

ACRONYMS — THIS CHAPTER

ACRONYMS

AAM	Active Appearance Model
AV1	AOMedia Video 1 (video codec)
CC	CameraCalibration matrix (in the DNG pipeline); Creative Commons in a licensing context
DCT	discrete cosine transform
DOF	depth of field
FLAME	Faces Learned with an Articulated Model and Expressions
ILM	Industrial Light & Magic
MPEG	Moving Picture Experts Group
PCA	principal component analysis
PDI	Pacific Data Images
PSNR	peak signal-to-noise ratio
RAFT	Recurrent All-pairs Field Transforms (optical flow)
RANSAC	RANdom SAmple Consensus
RMS	root mean square
SAD	sum of absolute differences
SMPL	Skinned Multi-Person Linear model
SSD	sum of squared differences

LECTURE NOTES 9 — MORPHING

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

Color TLDR

OPTIONAL
in the book ↗

Gamma, cross-fading, and linear operations

The single most common way the encoding boundary bites is the most innocent-looking: **interpolating** between two colors. A cross-fade, a gradient, an alpha blend, an antialiased edge, all *average* two colors, and averaging is a linear-light operation. Do it on gamma-encoded values and you do not just get a slightly-off brightness, you get **false intermediate colors**: fade encoded red to encoded green and the midpoint goes through a muddy dark brown instead of a bright yellow, because the encoding pulls the sum down before the two channels can both be high at once. The fix is the usual one, decode, interpolate, re-encode, and the difference is not subtle once you look (John Novak's "What every coder should know about gamma" is the definitive visual demonstration).

interpolated in LINEAR light
the mixture of the two lights



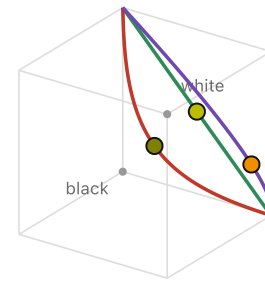
interpolated in GAMMA (code values)
what most software does



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



the three paths through the sRGB cube

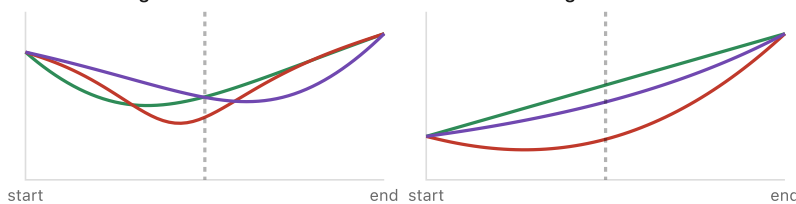


chroma along the fade

Oklab C

luminance along the fade

relative Y



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

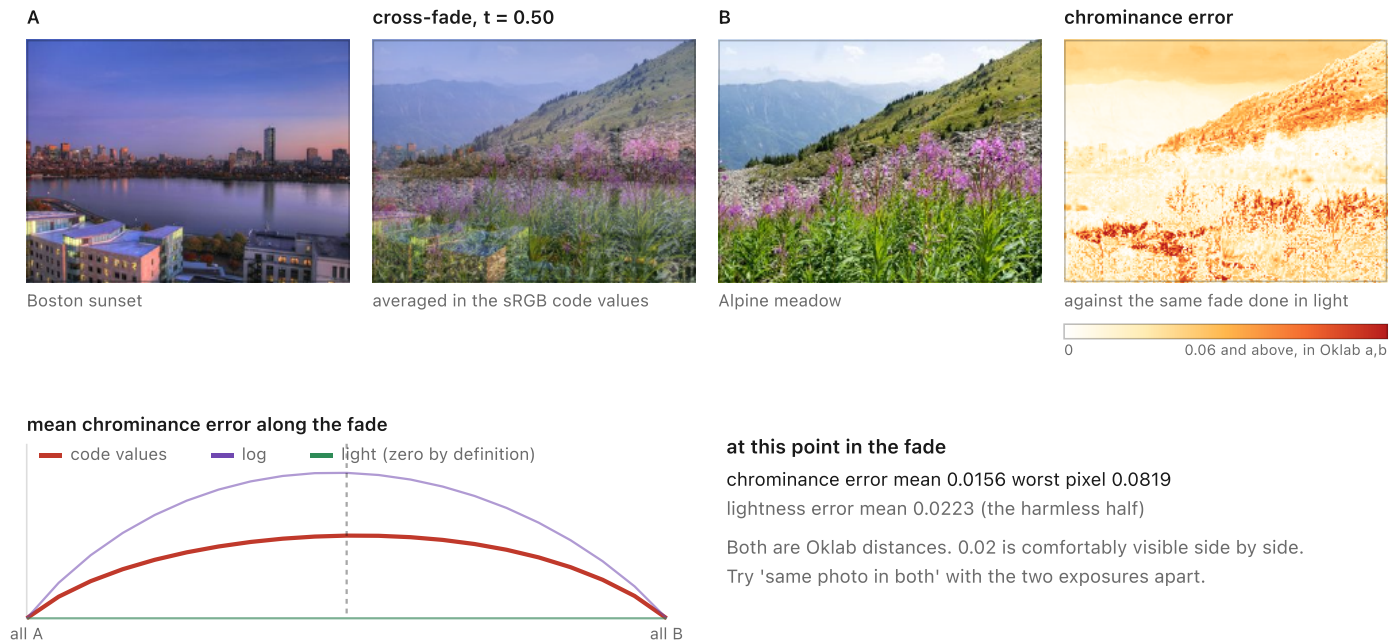
gamma bows toward black, log the other way



the two ends, and the three middles at this position

Why cross-fading in gamma makes colors that are in neither end (interactive). The same two colors are faded three times: by interpolating the sRGB code values directly, by decoding to light, interpolating and re-encoding, and by averaging in a **log** space, the geometric mean taken one cone at a time, which is the space Chong's illumination-invariant coordinates are built on. On red to green the gamma strip passes through a muddy brown neither endpoint contains, the linear strip passes through the yellow the two lights actually make, and the log strip swings the other way, through orange; the three middles are shown as patches between the two ends, where they can be compared as paint. **The dimming is the least of it.** A gamma fade does come out darker, and that half is nearly harmless: it looks like an exposure error and reads as one. The damage is in the chroma and the hue, which is why the chroma plot in Oklab is the one to read and the luminance plot sits beside it as a footnote. A wrong color does not look like an error, it looks like a color. **The trajectory panel is the reason.** All three paths join the same two corners of the sRGB cube and all three are different curves, because only one of them is straight in light: the code-value path bows toward black, since the encoding is concave and halfway in code values is well below halfway in light, while the log path bows the other way and can leave the cube entirely, because a geometric mean of two saturated lights is not required to be in gamut. Two behaviors are worth expecting rather than discovering: a log has no way to represent zero, so it is floored, and black to white in log therefore sits far down toward black rather than at a mid gray. None of this is a matter of taste: one of the three is the mixture of the two lights and the others are not, which is why it reaches far past gradients. Every blur, every resize, every anti-aliased edge and every alpha composite is this same average, done millions of times.

Flat patches make the argument; photographs make it land, because a photograph has thousands of chromaticities and the error goes where the color is. The next figure runs the same three averages on two pictures, and maps what the choice costs in chrominance alone ([the figure](#)).



Cross-fade two photographs, and watch where the encoding puts colors that are in neither one (interactive). Pick the two photos, drag the fade, and choose the space the average is taken in: the sRGB code values, which is what most software does unless it was written carefully; light; or a log space, the geometric mean one cone at a time. The fourth panel is the one to watch. It is not the difference between the two photographs: it is the **chrominance** part of the difference between the blend you chose and the blend in light, measured in Oklab, where a distance in *a* and *b* means something. Lightness is left out on purpose, since it is the half that looks like an exposure error. **The sharpest version of the experiment uses one photograph.** Set same photo in both and separate the two exposures with the exposure sliders. Every pixel of A and of B now has identical chromaticity, so nothing about the fade can legitimately change a hue: in light nothing does, and in the code values the saturated parts of the picture shift anyway, because a per-channel encoding does not commute with a mix and the three channels come back in the wrong ratio. On a sunset faded into a meadow the mean chrominance error in the code values runs around 0.016 in Oklab, with the worst pixels past 0.08, where 0.02 is comfortably visible side by side. The curve below the panels shows it peaking in the middle of the fade, which is where a dissolve, a resize, a blur or an anti-aliased edge spends its time.

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 $f : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ 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,

$$\text{out}(x, y) = \text{in}(f^{-1}(x, y)), \quad (1)$$

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 1

```
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.

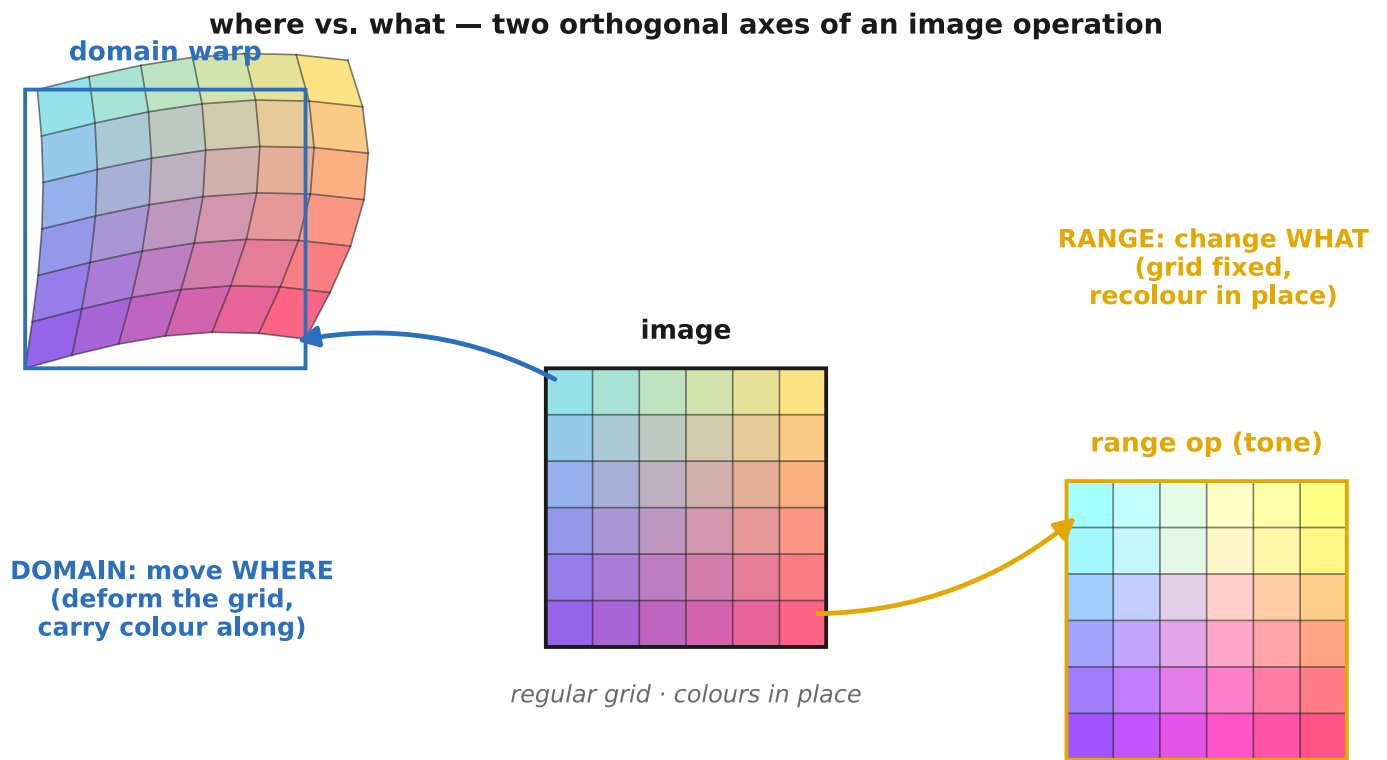


Figure 11.1.1 (reproduced). Domain vs. range. One image, two arrows. The domain arrow bends the underlying grid: a pixel slides to a new location and carries its color along (a warp). The range arrow leaves the grid fixed and recolors a pixel in place (a point/tone operation). Warping moves where, not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

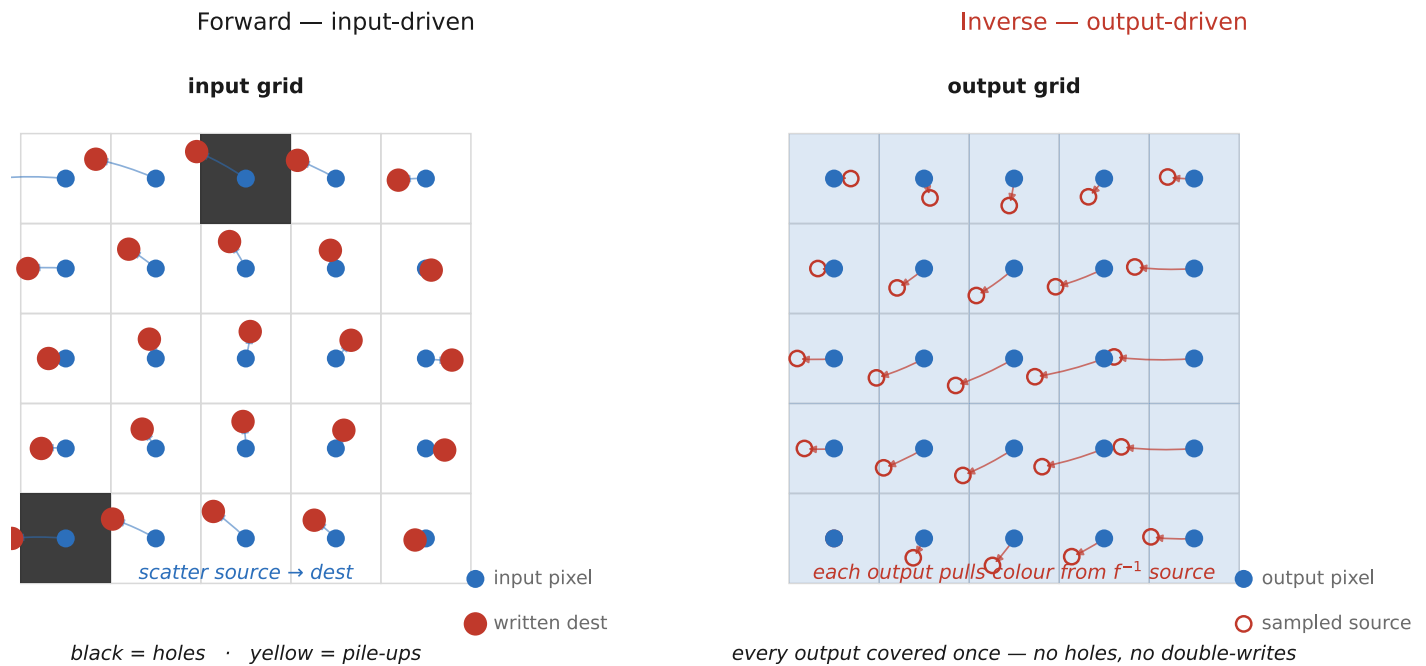


Figure 11.1.2 (reproduced). Inverse (output-driven) warping, the useful one. Loop over output pixels; push each back through f^{-1} into the input; sample the color there. Every output pixel is covered exactly once, with no holes or double-writes, and the only remaining difficulty, reading the input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

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),

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = A \begin{pmatrix} x \\ y \end{pmatrix} + \mathbf{t}, \quad (2)$$

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),

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = H \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \quad (x, y) \mapsto \left(\frac{x'}{w'}, \frac{y'}{w'} \right), \quad (3)$$

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.

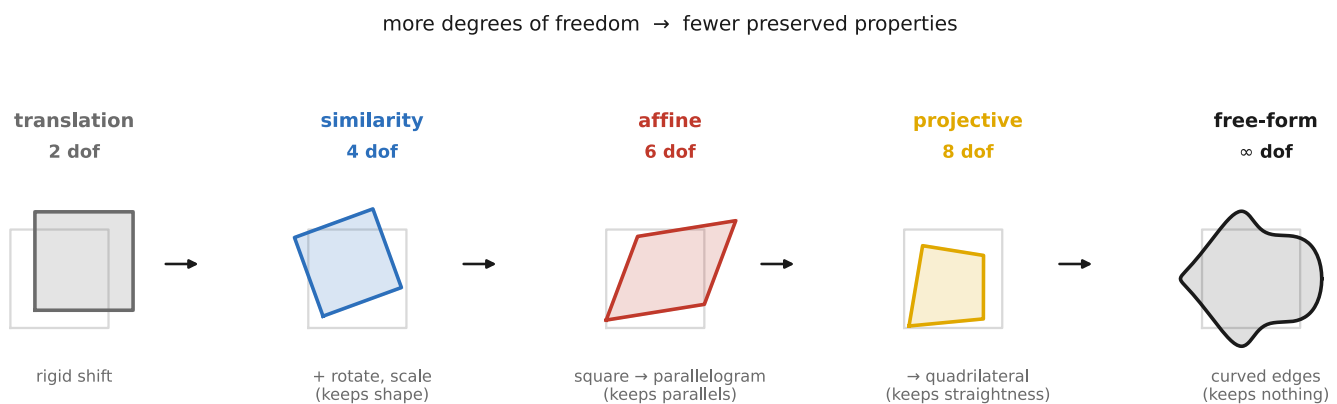
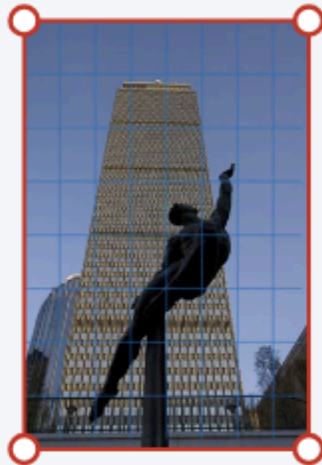


Figure 11.1.3 (reproduced). *The degrees-of-freedom ladder. A unit square under, left to right, translation (2) → similarity (4) → affine (6) → projective (8) → free-form (∞). Each rung adds freedom and breaks a preserved property: similarity keeps shape, affine keeps parallelism (square → parallelogram), 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

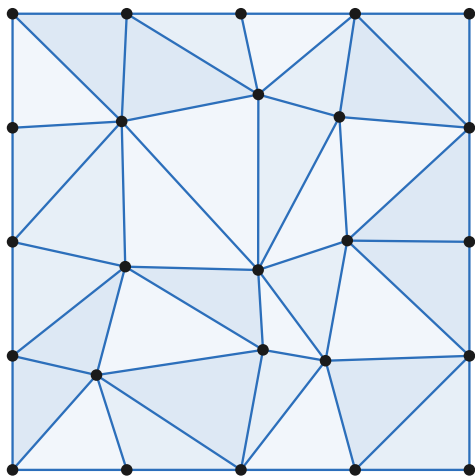
[in the book](#) ➤

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](#)).

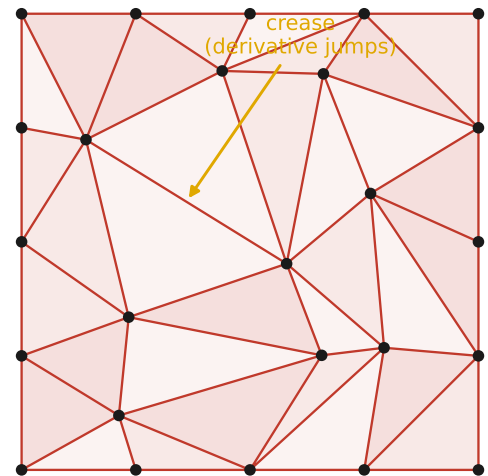
move mesh vertices → piecewise-affine free-form warp

source mesh + controls



triangulated correspondences

warped — each triangle its own affine



piecewise-affine · C^0 : creases at shared edges

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).

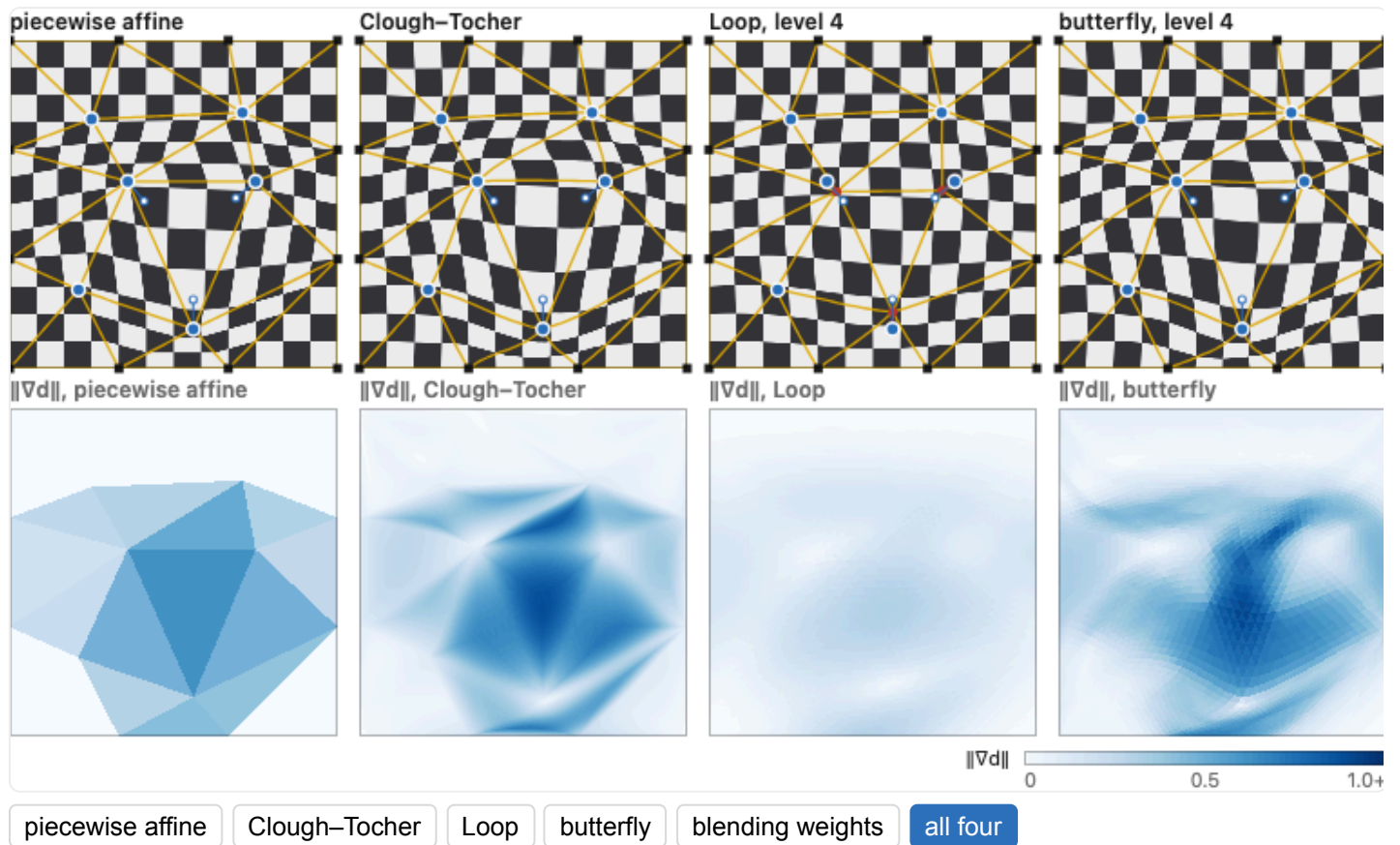


Figure 11.1.7, the compare view ([reproduced](#)). *Interactive: one set of control points, Delaunay-triangulated, and the same displacements d spread over the plane four ways: piecewise affine, Clough-Tocher, Loop subdivision and butterfly subdivision. Drag the blue points (the black squares on the 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.*

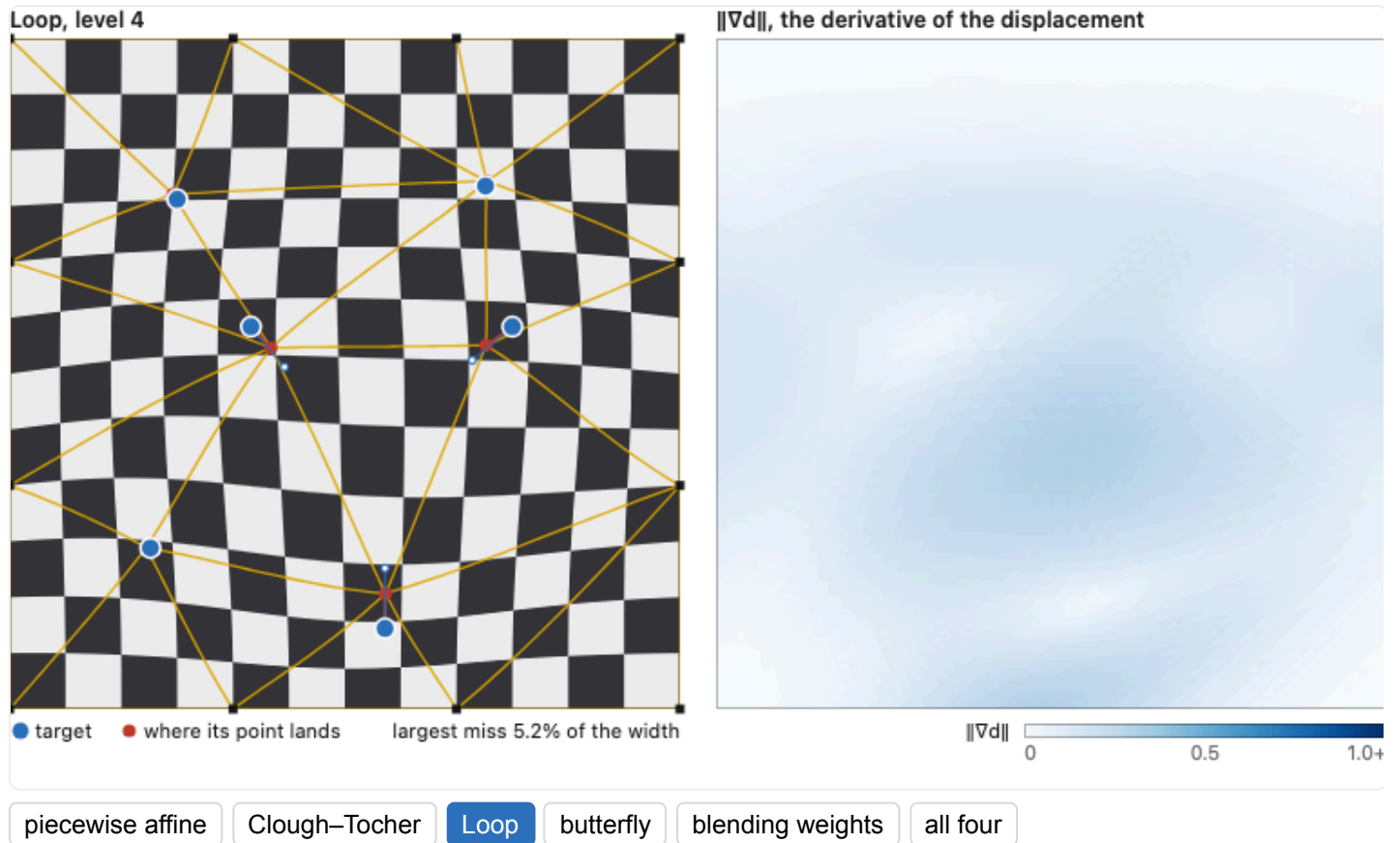


Figure 11.1.7, the loop view ([reproduced](#)). *Interactive: Loop subdivision of the warp's displacements, the rest and displaced positions of the vertices subdivided together. Loop approximates: each blue point is where a control was dropped, the red dot is where the picture under it lands, and the 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.*

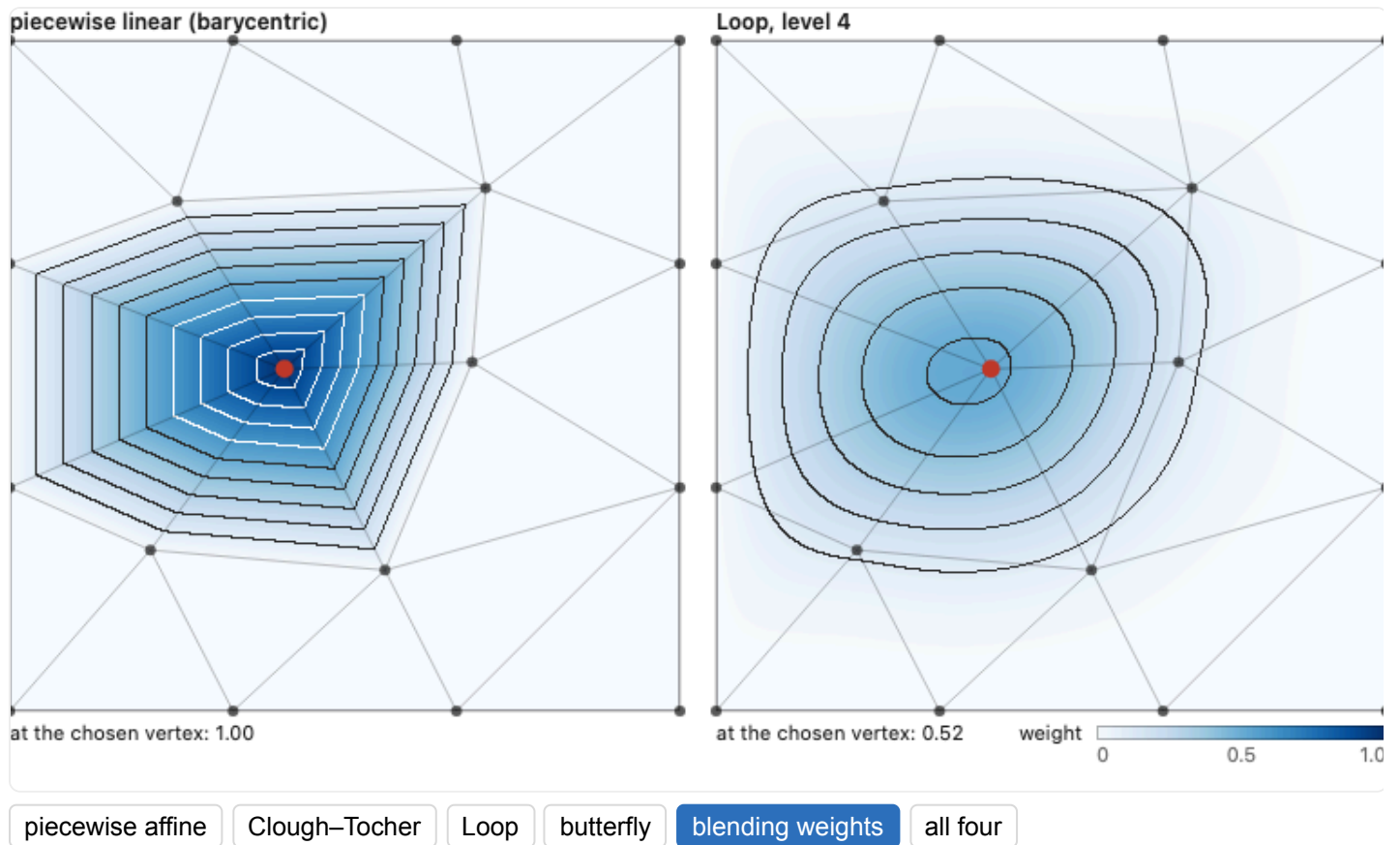


Figure 11.1.7, the weights view ([reproduced](#)). *Interactive: blending weights over the same triangulation, after Davis, Levoy and Durand [Davis et al. 2012](#). One vertex holds weight 1 and every other vertex 0. Left, the barycentric (piecewise-linear) hat, whose contour lines kink at every edge; 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.

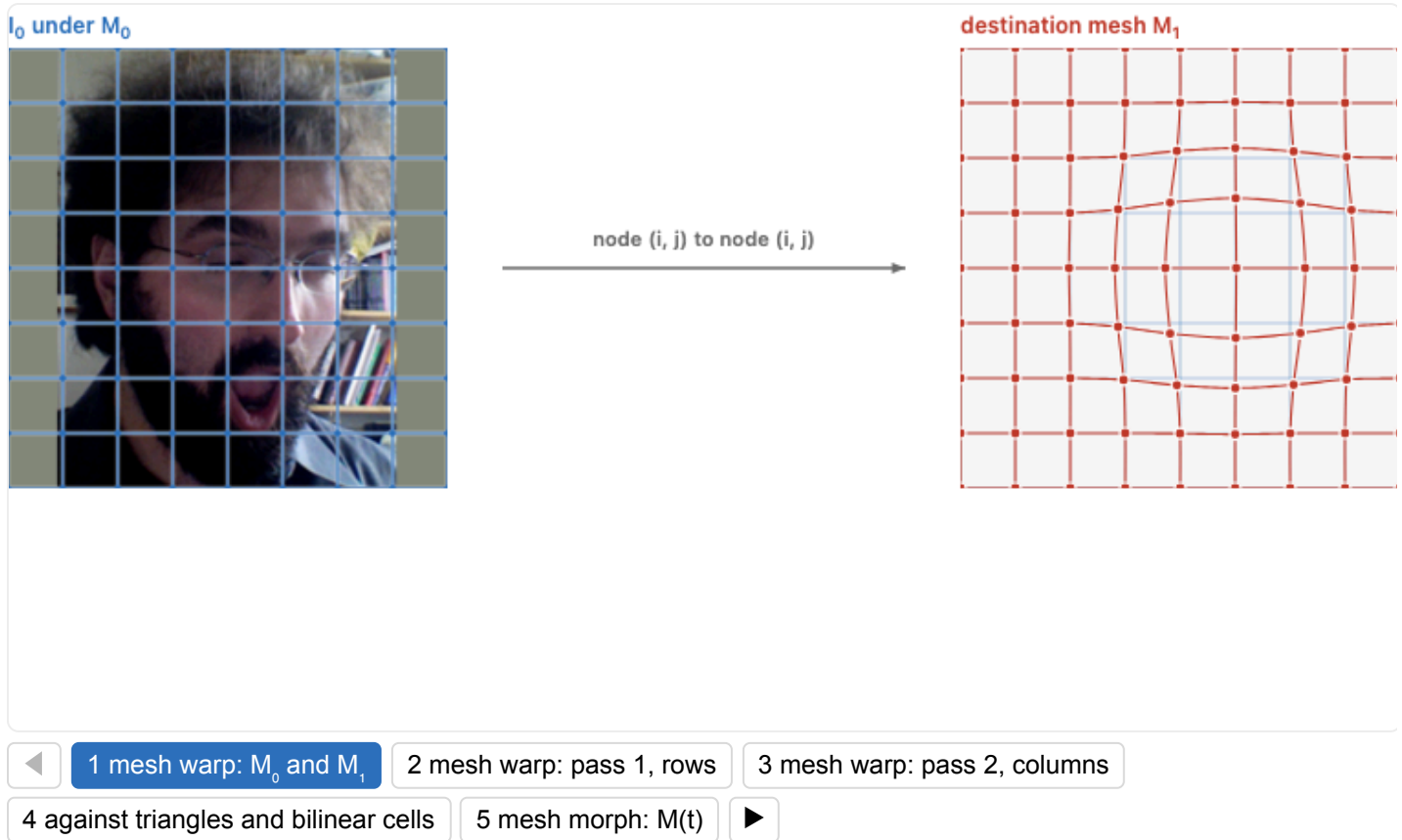


Figure 11.1.8, stage 1 ([reproduced](#)). The two meshes of Smythe's warp. Left, the source mesh M_0 , a regular grid of nodes over the photograph I_0 ; right, the destination mesh M_1 , with the same topology, so node (i, j) of one matches node (i, j) of the other (M_0 faint behind it). Drag any node 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.

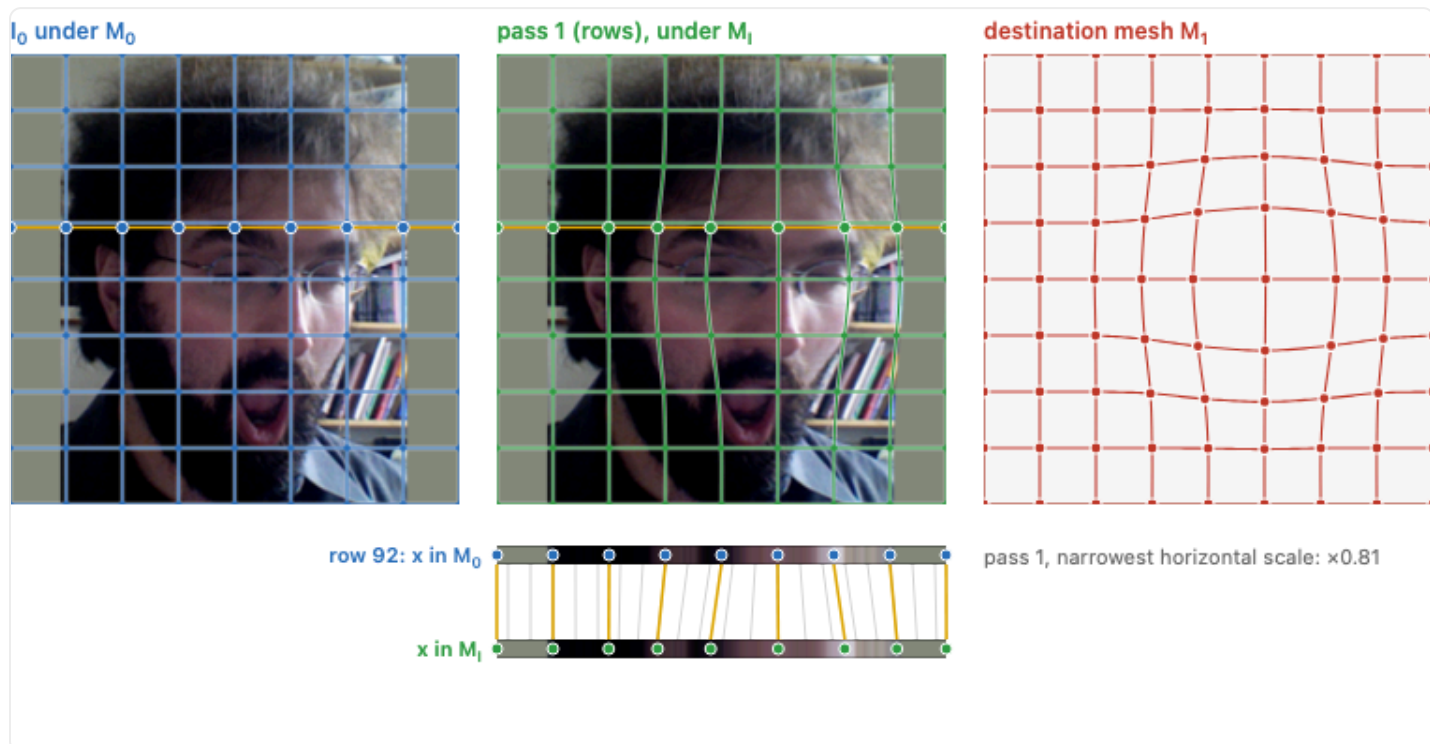
The warp itself is computed in two one-dimensional passes, following Catmull and Smith's separable scheme [Catmull & Smith 1980](#). An intermediate mesh M_I takes its x coordinates from the destination mesh M_1 and its y coordinates from the source mesh M_0 . The first pass moves pixels only horizontally, 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 2

```

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
                                                                # spline through that column
        xs0 = x where each column of M0 crosses row y
        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.

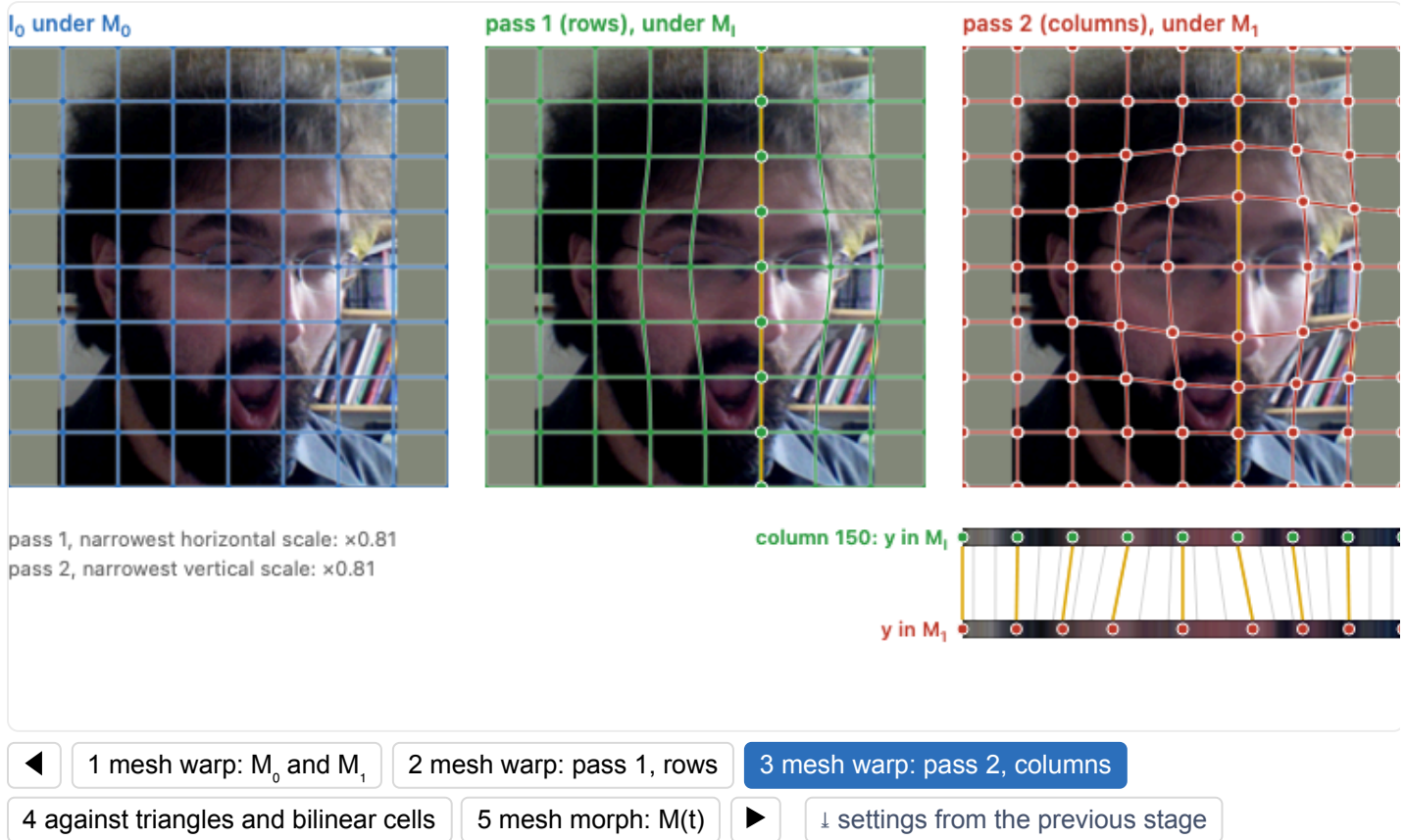
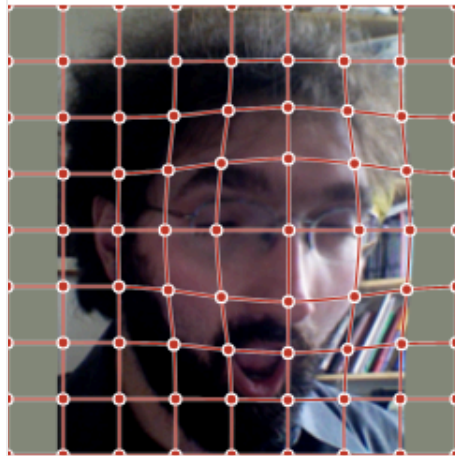


Figure 11.1.8, stage 3 ([reproduced](#)). Pass 2 does the same along columns, from M_I to M_1 . On the highlighted column (the column slider) the dots mark where each horizontal grid line crosses it, in M_I in the middle and in M_1 on the right, and the strip shows the column before and after the pass, 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.

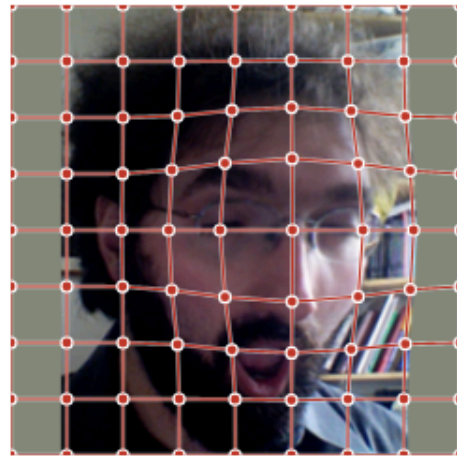
Each pass is a set of one-dimensional resamplings, which are cheap and easy to filter well, and this mattered on the computers of 1988. The separable scheme has a known weakness: where the warp rotates a region by a large angle, the first pass squeezes it into a sliver before the second pass expands 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).

two-pass, Catmull-Rom

RMS difference from bilinear cells: 4.0 levels
pass 1 narrowest scale: $\times 0.81$

triangles, 2 per cell

RMS difference from bilinear cells: 2.3 levels

bilinear per cell

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° .

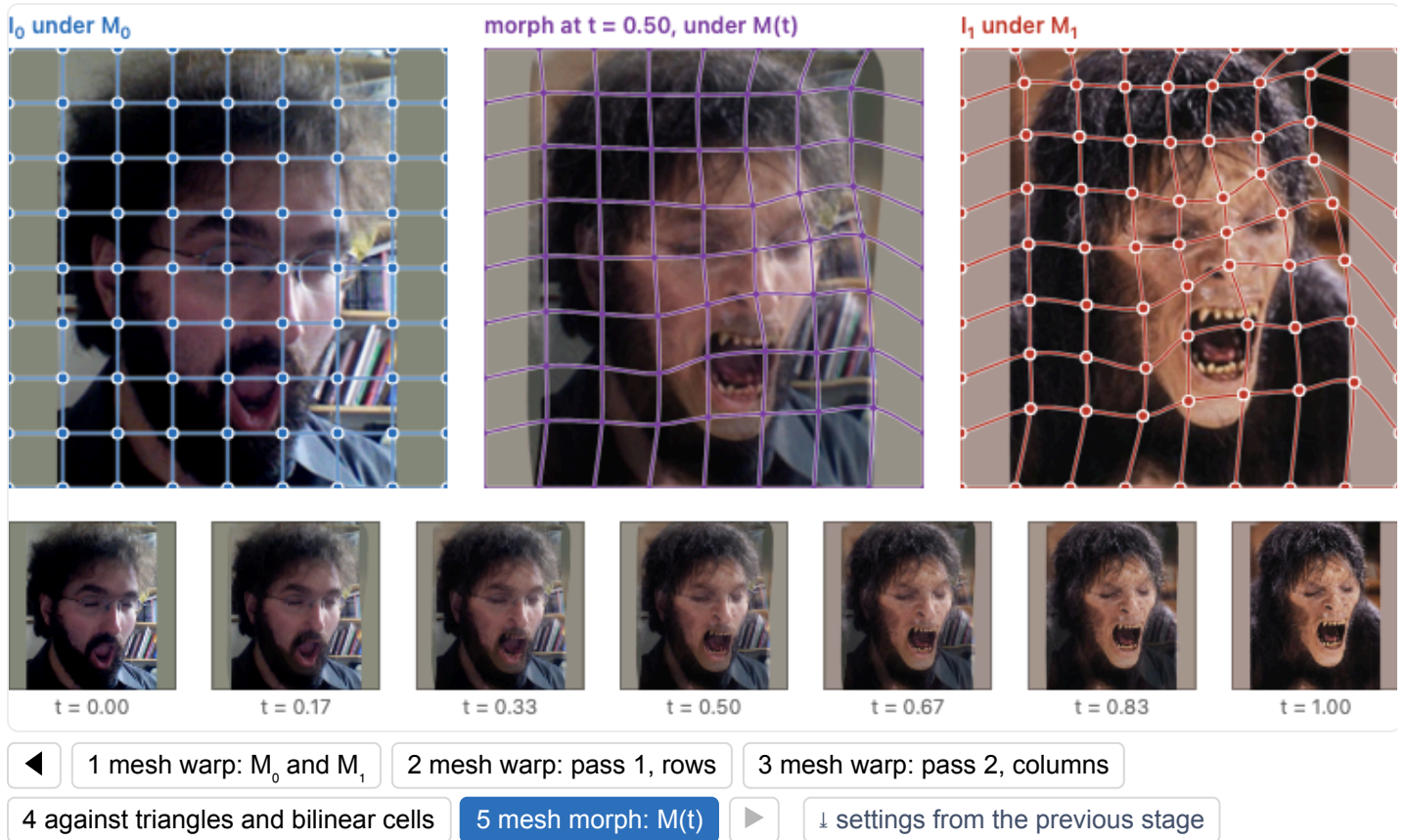


Figure 11.1.8, stage 5 ([reproduced](#)). *The mesh morph. The middle panel is the morph at t : the meshes are interpolated, $M(t) = (1 - t) M_0 + t M_1$, both photographs are warped to $M(t)$ by the two-pass warp, and the two results are cross-dissolved with the same t ; the row below shows seven values 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:

$$f(\mathbf{p}) = A\mathbf{p} + \mathbf{t} + \sum_i w_i \phi(\|\mathbf{p} - \mathbf{c}_i\|), \quad (4)$$

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.

Specifying the warp III: feature lines (Beier–Neely)

[in the book](#) ➤

Some features are more naturally specified by lines than by points, such as an eyebrow, a jawline, or the edge of a roof. Feature-based field warping [Beier & Neely](#) lets the user draw pairs of directed line segments, before and after. For the backward lookup below, unprimed endpoints specify the destination segment and primed endpoints the source segment. Each segment pair sets up a local coordinate frame: a coordinate u measured *along* the segment (normalized, 0 at $\mathbf{P}$ and 1 at $\mathbf{Q}$) and a coordinate v measured *perpendicular* to it (in absolute distance units). For a single segment we inverse-warp: read a destination point $\mathbf{X}$'s coordinates (u, v) in the destination frame and reconstruct the matching point in the source frame,

$$u = \frac{(\mathbf{X} - \mathbf{P}) \cdot (\mathbf{Q} - \mathbf{P})}{\|\mathbf{Q} - \mathbf{P}\|^2}, \quad v = \frac{(\mathbf{X} - \mathbf{P}) \cdot \perp(\mathbf{Q} - \mathbf{P})}{\|\mathbf{Q} - \mathbf{P}\|}, \quad (5)$$

$$\mathbf{X}' = \mathbf{P}' + u(\mathbf{Q}' - \mathbf{P}') + \frac{v \perp (\mathbf{Q}' - \mathbf{P}')}{\|\mathbf{Q}' - \mathbf{P}'\|}, \quad (6)$$

with $\perp (\cdot)$ the 90° rotation. A reported subtlety from the original work: u scales with the segment (so a stretched feature carries its interior along) but v is kept absolute. The authors tried scaling v too and it looked worse. With multiple segments, each segment i proposes its own displaced point $\mathbf{X}'_i$, and the final position is a distance-weighted average of those proposals, with weight

$$w_i = \left(\frac{\ell_i^p}{a + d_i} \right)^b, \quad (7)$$

where ℓ_i is the segment's length (longer segments carry more authority), d_i is the distance from the point to the segment (nearer segments dominate), and a, b, p tune the falloff. This is the technique behind the morph in Michael Jackson's *Black or White* video and a perennial course project; it is built here and used in [Chapter 11.3, Morphing](#) ([Figure 11.1.9](#)).

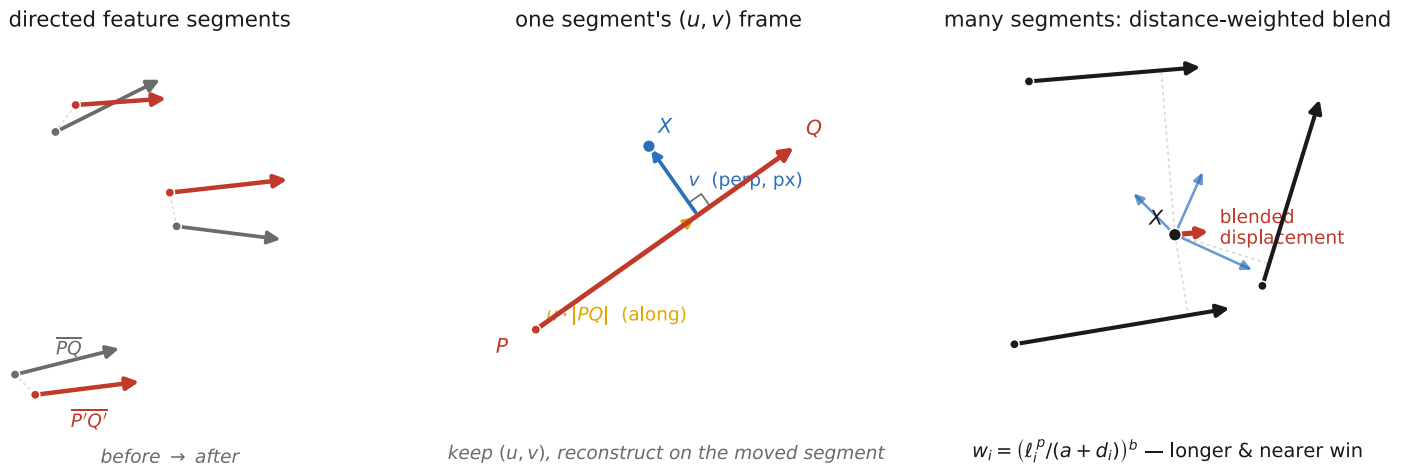
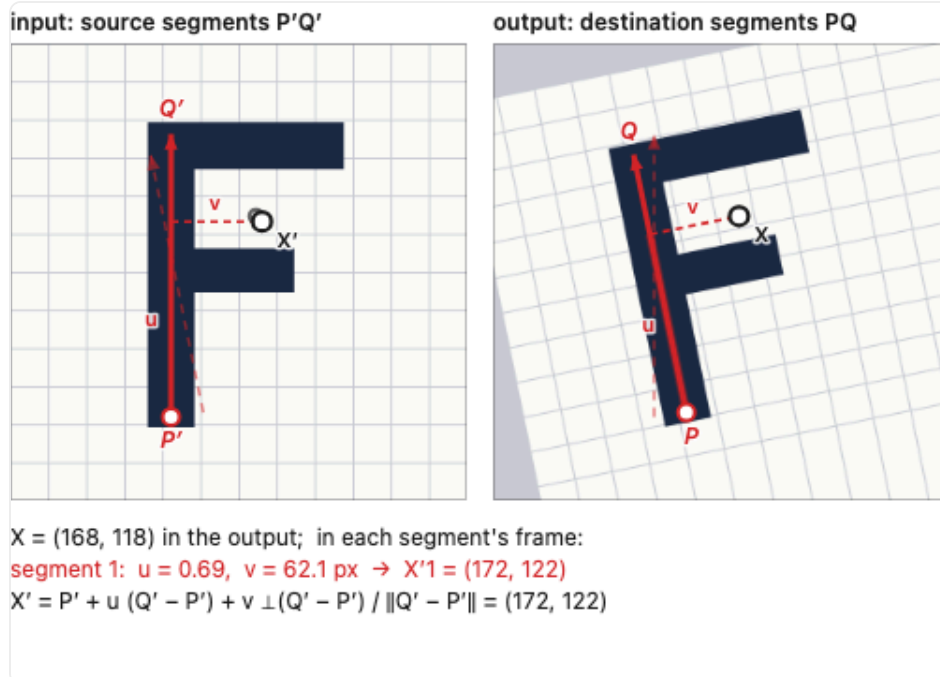


Figure 11.1.9 (reproduced). Beier–Neely feature-line warp. Left: before/after directed feature segments. Middle: one segment's local frame (u along, v perpendicular) used to map a point by reading (u, v) in the destination segment and reconstructing it in the source. Right: with several segments, each proposes a displacement and the result is their distance-weighted blend, weight $w_i = (\ell_i^p / (a + d_i))^b$; longer and nearer segments win.

[Figure 11.1.10](#) takes one output pixel through this computation, starting with a single segment.



◀

1 one segment

2 two segments

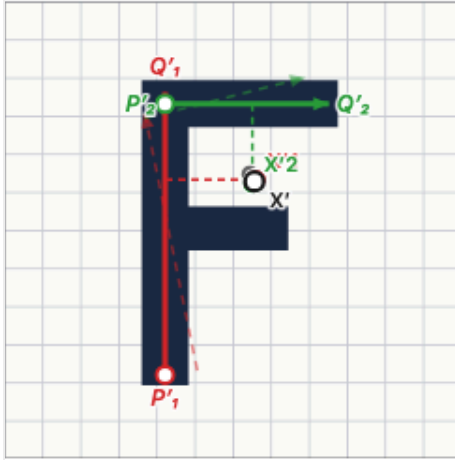
3 the weight field

▶

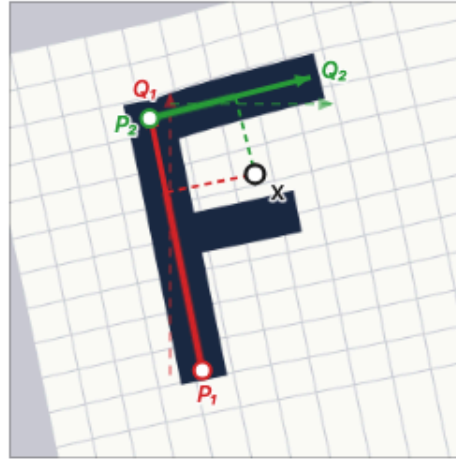
Figure 11.1.10, stage 1 ([reproduced](#)). The feature-line warp for one pixel, with a single segment. Left, the input with the source segment $P'Q'$; right, the output with the destination segment PQ ; each panel shows the other's segment as a dashed ghost. The output point X is read in the destination frame, u along the segment and v across it, and the same (u, v) rebuilt on the source segment gives X' , where the output pixel takes its color. The gray ring in the input marks where X itself sits. With one segment every pixel undergoes the same rotation, translation and stretch along the segment, so the whole picture turns. The input is the letter F that Beier and Neely's own figures warp; the menu also offers the author's portrait, a grid, other photographs, or your own. Click in the output to move X ; drag a segment's ends to move it. "Scale v like u " measures v in lengths of PQ , as u is, and lays it off in lengths of $P'Q'$, so the segment maps by a similarity: a segment twice as long in the output also doubles the distance across it. Beier and Neely keep v in pixels, which lets a segment stretch a feature along its length without widening it.

With two segments, each proposes its own source point and the warp averages them.

input: source segments $P'Q'$



output: destination segments PQ



$X = (168, 118)$ in the output; in each segment's frame:

segment 1: $u = 0.69, v = 62.1 \text{ px} \rightarrow X'_1 = (172, 122); d = 62.1 \text{ px}, \ell = 180 \text{ px}, w = 0.045, 55\% \text{ of the total}$

segment 2: $u = 0.53, v = 54.6 \text{ px} \rightarrow X'_2 = (170, 125); d = 54.6 \text{ px}, \ell = 114 \text{ px}, w = 0.037, 45\% \text{ of the total}$

$X' = \sum w_i X'_i / \sum w_i = (171, 123), w = (\ell^p / (a + d))^b$

◀ 1 one segment 2 two segments 3 the weight field ▶ ↓ settings from the previous stage

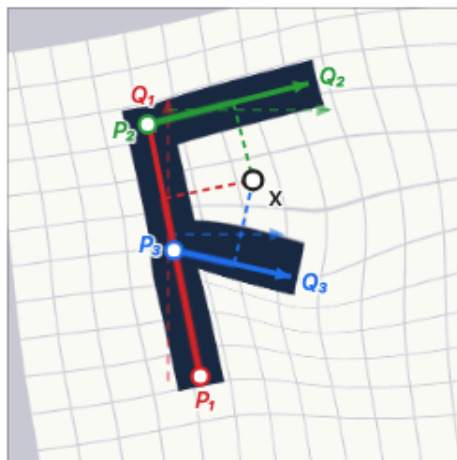
Figure 11.1.10, stage 2 ([reproduced](#)). Two segments. Each reads X in its own frame and proposes its own source point X'_i (the colored dots in the input, sized by weight); the output pixel samples the weighted average $X' = \sum_i w_i X'_i / \sum_i w_i$, with $w_i = (\ell_i^p / (a + d_i))^b$. Near a segment its own proposal dominates, so the feature it marks follows it closely, and between segments the picture bends from one transformation to the other. The readout gives each segment's u, v , distance d_i , length ℓ_i and share of the total weight. The segment buttons turn each segment on or off; an inactive segment stays in place, grayed, and drops out of the average. a softens the weight right next to a segment, b sets how fast it falls with distance, and p how much more a longer segment counts.

The weights form a field over the whole output.

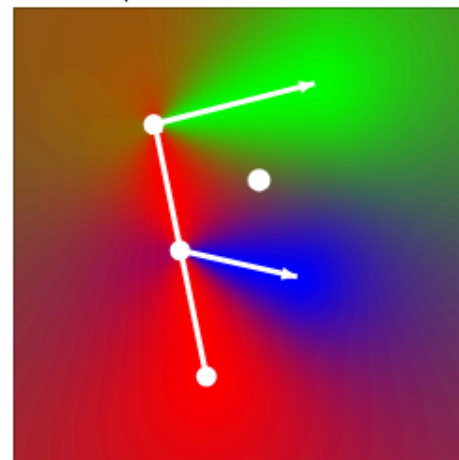
input: source segments $P'Q'$



output: destination segments PQ



weights $w_i / \sum w$, one channel each



$X = (168, 118)$ in the output; in each segment's frame:

segment 1: $u = 0.69$, $v = 62.1$ px $\rightarrow X'_1 = (172, 122)$; $d = 62.1$ px, $\ell = 180$ px, $w = 0.045$, 43% of the total

segment 2: $u = 0.53$, $v = 54.6$ px $\rightarrow X'_2 = (170, 125)$; $d = 54.6$ px, $\ell = 114$ px, $w = 0.037$, 35% of the total

segment 3: $u = 0.51$, $v = -58.7$ px $\rightarrow X'_3 = (151, 96)$; $d = 58.7$ px, $\ell = 82$ px, $w = 0.023$, 22% of the total

$X' = \sum w_i X'_i / \sum w_i = (167, 117)$, $w = (\ell^p / (a + d))^b$

◀ 1 one segment 2 two segments **3 the weight field** ▶ ⌵ settings from the previous stage

Figure 11.1.10, stage 3 ([reproduced](#)). The weight field. With three segments, the right panel colors every output pixel by its normalized weights $w_i / \sum_j w_j$, one color channel per segment (red, green, blue): a pure color where one segment rules, a mixture where the warp blends. The field has no hard boundaries, which is why the warp has no seams, and it covers the whole picture, so every segment pulls on every pixel a little. Click in the weight field to probe a pixel.

Beier–Neely field warp

+ add feature

– delete selected

≡ snap to identity

↺ reset

a  1.0 b  2.0 p  0.50

☐ original ☐ scale v like u

WARPED RESULT — DRAG THE HANDLES



Click near an endpoint to grab it; drag to move. Each  blue source segment is paired with the  orange destination segment it pulls toward (a thin dashed line links their midpoints). Segments are directed (arrowheads); the selected feature is highlighted. Try raising a brow or curving the mouth — the loaded state already has a gentle smile and lifted brows.

The math. Each feature is a source segment (P, Q) and a destination segment (P', Q') . This is an **inverse** warp: for each output pixel X and segment i , project X onto the *destination* segment to get u (fraction along, scales) and v (signed perpendicular distance, absolute), rebuild the matching point X'_i on the *source* segment, weight the displacement by $w_i = (\text{length}^p / (a + \text{dist}))^b$, and bilinearly sample the source picture at $X + \sum w_i D_i / \sum w_i$ (hole-free — every output pixel gets a colour). With every destination identical to its source, $D_i = 0$ and the output equals the input. With **scale v like u** checked, v is measured in destination segment lengths and laid off in source segment lengths, as u is, so a segment that doubles in length also doubles the distance across it.

Figure 11.1.11 [\(reproduced\)](#). *Interactive: a single-image Beier–Neely field warp. Draw pairs of directed feature segments, a blue source segment (where a feature is) and an orange destination (where it should move to), and the picture is inverse-warped so the source feature is pulled to the destination. Drag the orange ends to deform the face*

(raise a brow, curve the mouth); drag the blue ends to change what a segment controls. With every destination on top of its source the warp is the identity. This is the warp engine itself, the same primitive the morph calls twice per frame. "Scale v like u " switches to the variant in which each segment maps by a similarity, instead of keeping v in pixels as Beier and Neely do (Figure 11.1.10, stage 1).

Antialiasing for complex transforms

When the footprint stops being a square: the warp Jacobian

OPTIONAL
in the book ➔

Recall the engine. Inverse warping reads the source at $f^{-1}(x, y)$: for each output pixel, pull it back through the inverse map and look up the color there. But an output pixel is not a *point*. It is a little area, and what we owe it is the average of the source over the whole region that maps into it. Under a *uniform* scale that region is a square, the isotropic L16 kernel covers it exactly, and we are done. Under a general warp it is not a square, and the whole chapter follows from what shape it actually is.

Let $J = \partial f^{-1} / \partial(x, y)$, with output coordinates in pixels and mapped source coordinates in source pixels. A circular reference kernel transforms to an ellipse; a square pixel maps locally to a parallelogram. Thus the ellipse is a filter model, not the exact shape of every pixel. For a unit circular reference, the singular values of J give its axis scales and the left singular vectors give source-space directions. More generally its covariance is $J \Sigma_{\text{out}} J^T$. EWA combines this with source reconstruction support; a common Gaussian model adds Σ_{recon} . The ratio of singular values describes local anisotropy, not the absolute minification scale.

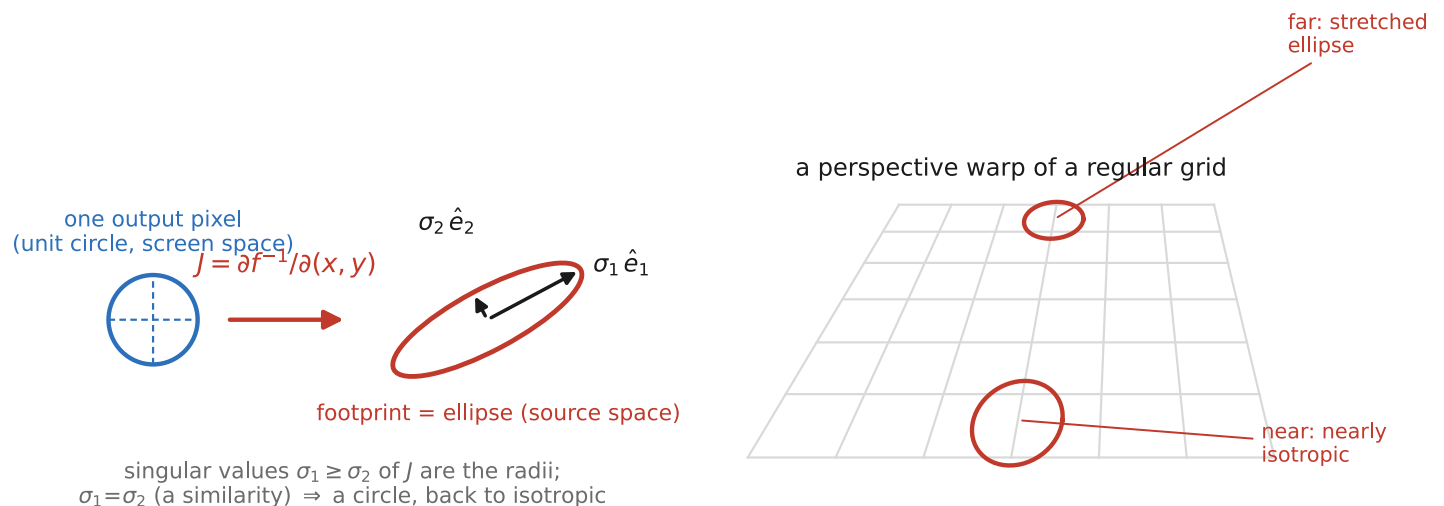


Figure 11.2.1 (reproduced). One output pixel, back-projected through a perspective warp. Left: the unit screen circle (one output pixel) is pushed through the local inverse-warp Jacobian $J = \partial f^{-1} / \partial(x, y)$ and lands in source space as an ellipse, the footprint. Its major and minor radii are the singular values $\sigma_1 \geq \sigma_2$ of J , along the singular vectors $\hat{e}_1, \hat{e}_2$; a square (round) footprint survives only under a similarity ($\sigma_1 = \sigma_2$). Right: a regular grid under a real perspective warp, where a small source cell back-projects to a nearly isotropic circle near the camera but to a long stretched ellipse toward the horizon. The footprint shape is the local Jacobian.

The canonical offender is exactly the floor we opened with: a checkerboard plane rendered in perspective, or rectified *to* perspective. Near the camera one pixel covers one tile and $\rho \approx 1$; toward the horizon one pixel covers many tiles in depth but only one across, and ρ climbs into the tens or hundreds. That is where every isotropic filter visibly fails, and the failure is not a matter of taste but of geometry. A round kernel has one radius, and the footprint has two very different ones. The fix is to make the kernel an ellipse that matches the footprint: the L16 prefilter, now carrying an orientation. [Figure 11.2.2](#) shows the three outcomes side by side on a genuine perspective checkerboard.

a receding checkerboard floor, filtered three ways

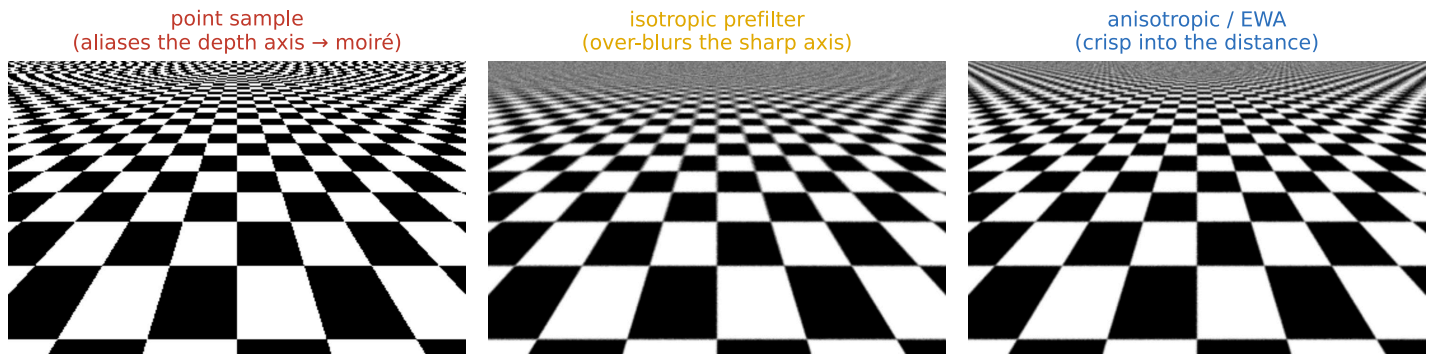


Figure 11.2.2 (reproduced). A receding checkerboard floor, filtered three ways (a real perspective render, each pixel filtered in numpy). Point sample (one tap per pixel): sharp up close but the compressed depth axis aliases into crawling moiré toward the horizon. Isotropic prefilter (a round average sized to the major axis): the moiré is gone, but the sharp across-axis is over-blurred and the far field washes to flat gray far too early. Anisotropic / EWA (an elliptical average matched to the footprint): tiles stay crisp into the distance with no moiré, the kernel averaging hard along depth and gently across it. One radius cannot win; an oriented ellipse can.

For example, suppose the inverse warp stretches one output-pixel distance into four source-pixel distances horizontally and one vertically. A circular source filter wide enough for the horizontal compression also averages too much vertically. A filter narrow enough to retain the vertical detail leaves horizontal frequencies that can alias. An elongated footprint expresses the two requirements separately. Rotate the warp and the long axis rotates as well; a scalar level-of-detail value cannot describe both its orientation and its unequal axis lengths. The ellipse is a convenient approximation based on a circular filter footprint and a local linearization, rather than the exact image of a square pixel under an arbitrary nonlinear warp.

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,

$$I_t[y, x] = (1 - t) I_0[y, x] + t I_1[y, x]. \quad (8)$$

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.

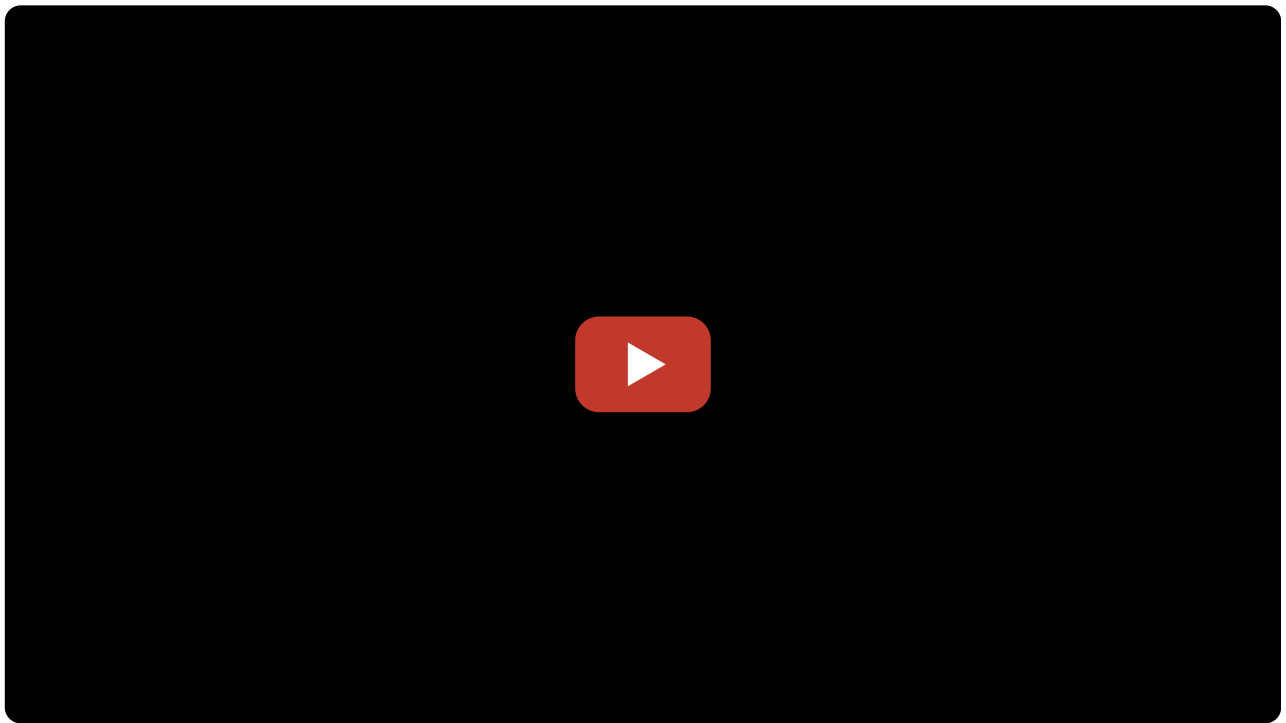
A morph uses correspondences to create an intermediate shape, warps both inputs into it, and blends their appearance. The correspondence model is what prevents doubled features; a color blend alone cannot do that.

💡 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 1. 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 2. 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,

$$C(x) = (1 - t) C_0(x) + t C_1(x), \quad (9)$$

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

$$V = (1 - t) P + t Q. \quad (10)$$

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, $C'(x, y) = C(f(x, y))$. In practice we use the inverse / backward map (for each *output* pixel, look up where it came from, $\text{out}(x, y) = \text{im}(f^{-1}(x, y))$), 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](#)).

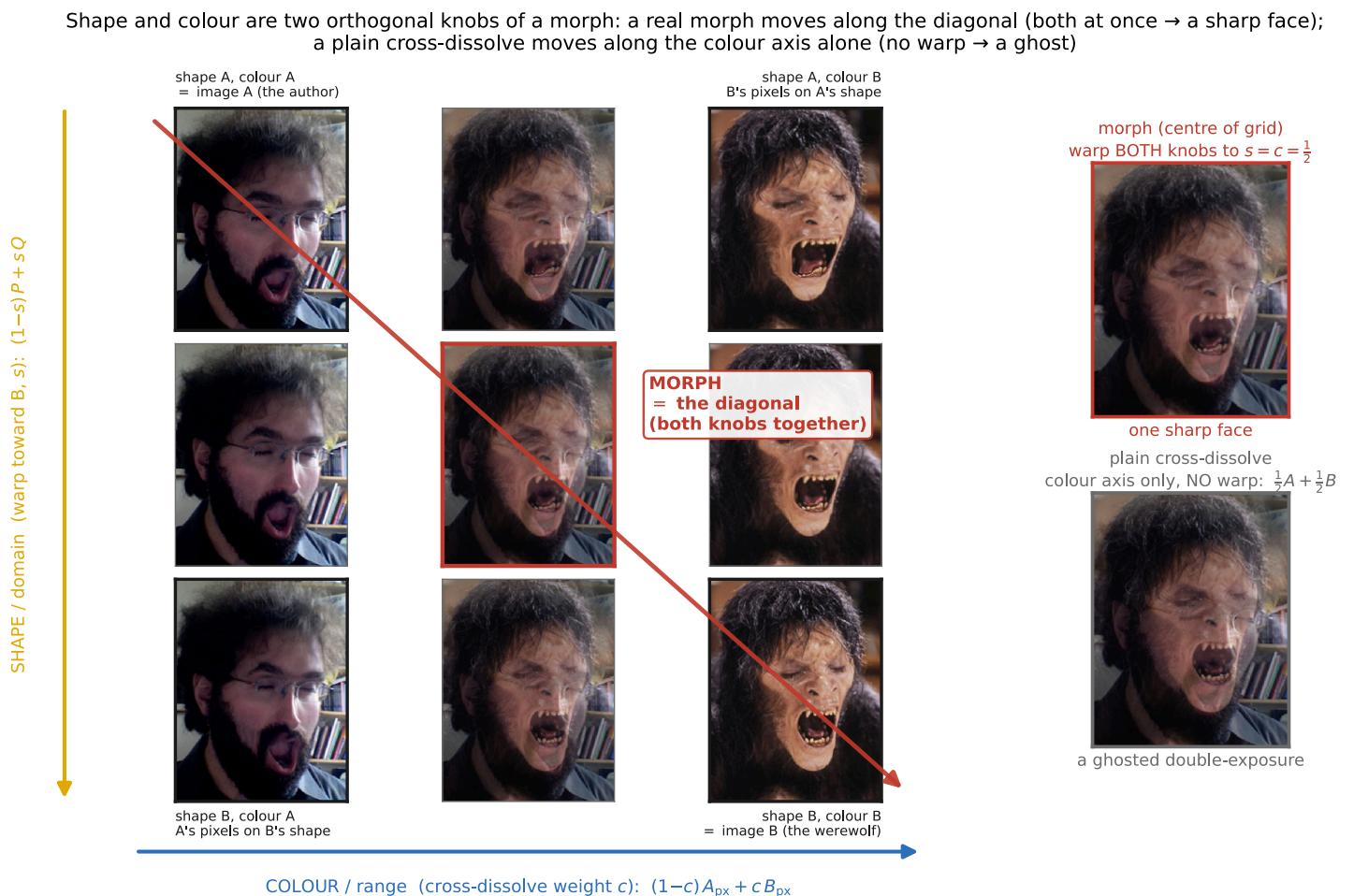


Figure 11.3.2 (reproduced). The two knobs of a morph, as orthogonal axes. One axis, range (color): the cross-dissolve $C(x) = (1 - t)C_0(x) + tC_1(x)$, a linear blend of pixel values. Other axis, domain (shape): a feature point interpolated as a vector, $V = (1 - t)P + tQ$, sliding the feature's location 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-

between 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, $I_t = (1 - t) \text{warp}(I_0) + t \text{warp}(I_1)$. 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.

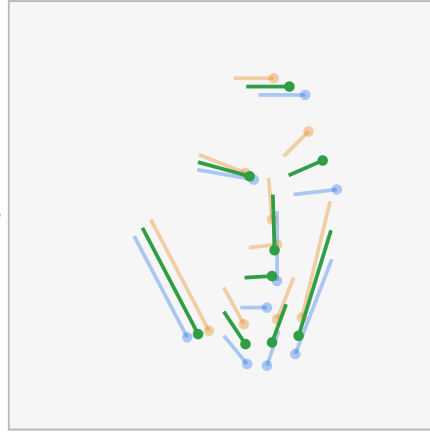
The recipe interpolates both shape and color, and the endpoints prove it. At $t = 0$ the intermediate shape equals I_0 's own shape, so the warp of I_0 is the identity, and the color weight on I_1 is zero: the output is exactly I_0 . Symmetrically at $t = 1$ the output is exactly I_1 . The morph passes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

I_0 and its segments



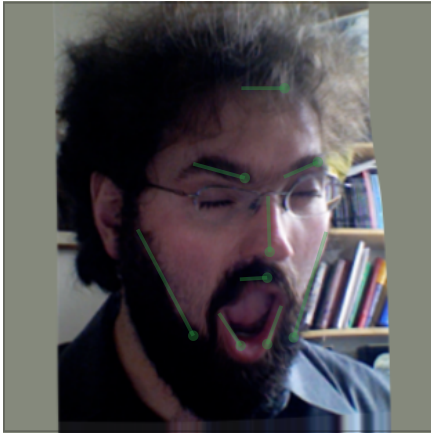
1. $S(t) = (1 - t) S_0 + t S_1$



I_1 and its segments



2. warp(I_0) to $S(t)$



3. $I_t = (1 - t) \text{warp}(I_0) + t \text{warp}(I_1)$



2. warp(I_1) to $S(t)$



$t = 0.50$: weights 0.50 and 0.50

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, $I_t = (1 - t)\text{warp}(I_0) + t\text{warp}(I_1)$. 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

$$I_t(x) = (1 - t) \tilde{I}_0(b_{0,t}(x)) + t \tilde{I}_1(b_{1,t}(x)). \quad (11)$$

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.

Field morphing: the Beier–Neely line-pair warp

[in the book](#) ↗

The first way to turn sparse correspondence into a dense warp is due to [Beier and Neely \(1992\)](#), whose *Feature-Based Image Metamorphosis* is the technique behind the *Black or White* video and a perennial problem-set favorite. The primitive is a pair of directed line segments, one drawn on the "before" image and one on the "after", declaring "*this feature edge here corresponds to that feature edge there*" (the jaw line, the bridge of the nose, the line of an eye). A handful of segments is enough; the warp interpolates everywhere between them.

The mechanics of that warp are built in full in [Chapter 11.1, Warping](#), and we only recall them here. A before/after pair $\overline{PQ} \rightarrow \overline{P'Q'}$ sets up a segment-relative coordinate frame: a normalized coordinate u along the supporting line, with $u = 0$ and $u = 1$ at the segment endpoints and a signed perpendicular distance v in pixels, and a query point keeps its (u, v) as it is reconstructed against the primed segment (u scales with the segment's length, v stays in absolute pixels). Several line pairs are reconciled by the distance- and length-weighted average of their proposed displacements, $w_i = (\text{length}_i^p / (a + \text{dist}_i))^b$ with the capped point-to-segment distance (cross-ref [Figure 11.1.9](#) in [Chapter 11.1, Warping](#), where the (u, v) frame and the weight are derived). As always the map runs backward, (u, v) is computed in the destination and looked up in the source, so every output pixel is filled.

Beier–Neely image morph

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

a 1.0 b 2.0 p 0.50 ☐ scale v like u

A_T — WARPED SOURCE



MORPH(T) — THE BLEND



B_T — WARPED TARGET

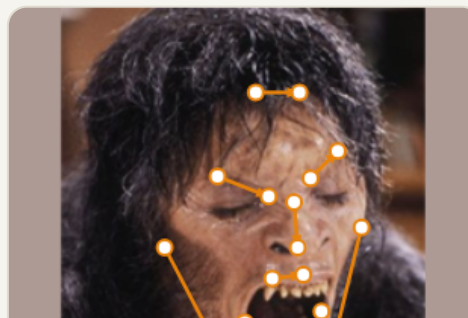


Left and right are the **individual warps** bent to the same intermediate features; the centre is their **cross-dissolve**. Slide t from 0 (pure A) to 1 (pure B).

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



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





Click near an endpoint to grab it; drag to move. Each ■ blue segment on A is paired with the ■ orange segment of the same index on B; segments are directed (arrowheads). The selected feature is highlighted on both images.

The math. Each feature is an A-segment (P, Q) and a B-segment (P'', Q'') . At time t the intermediate segment is $S(t) = (1-t) \cdot (P, Q) + t \cdot (P'', Q'')$. To make A_t , the Beier–Neely field warps A using $A \rightarrow S(t)$ as the source→destination pairs; for B_t it warps B using $B \rightarrow S(t)$. For each output pixel X and segment i , project X onto the destination segment $S(t)_i$ to get u (fraction along, scales) and v (signed perpendicular distance, absolute), rebuild X'_i on the matching source segment, weight the displacement by $w_i = (\text{length}^p / (a + \text{dist}))^b$, and bilinearly sample the source at $X + \sum w_i D_i / \sum w_i$ (inverse warp — hole-free). Finally $\text{morph}(t) = (1-t) \cdot A_t + t \cdot B_t$. With identical A- and B-segments the warps are the identity and the morph is a plain cross-dissolve.

Figure 11.3.4 (reproduced). Interactive: a full Beier–Neely morph between two real images, Frédo's portrait and a werewolf. Draw matching before/after feature segments (jaw, eyes, nose, mouth, hairline) on the two pictures; at each time t every segment is interpolated to an intermediate set, each source is warped to it, and the two are cross-dissolved. The three top panels show the individual warps A_t and B_t and their blend $\text{morph}(t)$; drag the t slider or press play to watch one coherent face, not a ghosted double, pass from Frédo to the werewolf. Drag the segment endpoints to change the correspondence. "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. Werewolf endpoint after the classic course morph. Frédo's portrait © Frédo Durand.

What matters for *morphing* is only how that warp is driven: at parameter t the line pairs are interpolated to an intermediate set $\overline{P_t Q_t} = (1-t)\overline{PQ} + t\overline{P'Q'}$, and each source is warped to that shared intermediate geometry before the cross-dissolve. The same handful of feature lines thus does double duty, defining the dense field *and*, by their own interpolation, the in-between shape.

Strengths and weaknesses. The strength is *sparse, intuitive control*: draw a few feature lines and get a smooth field, with no mesh to manage. The weakness is that every line affects every pixel (global support, so each frame costs $O(\#\text{pixels} \times \#\text{lines})$) and far-apart lines can fight, producing "ghost" pulls and folds where their proposed displacements disagree; getting a, b, p right and debugging the distance function are a real part of the work. Two practical bells and whistles: morph the foreground only (matte it first) so an unconstrained background does not drag artifacts into the result, and use non-uniform timing (let different features morph at different rates) for more lifelike transitions: a turning head might lead with the nose.

Morphable models

A single linear morph turns *one* face into *another* by interpolating a hand-drawn correspondence: it knows about exactly two faces and the straight line between them. Suppose instead you have a *hundred* faces, all registered to one another, and you ask a bigger question: what does the space of *all* of them look like, and can I synthesize, recognize, or edit a face by navigating that space? That is the move from a morph to a morphable model, and it is one of the most influential ideas in this part: a whole family of shapes (faces, bodies, a class of objects) captured as a mean plus a basis of deformations, so that members can be approximated by low-dimensional coordinate vectors. The basis also represents implausible shapes; a prior and validity checks restrict how it is used.

The construction is short to state and is built from machinery you already own. Put a training set of examples into dense correspondence (every example's nose-tip maps to every other's), align away the nuisance pose with Procrustes, and take the principal components (PCA) of the aligned shapes. PCA hands back a mean shape and an ordered basis of eigen-deformations, the directions in which the family varies most, and any instance is written

$$\mathbf{s}(\mathbf{c}) = \bar{\mathbf{s}} + \sum_k c_k \sigma_k \mathbf{e}_k, \quad (12)$$

the mean plus a weighted sum of eigen-modes, with coordinates $\mathbf{c} = (c_1, c_2, \dots)$ measured in units of each mode's standard deviation σ_k . Read in this part's language, $\bar{\mathbf{s}}$ is a *mean shape* and the $\mathbf{e}_k$ are a learned warp basis over it: each eigen-mode is a small canonical deformation of the mean, and an instance is the mean warped by a linear combination of them. A morphable model *is* morphing, with the correspondence field constrained to the subspace the data taught us is plausible ([Figure 11.4.1](#)).

The same idea recurs at three scales of fidelity, and they share one skeleton. In 2-D, Cootes, Edwards and Taylor's Active Shape / Active Appearance Models (ASM 1995 [Cootes et al. 1995 \(ASM\)](#), AAM 2001 [Cootes et al. 2001 \(AAM\)](#)) build PCA models of landmark shapes (and, for AAMs, of the shape-free texture too) and fit them to images. In 3-D, Blanz and Vetter's 3D Morphable Model of faces (1999) [Blanz & Vetter 1999](#) is the landmark result: a PCA model of 3-D face geometry and texture, fit to a *single* photograph by analysis-by-synthesis to recover shape, albedo, pose, and lighting at once. The same recipe gives SMPL for human bodies (Loper et al. 2015) [Loper et al. 2015 \(SMPL\)](#) and FLAME for faces (Li et al. 2017) [Li et al. 2017 \(FLAME\)](#). Egger et al.'s 2020 survey [Egger et al. 2020](#) is the map of the whole territory. We develop the skeleton once; the differences are scale (2-D vs 3-D), what gets a basis (shape, texture, expression), and how the prior is parameterized (linear PCA vs a neural net).

💡 **BIG LESSON (L17, RECURRENCE) — A MORPHABLE MODEL IS A LEARNED WARP BASIS OVER A MEAN**

Everywhere in this part the recipe is build a coordinate map, then transport the pixels; what changes is *how the map is obtained*. A parametric warp *solves* it from constraints; a morph *interpolates* a hand-drawn one between two images; a morphable model *learns the basis of maps* from a training population and then lets you pick one by its coordinates. The mean shape is the origin, the eigen-deformations are the basis vectors, and an instance is a short vector $\mathbf{c}$ of how far to move along each. Folding correspondence into a *learned* basis rather than a per-pair drawing buys a prior: not every warp is allowed, only the ones the family actually exhibits, which is exactly what lets the model fill in an occluded cheek, reject an implausible fit, and generalize from one photo to a full 3-D face. The same shape, *mean + learned basis of deformations + coordinates*, underlies SMPL bodies, FLAME faces, and, at the end of the chapter, the latent spaces of generative networks. (*L17 is registered in the part introduction, [Warping and morphing](#); here the map is a statistical model rather than a drawn or estimated one.*)

🔗 **Connection**

A morphable model is the family-constrained sibling of the per-pair morph of [Chapter 11.3, Morphing](#); its fitting step is an inverse-rendering analysis-by-synthesis that anticipates monocular 3-D reconstruction in [Part 36, 3D and depth](#); and its linear PCA prior is the classical ancestor of the neural image and shape priors in [Chapter 7.3, Deep learning](#) and [Chapter 14.1, What can super-resolution actually do?](#). Downstream it powers identity-preserving portrait editing, relighting, and avatar animation.

Step 4: shape and appearance are *separate* bases

OPTIONAL
in the book ↗

A face is a *shape* and also a *coloring*: skin tone, the dark of the brows and lips, shading. A morphable model handles these with two separate bases, and keeping them separate is exactly the *separate-shape-from-color* discipline of [Chapter 11.3, Morphing](#), now applied to a whole family. Geometry gets the shape PCA above. Appearance (texture) gets its *own* PCA, but the textures are first warped to the mean shape so they are compared in a common, shape-free frame (every face's texture sampled at the same canonical landmarks before averaging). The result is a mean texture $\bar{\mathbf{t}}$ and an appearance basis $\mathbf{a}_k$, and a full instance is the pair

$$\begin{aligned}\mathbf{s}(\mathbf{c}) &= \bar{\mathbf{s}} + \sum_k c_k \sigma_k \mathbf{e}_k, \\ \mathbf{t}(\mathbf{b}) &= \bar{\mathbf{t}} + \sum_k b_k \tau_k \mathbf{a}_k,\end{aligned}\tag{13}$$

with two coordinate vectors: $\mathbf{c}$ for geometry and $\mathbf{b}$ for appearance. To render an instance you synthesize the texture in the shape-free frame, then warp it onto the synthesized shape (the warp engine of [Chapter 11.1, Warping](#), the morphable model's biggest customer after morphing): texture-on-the-mean, deformed to the geometry the shape coordinates request ([Figure 11.4.1](#)). Why separate them? Because they describe different properties: a change in width need not change skin tone, and a smile is primarily a geometric change. Separate bases do not establish statistical independence, and because untangling them is what makes the model useful: relighting edits the appearance/shading without touching identity geometry; re-posing edits geometry without re-painting the texture; recognition can compare two faces in *shape* alone, robust to lighting. Active Appearance Models combine the two into a single joint appearance model; the 3DMM keeps shape, texture, and the imaging parameters (pose, illumination) explicit so each can be read off after fitting.

the face $s(c)$ with appearance $t(b)$



trained on 987 faces, 30 modes each
FFHQ, balanced across seven groups



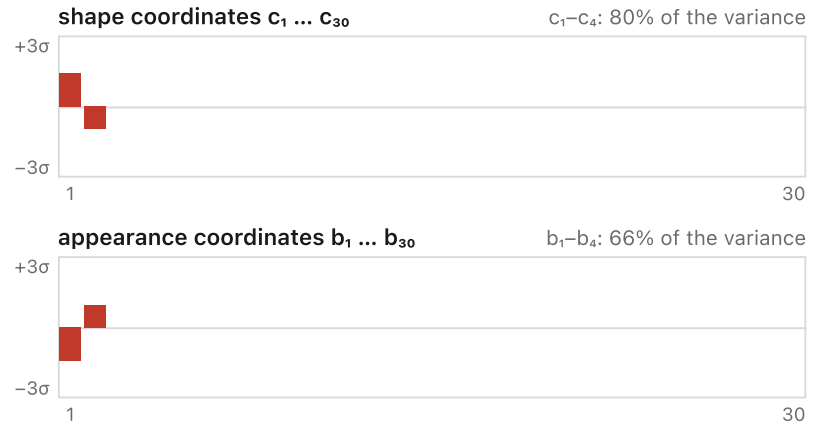
mean: $\bar{s}$, $\bar{t}$



shape only: $s(c)$, $\bar{t}$



appearance only: $\bar{s}$, $t(b)$



correspondence Procrustes modes face space **shape and appearance** explore morph

fit a photograph

Figure 11.4.1, the bases view ([reproduced](#)). Shape and appearance are separate bases. Left: a shape instance $s(c)$ with the mean appearance; blue dots are the mean's landmarks and red arrows their displacement to the instance, drawn twice as long as it is. Middle: an appearance instance $t(b)$ on the mean shape, the shape-free frame in which every face's texture is compared. Right: the two combined, $t(b)$ warped piecewise affinely from the mean shape onto $s(c)$. The sliders set the first four coordinates of each basis and the bars show all 30; a shape slider leaves the middle face unchanged, an appearance slider the left one.

Frame interpolation and slow-motion synthesis

Interpolation = morphing between adjacent frames

OPTIONAL
in the book [↗](#)

The cleanest way to understand frame interpolation is to recognize it as something you have already built. [Chapter 11.3, Morphing](#) takes two images plus a correspondence between them and produces in-betweens by warping both toward an intermediate shape and cross-dissolving. Frame interpolation is precisely that, with two consecutive video frames standing in for the two faces ([Figure 30.4.2](#)). The two endpoints, the warp-to-the-middle, the blend; all identical.

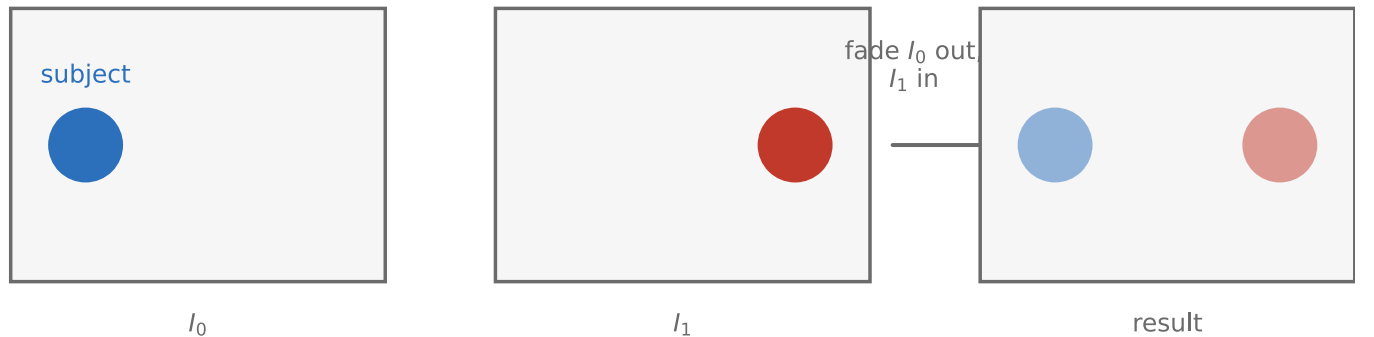
What changes is *where the correspondence comes from*. In classic morphing a human supplies it, drawing Beier–Neely feature lines from one image to the other by hand. In frame interpolation the correspondence is the optical flow, estimated automatically between the two frames. So the whole chapter reduces to one substitution. Frame interpolation is automatic morphing between adjacent frames: motion estimation replaces the hand-drawn correspondence.

Everything that makes interpolation hard, and everything the learned methods buy, lives in that one substitution: the correspondence is no longer given, it must be *recovered*, and it is recovered imperfectly.

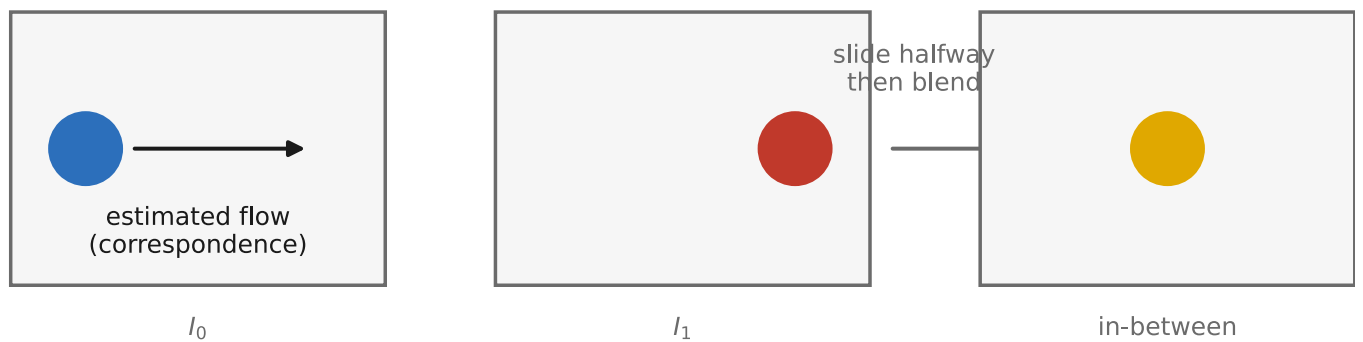
This also explains, in one stroke, why you cannot skip the warp. A plain temporal cross-dissolve (fade frame *A* out while frame *B* fades in, with no warp) does not show an object *moving*; it shows two ghosts of the object, one at each position, the first dimming as the second brightens ([Figure 30.4.2](#)). The eye reads this as a double exposure, not as motion. The warp is exactly what collapses those two ghosts into one object that *slides* from *A*'s position to *B*'s. Morphing's lesson, restated for video: the cross-dissolve hides the seam in *appearance*, but only the warp moves the *geometry*, and motion is geometry.

Frame interpolation = automatic morphing between adjacent frames

plain cross-dissolve



morph / flow-warp



The warp moves the geometry; the cross-dissolve only hides the seam in appearance.

Figure 30.4.2 (reproduced). Interpolation as a morph. Two adjacent video frames, I_0 and I_1 , with a subject in two positions. Plain cross-dissolve (top): fading one frame into the other yields two overlapping ghosts, a double exposure, the subject in both places at once. Morph / flow-warp (bottom): corresponded points slide halfway along the estimated flow before blending, so the subject appears once, in the in-between position: it moves rather than fading. The only difference from a hand-built morph is that the correspondence here is optical flow, estimated automatically rather than drawn by hand.

Flow-based interpolation: warp both frames to the midpoint and blend

OPTIONAL
in the book [↗](#)

Take the morphing recipe literally and you get the classical, flow-based interpolator. Start from the assumption that between two close frames a pixel moves along a roughly straight, constant-velocity path. Then to synthesize the frame at fraction $t \in (0, 1)$:

1. **Estimate flow both ways** between the frames: $\mathbf{F}_{0 \rightarrow 1}$ (where each pixel of I_0 goes in I_1) and $\mathbf{F}_{1 \rightarrow 0}$ (the reverse).

2. **Map source locations toward time t** along scaled forward trajectories: frame 0 is dragged a fraction t of the way toward frame 1, and frame 1 a fraction $1 - t$ of the way back toward frame 0.
3. **Blend** the two warped images, weighted by temporal closeness: a frame near $t = 0$ should look mostly like I_0 , one near $t = 1$ mostly like I_1 .

Written out, the synthesized frame is a convex blend of two backward-warped sources ([Figure 30.4.3](#)):

$$I_t(\mathbf{p}) = (1 - t)I_0(\mathbf{p} + \mathbf{F}_{t \rightarrow 0}(\mathbf{p})) + tI_1(\mathbf{p} + \mathbf{F}_{t \rightarrow 1}(\mathbf{p})). \quad (14)$$

Here $\mathbf{F}_{t \rightarrow 0}$ and $\mathbf{F}_{t \rightarrow 1}$ are backward sampling displacements defined on the intermediate output grid, not simply the endpoint flow fields evaluated at the same coordinates. Read it back: *for each output pixel $\mathbf{p}$, look back along the flow to where it sat in each real frame, sample the color there, and mix the two by how close t is to each frame.* The straight-line assumption is what lets us scale the flow vectors by t : we slide each pixel a fraction of the way along the one displacement we estimated.

Video compression and motion compensation

OPTIONAL
in the book [➔](#)

The still-image part of this book taught one compression idea over and over: don't store what the viewer won't miss, and don't store what you can predict. JPEG ([Chapter 8.11, File formats and compression](#)) commonly subsamples chroma and quantizes transform coefficients, often more coarsely at high frequencies where errors can be less visible. Video adds a second, far larger redundancy that stills cannot touch: the next frame looks almost exactly like this one. At 30–60 frames per second the camera and the world barely move between frames; most of the picture is *already on screen*. Coding each frame independently (so-called Motion-JPEG, literally JPEG per frame) ignores this entirely and is wildly wasteful. Inter-frame codecs exploit prediction from decoded references and code residuals along with the information needed to construct that prediction. That is the whole subject.

💡 BIG LESSON (L17, RECURRENCE)

Correspondence on a budget: motion compensation is optical flow you can afford.

A codec needs, for every block, the same thing [Chapter 12.8, Optical flow](#) asks (*where did this come from in a frame we already have?*), but it does not need a physically correct, dense, sub-pixel flow field. It needs a **coarse, block-constant** correspondence field that is **cheap to estimate** and, above all, **cheap to transmit**, and it chooses that field to **minimize total bits**, not endpoint error. So decades before learned optical flow, video codecs were already estimating dense-ish correspondence at massive scale, optimized for *rate*, not accuracy. The recurring move: *reuse what the decoder already has, and code only the difference*. It returns inverted in [Chapter 38.3, Motion and video magnification](#), which keeps the temporal difference instead of discarding it, and it is the rate-aware cousin of the correspondence story in [Chapter 12.8, Optical flow](#).

Why video compresses far better than still $\times N$: temporal redundancy

OPTIONAL
in the book ↗

Here is the single observation the chapter stands on. Take two consecutive frames of a camera on a tripod and subtract one from the other. The difference image $I_t - I_{t-1}$ is near-zero almost everywhere, black except at moving edges, occlusions, and lighting changes. That near-black difference image is *the only new information* the second frame carries; everything else was already on screen and need not be sent again ([Figure 30.2.2](#)). Measured on real clips, coding a tripod clip losslessly takes 3.1 bits per pixel frame by frame and 0.8 as frame differences. When the camera moves, the plain difference stops working: on a clip with real camera motion it takes 4.5 bits per pixel against 4.9 for the frame alone, and with a pan of 1.5 pixels a frame added it costs more than the frame alone on every clip. Compensating for the motion first brings it back down, to 4.05 bits per pixel on the moving clip, for about 0.01 bit per pixel spent on the motion vectors. That is the rest of this chapter.

The naive baseline ignores this. Motion-JPEG codes every frame as an independent JPEG: simple, edit-friendly, randomly accessible at any frame, but it pays the *full* still-image cost N times and exploits none of the temporal redundancy. It is the "still $\times N$ " strawman, and inter-frame coding can be much more efficient, with the gain depending strongly on content, quality, and codec settings.

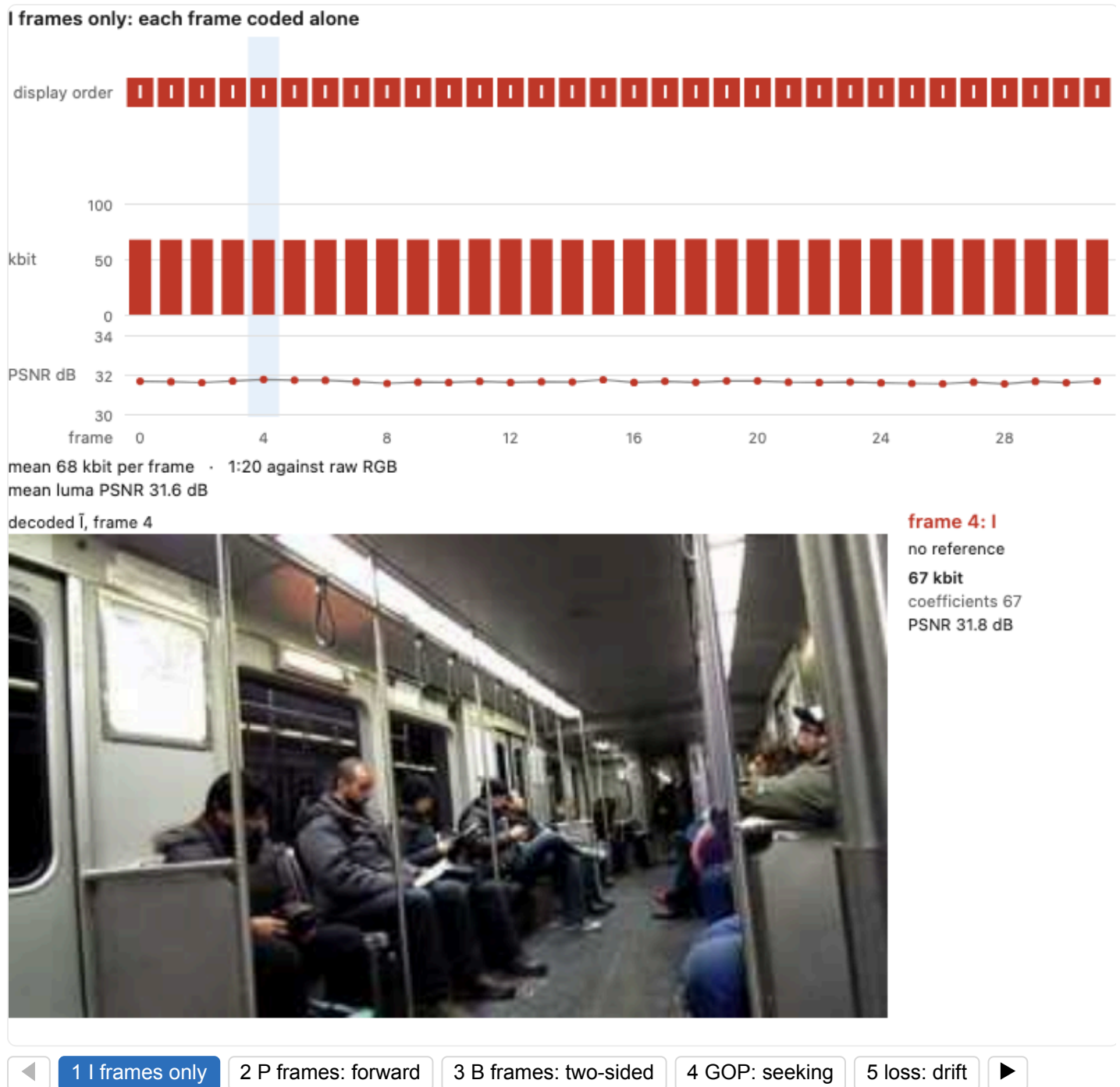


Figure 30.2.1, stage 1 ([reproduced](#)). Every frame of a real video coded alone as an I frame: an 8×8 DCT, the JPEG quantization table scaled by the quality slider, and the baseline Huffman bits each block costs. At quality 50 each subway frame costs about 68 kbit, the same from frame to frame, since nothing here uses the fact that consecutive frames repeat. Clips from Eulerian Video Magnification [Wu et al. 2012](#), and the zipline and the bear in rapids, the author's.

The lever is to stop coding the frame and start coding the surprise. Instead of coding I_t , code the residual between I_t and a prediction $\hat{I}_t$ assembled from already-decoded frames. If the prediction is good the residual is mostly zero, so after the DCT and quantization almost all coefficients vanish, and entropy coding then crushes a block that is nearly all zeros into almost nothing. The slogan is *good prediction = small residual = few bits*.

Two redundancies are now stacked. Video keeps JPEG's spatial and perceptual redundancy removal (the DCT, chroma subsampling, perceptual quantization, adapted from the same ideas as [Chapter 8.11, File formats and compression](#), with codec-specific transforms and coding tools) and adds temporal redundancy removal on top. The temporal one is the large win.

The only question left is how to build $\hat{I}_t$. The crude answer, predict I_t by the previous frame I_{t-1} and code $I_t - I_{t-1}$, works when nothing moves but fails the moment anything does: a moving sharp edge sits in a *different place* in the two frames, so the difference lights up as a bright residual band exactly along every edge, precisely where coding is expensive. The fix is to predict *with motion*: line each block up with where it actually went, so the residual stays small even as the scene moves. That is motion compensation.

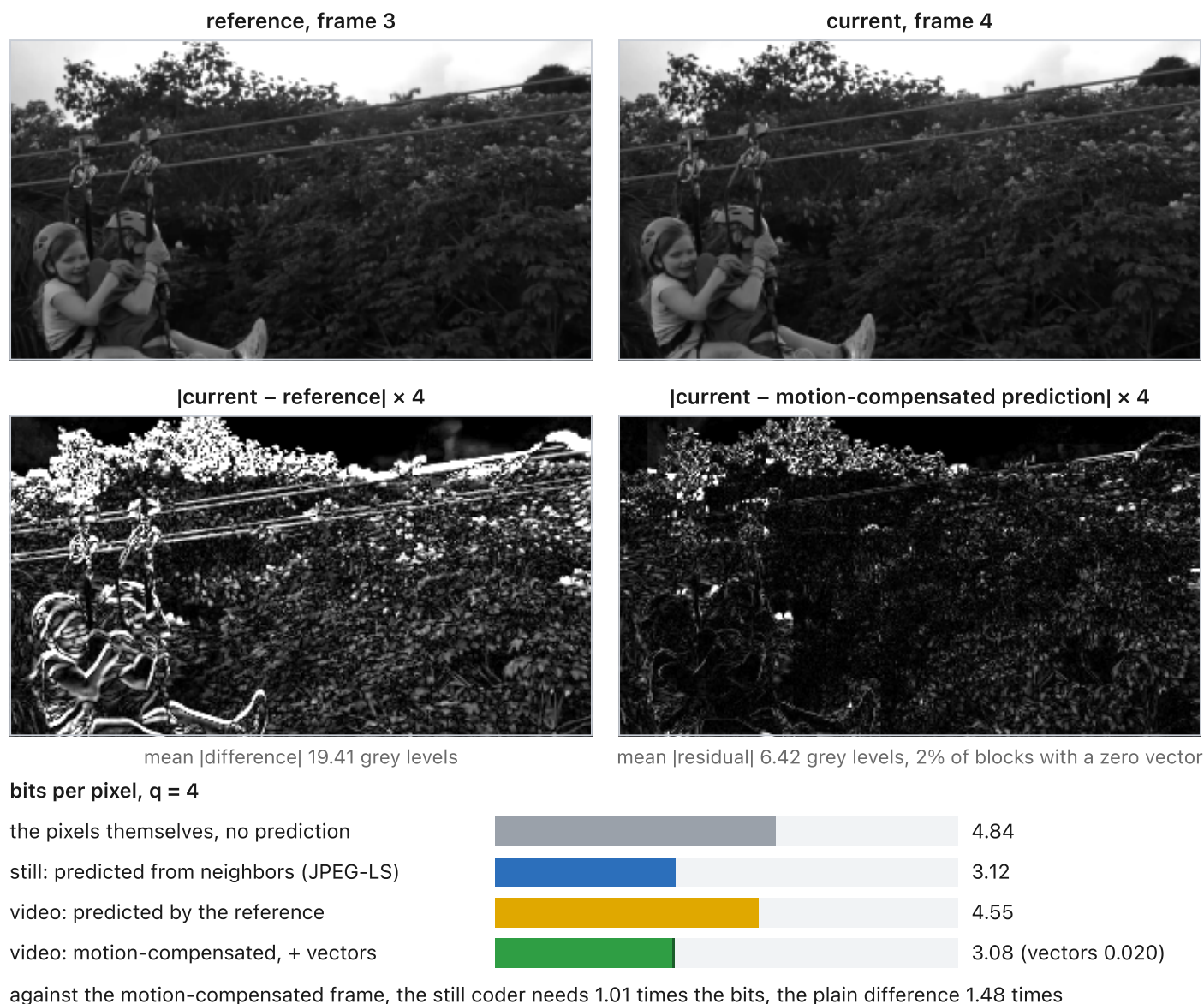


Figure 30.2.2 (reproduced). Temporal redundancy on real video. Top: a reference frame I_{t-k} and the current frame I_t (luma, 320×176). Bottom left: the plain difference $|I_t - I_{t-k}|$, amplified four times; bottom right: the motion-compensated residual $|I_t - \hat{I}_t|$, with $\hat{I}_t$ assembled from 16×16 blocks found by full search to half a pixel. The bars give the cost of each choice of prediction in bits per pixel, estimated as the zeroth-order entropy of the residual quantized with step q : the pixels themselves, the neighbor prediction of a lossless still coder (JPEG-LS [Weinberger, Seroussi & Sapiro 2000 \(LOCO-I, JPEG-LS\)](#)), the reference frame, and the motion-compensated reference plus the vectors' own bits. On the zipline the camera pans after the riders by about four pixels a frame, so the plain difference lights up across the whole picture, while the motion-compensated residual keeps mostly noise and what really changed (3.08 bits per pixel at the default frame, against 4.55 for the difference). Foliage this busy is predicted almost as well from its own neighbors (3.12), and on the bear in rapids, whose foam changes from frame to frame, the still coder wins: a codec decides block by block. The subway clip is shaken by the train by up to a pixel and a quarter a frame (2.51 bits per pixel motion-compensated at frame 4, against 3.69 for the difference and 2.93 for the still coder). The three tripod clips barely move, and there the plain difference is already as good as motion compensation. "Pan added" and "zoom added" move a camera over any clip by resampling its frames: the difference then costs more than coding the frame as a still, while the motion-compensated residual stays close to its tripod

value. Raising q widens every gap. The zipline, the bear and the clouds are the author's videos; the other clips are from the MIT Eulerian Video Magnification project [Wu et al. 2012](#).

Motion-compensated prediction: the core trick

OPTIONAL
in the book [↗](#)

The idea is geometric and simple. A moving object's block in frame t is, to first order, the same pixels as some block in a reference frame, merely displaced. So for each block we find that displacement, copy the matched patch from the reference as our prediction, and code only what is left over.

💡 BIG LESSON (CALLBACK TO L29 · EULERIAN VS LAGRANGIAN)

The crude **delta encoding** above (code $I_t - I_{t-1}$, the difference *at each fixed pixel*) is **Eulerian**: it watches a stationary grid and asks what changed *here*. That is exactly why it collapses under motion: a moving edge changes the value at every pixel it sweeps past, so the Eulerian difference lights up everywhere there is movement, precisely where coding is expensive. **Motion compensation is the Lagrangian fix**: instead of differencing at a fixed location, it *follows the content*: find where each block went, displace the reference to meet it, and code only the small residual. The whole **MPEG / H.264 family is Lagrangian video coding**, and it beats Eulerian delta coding for the same reason warping beats per-pixel editing when things move: *when the world moves, move with it*. (Callback to **L29**, the Eulerian/Lagrangian split introduced in [warping part](#).)

Macroblocks. Partition the frame into blocks, classically 16×16 luma "macroblocks," though modern codecs use variable sizes down to 4×4 . Each block is predicted independently, with its own motion.

Block matching is the correspondence step, and it is exactly the cost-volume search of [Chapter 12.8, Optical flow](#) run per block. For a block B in frame t , search a window in the reference frame for the best-matching patch and record the offset (u, v) , the motion vector:

$$(u, v) = \arg \min_{(u, v)} \sum_{(x, y) \in B} |I_t(x, y) - I_{\text{ref}}(x - u, y - v)|. \quad (15)$$

This cost is the sum of absolute differences (SAD); the sum of *squared* differences (SSD) is its smooth cousin; the same matching primitive as [Chapter 12.8, Optical flow](#), just summed over a whole block instead of a single pixel's neighborhood. Per block, it is a search over the displacement window for the cheapest match:

Algorithm 3

```
for each block B in frame I_t:
    best =  $\infty$ 
    for (u, v) in search window:
        sad =  $\sum_{(x,y) \text{ in } B} |I_t[x, y] - I_{\text{ref}}[x-u, y-v]|$ 
        if sad < best:
            best = sad
            mv[B] = (u, v)
    residual[B] = B - I_ref shifted by mv[B]
```

The search is sped up with hierarchical, coarse-to-fine search and with predicted starting points from neighboring blocks, and sub-pixel motion vectors (half- or quarter-pel, found by interpolating the reference) sharpen the match further. [Figure 30.2.3](#) runs this search on real video.

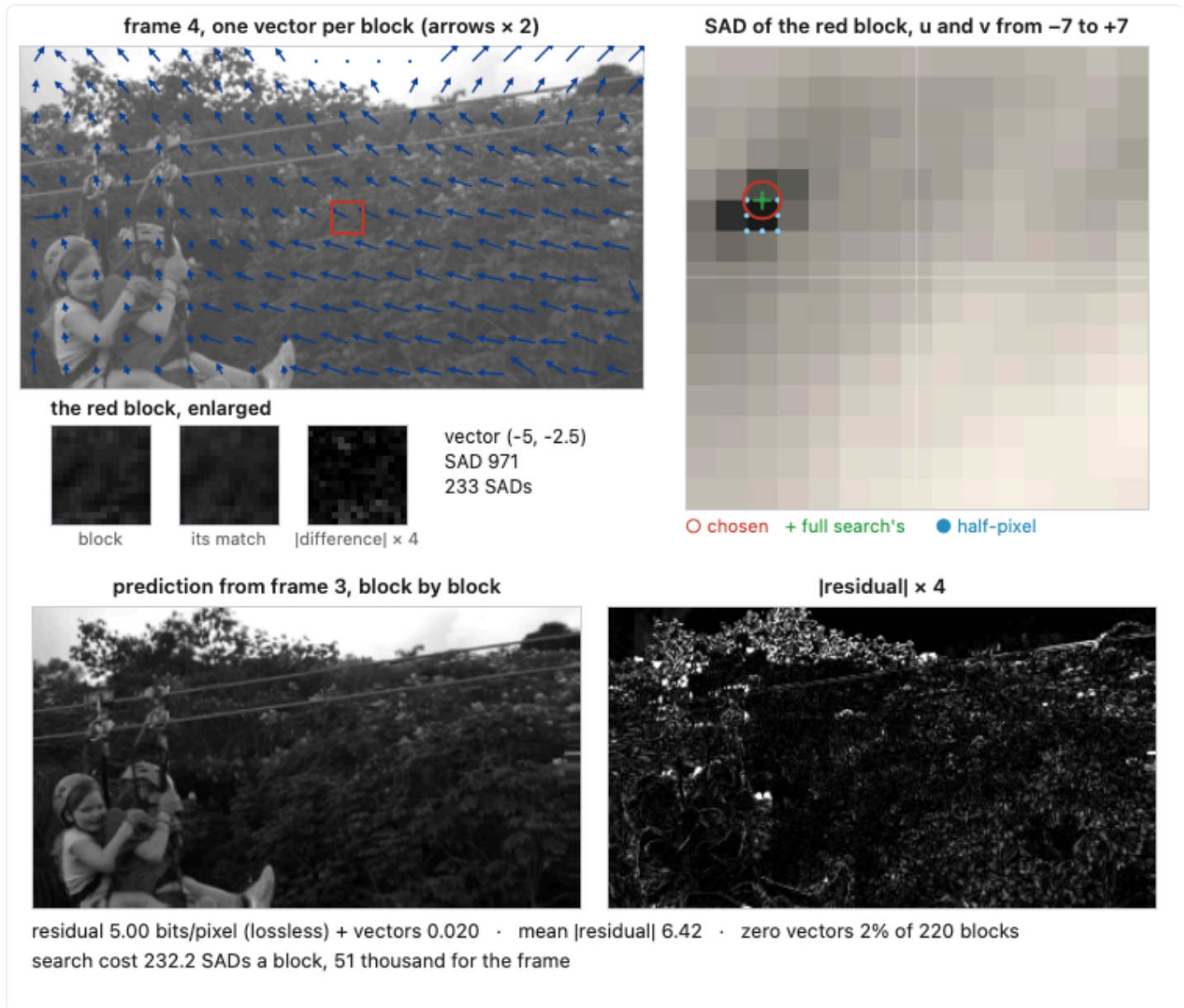


Figure 30.2.3 (reproduced). Block matching on real video. Top left: frame I_t cut into blocks, each block's motion vector (u, v) drawn as an arrow, lengthened to be visible. Click a block: top right shows its SAD over the whole search window (dark is small), the vector chosen and where full search lands; under the frame, the block, its match in I_{t-1} and their difference, enlarged. Bottom: the prediction $\hat{I}_t(x, y) = I_{\text{ref}}(x - u, y - v)$ assembled block by block, and the residual r . Full search evaluates every whole-pixel offset, $(2R + 1)^2$ SADs a block. Three-step search [Koga et al. 1981 \(three-step search\)](#) and diamond search [Zhu & Ma 2000 \(diamond search\)](#) evaluate 13 to 33 and assume the SAD surface has a single basin; half-pixel refinement adds 8 SADs on the codec's bilinear interpolation. On the subway clip, three-step and diamond search pick full search's vector on 95% and 97% of blocks at a ninth and a sixteenth of the cost; with a pan added, three-step falls to 75%. On the zipline, where the camera pans about four pixels a frame through foliage, they agree with full search on 70% and 66% of blocks. A block on a vertical pole has a valley in its SAD surface rather than a pit (the aperture problem), and flat regions, such as the zipline's sky, get noisy, arbitrary vectors. The zipline, the bear and the clouds are the author's videos; the other clips are from the MIT Eulerian Video Magnification project [Wu et al. 2012](#).

The prediction is then just the reference frame shifted by that vector,

$$\hat{I}_t(x, y) = I_{\text{ref}}(x - u, y - v), \quad (16)$$

and the residual, the thing we actually code, is what the prediction missed,

$$r(x, y) = I_t(x, y) - \hat{I}_t(x, y), \quad (17)$$

small wherever the match was good.

Transform-code the residual. Apply a codec-defined transform, quantize its coefficients under the chosen quality or quantization-parameter setting, and entropy-code them. This follows the transform/quantize/code pattern of [Chapter 8.11, File formats and compression](#), but modern codecs need not use JPEG’s transform sizes, tables, or entropy coder. The codec transmits, per block, prediction-mode and reference information, motion information, and a coded residual where needed. Skip and merge modes can infer some of these rather than sending them explicitly. (The motion vectors are themselves predictively coded from their neighbors, since they too are spatially correlated; adjacent blocks tend to move together.)

The decoder mirrors the encoder. It decodes the residual, fetches the same reference patch via the transmitted motion vector, and adds them: $\tilde{I}_t = \hat{I}_t + \tilde{r}$. Here tildes denote decoded quantities, including quantization error. One subtlety keeps the two sides in lock-step: the encoder predicts from the reconstructed (lossy) reference the decoder will actually have, not from the pristine original. If it predicted from the original, the small quantization errors would compound frame after frame and the two sides would slowly drift apart; predicting from the reconstruction makes encoder and decoder see *the same* reference and eliminates drift.

[Figure 30.2.1](#) runs this loop on real video.

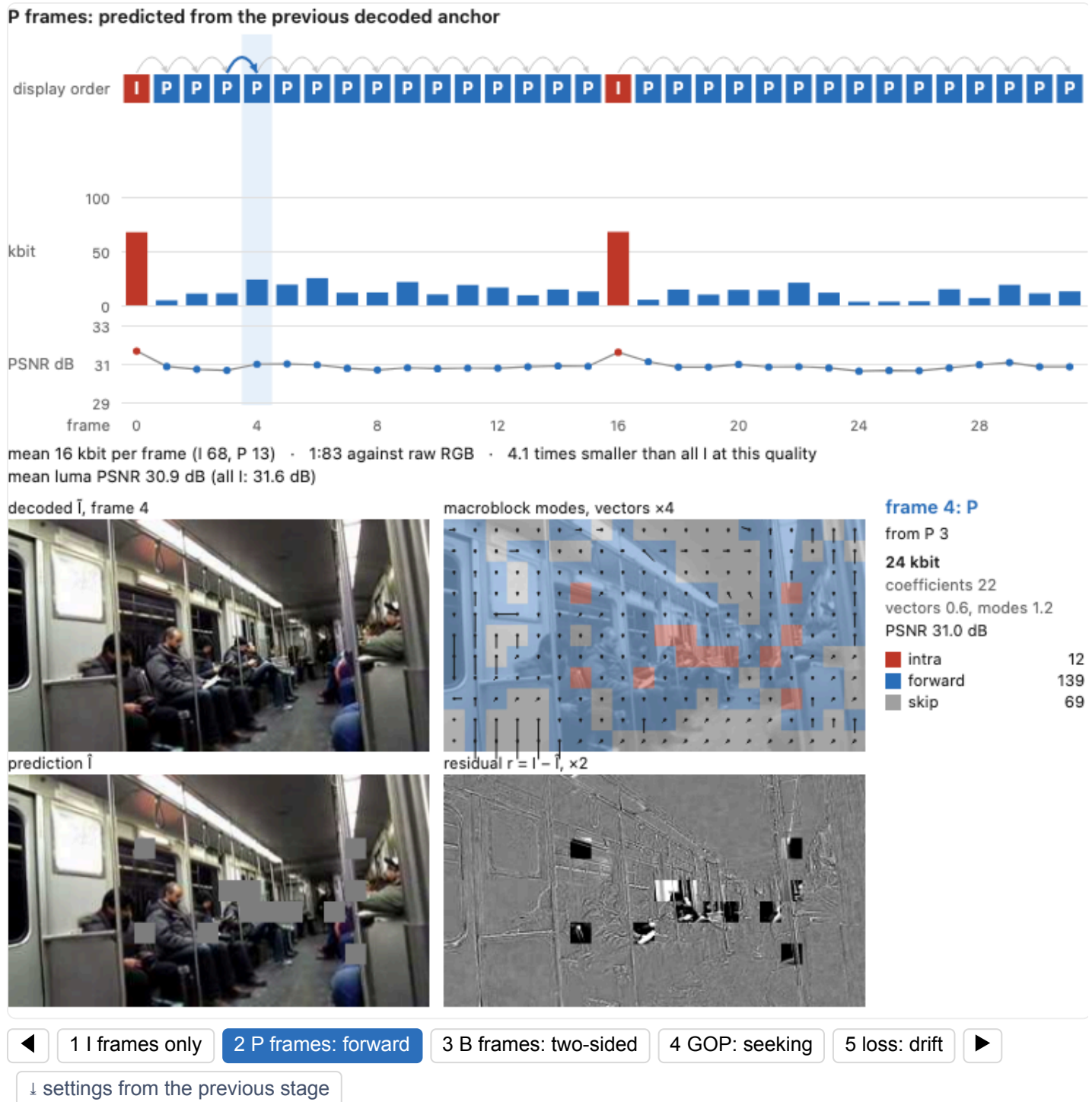


Figure 30.2.1, stage 2 (reproduced). A small MPEG-style codec running in the page on 32 frames of real video, here with P frames (I P P P ..., a GOP of 16). Each 16×16 macroblock of a P frame is predicted from the previous decoded frame by a half-pel motion vector (u, v) found by block matching, and the encoder picks per macroblock whichever of skip (the predicted vector, no residual, one bit), a coded residual $r = I_t - \hat{I}_t$, or intra coding minimizes $D + \lambda R$. The lower panels show the prediction $\hat{I}_t$ and the residual (gray is zero, contrast doubled); the mode map colors each macroblock by its choice and draws its vector four times longer. On the subway clip, shaken by the train, a P frame costs about a fifth of an I frame; on the guitar and the sleeping baby nearly every macroblock is skipped and a P frame costs under 1 kbit against 53 to 81 kbit for an I frame. On the zipline, where the camera pans through foliage, a P frame still costs about 60% of an I frame (36 against 59 kbit) and a sixth of its macroblocks are coded

intra, a third on the bear in rapids. An intra macroblock's prediction is flat gray, so its residual is the picture itself. Click a column to inspect a frame. Clips from Eulerian Video Magnification [Wu et al. 2012](#), and the zipline and the bear in rapids, the author's.

Rate–distortion is the real objective. The encoder does not pick the *most accurate* motion vector. It picks the one that minimizes

$$D + \lambda R, \tag{18}$$

the reconstruction distortion D plus λ times the bits R needed to send the vector and its residual. A slightly worse match that codes far cheaper wins outright. This $\min D + \lambda R$ trade-off is the same Lagrangian balance that sets a quantization table's aggressiveness in [Chapter 8.11, File formats and compression](#). It is *why* the motion field is on a budget: it is optimized for bits, not for physical correctness ([Figure 30.2.4](#)).

Motion-compensated prediction loop

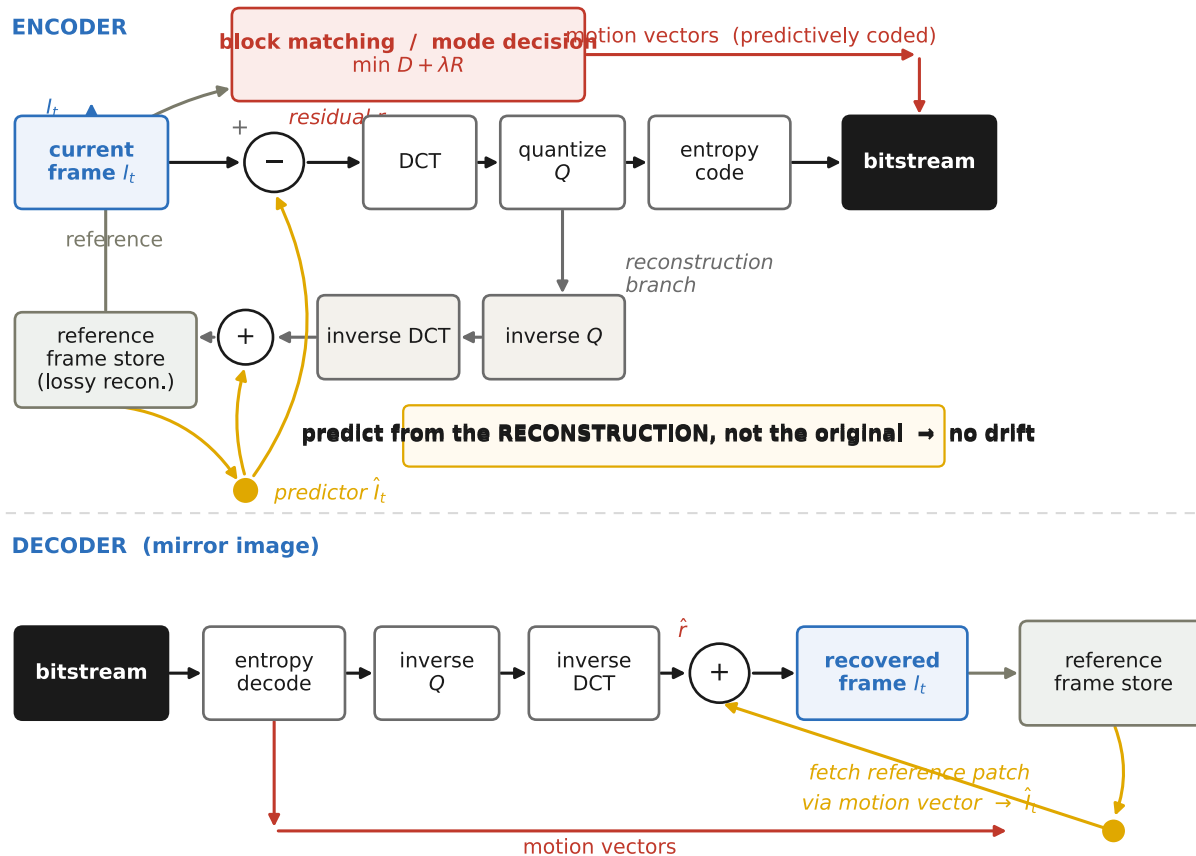


Figure 30.2.4 (reproduced). The motion-compensated prediction loop. Encoder (top): the current frame I_t and a stored reference frame enter block matching, which emits motion vectors; the predictor $\hat{I}_t$ is subtracted from I_t to form the residual r , which goes through DCT $\rightarrow$ quantize $\rightarrow$ entropy-code into the bitstream alongside the (predictively coded) motion vectors. A reconstruction branch inverse-quantizes and adds $\hat{I}_t$ back to store the lossy reference. Decoder (bottom): a mirror image. Entropy-decode, inverse DCT, fetch the reference patch via the motion vector, add to recover I_t . The whole rate–distortion choice $\min D + \lambda R$ lives in the block-matching / mode-decision box.

Why this is "optical flow on a budget"

OPTIONAL
in the book [↗](#)

State the equivalence plainly, because it is the whole reason this chapter lives next to [Chapter 12.8, Optical flow](#). The field of motion vectors over all the blocks of a frame is a correspondence field: a piecewise-constant, one-vector-per-block approximation to the optical flow between the frame and its reference. Block motion compensation supplies a prediction correspondence field, computed at scale in inter-frame codecs ([Figure 30.2.5](#)).

But it is the *budget* version, and the differences are the point.

- **Block-constant, not per-pixel.** One vector covers a whole block, even though the true flow varies within it. The residual absorbs the within-block error. The codec does not need the flow to be right, only cheap, because whatever it gets wrong is paid for once, in residual bits.

- **Rate-optimal, not accuracy-optimal.** The vector is chosen to minimize *bits* ($D + \lambda R$), so the encoder will happily pick a physically wrong vector when it codes cheaper. A codec's motion field is not a measurement of scene motion; do not read it as ground-truth flow.
- **Block-matching, not variational.** A simple encoder uses SAD/SSD search, related to [Chapter 12.8, Optical flow](#), without a global smoothness solve. Actual encoders use varied search and mode-decision strategies; the bitstream standard specifies decoding rather than one required search algorithm. What smoothness there is comes implicitly, from predictively coding the vectors and from the rate cost of disagreeing with neighbors.
- **Graceful fallback.** When prediction is hopeless, as in a scene cut, fast new content, or a large occlusion, the encoder simply codes the block, or the whole frame, intra, degrading gracefully back to the still-image path.

The conceptual bridge runs both ways. The *same* aperture problem, search window, and sub-pixel interpolation appear in the codec and in flow estimation. Learned optical flow (RAFT and its kin, [Teed & Deng 2020](#); see [Chapter 12.8, Optical flow](#)) and learned video codecs both build on this cost-volume correspondence skeleton; the codec just got there first, optimized for a different loss.

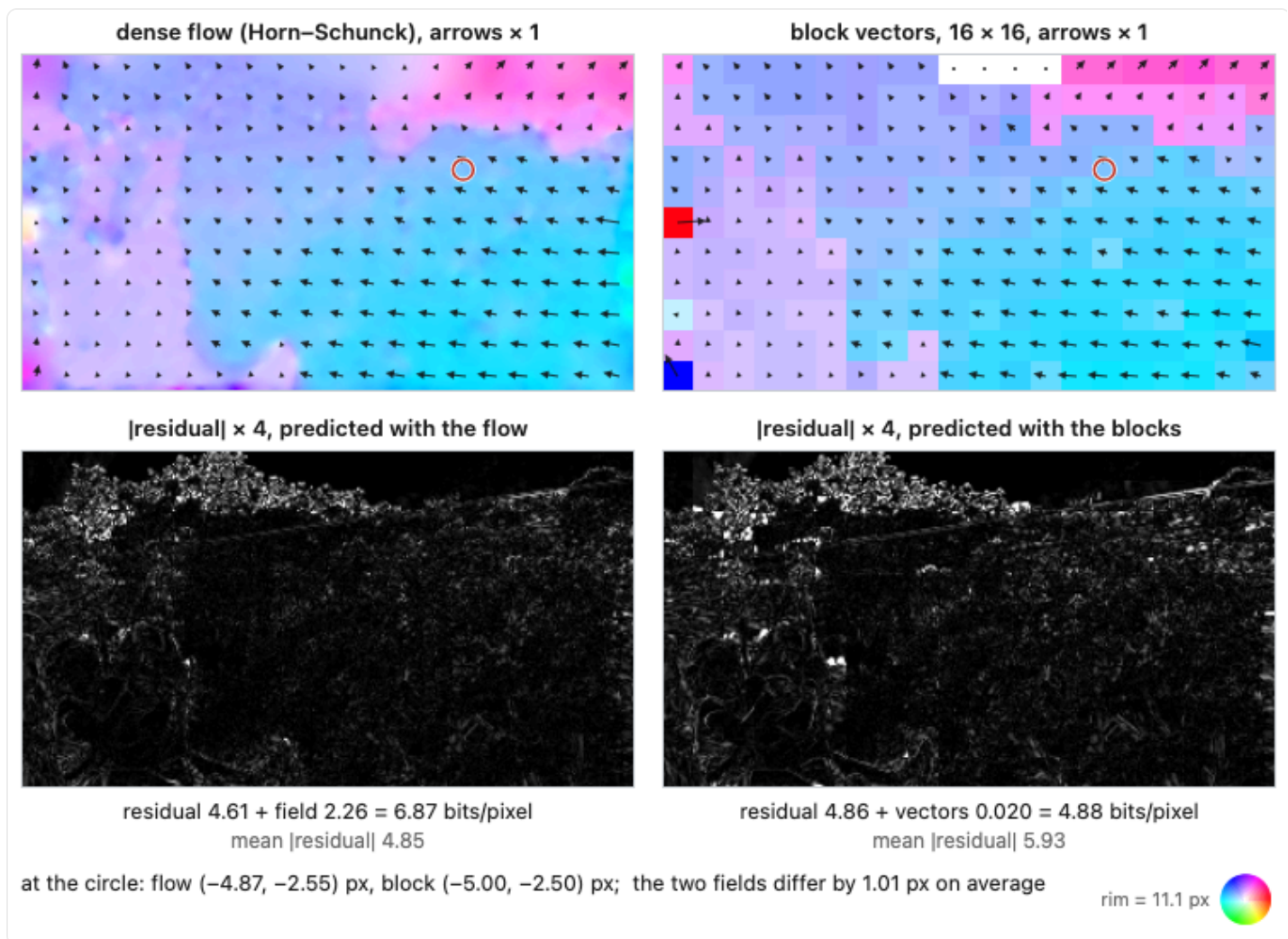


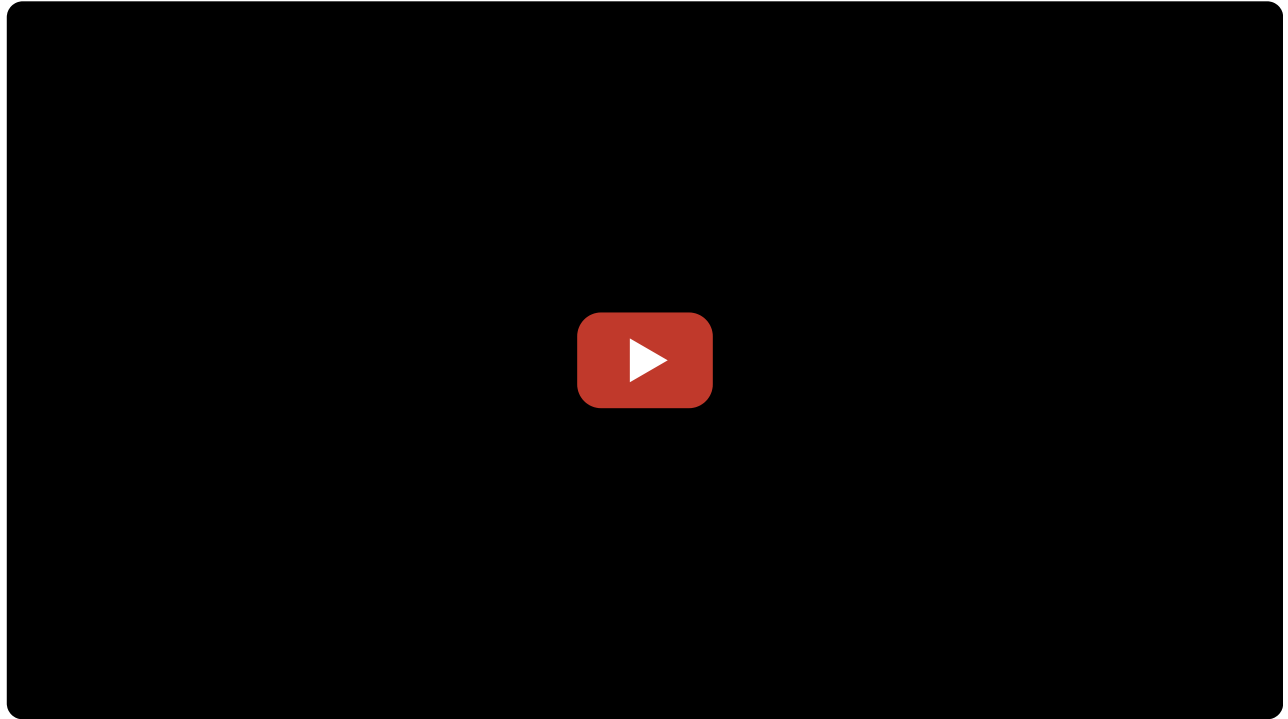
Figure 30.2.5 (reproduced). Same correspondence, two budgets, on real frames. Top left: a dense optical flow between I_{t-1} and I_t , Horn and Schunck's method run coarse to fine with warping and median filtering [horn-schunck-1981; @sun-et-al-2010]. Top right: the block vectors a codec would send, one per 16×16 or 8×8 block. Both use the chapter's convention, $I_t(x, y) \approx I_{ref}(x - u, y - v)$, and the same color code, hue for direction and saturation for length Baker et al. 2011. Bottom: the residual each field leaves and its cost in bits per pixel, residual plus field; block vectors are predicted from their neighbors, and the dense field is sent at a quarter pixel with each value predicted from its left neighbor. On the zipline as filmed, the dense field costs 2.26 bits per pixel and the blocks win, 4.88 to 6.87 at the default frame. With a zoom added to a tripod clip (baby and chart, frame 4) the true motion is known: the dense flow is closer to it (0.15 against 0.55 pixel) and leaves the smaller residual (2.96 against 3.30 bits per pixel), but describing it costs 0.70 bits per pixel against 0.01, so the blocks win on the total, as they do on every clip and camera motion here. The smoothness weight α trades one cost against the other. Click either field to read both vectors at a point. The zipline, the bear and the clouds are the author's videos; the other clips are from the MIT Eulerian Video Magnification project Wu et al. 2012.

The flip side of "graceful fallback" is a failure mode you have certainly seen: when the motion is genuinely unpredictable, as in falling snow, confetti, churning water, or a field of static, there is no good block match anywhere, the residual balloons, and at a fixed bitrate the whole frame smears into mush. It is the most vivid everyday demonstration that video compression depends strongly on predictability, and Tom Scott's popular explainer makes exactly this point (Figure 30.2.6).



Most of our universe consists of dark matter rendered completely undetectable by our spacetime codec's dynamic range issues.

Figure 30.2.6 [\(reproduced\)](#). xkcd #2414 "Solar System Compression Artifacts". Randall Munroe, xkcd.com/2414 (CC BY-NC 2.5).



Video 4. Video: *Why Snow and Confetti Ruin YouTube Video Quality*: a clear popular explainer of how motion-compensated, inter-frame video compression collapses when the motion is high-entropy and unpredictable (snow, confetti), defeating block prediction and ballooning the residual. Tom Scott (**YouTube**).

Recap: big lessons in this lecture

💡 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](#).

💡 **BIG LESSON (L17, RECURRENCE) — A MORPHABLE MODEL IS A LEARNED WARP BASIS OVER A MEAN**

Everywhere in this part the recipe is build a coordinate map, then transport the pixels; what changes is *how the map is obtained*. A parametric warp *solves* it from constraints; a morph *interpolates* a hand-drawn one between two images; a morphable model *learns the basis of maps* from a training population and then lets you pick one by its coordinates. The mean shape is the origin, the eigen-deformations are the basis vectors, and an instance is a short vector $\mathbf{c}$ of how far to move along each. Folding correspondence into a *learned* basis rather than a per-pair drawing buys a prior: not every warp is allowed, only the ones the family actually exhibits, which is exactly what lets the model fill in an occluded cheek, reject an implausible fit, and generalize from one photo to a full 3-D face. The same shape, *mean + learned basis of deformations + coordinates*, underlies SMPL bodies, FLAME faces, and, at the end of the chapter, the latent spaces of generative networks. (*L17 is registered in the part introduction, [Warping and morphing](#); here the map is a statistical model rather than a drawn or estimated one.*)

💡 **BIG LESSON (L17, RECURRENCE)**

Correspondence on a budget: motion compensation is optical flow you can afford.

A codec needs, for every block, the same thing [Chapter 12.8, Optical flow](#) asks (*where did this come from in a frame we already have?*), but it does not need a physically correct, dense, sub-pixel flow field. It needs a **coarse, block-constant** correspondence field that is **cheap to estimate** and, above all, **cheap to transmit**, and it chooses that field to **minimize total bits**, not endpoint error. So decades before learned optical flow, video codecs were already estimating dense-ish correspondence at massive scale, optimized for *rate*, not accuracy. The recurring move: *reuse what the decoder already has, and code only the difference*. It returns inverted in [Chapter 38.3, Motion and video magnification](#), which keeps the temporal difference instead of discarding it, and it is the rate-aware cousin of the correspondence story in [Chapter 12.8, Optical flow](#).

💡 **BIG LESSON (CALLBACK TO L29 · EULERIAN VS LAGRANGIAN)**

The crude **delta encoding** above (code $I_t - I_{t-1}$, the difference *at each fixed pixel*) is **Eulerian**: it watches a stationary grid and asks what changed *here*. That is exactly why it collapses under motion: a moving edge changes the value at every pixel it sweeps past, so the Eulerian difference lights up everywhere there is movement, precisely where coding is expensive. **Motion compensation is the Lagrangian fix**: instead of differencing at a fixed location, it *follows the content*: find where each block went, displace the reference to meet it, and code only the small residual. The whole **MPEG / H.264 family is Lagrangian video coding**, and it beats Eulerian delta coding for the same reason warping beats per-pixel editing when things move: *when the world moves, move with it*. (Callback to **L29**, the Eulerian/Lagrangian split introduced in [warping part](#).)