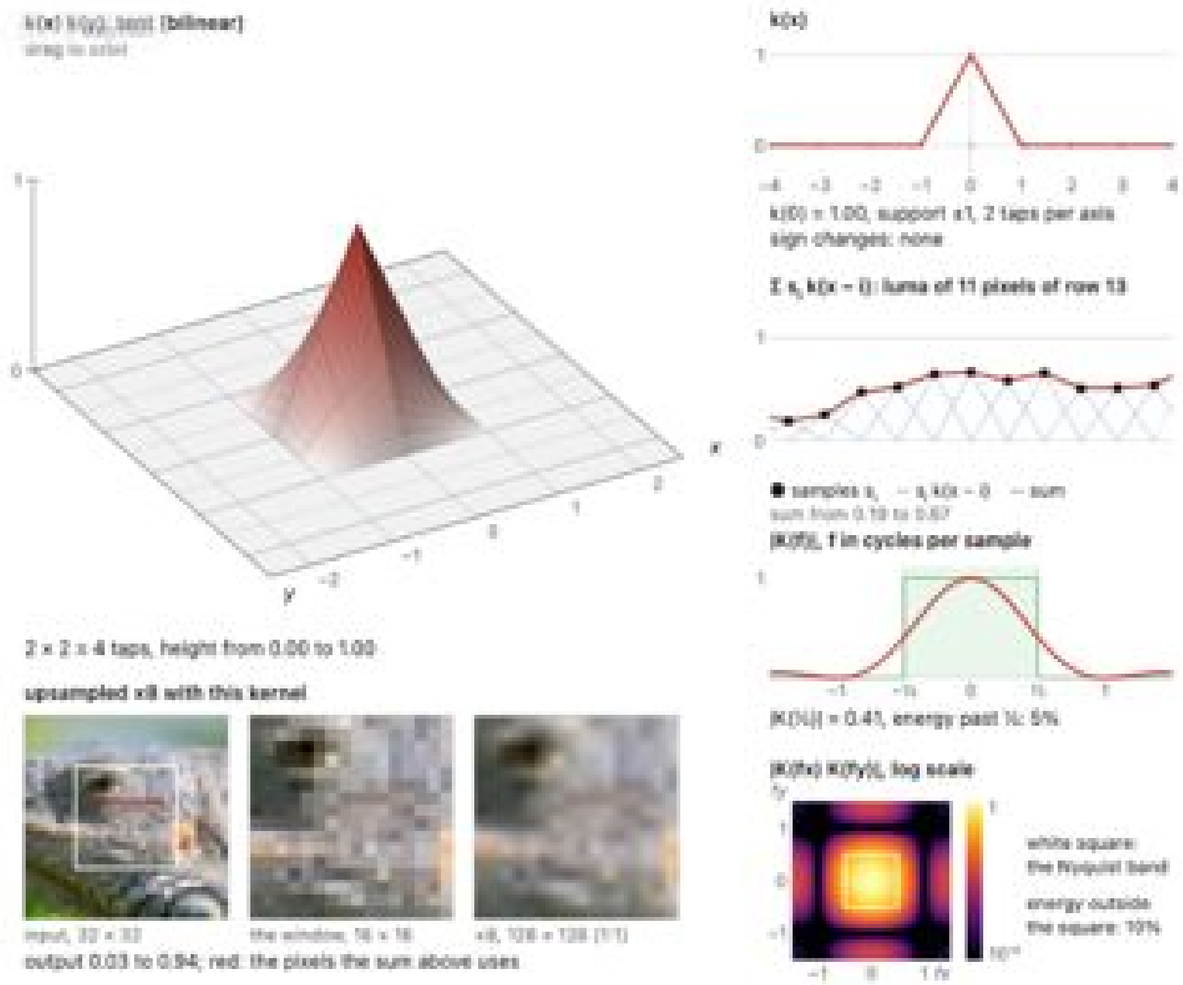






http://entropymine.co

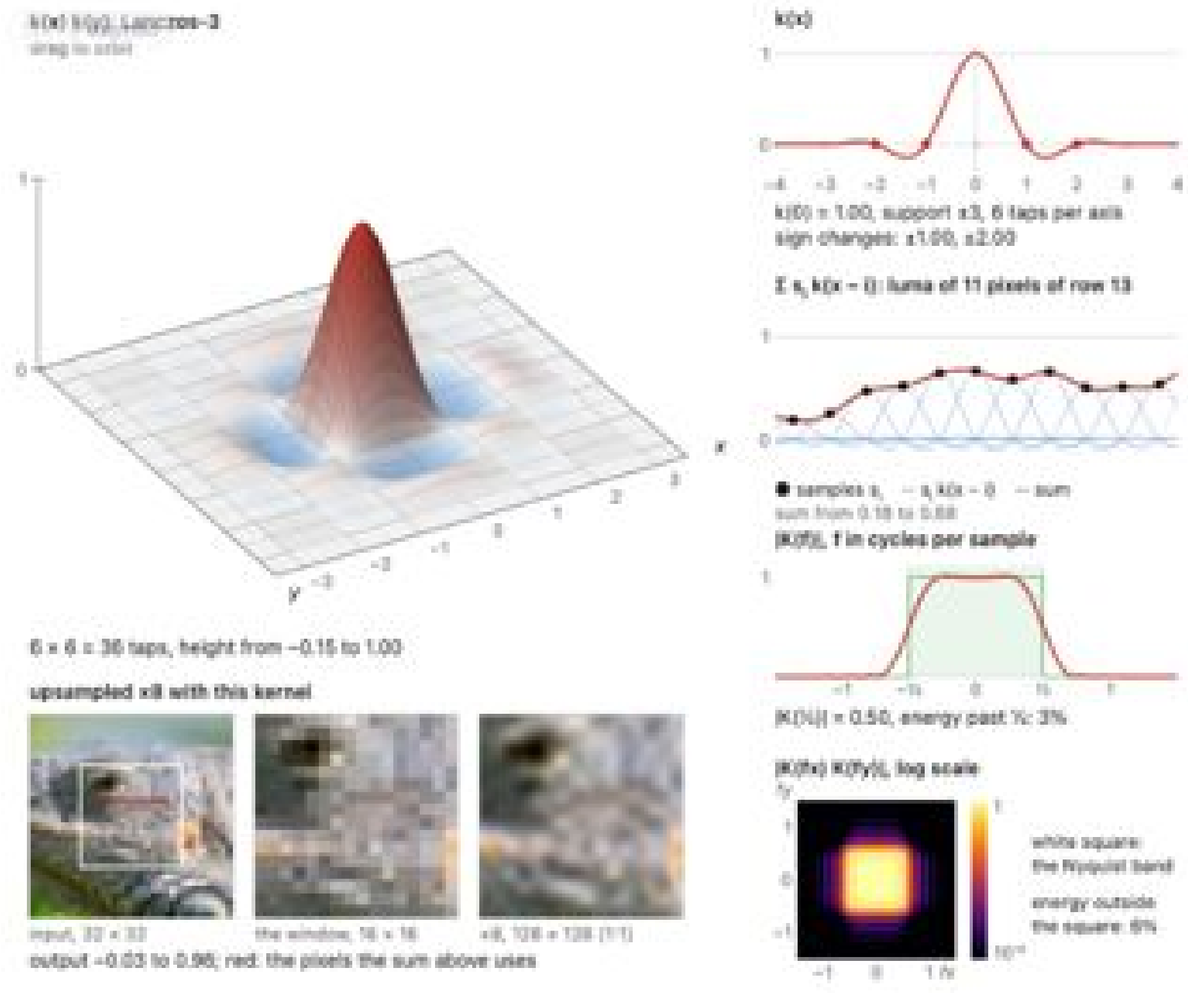
Demo: Figure 6.5.7, stage 2: bilinear, the tent



Demo courtesy of Frédo Durand, MIT.

How do we express the kernel?

Demo: Figure 6.5.7, stage 4: Lanczos, windowed sinc



Demo courtesy of Frédo Durand, MIT.

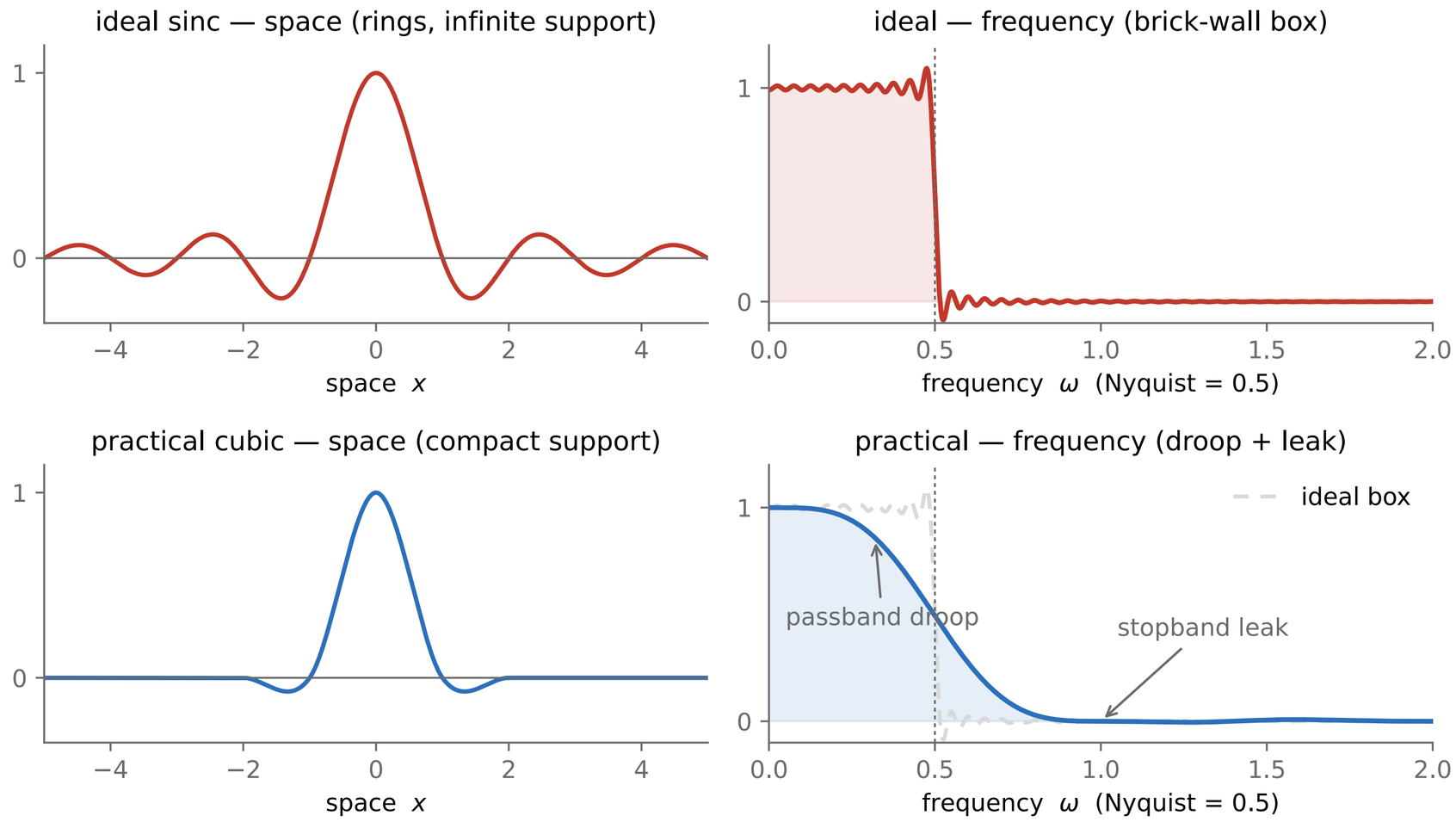
Demo: Figure 6.5.1, stage 7: Lanczos (order a)



Demo courtesy of Frédo Durand, MIT.

Figure 6.4.7: Ideal sinc vs a practical filter, in both domains

the perfect filter is unrealizable; practical filters trade blur vs aliasing



Top row (space): the ideal sinc reconstruction kernel ripples on forever, while a practical filter has finite support.

What this is NOT

- It is NOT about fitting a cubic to the data
- Don’t believe what you see online or even in books

Mitchell/Netravali bicubic filterS

- https://www.cs.utexas.edu/~fussell/courses/cs384g-fall2013/lectures/mitchell/Mitchell.pdf

Reconstruction Filters in Computer Graphics

Don P. Mitchell
Arun N. Netravali

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

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

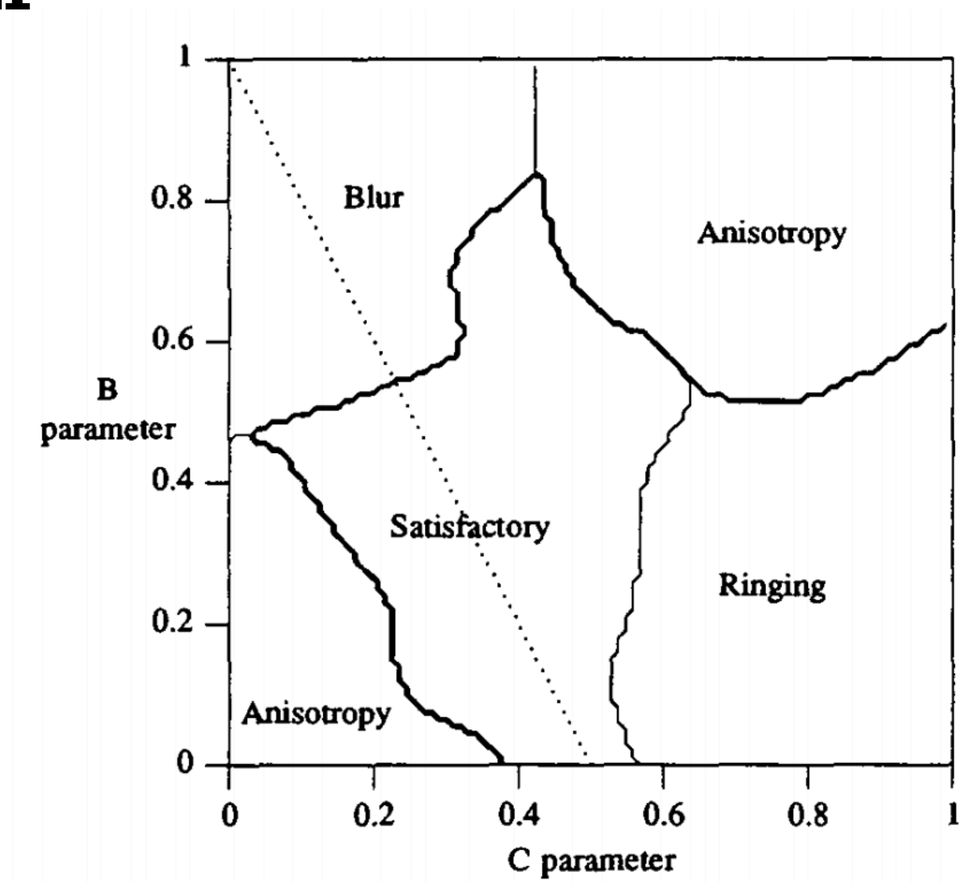


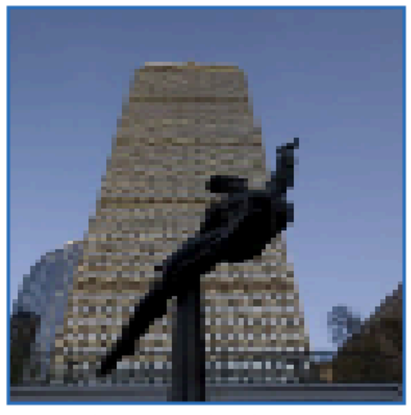
Figure 13. Regions of Dominant Subjective Behavior

Demo: Mitchell–Netravali cubics

source (72 px)



nearest · 2.2×

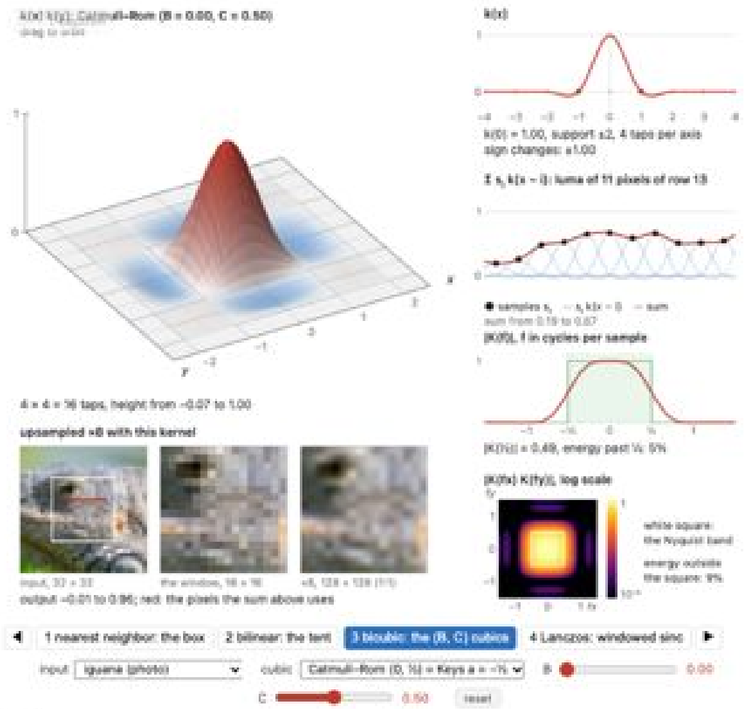


cubic (B,C) · 2.2×



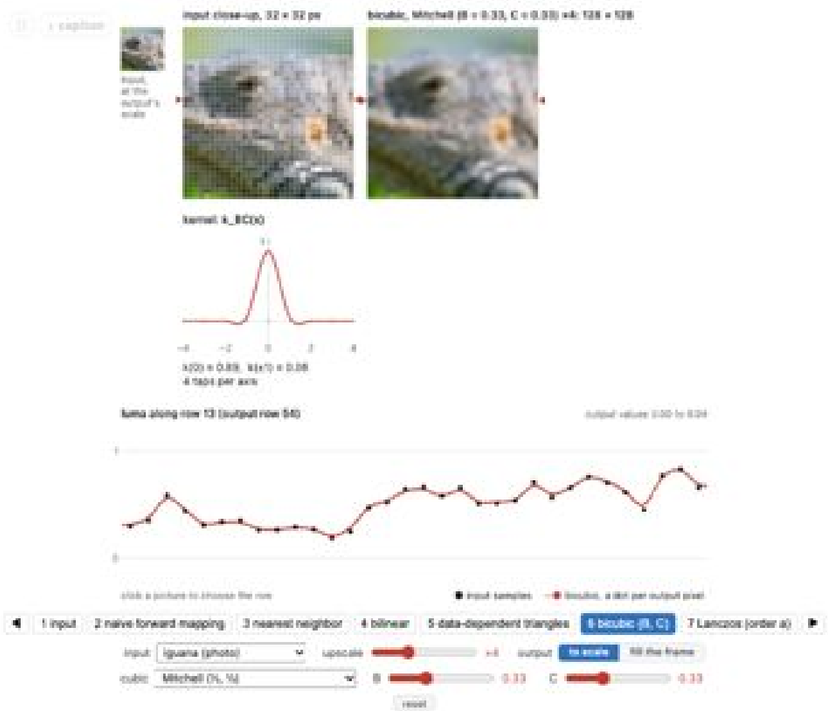
Upsampling: cubic reconstruction smooths the samples; negative lobes can ring.
Nearest uses point samples. Blur and residual aliasing depend on B and C.

Demo: Figure 6.5.7, stage 3: bicubic, the (B, C) cubics



Demo courtesy of Frédo Durand, MIT.

Demo: Figure 6.5.1, stage 6: bicubic (B, C)



Demo courtesy of Frédo Durand, MIT.

Basic implementation

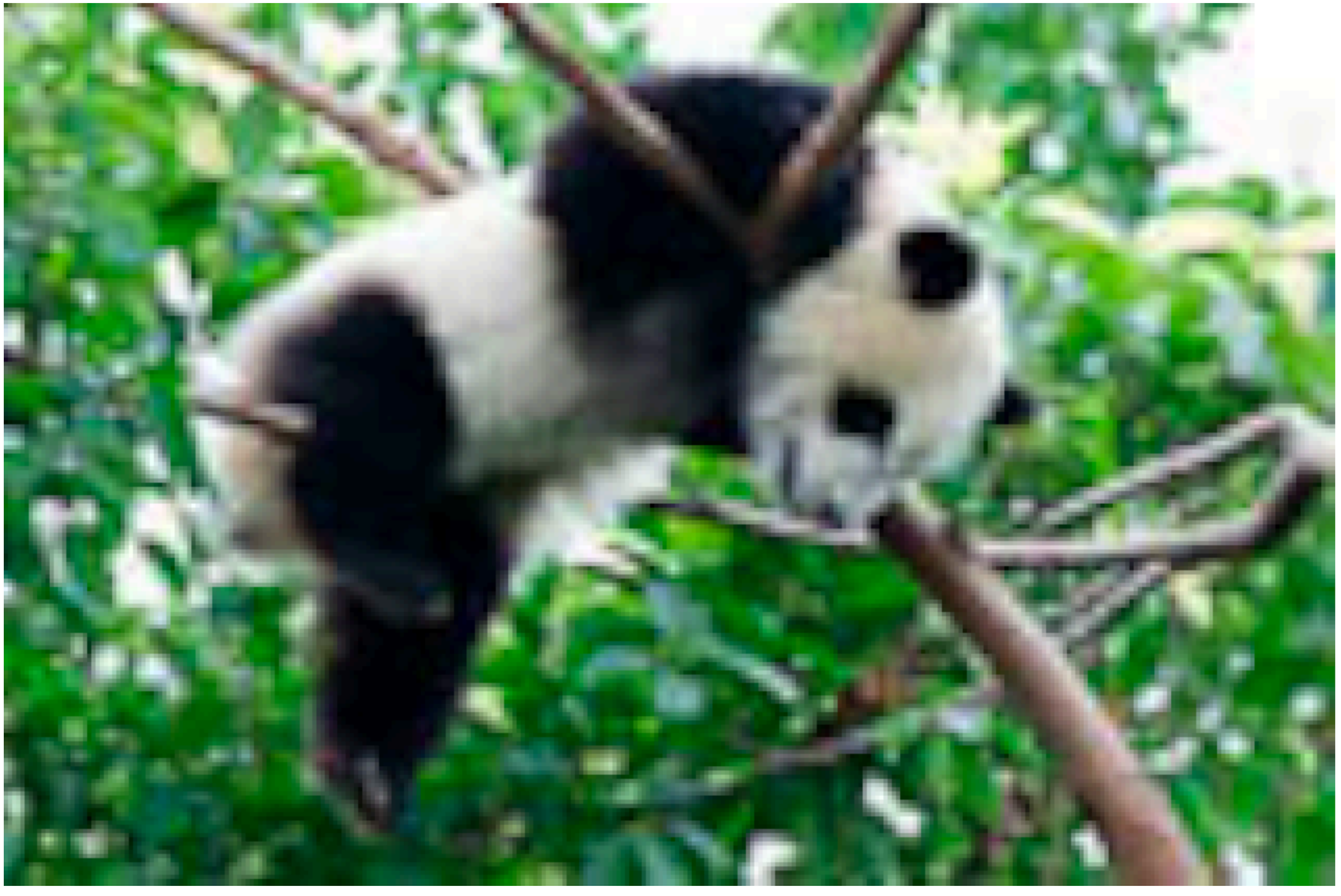
- **For each output pixel x, y**
  - Compute (fractional) inverse-transformed pixel coordinates $f^{-1}(x,y)$
    - For each input pixel x', y' in neighborhood of $f^{-1}(x,y)$
      - Compute kernel at $f^{-1}(x,y)$-(x', y')
      - Accumulate kernel * I(x', y')

- **Note: in general you can't pre-compute the kernel because the fractional values can be different for each pixel**
  - Although there are tricks

In practice

- **separable (scale in x, then in y)**
  - Intermediate image of size (scale*width ; height) (anisotropically scaled)
- **basic implementation:**
  recompute weights for each pixel each neighbor
- **smarter: separable, precompute weights by row/column, take advantage of fractional or integer factors**

Bilinear



Bicubic (Photoshop)



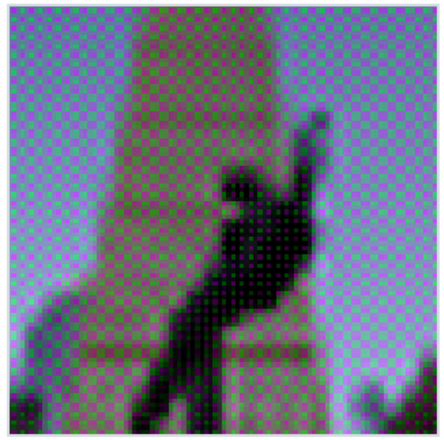
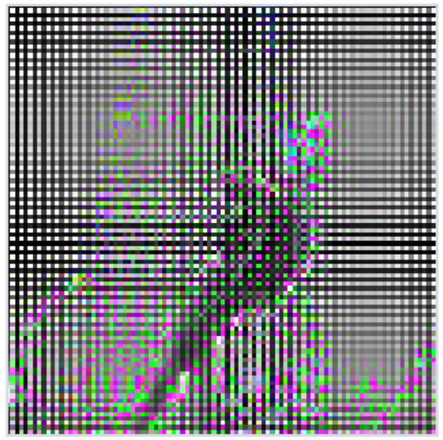
Better reconstruction

- Consider more than 4 pixels:
  - bicubic, Lanczos, etc.
- Try to sharpen edges
- Use training database of low-res/high-res pairs
  - **http://people.csail.mit.edu/billf/superres/index.html**
  - Deep learning

Demo: Everything the measurement cannot see

four different pictures that measure exactly the same

×4 downsampling: each candidate is projected onto the set consistent with the input

the measurement	the candidate	downsampled again	error against the truth
			
24×24 pixels		identical to the measurement	

consistency with the measurement: exact

agreement with the truth: 16.77 dB

Every candidate passes the first test. Only one passes the second, and the measurement cannot tell them apart.

Bill Freeman's super-resolution





Demo: One crop, five methods, one control

five toy upscalers, ×2, with the truth

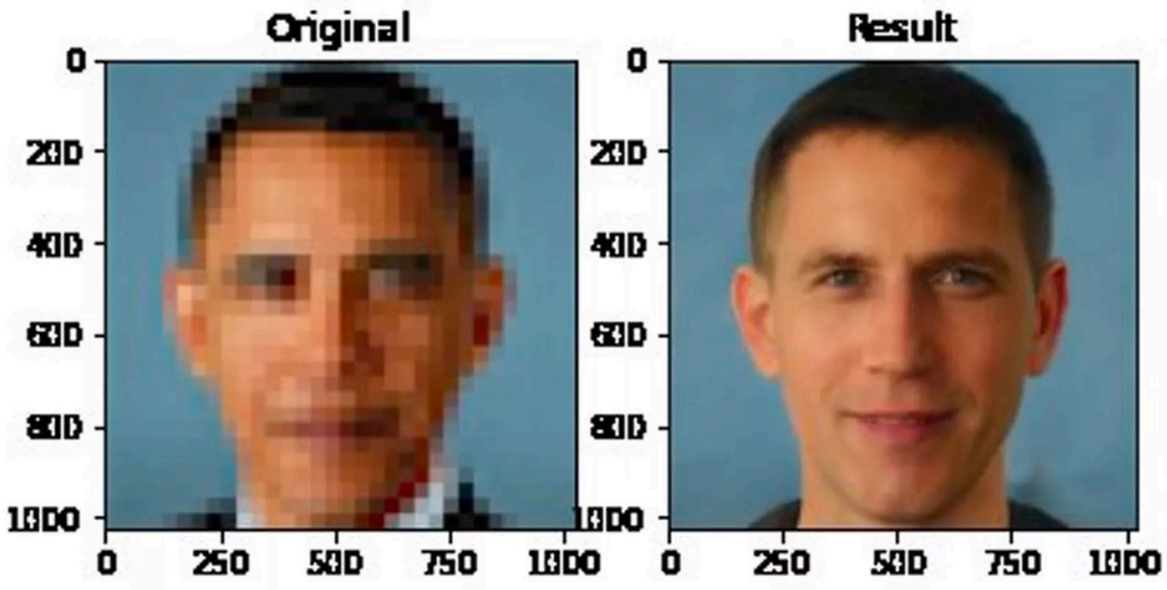
only patch lookup uses the external database; within-image detail borrows from the input alone



Texture matching uses clean-reference energies. These small examples do not reproduce published SR methods.

Better reconstruction

- Careful of fairness and bias when using ML
  - **https://www.theverge.com/21298762/face-depixelizer-ai-machine-learning-tool-pulse-stylegan-obama-bias**
  - **https://thegradient.pub/pulse-lessons/**

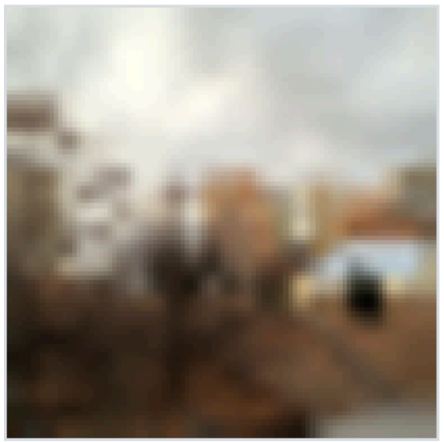


Demo: **The same crop upscaled by bicubic interpolation and by a seeded…**

compare upscaled detail with the known reference

×4 · toy texture statistics use the clean reference

bicubic interpolation



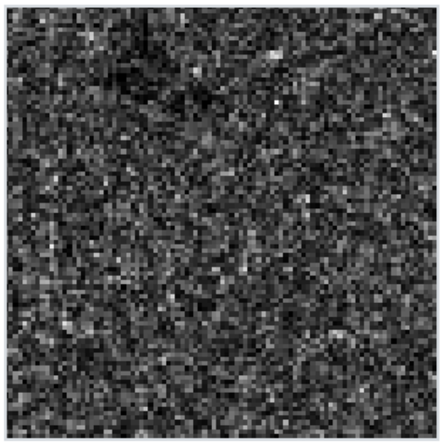
24.44 dB

texture-matching



21.97 dB

disagreement across runs



red-channel SD, rescaled

truth hidden

decide first

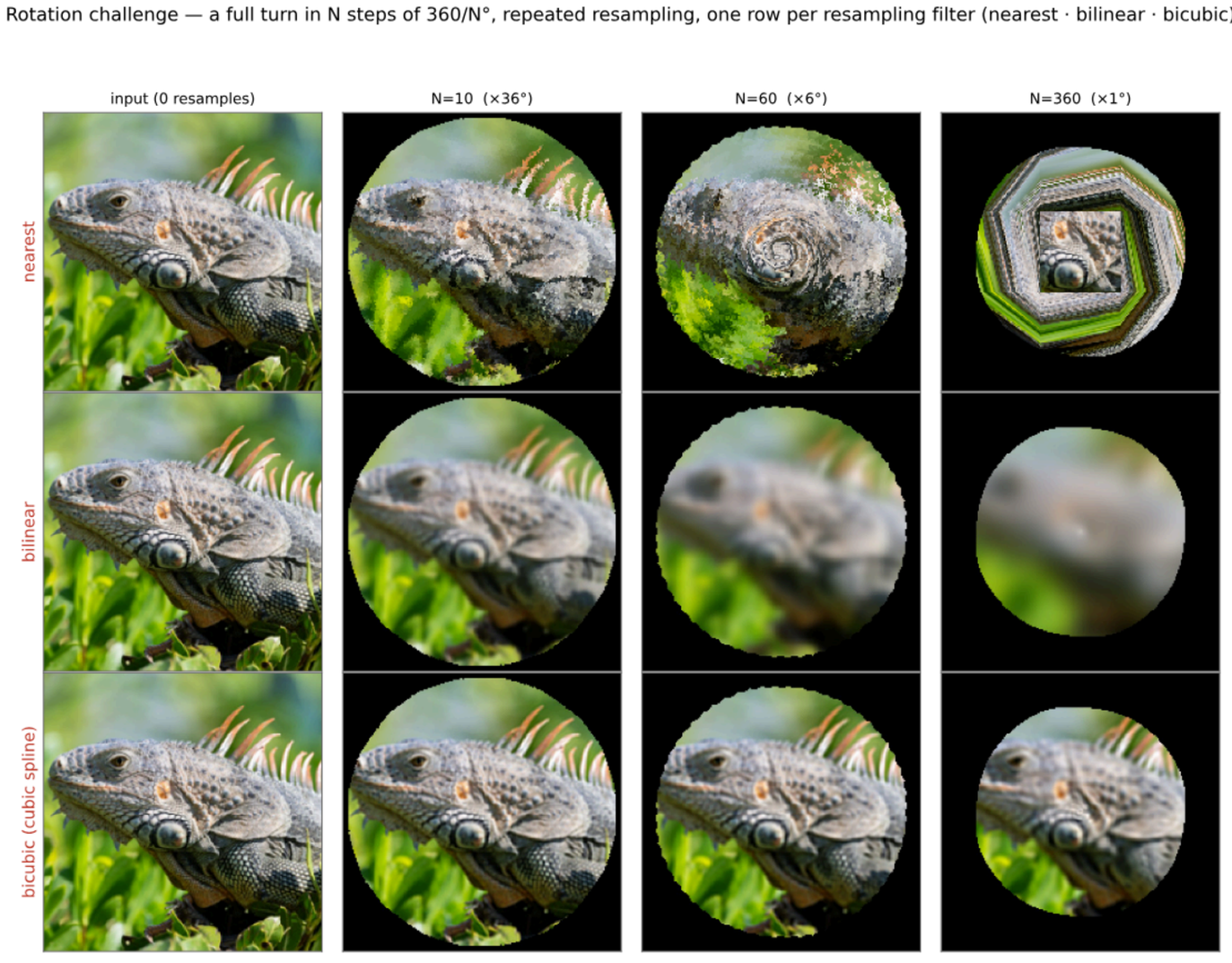
Reveal the reference to check specific leaves, building features, or facial details.

Fixed texture strength, independent random seeds. Agreement between runs does not establish accuracy.

Questions?



Figure 6.5.21: Andrew Adams' rotation challenge, and how much the resampling filter…

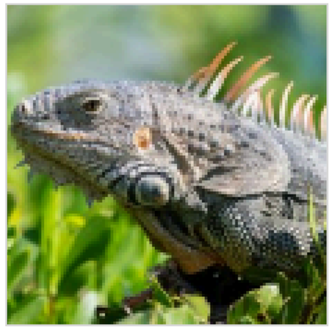


More, smaller rotations compound more damage; bilinear washes detail into mush by N=360, while bicubic (with its negative lobes) holds the iguana's scales far longer. Each step is a real reconstruct-then-resample at the labelled filter order.

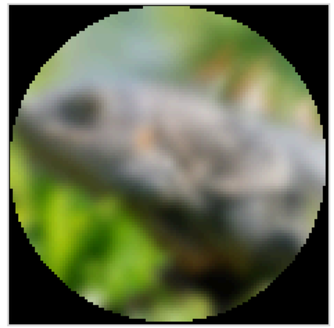
Andrew Adams' rotation challenge, and how much the resampling filter matters.

Demo: Rotation resample challenge

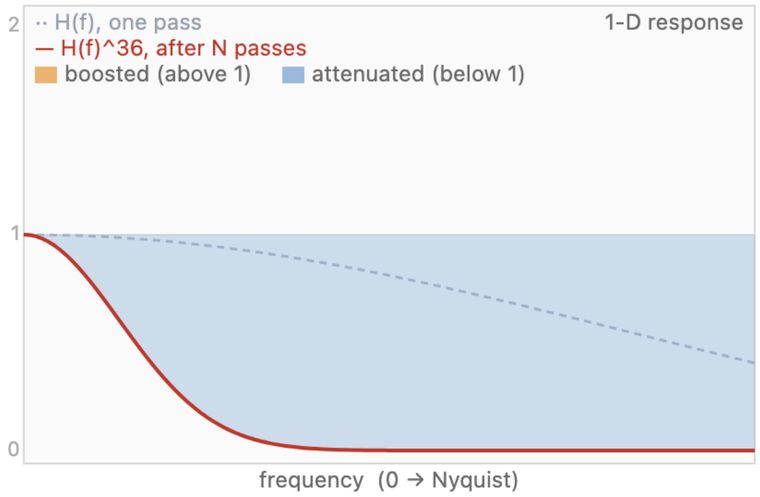
original



after 36 × 10°

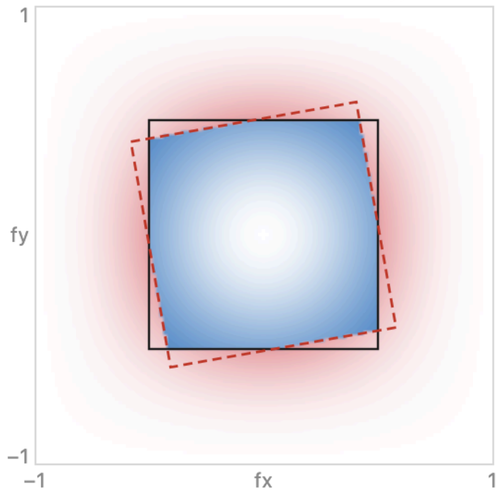


N = 36 passes of 10° = 360° total (back to upright) · bilinear

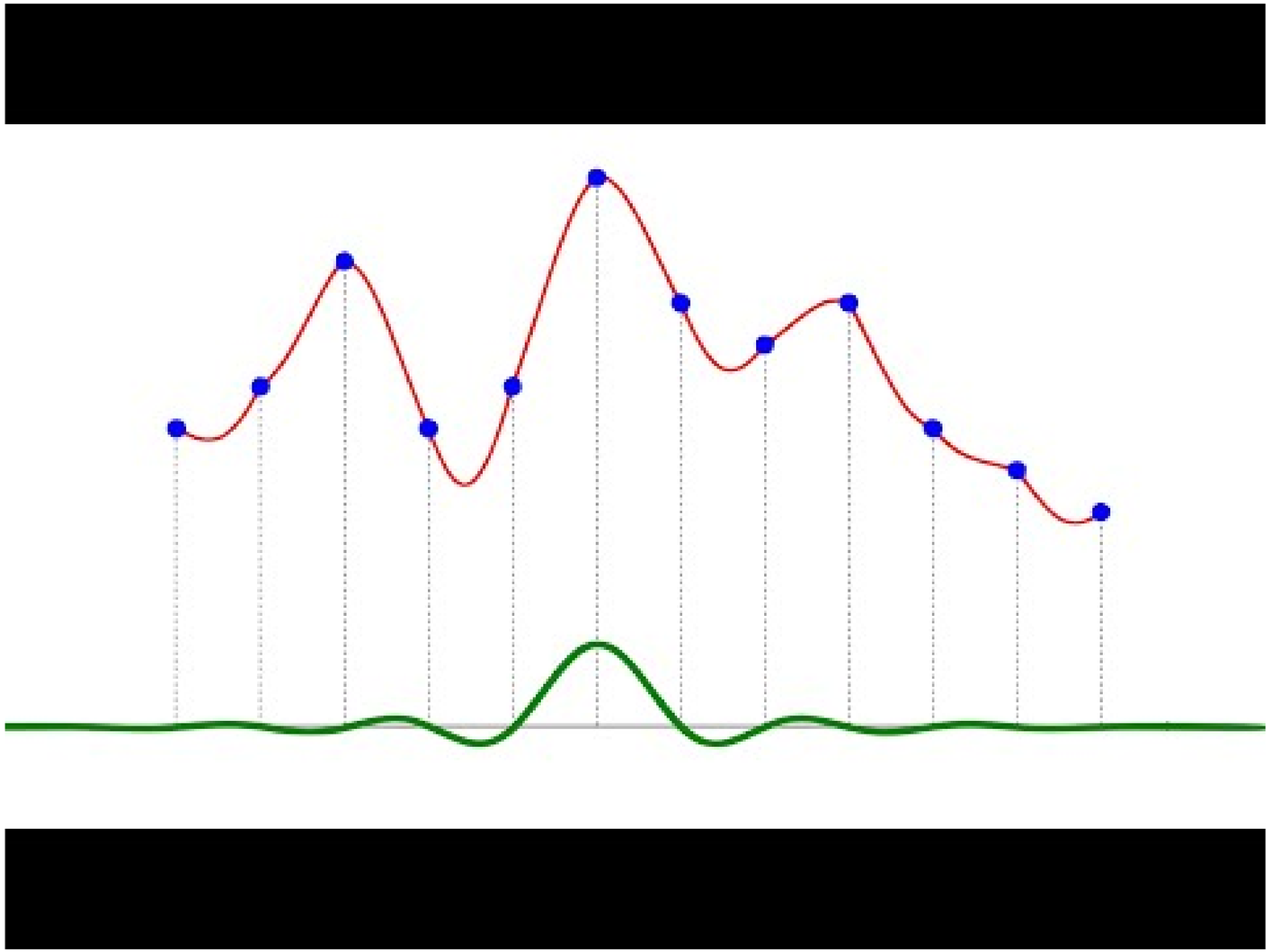


|H(fx) H(fy)|, one pass

aliased: 10.7% of the energy



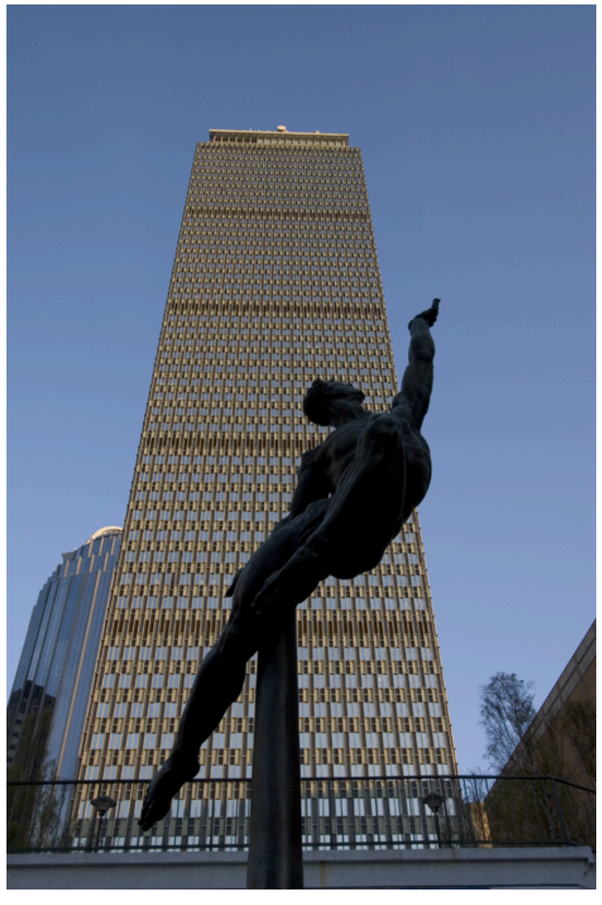
Video: **Interpolation in 5 minutes, a short visual tour of nearest-neighbor, linear…**

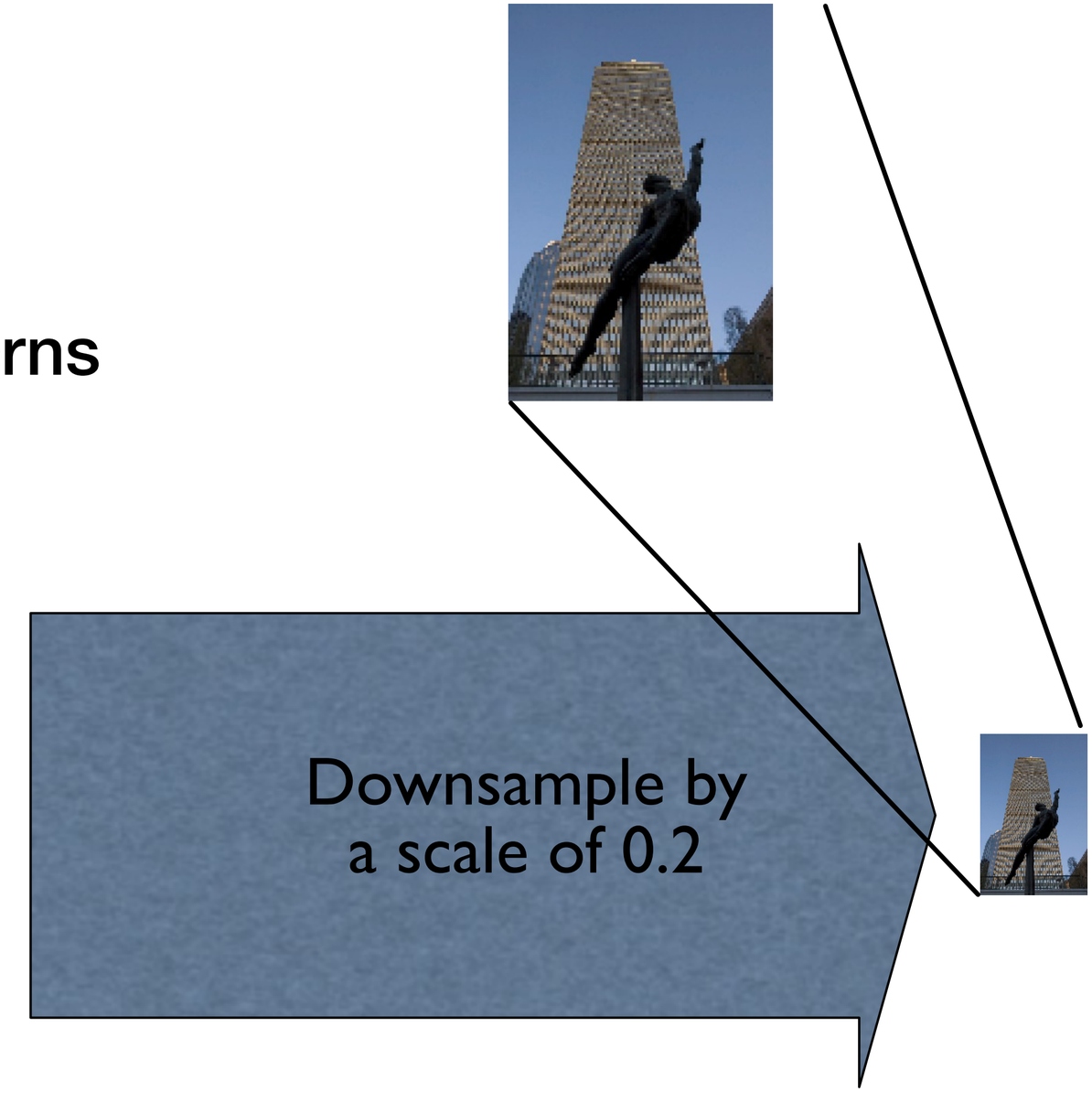


Downsampling

Downsampling

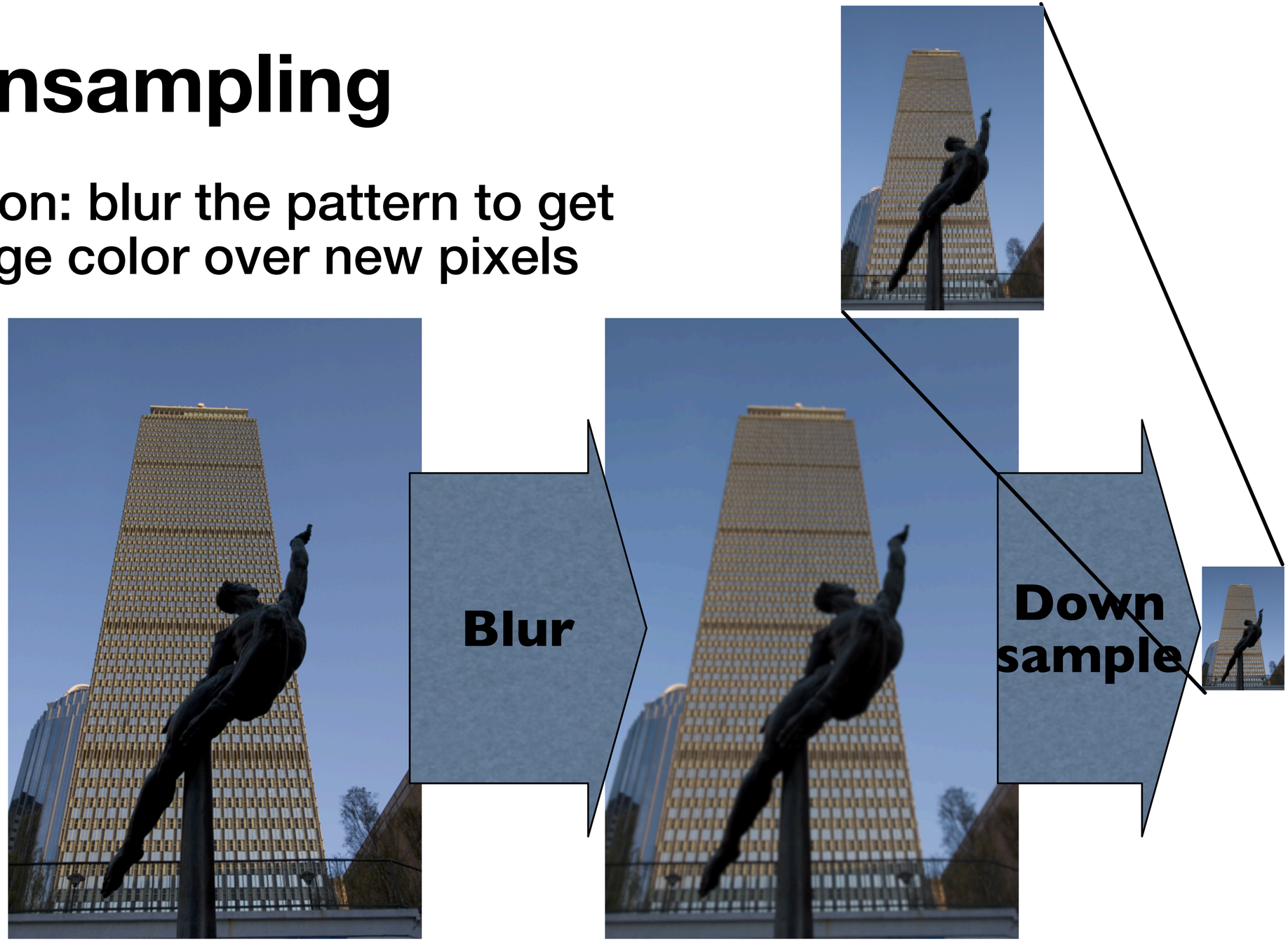
- Using bilinear
- Yikes! Herringbone patterns



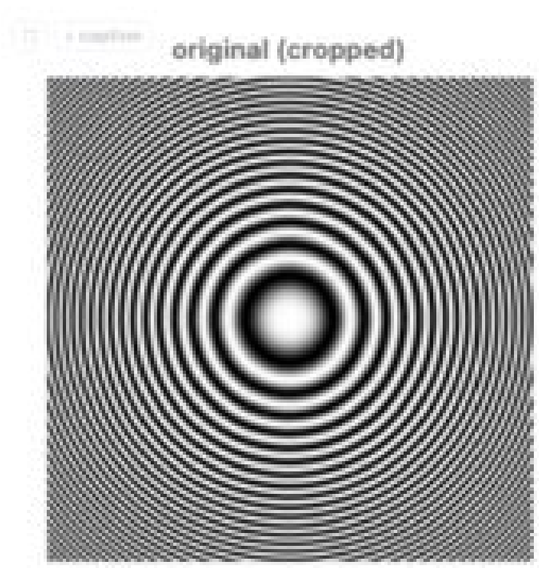


Downsampling

- Solution: blur the pattern to get average color over new pixels



Demo: Figure 6.4.5, stage 1: input

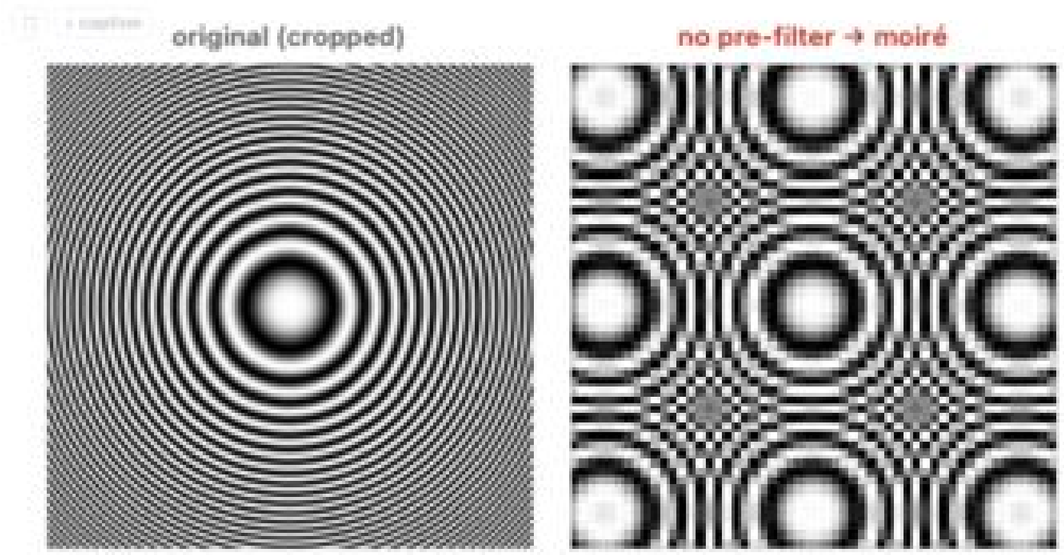


1 input | 2 aliased: point sampling | 3 pre-filtered

source: zone plate | downsample factor: ×4 | reset

Demo courtesy of Frédo Durand, MIT.

Demo: Figure 6.4.5, stage 2: aliased, point sampling



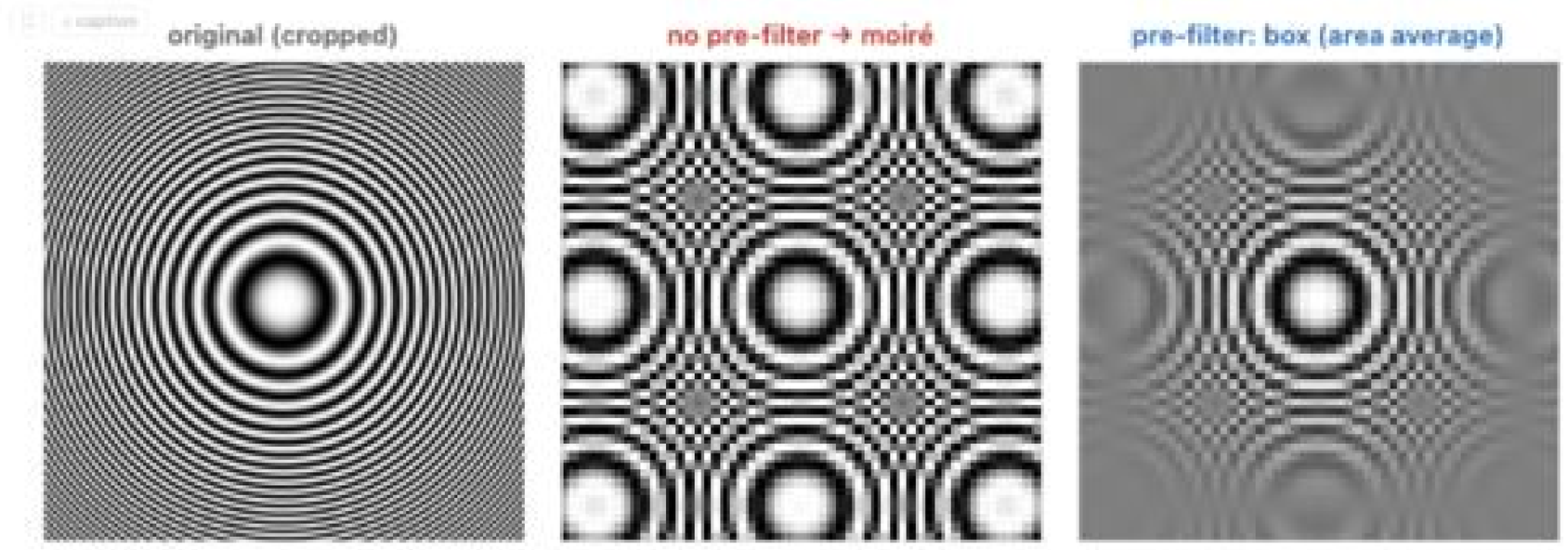
Downsampling ×4 by keeping one sample of every 4 × 4, with no pre-filter.
The panels use the same crop, fixed to the widest filter support.

◀ 1 input | 2 aliased: point sampling | 3 pre-filtered ▶

source: zone plate | downsample factor: ×4 | reset

Demo courtesy of Frédo Durand, MIT.

Demo: Figure 6.4.5, stage 3: pre-filtered



Downsampling ×4: pre-filtering reduces aliasing; residual moiré and ringing depend on the filter.
The panels use the same crop, fixed to the widest filter support.

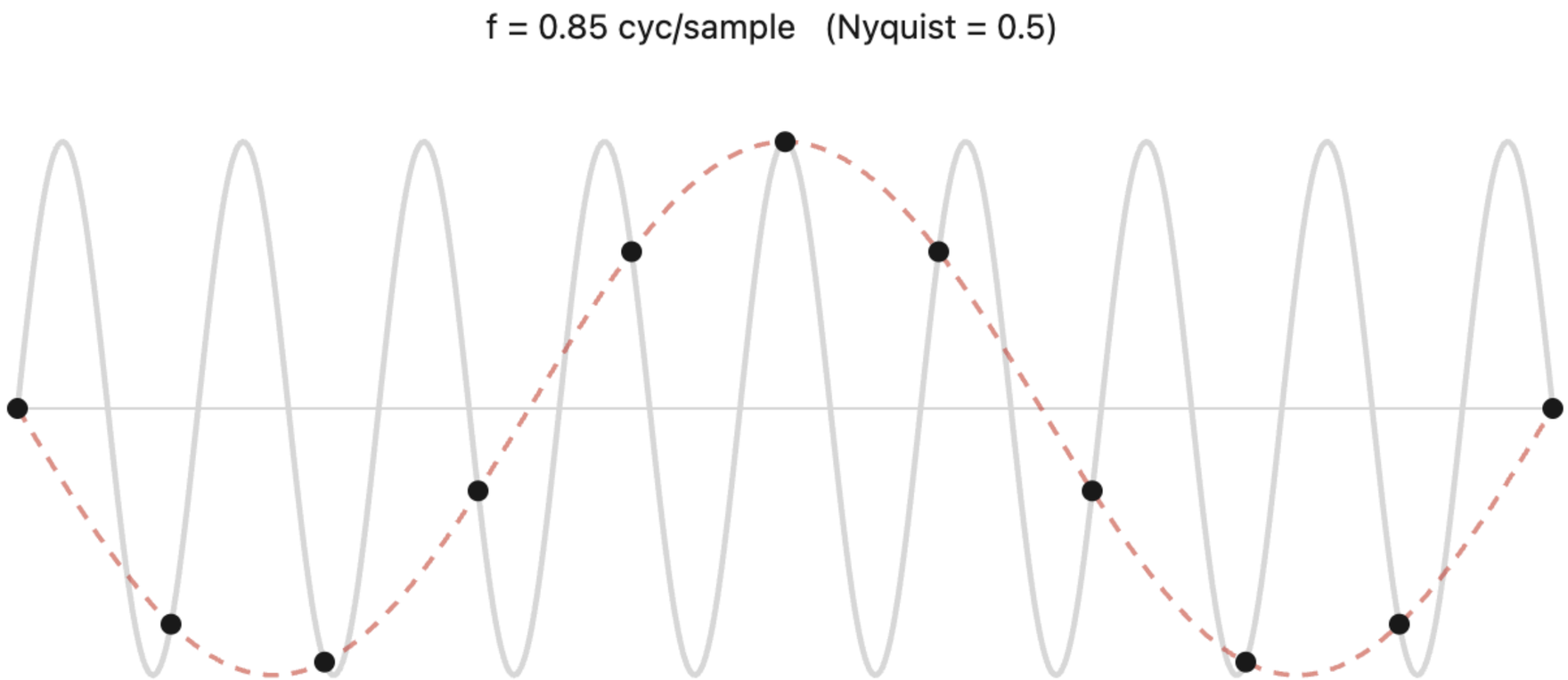
1 input | 2 aliased: point sampling | 3 pre-filtered

source: zone plate | downsample factor: ×4 | pre-filter: Box (area avg)

reset

Demo courtesy of Frédo Durand, MIT.

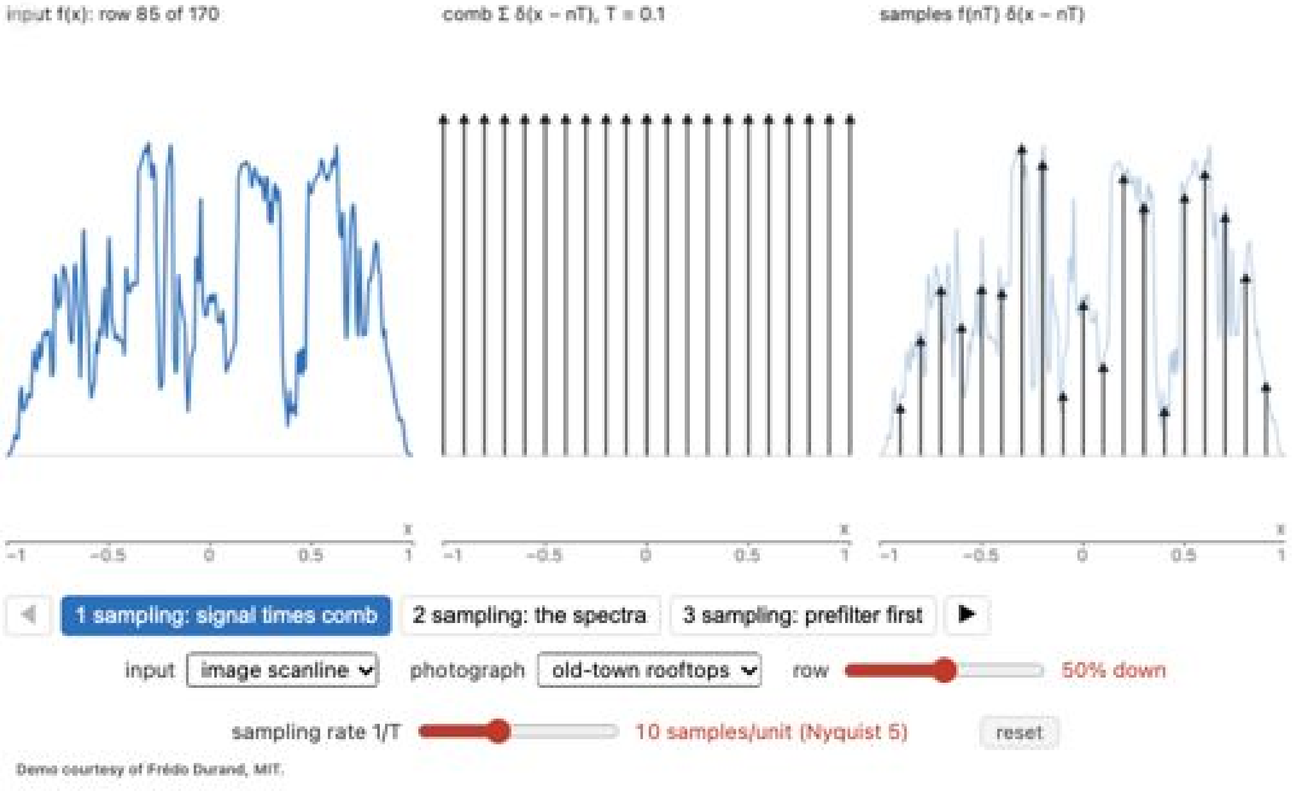
Demo: Aliasing (1-D sine)



(Down)sampling in Fourier

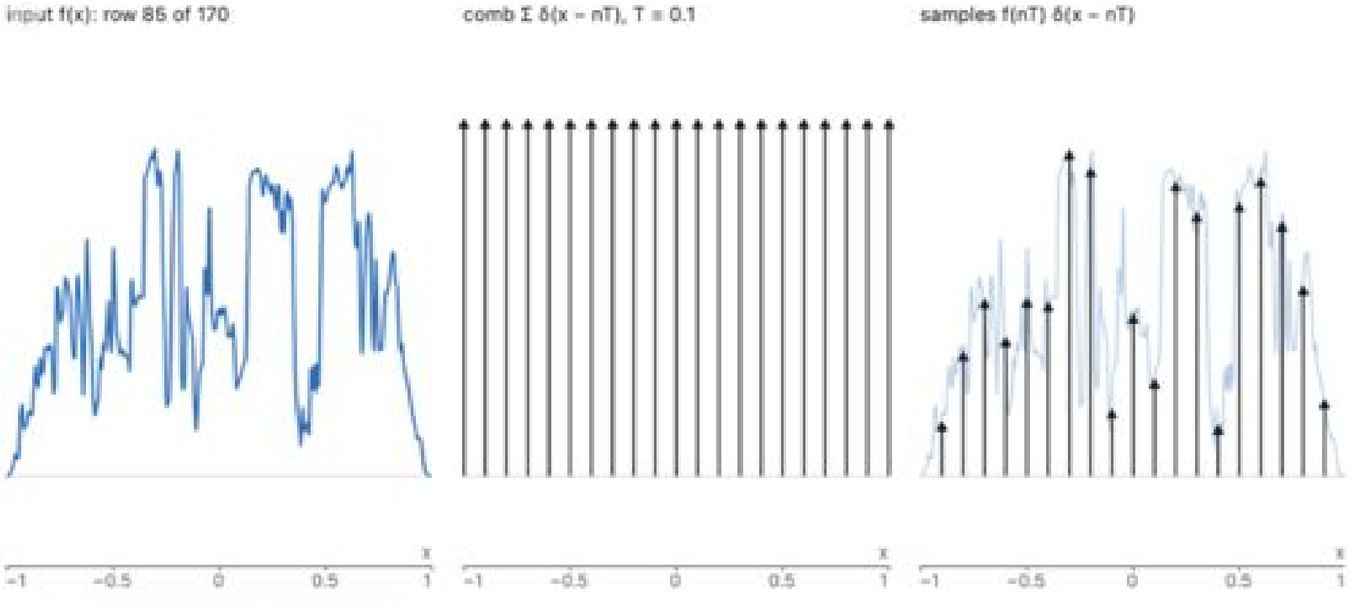
Aliasing in Fourier

- Sampling involves multiplication by a train of shifted impulses (the sampling grid)
- Well modeled in the Fourier domain
- And prefiltering (antialiasing) is about blur



Aliasing in Fourier

Sampling is a multiplication in the primal (original domain)

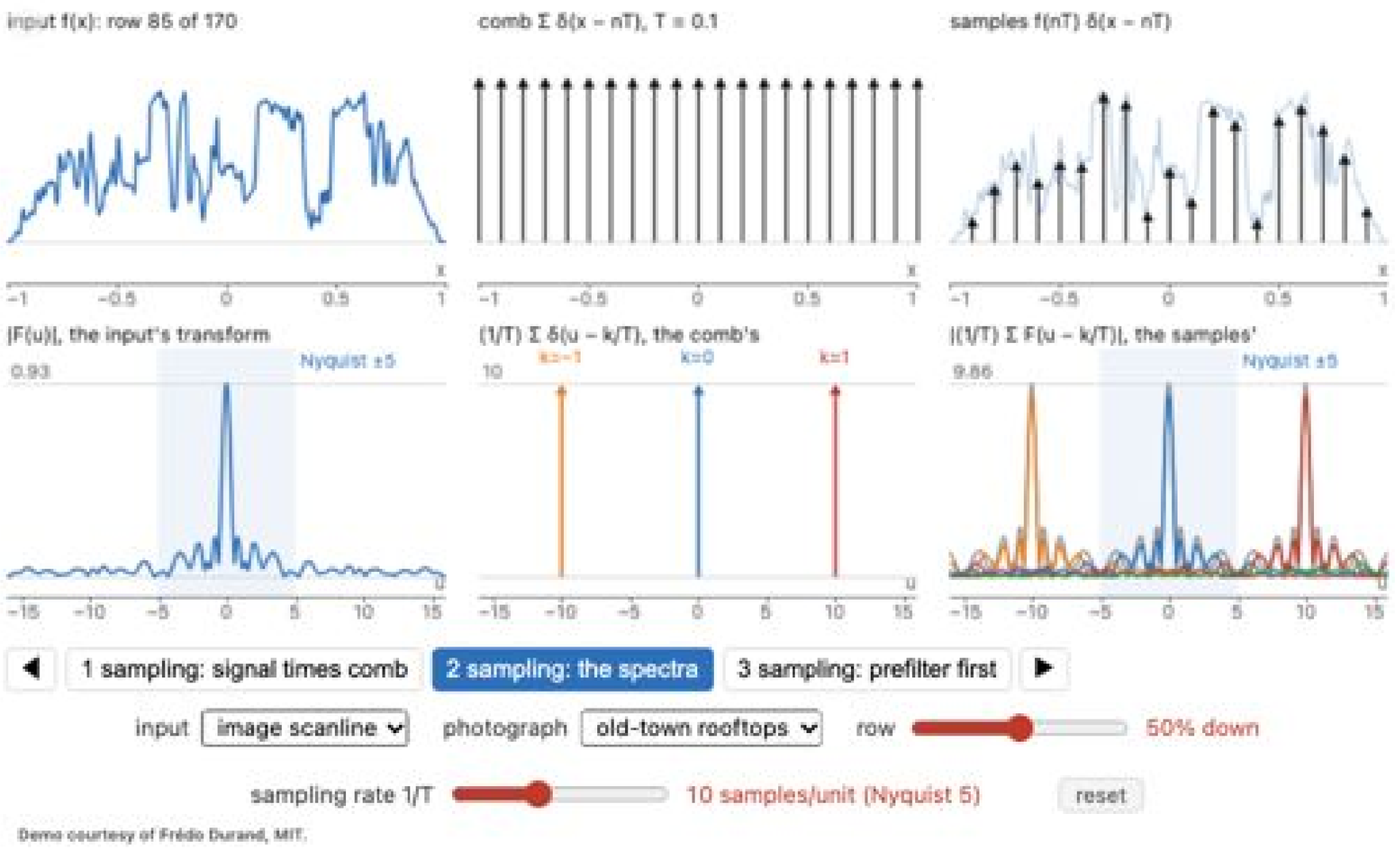


Aliasing in Fourier

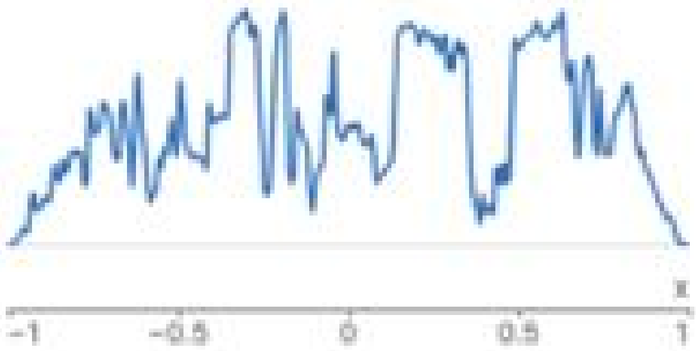
Sampling is a multiplication in the primal (original domain)

It is a convolution in Fourier.

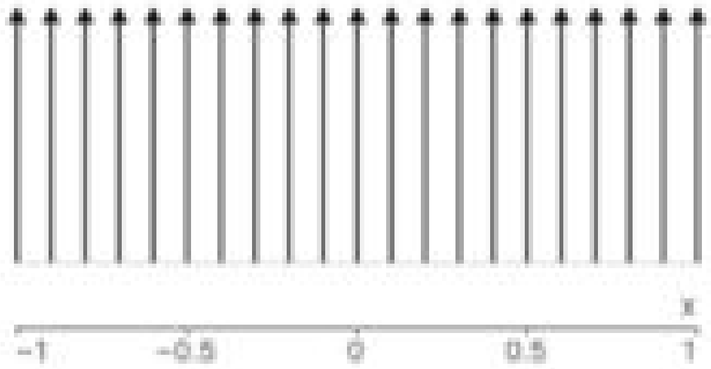
Convolution by a train of impulses creates copies (aliases)



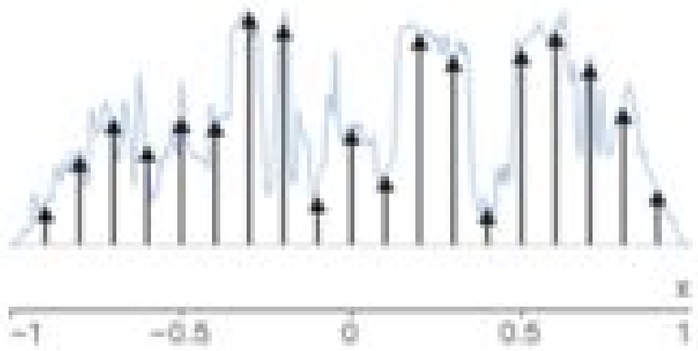
input f(x): row 85 of 170



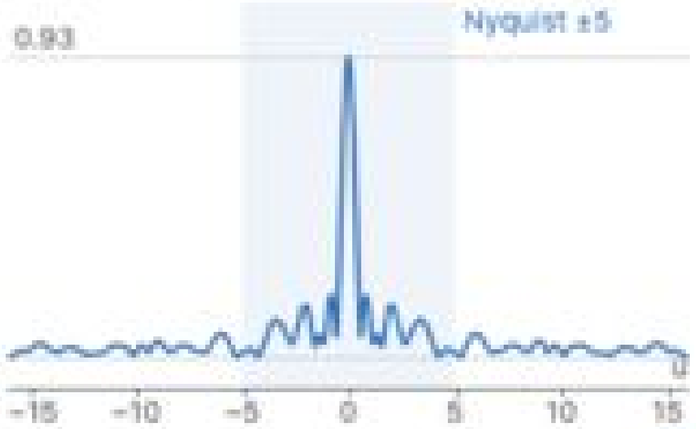
comb Σ δ(x − nT), T = 0.1



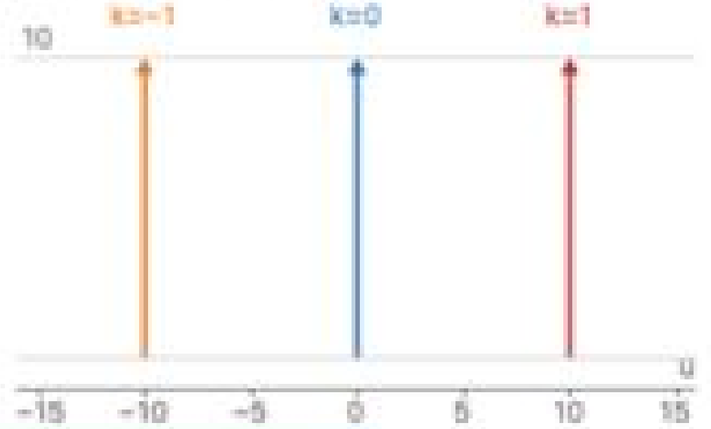
samples f(nT) δ(x − nT)



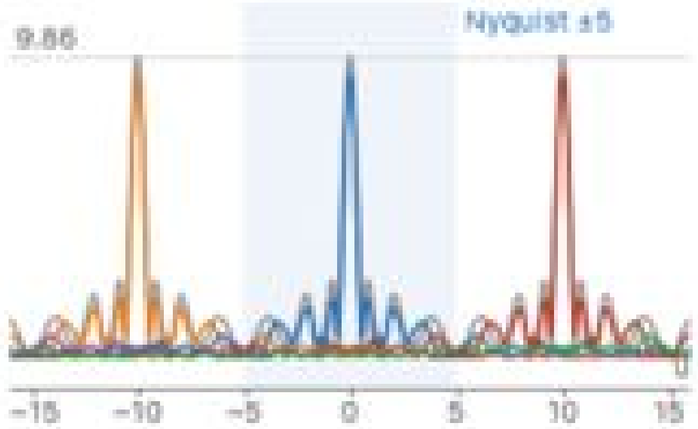
|F(u)|, the input's transform



(1/T) Σ δ(u − k/T), the comb's



|(1/T) Σ F(u − k/T)|, the samples'



◀ | 1 sampling: signal times comb | 2 sampling: the spectra | 3 sampling: prefilter first | ▶

input image scanline | photograph old-town rooftops | row 50% down

sampling rate 1/T 10 samples/unit (Nyquist 5) reset

Demo courtesy of Frédo Durand, MIT.

Sampling replicates the spectrum

- Sampling multiplies by a comb of deltas, whose Fourier transform is a comb

$$I(x)\sum_{n}\delta(x-nT) \quad \xrightarrow{\mathcal{F}} \quad \frac{1}{T}\sum_{m}\hat{I}\left(f-\frac{m}{T}\right)$$

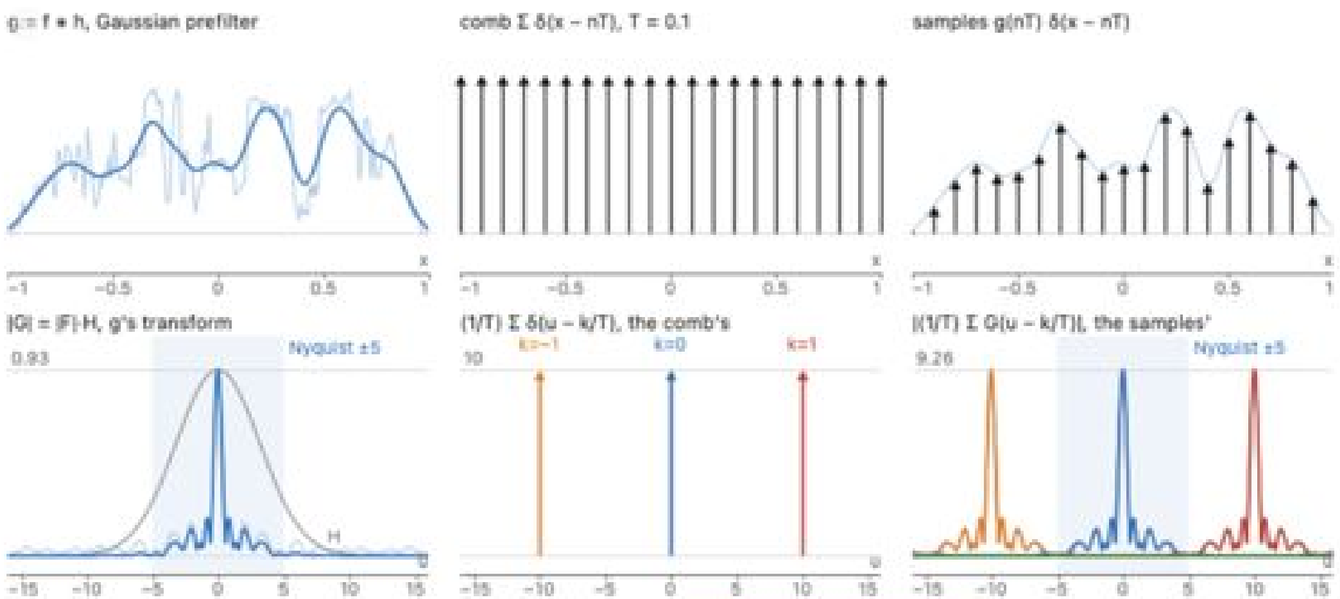
- A copy of the spectrum at every multiple of $f_s = 1/T$
- Spectrum narrower than $f_s/2$: the copies do not touch, and the original can be isolated
- Wider: the copies overlap and their tails fold onto the low frequencies. That overlap is aliasing
- Cure: low-pass first, so nothing is left above Nyquist, then sample

Book: Sampling in the frequency domain: spectral replicas

Antialiasing

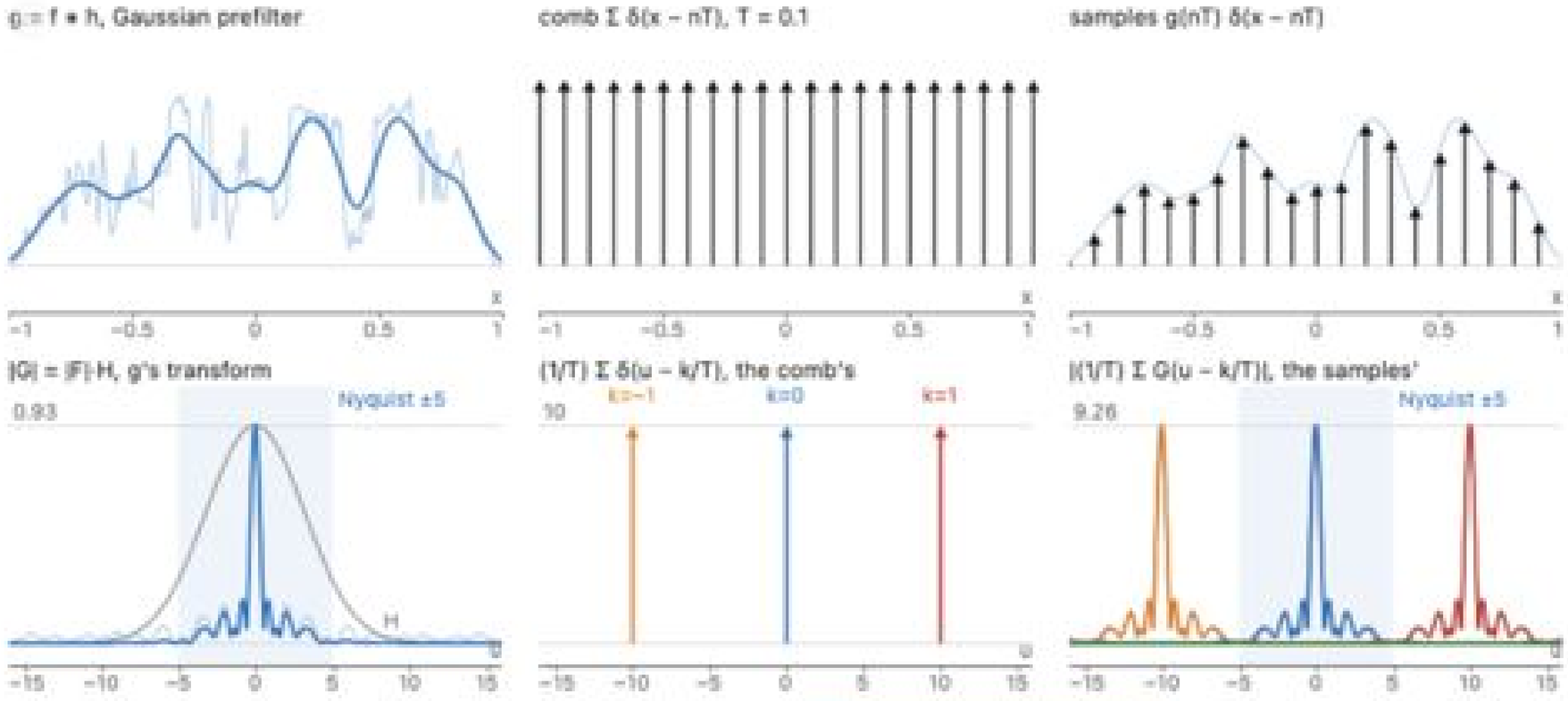
Make sure the copies (aliases) don't overlap with the spectrum that can be represented

Remove the high frequencies of the original.



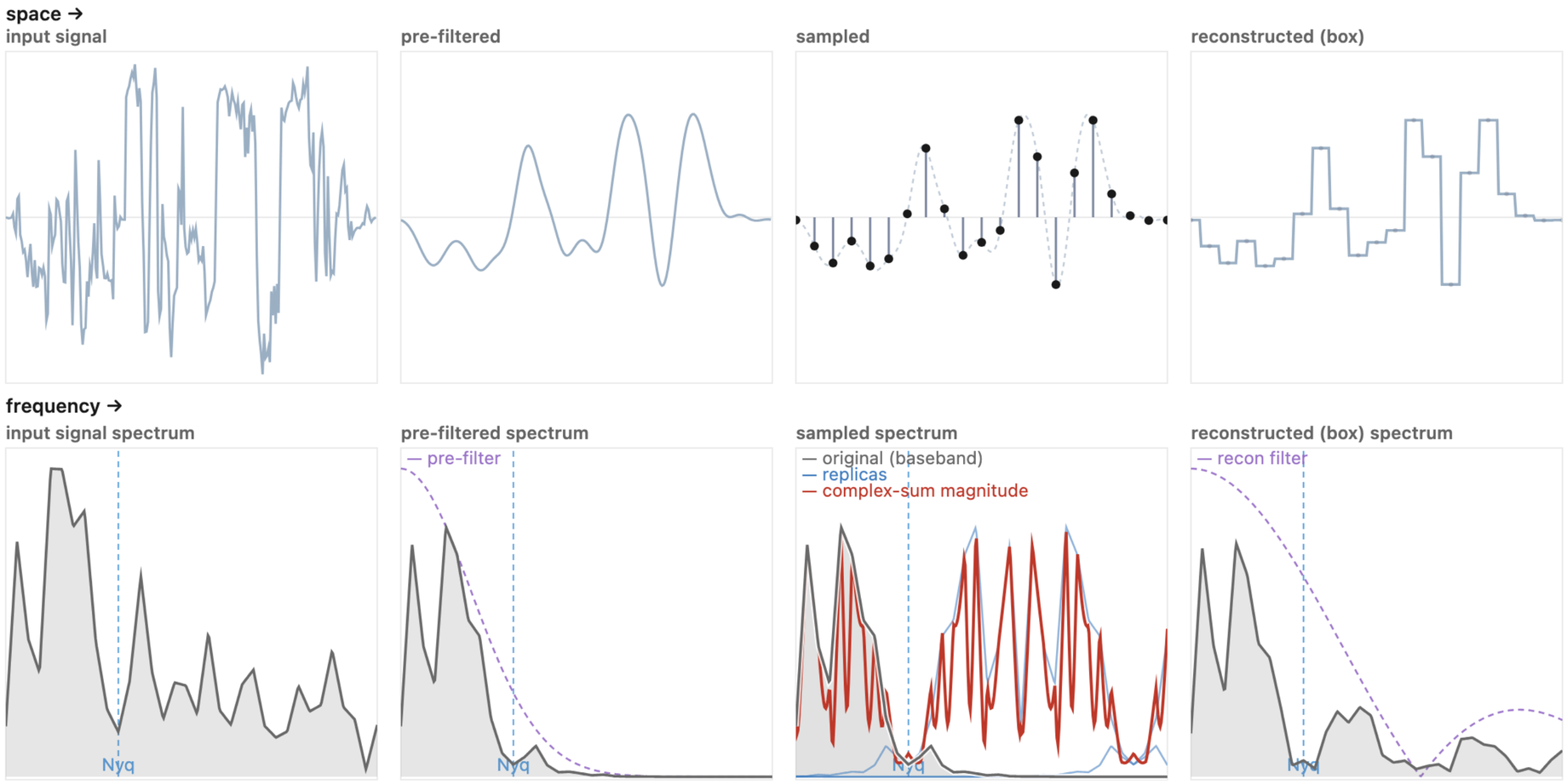
Demo courtesy of Frédo Durand, MIT.

Demo: Figure 6.4.4, stage 3: sampling, prefilter first



Demo courtesy of Frédo Durand, MIT.

Demo: 1-D sampling pipeline



replicas enter the baseband → aliasing can reinforce or cancel

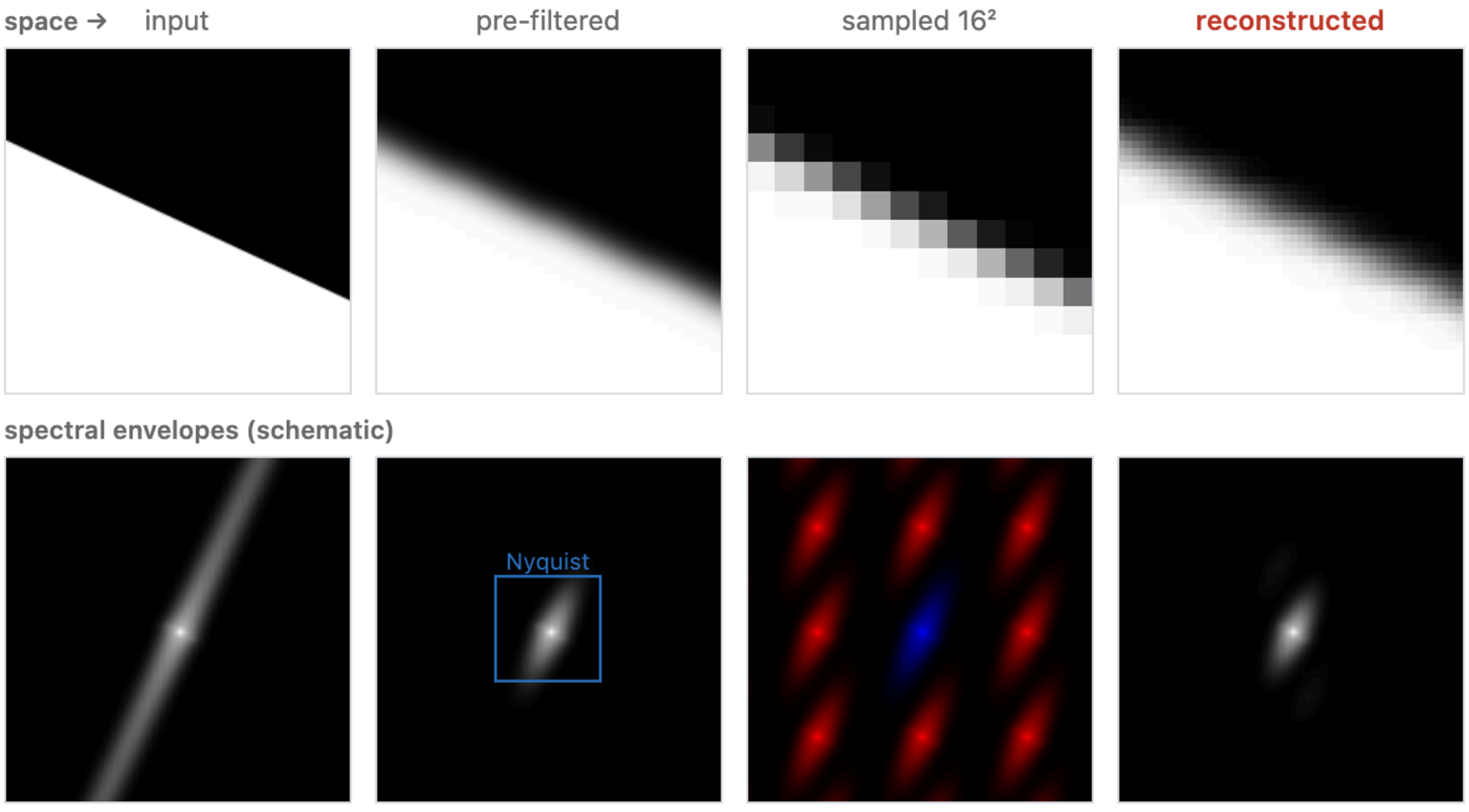
Downsampling, in frequency

- Downsampling by k divides the sampling rate by k:
  - Nyquist drops from 1/2 to 1/(2k) cycle per pixel

- Prefilter: remove what lies above the new Nyquist, then keep one sample in k
  - so the prefilter is the kernel widened k times
- Ideal prefilter: a box in frequency, a sinc in space. Practical kernels trade aliasing, blur and ringing
- In 2-D the copies sit on a lattice, and aliasing folds direction too:
  - (0.6, 0.2) cycle per pixel samples exactly like (–0.4, 0.2)

Book: In two dimensions, aliasing has a direction

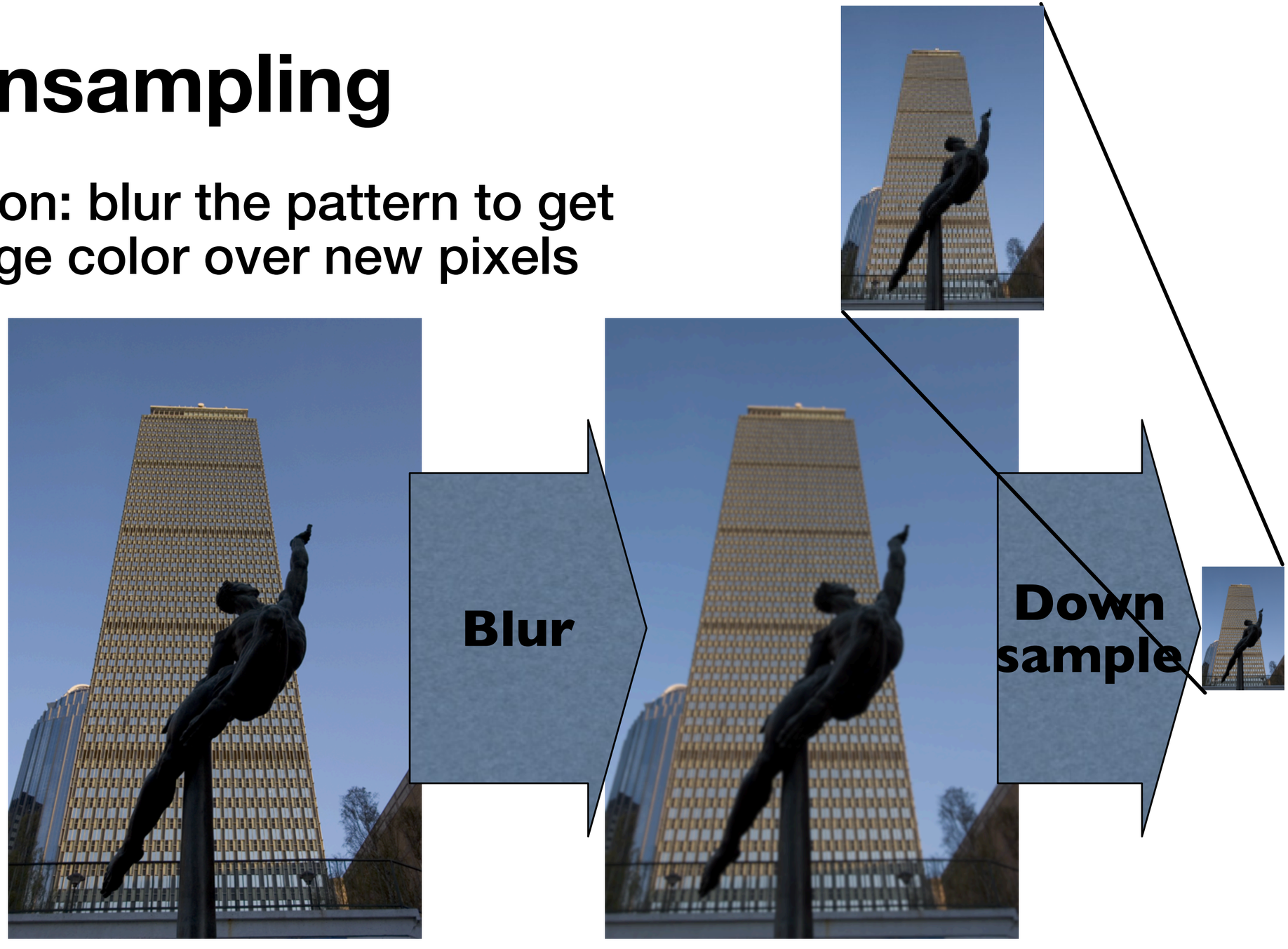
Demo: 2-D sampling pipeline



Downsampling in practice

Downsampling

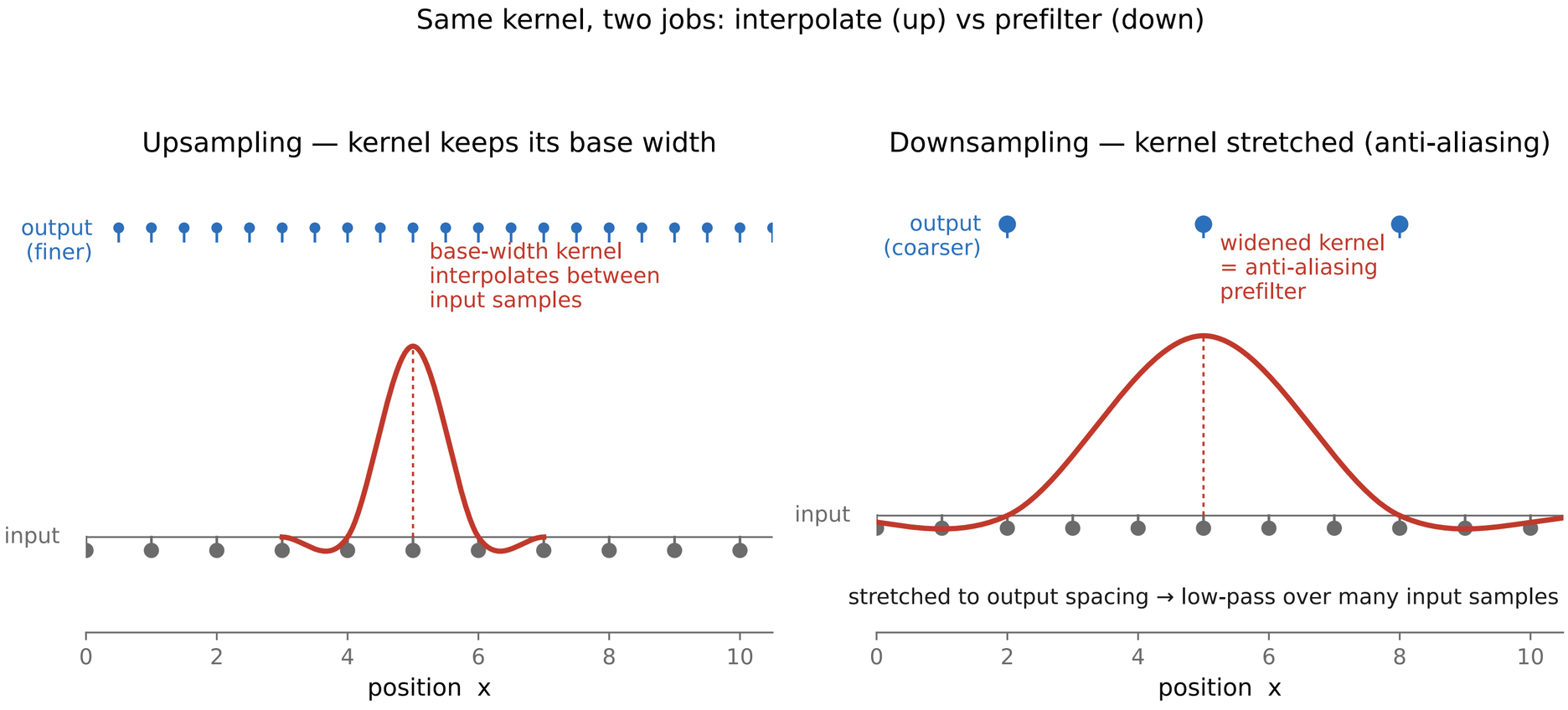
- Solution: blur the pattern to get average color over new pixels



Same code as upsampling

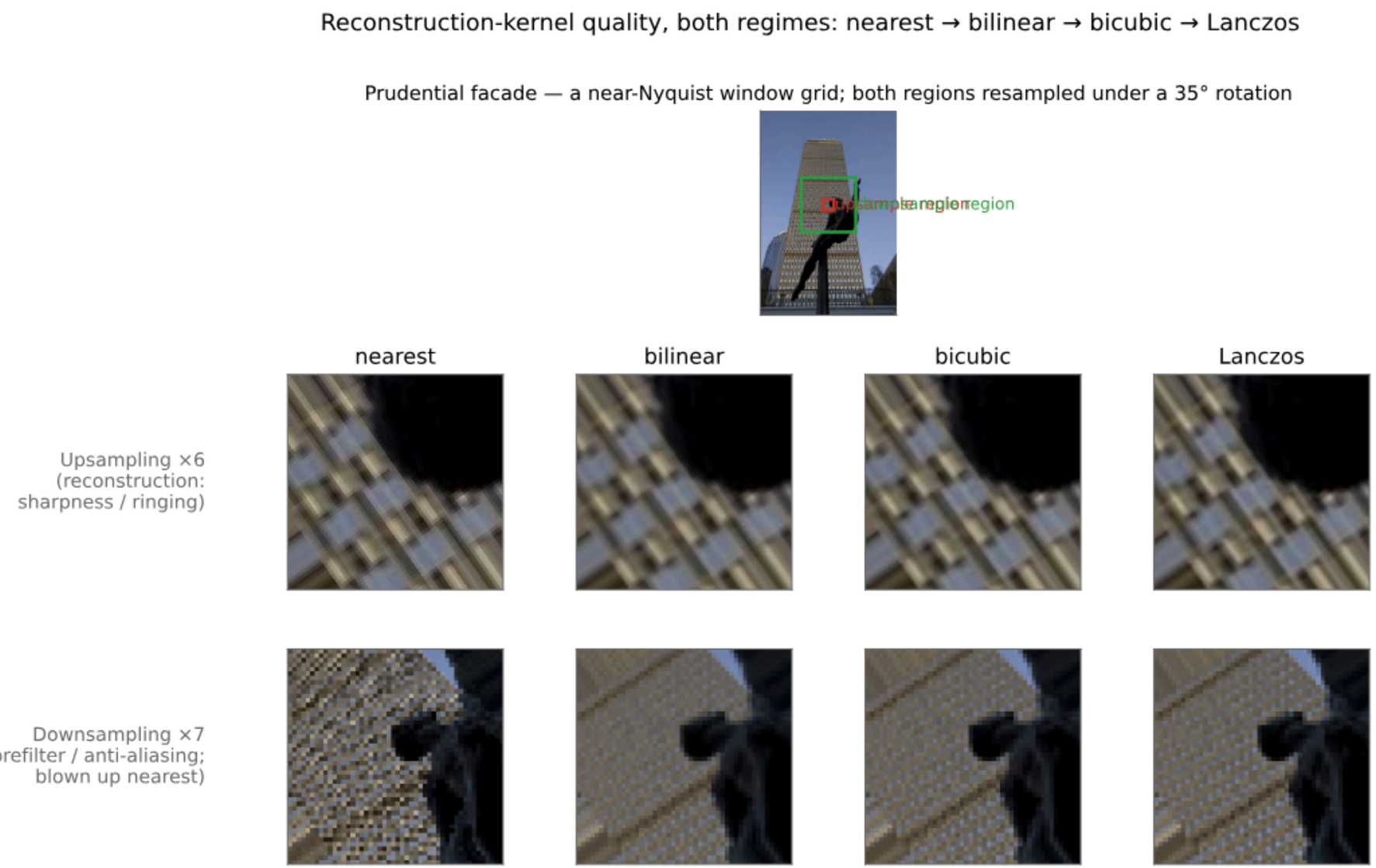
- Still a convolution
- But different scale for the kernel
-
- Always the same for upsampling
  - Proportional to scale factor for downsampling

Figure 6.5.17: The one knob that distinguishes up- from downsampling

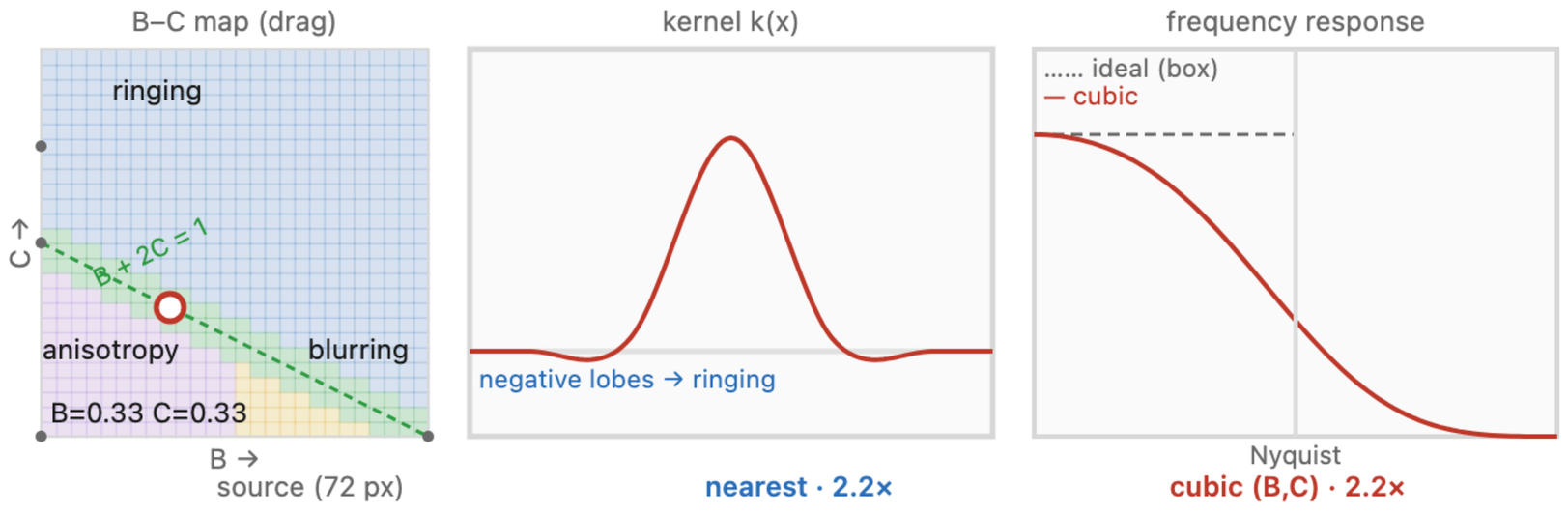


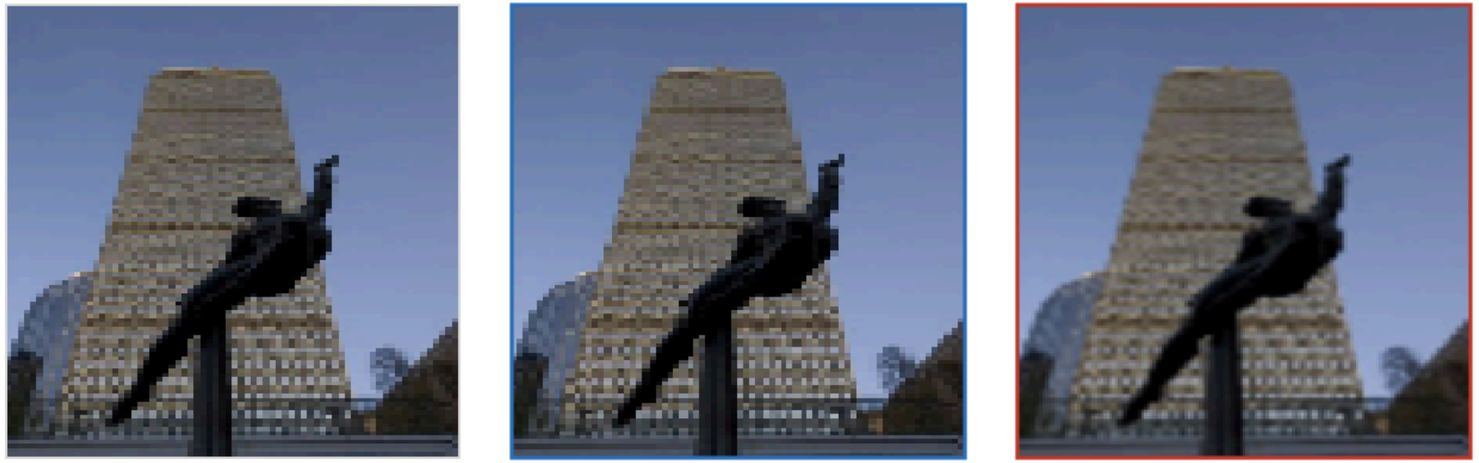
Upsampling: the kernel keeps its native width in input space (about four pixels for bicubic), reaching across the same handful of dense input samples no matter how much…

Figure 6.5.15: **The kernel ladder on a real photograph, in both regimes, resampled…**



The kernel ladder on a real photograph, in both regimes, resampled under a 35° rotation (the general case, not a clean axis-aligned zoom).





Upsampling: cubic reconstruction smooths the samples; negative lobes can ring.
Nearest uses point samples. Blur and residual aliasing depend on B and C.

In practice

- Very similar to upsampling, usually same code
- Just scale kernel up by scaling factor
  - this means we average over more neighbors
- Think how far you need to gather
- For integer scale factor, can be precomputed

scipy is doing it wrong

At least it used do. It used cubic interpolation rather than convolution

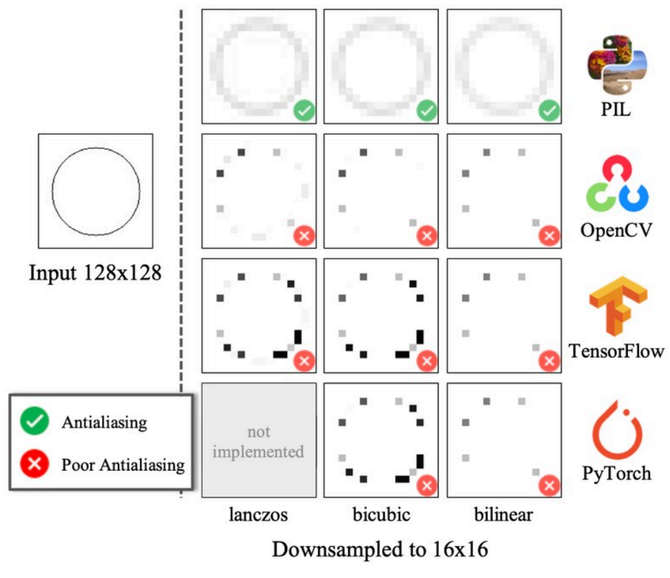
- https://arxiv.org/pdf/2104.11222.pdf

On Buggy Resizing Libraries and Surprising Subtleties in FID Calculation

Gaurav Parmar[1] Richard Zhang[2] Jun-Yan Zhu[1]
[1]Carnegie Mellon University [2]Adobe Research

Abstract

We investigate the sensitivity of the Fréchet Inception Distance (FID) score to inconsistent and often incorrect implementations across different image processing libraries. FID score is widely used to evaluate generative models, but each FID implementation uses a different low-level image processing process. Image resizing functions in commonly-used deep learning libraries often introduce aliasing artifacts. We observe that numerous subtle choices need to be made for FID calculation and a lack of consistencies in these choices can lead to vastly different FID scores. In particular, we show that the following choices are significant: (1) selecting what image resizing library to use, (2) choosing what interpolation kernel to use, (3) what encoding to use



- **https://twitter.com/AlexTamkin/status/1456149826337263621?fbclid=IwAR3_IH-3KwU3mf3eWfajZ0HiI-gIOAtCGkOxb5JkvzFUwfMCqqrg_PVdz80**

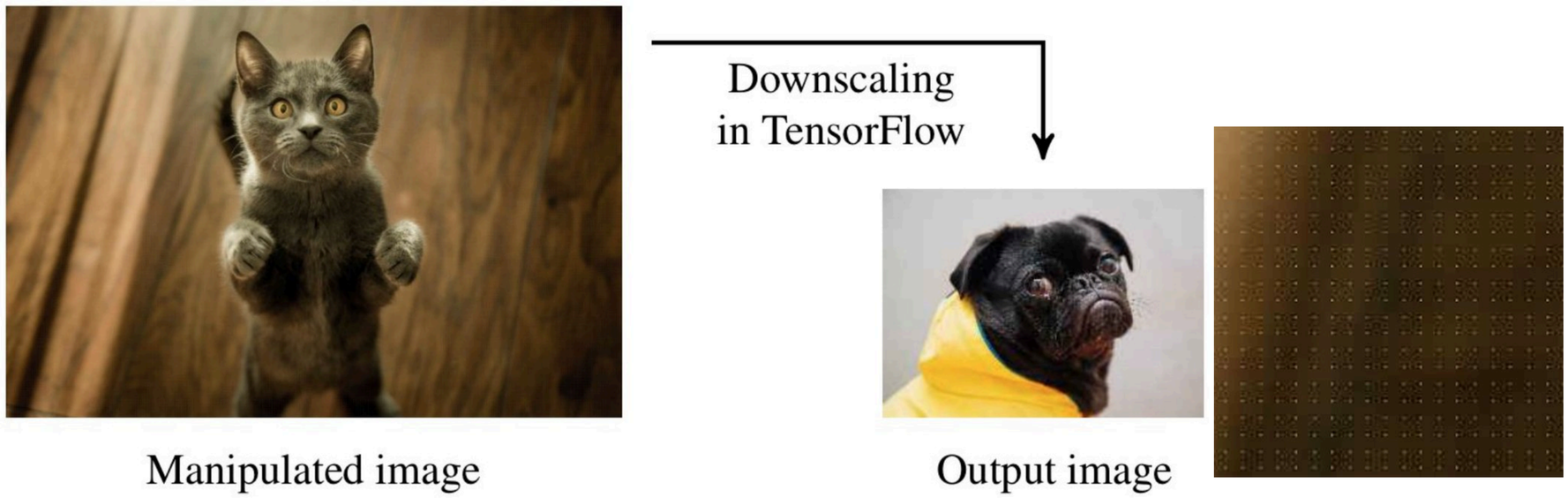
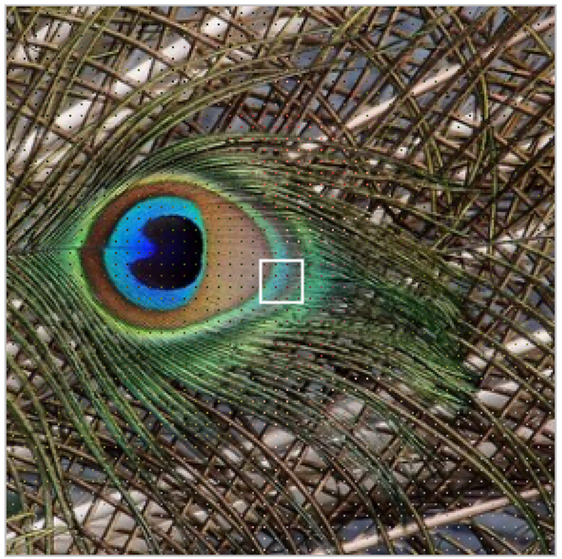


Figure 1: Example of an image-scaling attack. Left: a manipulated image showing a cat. The scaling operation produces the right image with a dog.

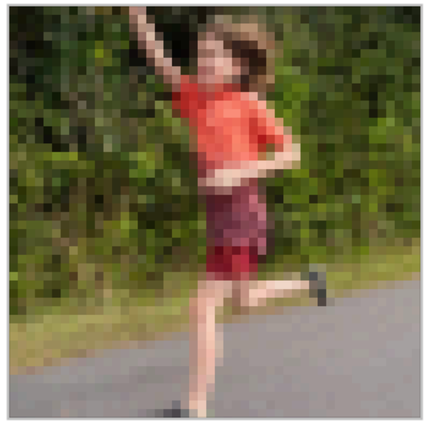
Demo: Image-scaling attack: a picture hidden in another

the manipulated image, 1:1 (288 × 288)



planted: 1 pixel in every 6 × 6 (2.8% of them)
click to move the zoom

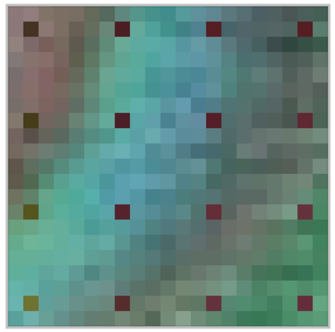
nearest neighbor
scale 1/6 → 48 × 48, enlarged



output, 1:1



zoom ×8



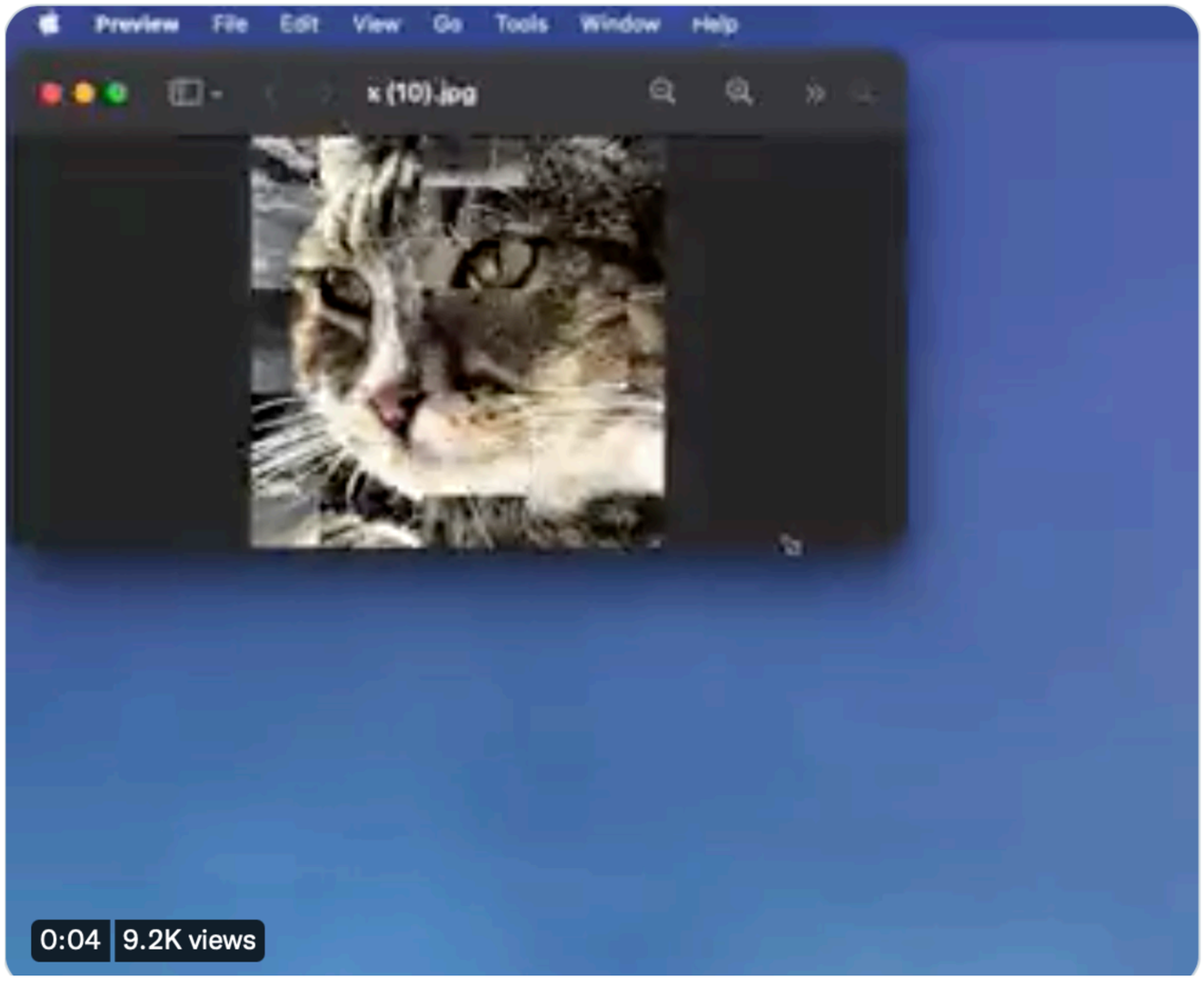
the output looks like

the hidden picture: r = 1.00

the visible picture: r = -0.21

correlation of luma, at the output's size

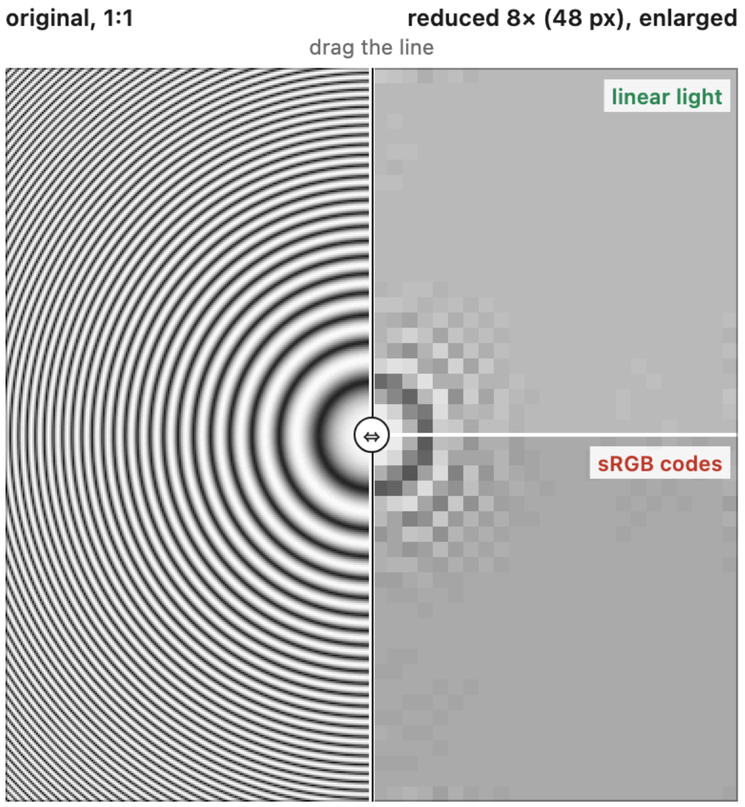
- **https://twitter.com/sigfpe/status/1457157479071199238**

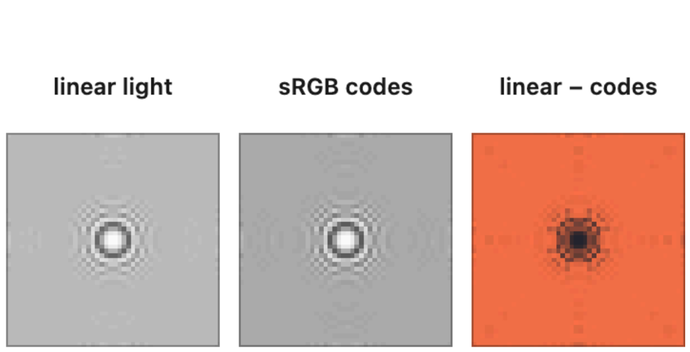


In practice

- Very similar to upsampling, usually same code
- Just scale kernel up by scaling factor
  - this means we average over more neighbors
- Think how far you need to gather
- For integer scale factor, can be precomputed
-
-
- Should be on linear values, often done incorrectly on gamma encoded ones
  - changes local brightness

Demo: Downsample on light or on code





difference: warm where the codes came out darker,
cool where brighter; normalized to its largest value

mean light, as a code value:

original: 188
linear light: 188
sRGB codes: 171

per pixel, codes darker by 16.4 on average, 20 at worst

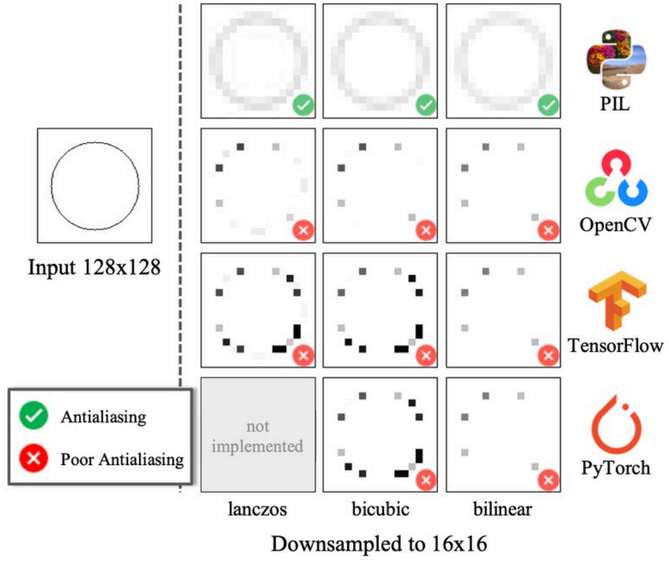
- **https://arxiv.org/pdf/2104.11222.pdf**

On Buggy Resizing Libraries and Surprising Subtleties in FID Calculation

Gaurav Parmar[1] Richard Zhang[2] Jun-Yan Zhu[1]
[1]Carnegie Mellon University [2]Adobe Research

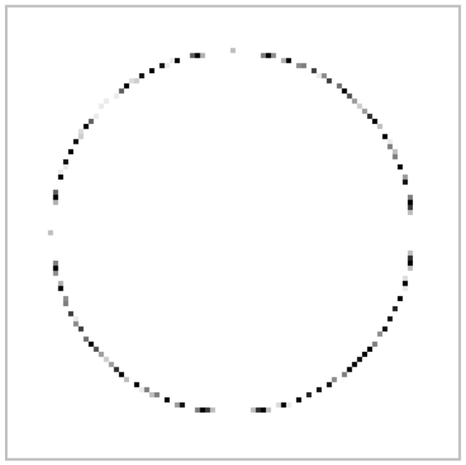
Abstract

We investigate the sensitivity of the Fréchet Inception Distance (FID) score to inconsistent and often incorrect implementations across different image processing libraries. FID score is widely used to evaluate generative models, but each FID implementation uses a different low-level image processing process. Image resizing functions in commonly-used deep learning libraries often introduce aliasing artifacts. We observe that numerous subtle choices need to be made for FID calculation and a lack of consistencies in these choices can lead to vastly different FID scores. In particular, we show that the following choices are significant: (1) selecting what image resizing library to use, (2) choosing what interpolation kernel to use, (3) what encoding to use

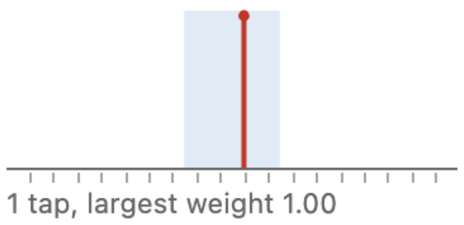


Demo: Four ways to shrink an image: only the widened kernel avoids aliasing

decimation
1 source pixel each

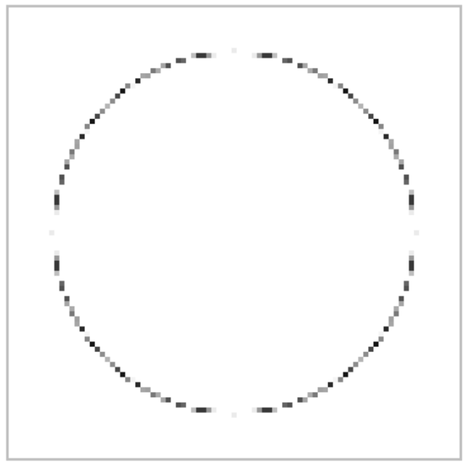


aliasing 16.3 RMS (of 255)
98% of energy above Nyquist

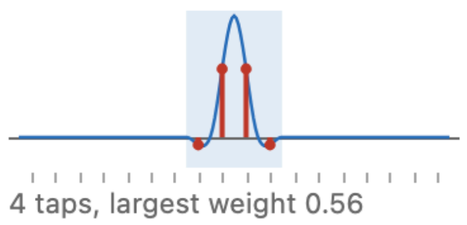


1 tap, largest weight 1.00

fixed-width kernel
bicubic, ±2 source px

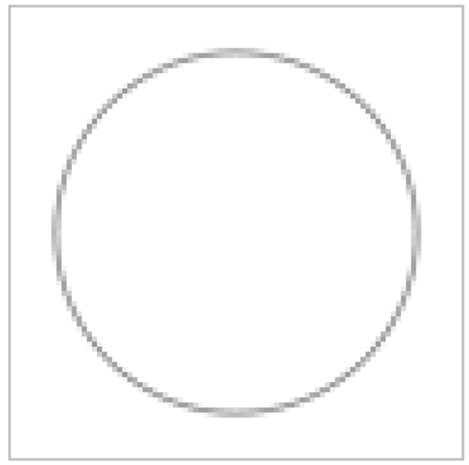


aliasing 13.6 RMS (of 255)
68% of energy above Nyquist

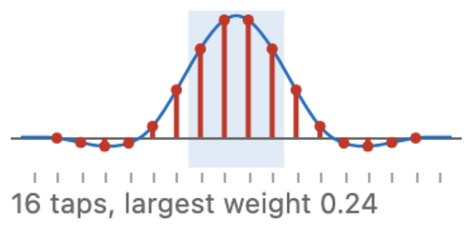


4 taps, largest weight 0.56

kernel widened ×4
bicubic, ±8 source px

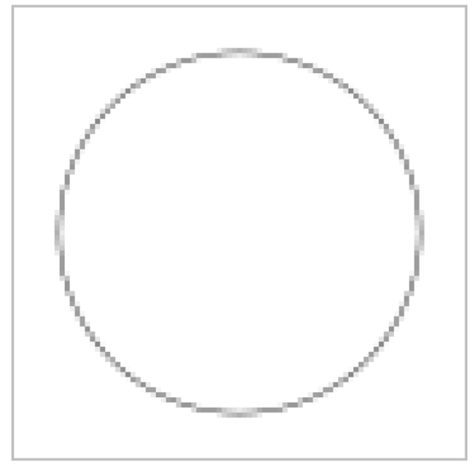


aliasing 2.5 RMS (of 255)
2.3% of energy above Nyquist

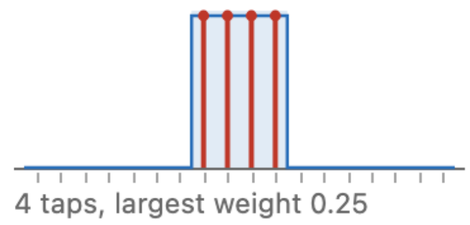


16 taps, largest weight 0.24

area average
box, 4 source px wide



aliasing 4.8 RMS (of 255)
8.6% of energy above Nyquist



4 taps, largest weight 0.25

More info

- http://web.stanford.edu/class/cs448f/lectures/1.2/Sampling.pdf
- https://en.wikipedia.org/wiki/Bicubic_interpolation
- http://entropymine.com/imageworsener/bicubic/
- http://entropymine.com/imageworsener/resample/
- separable!
- upsampling is same, just clamp kernel scale factor to 1.0
- downsampling not super easy to debug

Advice by Andrew Adams

I don't know of a good general reference, but I definitely have opinions on the right way to implement downsampling.

https://github.com/halide/Halide/blob/master/apps/resize/resize_generator.cpp

Downsampling is linear and separable, so the general idea is to explicitly compute the sparse matrix that downsamples a single row/col, then apply it to each row/col. The size of the sparse matrix is tiny relative to the image, so computing the weights and handling boundary conditions becomes approximately free.

In more detail to downsample in x:

1) For each x coordinate in the output, translate and scale the continuous downsampling kernel appropriately to have the right footprint on the input.

2) Sample that translated, scaled, continuous kernel at all integer locations on the x axis of the input space where it has non-zero values. You can compute closed-form expressions for the index of the first non-zero value and the maximum number of non-zero values, so it's easy to represent the sparse matrix as a simple max_kernel_taps x input_width table.

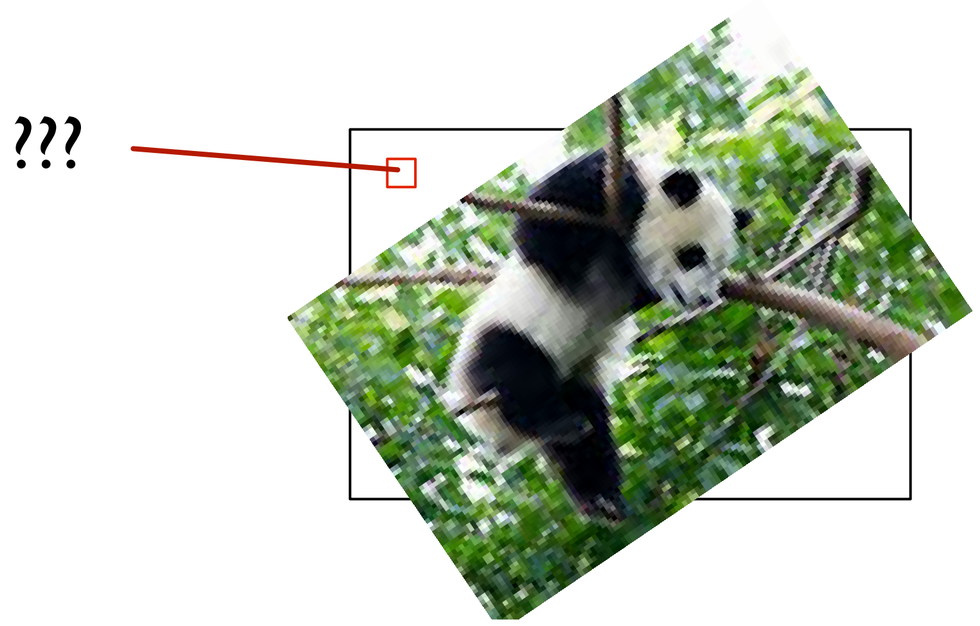
3) Renormalize each row of the matrix to sum to 1. I think you can avoid this if you're careful with the kernel, but this is also convenient way to handle boundary conditions if you want to just truncate the kernel at the image edges.

4) Optionally quantize the matrix and store it as integers that sum to a power of 2. Do this if your image is low precision and the machine you're on has high throughput for integer multiply-adds (e.g. a DSP). The example code doesn't do this. Forcing it to sum to a power of 2 is more complicated that just quantizing scaled values - you need error diffusion.

Padding

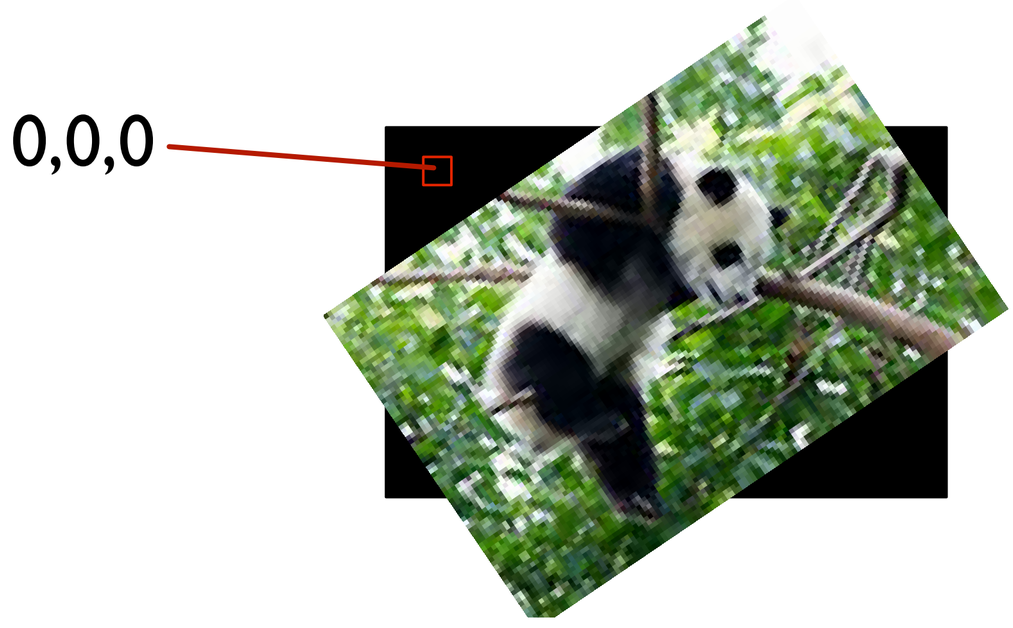
Padding problems

- **Sometimes, we try to read outside the image**
  - e.g. x or y is negative
  - For example, we try to rotate an image



Black Padding

```
getBlackPadded(im, x, y):
    if (x<0) or (x>=im.width()) or (y<0)
        or (y>=im.height()):
        return black
    else: return im(x, y)
```



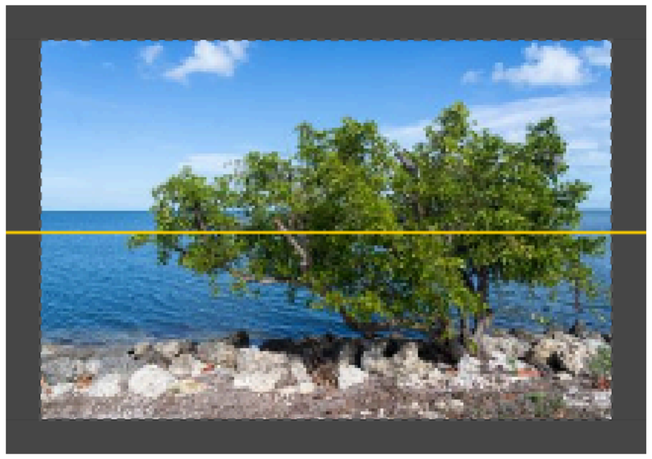
Edge Padding

```
clipX(im, x): return min(im.width()-1, max(x, 0))
clipY(im, y): return min(im.height()-1, max(y, 0))
getSafePix(im, x, y):
    return im(clipX(im, x), clipY(im, y))
```

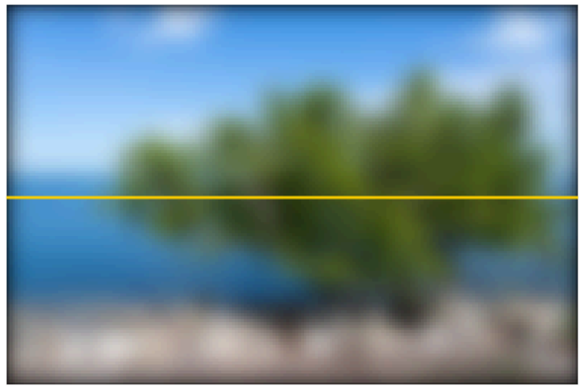


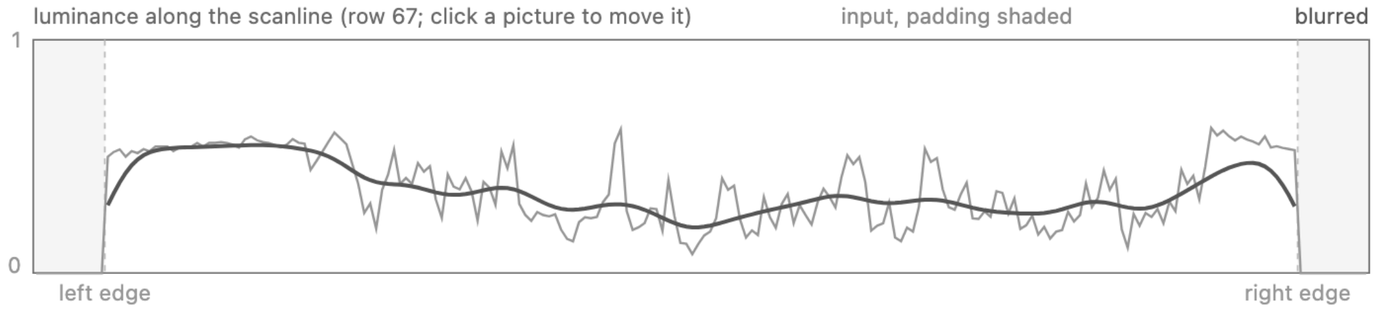
Demo: What the four boundary rules do to a photograph

input, with the padding a 12-pixel kernel reads (zero)



blurred





Questions?

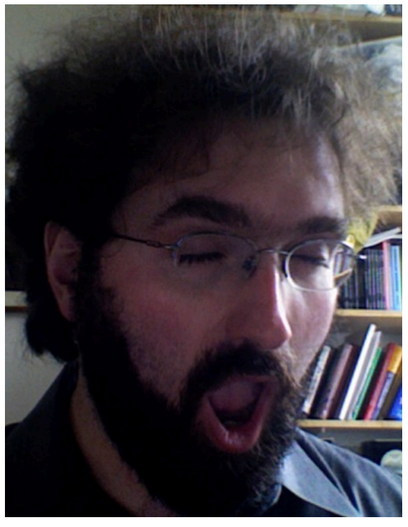


Warping & Morphing

Important scientific question

- **How to turn Dr. Jekyll into Mr. Hyde?**
- **How to turn a man into a werewolf?**

- **Powerpoint cross-fading?**



Important scientific question

- **How to turn Dr. Jekyll into Mr. Hyde?**
- **How to turn a man into a werewolf?**

- **Powerpoint cross-fading?**



From An American Werewolf in London

- **or**
- **Image Warping and Morphing?**

Digression: old metamorphoses

- http://en.wikipedia.org/wiki/The_Strange_Case_of_Dr._Jekyll_and_Mr._Hyde
- http://www.eatmybrains.com/showtopten.php?id=15
- http://www.horror-wood.com/next_gen_jekyll.htm
- **Unless I'm mistaken, both employ the trick of making already-applied makeup turn visible via changes in the color of the lighting, something that works only in black-and-white cinematography. It's an interesting alternative to the more familiar Wolf Man time-lapse dissolves. This technique was used to great effect on Fredric March in Rouben Mamoulian's 1932 film of *Dr. Jekyll and Mr. Hyde*, although Spencer Tracy eschewed extreme makeup for his 1941 portrayal.**



Dr. Jekyll and Mr. Hyde, 1932



Dr. Jekyll and Mr. Hyde, 1932



Dr. Jekyll and Mr. Hyde, 1932



Dr. Jekyll and Mr. Hyde, 1941



- **https://filmschoolrejects.com/sh-the-octopus-transformation/**
- **https://x.com/rainmaker1973/status/1983064162583368154?s=46&t=aUZMRTg07VYQGqu7K-19Lw**
- **https://filmschoolrejects.com/sh-the-octopus-transformation/**

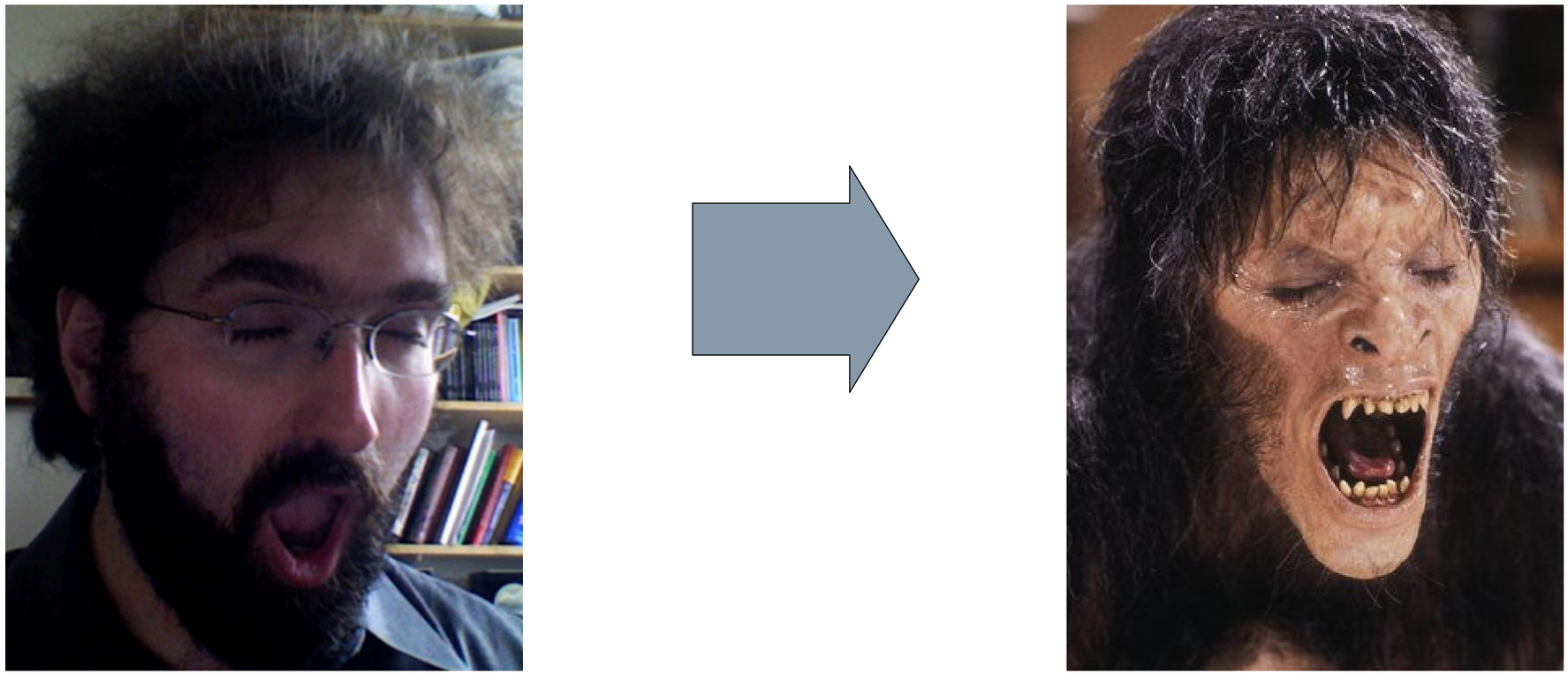


Video: Willow (1988, ILM): the first digital morph



Challenge

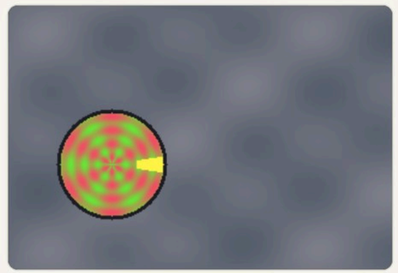
- **“Smoothly” transform a face into another**
- **Related: slow motion interpolation**
  interpolate between key frames



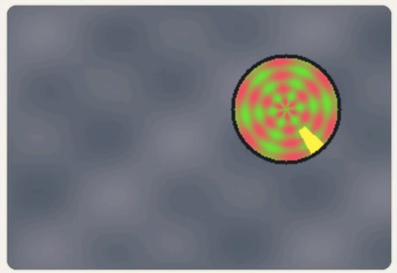
Demo: Frame interpolation

Frame interpolation: cross-dissolve vs. flow-warp

Two adjacent video frames I_0 and I_1 hold a subject in two places. To synthesize the in-between frame at time $t \in [0,1]$ you can either **cross-dissolve** the fixed pixels — which leaves a translucent **ghost / double image** — or **warp the pixels along the motion** (the optical flow) to time t and then blend, giving **one sharp subject** at the correct in-between position. Drag **t** or press **Play** and watch the difference. This is a **synthetic scene with known motion**, so the flow is exact and the warp-to-t is exact — the lesson, not an estimator, is on display.



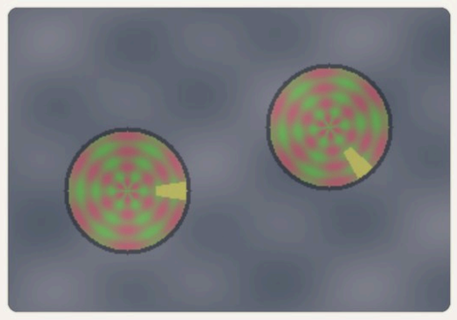
I_0 — frame at t=0



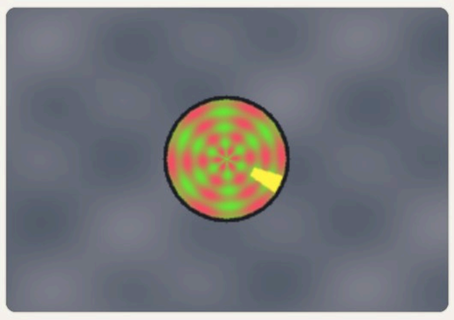
I_1 — frame at t=1

▶ Play t = **0.50** ☐ show flow field

☑ occlusion-aware blend



Naive cross-dissolve
$(1-t)\cdot I_0 + t\cdot I_1$
ghost energy **10.5**



Flow-based interpolation
warp I_0, I_1 to t along the flow, then blend
ghost energy **0.3**

A cross-dissolve blends colors at *fixed pixels*, so a moving subject appears in both places at once — a translucent

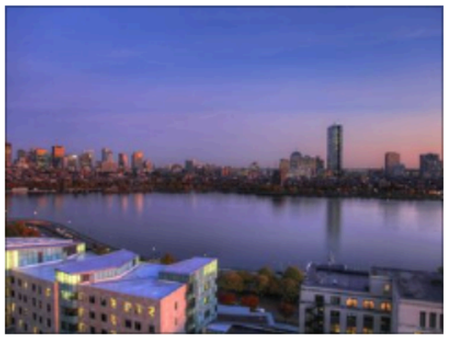
Averaging images

- **Cross-fading**
  output(x, y) = (1-t) * im1(x, y) + t * im2(x, y)



Demo: Cross-fade in three spaces

A



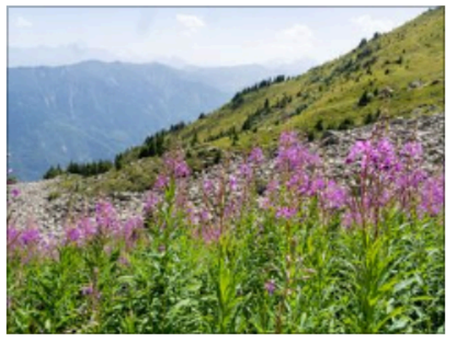
Boston sunset

cross-fade, t = 0.50



averaged in the sRGB code values

B



Alpine meadow

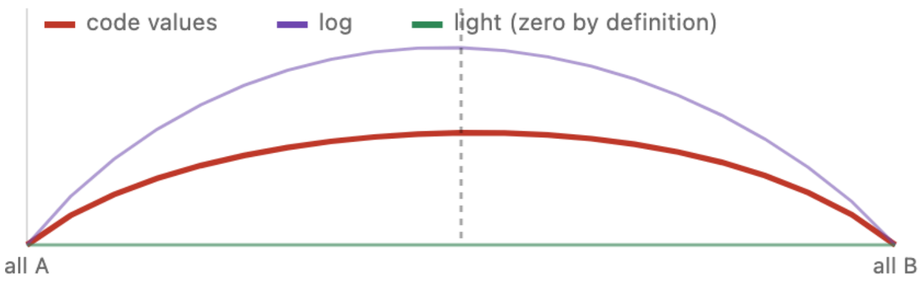
chrominance error



against the same fade done in light

0 — 0.06 and above, in Oklab a,b

mean chrominance error along the fade



at this point in the fade

chrominance error mean 0.0156 worst pixel 0.0819

lightness error mean 0.0223 (the harmless half)

Both are Oklab distances. 0.02 is comfortably visible side by side.
Try 'same photo in both' with the two exposures apart.

Demo: Why gamma fading invents colors

interpolated in LINEAR light
the mixture of the two lights



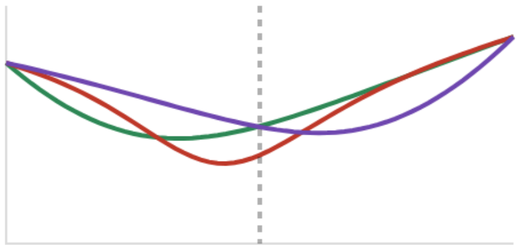
interpolated in GAMMA (code values)
what most software does



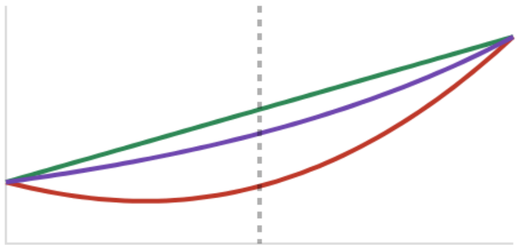
interpolated in LOG (cone log, Chong's space)
the geometric mean, cone by cone



chroma along the fade Oklab C

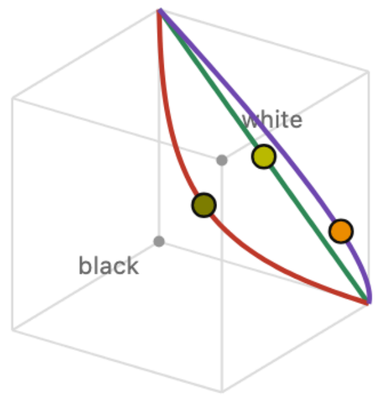


luminance along the fade relative Y



at this point the gamma fade is 25% less colourful, and 0.041 away in Oklab a,b
it is also 57% dimmer, which is the half that looks like an exposure error

the three paths through the sRGB cube



gamma bows toward black, log the other way



the two ends, and the three middles at this position

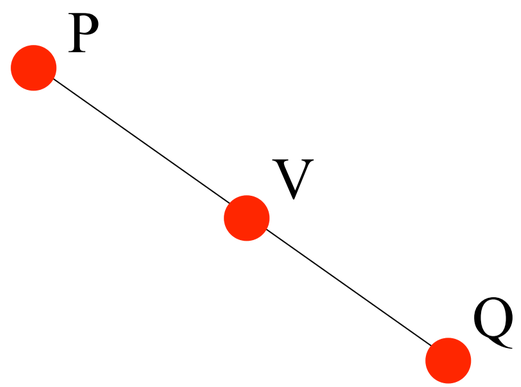
Problem with cross fading

- **Features (eyes, mouth, etc) are not aligned**
- **It is probably not possible to get a global alignment**
- **We need to interpolate the LOCATION of features**
  - domain transform!



Averaging points (location)

- **P & Q are two 2D points (in the “domain”)**
- **V= (1-t) P + t Q**

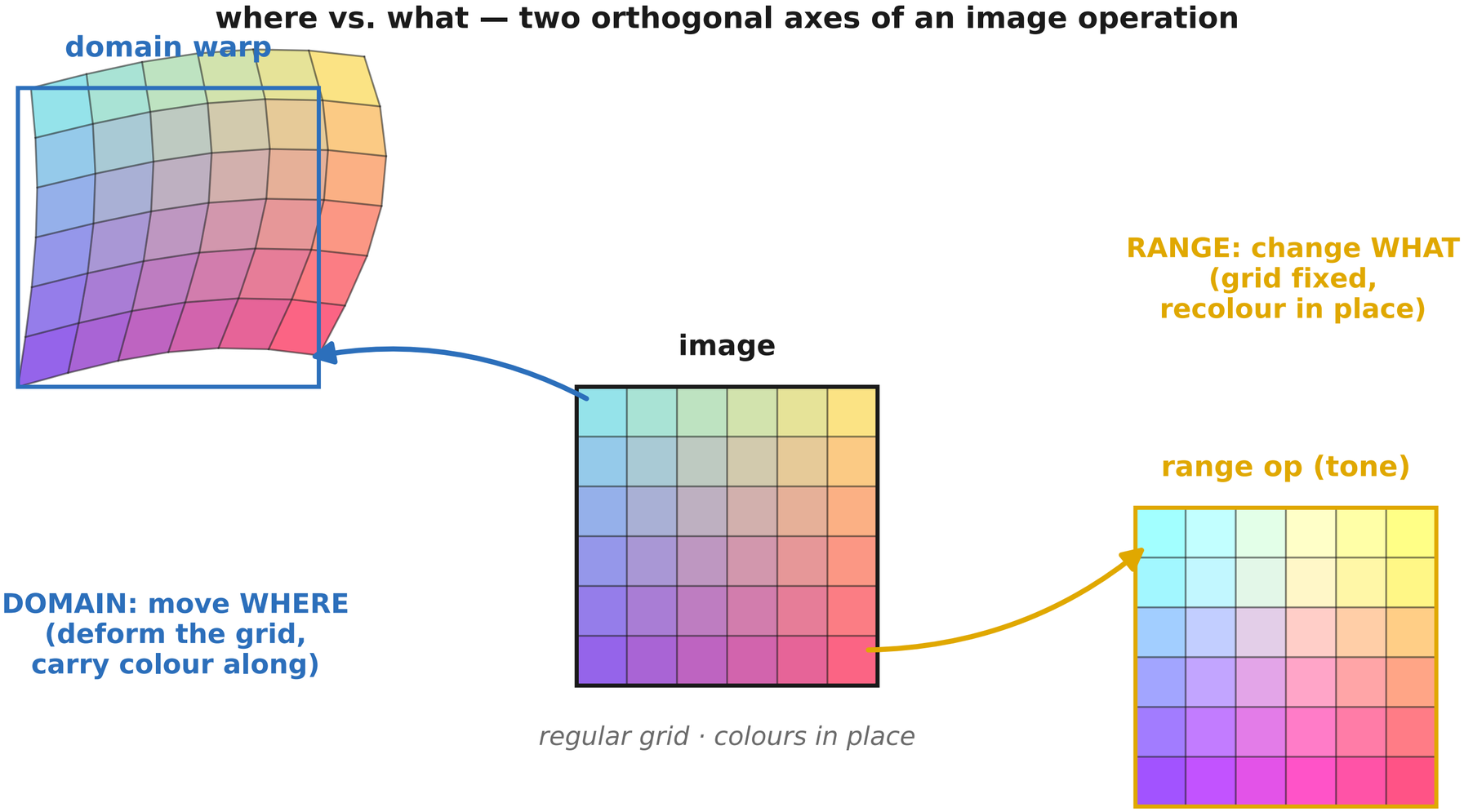


Warping

- **Move pixel spatially: C’(x,y)=C(f(x,y))**
- **Leave colors unchanged**



Figure 11.1.1: Domain vs. range



One image, two arrows.

Warping

- **Deform the domain of images (not range)**
- **Central to morphing**
- **Also useful for**
  - Optical aberration correction
  - Video stabilization
  - Slimming people down
  - ...



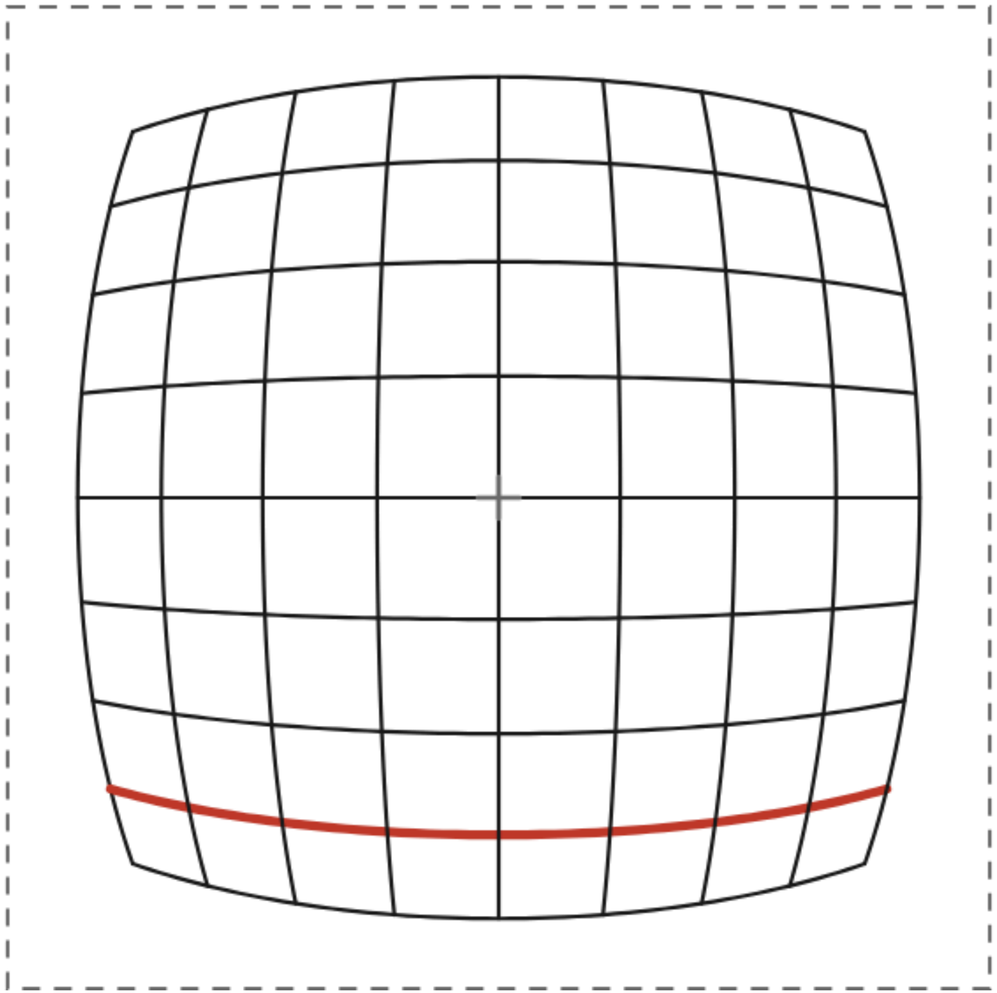
original image



DxO Optics Pro correction

Demo: Radial distortion correction

barrel distortion



Recap & questions

- **Color (range) interpolation:**
  output(x, y) = (1-t) * im1(x, y) + t * im2(x, y)
- **Location (domain) interpolation: V= (1-t) P + t Q**
- **Warping: domain transform out(x,y)=im(f^{-1}(x,y))**

Demo: EWA / anisotropic footprint

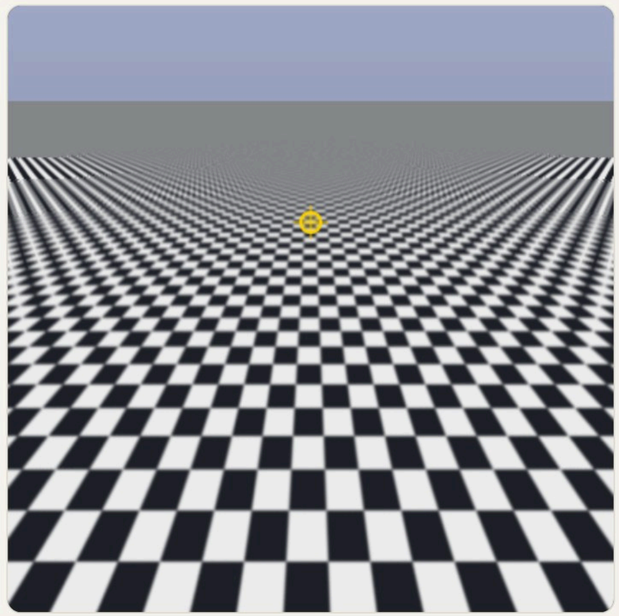
Anisotropic footprint & EWA filtering

A textured plane is warped into a **receding floor** under strong perspective. Toward the horizon one output pixel covers **many tiles in depth but only one across** — its back-projection into the texture is a long, thin **ellipse**, read straight off the local **Jacobian** $J=\partial f^{-1}/\partial(x,y)$ of the inverse warp. Flip between resampling filters and watch the far field: **point** and **bilinear** sampling **alias** into crawling moiré; an **isotropic MIP** filter must either alias (sized to the short axis) or **over-blur** (sized to the long one); **EWA** averages along the matched ellipse — crisp *and* alias-free. Drag the **probe** on the floor to see its footprint ellipse in texture space, and how each filter samples it.

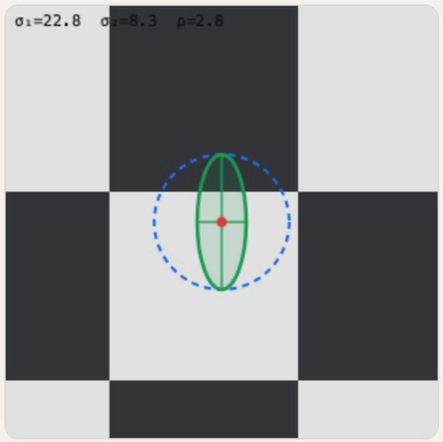
FILTER ○ nearest ○ bilinear ○ trilinear MIP ◉ anisotropic / EWA

TEXTURE ◉ checkerboard ○ fine grid ○ radial rings Tilt / foreshortening

Field of view Max anisotropy ρ max 16× ☑ show probe & footprint reset view



warped floor — drag the ○ **probe**

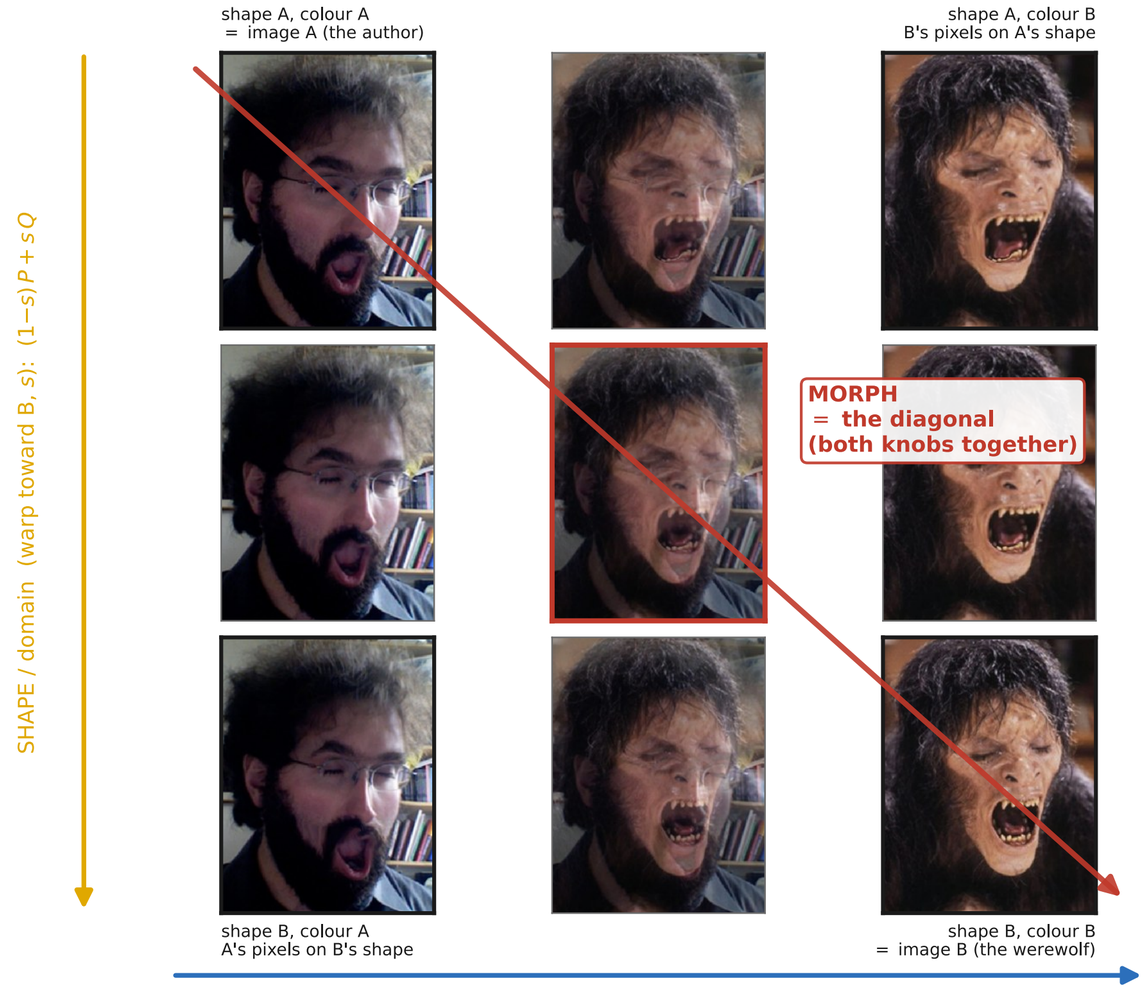


texture space — the probe pixel's **footprint**

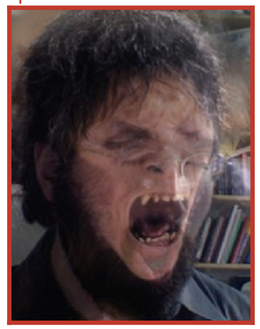
```
filter: EWA (anisotropic)    far-field horizontal high-freq energy ≈ 7.9 (lower = less
```

Figure 11.3.2: The two knobs of a morph, as orthogonal axes

Shape and colour are two orthogonal knobs of a morph: a real morph moves along the diagonal (both at once → a sharp face);
a plain cross-dissolve moves along the colour axis alone (no warp → a ghost)

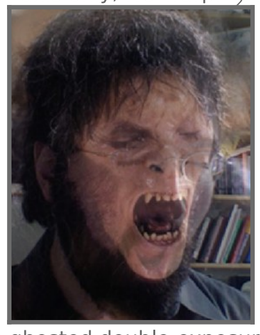


morph (centre of grid)
warp BOTH knobs to $s = c = \frac{1}{2}$



one sharp face

plain cross-dissolve
colour axis only, NO warp: $\frac{1}{2}A + \frac{1}{2}B$

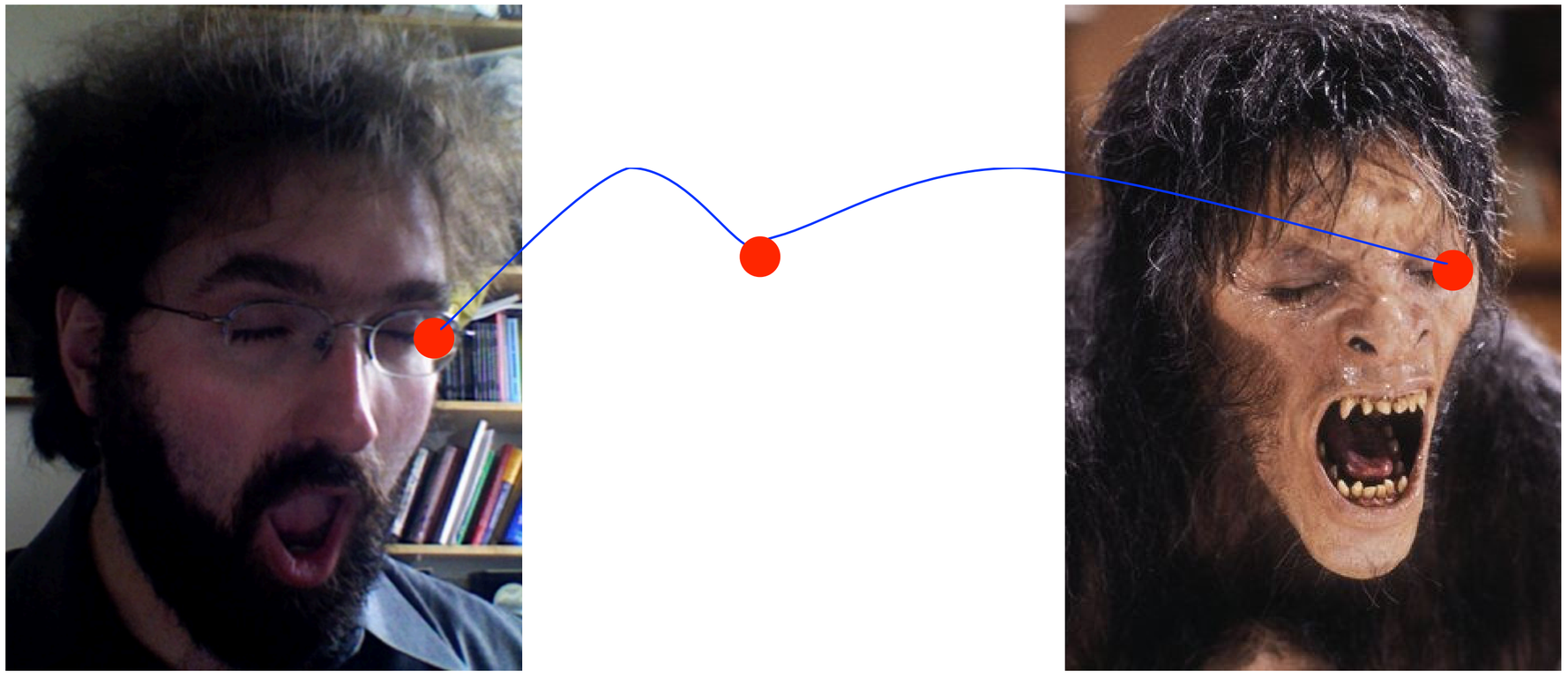


a ghosted double-exposure

One axis, range (color): the cross-dissolve C(x)=(1-t)C_0(x)+tC_1(x), a linear blend of pixel values.

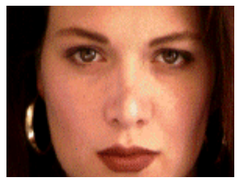
Morphing: combine both

- **For each pixel**
  - Transform its location like a vector (domain)
  - Then linearly interpolate colors (range)



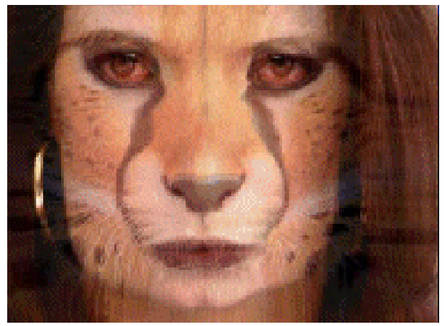
Morphing

- **Input: two images I_0 and I_{N+1}**

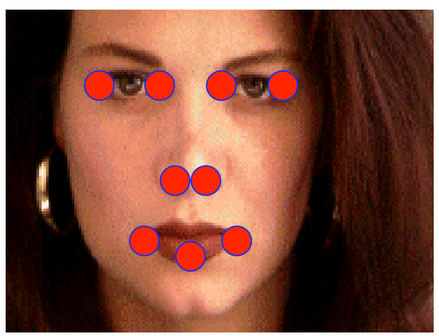


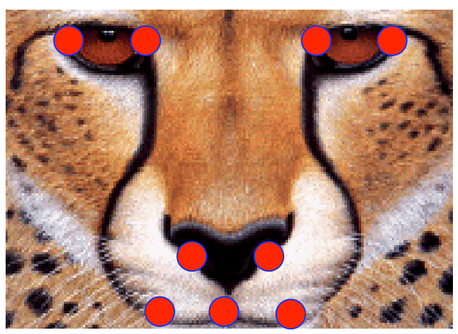


- **Expected output: image sequence I_i, with i∈ 1..N**



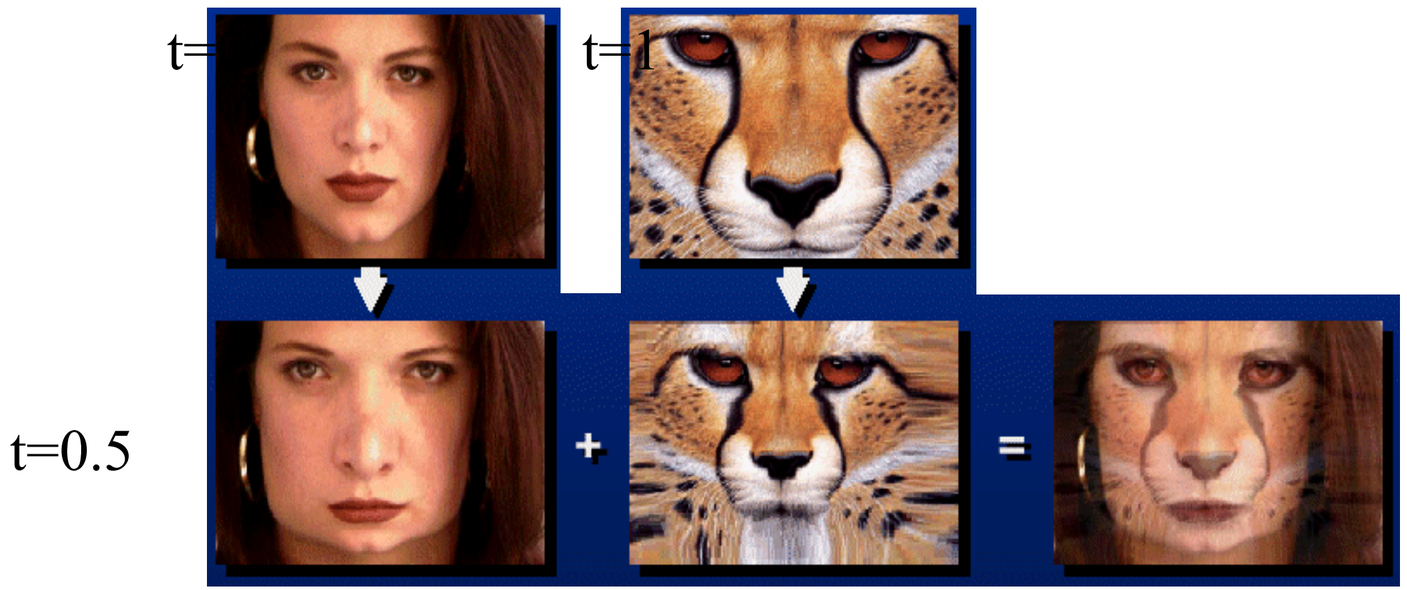
- **User specifies sparse correspondences on the images**





Morphing

- **For each intermediate frame I_t**
  - Interpolate feature locations $P^t_i = (1- t)\, P^0_i + t\, P^1_i$
  - Perform **two** warps: one for I_0, one for I_1
    - Deduce a dense warp field from the pairs of features
    - Warp the pixels
  - Linearly interpolate the two warped images



Demo: Beier–Neely morph

Beier–Neely image morph

A true two-image **morph** (not a fade): the reader draws matching pairs of **feature line segments** on image **A (before)** and image **B (after)** — jaw, eyes, nose, mouth, hairline. At time *t* each segment is interpolated to an intermediate set *S(t)*; **A is warped forward** to *S(t)*, **B is warped backward** to *S(t)* (both via the dense Beier–Neely field), and the two aligned warps are **cross-dissolved** $(1-t)\cdot A_t + t\cdot B_t$. Because the features line up, the in-between is a single coherent face, not a ghosted double exposure (Beier & Neely, SIGGRAPH 1992).

▶ play t 0.50 + add feature – delete selected ↺ reset

a 1.0 b 2.0 p 0.50

A_T — WARPED SOURCE

MORPH(T) — THE BLEND



B_T — WARPED TARGET

Left and right are the **individual warps** bent to the same intermediate features; the centre is their **cross-dissolve**.
Slide *t* from 0 (pure A) to 1 (pure B).

IMAGE A — DRAG THE **BLUE** FEATURE ENDS

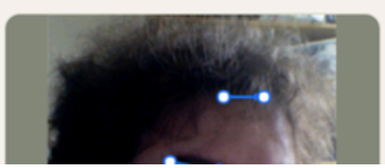
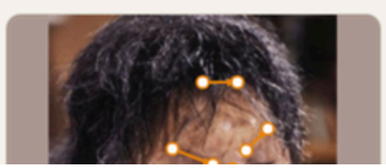


IMAGE B — DRAG THE **ORANGE** FEATURE ENDS



Demo: Mesh morph

Triangulation image morph

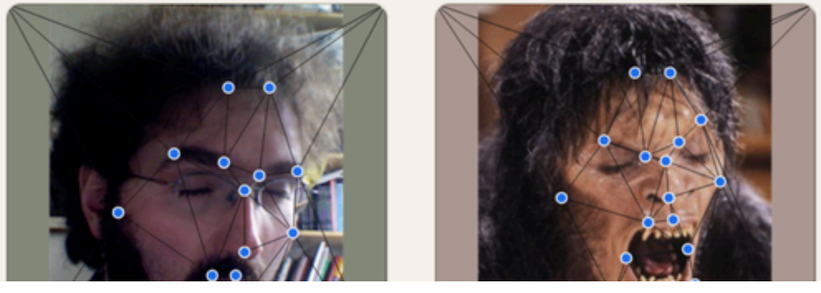
The mesh counterpart to the Beier–Neely morph. Each **control vertex** is a **correspondence**: a point on image **A** matched to a point on image **B** (here, a facial landmark on Frédo matched to the same landmark on the werewolf). One **Delaunay triangulation** ties the vertices into a mesh whose **topology stays fixed** while morphing. At time *t* each vertex sits at *(1−t)·A + t·B*; we **piecewise-affine warp** A and B onto that in-between mesh, then **cross-dissolve**. Slide *t* or press play; drag a landmark on either pane to re-shape the match.

▶ Play | ☑ show mesh | Delete vertex | Reset | 22 vertices (18 added) · 38 triangles

time t — t = 0.50



A warped to V(t) | **morph(t)** | B warped to V(t)



Warping

How do we specify the warp?

- **Before, we saw simple transformations**
  - linear, affine, perspective



translatio
n



rotation



aspect



affine



perspective

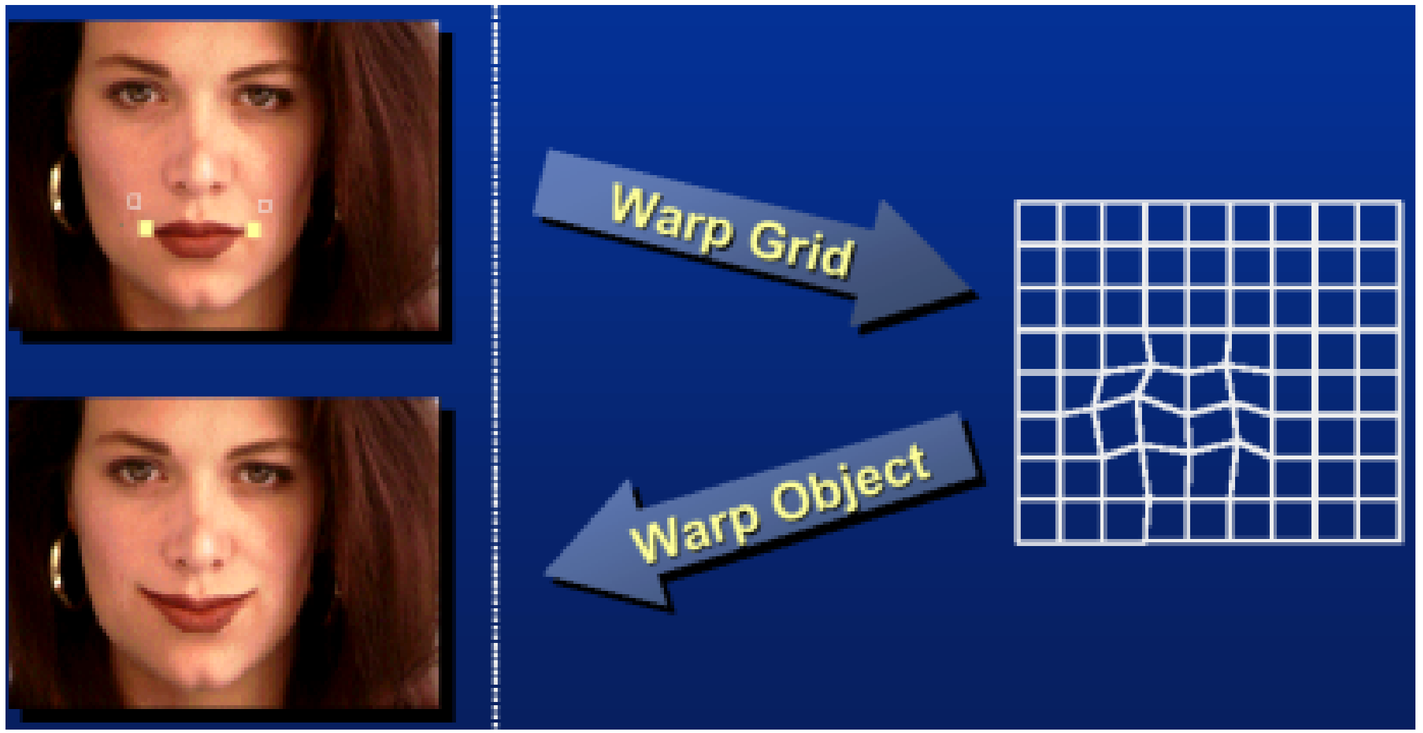


cylindrical

- **But we want more flexibility**

Image Warping – parametric

- **Move control points to specify a spline warp**
- **Spline produces a smooth vector field**



Slide Alyosha Efros

Warp specification - dense

- **How can we specify the warp?**

  Specify corresponding *spline control points*

  - *interpolate* to a complete warping function



But we want to specify only a few points, not a grid

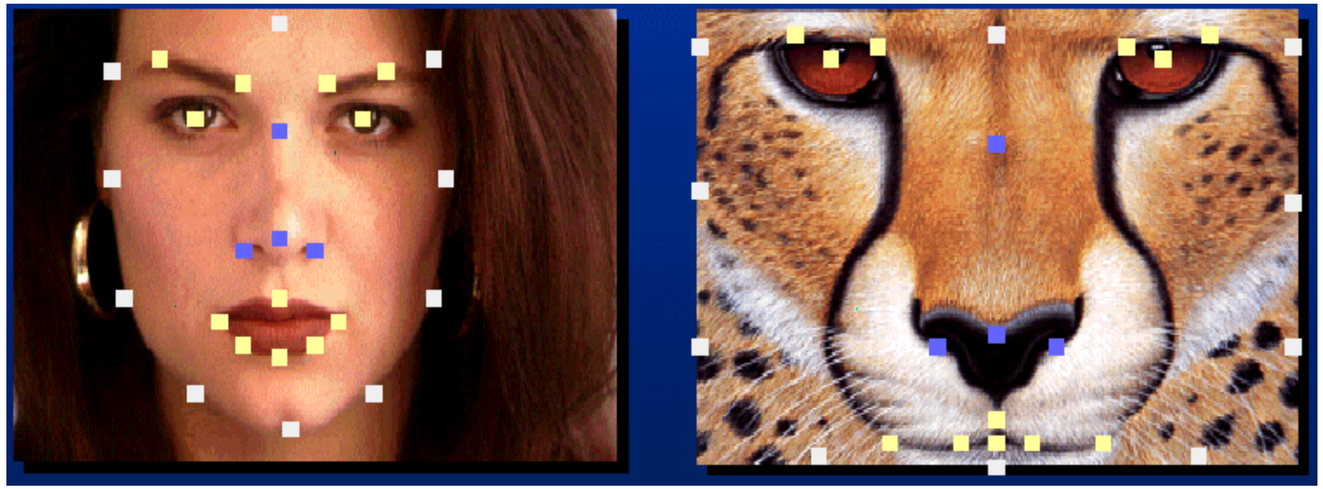
Slide Alyosha Efros

Warp specification - sparse

- **How can we specify the warp?**

  Specify corresponding *points*

  - *interpolate* to a complete warping function
  - How do we do it?



How do we go from feature points to pixels?

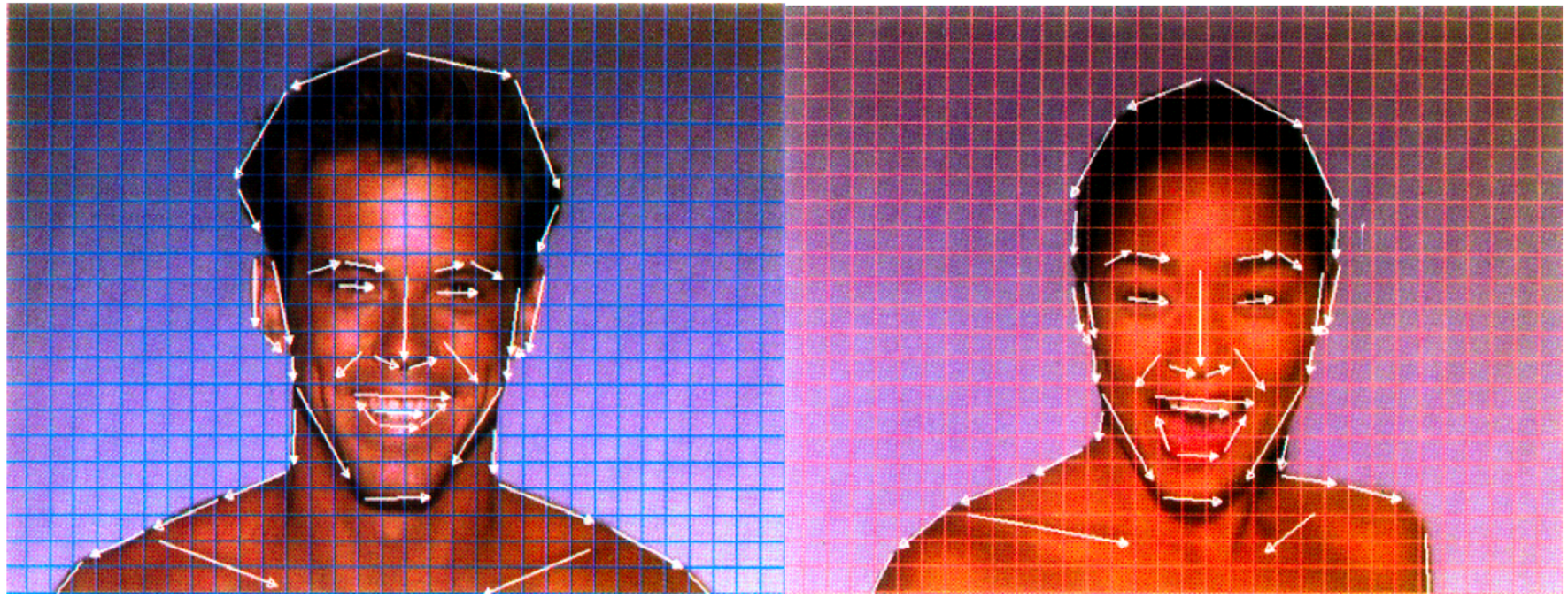
Slide Alyosha Efros

Demo: Mesh warp



Beier and Neely

- **Specify warp based on pairs of segments**
  - **http://dl.acm.org/citation.cfm?id=134003**
  - Feature-Based Metamorphosis, SIGGRAPH 1992
  - Used in Michael Jackson' Black and White Music Video
  - Pset 5!!



Video: Michael Jackson, Black or White (1991, PDI)



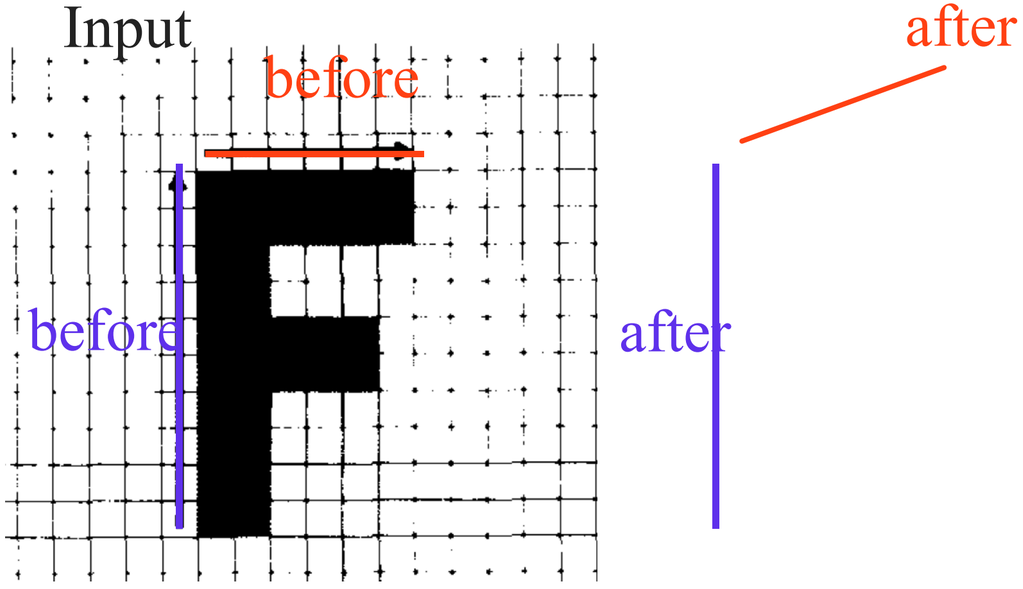
Questions?

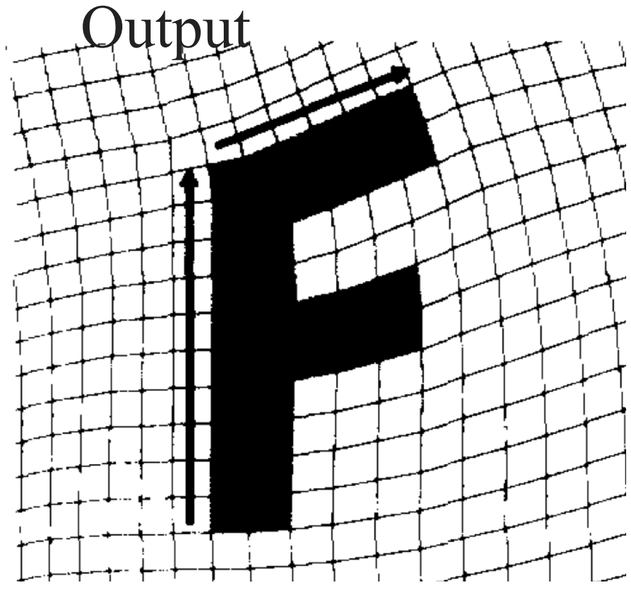


Segment-based warping

Problem statement

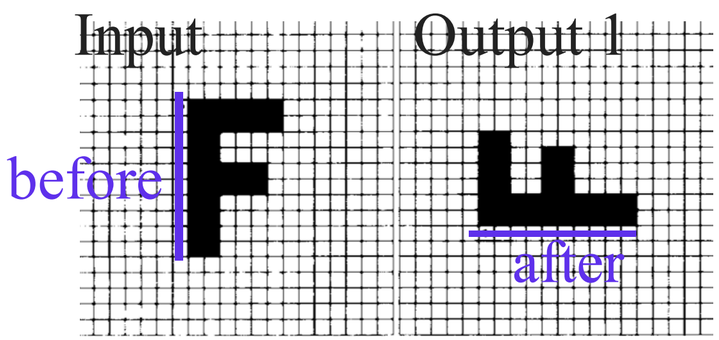
- **Inputs: One image, two lists of segments before and after, in the image domain**
- **Goal: warp the image "following" the displacement of the segments**





Idea

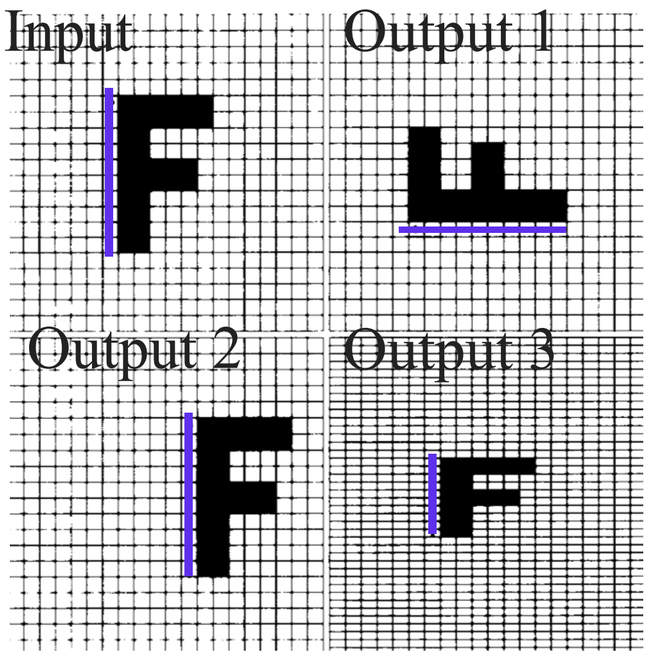
- **Each before/after pair of segment implies a planar transformation**
  - simple and affine



Single line transforms

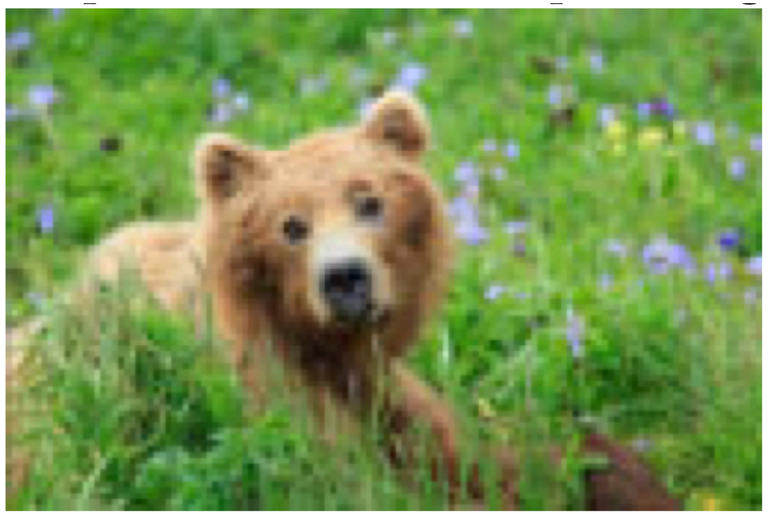
Idea

- **Each before/after pair of segment implies a planar transformation**
  - simple and affine

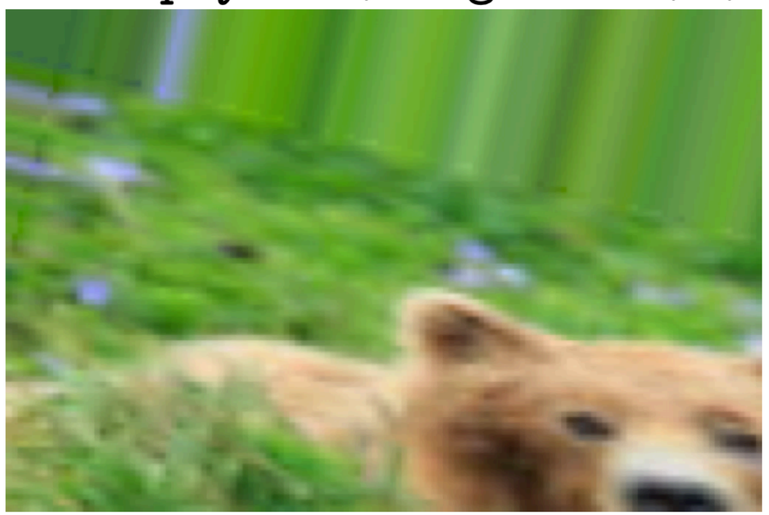


Single line transforms

Test

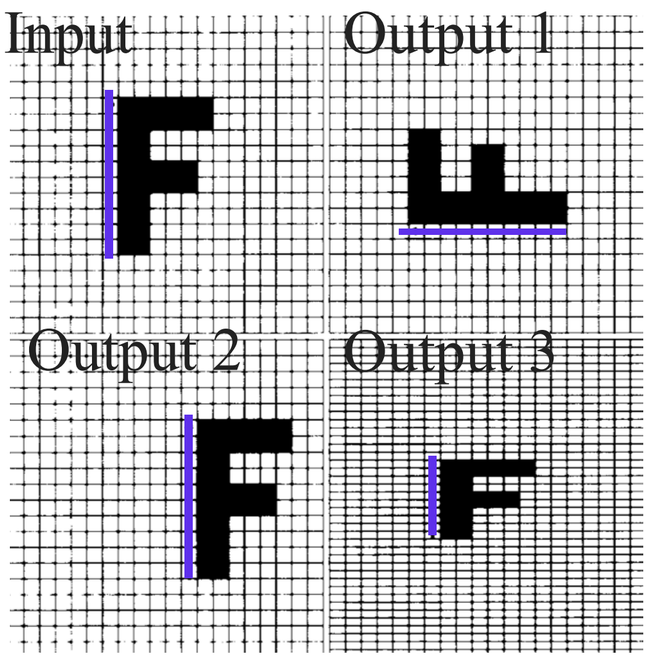


→ `warpBy1(im, segment(0,0, 10,0), segment(10, 10, 30, 15))` →

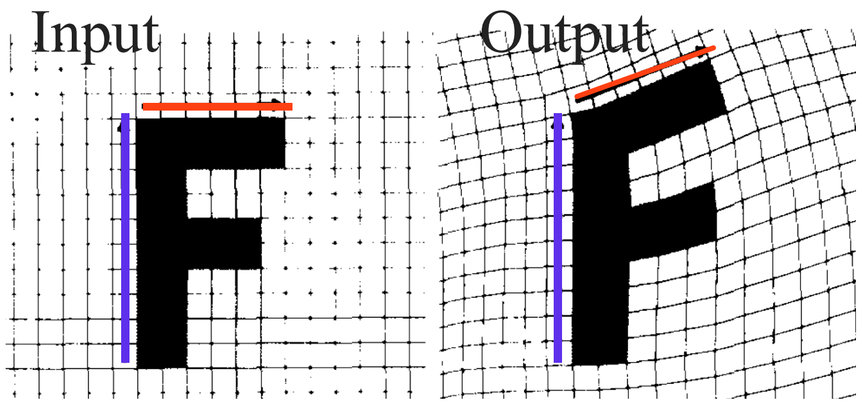


Idea

- **Each before/after pair of segment implies a planar transformation**
- **Then take weighted average of transformations**



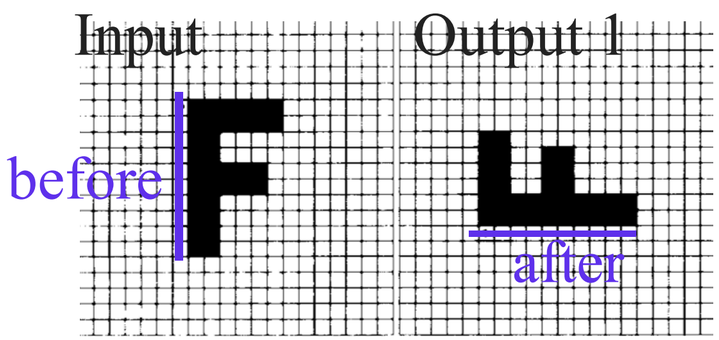
Single line transforms



Transform wrt 2 lines

Back to single segment

- **Each before/after pair of segment implies a planar transformation**
  - simple and affine

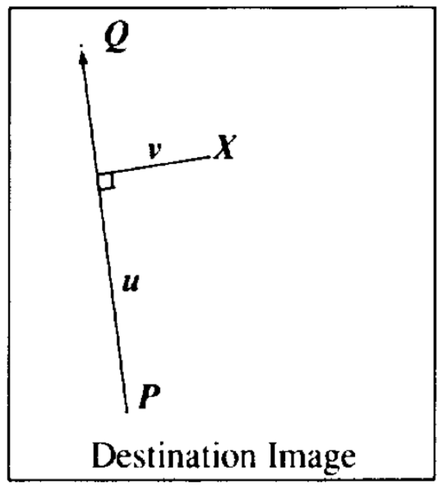


Single line transforms

Linear algebra refresher

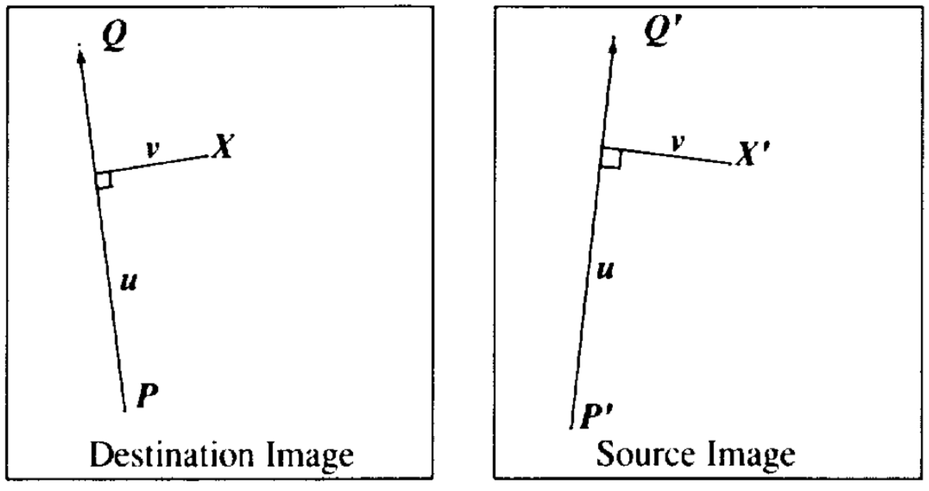
Transform wrt 1 segment

- **Define a coordinate system with respect to segment**
  - 1 dimension, *u*, along segment
  - 1 dimension, *v*, orthogonal to segment



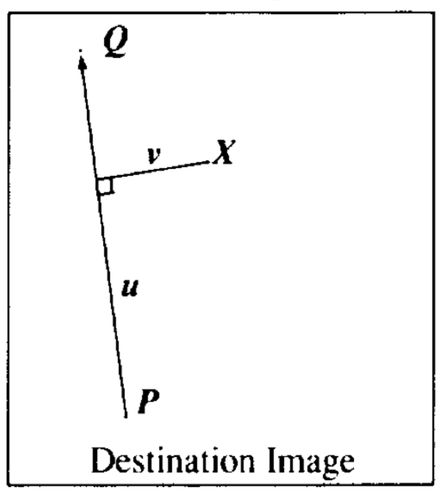
Transform wrt 1 segment

- **Define a coordinate system with respect to segment**
  - 1 dimension, *u*, along segment
  - 1 dimension, *v*, orthogonal to segment
- **Compute u, v in one image**
  - The after one, because we use the inverse transform
- **Compute point corresponding to u, v in second image**



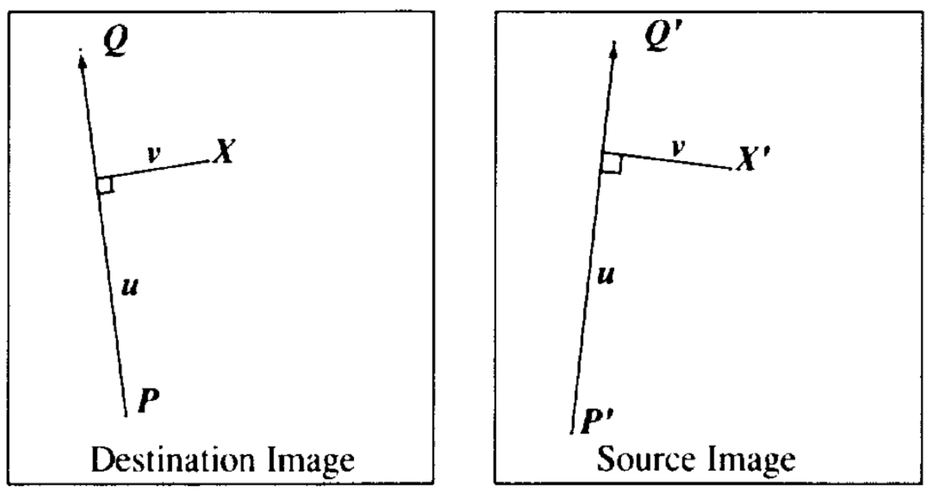
Computing u, v

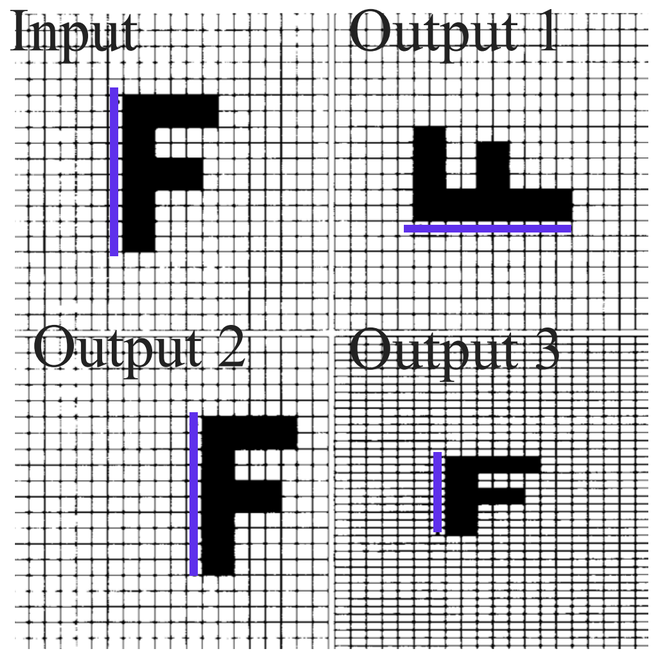
- **u = PX.PQ/||PQ||²**
  - this way u is 0 at P and 1 at Q
- **v=PX.perpendicular(PQ)/||PQ||**
  - where perpendicular(PQ) is PQ rotated by 90 degrees, and has length ||PQ||
  - unlike u which is normalized, v is in distance units



Transforming a point given u, v

- **X'=P'+uP'Q+v perpendicular(P'Q')/||P'Q'||**
- **The u component is scaled according to segment scaling**
- **But v is absolute (see output3)**
  - They say they tried to scale v as well but it didn't work as well





Single line transforms

Questions?

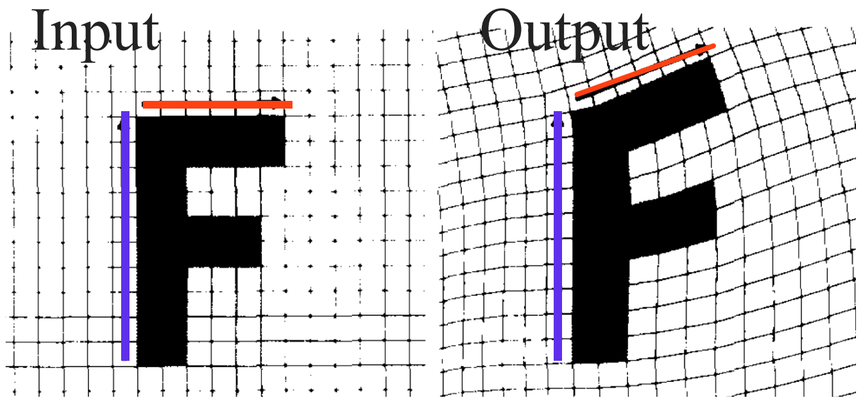


Multiple segments

- **For each point X**
  - For each segment pair s_before[i], s_after[i]
    - Transform X into X'i according to this pair
  - Compute weighted average of all transformed X'i
    - weight according to distance to segments

$$weight = \left(\frac{length^p}{a + dist} \right)^b$$

where a, b, p
control the influence



Transform wrt 2 lines

Demo: Beier–Neely warp

Beier–Neely field warp

A single-image **warp** (no second image, no morph, no blend): the reader draws pairs of directed **feature line segments** — a **source** segment (blue, where a feature *is*) and a **destination** segment (orange, where it should move *to*). The picture is **inverse-warped** by the dense Beier–Neely field so that, at each destination segment, it shows the warped-in source content. Drag the **orange** ends to deform the face; drag the **blue** ends to retarget a segment. With every destination on top of its source the warp is the **identity** (Beier & Neely, SIGGRAPH 1992).

+ add feature | – delete selected | ≡ snap to identity | ↺ reset

a 1.0 b 2.0 p 0.50 ☐ original

WARPED RESULT — DRAG THE HANDLES



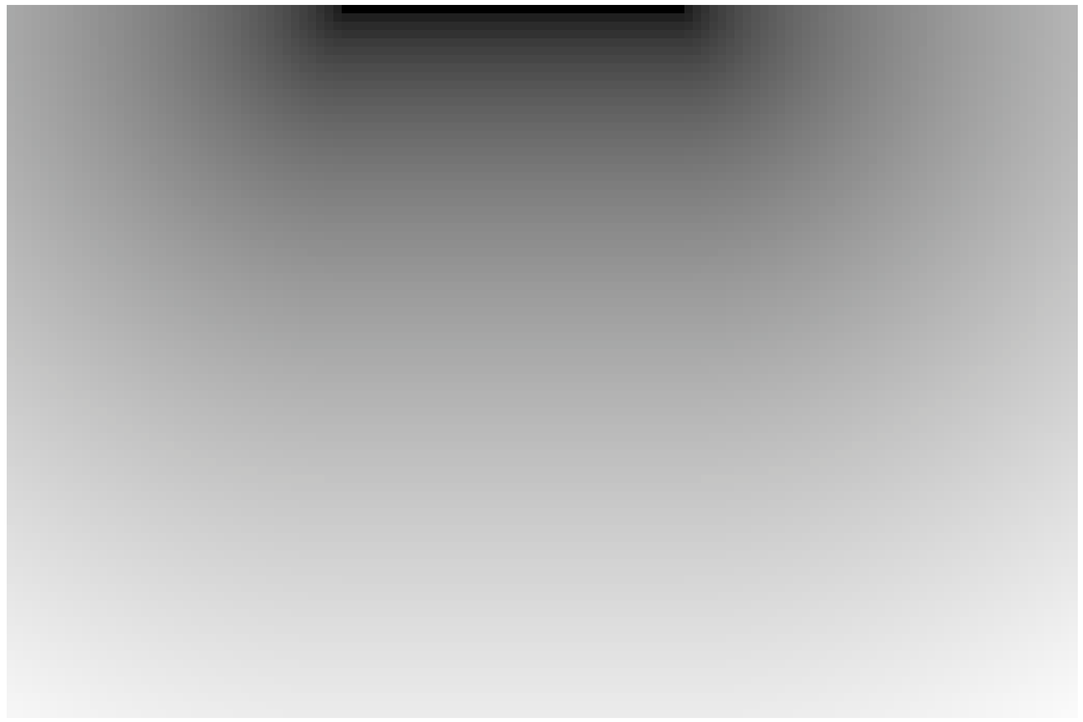
Click near an endpoint to grab it; drag to move. Each ▣ blue source segment is paired with the ▣ orange destination

Distance to a segment

- **Multiple cases...**
  - dot product, test >0, <1

Debugging: example

- **Debugging my distance function**



Questions?

Speaker notes

- **More debugging**

