ACRONYMS — THIS CHAPTER

ACRONYMS

CC	CameraCalibration matrix (in the DNG pipeline); Creative Commons in a licensing context
CC0	Creative Commons Zero (public-domain dedication)
DC	direct current (the zero-frequency, average component)
DCT	discrete cosine transform
DNG	Digital Negative (raw file format)
DOF	depth of field
FFT	fast Fourier transform
HDR	high dynamic range
ILM	Industrial Light & Magic
ISP	image signal processor
MPEG	Moving Picture Experts Group
PDI	Pacific Data Images
RANSAC	RANdom SAmple Consensus
RMS	root mean square

Course notes · 33 sections from the book · 14 optional hide optional sections the full book ↗

LECTURE NOTES 8 — RESAMPLING AND WARPING

Course notes cut from *Computational Photography, an AI-powered Slopendium* for this lecture. Each section links back to its home in the book; sections marked **optional** go beyond what the lecture needs.

▸ Contents · 29 sections on this page, from 5 chapters of the book

Sampling, downsampling, and aliasing

in the book ↗

A camera integrates light over each pixel's spatial footprint and exposure interval, then records those measurements on a grid. We can model this as filtering followed by point sampling. The filtering reduces fine detail, but generally does not remove every frequency the grid cannot resolve. Resizing introduces another sampling step and another opportunity for aliasing. Chapter 6.5, Resampling and upsampling treats reconstruction between samples; here we examine the information lost or confused by the sampling grid.

Aliasing: a high frequency in disguise

in the book ↗

Picture photographing a building's facade (rows of fine windows, or a striped shirt) and shrinking the result. Where the stripes are finer than the output pixels, they do not melt into a uniform gray; they reorganize into bold, *wrong*, low-frequency bands and swirls that are not in the scene at all: moiré. Intuitively, when a pixel must summarize detail it cannot represent, it ought to return the local *average*, a neutral gray. Instead, the leftover high frequency gets *misread* as a low one.

The clean case is a single sine wave (Figure 6.4.1). Sample a high-frequency sine too coarsely (fewer than two samples per cycle) and the samples you collect fall into a pattern *indistinguishable* from a lower-frequency sine. The original travels in disguise as a different frequency; that impostor frequency is its alias. The damage is not blur (lost detail you could shrug off); it is corruption: energy that lands at the wrong frequency and masquerades as real structure. The same masquerade plays out in 2-D on a real photograph (Figure 6.4.2): a facade's regular window grid is a high spatial frequency, and sampling it too coarsely without a pre-filter folds that grid down into bold, slow moiré bands that are not in the building.

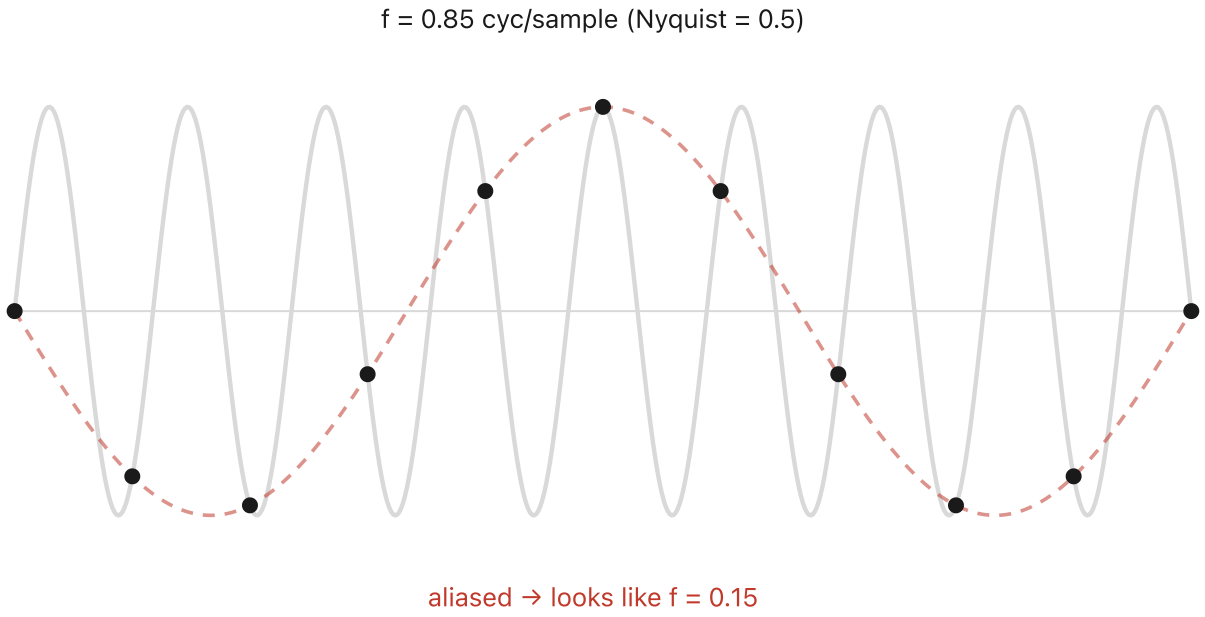


Figure 6.4.1 (reproduced). *Sampling a sine too coarsely → aliasing*. A high-frequency sinusoid (light) is sampled at too few points per cycle; the samples (dots) are consistent with a completely different, lower-frequency sinusoid (dark), its alias. Below twice the signal's frequency (the Nyquist rate) the two are indistinguishable from the samples alone: the high frequency "folds" down and masquerades as a low one. This is the moiré / wagon-wheel effect.

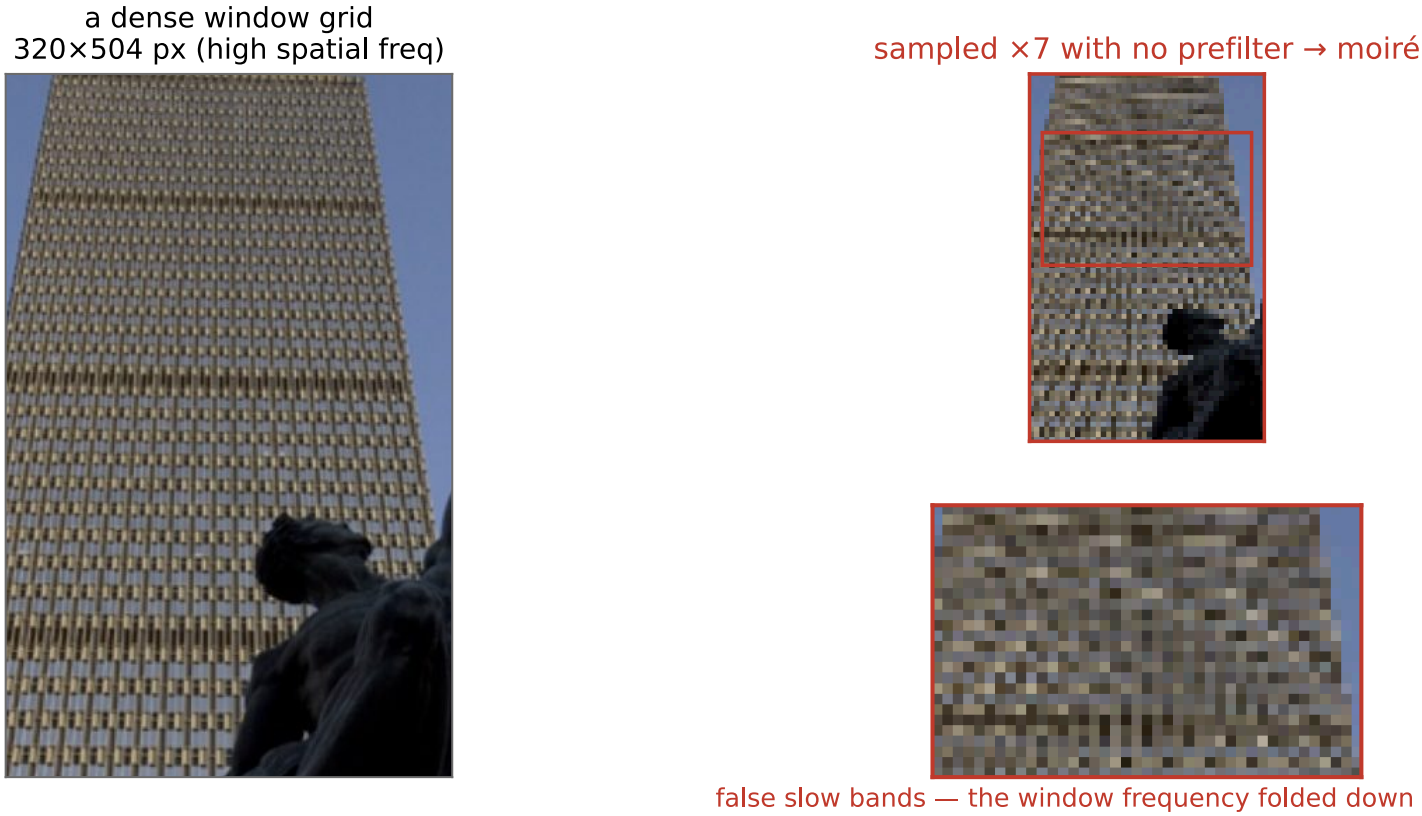


Figure 6.4.2 (reproduced). *The same aliasing on a real photo.* A building facade, a dense, near-periodic grid of windows, i.e. a high spatial frequency, sampled ×7 by simply dropping samples, with no pre-filter. The window frequency, above the coarse grid's Nyquist, folds back into bold, slow, curved moiré bands that are not in the building (center; zoom at the bottom). It is the one-dimensional sine of the previous figure acting in two dimensions on a real image: high frequency masquerading as low. The cure (pre-filter, then sample) is below.

Nyquist and the sampling theorem

OPTIONAL
in the book ↗

For an ideal band-limited signal with highest frequency $f_{\max}$, uniform samples at rate $f_s > 2f_{\max}$ determine the signal exactly. The Nyquist frequency is $f_s/2$. The strict inequality avoids the boundary case: a sine exactly at Nyquist can vanish at every sample for one phase. The theorem assumes an unlimited sequence of exact samples; finite, noisy measurements require further choices.

Frequencies separated by an integer multiple of the sampling rate produce identical complex samples:

$$e^{2\pi i(f+mf_s)n/f_s} = e^{2\pi i f n/f_s}, \qquad m, n \in \mathbb{Z}. \tag{1}$$

For example, at 100 samples per unit length, cosines at 70 and 30 cycles per unit length give the same samples. A lone sampled image cannot distinguish them without assumptions. Additional shifted measurements or a

restrictive scene model can resolve some ambiguities; applying a blur afterward cannot generally separate components already mixed by aliasing.

> **BIG LESSON (L6.13) — NYQUIST–SHANNON: SAMPLE ABOVE TWICE THE HIGHEST FREQUENCY, OR IT ALIASES**
>
> Sample above twice the highest retained frequency. If the signal is too detailed for the grid, low-pass it before sampling. This applies to space and time. Real prefilters have finite transition bands, so allow a margin below Nyquist and check residual aliasing.

The theorem cuts both ways, which is why it is so central. It tells you when sampling is *safe* (the signal is already band-limited below Nyquist, so nothing folds) and exactly what to do when it is *not*: kill the offending frequencies before they fold. And because it is a statement about any sampled signal, it reappears far beyond image shrinking: in audio (the 44.1 kHz CD rate is a little over twice the ~20 kHz of human hearing), in video frame rates, and (on the time rather than the space axis) as the temporal aliasing we just named.

In two dimensions, aliasing has a direction

OPTIONAL
in the book ↗

On a rectangular 2-D grid, aliasing folds each frequency coordinate by the sampling rate along that axis. A frequency vector can therefore change both length and orientation. A purely vertical grille does not acquire an arbitrary diagonal orientation on an aligned rectangular grid; both frequency coordinates and the grid geometry determine the alias.

A sharp slanted edge contains arbitrarily high frequencies perpendicular to the edge. Point-sampling it produces a staircase whose pattern depends on its position relative to the grid. Pixel-area integration softens the staircase, though a box footprint is not an ideal band limit. Rendering and image reduction use the same prefiltering principle.

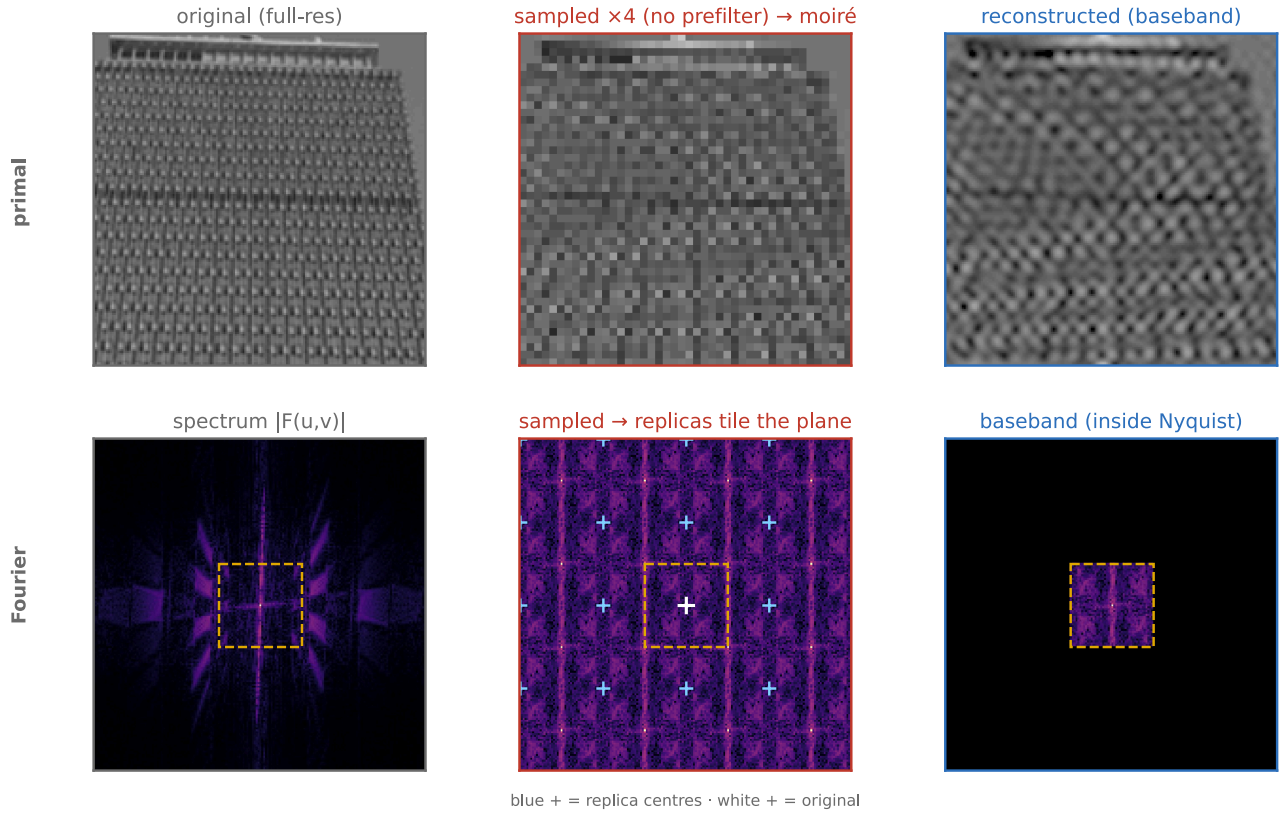


Figure 6.4.3 (reproduced). *Aliasing is directional in 2-D. A regular 2-D pattern (a near-periodic grid and a slanted edge), point-sampled on a coarse grid without a prefilter, aliases in ways 1-D cannot: the grid's high spatial frequency folds to a low frequency of a different orientation, so the moiré bands run in a direction unrelated to the windows that caused them, while the slanted edge breaks into the canonical stair-step / "jaggies." A 2-D frequency carries a direction as well as a rate, and aliasing folds both, which is why 2-D moiré appears in unexpected directions. The cure is the same as in 1-D: prefilter (area-average) before sampling.*

For example, sample a sinusoid with frequency components 0.6 cycles per pixel horizontally and 0.2 vertically on an integer pixel grid. Subtracting one horizontal cycle per pixel changes its frequency vector to (−0.4, 0.2) without changing the samples: the phase difference at every integer horizontal coordinate is a whole number of cycles. The apparent orientation changes because the horizontal component has folded while the vertical component has not. A regular texture containing several frequency components can therefore produce moiré at orientations absent from the visible coarse structure of the original texture.

Sampling in the frequency domain: spectral replicas

OPTIONAL
in the book ↗

In the frequency domain the mechanism is vivid. Sampling on a grid is multiplying the signal by a comb of spikes, and multiplying by a comb in space *replicates* the spectrum: it pastes shifted copies of the signal's spectrum at every multiple of the sampling rate. If the original spectrum is narrow enough (band-limited below Nyquist), the copies sit side by side without touching and the original can be isolated. If it is too wide, the copies overlap, the tails fold back over the genuine low frequencies, and that overlap *is* aliasing. Seen this way the fix is obvious: band-limit

the signal before you sample it. Run a low-pass filter (a blur) first, to delete the frequencies above Nyquist that would otherwise fold, *then* sample. This pre-filter (anti-aliasing) trades a little sharpness for the removal of false structure, and it is why a well-written image-shrinking routine blurs before it downsamples. The space-versus-frequency tradeoff returns: a perfectly sharp cutoff in frequency is a sinc spread out forever in space, so practical pre-filters compromise.

Figure 6.4.4 draws the multiplication in space first, then the replicated spectrum it produces.

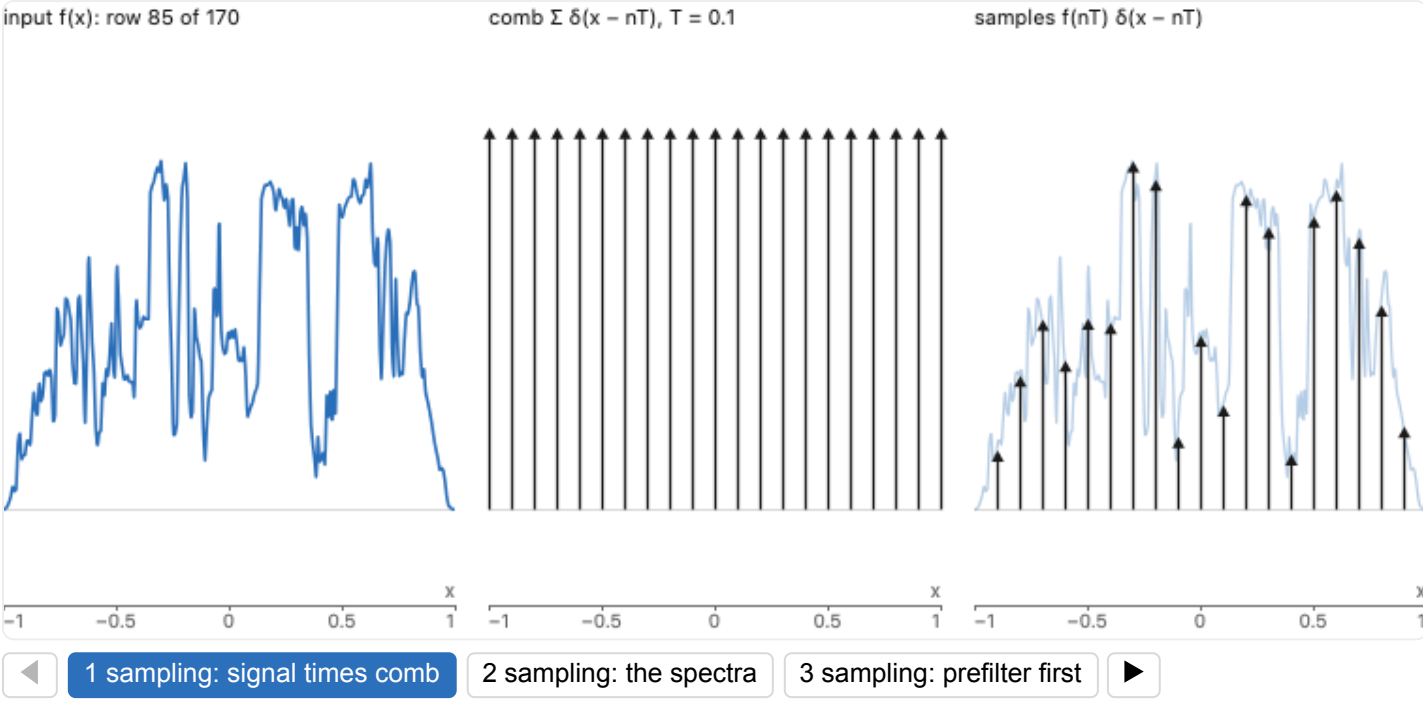


Figure 6.4.4, stage 1 (reproduced). *Sampling as multiplication by a Dirac comb. Left: the input f(x), a sine, a box, a sawtooth, a box train, or one row of a photograph (linearly interpolated and tapered at its ends). Middle: the comb Σ δ(x − nT), unit Diracs drawn as arrows every T. Right: their product, Diracs of weight f(nT), with f drawn faintly behind. Change the input, its parameters and the sampling rate 1/T: the arrows on the right are all that remains of f.*

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

⤓ settings from the previous stage

Figure 6.4.4, stage 2 (reproduced). *The same three signals with, below each, its Fourier transform, u in cycles per unit length. Left: the input's spectrum, Diracs (arrows) for the periodic inputs and a continuous curve for the box and the scanline; real spectra (box, box train) are drawn signed, the others as magnitudes. Middle: the comb's transform, another comb, with teeth of weight 1/T every 1/T, each in its own color, k and −k different (blue for k = 0, red for +1, orange for −1, green for +2, purple for −2). Right: the samples' transform, (1/T) Σ F(u − k/T), one copy of the input's spectrum per tooth in that tooth's color, drawn over their sum in gray. Dirac copies that land on the same frequency add, and that sum is drawn black too. The black curve is computed from the samples themselves (their discrete-time Fourier transform) and matches the sum of copies, which is Poisson summation. The shaded band is ±1/(2T), the Nyquist band. Lower the rate until a copy with k ≠ 0 enters the band: that is aliasing. A sine at 3 cycles per unit sampled at 4.5 per unit shows its k = ±1 copies at ±1.5.*

loading the photograph...

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

↓ settings from the previous stage

Figure 6.4.4, stage 3 (reproduced). *Prefilter first. The same pipeline with the input replaced by $g = f * h$, where h is a box of width T, a Gaussian with $\sigma = T/2$, or a Lanczos-3 kernel stretched to the sampling period (Gaussian by default; "none" brings back the unfiltered case to compare). Top left, g with f faint behind it; below, $G = F \cdot H$ with $|F|$ faint and H in gray. The samples now read g, and the samples' spectrum keeps the scale the unfiltered copies had, so what the prefilter removed shows: the copies' tails that reached into the Nyquist band, the aliasing of the previous stage, are mostly gone, at the price of the detail H attenuates below Nyquist. A sine at 7 cycles per unit sampled at 10 per unit still aliases to 3 cycles per unit, but the Gaussian has kept only 0.09 of it.*

💡 **BIG LESSON (L6.14) — SAMPLING REPLICATES THE SPECTRUM; ALIASING IS THE REPLICAS OVERLAPPING**

Sampling on a grid is multiplication by a comb (a train of deltas), and a comb's Fourier transform is another comb, so sampling replicates the signal's spectrum at every multiple of the sampling rate. If the spectrum is band-limited below Nyquist the copies sit side by side and the original is recoverable; if it is not, the copies overlap, and the folded-over tails land on top of the genuine low frequencies as aliasing: energy appearing at completely *wrong* frequencies (moiré, the backward wagon-wheel). The cure is to band-limit before sampling: low-pass away everything above Nyquist first, then sample.

Seeing it: sampling and pre-filtering, hands on

in the book ↗

The demos below show this directly. The first demo is the one-dimensional story (a single signal, your hand on the sampling rate), and the second is the two-dimensional one, the same physics on a real photograph where aliasing shows up as moiré.

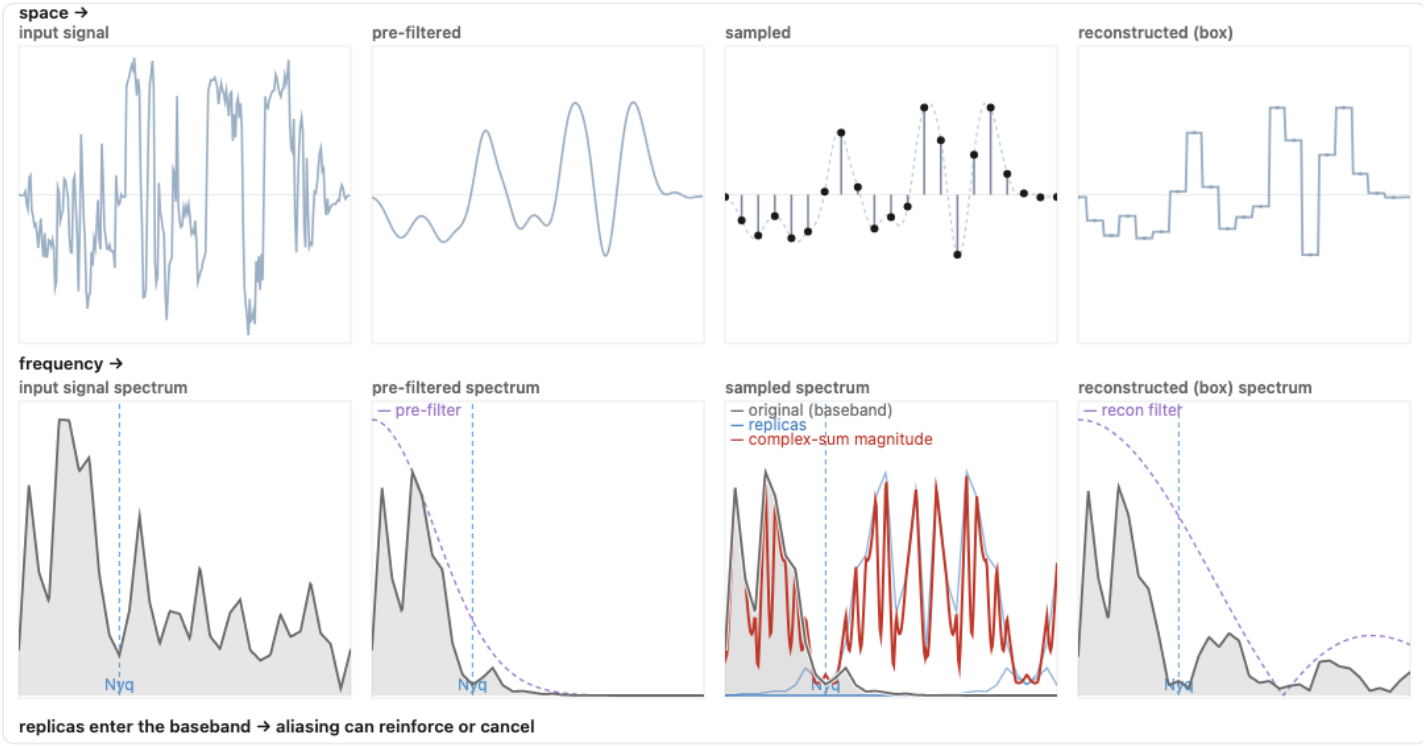
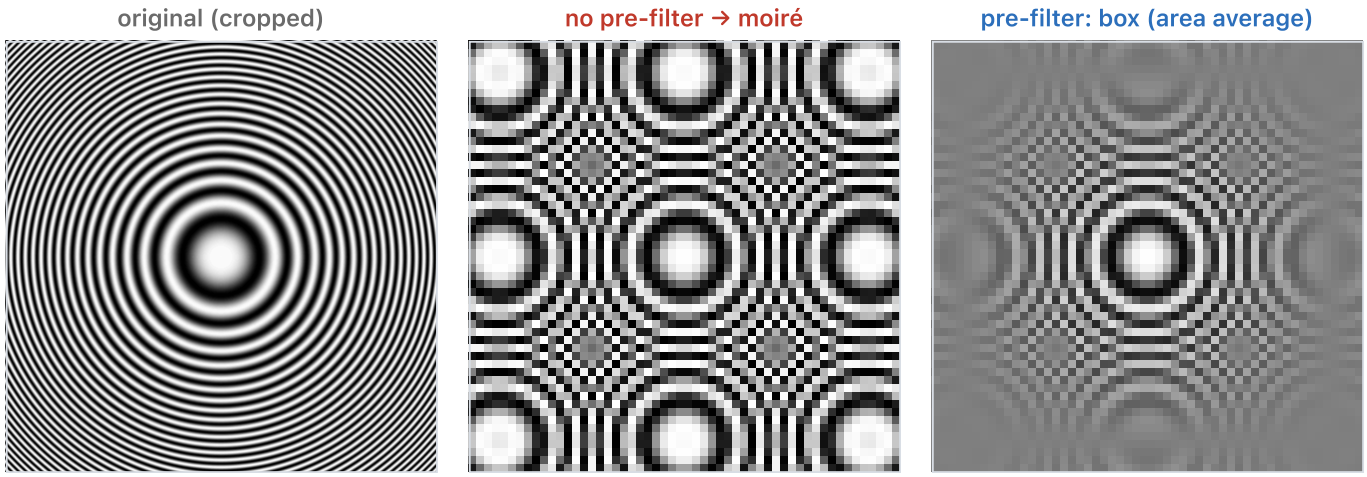


Figure 6.4.5 (reproduced). *Interactive (1-D): sampling, aliasing, and the pre-filter. An input, by default the same image scanline as Figure 6.4.4 (the middle row of the old-town rooftops), or a periodic signal (a sine, square or sawtooth at a frequency you choose), is shown with its samples; drag the sampling rate and watch the reconstructed signal alias to a lower frequency the moment you drop below Nyquist (two samples per cycle), with both the signal and its spectrum (the replicated spectral copies) drawn side by side. Where replicas overlap, the spectrum shows the magnitude of their complex sum, which includes reinforcement and cancellation; the magnitudes of overlapping copies do not simply add. Toggle the anti-aliasing pre-filter (a low-pass blur applied before sampling) and watch the offending high frequencies attenuated* before *they can fold: the alias fades, at the cost of a little blur. A Gaussian or other finite filter attenuates rather than strictly band-limits, so some alias can remain. Tick marks mark the sample locations.*



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 ▶

Figure 6.4.6, stage 3 (reproduced). *Interactive (2-D): downsampling, aliasing, and the pre-filter. Choose a source, a high-frequency zone-plate (the default; concentric rings that get finer toward the edge), a resolution chart, or a real photo, and downsample it by an adjustable factor, shown three ways side by side. Original: the true rings. No pre-filter: point-sampling the coarse grid folds the fine outer rings into bold, spurious moiré rosettes that are nowhere in the pattern: 2-D aliasing. With pre-filter: attenuating the fine rings before decimating removes most of the moiré; no finite filter removes all of it. Choose the filter, box (area average), bilinear, the bicubic (Mitchell–Netravali) family, or Lanczos (windowed sinc, the sharpest), and raise the factor to watch the middle's moiré grow while the right mostly softens. Compare residual moiré, blur and ringing across the filters: none is best on all three. The panels are cropped to the valid interior so every pixel shown had the pre-filter's full support: a kernel near the image edge reaches past it and the demo edge-clamps those reads, which would bias the border, so the artifact you might worry about there is boundary handling, not kernel truncation (the truncated kernel is renormalised), and cropping to full-support pixels removes it from the display.*

The full sampling *pipeline* generalizes to two dimensions just as cleanly, and the spectral picture is the most revealing part: in 2-D, sampling replicates the spectrum on a whole lattice of copies, and aliasing is a copy drifting in to overlap the baseband.

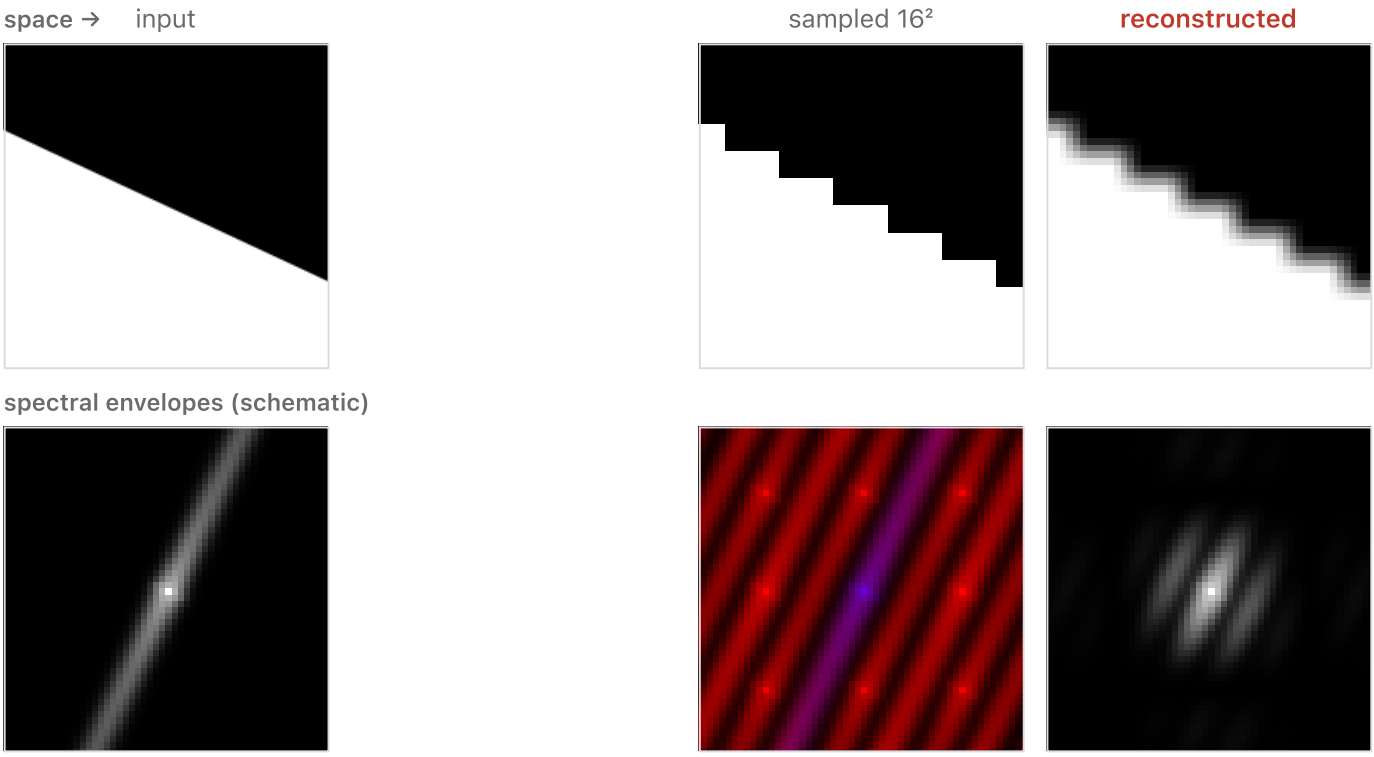


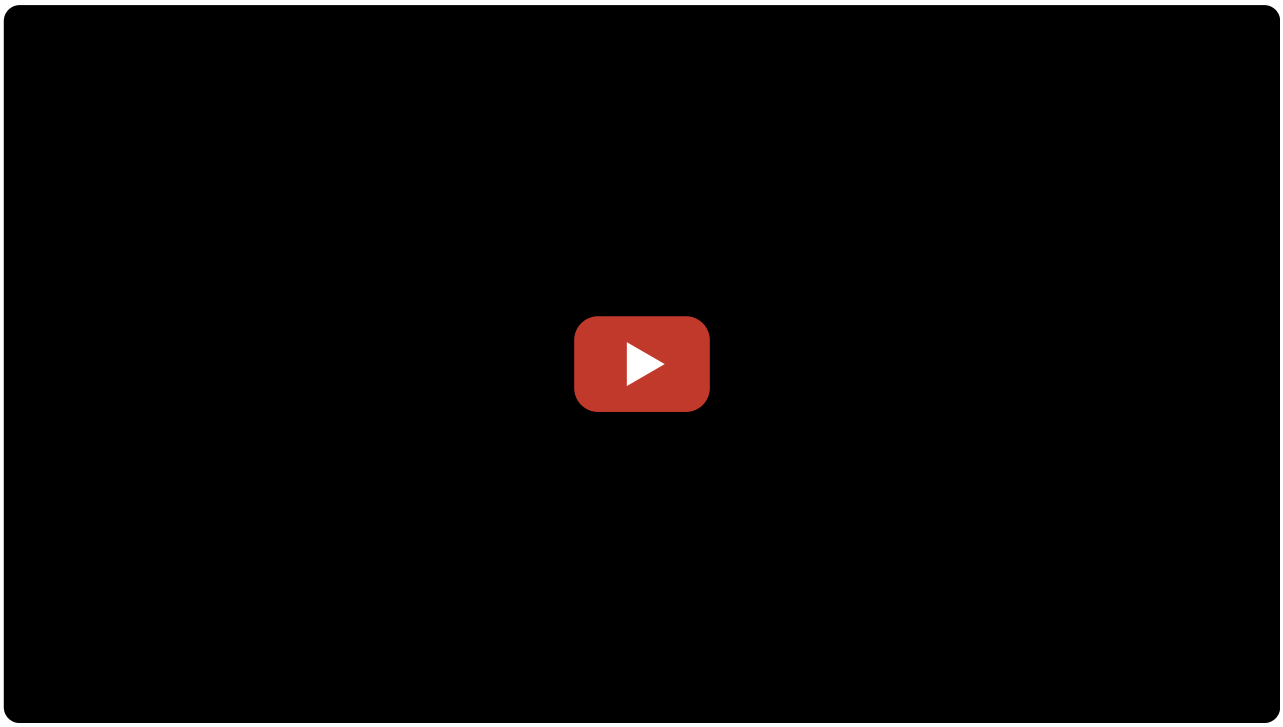
Figure 6.4.7 (reproduced). *Interactive (2-D pipeline): the planar version of the 1-D figure, four columns (input → pre-filtered → sampled → reconstructed), each shown both as an image (top) and as its 2-D Fourier magnitude (bottom, DC centered, out to about 3.3× the sampling Nyquist); with no pre-filter, the second column is left empty. The input is a 2-D sine (set frequency and orientation) or a straight edge (set its angle). Sampling on a coarse grid tiles the spectrum on a lattice: drop the rate and the replicas march inward until one overlaps the baseband: 2-D aliasing, which the reconstruction then renders as a wrong, lower-frequency moiré. A dashed box on the pre-filtered spectrum marks the sampling Nyquist, and in the sampled spectrum the baseband is blue and the alias replicas red, so red over blue is the aliasing at a glance. A pre-filter attenuates the baseband's high frequencies so the copies overlap less, though a finite filter does not remove every alias; the reconstruction filter interpolates the samples back. The pre-filter and reconstruction run as real spatial convolutions, and the upper images are what to judge. The spectrum panels are illustrations: for the sine they come from windowed FFT magnitudes, for the edge from a broadened, flattened model of an ideal step's spectral line (drawn for legibility, not the exact $1/f$ transform), and the red and blue overlays add magnitudes to show where copies could overlap, without the phase that decides the actual sampled transform. Each panel is scaled on its own, so brightness does not compare energy across panels.*

Sidebar 1: aliasing without Fourier

Two continuous signals that agree at every point sample are indistinguishable on that grid. A sensor pixel averages a spatial footprint instead, which attenuates some fine variation before the grid samples it. That helps, but a box average has spectral side lobes and does not eliminate all aliases. The footprint and the sample spacing are separate parts of the model.

Sidebar 2: aliasing on purpose

Moiré can also measure the relative displacement of two known fine patterns: a small change in alignment moves their broad beat pattern. The known pattern supplies information absent from arbitrary-image reconstruction. A related printed-animation example is KineCAM, which interleaves captured frames into a kinegram viewed through a striped overlay. It does not recover camera pose from sensor-grid moiré.



Video 1. Video: The more general uncertainty principle, regarding Fourier transforms, by 3Blue1Brown (**YouTube**).

The ideal reconstruction filter: sinc, and why it is unreachable

OPTIONAL
in the book ↗

For uniform spacing $T = 1/f_s$, ideal reconstruction of a signal band-limited below Nyquist is

$$I(x) = \sum_{n=-\infty}^{\infty} I(nT)\,\mathrm{sinc}(x/T - n), \qquad \mathrm{sinc}(z) = \frac{\sin(\pi z)}{\pi z}, \quad \mathrm{sinc}(0) = 1. \tag{2}$$

Each sample contributes a shifted sinc. At the sample locations, every other sinc is zero. The infinite sum makes exact reconstruction impractical for a finite image. Practical kernels limit support and trade passband attenuation, stopband leakage, and ringing. Sharp low-pass filtering can overshoot an edge even before truncation; truncation is not the sole cause. Chapter 6.5, Resampling and upsampling develops the usable kernels.

BIG LESSON (L6.15) — THE IDEAL SAMPLING FILTER IS A SINC, WHICH DOES NOT EXIST IN PRACTICE

Ideal band limiting and ideal interpolation use sinc kernels, scaled for their respective grids. Their infinite support requires approximation in finite computations. A finite filter trades computation and frequency selectivity; inspect both aliasing and edge overshoot.

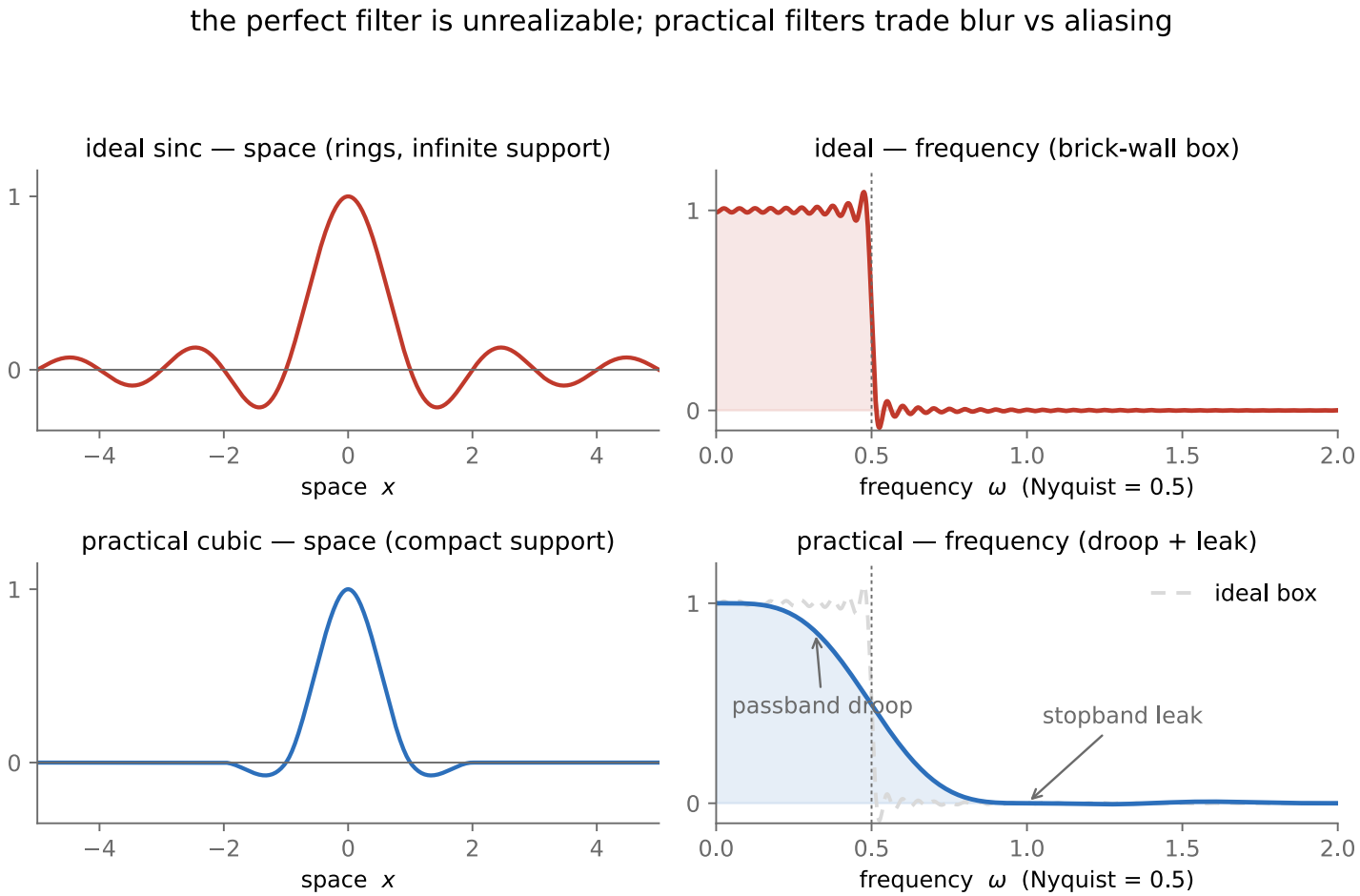
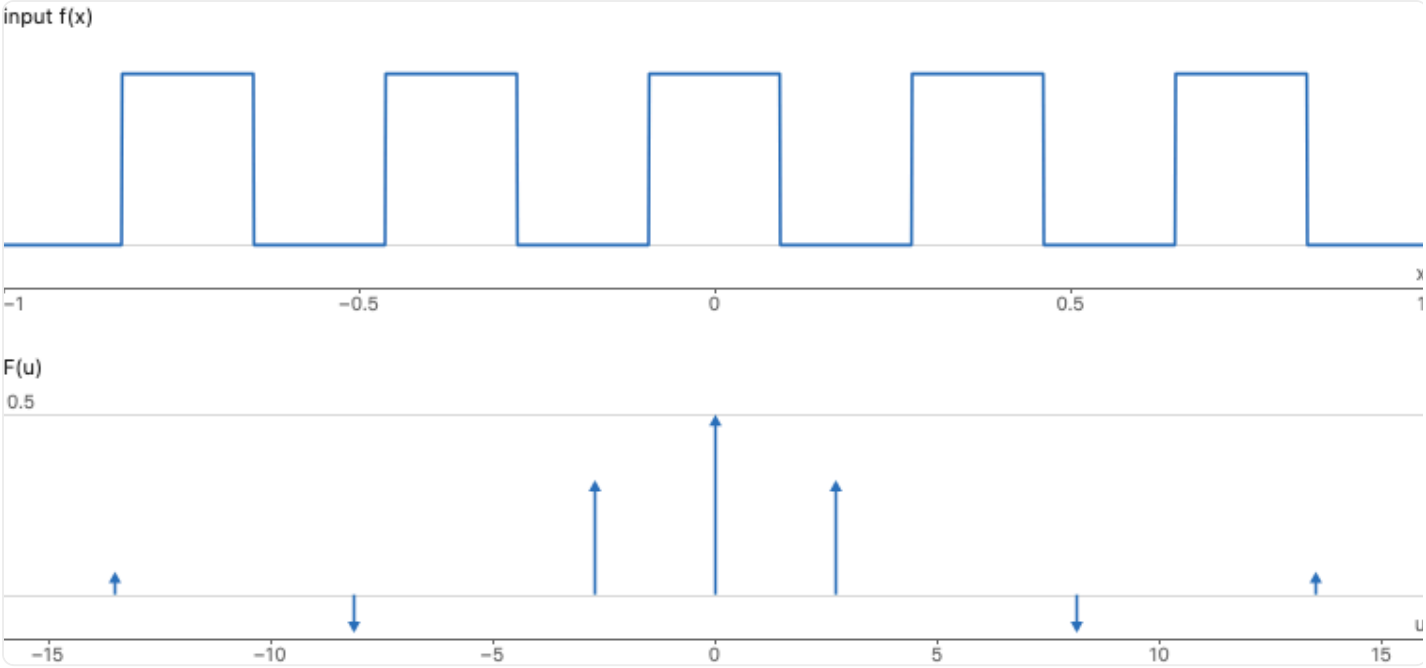


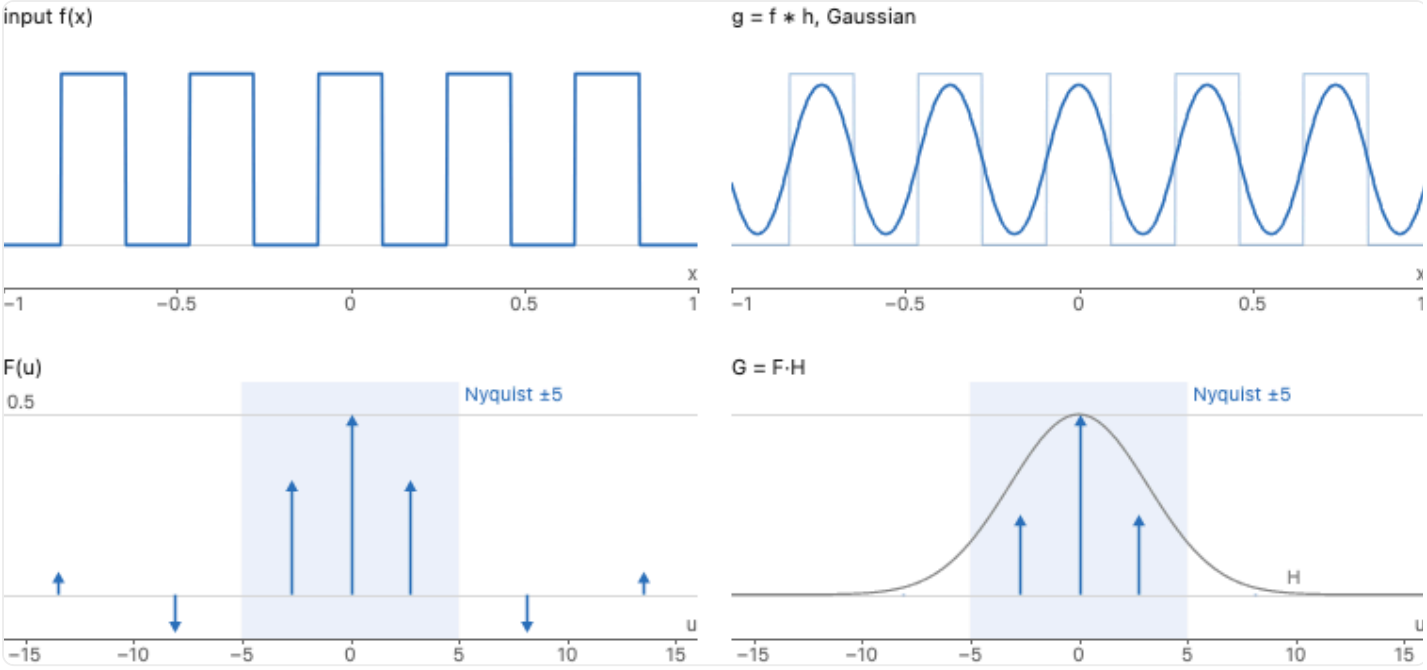
Figure 6.4.8 (reproduced). *Ideal sinc vs a practical filter, in both domains.* Top row (space): the ideal sinc reconstruction kernel ripples on forever, while a practical filter has finite support. Bottom row (frequency): the sinc is a perfect box (flat passband, vertical cutoff at Nyquist, zero beyond), whereas the practical filter rolls off gradually, slightly softening mid-frequencies and leaking a little past Nyquist. The vertical wall in frequency is exactly what forces the infinite ripple in space.

Figure 6.4.9 puts the whole chain in one place, one column per step: the input, the prefilter, the samples, the reconstruction, and the reconstruction as a viewer perceives it.



1 pipeline: the input | 2 pipeline: prefilter | 3 pipeline: sample | 4 pipeline: reconstruct | 5 pipeline: perceive

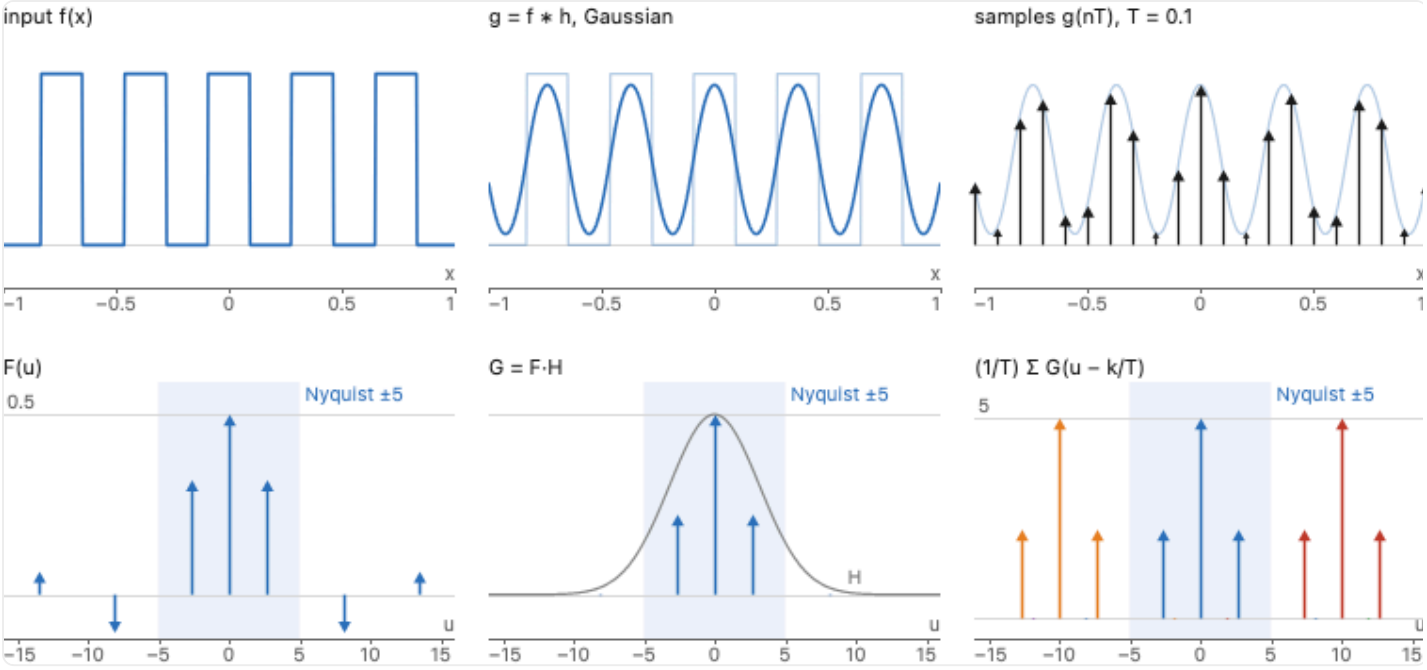
Figure 6.4.9, stage 1 (reproduced). *The sampling pipeline in one dimension, one column per step, each signal above and its Fourier transform below (u in cycles per unit length). The first column is the input: a box train (period 0.37, half on, the default), a sine, a box, a sawtooth, or one row of a photograph. Periodic inputs have Diracs for spectra, drawn as arrows; real spectra are drawn signed, the others as magnitudes.*



◀ 1 pipeline: the input | 2 pipeline: prefilter | 3 pipeline: sample | 4 pipeline: reconstruct | 5 pipeline: perceive

▶ ↓ settings from the previous stage

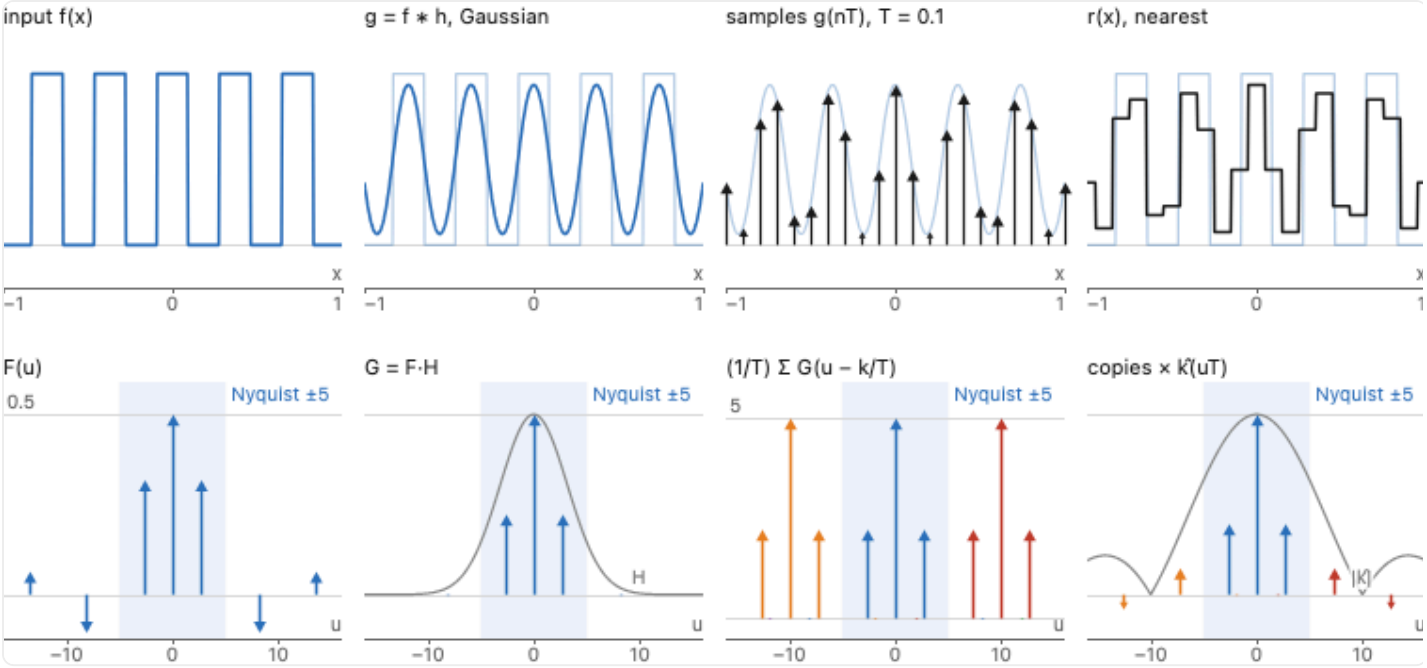
Figure 6.4.9, stage 2 (reproduced). *The prefilter, $g = f * h$, with h tied to the sample spacing T: a box of width T, a Gaussian of standard deviation T/2, or a Lanczos-3 windowed sinc scaled by T. Below, G = F·H, with H in gray on a scale where the top rule is 1, and the Nyquist band ±1/(2T) shaded. The stage opens on the Gaussian, which at 10 samples per unit keeps 0.70 of the box train's fundamental (2.70 cycles per unit) and 0.04 of its third harmonic (8.11). Switch the filter and move the rate to see what each keeps inside the band and lets through beyond it.*



◀ 1 pipeline: the input | 2 pipeline: prefilter | 3 pipeline: sample | 4 pipeline: reconstruct | 5 pipeline: perceive

▶ ↓ settings from the previous stage

Figure 6.4.9, stage 3 (reproduced). *Sampling: Diracs of weight g(nT), with g faint behind. Below, the samples' spectrum (1/T) Σ G(u − k/T), one copy of G per multiple of 1/T, each in the color of its tooth in Figure 6.4.4 (blue for k = 0, red for +1, orange for −1, green for +2, purple for −2). Where copies land on the same frequency they add, and an arrow two or more copies share is black. Set the prefilter to none: the box train's third harmonic, at 8.11 cycles per unit, comes back at 1.89 as a red arrow inside the band, and higher harmonics scatter colored arrows across it. That is aliasing.*



◀ 1 pipeline: the input | 2 pipeline: prefilter | 3 pipeline: sample | 4 pipeline: reconstruct | 5 pipeline: perceive

▶ ↓ settings from the previous stage

Figure 6.4.9, stage 4 (reproduced). *Reconstruction: $r(x) = \Sigma\, g(nT)\, k((x - nT)/T)$, with the kernel k nearest, linear, cubic (Keys) or a Lanczos-3 windowed sinc, and the input f faint behind. Below, the copies multiplied by the kernel's transform $\hat{k}(uT)$, its magnitude in gray. The $k = 0$ copy is the one we want; what survives of the others is imaging, and it is what makes the steps of a nearest reconstruction: at 10 samples per unit the fundamental's first image, at 7.30 cycles per unit, keeps 0.33 of its weight. Linear leaves 0.12 of that image, cubic 0.10 and the windowed sinc 0.004; linear also attenuates the fundamental most (0.76, against 0.87 for nearest, 0.90 for cubic and 0.995 for the windowed sinc).*

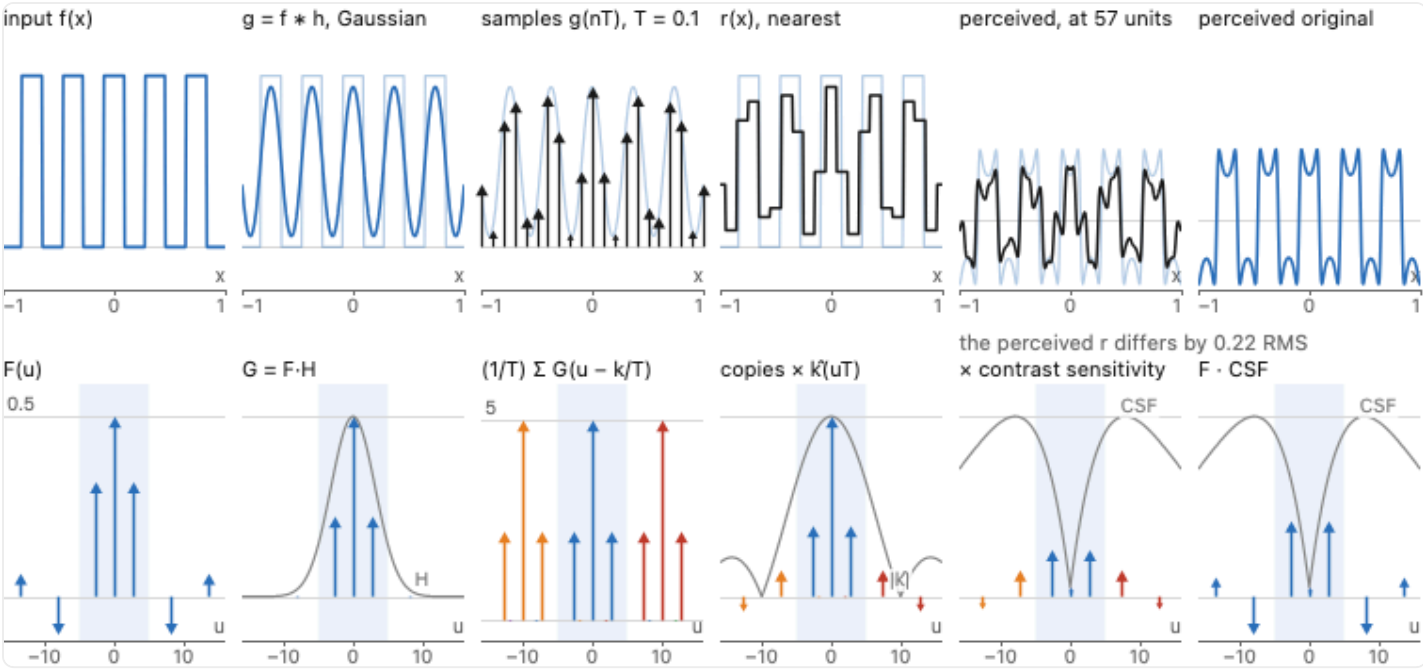


Figure 6.4.9, stage 5 (reproduced). *Perception: the reconstruction filtered by the contrast sensitivity function of Mannos and Sakrison (1974), drawn in gray below, with frequency converted to cycles per degree by the viewing distance (a unit seen from d units away spans 180/(π d) degrees). The faint curve is the input under the same filter. At 57 units, one cycle per unit is about one cycle per degree and the image copy at 7.30 cycles per unit sits at the CSF's peak (0.996); at 300 units it is at 38 cycles per degree and the CSF weighs it 0.075, while the fundamental keeps 0.80, so the steps disappear. The CSF also removes most of the mean (0.05 at zero frequency), since vision responds to contrast. The last column is the original as perceived, the input under the same filter, with how far the perceived reconstruction is from it (RMS over the window shown): the pipeline succeeds when the two last columns look alike, even if the reconstruction itself does not match the input.*

Practical filters: from the box to bicubic and Lanczos

in the book ↗

Downsampling needs a prefilter, and the previous section named the ideal one: a sinc scaled to the output spacing, whose transform is a box that keeps every frequency the coarser grid can hold and removes all the others. The sinc cannot be used as it is. It is infinitely wide, so every output pixel would read the whole image, and its lobes decay so slowly that truncating it changes its response, while its negative lobes overshoot at edges (Figure 6.4.10). Every practical prefilter is therefore a finite kernel that approximates it. The crudest is the box: averaging each $q \times q$ block of pixels into one output pixel, the usual way to halve an image. Its transform is itself a sinc, which keeps only 0.64 of the signal at the new Nyquist frequency and lets through as much as a fifth beyond it. The tent and the Gaussian are smoother, and the cubics and the Lanczos windowed sinc come closest to the ideal. The same kernels, used at the input's spacing rather than the output's, reconstruct a continuous image from its samples, which is how upsampling uses them (Chapter 6.5, Resampling and upsampling), and the trade-offs below hold for both uses.

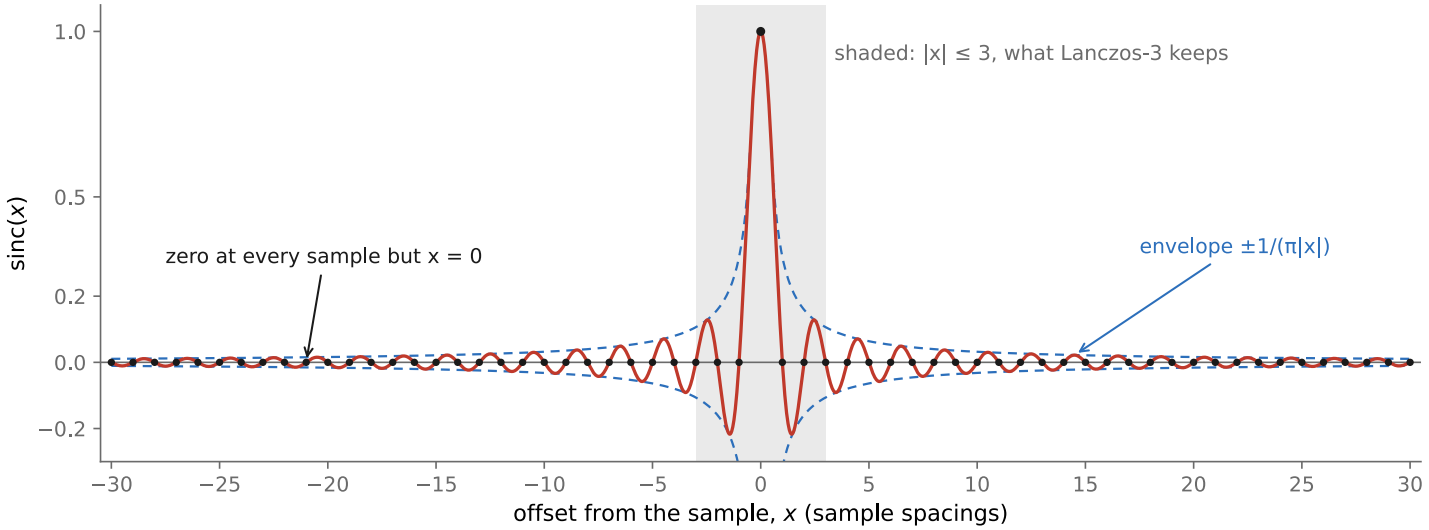


Figure 6.4.10 (reproduced). *The sinc, $\mathrm{sinc}(x) = \sin(\pi x)/(\pi x)$, over 30 sample spacings on each side. It is 1 at its own sample and 0 at every other one (dots), and between them it oscillates with alternating sign, the lobes shrinking only as $1/(\pi|x|)$ (dashed envelope): the lobe near $x = 30$ is still about 1% of the peak. The shaded band, $|x| \le 3$, is the part of it that Lanczos-3 keeps.*

So every practical prefilter or reconstruction kernel is an approximation of sinc with finite support, balancing two competing evils that are clearest in the frequency domain (Figure 6.4.8). A kernel that is too gentle loses medium frequencies (its response rolls off early and the image goes soft), while one too aggressive lets high frequencies alias: it leaks past the grid's limit and admits jaggies and moiré. Every kernel is a point on this blur-versus-alias tradeoff. The two famous families both live on it.

BIG LESSON (L6.16) — A RESAMPLING / ANTI-ALIASING FILTER TRADES THE IDEAL SINC AGAINST COMPACT SUPPORT

Fourier analysis says the *perfect* reconstruction and anti-aliasing filter is the sinc, a perfect box in frequency. But the sinc is infinitely wide and rings when truncated, so no real filter can be it. Every practical kernel (bilinear, bicubic, Lanczos) is a deliberate compromise between approximating that idealized sinc (what Fourier wants) and staying compactly supported (what computation, and ringing, demand): reach wider and you approach sinc; truncate harder and it gets cheaper and less ringy but further from ideal.

BIG LESSON (L6.17) — CHOOSING A RESAMPLING FILTER TRADES PASSBAND DROOP AGAINST STOPBAND LEAK

Read off the frequency response, the same compromise is a tug-of-war between two errors. Passband droop: a too-gentle filter attenuates frequencies *below* Nyquist that the grid *could* have represented, so the image goes soft. Stopband leak: a too-aggressive filter lets frequencies *above* Nyquist through, which alias into jaggies and moiré. A good kernel sits at the sweet spot (flat passband up to Nyquist, sharp rejection beyond) and box, tent, bicubic and Lanczos are just different points on this droop-versus-leak curve.

Lanczos is sinc literally windowed by a wider sinc:

$$k(x) \;=\; \operatorname{sinc}(x)\cdot\operatorname{sinc}\!\left(x/a\right), \qquad |x|<a, \tag{3}$$

over a support of a samples on each side (and zero beyond, with $k(0)=1$). It hugs the ideal closely and looks crisp, at the cost of some ringing near strong edges, the surviving side-lobes of the truncated sinc (Figure 6.4.11).

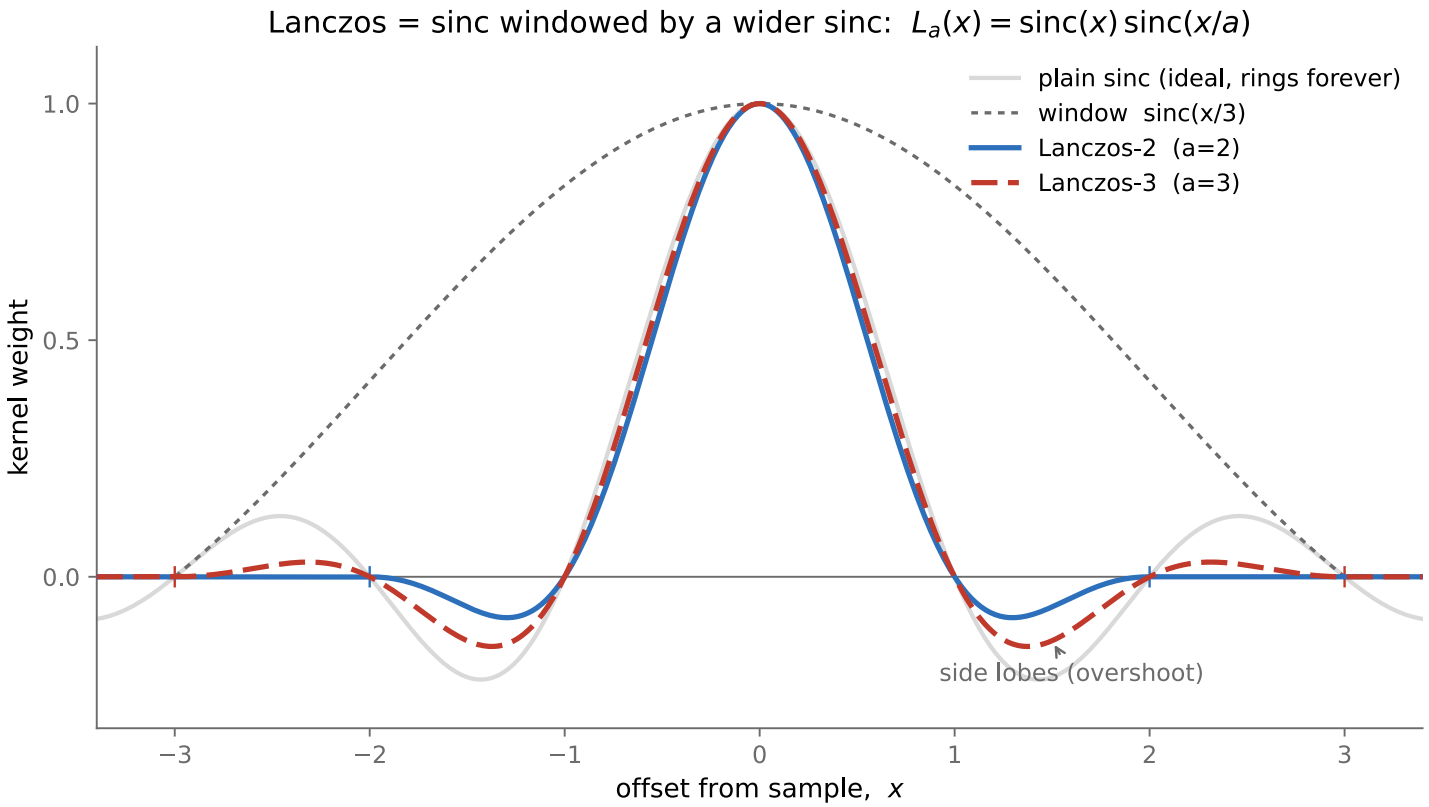


Figure 6.4.11 (reproduced). *The Lanczos kernel* $\operatorname{sinc}(\pi x)\ \operatorname{sinc}(\pi x/a)$ *over its a-sample support (Lanczos-2 and Lanczos-3) against the plain infinite sinc: a tall central lobe flanked by alternating side-lobes, truncated at* $\pm a$*, closest to the ideal box spectrum, but the surviving side-lobes are what produce edge ringing.*

Bicubic uses a piecewise cubic polynomial: smooth, compact (four samples wide in 1-D), the standard default in image editors. It typically carries small negative lobes to the sides, and those are not a bug: they give the reconstruction a built-in mild sharpening (much like the unsharp mask) that makes upscaled images crisper. The cubic kernel is not unique: there is a family. Once you impose the natural requirements (the kernel should be symmetric, should be smooth where the cubic pieces meet, and should sum to one so it preserves brightness), a cubic still has two free parameters left. Mitchell and Netravali (1988) named them B and C (an open-access summary with the kernel formula is the Wikipedia article on Mitchell–Netravali filters); the result is a single piecewise cubic, supported over four samples (two on each side):

$$k_{BC}(x) = \frac{1}{6}\begin{cases} (12 - 9B - 6C)\,|x|^3 + (-18 + 12B + 6C)\,|x|^2 + (6 - 2B), & |x| < 1 \\ (-B - 6C)\,|x|^3 + (6B + 30C)\,|x|^2 + (-12B - 48C)\,|x| + (8B + 24C), & 1 \le |x| < 2 \\ 0, & |x| \ge 2 \end{cases} \quad (4$$

one cubic whose two knobs slide it between sharper and smoother. Three points of the (B, C) plane are the canonical members worth knowing by name:

- $(B, C) = (1, 0)$, the cubic B-spline: the smoothest member, all-positive (no negative lobes), and *approximating* rather than interpolating, so it visibly blurs;
- $(B, C) = (0, \frac{1}{2})$, Catmull–Rom: *interpolating* (it passes exactly through the samples) with pronounced negative lobes, so it looks sharp;
- $(B, C) = (\frac{1}{3}, \frac{1}{3})$, Mitchell–Netravali: the balanced default, sitting between the two and chosen as a useful compromise among visible artifacts.

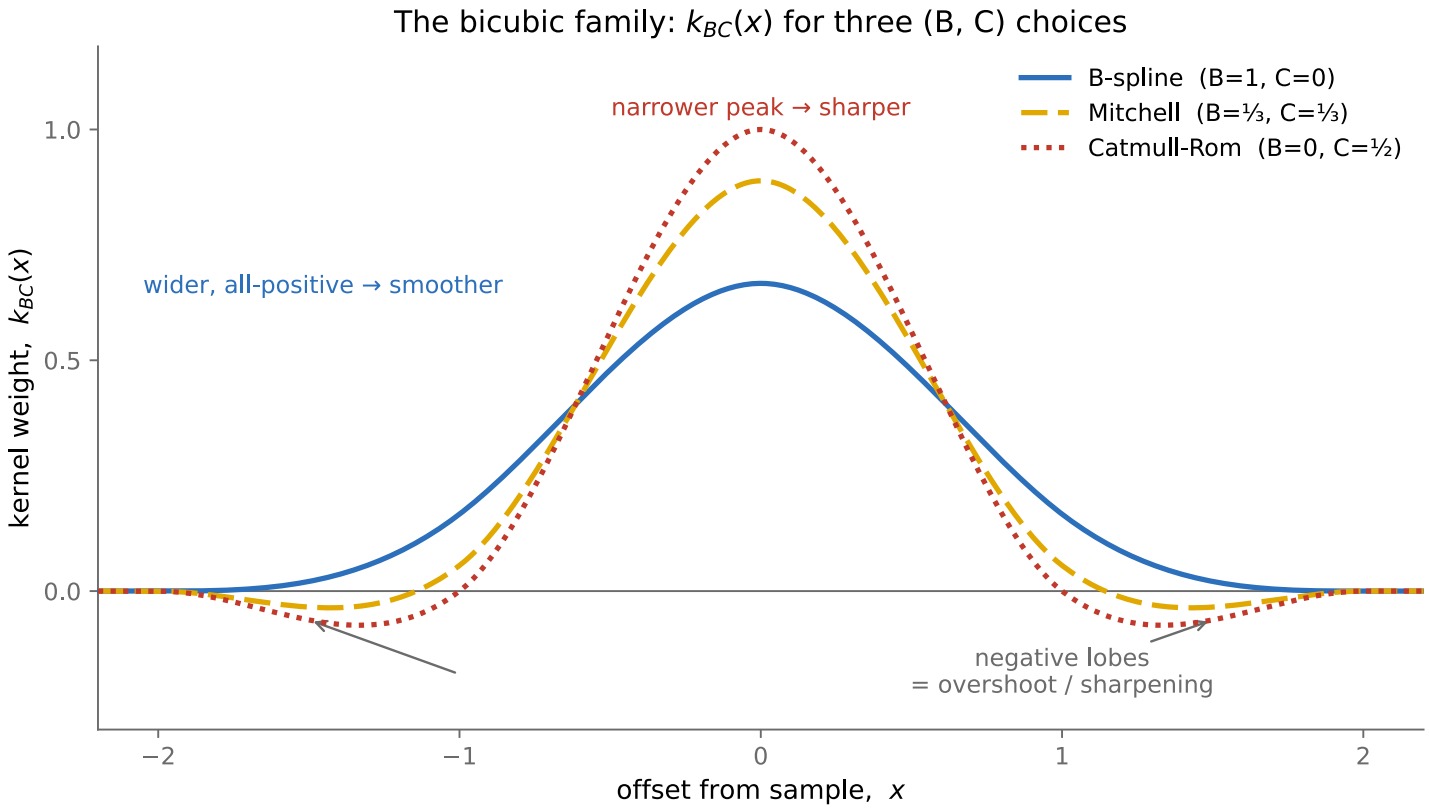


Figure 6.4.12 (reproduced). *The cubic kernel $k_{BC}(x)$ as a one-parameter-pair family, plotted for the three standard members: cubic B-spline $(B, C) = (1, 0)$, all-positive and smooth; Catmull–Rom $(0, \frac{1}{2})$, interpolating with strong negative side-lobes; and Mitchell–Netravali $(\frac{1}{3}, \frac{1}{3})$, the balanced default between them. The deeper the negative lobes, the sharper (and ringier) the reconstruction.*

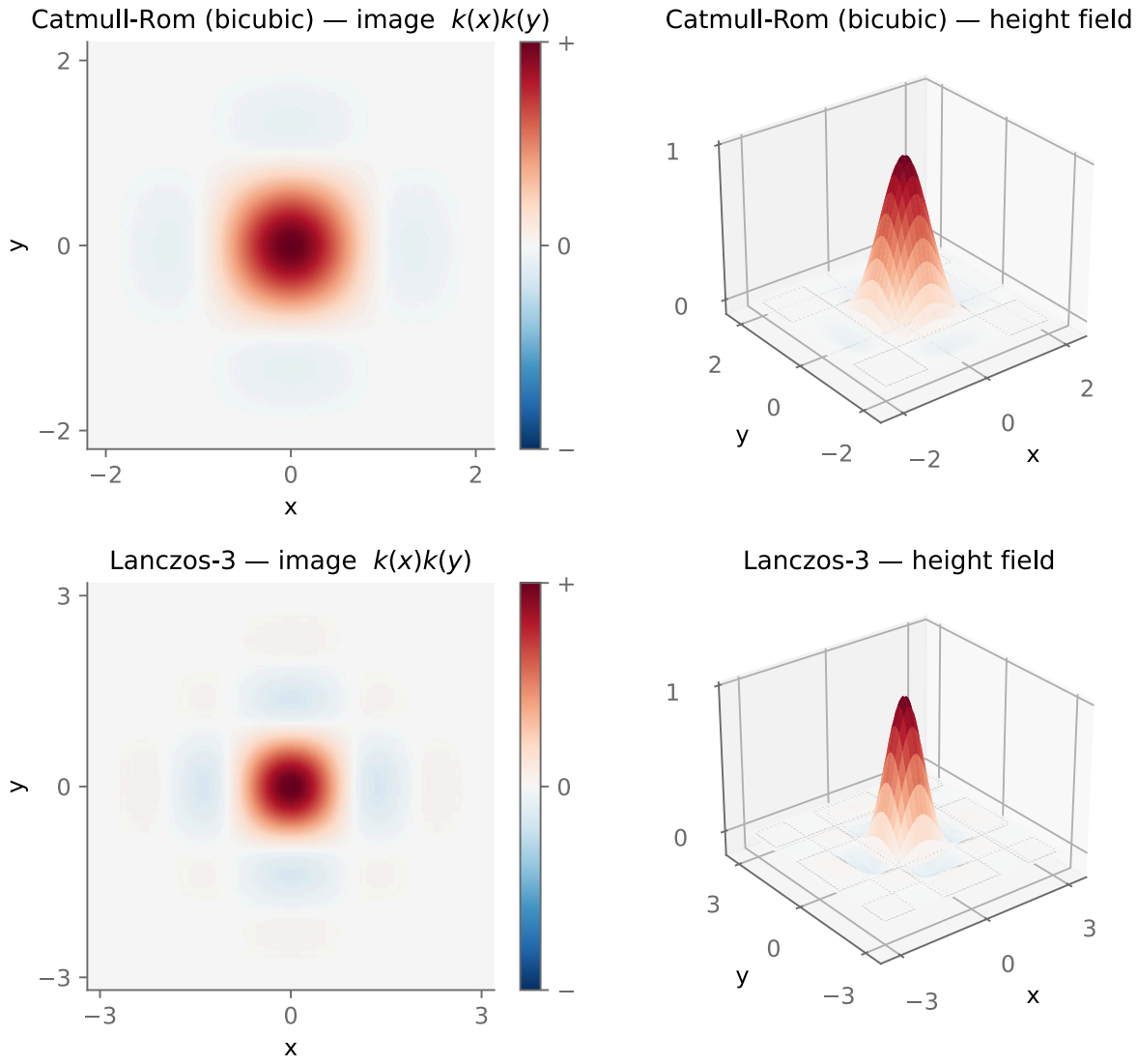


Figure 6.4.13 (reproduced). *The kernels in 2-D, formed as the separable outer product $k(x)\,k(y)$ for Catmull–Rom and Lanczos-3, each shown as a diverging-color image and as a height field. The smooth central mound is ringed by a shallow negative moat, the sharpening side-lobes seen from above.*

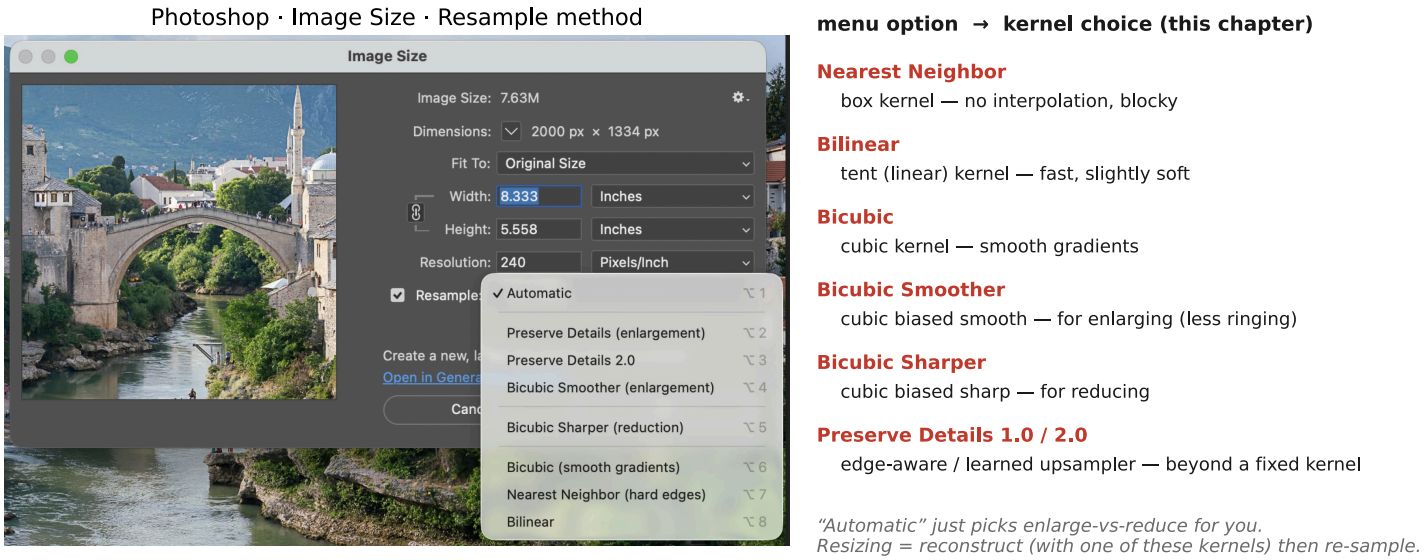
A separable 2-D kernel $k(x)k(y)$ generally treats axes and diagonals differently. The continuous Gaussian is both separable and radially symmetric; truncation on a square only approximates that symmetry. For a rectangular sampling lattice, ideal rectangular-band reconstruction uses a separable product of sincs. It is not radially symmetric. A circular passband has a radial jinc kernel and discards some frequencies allowed near the corners of the rectangular Nyquist region.

BIG LESSON (L6.18) — BICUBIC (AND FRIENDS) ARE CONVOLUTION WITH A FIXED KERNEL, NOT POLYNOMIAL REGRESSION

The cubic-convolution filters used here evaluate a fixed piecewise-cubic kernel at fractional offsets and sum weighted samples. They do not solve a regression for every output pixel. Other methods also called bicubic interpolation use polynomial or spline constructions; the name alone does not identify the algorithm or its downsampling behavior.

Mitchell and Netravali (1988) found their balance by running a perceptual study mapping which (B, C) choices look best, charting where each lands among three failure modes: residual blur (too little negative lobe), ringing (ghost edges from too-strong negative lobes), and anisotropy (a visible preference for the horizontal / vertical / diagonal directions). A broad region of the (B, C) map looks satisfactory, but different points suit different images, which is exactly why Photoshop and friends offer you a *choice* of bicubic ("bicubic smoother" for enlargements, "bicubic sharper" for reductions) rather than one fixed filter. There is no universally best answer, only a tradeoff you tune to the image: a slightly blurry original can afford a sharper kernel, a crisp one cannot.

Real software exposes exactly these kernel choices. Photoshop's *Image Size* resample menu lets you pick Nearest Neighbor, Bilinear, Bicubic, Bicubic Smoother (for enlargements), Bicubic Sharper (for reductions), and Preserve Details: the very box / tent / cubic family, and the smooth-versus-sharp bicubic tradeoff, turned into a dropdown (Figure 6.4.14).



Screenshot of Adobe Photoshop's resampling-method menu — © Adobe Inc., reproduced for illustration / educational commentary.

Figure 6.4.14 (reproduced). *Adobe Photoshop's* Image Size *resample-method dropdown (© Adobe Inc., reproduced for educational commentary), with each menu option mapped to its kernel: Nearest Neighbor → box, Bilinear → tent, Bicubic → cubic, Bicubic Smoother → cubic biased smooth (enlarge), Bicubic Sharper → cubic biased sharp (reduce), Preserve Details → an edge-aware / learned upsampler. The shipping product is this chapter's kernel zoo as a menu.*

A surprising amount of "bicubic" on the internet is wrong

Specify the cubic kernel, coordinate convention, boundary rule, and whether its support widens during reduction. Two libraries can both say “bicubic” and produce different results. Resizing differences can substantially alter generative-model evaluation (Parmar et al.). Scaling is also part of an adversarial attack surface (Gao et al.); the simple nearest-neighbor example below illustrates one mechanism, not the full class of attacks.

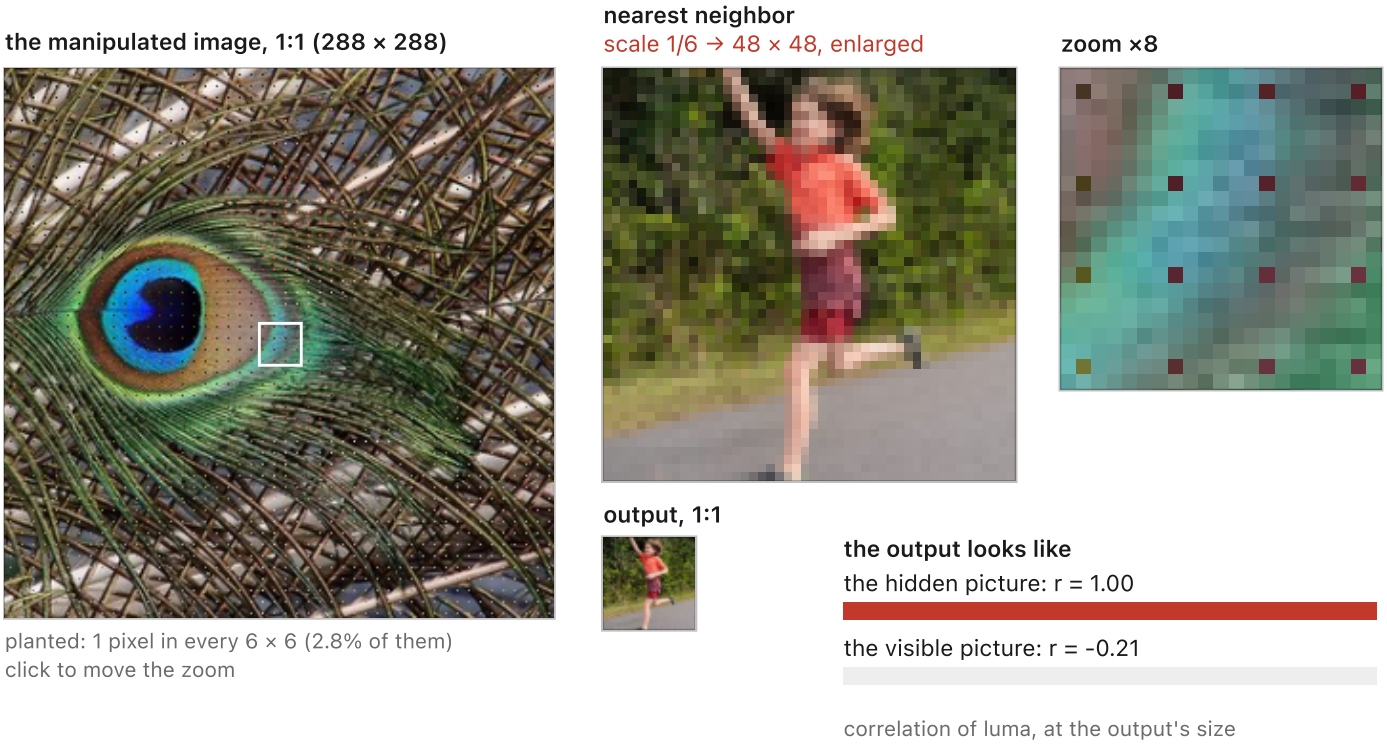


Figure 6.4.15 (reproduced). *An adversarial-resampling demo (interactive). The visible picture on the left carries a second, hidden picture: one pixel of every $k \times k$ block, the one nearest-neighbor downsampling by k reads, is replaced by the hidden picture's pixel, so only $1/k^2$ of the pixels change and the image still reads as the visible one (the zoom shows the planted lattice; click the picture to move it). Downsampled by exactly $1/k$ with nearest neighbor, the output is the hidden picture. Move the reduction slider a few thousandths off k (it snaps onto whole numbers, and a button sets it back to the planted factor) and the visible picture snaps back: the sampling lattice drifts off the planted pixels, and only bands where it still lines up show the hidden picture. Box filtering averages each planted pixel with the $k^2 - 1$ around it, so its weight is $1/k^2$ and the visible picture wins. Bilinear and bicubic without antialiasing read only a few pixels around each sample, as many resize calls do: for odd k the bilinear sample lands exactly on the planted pixel and reveals the hidden picture, for even k it weighs a quarter; widening the kernel by the scale (antialias) defeats the attack. The readout correlates the output with each picture, so which one it shows is a number. This is the simplest image-scaling attack (arXiv:2104.11222, arXiv:2104.08690); variants target other resizers, and a prefilter matched to the scale is the defense. The default cover, the peacock feathers, is the one whose planted pixels show least, here hiding a child running; they show mostly on the smooth eye. Peacock photograph by Bernard Spragg (CC0).*

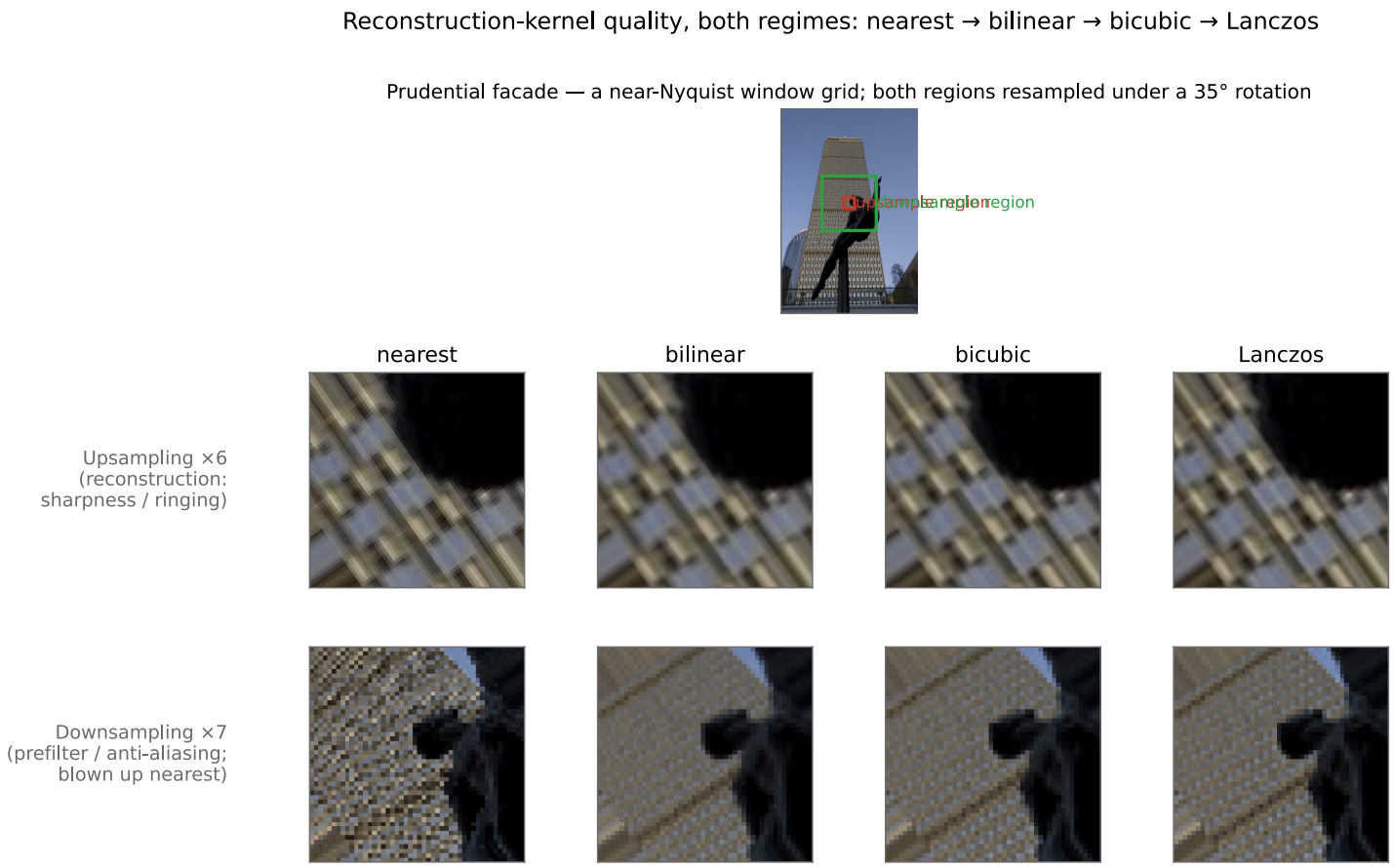


Figure 6.4.16 (reproduced). *The kernel ladder on a real photograph, in both regimes, resampled under a 35° rotation (the general case, not a clean axis-aligned zoom). The subject is a near-Nyquist window grid (the Prudential facade) chosen because fine repeating structure is exactly where the kernel choice bites. Top row, upsampling (reconstruction): magnify a small patch and the kernel reconstructs the in-between values: nearest holds each sample as a hard square (blocky), bilinear is smooth but soft, bicubic (Catmull–Rom) restores edge contrast with its negative side-lobes, Lanczos (a windowed sinc) is the sharpest; on smooth natural content bicubic and Lanczos are close, their negative-lobe ringing visible mainly at high-contrast edges. Bottom row, downsampling (prefilter / anti-aliasing): shrink the grid and the kernel must now act as a prefilter (its footprint widened to the new sample spacing). Nearest does not prefilter: it point-samples and the grid aliases into moiré; bilinear, bicubic and Lanczos average over the discarded detail and stay clean. The kernels were implemented explicitly and evaluated in floating point so their true character (including over/undershoot) is preserved, not clamped away.*

A separable kernel is cheap because a horizontal one-dimensional operation followed by a vertical one replaces a full two-dimensional neighborhood sum. The price can be directional behavior. Rotate a slanted edge while keeping the sampling grid fixed and compare the result: a square product footprint can treat the axis-aligned and diagonal cases differently. The continuous Gaussian shows that separability and radial symmetry can coexist, because the product of its two one-dimensional exponentials depends only on squared radius. A product of sinc kernels instead corresponds to the rectangular frequency region of the sampling grid. Choosing a circular footprint changes that region; it is not merely a more accurate implementation of the same filter.

Resampling and upsampling

Domain operations: moving pixels around

in the book ↗

Recall the three families of image operation from Chapter 8.3, Tone point operations, sorted by what each output pixel is allowed to read. Resampling is the domain family: the underlying continuous image is remapped in position; discrete resampling generally blends its recorded colors. Formally a domain transform applies a function f from the plane to itself (from $\mathbb{R}^2$ to $\mathbb{R}^2$) to the image's coordinates, with the rule that if the point (x, y) carried color c in the input, then $f(x, y)$ carries that same color c in the output. The range (the colors) is untouched; only the domain (the coordinates) is remapped.

That one definition spans a huge amount of ground. When f is a uniform scale it is resizing; a rotation straightens a horizon; an affine or perspective f fixes keystoning or stitches a panorama; and a freely chosen f is a warp: picture the image printed on rubber and stretched. The free-form case, including how a user even *specifies* such an f (and the morphing it unlocks), gets its own Chapter 11.2, Antialiasing for complex transforms. Here we take f as given and concentrate on the hard part shared by *every* domain transform: the new positions almost never land on the input grid, so we must invent values between the samples.

Start with scaling up

OPTIONAL
in the book ↗

Start with the friendliest case, scaling an image up, say by an integer factor k. The output has k times as many pixels along each axis, so most output pixels sit *between* input pixels and have no value of their own: we have too few samples and must fill in the gaps. (Downsampling is the opposite complaint, too *many* input samples crowding into each output pixel, and, surprisingly, it needs a slightly different fix. Hold that thought; the *machinery* is almost identical.) Upsampling is the cleaner place to build intuition, so we begin there.

Figure 6.5.1 follows one small image through every method of this chapter, one stage per method; its first stage is the input itself.

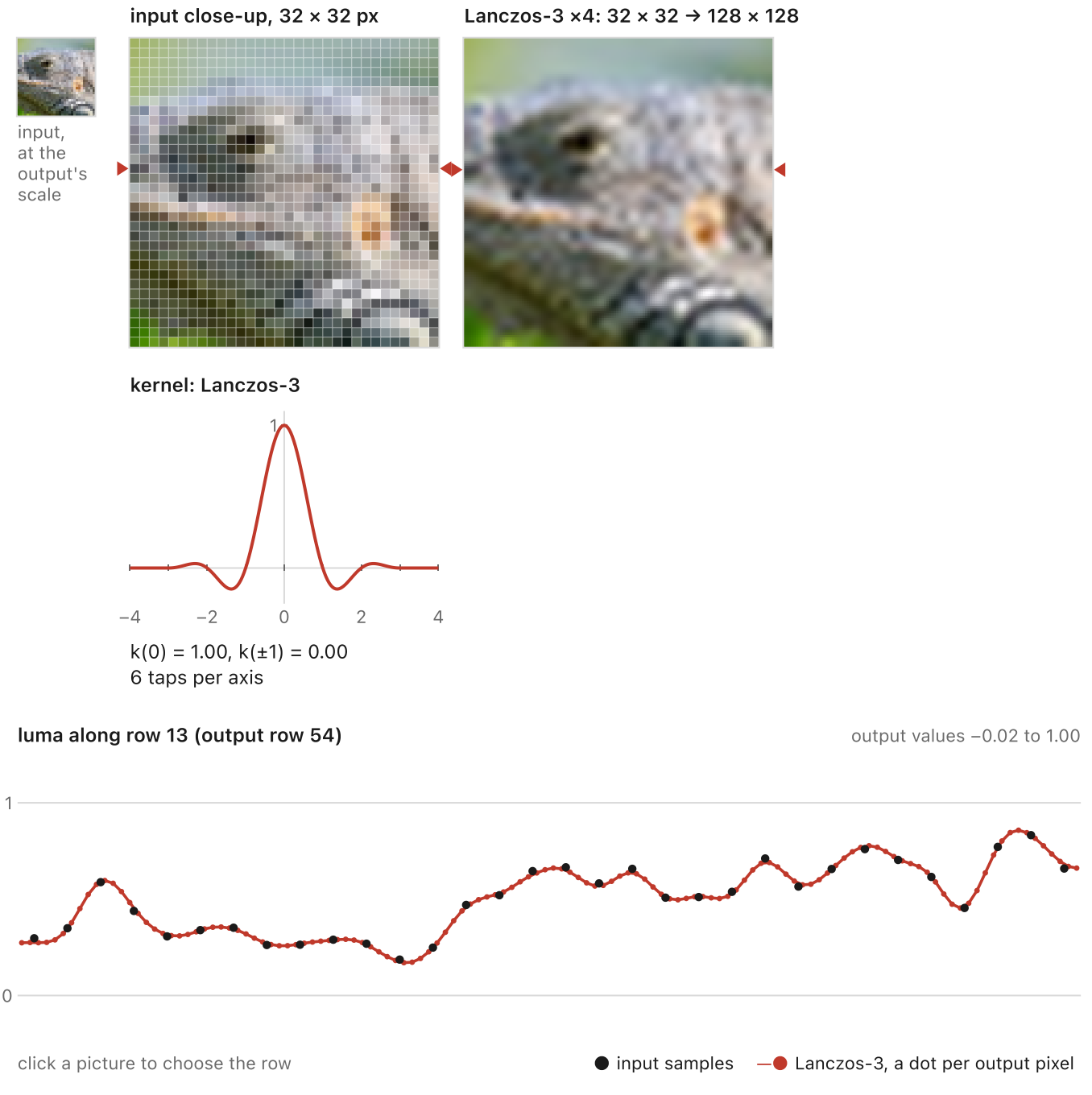


Figure 6.5.1, stage 1 (reproduced). *One small image followed through every upsampling method of this chapter, one stage per method. Here it is the 32 × 32 input, each pixel drawn as a square. The inputs are two photograph crops (the iguana and the Prudential tower, used elsewhere in this chapter) and four synthetic patterns: letters, a 45° edge with thin lines, a checkerboard, and a zone plate whose rings approach the Nyquist frequency at the edges. The plot underneath gives the luma of one row; click a picture to choose the row. Every stage resamples the sRGB code values, as most editors do; the section on resampling and encoding below says what that changes.*

The naive idea, and why it leaves gaps

in the book ↗

Here is the instinct nearly everyone has first: loop over the input pixels, and for each one compute where it lands in the output and write its color there. To scale up by k, the input pixel at (x, y) goes to output location $(k\,x,\ k\,y)$:

$$\text{out}[\,k\,y,\ k\,x\,] \ \leftarrow\ I[\,y,\ x\,]. \tag{5}$$

This is forward mapping: push each input pixel forward to its destination. It is intuitive, and it is broken. Scale up by $k = 2$ and you write output pixels $(0,0), (2,0), (4,0), \ldots$ while every odd-coordinate pixel is never touched: a regular lattice of black holes (Figure 6.5.2, left). The output is mostly gaps, because spreading N input pixels over $k^2 N$ output positions can fill at most N of them: one cell of every 2×2 block, three left empty.

You might try to patch it with an inner loop that paints the little square between four placed pixels. That limps along for an axis-aligned enlargement, but the instant f is a rotation the gaps stop being neat squares: they become slivers at odd angles, and "fill the gap" no longer has an easy shape. Forward mapping fights you. We need a different loop.

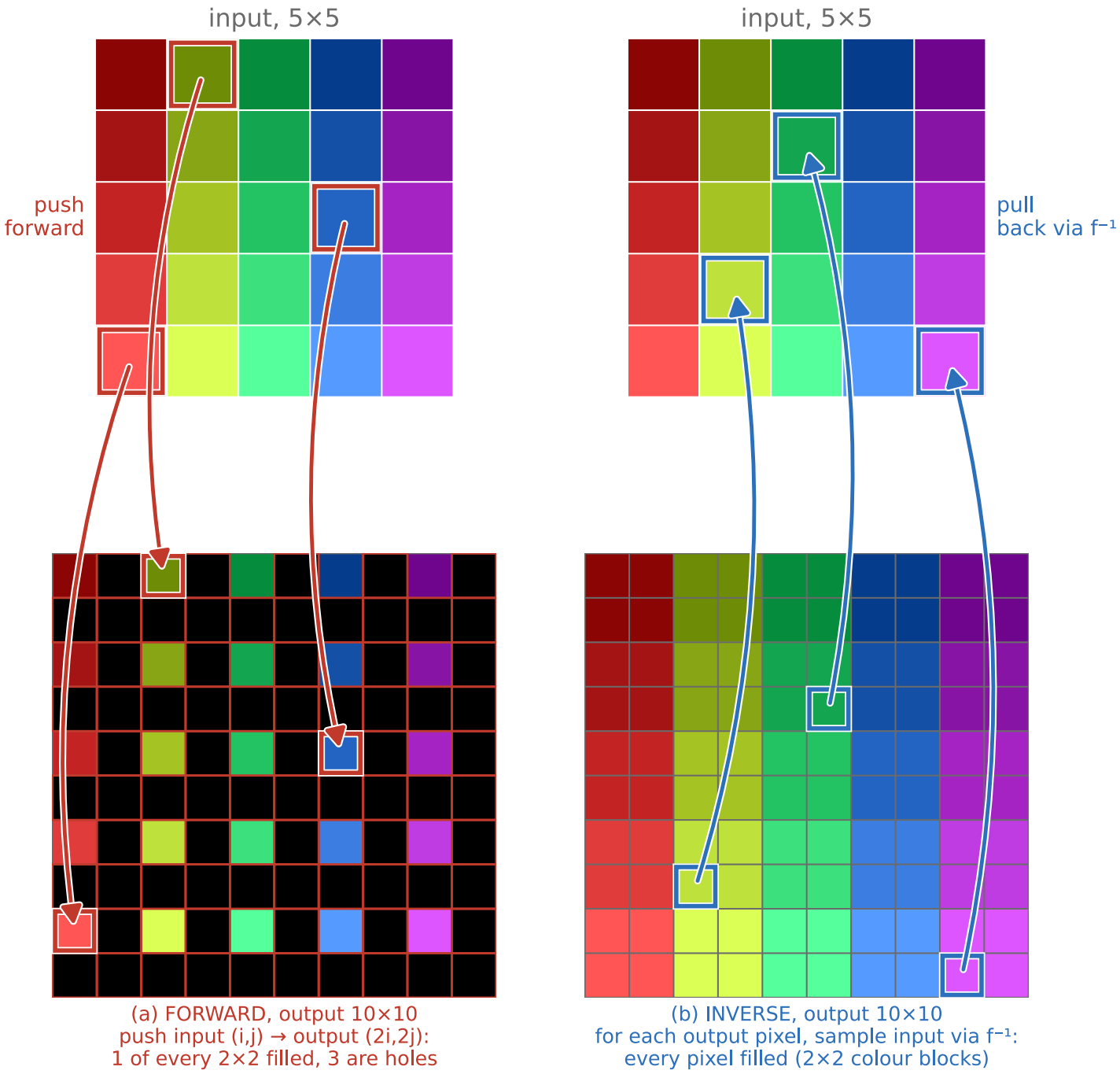


Figure 6.5.2 (reproduced). *Forward vs. inverse mapping on a 2× upscale of a small rainbow grid. Left (forward): loop over input pixels and push each to $(2i, 2j)$; only one cell of every 2×2 output block gets written, leaving the other three as black gaps (a regular lattice of holes; worse under a rotation, where the holes become tilted slivers). Right (inverse): loop over output pixels and, for each, apply f^{-1} to reach back into the input and sample, so every pixel is filled, no gaps. The arrows trace the two routes for a few pixels: forward arrows start at an input pixel and end where it is written, inverse arrows start at an output pixel and end at the input pixel it reads. The whole chapter loops over the output and reaches back into the input.*

The second stage of Figure 6.5.1 runs this naive loop on a real image.

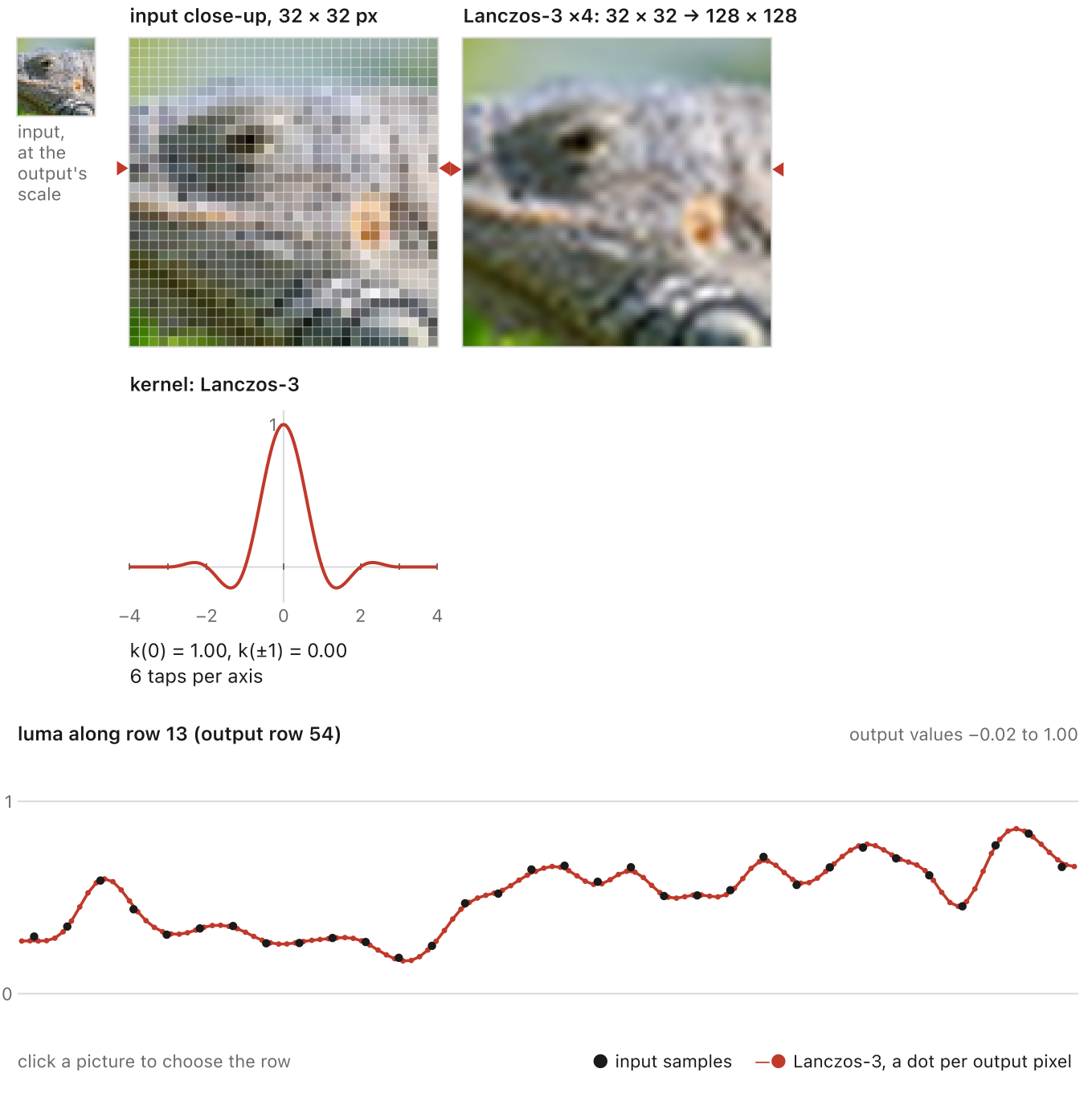


Figure 6.5.1, stage 2 (reproduced). *Naive forward mapping: each input pixel is written to the one output pixel under its center, at k times its coordinates, and nothing else is written. Each $k \times k$ block of the output gets one pixel and the rest stay black, so at ×4 the 128 × 128 output has 1,024 written pixels and 15,360 holes; the title counts them for any factor. In the row plot the red dots are the only values the output row has. From this stage on the output is drawn to scale: an output pixel takes the same room on the screen whatever the factor, so ×2 comes out a quarter as wide as ×8, and the input appears at that same scale at the far left, beside its close-up. "Fill the frame" stretches every factor to the same size, which shows the detail and hides the factor, because every factor samples the same reconstruction, only more or less densely; the row plot shows that density as one dot per output pixel.*

Loop over the output, use the inverse transform

in the book ↗

The fix is to turn the loop inside out. Instead of pushing inputs forward, iterate over the output pixels (all of them, so none can be skipped) and for each one ask *where did this come from in the input?* That means applying the inverse transform f^{-1} to the output coordinate and reading the input there:

$$\text{out}[\,y,\ x\,] \ \leftarrow\ I\big[\,f^{-1}(x, y)\,\big]. \tag{6}$$

For a scale-up by k the inverse divides by k, so output pixel (x, y) reads the input at $(x/k,\ y/k)$. Now every output pixel receives exactly one value and there are no gaps by construction. This is inverse mapping: loop over output pixels and use the inverse transform when it exists. An occlusion, a fold, or a location outside the source also needs an explicit visibility or boundary rule. It is the same "where-from, not where-to" reflex as the flip in convolution (to fill a destination, reach back into the source), and it is the single most important habit in this chapter. (Figure 6.5.2 sets the two loops side by side.)

> 💡 **BIG LESSON (L6.19) — RESAMPLE BY GATHERING, RARELY BY SCATTERING**
>
> Loop over output pixels and pull back through f^{-1} to find where each one reads from. Gathering fills every output exactly once; scattering pushes input pixels forward and leaves holes whose shape depends on the transform, a lattice of gaps under a scale and tilted slivers under a rotation. The same reflex governs warping, morphing, texture mapping, and flow-based reprojection later. Scattering earns its place only when the forward map is the thing you actually have, as in splatting a point cloud, and then it needs an explicit accumulation and normalization pass.

There is one immediate wrinkle, and it is the real subject of everything that follows. Inverse mapping is clean, but $f^{-1}(x, y)$ lands at a fractional coordinate: $(x/k, y/k)$ is rarely an integer, and under a rotation it essentially never is. There is no pixel sitting at $(1.3, 4.7)$. So the genuine content of resampling is the question we have been circling: given the input samples, what value should we report at a position *between* them? Every method below is one answer.

Nearest neighbor

in the book ↗

Round the fractional coordinate to the nearest sample and copy its value. At 1.3, return $I[1]$; at 1.8, return $I[2]$. Truncating the coordinate instead gives a different, biased convention. Nearest neighbor is appropriate for categorical labels and for deliberately showing the original pixels, but looks blocky on photographs.

It works and it is fast, but the result is conspicuously blocky: a whole range of fractional positions all snap to the same input pixel, so an upscaled image looks like fat rectangular tiles (Figure 6.4.16, nearest panel), each input pixel becoming a hard $k \times k$ block. Occasionally that is precisely what you want (pixel art, or a debug view where you want to see the literal samples), but as a general-purpose resize it is the floor, not the goal. As the challenge for nearest neighbor: try upscaling a face by $8\times$ this way and watch it turn to mosaic. We do much better by *interpolating* between samples instead of snapping to one.

The third stage of Figure 6.5.1 replaces that scatter with a gather and rounds.

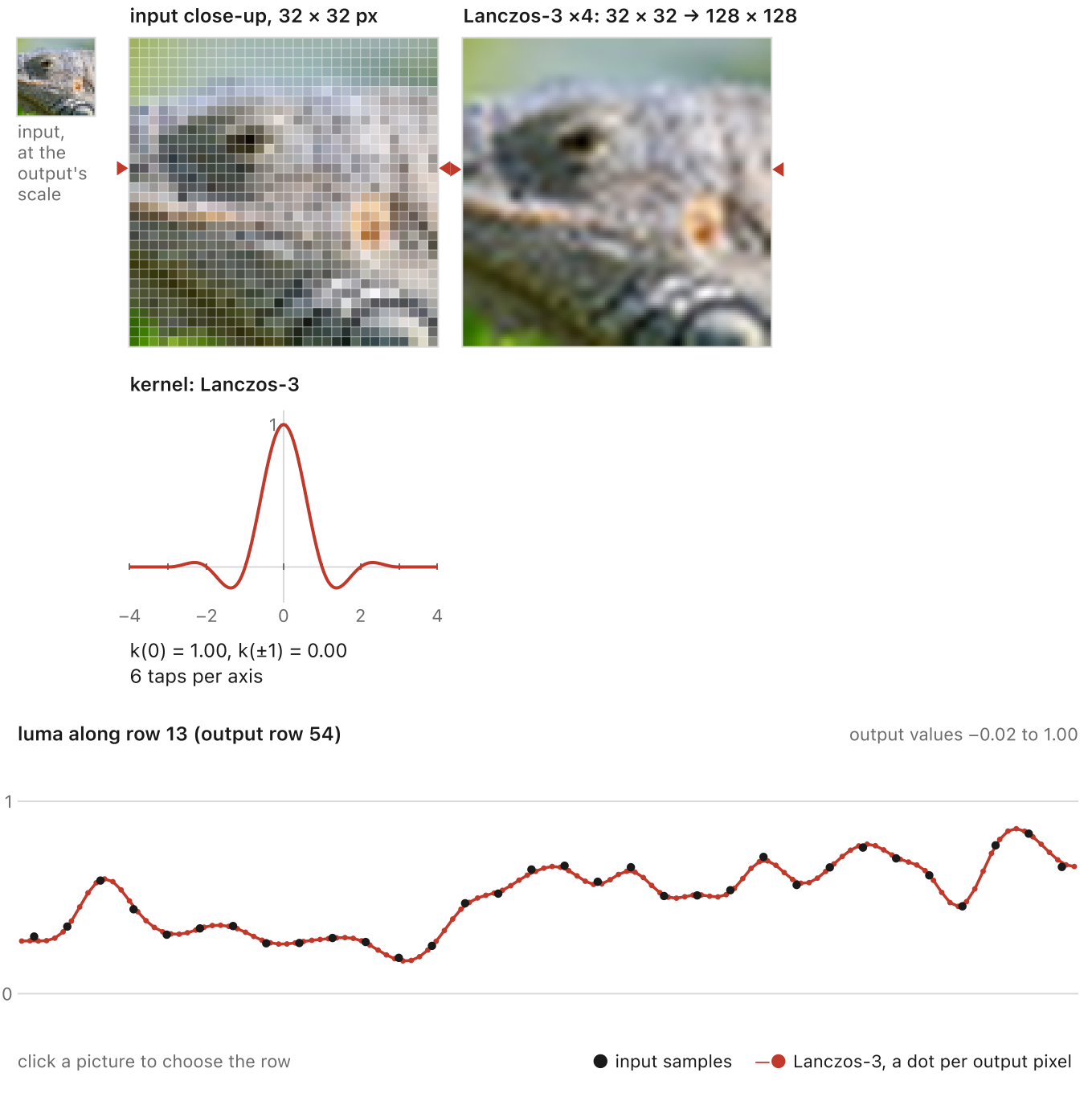


Figure 6.5.1, stage 3 (reproduced). *Nearest neighbor, by inverse mapping: every output pixel reads the input sample nearest to the position it maps back to, so each input pixel becomes a $k \times k$ block and nothing is left empty. The kernel plot shows the box this amounts to; the row plot is a staircase through the samples.*

Linear interpolation, in 1-D first

in the book ↗

Drop to one dimension, where the idea is cleanest. An image row is a 1-D array of values, and we want the value at 1.3, between the samples at 1 and 2. The natural move is to blend the two neighbors in proportion to how close the query sits to each. At 1.3 we are 0.3 of the way from 1 toward 2, so

$$I[1.3] \;=\; 0.7 \cdot I[1] \;+\; 0.3 \cdot I[2]. \tag{7}$$

This is linear interpolation, and there is one detail that trips up everyone at first: the weights run *opposite* to the distances. The neighbor you are 0.3 away from gets weight 0.7, not 0.3, because you want weight 1 when you sit exactly *on* a sample and 0 when you are a full step away. Say it back in words: take a weighted average of the two bracketing samples, weighting each by *one minus* the distance to it. Geometrically, the reconstructed signal is just the straight line connecting consecutive samples (Figure 6.5.3).

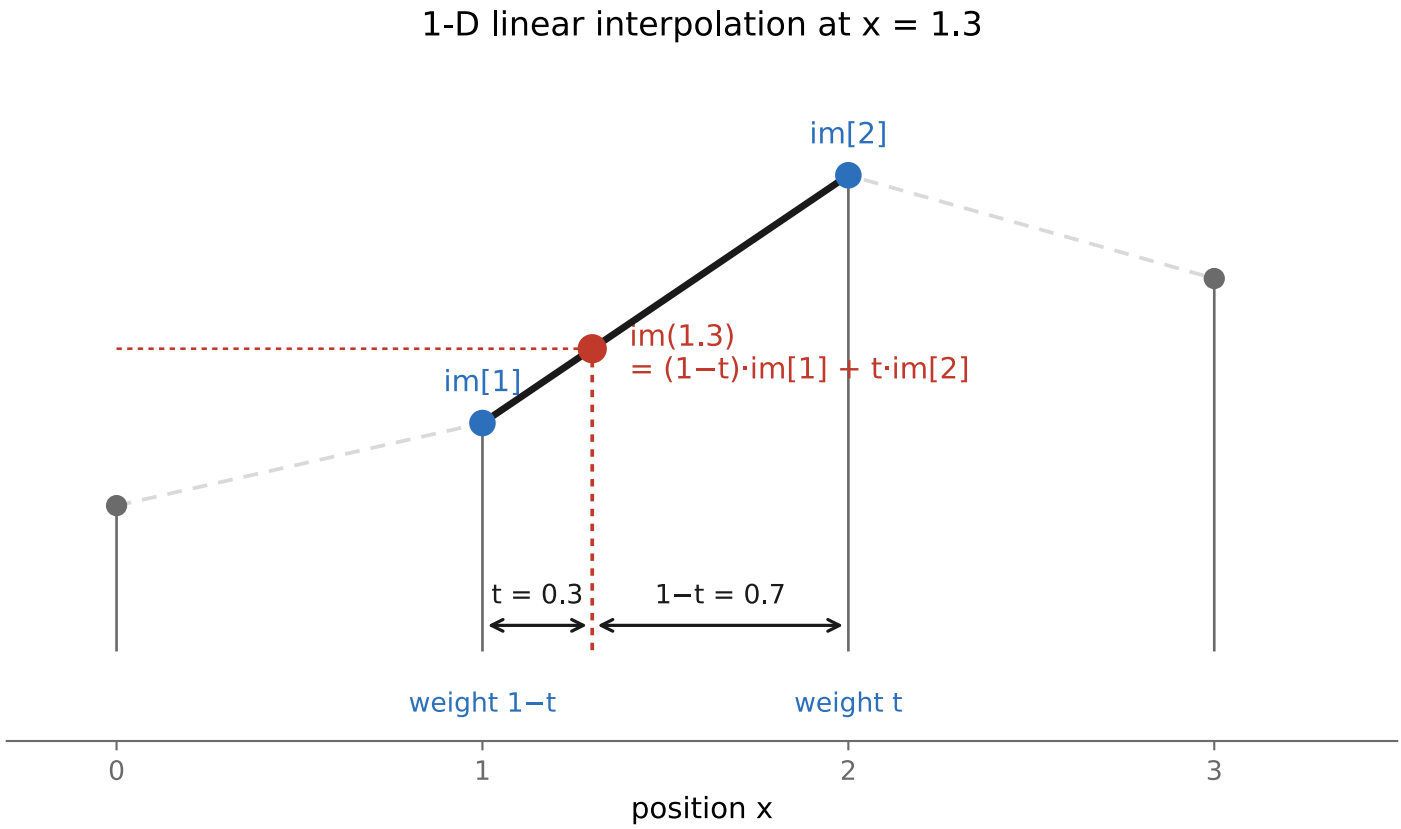


Figure 6.5.3 (reproduced). *1-D linear interpolation. To reconstruct the value at the fractional position* 1.3 *between samples* $I[1]$ *and* $I[2]$*, blend them in proportion to* one minus *the distance to each:* $I[1.3] = 0.7 \cdot I[1] + 0.3 \cdot I[2]$*. The reconstructed signal is the straight line joining consecutive samples; a neighbor's weight is* 1 *when you sit on it and falls linearly to* 0 *a full sample away, the opposite of the raw distance.*

Bilinear, in 2-D

in the book ↗

In two dimensions we now have *four* surrounding samples (the corners of the unit square the query falls inside) and two fractional coordinates to interpolate against. The clean construction is to do 1-D linear interpolation twice:

first along x, then along y (Figure 6.5.4). Blend the top pair of pixels horizontally to get one value, blend the bottom pair horizontally to get another, then blend those two results *vertically* with the y fraction. The four corner weights come out as products of the x- and y-blend factors, which is exactly why this is bilinear: linear in each axis, but their *product*, not a single plane through all four points.

That last point is a real subtlety. Four arbitrary corner values do not in general lie on a flat plane, so a "purely linear" 2-D surface through all four simply does not exist. Bilinear sidesteps this by being linear *separately* in each direction (the product of two tents), which is well-defined and cheap. The resulting surface is doubly ruled: dead straight along every horizontal and every vertical line, yet curved overall, twisting into a saddle when the four corners disagree (Figure 6.5.5). The only way to get a *truly* planar piecewise-linear reconstruction is to triangulate: cut each square into two triangles and interpolate within each, since three corners *do* fix a plane.

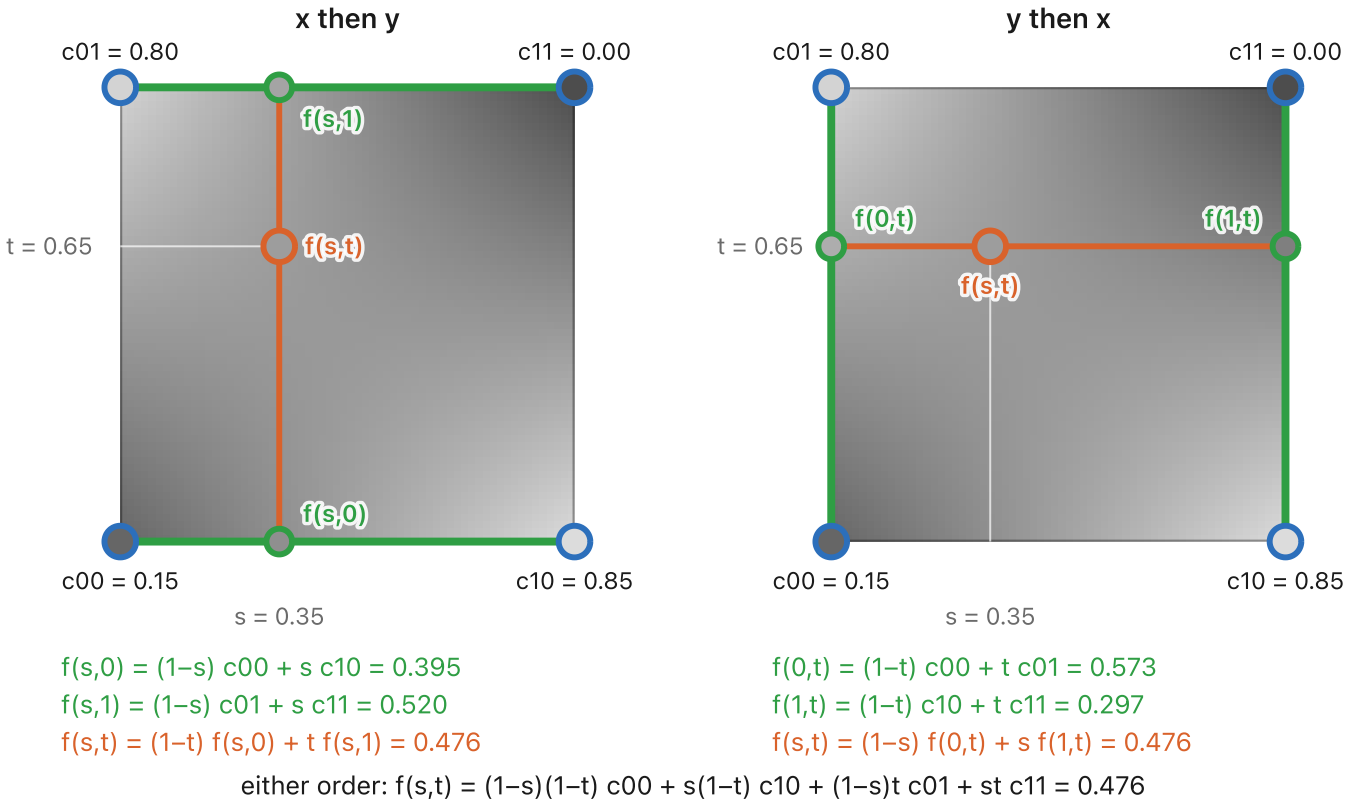


Figure 6.5.4 (reproduced). *Bilinear interpolation as two 1-D passes, in both orders (interactive). The query (s, t) falls inside the unit square spanned by its four nearest input pixels, c_{00} to c_{11}; each square is shaded with the interpolated values. On the left we interpolate along x first, on the bottom and top edges (green), $f(s, 0) = (1 - s)\, c_{00} + s\, c_{10}$ and $f(s, 1) = (1 - s)\, c_{01} + s\, c_{11}$, then along y between those two (orange). On the right we interpolate along y first, on the left and right edges, $f(0, t)$ and $f(1, t)$, then along x. The intermediate values differ, the result does not: both orders expand to $(1 - s)(1 - t)\, c_{00} + s(1 - t)\, c_{10} + (1 - s)t\, c_{01} + st\, c_{11}$, four weights that are products of the x- and y-blend factors, linear in each axis, hence* bilinear. *Drag the query point in either square, or set the corner values; the defaults are those of the height field that follows.*

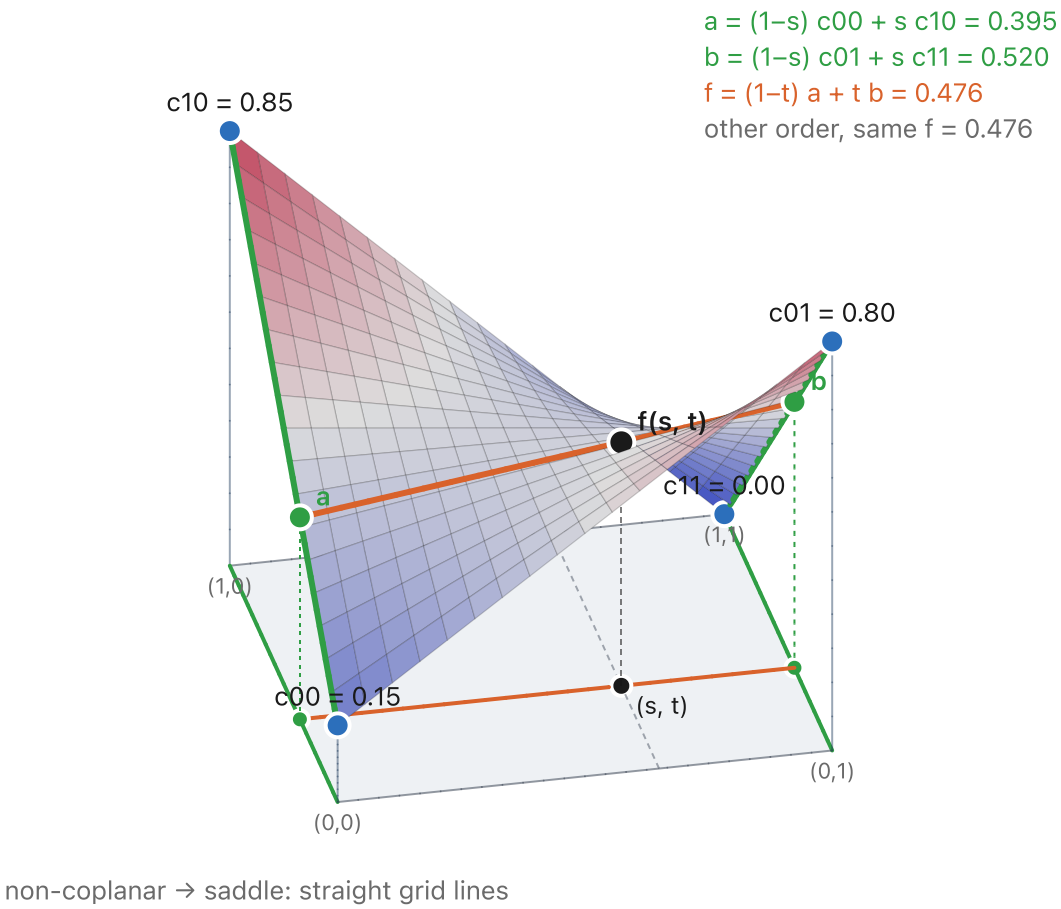


Figure 6.5.5 (reproduced). *The bilinear surface over one unit cell, drawn as a height field whose four corner samples are the controlling heights. Every line of constant x and every line of constant y is dead straight (the sheet is* ruled *in both directions), yet the surface as a whole bows into a saddle, proof that bilinear is linear in each axis separately but not one flat plane through the four corners. At a query point (s, t), marked with its construction lines on the unit square under the sheet, the figure draws the two 1-D passes that compute it: interpolate along x on the two edges $t = 0$ and $t = 1$, then along y between the results, or along y first and x second; both orders reach the same point of the sheet. Drag to orbit; toggles add the two diagonals, which curve unless the corners are coplanar, and shading.*

The next stage of Figure 6.5.1 applies bilinear interpolation to the whole image.

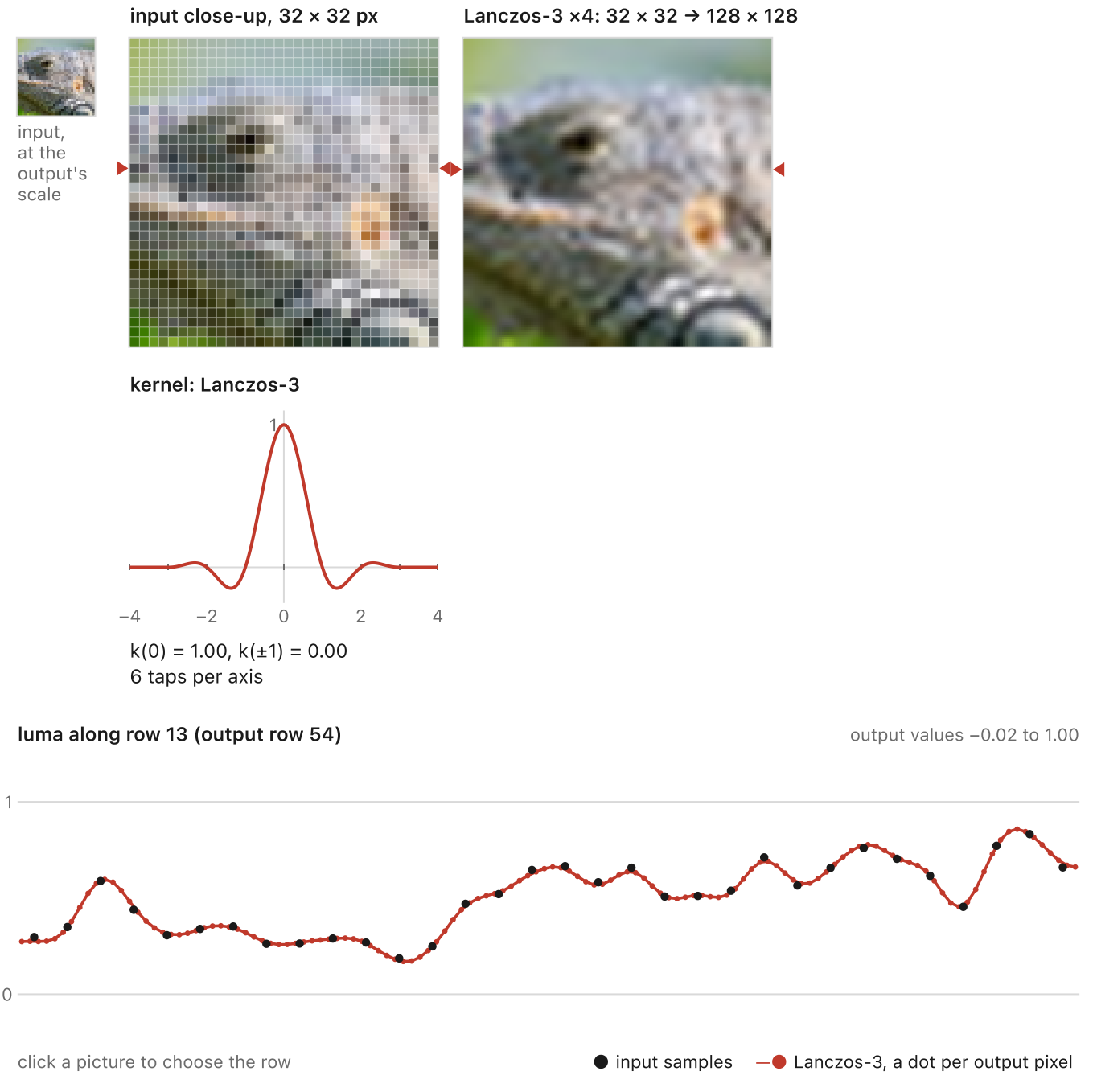


Figure 6.5.1, stage 4 (reproduced). *Bilinear: each output pixel blends its four nearest samples with the tent kernel. The blocks are gone and the edges are soft. In the row plot the curve joins the samples with straight segments and kinks at each.*

Sidebar 3: which diagonal to triangulate on?

If you do triangulate, you must choose *which* diagonal to cut each square along, and the two choices differ near edges. A naive splitter always cuts the same way, secretly preferring one diagonal and

producing a faint directional bias. The smarter trick picks, per square, the diagonal joining the two most similar corner values, cutting *along* an edge rather than across it, so a sharp boundary isn't smeared. Bilinear's quiet advantage is that it has *no* diagonal to choose: it needs no branch and favors no direction.

Choosing the diagonal of each cell from the data is data-dependent triangulation, introduced for image reconstruction by Yu, Morse and Sederberg (2001); the per-cell rule of the sidebar is the pixel-level version of Su and Willis (2004), and the fifth stage of Figure 6.5.1 runs it after bilinear.

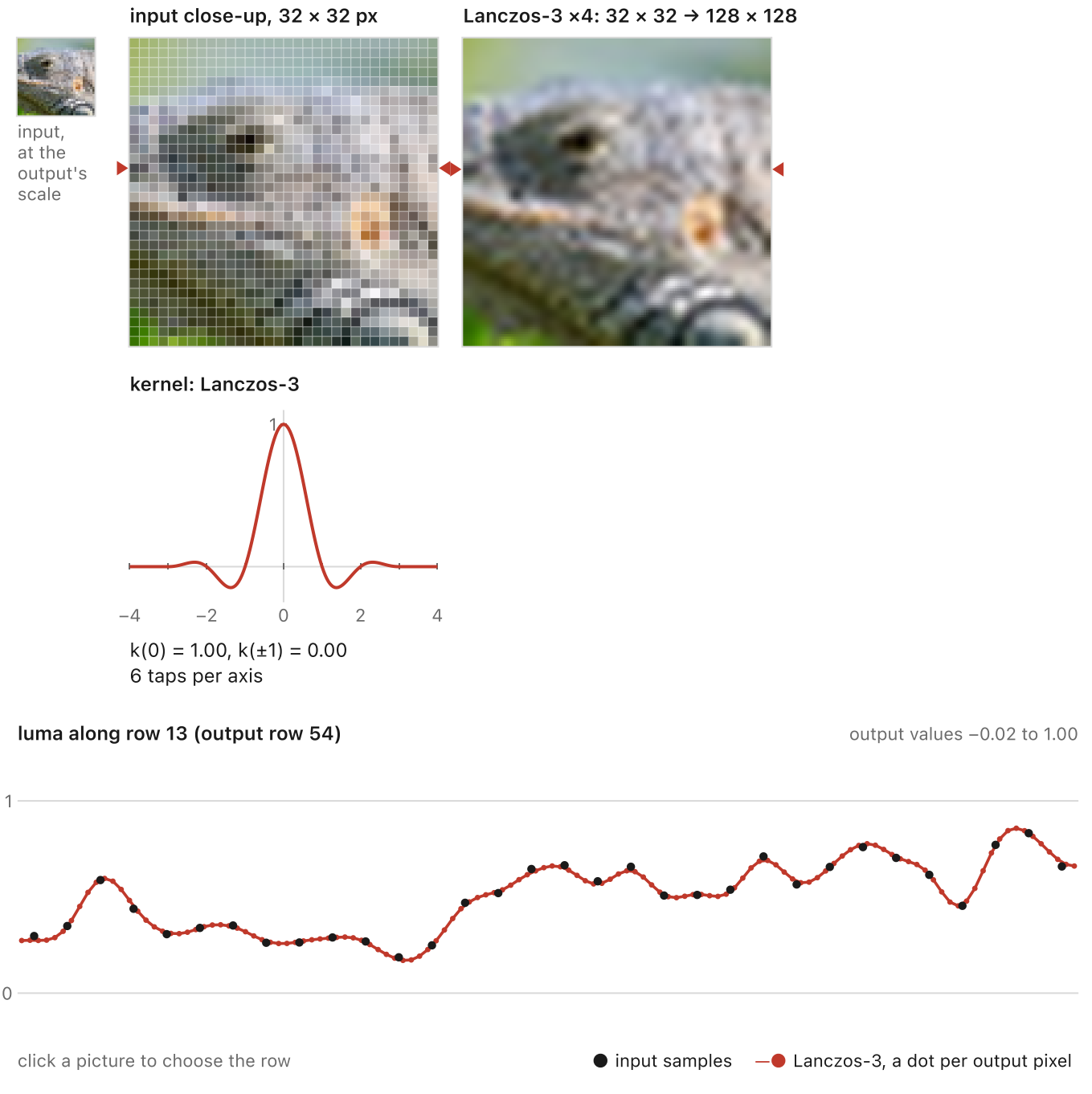


Figure 6.5.1, stage 5 (reproduced). *Data-dependent triangles: each cell of four samples, a b above c d, is cut along the diagonal whose two end samples differ less in luma, and each triangle is interpolated linearly. The counts under the input give how many cells took each cut; the toggle draws the chosen diagonals on the input, leaving out the ties, which are cut a–d. On the 45° edge every cell the edge crosses is cut along it, and the edge stays sharper than with bilinear. A one-pixel line at 45° is the method's own failure: in every cell it crosses, both diagonals join equal samples (line to line, background to background), so the tie-break decides. The line that runs along the tie-break diagonal comes out smooth, and the one that runs across it breaks into beads. Method: Su and Willis (2004), after Yu, Morse and Sederberg (2001).*

Exercise

Show that interpolating y first, then x gives exactly the same result as x first, then y. (Write both as weighted sums of the four corners and compare the coefficients, the order of the two separable passes does not matter.)

Putting the two halves together (the inverse-mapping loop and the bilinear sample) gives the whole upsampler. Loop over output pixels, send each back through f^{-1} to a fractional input location, split that into its integer cell and fractional offsets, and blend the four corners:

Algorithm 1

```
for each output pixel (x, y):
    (u, v) = f⁻¹(x, y)                  # fractional input location
    x0, y0 = floor(u), floor(v)
    s, t = u − x0, v − y0               # fractional offsets in [0, 1)
    top = (1 − s)·I[x0, y0]   + s·I[x0+1, y0]
    bot = (1 − s)·I[x0, y0+1] + s·I[x0+1, y0+1]
    out[x, y] = (1 − t)·top + t·bot     # blend the two rows vertically
```

Read it back: the two horizontal blends collapse the top and bottom edges of the cell, and the final vertical blend combines them: 1-D linear interpolation done twice. Bilinear is the workhorse default: cheap, no blocky tiles, smooth enough for most resizing, and the reconstruction you get for free from a graphics-hardware texture lookup (Figure 6.4.16, bilinear panel). But it has a ceiling. The reconstructed surface has *creases* (it is only continuous, not smooth, kinking at every sample) and on a strong upscale it reads a touch soft. To push past it we need to look at more than four neighbors, and the "interpolate between neighbors" story does not generalize gracefully. We need a better lens, but first a word on what any of this *costs*.

Sidebar 4: a practical note on cost

For axis-aligned resizing, the horizontal coordinates and weights repeat across rows, and the vertical weights repeat across columns. Precompute both weight tables and apply two 1-D passes. A rational scale has repeating fractional phases away from boundaries. An arbitrary warp does not generally factor into two image-wide passes, even when its local kernel is separable. Normalize weights according to the selected boundary rule; dropping out-of-bounds taps and renormalizing is one rule, not a substitute for choosing one.

The convolution perspective

in the book ↗

Here is the more general formulation. So far we have described reconstruction as *interpolating between neighbors*. That picture is fine for linear interpolation and for upsampling, but it has two failings: it does not extend cleanly to

fancier reconstructions, and it does not extend to downsampling at all. Since this whole chapter is fundamentally about (re)sampling, the Fourier-flavored view from Chapter 6.4, Sampling, downsampling, and aliasing has much better insight to offer. In practice that view means one thing: reconstruction is convolution.

Concretely, the output value at a fractional location is a weighted sum of nearby input pixels, with the weights read off a smooth analytical kernel k centered at the desired (non-integer) location. Slide a continuous kernel so its center sits exactly on the query point $f^{-1}(x, y)$, evaluate the kernel at each surrounding *integer* input pixel to get that pixel's weight, and sum. The general resampling formula is

$$\mathrm{out}(x, y) \;=\; \sum_{x'} I(x') \cdot k\big(f^{-1}(x, y) - x'\big), \tag{8}$$

summing over input pixels x' in the neighborhood, the kernel evaluated at the offset between the query and each pixel. In words: to find an output pixel, inverse-transform it to a fractional input location, drop a smooth kernel centered there, and accumulate kernel-weight times pixel-value over the nearby samples. It is convolution, but where convolution in the earlier chapter lived purely on the integer grid, here the kernel is a *continuous function* we can evaluate at any fractional offset (ask for $k(-0.7)$ and it answers), because the center is non-integer.

The resampler needs weights for fractional offsets. A formula for a real-valued argument supplies them, as do a lookup table or a bank of filters for quantized fractional phases. A list of integer-offset convolution taps alone does not specify interpolation between taps. Nearest neighbor's box kernel is discontinuous, so continuity is not a requirement.

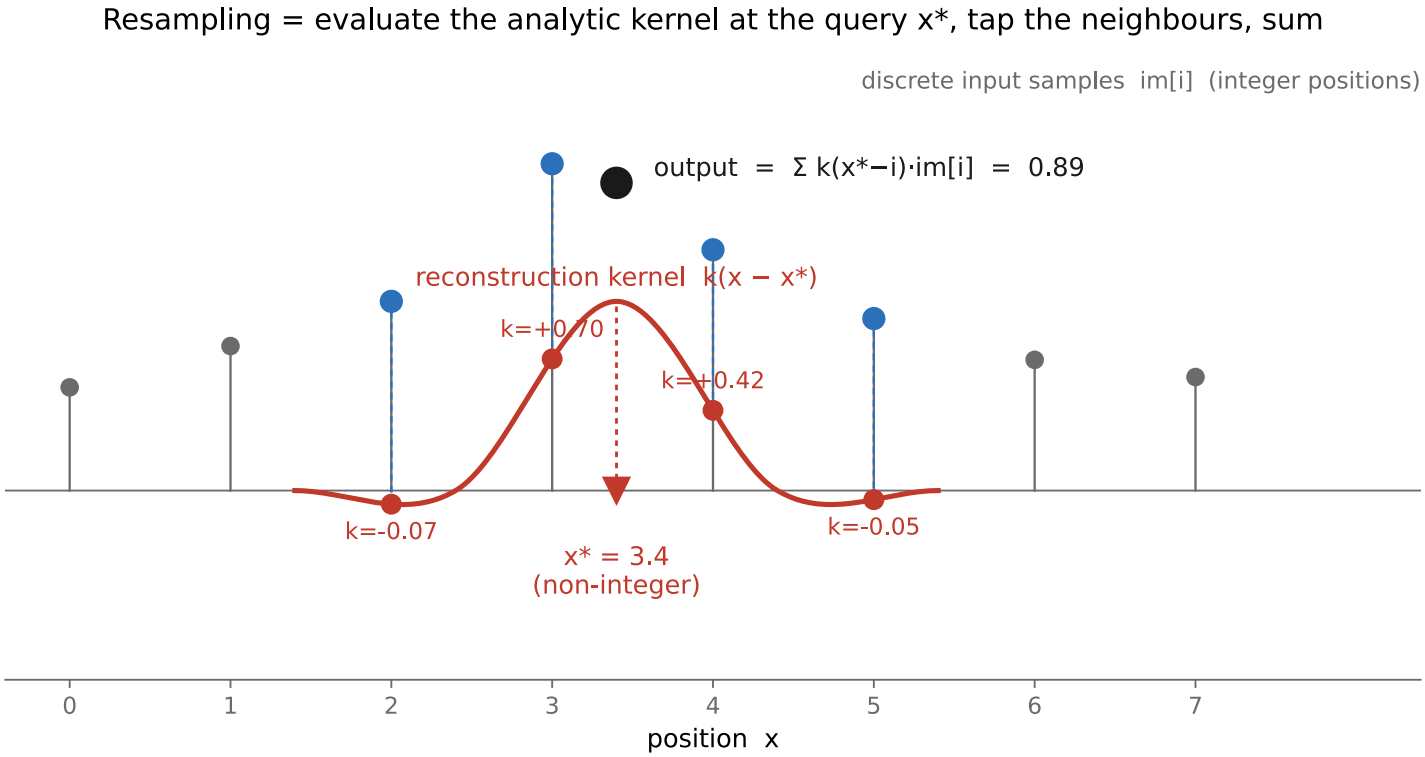


Figure 6.5.6 (reproduced). *Why the kernel must be a continuous function. A row of discrete input samples sits on the integer grid; a smooth kernel $k(x)$ is centered at a non-integer query location x^* between two of them, and the output is the sum of each sample times the kernel's height* at that sample's offset. *Because the center is fractional, those offsets are fractional too, so the kernel has to be a closed-form curve readable at any real argument, not a fixed stencil of integer taps.*

Bilinear is already a special case. The linear-interpolation weights 0.7 and 0.3 are exactly the values of a triangular (tent) kernel, one that is 1 at its center and falls linearly to 0 one sample away on each side, evaluated at the offsets to the two neighbors. So "interpolate linearly" and "convolve with a tent" are the *same operation*, seen two ways. Nearest neighbor is the same story with a box kernel (a flat 1 over a one-pixel window). Once reconstruction is "convolve with a kernel," choosing a better reconstruction is just choosing a better kernel.

Figure 6.5.7 draws these kernels as 2-D kernels $k(x)\,k(y)$, each with its profile, an upsampling made with it, and its Fourier transform, starting with the box and the tent.

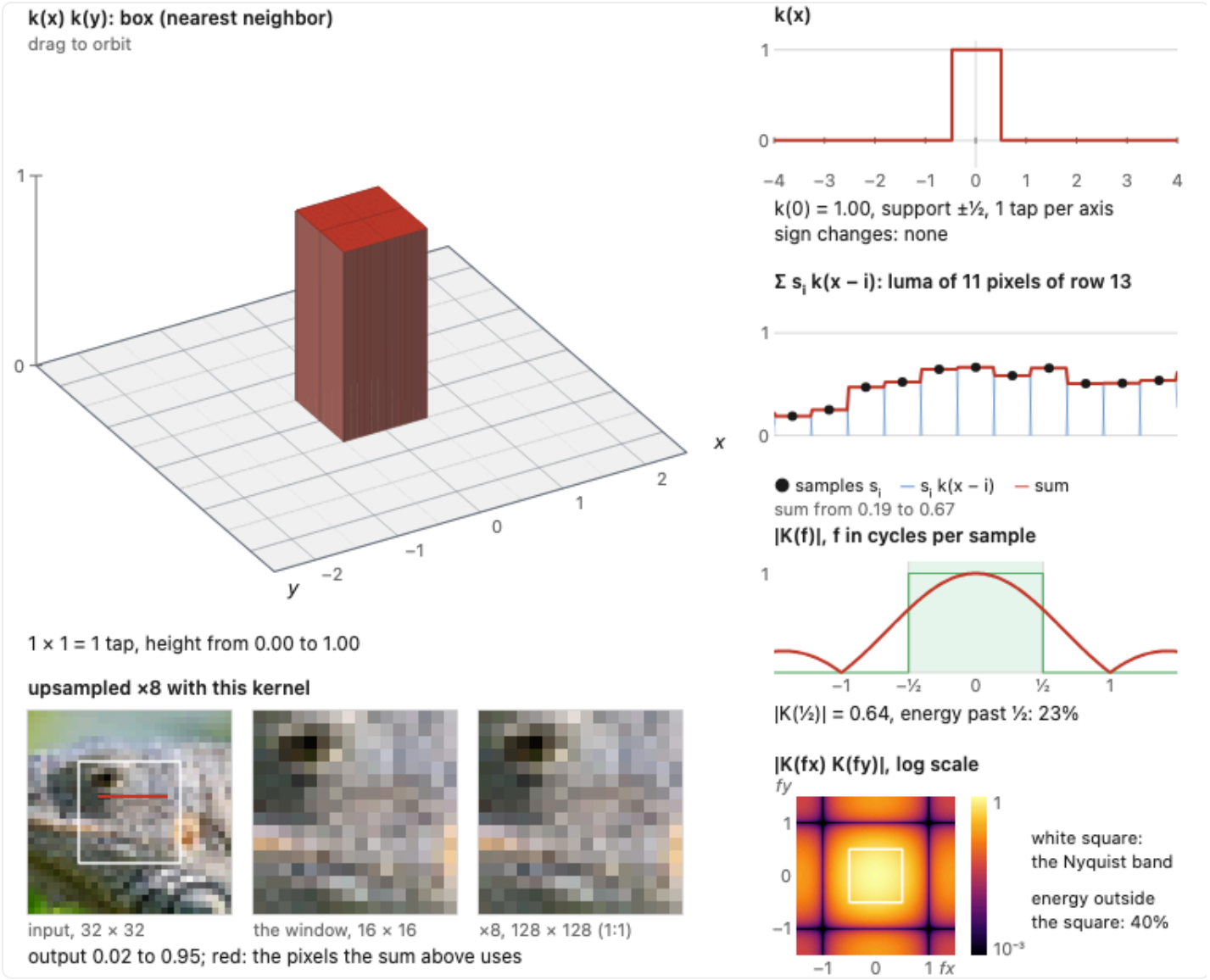


Figure 6.5.7, stage 1 (reproduced). *The box kernel of nearest neighbor, $k(x) = 1$ for $|x| < \frac{1}{2}$, as the separable 2-D kernel $k(x)\,k(y)$: a column one sample wide and of height 1 over the grid of sample positions. Drag the height field to turn it. On the right, top to bottom: the profile $k(x)$, with a dot wherever the kernel changes sign; eleven pixels of one row of the input (their luma, marked red on the input below) reconstructed as a sum of copies of the kernel, each centered on a sample and scaled by its value (blue); the magnitude of the Fourier transform along $f_y = 0$ against the ideal reconstruction filter (green), which passes every frequency up to Nyquist, half a cycle per sample, and none beyond; and the 2-D magnitude $|K(f_x)\,K(f_y)|$ on a log scale, with the Nyquist square outlined. Bottom left, the input, a 16 × 16 window of it, and that window upsampled ×8 with the same kernel, shown at 1:1. The menu picks the input, the same six as Figure 6.5.1, the iguana first. The box's transform is $\mathrm{sinc}(f)$: already down to 0.64 at Nyquist, it leaves 23% of its energy beyond Nyquist in 1-D and 40% outside the square in 2-D, the aliasing that shows as blocks.*

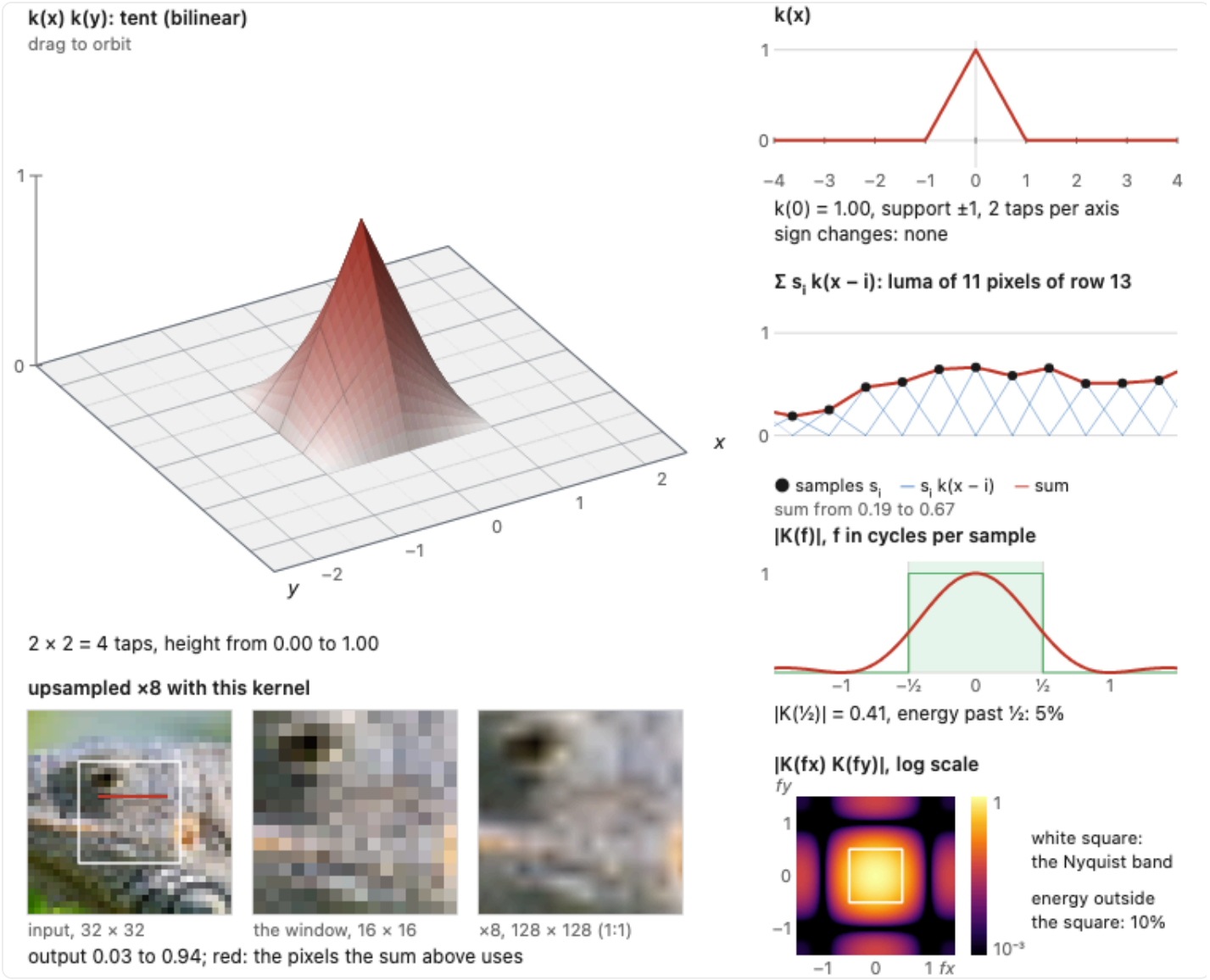
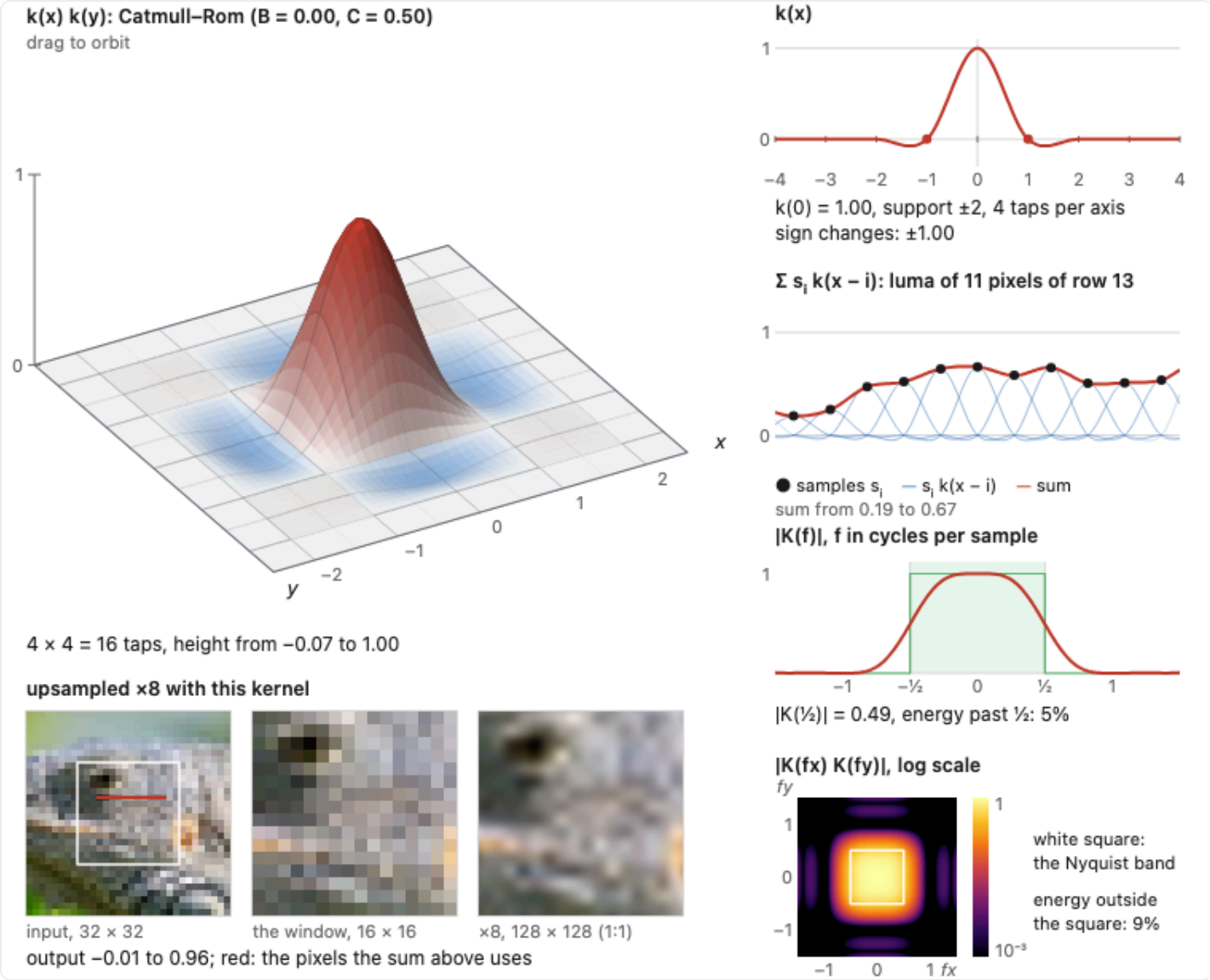


Figure 6.5.7, stage 2 (reproduced). *The tent, $k(x) = 1 - |x|$ for $|x| < 1$, in 2-D. The product $k(x)\,k(y)$ is not a pyramid: along a diagonal it falls as $(1-t)^2$, and its four taps are the four bilinear weights. The reconstruction of the row is the polyline through the samples. Its transform is $\mathrm{sinc}^2(f)$, 0.41 at Nyquist, so the passband droops more than the box's, while only 5% of the energy lies beyond Nyquist in 1-D and 10% outside the square in 2-D.*

The box and the tent are the crudest kernels. Better ones, the Mitchell–Netravali cubics k_{BC} and the Lanczos windowed sinc, are developed as approximations of the ideal sinc in Chapter 6.4, Sampling, downsampling, and aliasing, where they first serve as prefilters for downsampling. As reconstruction kernels they upsample with less blur than the tent and without the box's blocks, at the price of some overshoot near edges.

The third stage of Figure 6.5.7 draws any member of the family in 2-D, with its Fourier transform.



1 nearest neighbor: the box | 2 bilinear: the tent | 3 bicubic: the (B, C) cubics | 4 Lanczos: windowed sinc

↓ settings from the previous stage

Figure 6.5.7, stage 3 (reproduced). *The Mitchell–Netravali cubics k_{BC} in 2-D. The menu sets the named members and the sliders any (B, C). Below zero the sheet turns blue: Catmull–Rom's negative lobes are four shallow troughs along the axes, down to −0.07, and on the diagonals, where both factors are negative, the product is positive again. Catmull–Rom's reconstruction passes through the samples and overshoots at sharp transitions; the cubic B-spline has no negative lobe, does not overshoot, and misses the samples. The B-spline is the box convolved with itself four times, so its transform is $\operatorname{sinc}^4(f)$: 0.16 at Nyquist with 0.6% of the energy beyond it, against 0.49 and 4.7% for Catmull–Rom and 0.38 and 3.2% for Mitchell. For every (B, C) the transform is zero at every nonzero integer frequency, because the weights sum to one at every phase.*

The sixth stage of Figure 6.5.1 upsamples with any member of this family.

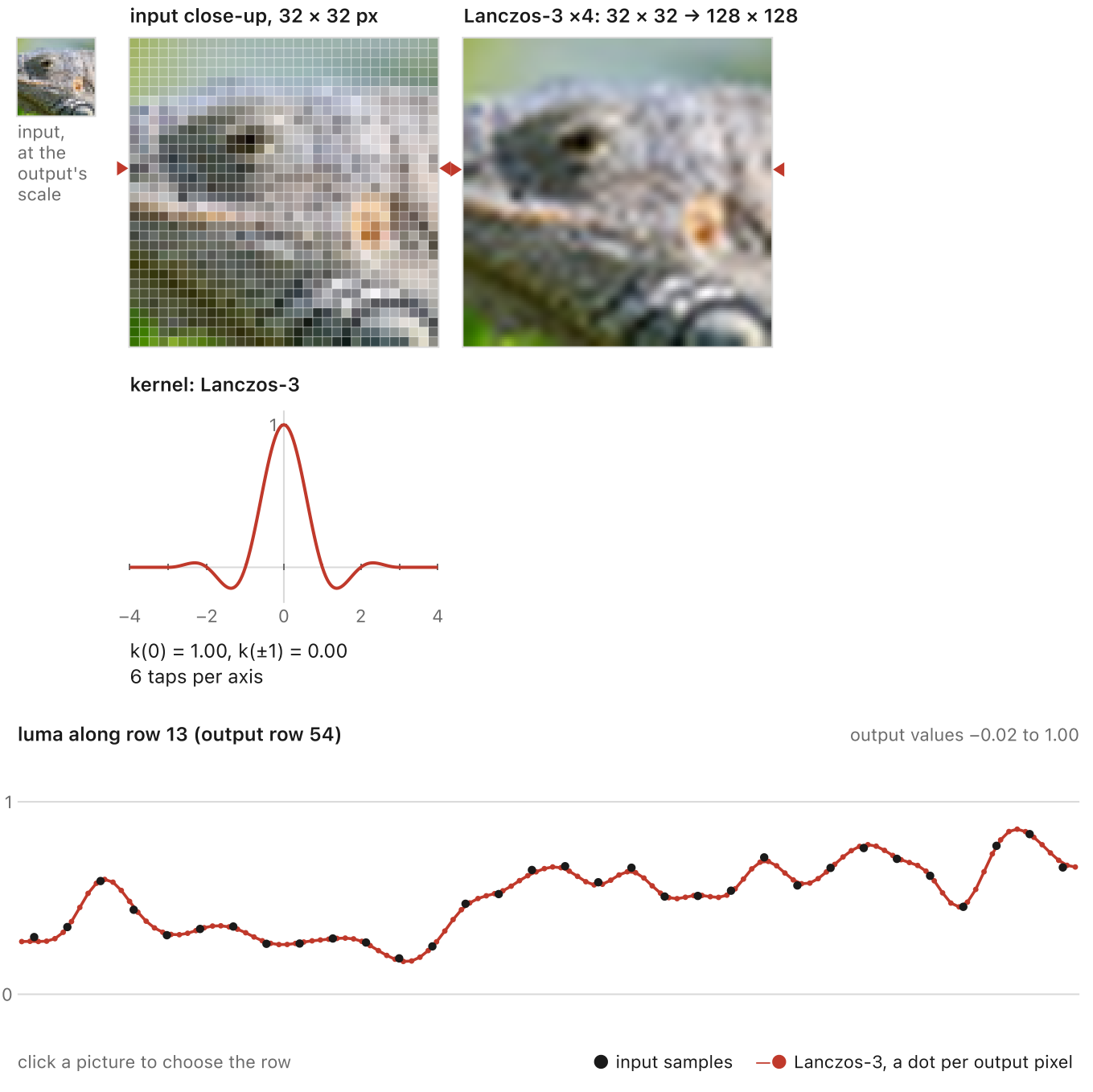


Figure 6.5.1, stage 6 (reproduced). *The Mitchell–Netravali cubics. The menu sets the named members (Catmull–Rom, which is Keys' cubic convolution with $a = -\frac{1}{2}$; Mitchell; the cubic B-spline; and Keys with $a = -\frac{3}{4}$, the cubic OpenCV uses) and the sliders set any (B, C). The readouts under the kernel give $k(0) = 1 - B/3$ and $k(\pm 1) = B/6$, so only $B = 0$ passes through the samples; with the B-spline the row curve misses the dots. Negative lobes take the output outside the input's range, printed above the row plot: on the checkerboard (squares of 0.1 and 0.9, at ×4) Catmull–Rom reaches 1.03, Mitchell 0.96, and the B-spline stays at 0.9.*

Its last stage swaps the cubic for the Lanczos kernel, with order a from 2 to 4.

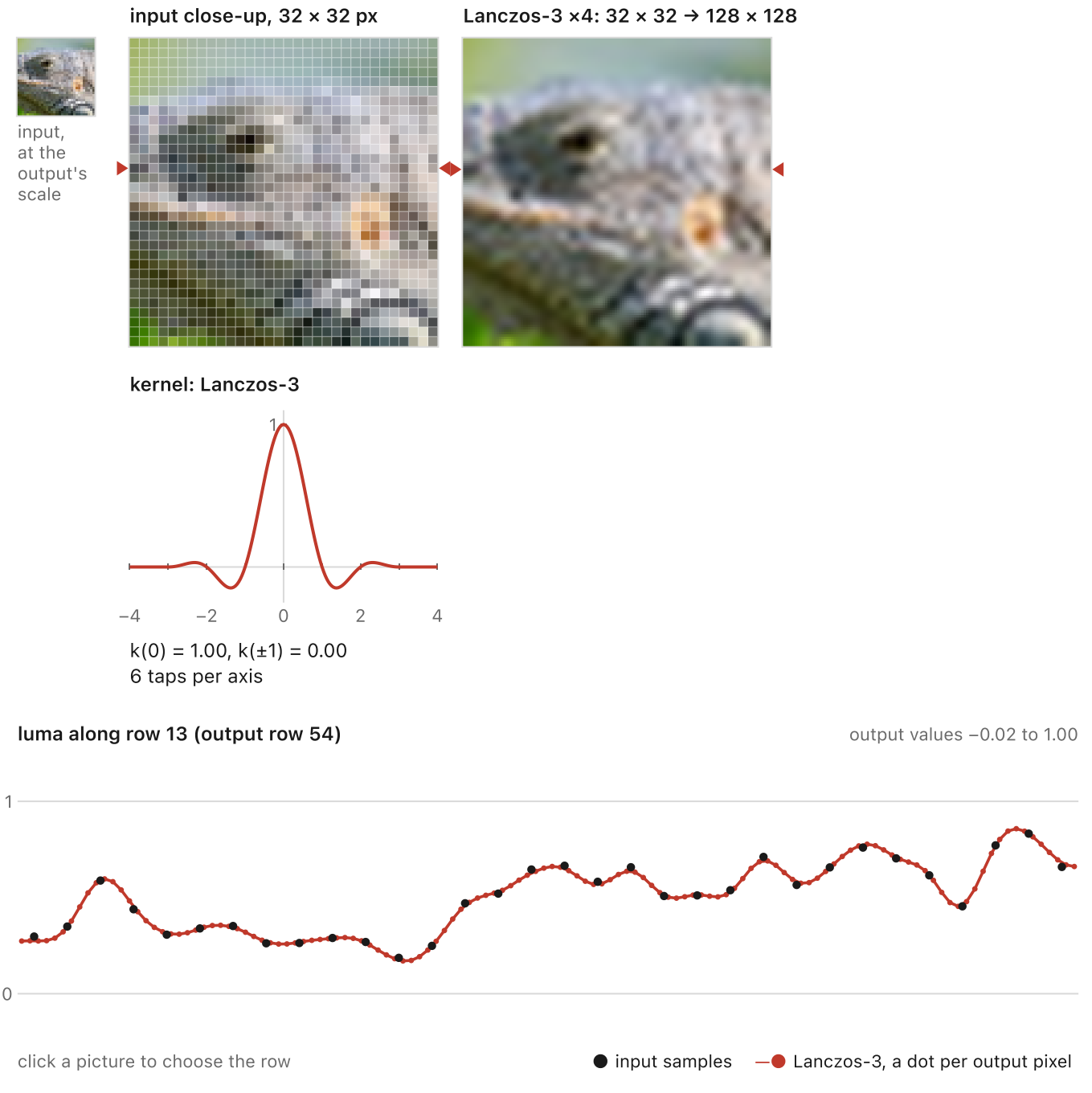
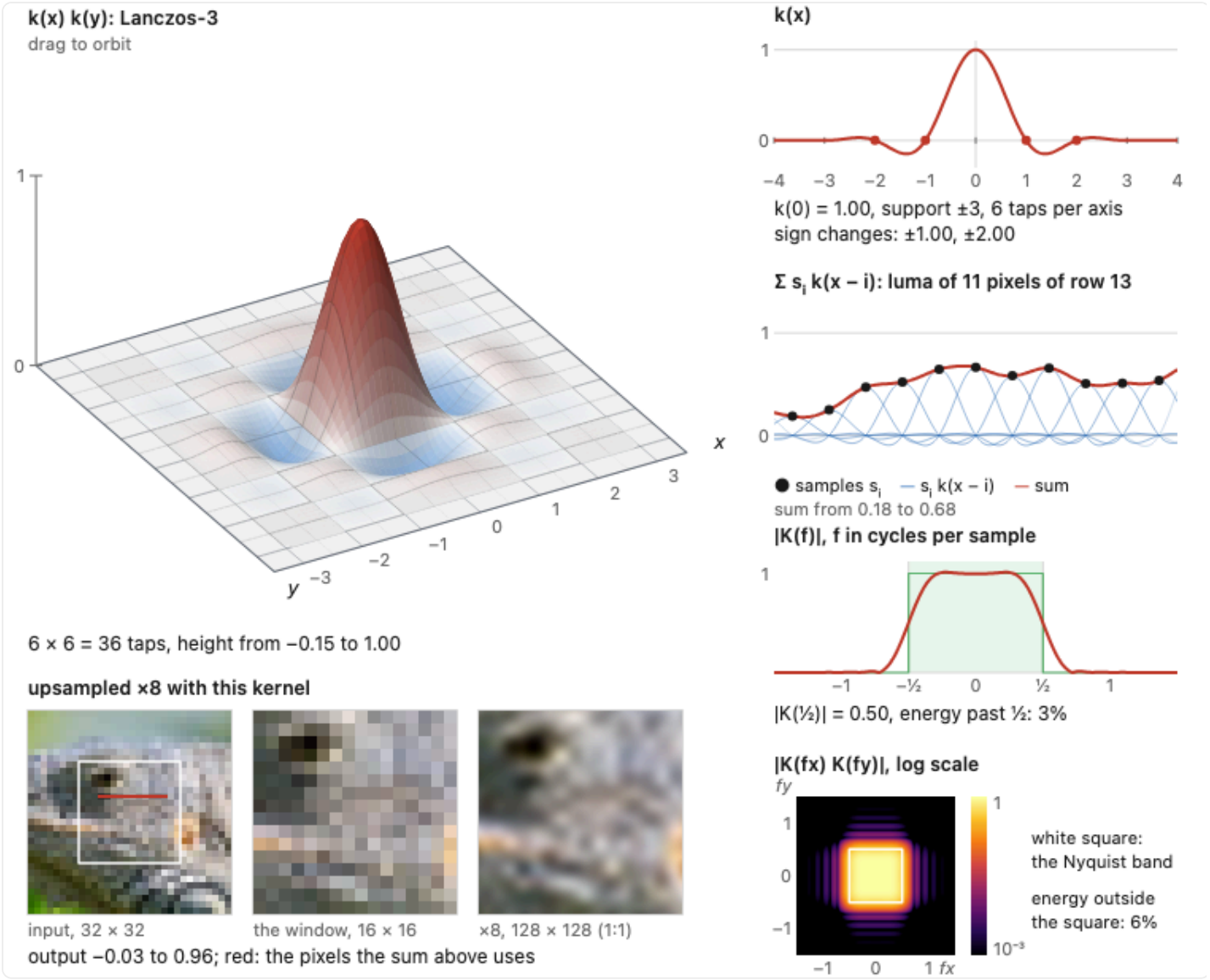


Figure 6.5.1, stage 7 (reproduced). *Lanczos of order a from 2 to 4: the sinc windowed by a sinc a times wider, 2a taps per axis, with the weights renormalized to sum to one. More lobes, more ringing: on the checkerboard at ×4 the output reaches 1.04, 1.11 and 1.14 for $a = 2, 3$ and 4.*

The last stage of Figure 6.5.7 draws the Lanczos kernel the same way, for orders 2 to 4.



1 nearest neighbor: the box | 2 bilinear: the tent | 3 bicubic: the (B, C) cubics | 4 Lanczos: windowed sinc

↓ settings from the previous stage

Figure 6.5.7, stage 4 (reproduced). *The Lanczos kernel* $\operatorname{sinc}(x)\operatorname{sinc}(x/a)$ *in 2-D, order* a *from 2 to 4:* $2a$ *taps per axis,* $4a^2$ *in 2-D, and* $a-1$ *sign changes on each side. Its transform is the closest of the four kernels to the ideal: about 0.50 at Nyquist for every order, with 4.6%, 2.9% and 2.1% of the energy beyond Nyquist for* $a=2$*, 3 and 4 (9%, 6% and 4% outside the square in 2-D), and the step from passband to stopband steepens as* a *grows. The price is the lobes: the 2-D kernel dips to −0.09, −0.15 and −0.18, and the reconstruction rings at sharp transitions. The weights are not renormalized here, and Lanczos weights do not sum exactly to one, so a constant row comes back with a ripple of up to 1.9%, 0.6% and 0.2%; practical resamplers renormalize them.*

Upsampling vs. downsampling: scale the kernel

in the book ↗

We have promised twice that downsampling reuses the same machinery; now we cash it in. Whether you scale up or down, you run the *same* convolution with the *same* family of kernels. The only thing that changes is how you

scale the kernel.

When upsampling, the kernel keeps a fixed footprint in the input image (about four pixels wide for bicubic, however much you enlarge) because the input samples are already dense enough; you are just reading between them. So for upsampling the kernel width in input space is constant (a useful corollary: clamp the kernel's scale factor at 1, never narrower).

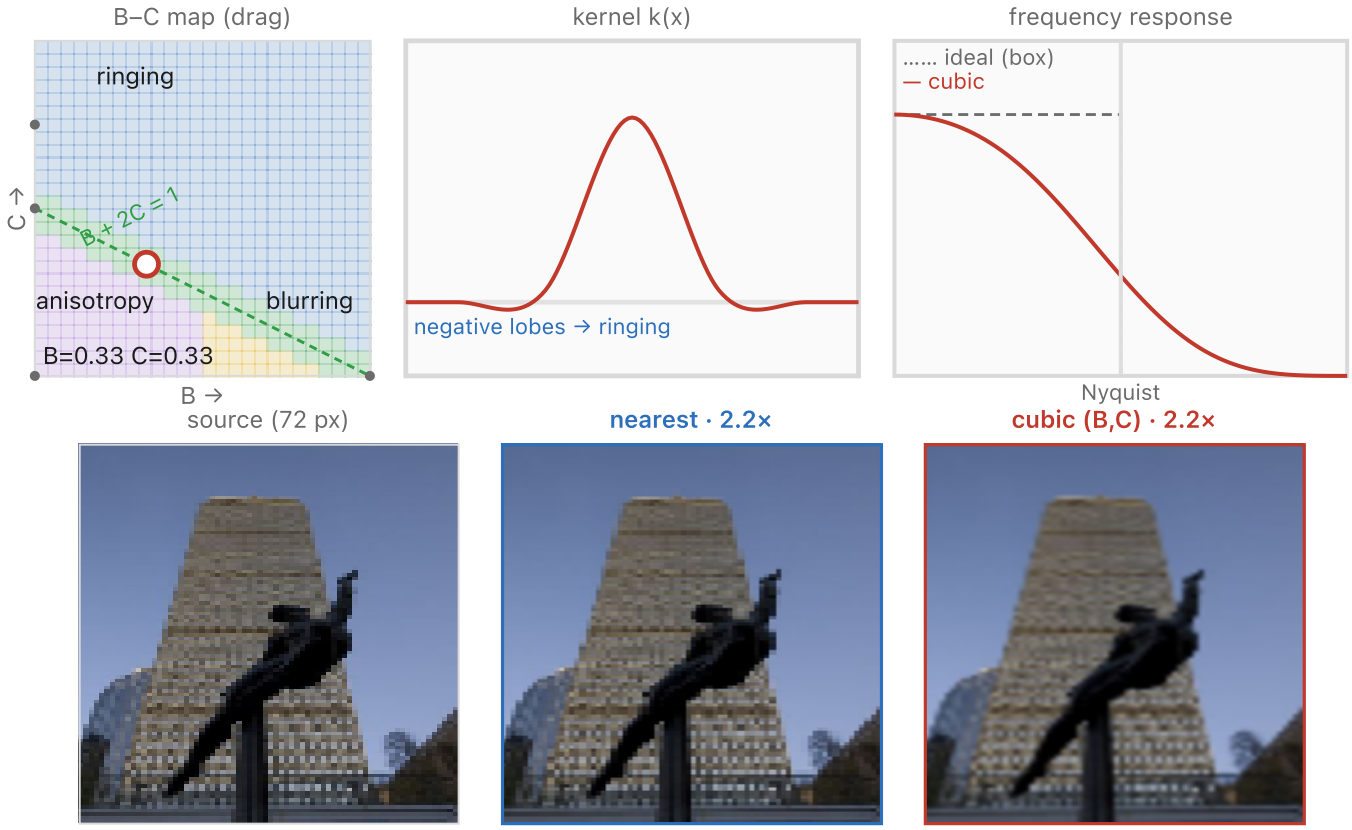
Downsampling is the harder case, and not by a little. Pick one input pixel per output pixel (nearest neighbor, no blur) and you get aliasing: a fine texture turns into bold, wrong, low-frequency moiré (a brick wall or a striped shirt shot from far away dissolving into wide phantom bands; Figure 6.5.13). This is not a cosmetic glitch but the wagon-wheel / Prudential-tower effect from Chapter 6.4, Sampling, downsampling, and aliasing: detail finer than the output grid can represent does not vanish, it *folds down* and masquerades as coarse structure.

Widen the filter in proportion to the reduction. In one dimension, let $q = W_{\text{in}}/W_{\text{out}}$ and $a_s = \max(1, q)$. A practical normalized resampler uses

$$w_j = k((u - j)/a_s), \qquad O(x) = \frac{\sum_j w_j I(j)}{\sum_j w_j}. \tag{9}$$

The source coordinate is u. The sum covers the stretched kernel's support. A fivefold reduction stretches a four-sample cubic support to twenty input samples, not five. Normalization preserves a constant image when the denominator is nonzero. Finite support still permits residual aliasing. How much depends on the kernel as well as its width. A box as wide as the reduction, the plain average of each block, is the most common prefilter and one of the leakiest: the sidelobes of its transform pass a fine periodic texture. Shrinking the facade of Figure 6.5.8 threefold, 88% of the energy that ends up in the output band is aliased with a box prefilter, against 9% with a Gaussian and 2% with Lanczos-3 (stage prefilter).

Pixel-center conventions also matter. With centers at integer indices and image edges at half-integers, use $u = (x + 1/2)W_{\text{in}}/W_{\text{out}} - 1/2$. This differs from aligning the first and last centers. Choose one convention and use it consistently when composing transforms.



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

Figure 6.5.10 (reproduced). *The Mitchell–Netravali cubic family. A single pair of parameters (b, c) tunes the cubic resampling kernel across a whole family: from blurry (the cubic B-spline, $b = 1$) through the balanced Mitchell ($b = c = \frac{1}{3}$) to sharp-but-ringing (Catmull-Rom, $b = 0$, $c = \frac{1}{2}$). Drag the (b, c) point to trade blur against ringing and overshoot, shown on the kernel, its frequency response, and a resampled edge.*

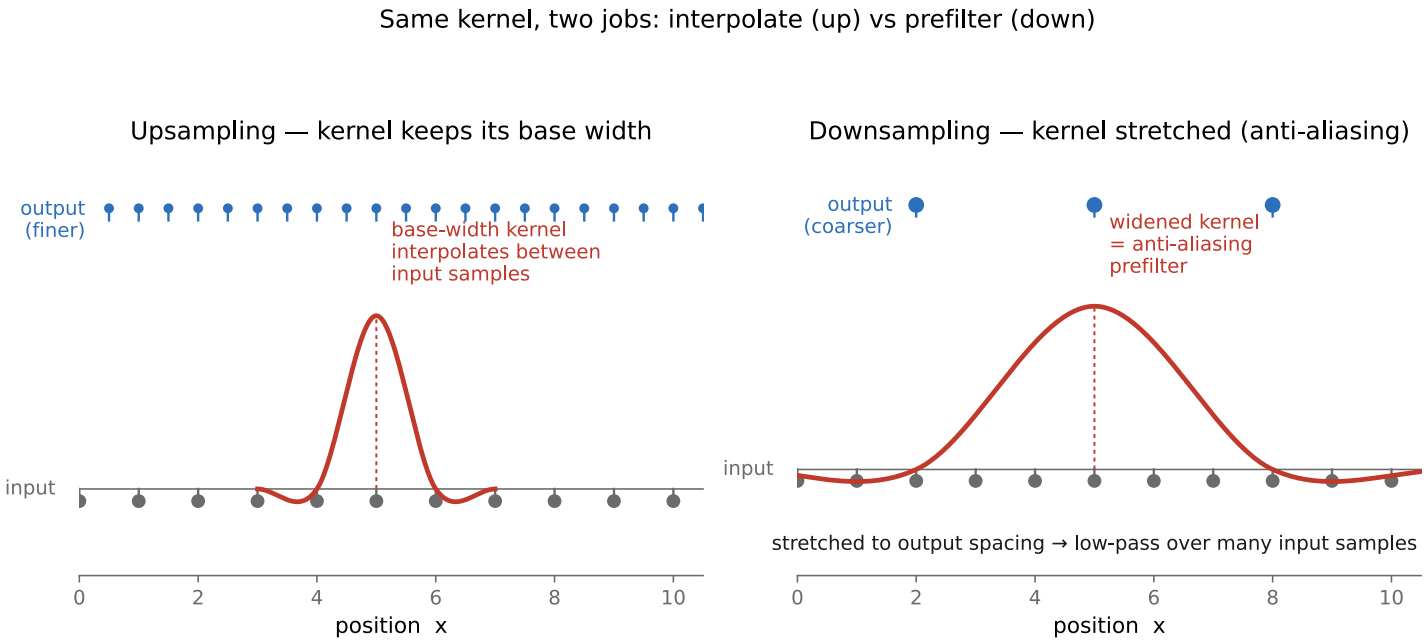


Figure 6.5.11 (reproduced). *The one knob that distinguishes up- from downsampling. Upsampling: the kernel keeps its native width in* input *space (about four pixels for bicubic), reaching across the same handful of dense input samples no matter how much you enlarge. Downsampling: the kernel is* stretched *to span the output-pixel spacing, by $s\times$ for an $s\times$ reduction, so it averages over all the input detail crammed into each new pixel, doubling as the anti-aliasing prefilter. Same kernel shape, scaled differently.*

Not every resize function widens it. Parmar et al. found that several widely used libraries reduce an image with a kernel of fixed width by default; Figure 6.5.12 puts that behavior beside the alternatives and traces the aliasing to the kernel.

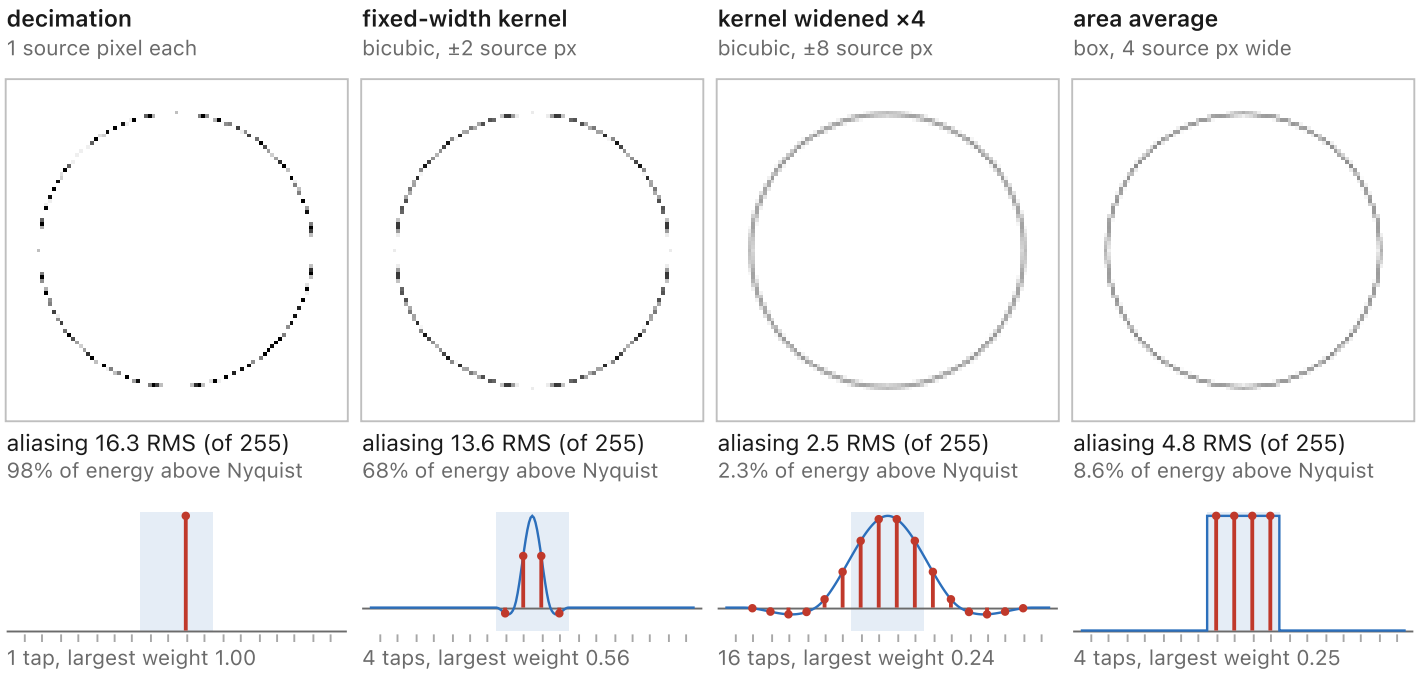


Figure 6.5.12 (reproduced). *Four downsamplers on one input, side by side, each with the kernel it applies drawn under it in source pixels: the dots are the weights one output pixel takes from the source pixels (the ticks), and the shaded band is that output pixel's footprint, s source pixels wide. Decimation reads one source pixel. A fixed-width kernel, bilinear or bicubic at the source's own scale, reads two to four pixels near the middle of the footprint and ignores the rest, the default of several of the libraries Parmar et al. compare on a thin circle. The kernel widened by the factor covers the footprint, and so does the area average. On the thin circle the first two break the line into scattered dots; on the zone plate they bring back rings that are not in the source. The readout under each result measures its aliasing: the part of the source above the output's Nyquist frequency (what an ideal low-pass in the DCT basis removes) is passed through the same downsampler, and whatever comes out is content the output cannot hold. It is given as an RMS in 8-bit code values and as a share of that part's energy. At 1/4 on the circle with bicubic, decimation keeps 98% of it, the fixed kernel 68%, the area average 8.6% and the widened kernel 2.3%. At odd whole factors the fixed tent lands on a single source pixel and becomes decimation; at other factors its taps shift from one output pixel to the next. The readout measures aliasing, not blur: the widened tent scores lowest and is also the softest. The arithmetic is on sRGB code values, as the libraries do, and each result is enlarged by a whole number with nearest neighbor. Peacock photograph by Bernard Spragg (CC0).*

The effect is ugliest on real, regular structure: the rows of windows on a glass facade, shrunk for a thumbnail, break into sweeping moiré if you decimate without prefiltering (Figure 6.5.13).

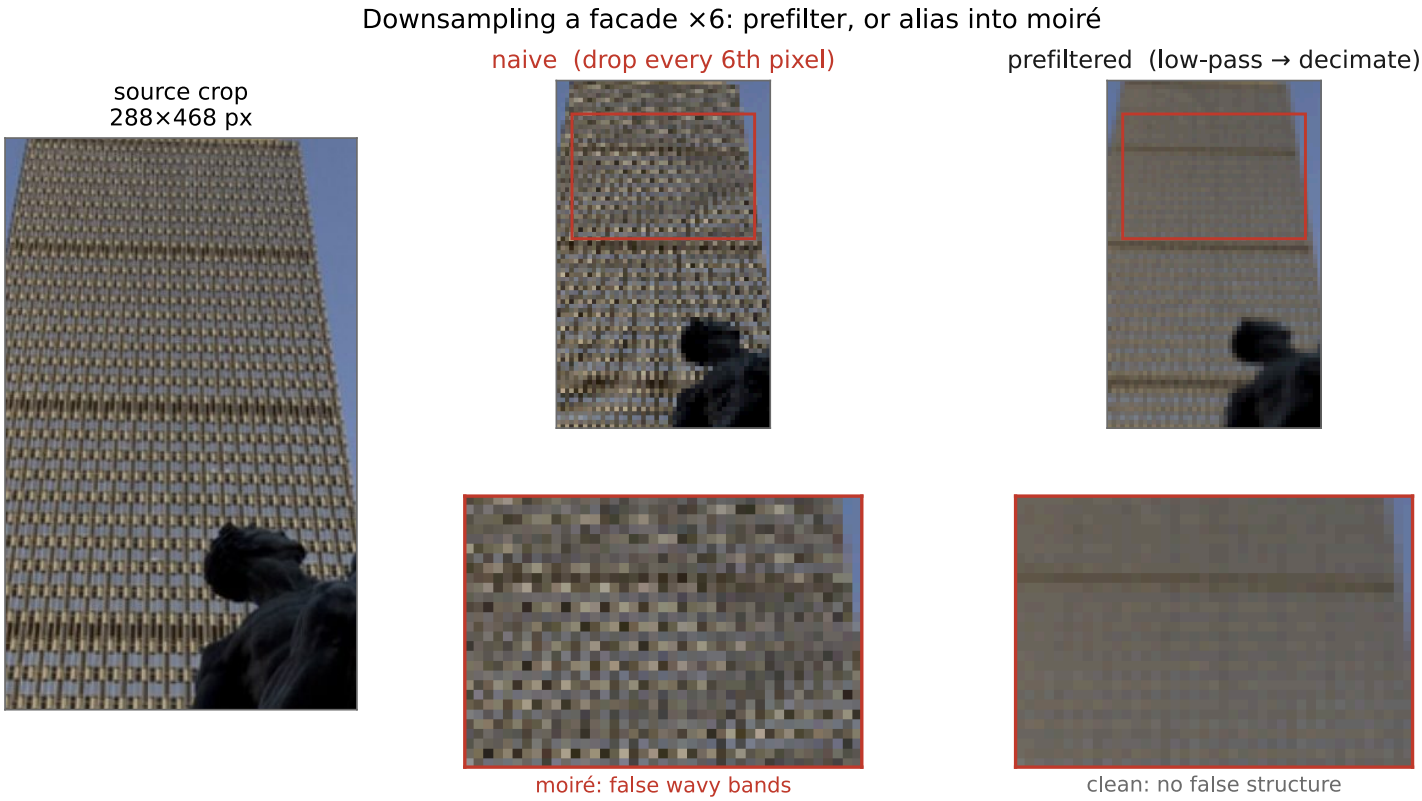


Figure 6.5.13 (reproduced). *Downsampling a real photograph of a building facade. Left: naive decimation (pick one pixel per output) turns the regular grid of windows and mullions into coarse, wrong moiré bands sweeping across the facade: aliasing on actual content, not a test pattern. Right: the same shrink with a properly scaled prefilter first, a clean, sharp miniature with the window grid faithfully averaged down. The artifact is exactly the aliasing of the previous figure, now on the kind of image you would really downsample.*

Sidebar 5: Andrew Adams' rotation challenge

Rotate an image through a full turn in N equal steps, each resampling the previous result. Compare it with one resampling using the composed transform. The repeated version accumulates interpolation error even though the net geometry is identity. Sharper kernels can retain more detail but may also amplify ringing. No finite resampler guarantees that hundreds of rotations recover the original. Evaluate only the region whose source support remains valid throughout.

BIG LESSON (L6.20) — EVERY RESAMPLING PASS MULTIPLIES THE SPECTRUM AGAIN, SO RESAMPLE ONCE

Repeated resampling accumulates blur, ringing, and sampling error. For repeated fixed convolution the frequency response is exactly H^N; for rotations it is only an illustration because the grid and effective filter change at each pass. Compose geometric transforms and resample the original once.

It's okay, we can just feed the one-pixel image into an AI upscaler and recover the original image, or at least one that's just as cool.

Figure 6.5.14 (reproduced). xkcd #2671 "Rotation", Randall Munroe, xkcd.com/2671 (CC BY-NC 2.5).

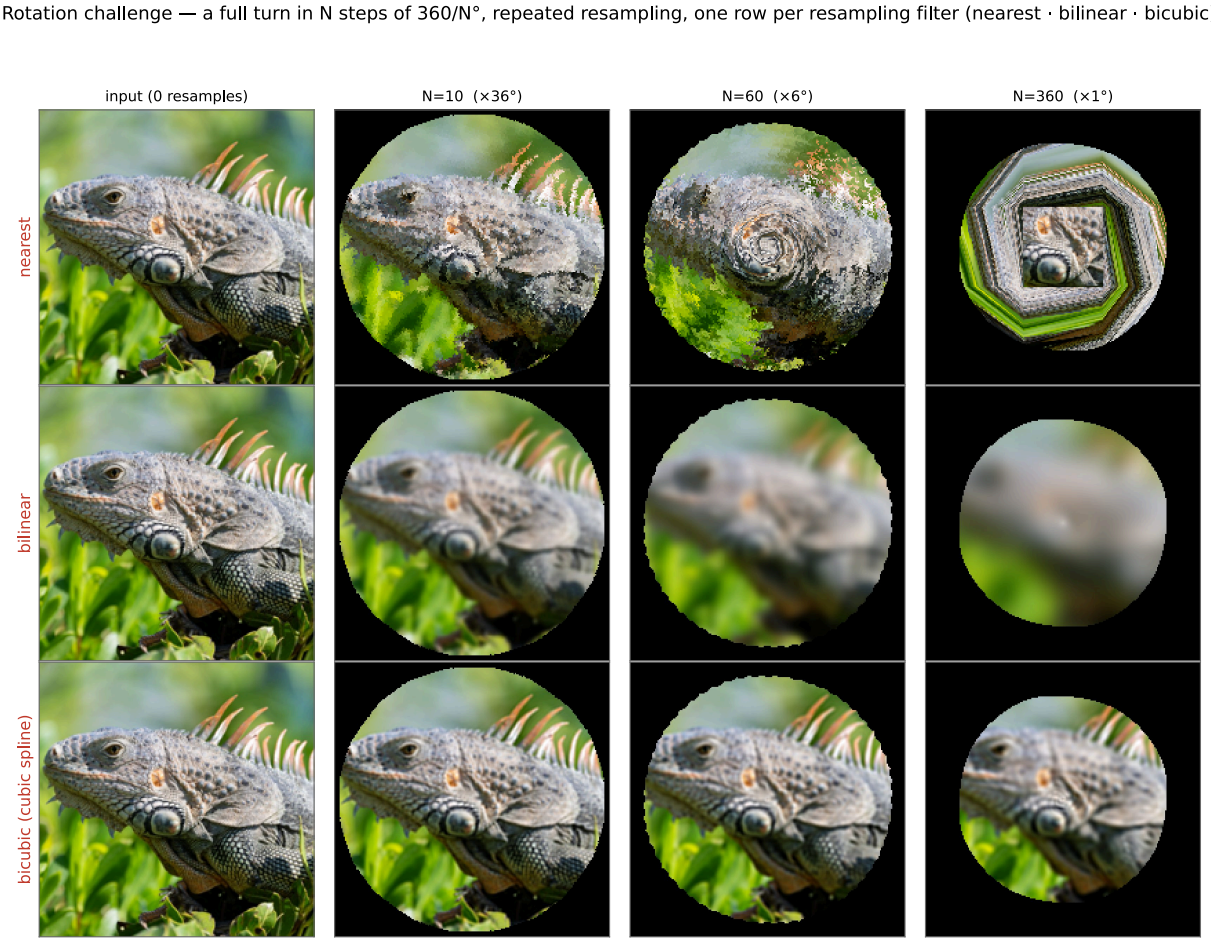


Figure 6.5.15 (reproduced). *Andrew Adams' rotation challenge, and how much the resampling filter matters. Each image is rotated a full* 360° *in* $N = 10$, 60, *or* 360 *equal steps, every step resampling the* previous *result; it is the same total rotation, just chopped into more or fewer passes, so more (smaller) steps compound more damage. The three rows repeat the experiment with a different resampling filter: nearest-neighbor, bilinear, and bicubic (cubic spline). Nearest jitters and loses registration; bilinear (a mild low-pass) washes the iguana's fine scales into mush by* $N = 360$*; bicubic, whose negative lobes restore some of the contrast each pass would otherwise lose, holds the detail markedly longer. A validity mask blacks out the corners that ever rotate out of frame, so the kept region shrinks toward the inscribed disc. The lesson: the resampling kernel is not a detail: it is the difference between a tool you can trust and one that quietly destroys your image.*

Run the same experiment yourself in the interactive version below: one image, your choice of total angle and step size (the total is rounded to a whole number of steps), and any of five filters, and when you pick bicubic, the same (B, C) handle from above to dial the cubic between blur and ring.

For a far deeper practical treatment of resampling filters, their windows, and the subtle quality differences between them, Nicolas Robidoux's analysis for ImageMagick is the reference worth reading

(<https://usage.imagemagick.org/filter/nicolas/>); it is the kind of care that separates a library taking resampling seriously from one that ships a default and hopes.

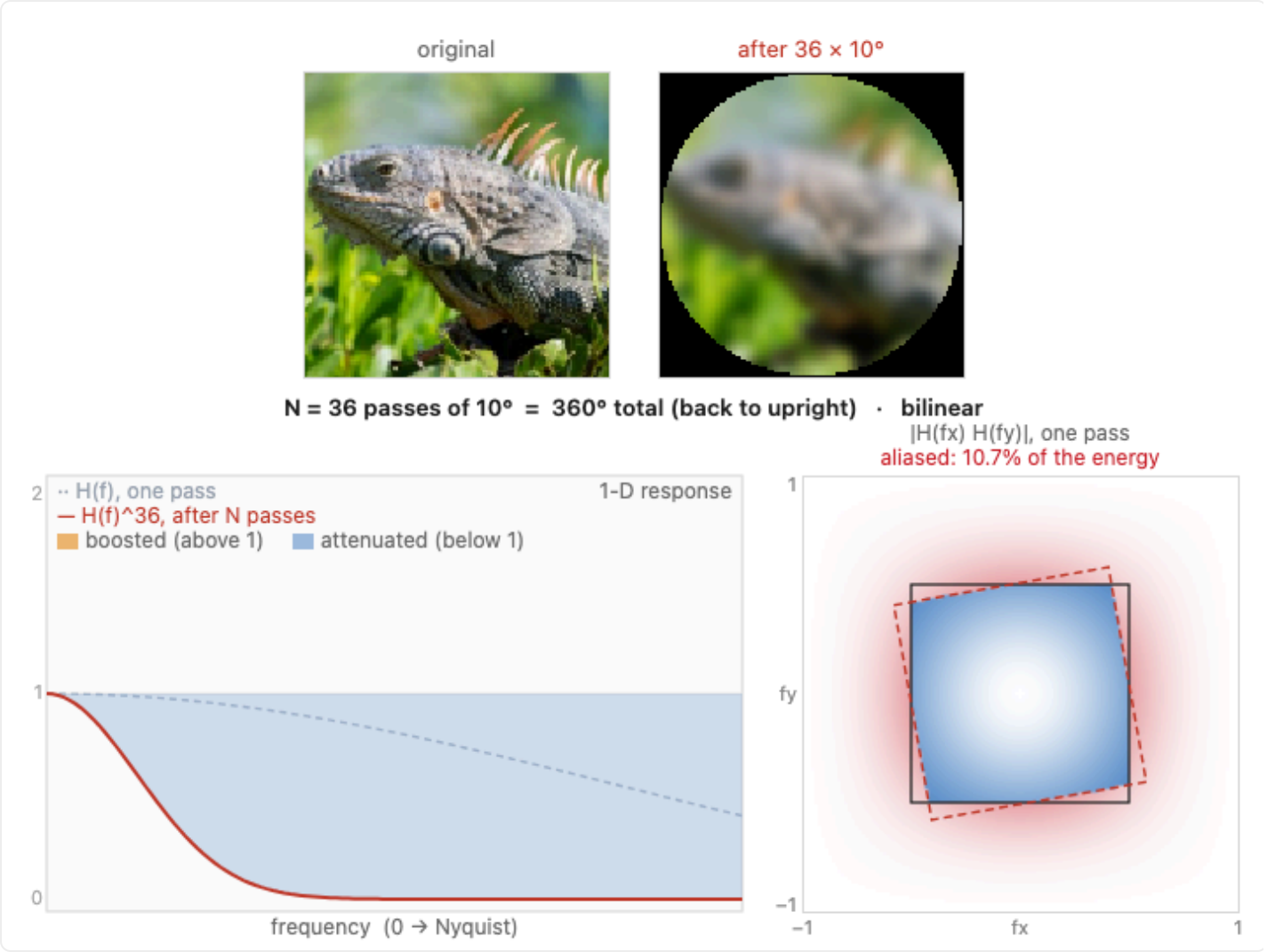


Figure 6.5.16 (reproduced). *The rotation challenge, interactive. Rotate one image to a chosen total angle in equal steps, each step re-sampling the previous result with the selected filter (nearest · box · bilinear · Lanczos · bicubic). The rotation runs on sRGB code values, and "box" here has two-pixel support, unlike nearest-neighbor. The orientation depends only on the total; the damage depends on the number of passes N and on the filter: drop the step size to compound more passes, switch filters to watch nearest jitter, box and bilinear blur to mush, and Lanczos or a well-tuned bicubic hold the detail. For bicubic, drag the (B, C) point to choose the cubic. Set the total to 360° to land back at the original orientation and read off pure resampling loss. A bottom panel plots the kernel's 1-D frequency response $H(f)$ and, as an approximation of the N stacked passes, $H(f)^N$: the curve collapsing toward DC is why detail vanishes: each pass multiplies the spectrum by H, so after N passes the surviving spectrum is roughly H^N. The plot keeps one scale, 0 to 2, and shades orange where the passes boost a frequency (the sharpening that rings) and blue where they attenuate it. The square panel is the 2-D spectrum of the separable kernel, $|H(f_x)H(f_y)|$, out to one cycle per sample, with the source grid's Nyquist square in black and the turned grid's dashed. Only frequencies inside both pass as themselves; what the kernel lets through elsewhere, shown red, comes from the spectrum's copies or from corners the turned grid cannot hold, and folds back as aliasing. The readout gives its share of the kernel's energy: about 29% for nearest and 8% for Lanczos at a 10° step. Switched to N passes, the panel multiplies the kernel turned by every step, and only what fits inside every grid along the way passes as itself, outlined dashed: the square rounds into a near disc of radius one half, and the readout gives both the share left outside after all the passes and the share a single pass aliases.*

Prefiltering when the transform isn't a clean scale

OPTIONAL
in the book ↗

An affine map sends a square output footprint to a parallelogram in the input. Elliptical weighted average (EWA) filtering approximates the footprint with an ellipse. Its shape can be obtained from the Jacobian J of the inverse map: a circular footprint with covariance C maps locally to covariance $JCJ^\top$. An affine map has the same Jacobian everywhere; a perspective or nonlinear map generally does not. A pure rotation changes orientation without changing area or introducing unequal axis scales.

Mipmaps choose among prefiltered resolutions cheaply; EWA accounts more closely for directional compression. Both approximate the combined reconstruction and output filtering. Chapter 11.2, Antialiasing for complex transforms develops the footprint model.

Margin note: resample in *linear* light, not in gamma

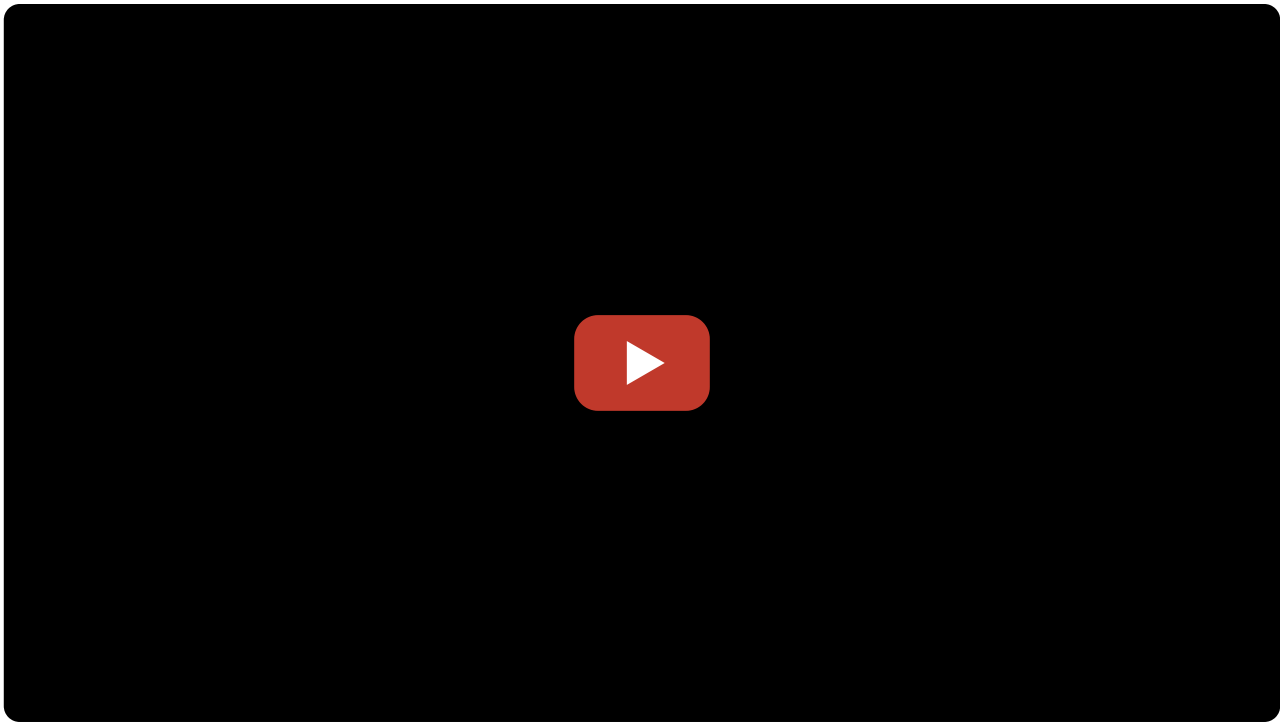
A resample is a weighted average of neighboring pixels, and an average is only physically meaningful on values proportional to light, i.e. linear radiance. Most image files are gamma-encoded (sRGB), and averaging the encoded values is wrong: the encoding is concave, so the mean of two encoded values is *darker* than the encoding of their mean. The visible symptom is the classic "gamma-incorrect" darkening on downsampling and on antialiased edges: thin bright features on dark backgrounds lose energy, and mip-mapped textures go muddy. The fix is the standard ISP discipline (Chapter 45.15, Rendering a raw DNG): decode to linear, resample, re-encode, once, around the whole pass. Log space is a third option, but a *perceptual / dynamic-range* one rather than a physically correct one: averaging in log is a geometric mean that down-weights a few very bright pixels and can look better on extreme-HDR content, but for ordinary resampling linear is the correct default and gamma is the trap.

This matters on a receding surface. One output pixel may cover little source extent in one direction and a long strip in the other. Choosing one isotropic mipmap level large enough for the strip can erase useful detail across it; choosing a finer level can leave aliasing along it. An anisotropic footprint spends its averaging in the compressed direction. Ripmaps, multiple texture probes and EWA make different approximations to this same coverage problem, developed in Chapter 11.2, Antialiasing for complex transforms.

Practical implementation choices

OPTIONAL
in the book ↗

For a reproducible resize, specify the coordinate convention, kernel and scale, boundary extension, processing domain, and alpha treatment. Check that a constant image stays constant, an identity transform reproduces samples for an interpolating kernel, and a fine pattern is adequately filtered during reduction. Compose successive transforms before sampling.



Video 2. Video: Interpolation in 5 minutes, a short visual tour of nearest-neighbor, linear, and cubic interpolation, by Graphics in 5 Minutes (**YouTube**).

Figure 6.5.20 (reproduced). xkcd #1416 "Pixels", Randall Munroe, xkcd.com/1416 (CC BY-NC 2.5). A recursive zoom into ever-finer pixels, an apt closing image for a chapter about reconstructing values between samples.

Warping, morphing, and lagrangian approaches

OPTIONAL
in the book ↗

Straightening a façade, aligning a burst and moving a face landmark all change where image samples contribute. A warp specifies that geometry; resampling determines the output values. Keeping those two tasks separate lets us reuse one sampling implementation for many kinds of edit.

Chapter 11.1, Warping defines forward and backward maps, boundaries and interpolation. Chapter 11.2, Antialiasing for complex transforms handles footprints that stretch or rotate across the image. A destination pixel may cover many source samples, so a sharper interpolator alone cannot prevent aliasing.

Chapter 11.3, Morphing combines correspondence with appearance blending. Chapter 11.4, Morphable models restricts possible shapes using a learned population model, while Chapter 11.5, Shape-preserving warping penalizes local deformation. These constraints can improve an edit without guaranteeing physical validity or preventing every fold.

Chapter 11.6, Perspective distortion and its correction separates planar rectification, virtual camera rotation and local perceptual adjustments. None can recover unseen surfaces from a single photograph without additional information or inference.

The distinction between changing values and moving content is related to two descriptions of fluid motion. An Eulerian description observes a fixed location, like a weather station measuring the air that passes it. A Lagrangian description follows material, like a balloon drifting with that air. A tone curve changes values on a fixed image grid. A geometric warp instead describes where scene content moves. The implementation may still visit a fixed output grid through backward mapping; the Lagrangian analogy concerns the motion being represented. Video compression shows why the distinction matters in practice. Coding each frame as its per-pixel difference from the previous one is an Eulerian scheme, and it wastes bits whenever something moves, since a moving edge produces large differences at every pixel it crosses. Motion-compensated coding, as in the MPEG and H.264 families, predicts each block from where its content was in the previous frame and codes only the correction, which is cheap when the motion is well described (Chapter 30.2, Video compression and motion compensation).

That separation also explains the organization of these chapters. One question is how to specify the map: a few parameters, sparse landmarks, a deformation mesh or a dense field. Another is how to apply it to sampled data. Translation, lens correction, stabilization and a facial morph can share a resampling engine even though their maps come from different models. Matching pixels across space and time addresses estimating those correspondences; this part develops their use to transport image content.

Warping

in the book ↗

> 📎 **Problem set**
>
> PS5 implements resampling kernels and Beier–Neely segment warps. → Chapter 1.5, Problem sets (appendix).

Print a photograph onto a sheet of rubber and stretch it: pinch a corner, bow the middle, slide one edge. The pixels glide to new positions and the picture rides along: a face slims, a bent line straightens, a tilted building stands up. Nothing about the *colors* changed; what changed is *where each color lives*. That is a warp, and it is the single operation underneath a large range of this book's geometry: lens-distortion correction, video stabilization, panorama reprojection, morphing, image-based rendering, even the cosmetic "liquify" slider. One primitive, reused everywhere, which is exactly why it opens this part.

Formally a warp is a map of the image *domain*: a function $f : \mathbb{R}^2 \to \mathbb{R}^2$ that moves positions while leaving the *range* (the colors) untouched. If the input had color c at (x, y), the output has that same c at $f(x, y)$. It is the geometric twin of a point operation: a point operation edits the range (the colors) and leaves the domain fixed; a warp edits the domain (the positions) and leaves the range fixed.

The map and the sampler have separate responsibilities. The map supplies coordinates; the sampler reconstructs values and filters the source footprint. A valid coordinate alone does not establish visibility or adequate antialiasing.

Warping is a domain transform: move *where*, not *what*

in the book ↗

Hold onto the rubber-sheet image, because it is the correct mental model for every warp in the book, from a sub-pixel lens-distortion correction to a full perspective rectification to "slim the subject." The warp deforms where pixels live and leaves their colors alone. Contrast it once more with a tone or point operation, which recolors each pixel in place: warping is the *domain* operation to that *range* operation, and the two are orthogonal; you can do either without the other, and morphing (later in the part) does both at once, a domain warp plus a range cross-fade.

Before building the machinery, note why this one primitive is everywhere, because the motivation is the whole reason the chapter comes first. Optical-aberration and lens-distortion correction are warps. Video stabilization is a warp (a different one per frame). Panorama reprojection, aligning each photo to a reference view, fixing the apparent shift of a rotated camera, is a warp. Wide-angle distortion correction, morphable face models, image-based rendering, the cosmetic slim: all warps. Learn to transport pixels along a map once and you have the engine for all of them. That observation, that nearly every geometric operation in the book is "build a map, then move the pixels along it," is the part's central lesson L17.

> 💡 **BIG LESSON (L17, RECURRENCE)**
>
> Almost every geometric algorithm in the book has the same two-part shape: first establish a coordinate map between two images (or between an image and a target frame), then transport the pixels along it. This part builds that machinery and its first uses: warping makes the map explicit and applies it, perspective rectification *solves* the map (a homography) from geometric constraints, and morphing *interpolates between* two maps. The same shape recurs across the book: panorama stitching *estimates* the map from matched features (in the multiple-exposure part), optical flow *estimates a dense* map from brightness constancy (in the matching part), and stabilization *smooths* a map over time while frame interpolation *renders along* one (in the video part). Separating the two halves, what the map is versus how you transport pixels through it, is what lets one transport engine (inverse warp + resample, built in the next two sections) serve this part and recur throughout the book; only the *map* changes from task to task. *(L17 is registered in the part introduction, Warping and morphing; this chapter is the transport engine, and the chapters that follow are different ways of finding the map.)*

Forward vs inverse warping: why output-driven wins

in the book ↗

We met forward and inverse mapping for scaling in Chapter 6.5, Resampling and upsampling, and the conclusion carries over to any warp f. Pushing each input pixel to $f(x, y)$ leaves holes where the warp stretches and piles pixels up where it compresses. Looping over the output pixels instead, and pulling each one back through the inverse map,

$$\mathrm{out}(x, y) = \mathrm{in}\big(f^{-1}(x, y)\big), \tag{10}$$

fills every output pixel exactly once, and leaves a single question: how to read the input at the fractional location $f^{-1}(x, y)$ (Figure 11.1.2).

Algorithm 2

```
for each output pixel (x, y):
    (x', y') = f⁻¹(x, y)               # pull back to the source location
    out[x, y] = resample(in, x', y')   # read in at the fractional spot
```

Forward splatting remains useful when only a forward map is known, as with optical flow, but it then needs explicit handling of holes and overlaps.

An invertible affine or projective map has a matrix inverse. A fitted map may be singular or poorly conditioned, however, and a free-form field can fold or leave gaps. Forward flow cannot generally be inverted by negating it at the same coordinates: corresponding vectors live at different locations. Estimate a backward field or invert/splat with explicit visibility and coverage handling. Occluded or newly exposed content has no valid correspondence.

One more practical point that is *not* a corner case: $f^{-1}(x, y)$ can land outside the input. You must pick a boundary policy: zero/black padding, clamp-to-edge (replicate the border pixel), mirror, or mark as undefined (transparent, via an alpha channel). This matters constantly, because rectification and stabilization both produce outputs whose corners read *past* the original image border. Reading off the edge is the common case, not the exception.

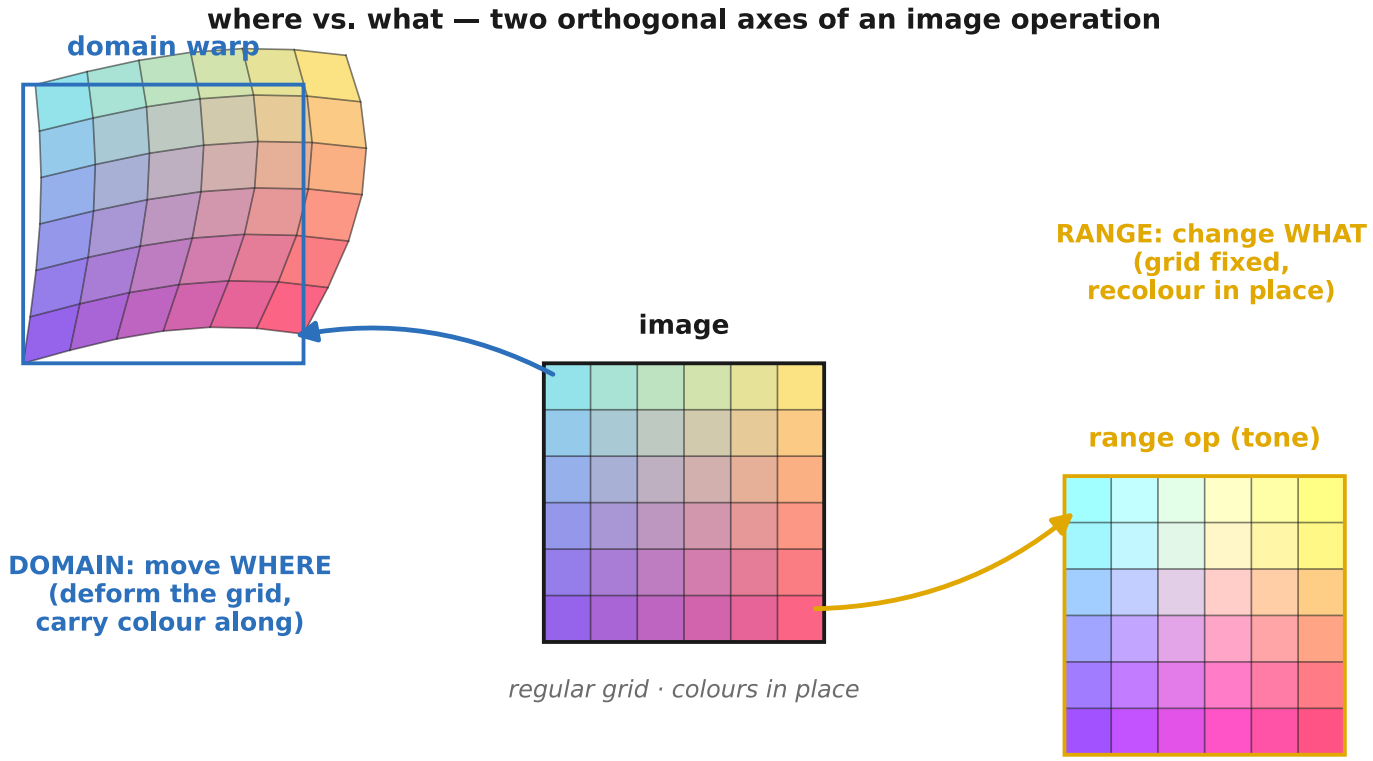


Figure 11.1.1 (reproduced). *Domain vs. range. One image, two arrows. The domain arrow bends the underlying grid: a pixel slides to a new location and carries its color along (a warp). The range arrow leaves the grid fixed and recolors a pixel in place (a point/tone operation). Warping moves where, not what; the two operations are orthogonal.*

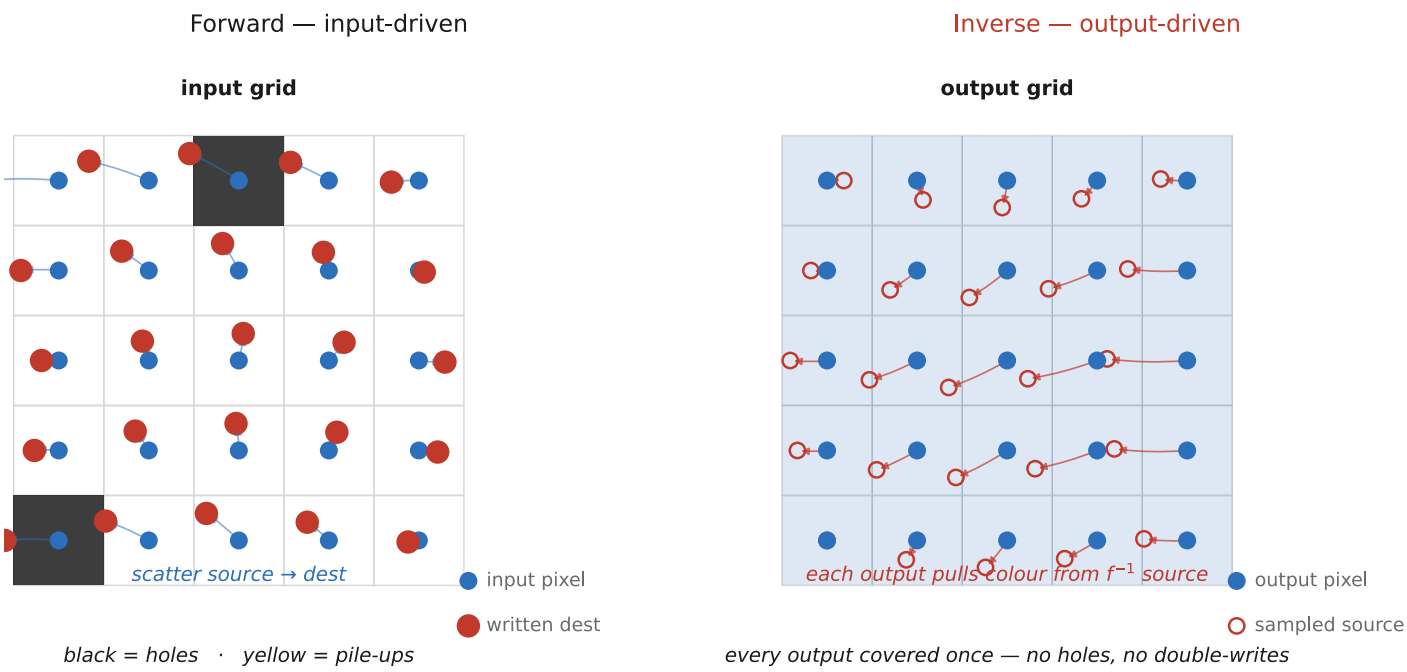


Figure 11.1.2 (reproduced). *Inverse (output-driven) warping, the useful one. Loop over output pixels; push each back through f^{-1} into the input; sample the color there. Every output pixel is covered exactly once, with no holes or double-writes, and the only remaining difficulty, reading the input at a fractional location, is a clean resampling problem.*

Resampling: a quick reminder

OPTIONAL
in the book ↗

Inverse warping leaves one question: read in at the fractional coordinate $f^{-1}(x, y)$. An image stores only samples, so to read between them we reconstruct the continuous signal and re-sample it, or *resample* it, and the reconstruction filter is the whole choice: nearest neighbor (blocky, but the only safe option for label maps, where averaging classes is meaningless), bilinear (the smooth, cheap default, mildly soft), and bicubic / Lanczos (sharper, via a wider analytic kernel with small negative lobes, at the cost of a little ringing). This is exactly the reconstruction-filter machinery built in full in Chapter 6.5, Resampling and upsampling, in Part 6, Linear methods and Fourier analysis. The warp engine simply *calls* it at each output pixel, so we do not re-derive it here.

The one thing a warp must keep front of mind is the split between enlarging and shrinking. Upsampling reads more densely than the samples sit, so reconstruction uses its native source-scale support (a cubic kernel still spans several source samples). Downsampling throws samples away, and reconstruction alone aliases; detail too fine for the coarser grid folds down into bold false moiré. The fix is the discipline from BASIC restated for warps: prefilter before you downsample: area-average first, equivalently widen the resampling kernel in proportion to the downscale factor and renormalize so each output pixel gathers over its whole input footprint. *Enlarging needs no prefilter; shrinking always does.* This is big lesson L16 (placed in Chapter 6.5, Resampling and upsampling; the frequency-domain *why*, folding above the new Nyquist, is L6.8 show, Chapter 6.3, Fourier), and it governs every geometric resample in this part: the shrinking corners of a rectification, a stabilization crop, optical-flow rendering. We lean on it throughout and flag it where it bites.

Specifying the warp I: parametric models and the degrees-of-freedom (DOF) ladder

OPTIONAL
in the book ↗

We have a transport engine; we now need maps f to feed it. The easiest maps are parametric, low-parameter functions you fit to a handful of correspondences and then apply everywhere. They form a natural degrees-of-freedom ladder, each rung adding freedom and, in exchange, giving up a preserved property (Figure 11.1.3).

At the bottom is translation (2 DOF), which preserves everything but absolute position. Add rotation and you get a rigid / Euclidean transform (3 DOF), which preserves lengths and angles. Add uniform scale and you get a similarity (4 DOF), which preserves angles and length *ratios*, that is, shape. Loosen further to an affine map (6 DOF),

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = A \begin{pmatrix} x \\ y \end{pmatrix} + \mathbf{t}, \tag{11}$$

which preserves parallelism and ratios *along* a line but no longer angles or lengths: straight lines stay straight, parallel lines stay parallel, and a square becomes a parallelogram. At the top of the practical ladder is the projective transform or homography (8 DOF),

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = H \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \qquad (x, y) \mapsto \left(\frac{x'}{w'}, \frac{y'}{w'} \right), \tag{12}$$

which preserves straight lines only. Parallels may now converge to a vanishing point, and a square can become any quadrilateral. This is the rung that does genuine perspective, and it is the engine of the next chapter's rectification and of panorama stitching.

The homography looks non-linear in (x, y) because of the division by w', and that is exactly why we use homogeneous coordinates: in homogeneous coordinates it is a single 3×3 matrix multiply followed by the perspective divide, restoring linearity to the part that matters. A plane induces a homography between views even when the camera translates. Pure camera rotation also induces a homography, for points at any depth, provided the camera model is appropriate. These are two distinct reasons a homography can align images without a per-pixel depth map. We do not re-derive any of this here; the full development lives in FUNDAMENTALS: Pinhole, and the *solve* for H from correspondences is built in Chapter 20.1, Manual panorama stitching from multiple views, whose homography machinery this chapter shares wholesale.

A nondegenerate affine fit needs three noncollinear point pairs; a homography needs four pairs in general position. Additional pairs support a least-squares fit, but outliers require rejection or a resistant objective. Normalize coordinates for a homography's linear initialization, then evaluate geometric reprojection error. Chapter 12.5, Robustness: the ratio test and RANSAC develops the outlier handling.

The strengths and the limit are two sides of one coin. Parametric warps are global, smooth, and low-parameter, which makes them a good choice when the deformation really *is* rigid, affine, or projective: a flat surface, a rotating camera, a lens model. But by the same token they cannot express a local, free-form deformation such as a smile, a face morph, or a hand-drawn stretch of one limb. For that we need a field built from sparse, local control, which is the next section.

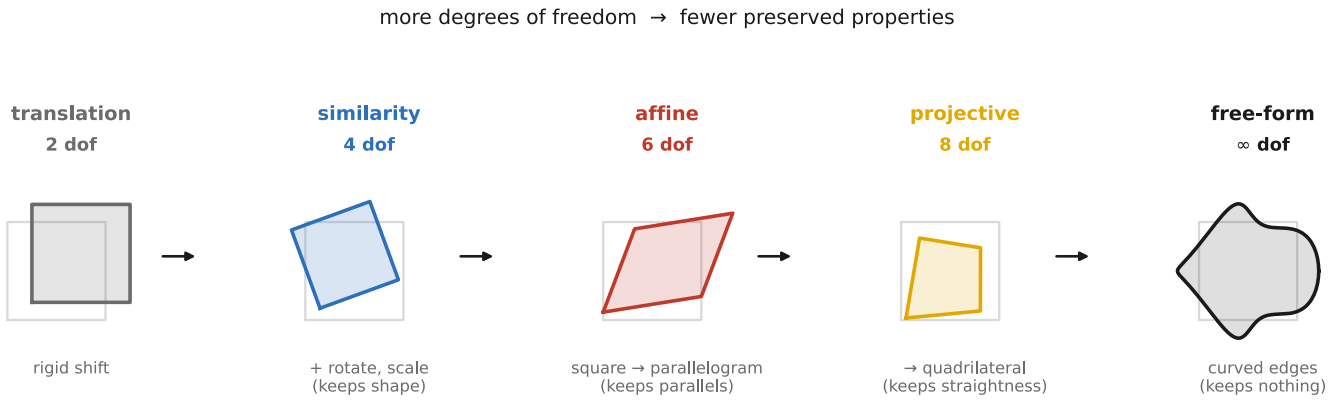
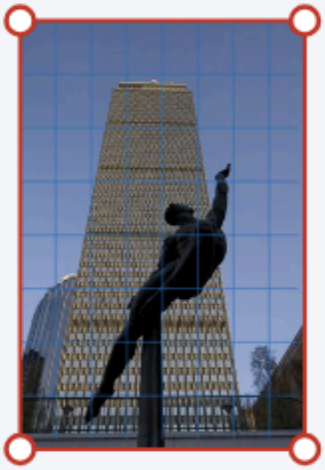


Figure 11.1.3 (reproduced). *The degrees-of-freedom ladder. A unit square under, left to right, translation (2) → similarity (4) → affine (6) → projective (8) → free-form (∞). Each rung adds freedom and breaks a preserved property: similarity keeps shape, affine keeps parallelism (square → parallelogram), projective keeps only straightness (square → quadrilateral, parallels converging), and free-form keeps nothing globally.*

The homography, by hand



Homography matrix (image-space; display placement follows)

1.000	0.0000	0.0000
0.0000	1.000	0.0000
0.0000	0.0000	1

Reset to identity | Grid

Upload image | façade

Figure 11.1.4 (reproduced). *The homography by hand (interactive). Drag the four corners to reproject the image (the top, 8-DOF rung of the ladder above) and the* 3×3 *matrix updates live; edit the matrix and the corners move instead. Zero the bottom-row perspective terms* (h_{31}, h_{32}) *and the warp drops to affine (parallels preserved); restore them for full projective freedom. The four draggable corners are exactly the four point-correspondences that fix all eight degrees of freedom, as long as no three of them line up. A matrix edit that would make the transform singular or numerically unstable is refused and the last valid transform kept (Reset restores the identity), and where the warped image sits on screen is a separate display offset, not part of the matrix.*

This is regression with the warp parameters as the unknowns. Each point pair contributes two coordinate equations. An affine fit is linear in its six parameters. A homogeneous linear initialization for a homography instead finds a null vector, commonly by singular value decomposition, before any geometric refinement. Counting equations explains the minimum number of pairs; checking their configuration explains why that count alone is insufficient.

Specifying the warp II: free-form warps from sparse correspondences

OPTIONAL
in the book ↗

When the deformation is local, the user supplies a sparse set of correspondences, such as a few control points or pairs of feature segments, and we must manufacture a dense warp field f, defined at *every* pixel, by interpolating those sparse constraints. Three constructions interpolate point controls, trading smoothness against locality: a triangulation, a grid of splines, and a thin-plate spline. Feature lines, which take segments instead of points, have their own section next, and a further idea layers a shape-preserving regularizer on top. The canonical reference across all of this is Wolberg's *Digital Image Warping*.

The simplest is the mesh / triangulation warp, which is piecewise-affine. Triangulate the control points (Delaunay, say); inside each triangle, the three vertex correspondences exactly determine an affine map (6 DOF, 3 points), so you warp each triangle by its own affine. It is fast, exact at the controls, and the standard morphing workhorse. Its weakness is that it is only C^0, continuous but creased, with derivative jumps at triangle edges, and its quality depends on how well the triangulation is chosen (Figure 11.1.5).

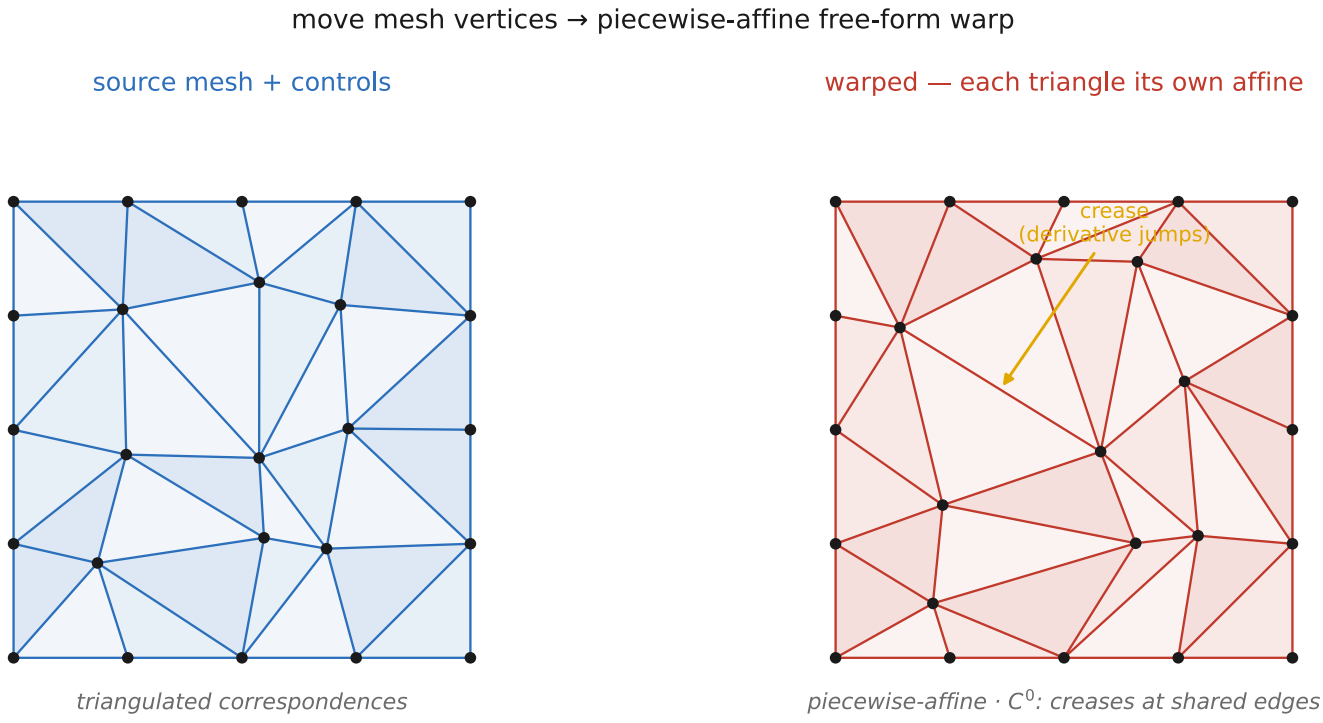


Figure 11.1.5 (reproduced). *Piecewise-affine mesh warp. The control correspondences are triangulated; each triangle's three vertex matches determine its own affine map, and the triangle is warped by it. Exact at the controls and fast, but only C^0: the field creases along triangle edges, where the derivative jumps.*

Triangulation mesh warp

SUBJECT ◉ Frédo (photo) ○ werewolf (photo) ○ drawn face ○ checkerboard

☑ show mesh ☐ show rest mesh Delete vertex Reset

7 vertices (3 added) · 8 triangles

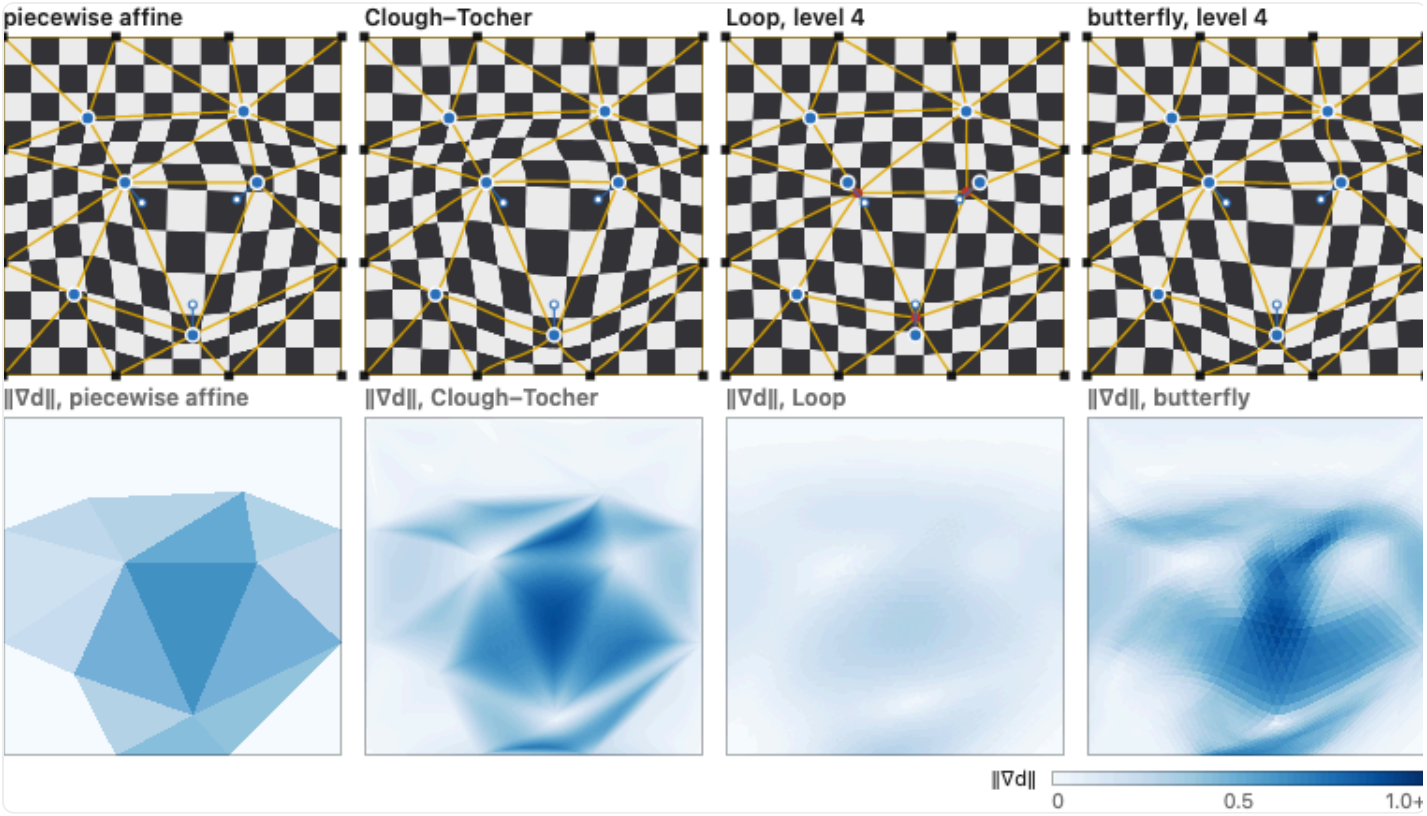
Near a vertex → **drag** to deform · far from any vertex → **click** to add · selected vertex → **Delete** · corners are pinned.

Each triangle's three vertex correspondences (rest → current) determine its own **2×3 affine** map, so the deformation is a different linear map inside every triangle — continuous across shared edges (both triangles agree on the two shared vertices) but **kinked** there, because the gradient of the map jumps. We render by the **inverse** map: rasterize each *destination* triangle and, for each pixel, take its barycentric coordinates in the current vertices and read the source image at the *same* barycentric point of the rest vertices, bilinearly sampled — so it is hole-free and exact at the controls. The four image corners are always present as pinned vertices, so the triangulation covers the whole frame and the boundary stays put. With nothing dragged the map is the identity (output = input). Dragging one interior vertex distorts only the triangles that touch it.

Figure 11.1.6 (reproduced). *Interactive: a triangulation (piecewise-affine mesh) warp. Click to add control vertices; the demo Delaunay-triangulates them and lets you drag any vertex to deform the picture, each triangle carrying its own affine map. It opens on a photograph of the author, with control vertices on the eyes and mouth; the same*

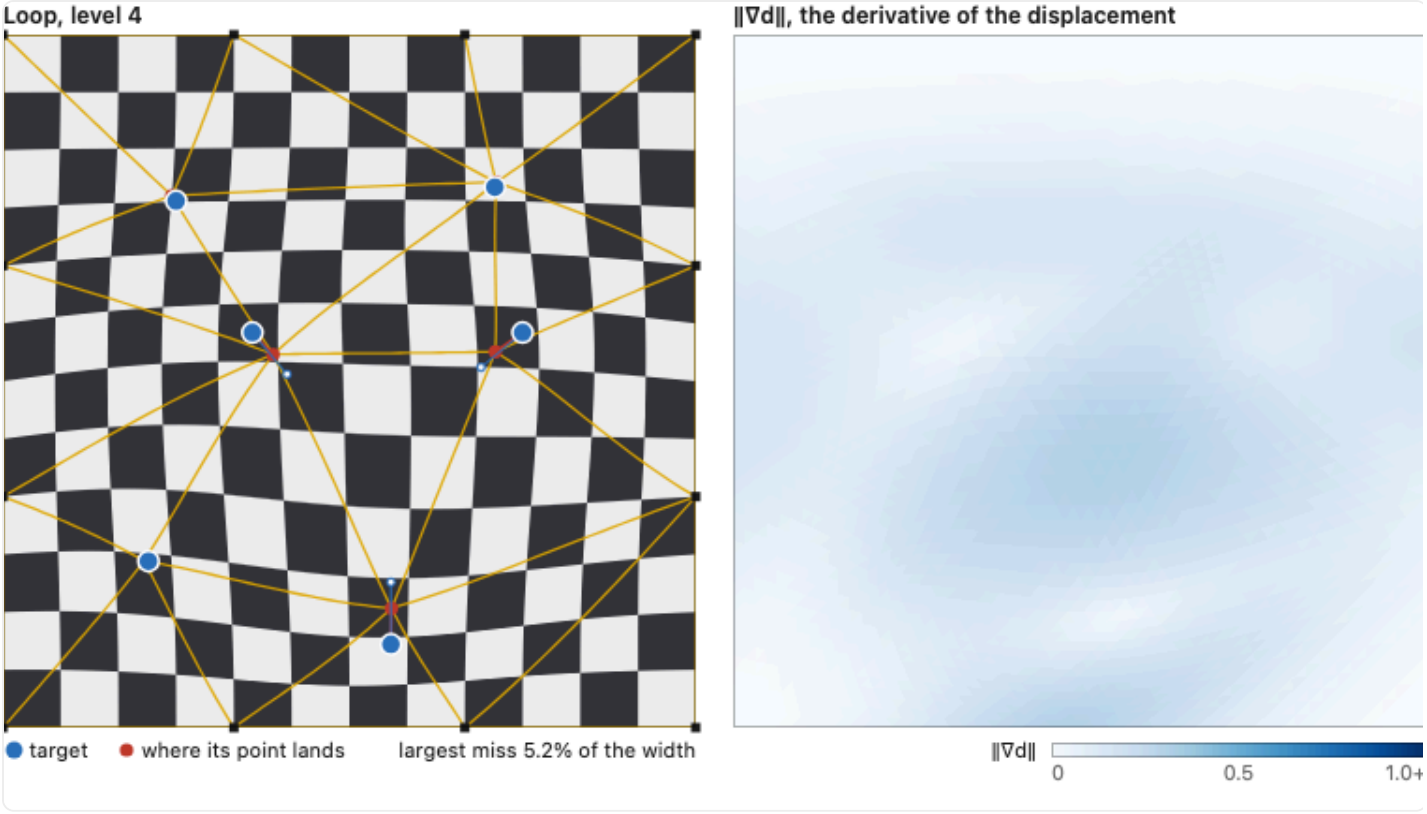
photograph's werewolf counterpart from the morphing demo, a drawn face and a checkerboard are the other subjects. Toggle the checkerboard subject to see the tell-tale C^0 creasing along triangle edges, and note that dragging does not re-triangulate (the topology is fixed once the points are placed).

A triangulation can be made smooth without giving up its locality. The classical answer comes from finite elements: the Clough–Tocher interpolant splits each triangle into three at an interior point, fits a cubic on each of the three pieces, and matches them so that the field and its first derivatives are continuous across every edge Clough & Tocher 1965. It still passes through the control displacements, needs a gradient estimated at each vertex, and is the standard smooth interpolant for scattered two-dimensional data. Subdivision is another route. Subdivision surfaces smooth a triangle mesh by splitting every triangle into four, again and again, and replacing each value by a local average of its neighbors; Loop's rules converge to a surface that is twice differentiable except at vertices with an unusual number of neighbors Loop 1987 (Loop subdivision). Applied in the plane, to a displacement or any other value stored at the vertices, the same rules turn the creased piecewise-affine field into a smooth one. Davis, Levoy and Durand used exactly this to blend the views of an unstructured light field: each captured view gets a weight of one at its own vertex of a triangulation of the viewpoints and zero at the others, and subdividing those weights gives smooth blending functions where barycentric weights would crease Davis et al. 2012. The catch is that Loop's scheme approximates rather than interpolates: the limit does not pass through the vertex values, so a dragged control point does not end up where it was dropped. That is harmless for blending weights and wrong for a warp, which needs either a sparse linear solve for the vertex values whose limits hit the targets, or an interpolating scheme. The butterfly scheme keeps every existing vertex and places each new one with a stencil of eight neighbors shaped like a butterfly, so its limit passes through the controls Dyn, Levin & Gregory 1990 (butterfly); a modified version extends it to meshes of any connectivity and makes it continuously differentiable everywhere Zorin, Schröder & Sweldens 1996 (modified butterfly). Figure 11.1.7 compares the four on one set of controls (view compare). A map of the warp's derivative shows ridges along the triangle edges for Clough–Tocher and butterfly, which are only once differentiable, and none for Loop. Unsolved, Loop's limit misses a dragged control by up to about five percent of the image width and shrinks the whole warp; solving for the vertex values removes the miss (view loop). The last view shows the light-field use: one vertex at one and the others at zero, piecewise linear against Loop (view weights).



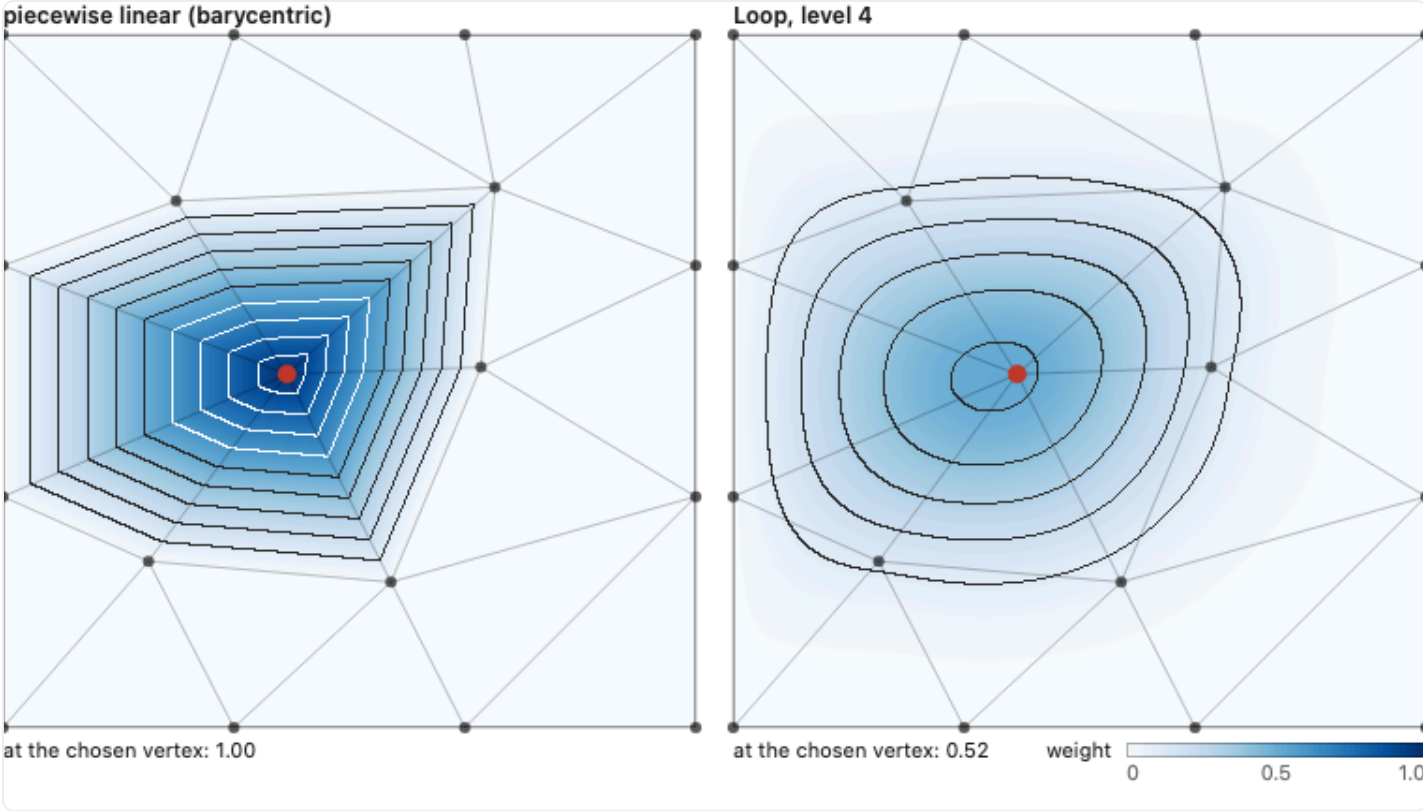
piecewise affine | Clough–Tocher | Loop | butterfly | blending weights | all four

Figure 11.1.7, the compare view (reproduced). *Interactive: one set of control points, Delaunay-triangulated, and the same displacements d spread over the plane four ways: piecewise affine, Clough–Tocher, Loop subdivision and butterfly subdivision. Drag the blue points (the black squares on the frame are pinned); the small hollow dots are where each point started. The bottom row maps $\|\nabla d\|$: the piecewise-affine map is a mosaic whose facet boundaries are the creases; Clough–Tocher and butterfly are continuous but keep ridges where their pieces meet, since they are C^1 and not C^2; Loop's is smooth. Loop also misses: the red segment runs from where a point was dropped to where the picture under it lands. The subdivision level sets how many times Loop and butterfly split the mesh. The checkerboard shows the creases best; the portrait and the werewolf are the other subjects.*



piecewise affine | Clough–Tocher | Loop | butterfly | blending weights | all four

Figure 11.1.7, the loop view (reproduced). *Interactive: Loop subdivision of the warp's displacements, the rest and displaced positions of the vertices subdivided together. Loop approximates: each blue point is where a control was dropped, the red dot is where the picture under it lands, and the readout gives the largest miss. Tick "solve for targets" to replace the control values by those whose limits hit the targets, a small linear solve with Loop's limit stencils: the misses drop to zero and the warp stays smooth, but it bends more strongly, as the map of $\|\nabla d\|$ on the right shows.*



piecewise affine | Clough–Tocher | Loop | butterfly | blending weights | all four

Figure 11.1.7, the weights view (reproduced). *Interactive: blending weights over the same triangulation, after Davis, Levoy and Durand Davis et al. 2012. One vertex holds weight 1 and every other vertex 0. Left, the barycentric (piecewise-linear) hat, whose contour lines kink at every edge; right, the same values subdivided with Loop: smooth, reaching past the vertex's immediate neighbors, and below 1 at the vertex itself. Click any vertex to move the weight there. Both sets of weights sum to one everywhere.*

The first morph in a feature film used a different kind of mesh, a grid with splines between its nodes. For *Willow* (1988), in which a sorceress changes through a series of animals back into a woman, Doug Smythe at Industrial Light & Magic wrote a two-pass mesh warp Smythe 1990 (two-pass mesh warping). The artist lays a regular grid of control points over the first image and moves its nodes onto the matching features of the second. The two meshes have the same topology, so node (i, j) of one corresponds to node (i, j) of the other, and nothing needs to be triangulated. Between the nodes the warp is interpolated with cubic splines (Catmull–Rom splines in Wolberg's description), not with flat triangles, so the field is smooth across the grid lines and shows none of the creases of a triangulation Wolberg 1990 (Figure 11.1.8, stage grids). We follow the algorithm on a photograph in Figure 11.1.8, one step per stage.

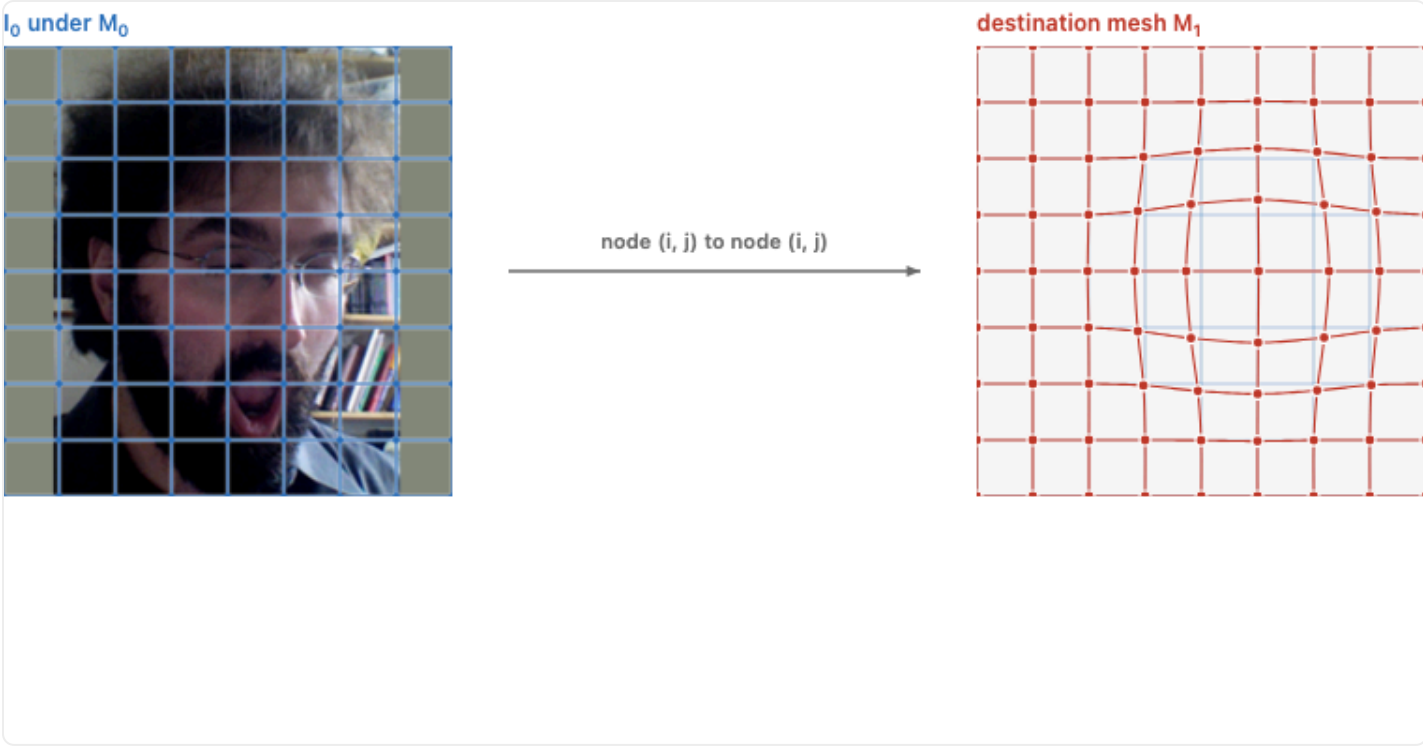
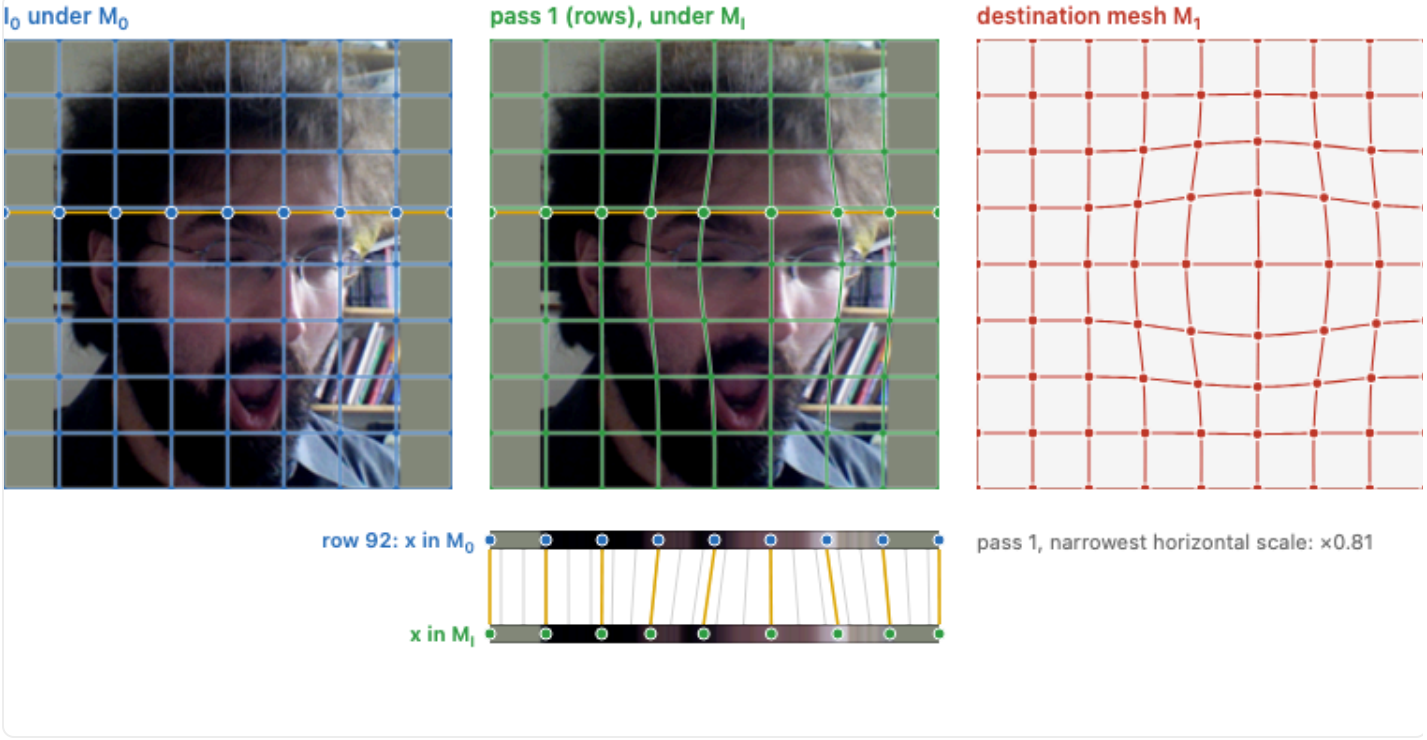


Figure 11.1.8, stage 1 (reproduced). *The two meshes of Smythe's warp. Left, the source mesh M_0, a regular grid of nodes over the photograph I_0; right, the destination mesh M_1, with the same topology, so node (i, j) of one matches node (i, j) of the other (M_0 faint behind it). Drag any node of M_1, or choose a preset: a bulge, a swirl, rotations by 30°, 60° and 80°, or the identity. The grid lines are drawn as the splines the warp uses, Catmull–Rom through the nodes or straight between them. Grid cells sets the number of cells per side; changing it keeps the motion already dragged.*

The warp itself is computed in two one-dimensional passes, following Catmull and Smith's separable scheme Catmull & Smith 1980. An intermediate mesh M_I takes its x coordinates from the destination mesh M_1 and its y coordinates from the source mesh M_0. The first pass moves pixels only horizontally, from M_0 to M_I: a spline through each column of nodes gives, on every image row, the x positions where the vertical grid lines cross that row in both meshes, and a spline through those crossings maps every x of the row to its new position. The row is then resampled with a one-dimensional filter. The second pass does the same vertically, from M_I to M_1 (Figure 11.1.8, stages pass1 and pass2).

Algorithm 3

```
mesh_warp(img, M0, M1):                          # M0, M1: grids of node positions, same
    MI = nodes with x from M1 and y from M0
    for each row y of the image:                                  # pass 1: horizontal
        xs0 = x where each column of M0 crosses row y             # spline through that colum
        xsI = x where each column of MI crosses row y
        g = spline through the pairs (xs0, xsI)                   # 1-D map for this row
        tmp[y, :] = resample_1d(img[y, :], g)
    for each column x of tmp:                                     # pass 2: vertical
        ysI = y where each row of MI crosses column x
        ys1 = y where each row of M1 crosses column x
        h = spline through the pairs (ysI, ys1)
        out[:, x] = resample_1d(tmp[:, x], h)
    return out
```



◀ | 1 mesh warp: M_0 and M_1 | 2 mesh warp: pass 1, rows | 3 mesh warp: pass 2, columns | 4 against triangles and bilinear cells | 5 mesh morph: M(t) | ▶ | ↓ settings from the previous stage

Figure 11.1.8, stage 2 (reproduced). *Pass 1 moves pixels along rows only. The intermediate mesh M_I (middle) takes its x coordinates from M_1 and its y coordinates from M_0, so its horizontal grid lines stay where M_0's are. On the highlighted row (the row slider) the dots mark where each vertical grid line crosses it, in M_0 on the left and in M_I in the middle. The strip below shows that row before and after the pass: each crossing in M_0 is joined to its partner in M_I, and the gray lines follow the spline through those pairs, the row's one-dimensional map. The readout on the right is the narrowest horizontal scale the pass applies anywhere in the picture. Choose "rotate 80°": the intermediate image collapses to a sliver, squeezed to* $\cos 80^\circ \approx 0.17$ *of its width.*

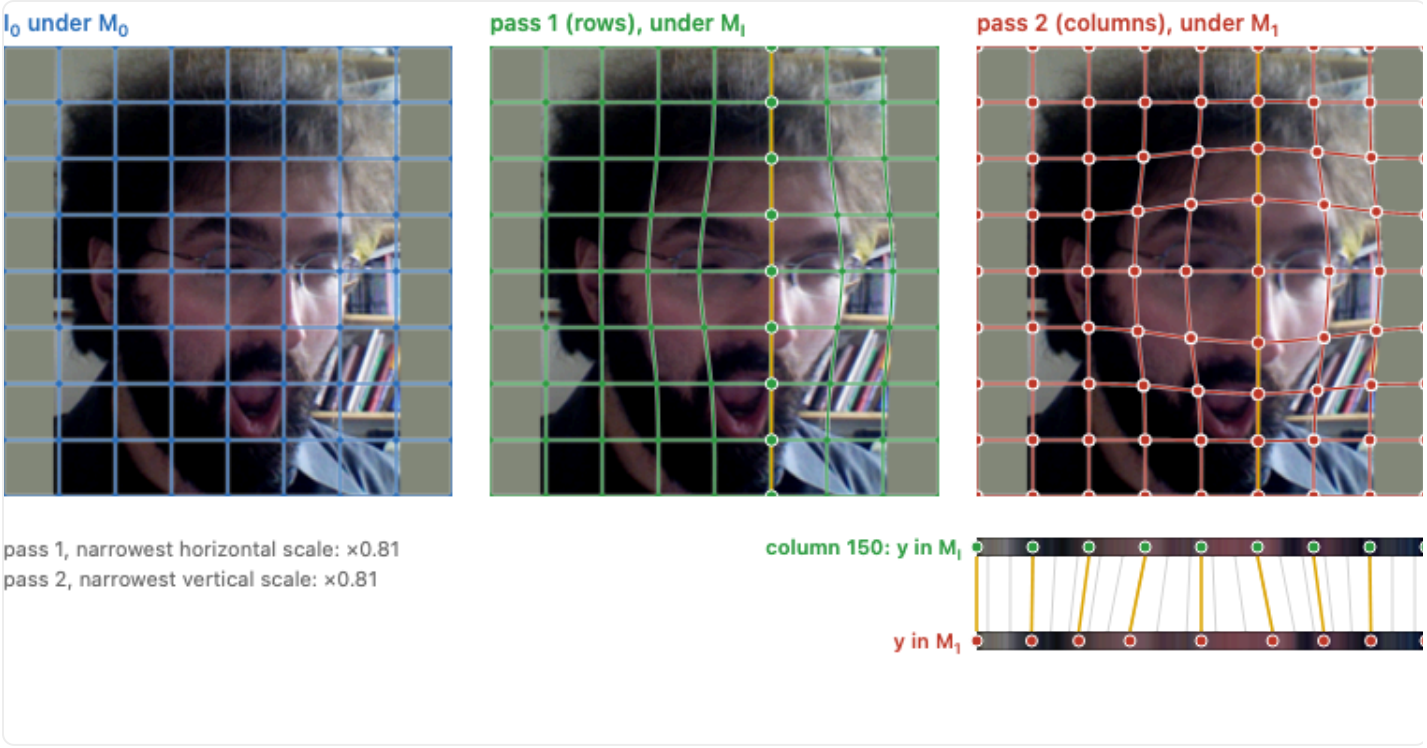


Figure 11.1.8, stage 3 (reproduced). *Pass 2 does the same along columns, from M_I to M_1. On the highlighted column (the column slider) the dots mark where each horizontal grid line crosses it, in M_I in the middle and in M_1 on the right, and the strip shows the column before and after the pass, joined at those crossings. The right panel is the warped photograph. After "rotate 80°", pass 2 stretches each column by $1/\cos 80^\circ \approx 5.8$ to undo the squeeze of pass 1, and the detail lost in the sliver does not come back. Drag a node past its neighbor and the grid lines fold over; the readouts then say so.*

Each pass is a set of one-dimensional resamplings, which are cheap and easy to filter well, and this mattered on the computers of 1988. The separable scheme has a known weakness: where the warp rotates a region by a large angle, the first pass squeezes it into a sliver before the second pass expands it again, the bottleneck problem of two-pass warps. For a rotation by θ the first pass compresses the region to $\cos\theta$ of its width and the second stretches it back by $1/\cos\theta$. The decomposition itself is exact; the loss comes from resampling a squeezed image, and it grows quickly with the angle: on a portrait, about 1 gray level of error against a one-pass warp at 30°, 4 at 60° and 12 at 80° (stage compare, which also sets the grid warp beside triangles and bilinear cells). The grid lines also must not cross, or the one-dimensional maps fold over. For a morph, the node positions are interpolated, $M(t) = (1-t)\,M_0 + t\,M_1$, both images are warped to $M(t)$, and the two warped images are cross-dissolved (Chapter 11.3, Morphing; stage morph).

◀ | 1 mesh warp: M_0 and M_1 | 2 mesh warp: pass 1, rows | 3 mesh warp: pass 2, columns

4 against triangles and bilinear cells | 5 mesh morph: M(t) | ▶ | ↓ settings from the previous stage

Figure 11.1.8, stage 4 (reproduced). *The same node motion warped three ways. Left, the two-pass spline warp; middle, triangles, two per cell, each with its own affine map (the diagonals drawn faint); right, bilinear interpolation of the node positions in each cell. Turn the grids off and drag a node: the triangles crease along cell edges and diagonals, the bilinear cells along cell edges, the Catmull–Rom warp nowhere. The readouts are RMS differences from the bilinear-cell warp, in 8-bit levels. Under a rotation all three compute the same affine map, so what remains in the two-pass result is what its two resamplings lose: on this photograph, with 8 by 8 cells, about 1 level at 30°, 4 at 60° and 12 at 80°.*

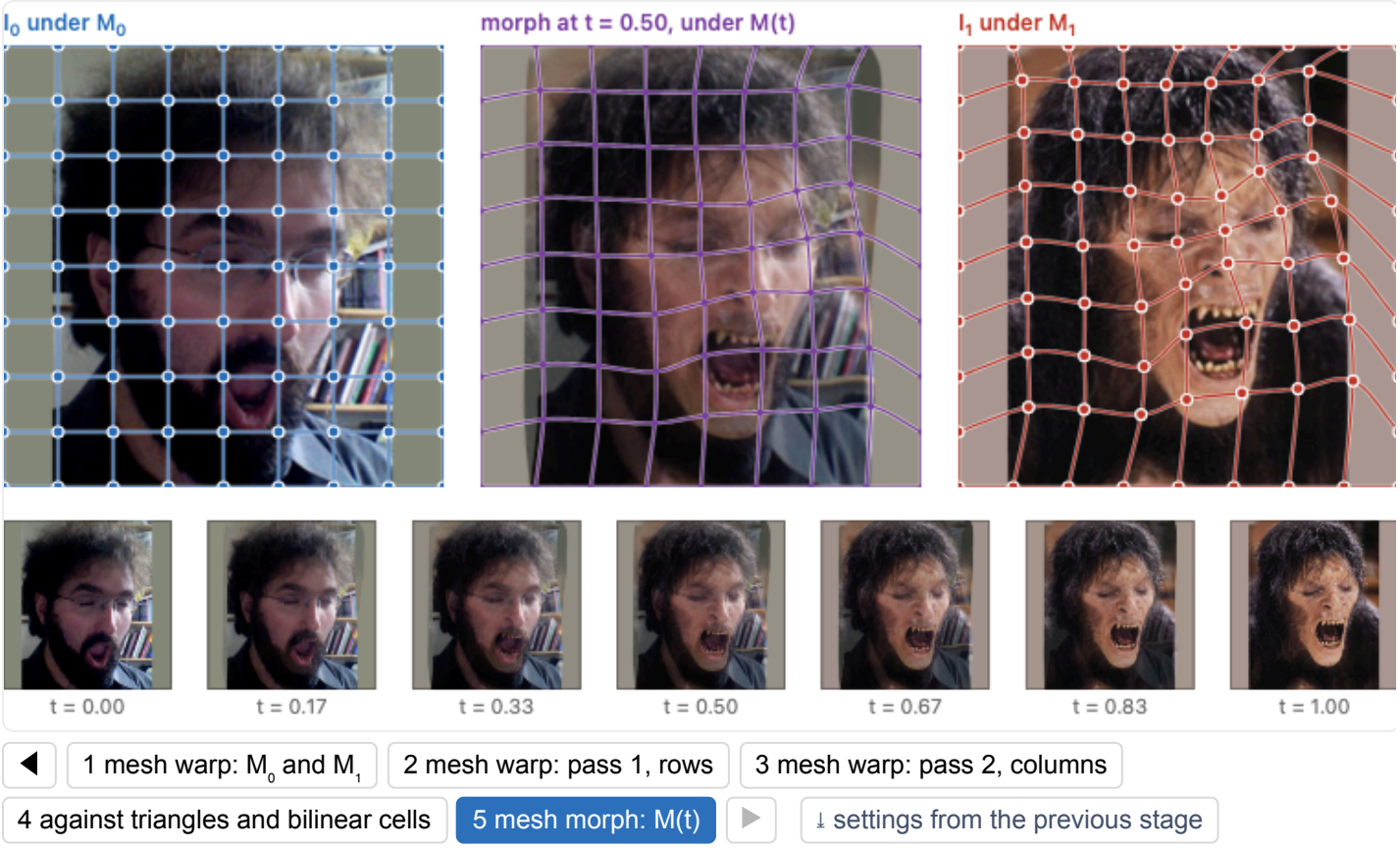


Figure 11.1.8, stage 5 (reproduced). *The mesh morph. The middle panel is the morph at t: the meshes are interpolated, $M(t) = (1-t)\,M_0 + t\,M_1$, both photographs are warped to $M(t)$ by the two-pass warp, and the two results are cross-dissolved with the same t; the row below shows seven values of t. M_1 on the werewolf was not placed by hand: each node of M_0 on the portrait was sent through the Beier and Neely field warp of the morphing chapter, with its nine segment pairs, which gives for every point of the portrait the matching point of the werewolf; the border nodes stay on the frame. Drag nodes on either face to improve the match.*

A grid trades freedom for smoothness and speed. The artist must work within the grid's topology and often moves many nodes to describe one feature, an edge that runs diagonally across the cells for instance. That burden is what Beier and Neely's feature lines, below, set out to remove Beier & Neely 1992. Later spline methods kept the grid but freed the controls: a tensor-product B-spline grid can be fitted to scattered points or lines, with a coarse-to-fine hierarchy of grids that adds detail only where the constraints require it Lee, Wolberg & Shin 1997.

For smoothness without creases, the thin-plate spline (TPS), a radial-basis-function interpolant. Fit a field that passes through the control displacements while bending as little as possible:

$$f(\mathbf{p}) = A\mathbf{p} + \mathbf{t} + \sum_i w_i\,\phi\big(\|\mathbf{p} - \mathbf{c}_i\|\big), \tag{13}$$

an affine "global" part plus a sum of radial basis functions centered at the control points $\mathbf{c}_i$. The thin-plate kernel $\phi(r) = r^2 \log r$ is the one that minimizes integrated second-derivative (bending) energy, giving the smoothest possible interpolant, with the literal physical analogy of a thin metal plate bent to pass through pins at the

constraints. The weights w_i (together with $A, \mathbf{t}$) come from a small linear system. The result is smooth everywhere, with no creases, but it has global support: every control point influences every pixel, so it can be *less local* than a mesh, and the solve costs roughly $O(n^2)$–$O(n^3)$ in the number of controls. It is the go-to for smooth medical and facial registration and for morphometrics, following Bookstein.

The catch shared by all these constructions is that interpolating *positions* alone lets shapes shear and squash unnaturally: drag one handle and an unconstrained fit will stretch and skew the whole region. The fix is a shape-preserving regularizer: as-rigid-as-possible (ARAP) Igarashi et al. 2005 or moving-least-squares (MLS) Schaefer et al. 2006 deformation, which keeps each local neighborhood close to a pure rotation so a dragged limb bends without stretching. That regularizer is important enough to get its own chapter: we defer it to Chapter 11.5, Shape-preserving warping and only flag here that plain position-interpolation is *not* shape-preserving.

Compose maps before resampling where possible. If forward maps are f_1 then f_2, the combined backward lookup is $f_1^{-1} \circ f_2^{-1}$. Repeatedly resampling the image adds blur and rounding. Test an identity map, an integer translation and an affine ramp; declare whether integer coordinates denote pixel centers.

For a thin-plate spline in two dimensions, a common basis is $\phi(r) = r^2 \log r$ with $\phi(0) = 0$. The radial coefficients need affine side constraints, such as $\sum_i w_i = 0$ and $\sum_i w_i \mathbf{c}_i^\top = 0$. Smooth interpolation does not guarantee an invertible map. Reject collapsed triangles or folded regions if the application requires one-to-one transport.

The constructions trade locality, smoothness and control against one another, and the choice usually follows from the input you have. A piecewise-affine mesh is fast and exact at the controls, and it suits a face morph where creases along triangle edges are tolerable. A thin-plate spline gives a smooth field from a few scattered points, at the price of global support, so every control moves every pixel. Beier–Neely suits controls that are naturally lines, such as the edge of a jaw or an eyebrow. When dragging handles must not shear or squash the content, use a shape-preserving warp (Chapter 11.5, Shape-preserving warping). A camera rotation, a plane or a lens model calls for a parametric fit by least squares instead. In every case the application step is the same: build the map, then inverse-warp and resample.

Liquify: brush-driven warping

OPTIONAL
in the book ↗

Everything this chapter built, a dense map f authored from sparse local input, then applied by resampling, is exactly what you are doing when you push pixels around with Photoshop's Liquify tool (the cosmetic "slim the waist / widen the smile" slider in spirit). Adobe does not publish the algorithm, but its documented behavior is the standard mesh warp. Under the hood Liquify keeps an editable mesh over the image, which you can show on screen and save to disk, and each brush stamps a smooth, radially-weighted displacement onto the mesh vertices beneath the cursor: Forward Warp pushes them along the drag, Twirl rotates them, Pucker and Bloat pull them in or out. The displacements accumulate as you stroke, Freeze masks pin regions you want protected, and Reconstruct relaxes the field back toward the identity. Because the brush specifies *where pixels go*, it authors the warp in the forward direction; the picture you see is that deformed grid resampled with interpolation. This is the forward-vs-inverse and reconstruction-filter machinery of this chapter, now driven by a paintbrush.

Face-aware deformation can use detected landmarks as semantic handles for eye, mouth or jaw adjustments. The detector and deformation model both affect the result; similar controls do not establish that different software uses an identical mesh algorithm.

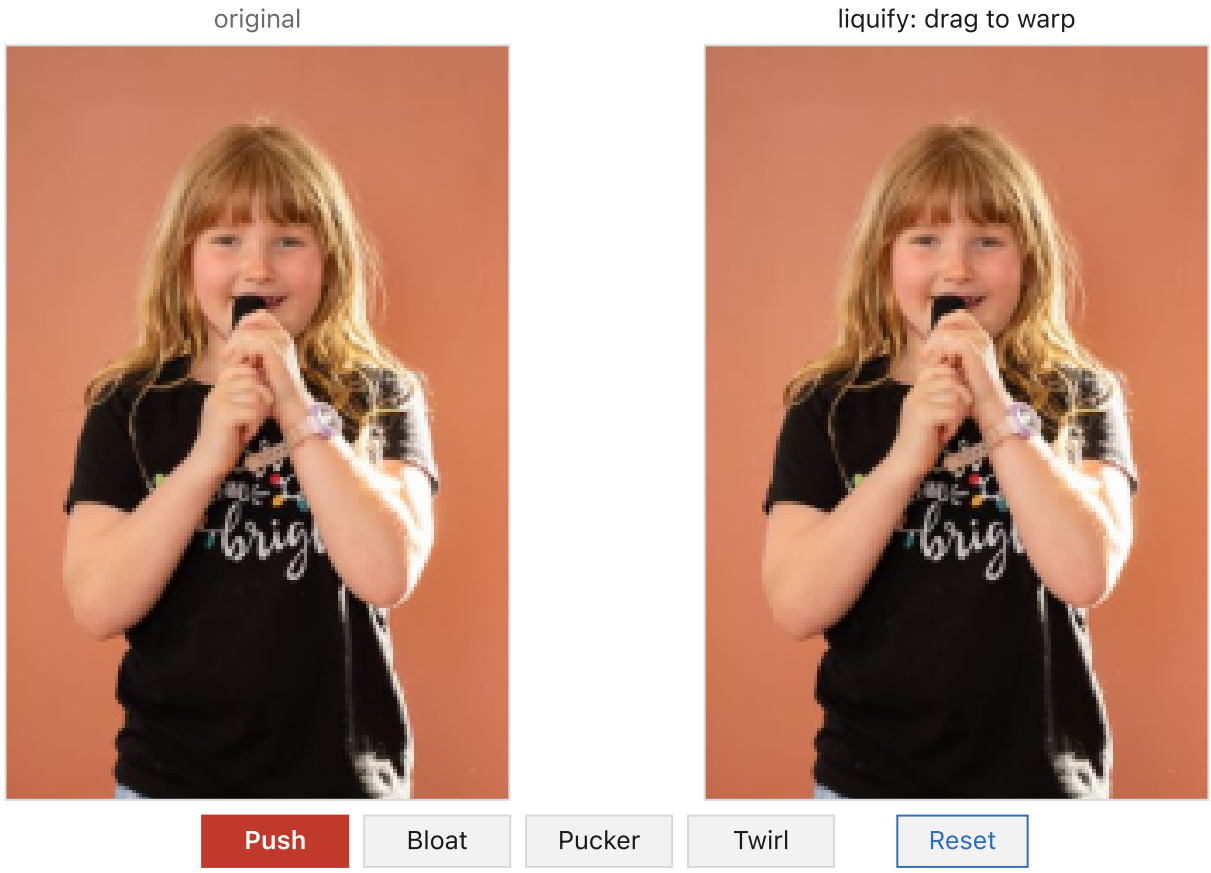


Figure 11.1.12 (reproduced). *Interactive Liquify, a mesh warp driven by a brush. Each brush stamps a smooth, radially-weighted offset into a dense displacement field under the cursor: Push shoves pixels along the drag (forward warp), Bloat balloons them outward, Pucker sucks them in, Twirl rotates them. The offsets accumulate as you stroke, and Reset relaxes the field back toward the identity (Liquify's "Reconstruct"). The brushes edit an inverse displacement field: every output pixel reads the source at $p + d(p)$ with bilinear interpolation and clamped borders, so there are no holes. Strokes add their offsets to that field rather than composing warps exactly, an approximation that stays close for gentle edits and can fold the image over itself under large ones. This is exactly this chapter's forward-vs-inverse machinery, now driven by a paintbrush. Turn on "show mesh" to watch the lattice bend; the mesh approximates the forward displacement as $-d$, so it is a guide rather than the exact forward map. Nothing here preserves shape: keep pushing and the anatomy genuinely distorts, which is why over-Liquified portraits read as wrong and why the shape-preserving warps of the next chapter exist. Portraits by Frédo Durand.*

A deformation brush also makes the need for shape constraints visible. A position edit can bend a nearby architectural line or narrow a limb along with the intended feature. The operator must inspect the surrounding geometry, protect it with controls, or use a deformation model that penalizes the unwanted change. Chapter 11.5, Shape-preserving warping develops that last option.

Morphing

in the book ↗

📎 **Problem set**

PS5 implements a full Beier–Neely morph (warp + cross-dissolve). → Chapter 1.5, Problem sets (appendix).

Two photographs of two different faces, and the goal is a film clip that turns one into the other, not as a cut or a fade, but a single face that *becomes* another, the way the dancers shapeshift through each other at the end of Michael Jackson's *Black or White*. The obvious tool is the cross-dissolve: at each in-between time t, set every output pixel to the linear blend of the two inputs,

$$I_t[y, x] = (1 - t)\, I_0[y, x] + t\, I_1[y, x]. \tag{14}$$

Run it and the result disappoints in one very specific way. Halfway through, at $t = 0.5$, you do not see a plausible single face. You see a ghost: four faint eyes, two noses, a doubled mouth, all translucent and superimposed. The blend is doing exactly what you asked, averaging colors at fixed pixel locations, but the two faces' features *sit at different pixels*. The left image's left eye lands on the right image's eyebrow; the right image's mouth lands on the left image's chin. Averaging colors across that misalignment double-exposes every feature.

The diagnosis is sharp, and it is the whole chapter. A cross-dissolve interpolates only the range, the color at each location. To get a believable in-between you must *also* interpolate the domain, the *location* of every feature, so that both pairs of eyes travel to the same place before you average them. Metamorphosis is what you get when you do both: warp each image so its features move toward a common intermediate shape, *then* cross-dissolve the now-aligned images. The one idea to hold is the separation of shape and color: interpolate *where things are* and *what color they are* as two independent linear blends, applied together.

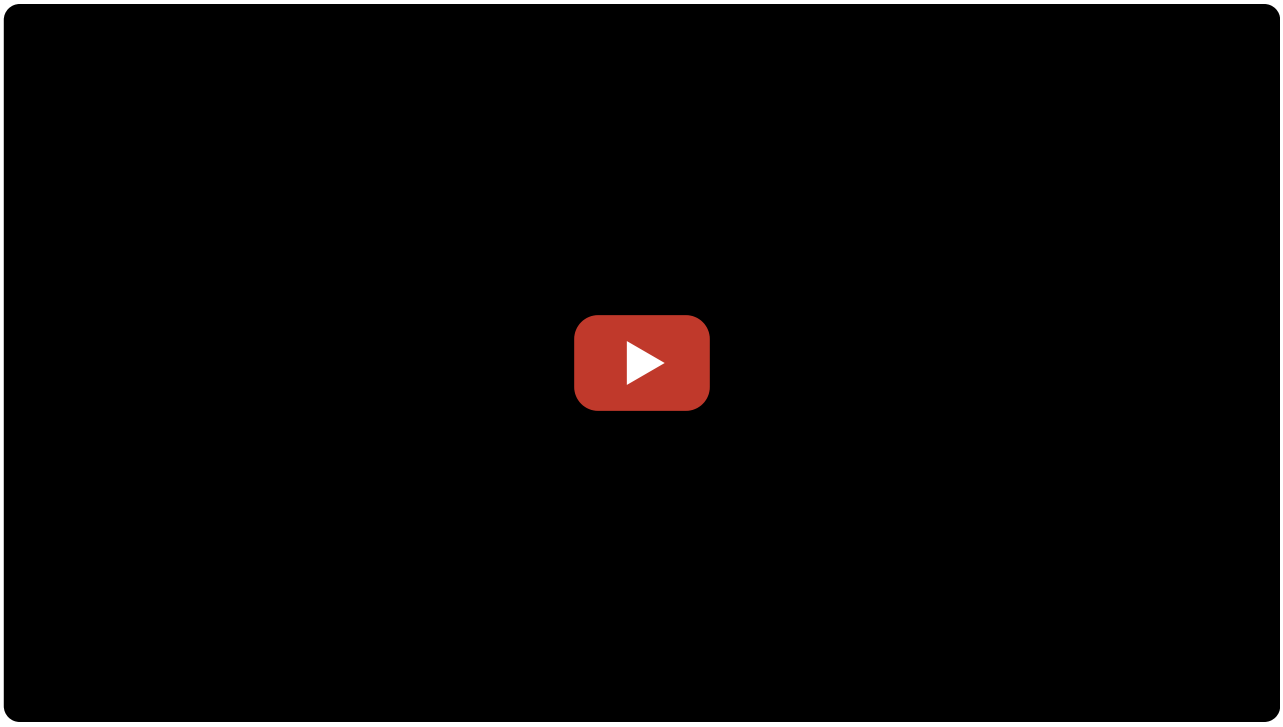
A morph uses correspondences to create an intermediate shape, warps both inputs into it, and blends their appearance. The correspondence model is what prevents doubled features; a color blend alone cannot do that.

BIG LESSON (L17, RECURRENCE)

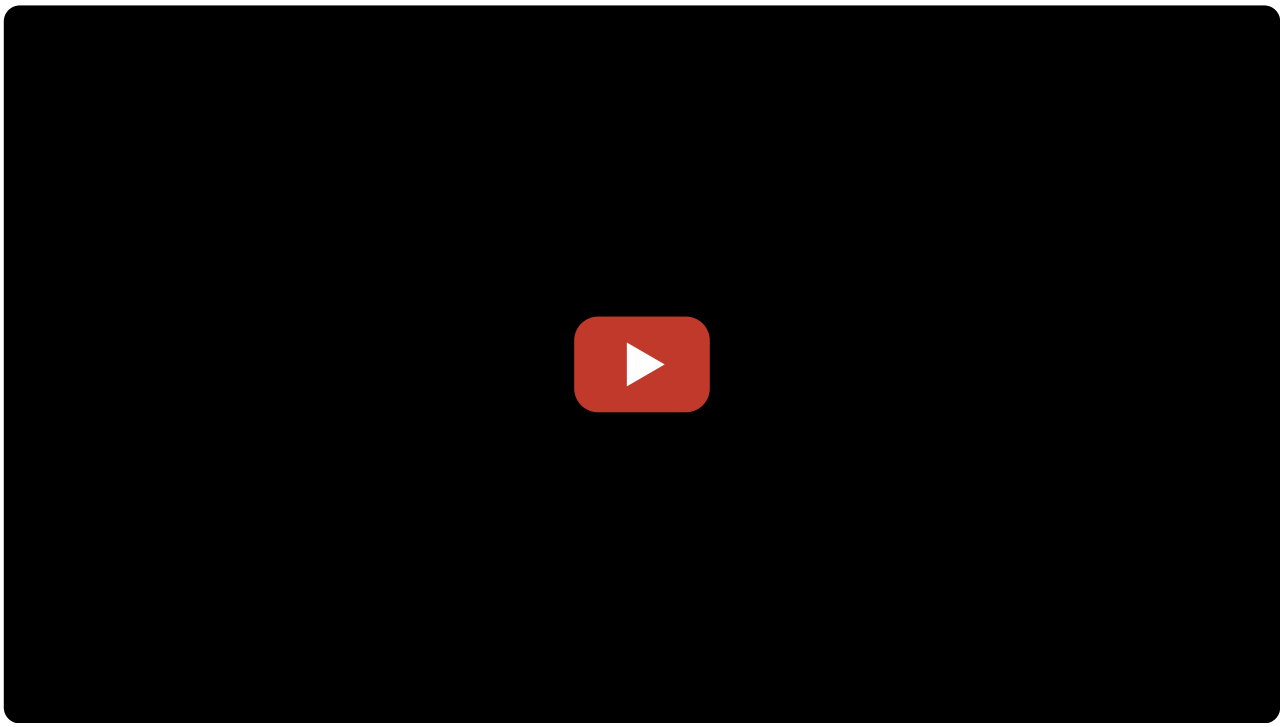
A morph is a correspondence field plus transport, then a blend. To interpolate between two images you cannot average their colors where they sit; you must first establish a correspondence between their features, transport (warp) both images along that correspondence to a common intermediate shape, and *only then* blend. Separating the manipulation into a domain part (the warp: *where* features are) and a range part (the color blend: *what* they are), and interpolating each linearly, is what makes the in-between read as one object transforming rather than two pictures fighting. The same three-part shape (*correspond, transport, blend*) drives panorama de-ghosting, multi-frame super-resolution's sub-pixel alignment, and flow-driven frame interpolation; the only thing that changes is *how* the correspondence is obtained (here, sparse user lines; in Chapter 12.8, Optical flow, a dense automatic field).

BIG LESSON (RECURRENCE) — *ALIGN BEFORE YOU BLEND*

Averaging two images only makes sense once their features sit at the same place. A plain cross-dissolve blends colors at fixed coordinates, so any misalignment double-exposes into a ghost; the cure is always to warp into correspondence first, then average. You have already met this move under other names: panorama de-ghosting registers overlapping frames before feathering them, and multi-frame super-resolution aligns sub-pixel-shifted shots before merging (cross-ref Chapter 20.3, Blending). Morphing takes the lesson to its limit: it interpolates the *alignment itself* smoothly over t, rather than aligning once. The dense, automatic aligner that makes this work without hand-drawn lines is Chapter 12.8, Optical flow.



Video 3. Michael Jackson, *Black or White* (1991, PDI): the face-morph finale. The first famous photorealistic face morphing, and the moment "morphing" entered everyone's vocabulary, exactly the *warp-plus-blend* this chapter builds, driven by hand-placed feature correspondences. *(web edition)*



Video 4. Willow (1988, ILM): the first digital morph. ILM's "morf" turning one creature smoothly into another: the original production morph, which predates and motivated the Beier–Neely line-segment method developed below. *(web edition)*

Why a cross-dissolve isn't enough: the ghosting motivation

in the book ↗

The goal is to *smoothly* turn one image into another, where "smooth" carries a strong requirement: each in-between I_t should look like a plausible single image, not a visible blend of two. A face becoming a face; one video keyframe becoming the next.

The naive attempt is cross-fading (a.k.a. cross-dissolve): linearly interpolate colors pixel by pixel, $\text{out}[y, x] = (1 - t)\, I_0[y, x] + t\, I_1[y, x]$. This is the *right* idea for color (at $t = 0$ you get I_0, at $t = 1$ you get I_1, and the color glides linearly between), but it ignores position entirely.

It ghosts because the two images' features are not aligned: eyes, mouth, and silhouette sit at different pixels. Blending colors at fixed locations superimposes a left-image eye onto a right-image cheek, and the result is a translucent double exposure, four eyes, two mouths (Figure 11.3.1). And you generally *cannot* repair this with a single global alignment. Translating, rotating, or scaling one image as a whole can register a couple of features at the cost of the rest, because two faces differ *locally* and *non-rigidly*: the jaws are differently shaped, the eyes differently spaced. No one rigid motion brings all the features into register at once.

So cross-fading interpolates only the range (color). A believable in-between demands that we *also* interpolate the domain, the location of every feature, and that is a warp, the missing half. The plan for the rest of the chapter: get a dense shape interpolation from sparse correspondence, and combine it with the color interpolation we already have.

Beier and Neely field morph, 9 segment pairs; both middle panels use the same t

Figure 11.3.1 (reproduced). *Why a cross-dissolve ghosts, on a man-to-werewolf morph (interactive: drag t; the toggle shows the nine feature segments and where they sit at t). The two endpoints (I_0, a face; I_1, the creature) have their features at different pixels. The naive cross-dissolve at $t = 0.5$, $\text{out} = (1 - t)I_0 + tI_1$, averages colors at fixed locations and so superimposes them: a translucent four-eyed, two-mouthed ghost, not a plausible single face, since the blend handled color but ignored position. The morph panel beside it is what the rest of the chapter builds (warp both images to a common intermediate shape* first *so the features coincide, then cross-dissolve) yielding one coherent in-between face. A single global translate/rotate/scale cannot substitute for the warp, because the two faces differ locally and non-rigidly. "Scale v like u" switches to the variant in which each segment maps by a similarity (Figure 11.1.10), instead of keeping v in pixels as Beier and Neely do.*

Two interpolations: domain (shape) and range (color)

in the book ↗

The cure has two knobs. Name them precisely, because the method is their product.

Range interpolation (color) is the cross-dissolve we already have: at a location x, blend the two colors,

$$C(x) = (1-t)\,C_0(x) + t\,C_1(x), \tag{15}$$

a straight linear blend of pixel values.

Domain interpolation (location) treats a feature *point* as a vector and interpolates *where it is*. For a corresponding pair, a point P in I_0 and the matching point Q in I_1, the in-between location is

$$V = (1-t)\,P + t\,Q. \tag{16}$$

At $t = 0$ the feature sits where I_0 put it; at $t = 1$ where I_1 put it; in between it travels straight along the segment PQ (Figure 11.3.2). This is interpolation of the domain, the coordinate grid itself, as opposed to the color living on it.

To realize a domain interpolation as an image operation we warp: move pixels spatially while leaving their colors alone, $C'(x,y) = C\big(f(x,y)\big)$. In practice we use the inverse / backward map (for each *output* pixel, look up where it came from, $\mathrm{out}(x,y) = \mathrm{im}\big(f^{-1}(x,y)\big)$), so every output pixel is filled (no holes) and resampling is well-defined. This is exactly the engine of Chapter 11.1, Warping; morphing is one of its biggest customers, calling it *twice per frame* (once for each source image).

The morph in one line: interpolate the domain like a vector, then interpolate the range (color). A morph is the product of a shape interpolation and a color interpolation: two independent linear blends, applied together (Figure 11.3.2).

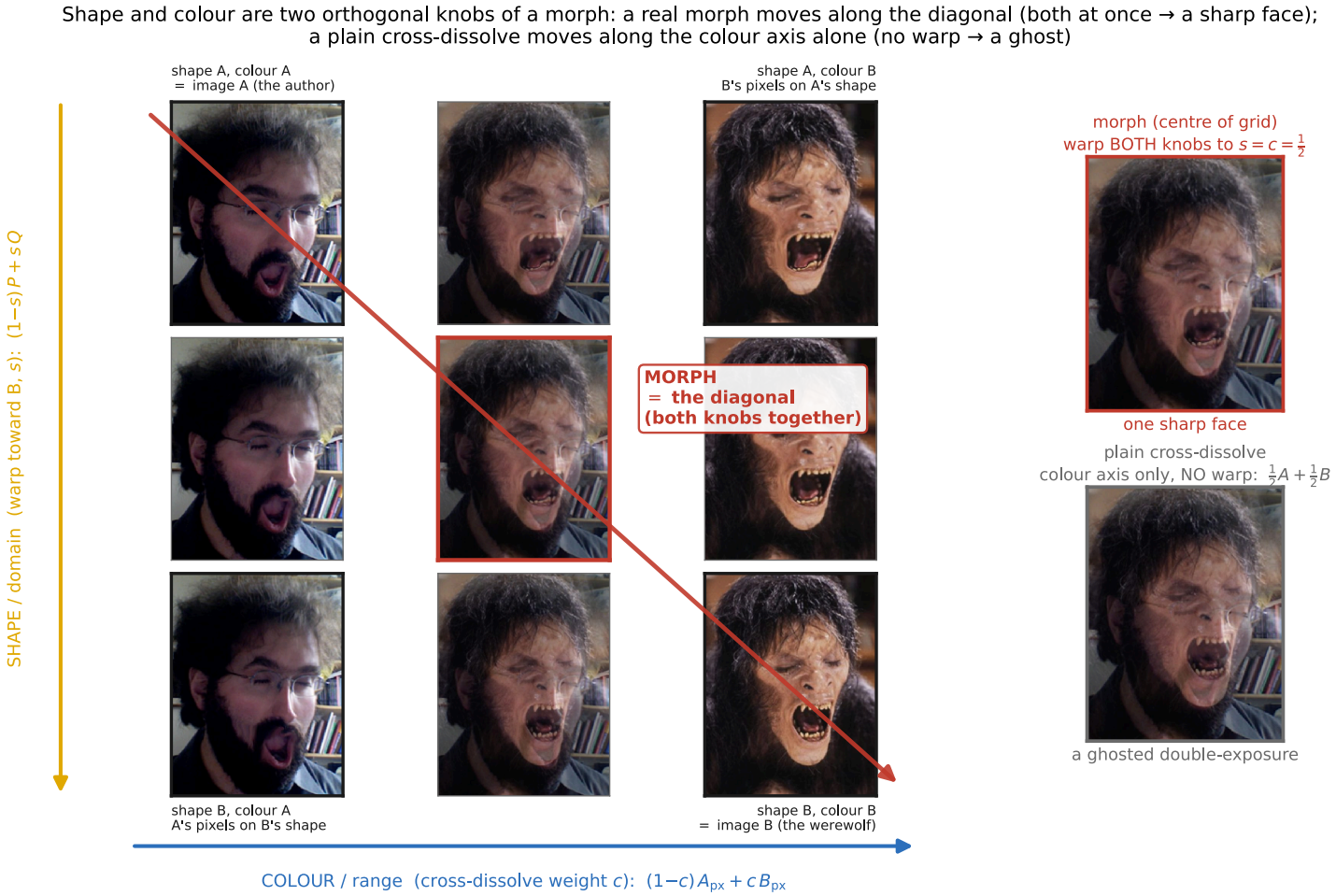


Figure 11.3.2 (reproduced). *The two knobs of a morph, as orthogonal axes. One axis, range (color): the cross-dissolve $C(x) = (1-t)C_0(x) + tC_1(x)$, a linear blend of pixel values. Other axis, domain (shape): a feature point interpolated as a vector, $V = (1-t)P + tQ$, sliding the feature's* location *from where I_0 puts it to where I_1 puts it. A plain cross-dissolve moves along the color axis only (ghosts); a true morph is the product: interpolate position* and *color together. The corners of the square label the four extreme combinations.*

The morphing recipe (combine both)

in the book ↗

The inputs are the two images I_0 and I_1, plus sparse correspondences the user marks on both: matching line segments, or matching mesh vertices (which primitive is the next two sections). The output is a sequence of in-betweens I_t, $t \in (0, 1)$. Each frame is built in four steps:

1. Interpolate the feature geometry to an intermediate shape: each corresponding pair (point, segment, or vertex) goes to its in-between location $P_t = (1-t)\,P_0 + t\,P_1$.
2. Build two dense warp fields from the sparse pairs: one carrying I_0's features onto the intermediate shape, the other carrying I_1's features onto the *same* intermediate shape.
3. Warp both images to that common shape. Now I_0 and I_1 are feature-aligned: both pairs of eyes land on the same pixels, both mouths coincide.

4. Cross-dissolve the two warped images, $I_t = (1 - t)\,\mathrm{warp}(I_0) + t\,\mathrm{warp}(I_1)$. Because they are now aligned, the blend is sharp, with no ghosting (Figure 11.3.3).

A natural question is why warp *both* images, not just one. You *could* warp I_0 all the way onto I_1's shape and dissolve, but then features would *slide* the whole distance while colors merely faded, and the early and late frames would be badly over-distorted (one image stretched to fit the other's geometry while still wearing its own colors). Meeting in the middle, with both images warped only partway, to the shared intermediate shape, keeps each warp small and symmetric, and is what makes the motion read as a single object transforming rather than one picture deforming into another.

The recipe interpolates both shape and color, and the endpoints prove it. At $t = 0$ the intermediate shape equals I_0's own shape, so the warp of I_0 is the identity, and the color weight on I_1 is zero: the output is exactly I_0. Symmetrically at $t = 1$ the output is exactly I_1. The morph passes cleanly through both originals.

One cross-reference closes the loop. The correspondences here are user-specified and sparse: a person draws a few lines or drags a few vertices. The automatic, dense way to obtain correspondence between two images is Chapter 12.8, Optical flow, and that is precisely what frame-interpolation and slow-motion methods use to morph between *video* frames without any hand-drawn lines (Chapter 30.4, Frame interpolation and slow-motion synthesis).

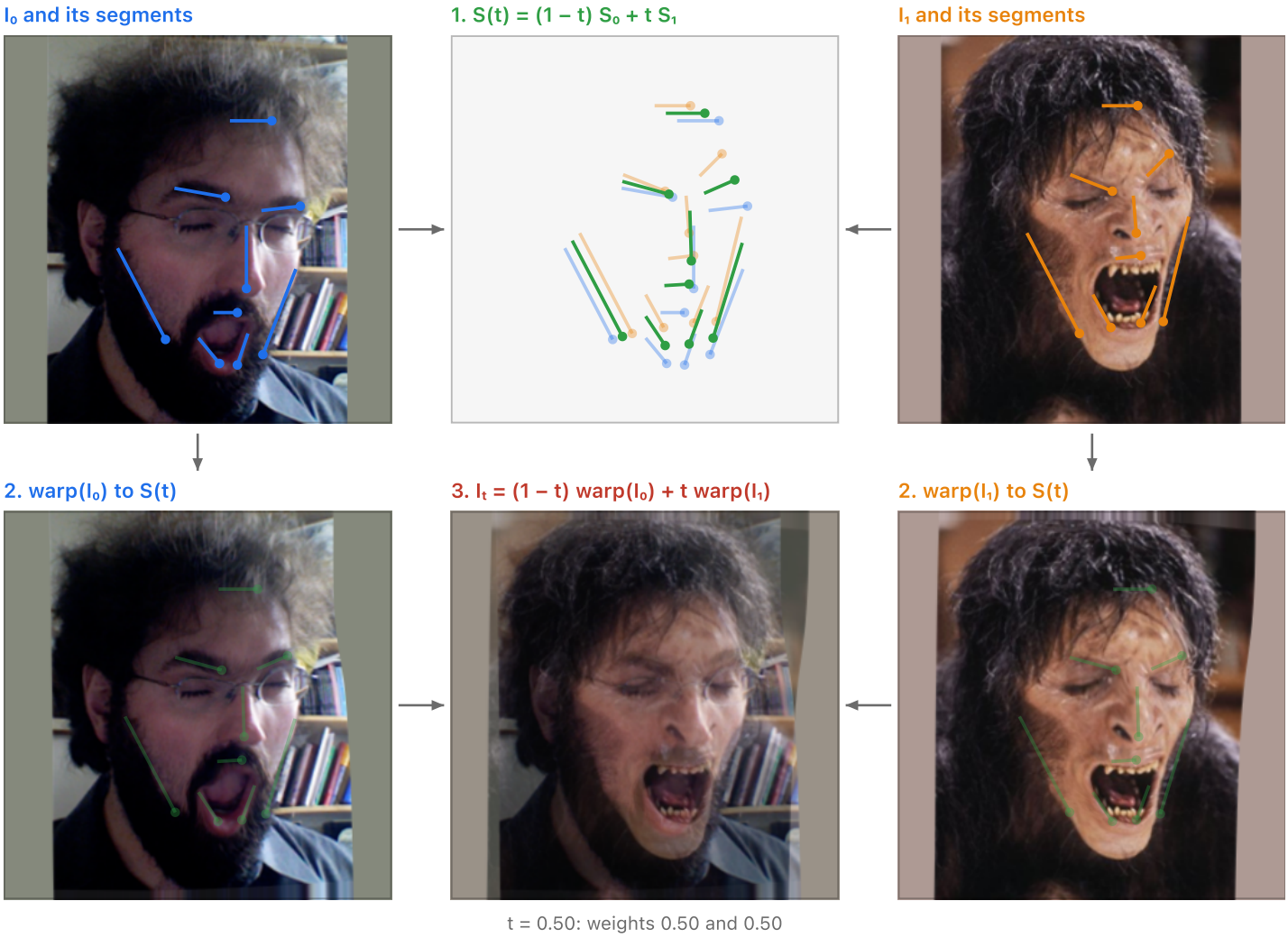


Figure 11.3.3 (reproduced). *The three-step morph pipeline on the same man-to-werewolf pair, for the t you choose (interactive;* 0.5 *at first). (1) Interpolate the features: each of the nine corresponding segments moves to* $P_t = (1-t)P_0 + tP_1$ *(top middle, green, between* I_0*'s in blue and* I_1*'s in orange), defining the intermediate shape. (2) Warp both:* I_0 *and* I_1 *are each warped to that* same *intermediate shape, so their features now coincide pixel-for-pixel. (3) Cross-dissolve the two warped, now-aligned images,* $I_t = (1-t)\mathrm{warp}(I_0) + t\,\mathrm{warp}(I_1)$*. The result is a sharp single face with no ghost, because alignment preceded the blend. Both images warp only partway (meeting in the middle) to keep distortions small and symmetric; move t and the share each image travels, and its weight in the blend, change together. The warps are the Beier and Neely field warp of Figure 11.3.4, with its segment pairs. "Scale v like u" switches to the variant in which each segment maps by a similarity (Figure 11.1.10), instead of keeping v in pixels as Beier and Neely do.*

Write the two backward maps from intermediate coordinates to the inputs as $b_{0,t}$ and $b_{1,t}$. Then

$$I_t(x) = (1-t)\,\widetilde{I}_0(b_{0,t}(x)) + t\,\widetilde{I}_1(b_{1,t}(x)). \tag{17}$$

The tildes indicate reconstruction and footprint filtering. For radiometric blending use a common linear-light representation; encoded-value blending gives a different transition. Use masks where one input is invalid or

occluded. With meshes, share connectivity across both inputs and the intermediate shape, and reject zero-area or flipped triangles. Matching endpoints does not guarantee valid geometry at every intermediate time.

Mesh / triangulation morphing (the alternative warp)

OPTIONAL
in the book ↗

The second way to get a dense warp swaps lines for a mesh of corresponding control points placed on both images (or a regular grid deformed to match features). Triangulate the points; each triangle in I_0 corresponds to the matching triangle in I_1.

The warp is then trivially local: within each triangle the map is a single affine (barycentric) transform, which is fast to evaluate and invertible when both triangles have nonzero area. The intermediate shape is the mesh with its vertices interpolated to $P_t = (1-t)P_0 + tP_1$; warp each source by its per-triangle affine onto that mesh, then cross-dissolve. Spline and free-form mesh variants (Wolberg's *Digital Image Warping*; multilevel free-form warps) trade the piecewise-affine creases at triangle edges for a smoother field.

Field versus mesh, the trade-off (Figure 11.3.6). A *mesh* warp has local support: moving a vertex disturbs only its incident triangles, which makes it fast, predictable, and free of the far-line fights that plague field warps, but the control net is fiddly to author, and triangle edges can crease if the triangulation is poor. A *field* (Beier–Neely) warp has global, line-based control that is sparse and intuitive to draw, but it is slower and can fold. Same job: turn sparse correspondence into a dense warp, checking separately for folds and degeneracy, expressed in two different bases (a triangulation versus a set of influence lines).

Triangulation image morph

▶ 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)

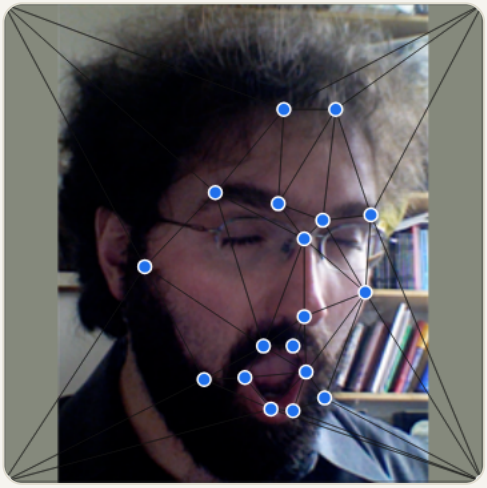


Image **A** — Frédo (drag the A endpoints)

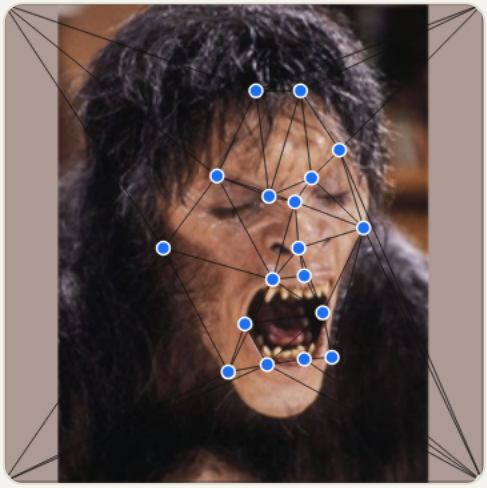


Image **B** — werewolf (drag the B endpoints)

A morph is a warp *and* a dissolve done together. The mesh is a single Delaunay triangulation of the shared vertex set (computed on the A positions, recomputed when those positions change, and held fixed while varying t); at each *t* every vertex is linearly interpolated between its A and B endpoints to give an in-between mesh *V(t)*. We piecewise-affine warp A (its A vertices → V(t)) and B (its B vertices → V(t)) so both land on the *same* intermediate shape, then blend them $(1-t)\cdot A_t + t\cdot B_t$. For a nonfolding mesh, the endpoints reproduce A and B (up to rasterization). The four image corners are pinned vertices, so the frame stays put. Interior handles can still fold or collapse triangles; such configurations need not give a valid image warp. **Click empty space** on a pane to add a correspondence (it starts at the same spot on A and B; re-triangulates), **drag** to

move that endpoint, **Delete** to remove the selected one.

Figure 11.3.5 (reproduced). *Interactive: the same Frédo↔werewolf morph, driven by a triangulation instead of feature lines. Place correspondence vertices (each a point on Frédo matched to a point on the werewolf); the demo builds one Delaunay mesh and, at time t, moves every vertex to its in-between position, warps both images onto that mesh by piecewise-affine, and cross-dissolves. Drag a vertex on either face to refine the match; slide t or play. Adding or editing points on the first face rebuilds the triangulation, and the in-between frames tile cleanly only while no triangle folds over, so a vertex dragged across its neighbors breaks the morph locally. Compare its local, predictable control to the field warp above, the same morph in two different bases. Frédo's portrait © Frédo Durand; werewolf endpoint after the classic course morph.*

Either way, the warp is a domain transform that must reconstruct and prefilter as it resamples (minified regions averaged down, magnified regions smoothly interpolated up), exactly the elliptical weighted average (EWA) Heckbert 1989 / mip machinery of Chapter 11.1, Warping. Because a morph stretches some regions hard while crushing others, it is a genuine stress test for good resampling: cut corners on the prefilter and the in-betweens alias and shimmer.

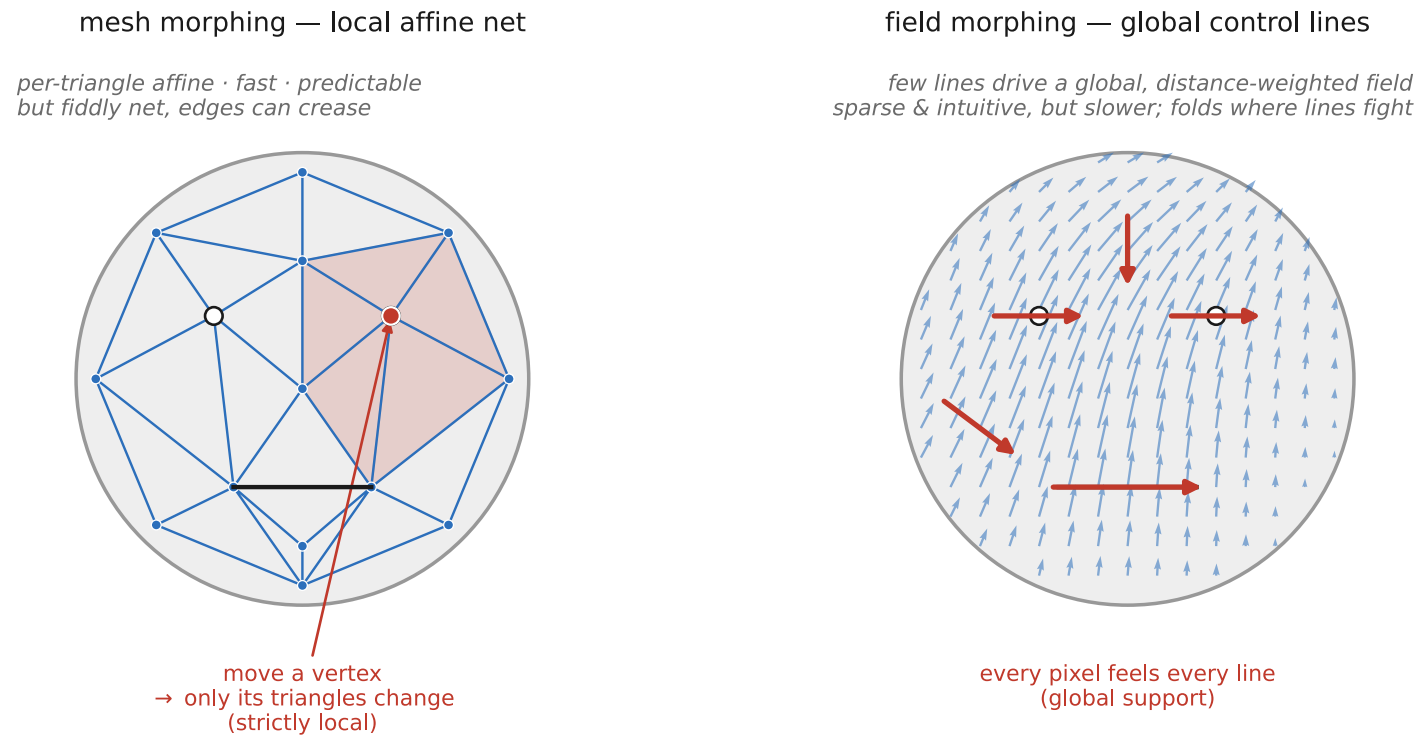


Figure 11.3.6 (reproduced). *Two ways to drive the same morph. Left, mesh morphing: a triangulated control net on each face; each triangle maps to its match by a per-triangle affine, with strictly local support (a moved vertex disturbs only its triangles). Fast and predictable, but the net is fiddly and triangle edges can crease. Right, field (Beier–Neely) morphing: a few control lines on each face drive a globally-supported, distance-weighted field. Sparse and intuitive to draw, but slower and prone to folds where lines fight. Same input faces, same goal, a dense warp, and two different control primitives. Neither is invertible automatically: the mesh warp is invertible only while no triangle flips, and the field warp only while its lines do not fight hard enough to fold it.*

View morphing: the geometrically-correct in-between of two views

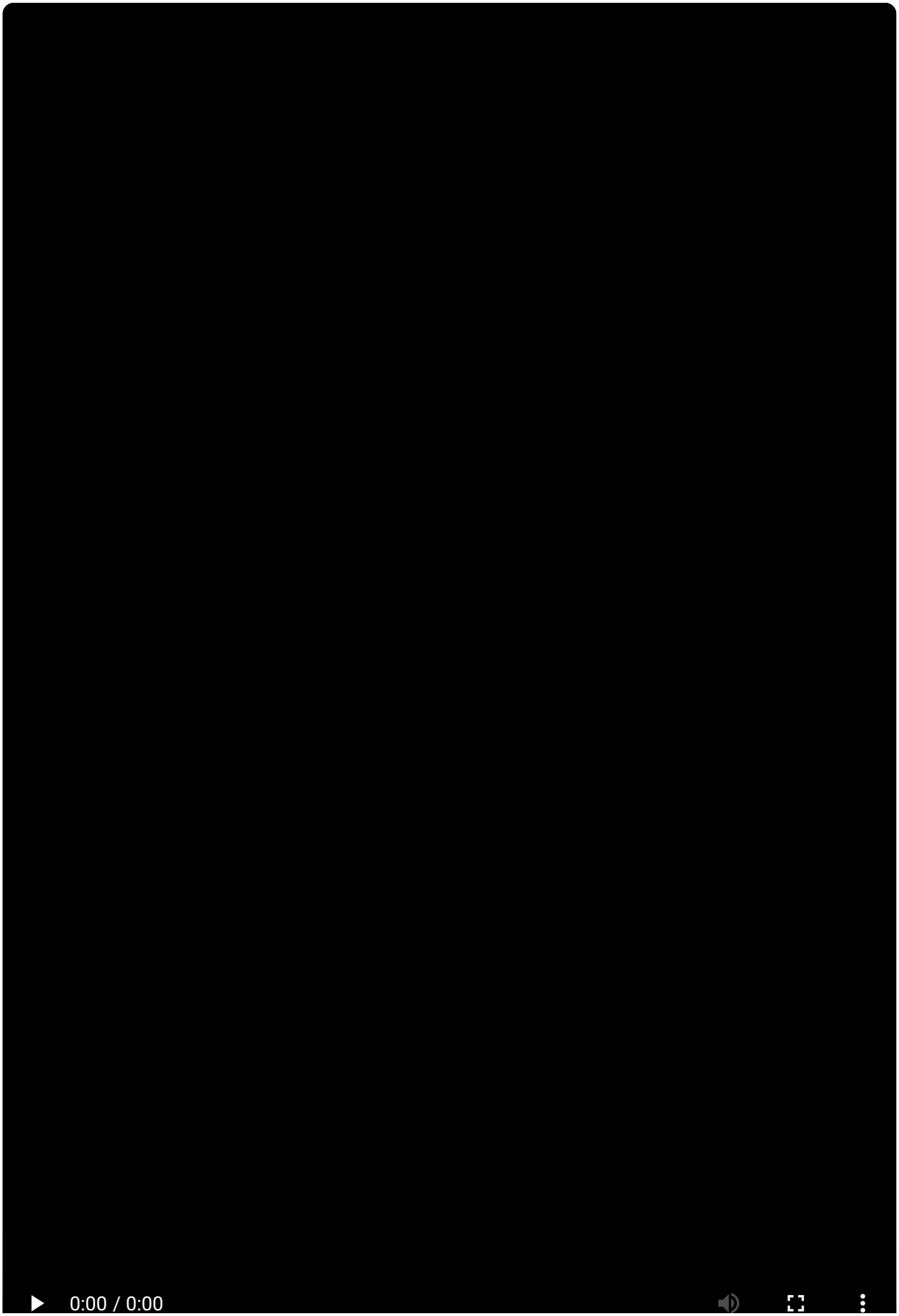
OPTIONAL
in the book ↗

There is a case where the flat 2-D morph above is silently *wrong*. Suppose I_0 and I_1 are not two arbitrary pictures but two photographs of the same 3-D scene from different viewpoints, such as a face rotating between two camera angles. A naive linear morph then bends straight lines and shrinks the object: the in-between is not a valid view of *any* rigid scene (Figure 11.3.7). The reason is fundamental. Linearly interpolating image positions is not the same as interpolating the underlying 3-D geometry. A straight edge in the world projects to a straight line in both photos, but the midpoint of its two image projections is generally *not* where that edge would project from an in-between camera; the linear average cuts a chord across the true projective path, so straight lines sag and the silhouette pinches inward.

The fix, due to Seitz and Dyer (1996) in *View Morphing*, restores physical validity with three steps:

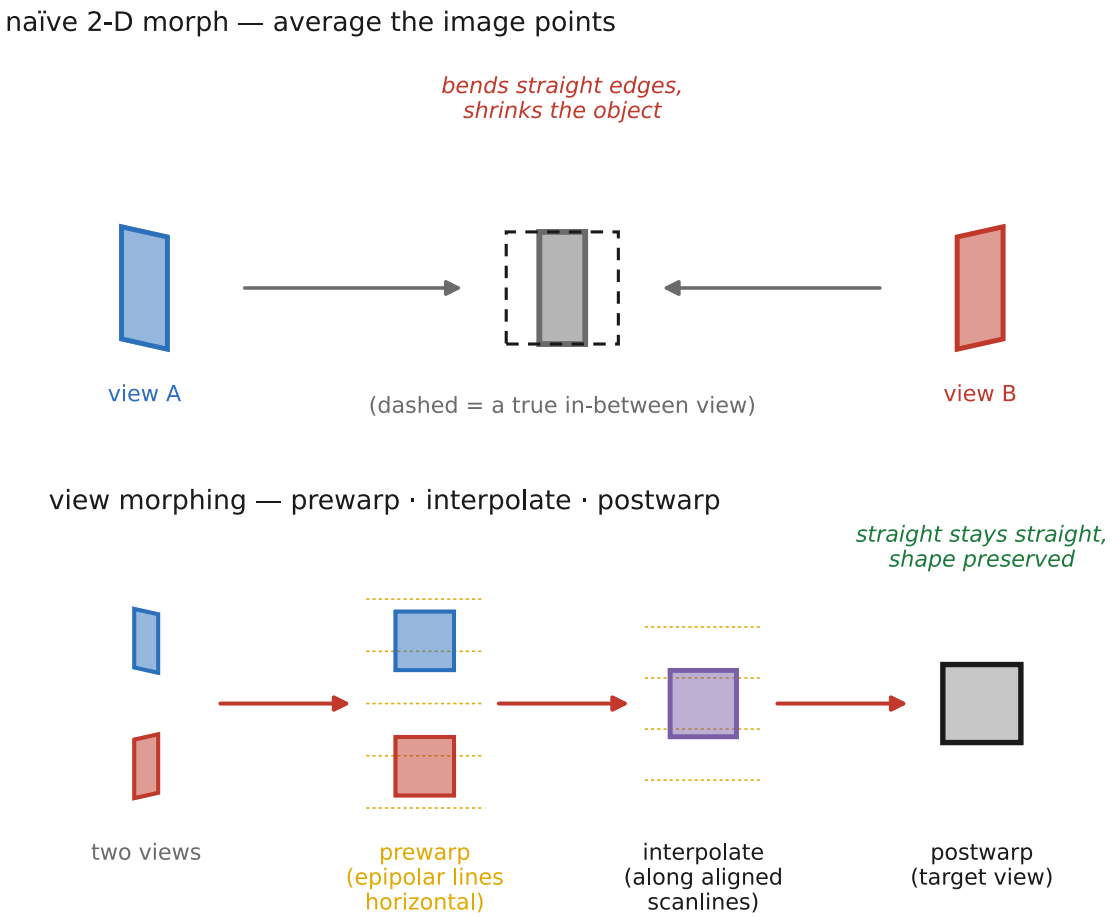
1. Prewarp both images by homographies that rectify them to a common plane, so the two image planes become parallel and corresponding epipolar lines are horizontal and aligned (the same rectification used in stereo).
2. Linearly interpolate the now-rectified images, a plain morph. But in this rectified frame the linear interpolation *is* geometrically correct, because corresponding points are constrained to move along aligned scanlines, so averaging their positions tracks the true motion.
3. Postwarp the interpolated result by an interpolated homography to the desired in-between view.

Under rectified equal-intrinsic cameras translated parallel to their image planes, a fixed visible 3-D point moves linearly in image coordinates along the baseline. This is the special geometry exploited by view morphing Seitz & Dyer 1996. It does not imply that every camera motion admits a plain linear image blend. Correspondences, visibility and the pre/postwarps still determine whether the interpolated view is valid.


0:00 / 0:00

Video 5. Video: *View Morphing*, interpolating between two photographs along a physically valid path, so the in-between frames look like real camera motion rather than a flat cross-dissolve. Seitz & Dyer 1996 (**Seitz & Dyer 1996**), reproduced with permission.

The connection outward is direct. View morphing needs corresponding points between the two views and the epipolar geometry relating them: exactly the correspondence problem of Chapter 12.8, Optical flow and stereo. Supply that correspondence *densely and automatically* and you can synthesize in-between views with no hand-drawn lines at all: the bridge from morphing to image-based rendering, and to flow-driven frame interpolation (Chapter 30.4, Frame interpolation and slow-motion synthesis), where the two "views" are simply adjacent video frames.



Averaging image points is not averaging 3-D geometry — rectify first, then a linear interpolation is geometrically correct.

Figure 11.3.7 (reproduced). *Why two* views *need view morphing. Top, naive 2-D morph: linearly interpolating image positions between two views of a rotating object bends straight edges and shrinks the object; the in-between is not a valid view of any rigid scene, because averaging image points is not averaging 3-D geometry. Bottom, view morphing (Seitz & Dyer): prewarp (rectify both images to a common plane so epipolar lines are horizontal and aligned) → linearly interpolate (now correct, since corresponding points move along aligned scanlines) → postwarp to the target in-between view. Straight lines stay straight and the object keeps its shape, a true perspective in-between.*

Recap: big lessons in this lecture

💡 BIG LESSON (L6.13) — NYQUIST–SHANNON: SAMPLE ABOVE TWICE THE HIGHEST FREQUENCY, OR IT ALIASES

Sample above twice the highest retained frequency. If the signal is too detailed for the grid, low-pass it before sampling. This applies to space and time. Real prefilters have finite transition bands, so allow a margin below Nyquist and check residual aliasing.

💡 BIG LESSON (L6.14) — SAMPLING REPLICATES THE SPECTRUM; ALIASING IS THE REPLICAS OVERLAPPING

Sampling on a grid is multiplication by a comb (a train of deltas), and a comb's Fourier transform is another comb, so sampling replicates the signal's spectrum at every multiple of the sampling rate. If the spectrum is band-limited below Nyquist the copies sit side by side and the original is recoverable; if it is not, the copies overlap, and the folded-over tails land on top of the genuine low frequencies as aliasing: energy appearing at completely *wrong* frequencies (moiré, the backward wagon-wheel). The cure is to band-limit before sampling: low-pass away everything above Nyquist first, then sample.

💡 BIG LESSON (L6.15) — THE IDEAL SAMPLING FILTER IS A SINC, WHICH DOES NOT EXIST IN PRACTICE

Ideal band limiting and ideal interpolation use sinc kernels, scaled for their respective grids. Their infinite support requires approximation in finite computations. A finite filter trades computation and frequency selectivity; inspect both aliasing and edge overshoot.

💡 BIG LESSON (L6.16) — A RESAMPLING / ANTI-ALIASING FILTER TRADES THE IDEAL SINC AGAINST COMPACT SUPPORT

Fourier analysis says the *perfect* reconstruction and anti-aliasing filter is the sinc, a perfect box in frequency. But the sinc is infinitely wide and rings when truncated, so no real filter can be it. Every practical kernel (bilinear, bicubic, Lanczos) is a deliberate compromise between approximating that idealized sinc (what Fourier wants) and staying compactly supported (what computation, and ringing, demand): reach wider and you approach sinc; truncate harder and it gets cheaper and less ringy but further from ideal.

💡 BIG LESSON (L6.17) — CHOOSING A RESAMPLING FILTER TRADES PASSBAND DROOP AGAINST STOPBAND LEAK

Read off the frequency response, the same compromise is a tug-of-war between two errors. Passband droop: a too-gentle filter attenuates frequencies *below* Nyquist that the grid *could* have represented, so the image goes soft. Stopband leak: a too-aggressive filter lets frequencies *above* Nyquist through, which alias into jaggies and moiré. A good kernel sits at the sweet spot (flat passband up to Nyquist, sharp rejection beyond) and box, tent, bicubic and Lanczos are just different points on this droop-versus-leak curve.

💡 BIG LESSON (L6.18) — BICUBIC (AND FRIENDS) ARE CONVOLUTION WITH A FIXED KERNEL, NOT POLYNOMIAL REGRESSION

The cubic-convolution filters used here evaluate a fixed piecewise-cubic kernel at fractional offsets and sum weighted samples. They do not solve a regression for every output pixel. Other methods also called bicubic interpolation use polynomial or spline constructions; the name alone does not identify the algorithm or its downsampling behavior.

BIG LESSON (L6.19) — RESAMPLE BY GATHERING, RARELY BY SCATTERING

Loop over output pixels and pull back through f^{-1} to find where each one reads from. Gathering fills every output exactly once; scattering pushes input pixels forward and leaves holes whose shape depends on the transform, a lattice of gaps under a scale and tilted slivers under a rotation. The same reflex governs warping, morphing, texture mapping, and flow-based reprojection later. Scattering earns its place only when the forward map is the thing you actually have, as in splatting a point cloud, and then it needs an explicit accumulation and normalization pass.

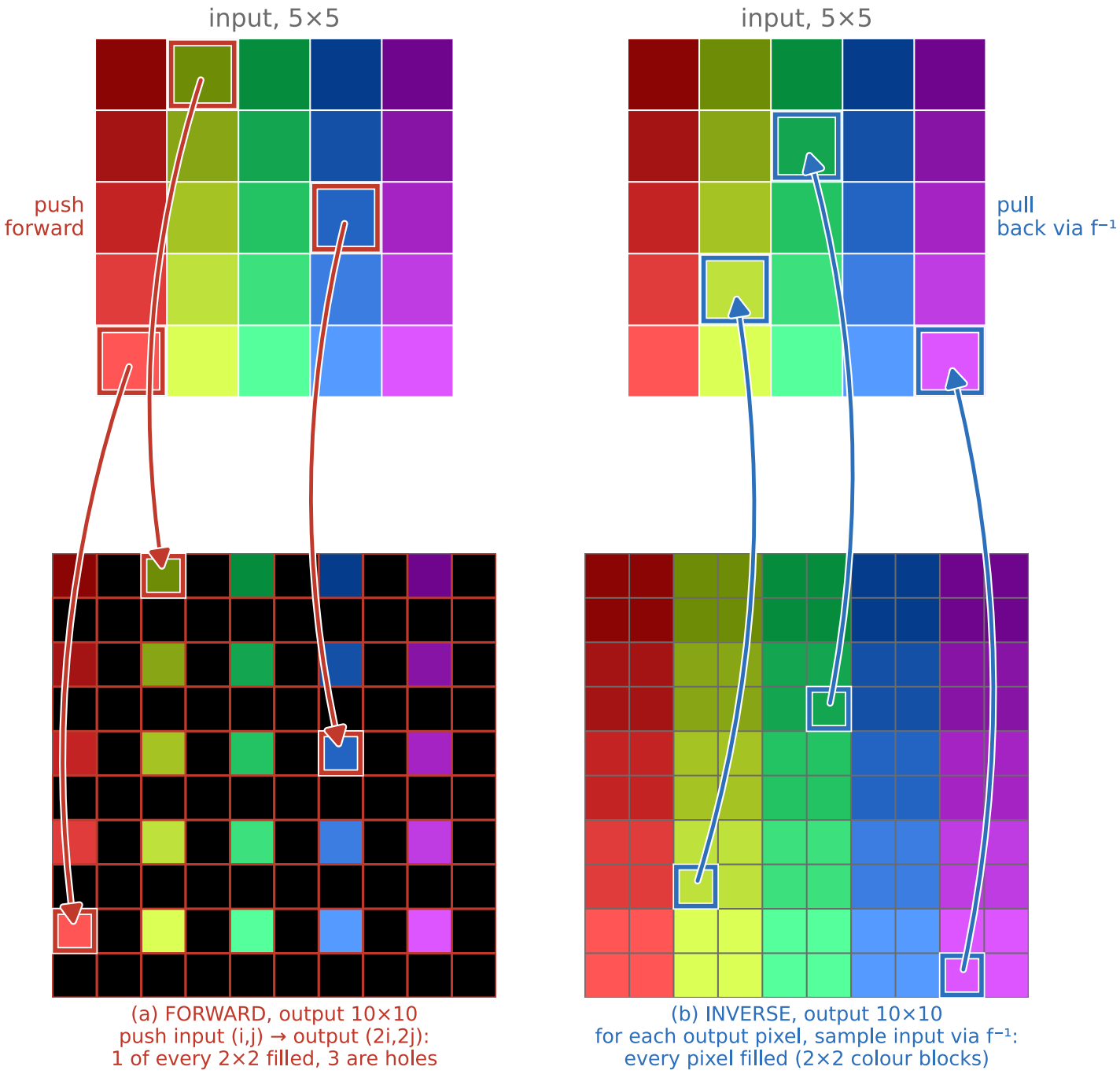


Figure 6.5.2 (reproduced). *Forward vs. inverse mapping on a 2× upscale of a small rainbow grid. Left (forward): loop over input pixels and push each to $(2i, 2j)$; only one cell of every 2×2 output block gets written, leaving the other three as black gaps (a regular lattice of holes; worse under a rotation, where the holes become tilted slivers). Right (inverse): loop over output pixels and, for each, apply f^{-1} to reach back into the input and sample, so every pixel is filled, no gaps. The arrows trace the two routes for a few pixels: forward arrows start at an input pixel and end where it is written, inverse arrows start at an output pixel and end at the input pixel it reads. The whole chapter loops over the output and reaches back into the input.*

BIG LESSON (L6.20) — EVERY RESAMPLING PASS MULTIPLIES THE SPECTRUM AGAIN, SO RESAMPLE ONCE

Repeated resampling accumulates blur, ringing, and sampling error. For repeated fixed convolution the frequency response is exactly H^N; for rotations it is only an illustration because the grid and effective filter change at each pass. Compose geometric transforms and resample the original once.

BIG LESSON (L17, RECURRENCE)

Almost every geometric algorithm in the book has the same two-part shape: first establish a coordinate map between two images (or between an image and a target frame), then transport the pixels along it. This part builds that machinery and its first uses: warping makes the map explicit and applies it, perspective rectification *solves* the map (a homography) from geometric constraints, and morphing *interpolates between* two maps. The same shape recurs across the book: panorama stitching *estimates* the map from matched features (in the multiple-exposure part), optical flow *estimates a dense* map from brightness constancy (in the matching part), and stabilization *smooths* a map over time while frame interpolation *renders along* one (in the video part). Separating the two halves, what the map is versus how you transport pixels through it, is what lets one transport engine (inverse warp + resample, built in the next two sections) serve this part and recur throughout the book; only the *map* changes from task to task. *(L17 is registered in the part introduction, Warping and morphing; this chapter is the transport engine, and the chapters that follow are different ways of finding the map.)*

BIG LESSON (L17, RECURRENCE)

A morph is a correspondence field plus transport, then a blend. To interpolate between two images you cannot average their colors where they sit; you must first establish a correspondence between their features, transport (warp) both images along that correspondence to a common intermediate shape, and *only then* blend. Separating the manipulation into a domain part (the warp: *where* features are) and a range part (the color blend: *what* they are), and interpolating each linearly, is what makes the in-between read as one object transforming rather than two pictures fighting. The same three-part shape (*correspond, transport, blend*) drives panorama de-ghosting, multi-frame super-resolution's sub-pixel alignment, and flow-driven frame interpolation; the only thing that changes is *how* the correspondence is obtained (here, sparse user lines; in Chapter 12.8, Optical flow, a dense automatic field).

BIG LESSON (RECURRENCE) — *ALIGN BEFORE YOU BLEND*

Averaging two images only makes sense once their features sit at the same place. A plain cross-dissolve blends colors at fixed coordinates, so any misalignment double-exposes into a ghost; the cure is always to warp into correspondence first, then average. You have already met this move under other names: panorama de-ghosting registers overlapping frames before feathering them, and multi-frame super-resolution aligns sub-pixel-shifted shots before merging (cross-ref Chapter 20.3, Blending). Morphing takes the lesson to its limit: it interpolates the *alignment itself* smoothly over t, rather than aligning once. The dense, automatic aligner that makes this work without hand-drawn lines is Chapter 12.8, Optical flow.