

Image Warping and Morphing

Paul Heckbert, Sept. 1999

15-869, Image-Based Modeling and Rendering

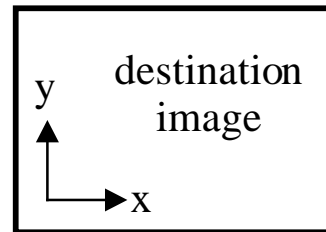
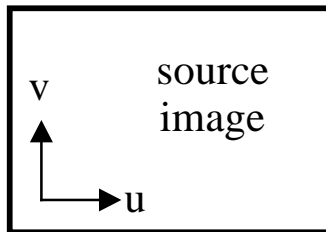
Image Warping

Image warping = rearranging the pixels of a picture.

Also called “image distortion”, “geometric image transformation”, and sometimes “geometric correction”.

It’s useful for both image processing and for computer graphics (namely, for texture mapping).

To do image warping, you need the function that maps points between corresponding points in the source and destination images. This function is called the **mapping** or “transformation”.



If (u,v) are source coordinates and (x,y) are destination coords, you need either the forward mapping: $x=x(u,v)$ & $y=y(u,v)$ or the inverse mapping : $u=u(x,y)$ & $v=v(x,y)$.

Mapping Types

Simple mappings:

- affine mapping
- projective mapping
- bilinear mapping

These can be applied globally or over a subdivision of the plane, e.g.

- piecewise affine over a triangulation
- piecewise projective over a quadrilaterization (sp?)
- piecewise bilinear over a rectangular grid

Or other, arbitrary functions can be used, e.g.

- Beier-Neely warp (popular for morphs)
- piecewise cubic
- store $u[x,y]$ and $v[x,y]$ in large arrays

Affine Mappings

Have the form:

$$\begin{aligned}u &= ax+by+c \\v &= dx+ey+f\end{aligned}$$

In matrix notation:

$$\begin{bmatrix}u \\v \\1\end{bmatrix} = \begin{bmatrix}a & b & c \\d & e & f\end{bmatrix} \begin{bmatrix}x \\y \\1\end{bmatrix} \quad \text{or} \quad \begin{bmatrix}u \\v \\1\end{bmatrix} = \begin{bmatrix}a & b & c \\d & e & f \\0 & 0 & 1\end{bmatrix} \begin{bmatrix}x \\y \\1\end{bmatrix}$$

A combination of 2-D scale, rotation, and translation transformations.

This is the class of transformations that Postscript supports.

Allows a square to be distorted into any parallelogram.

6 degrees of freedom ($a-f$).

Inverse is of same form (is also affine). Given by inverse of 3x3 matrix above.

Good when controlling a warp with triangles, since 3 points in 2-D determine the 6 degrees of freedom.

Projective Mappings (a.k.a. “perspective”)

Have the form:

$$u = (ax+by+c)/(gx+hy+i)$$

$$v = (dx+ey+f)/(gx+hy+i)$$

In matrix notation:

$$\begin{bmatrix} uq \\ vq \\ q \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \text{ and } \begin{matrix} u = uq/q \\ v = vq/q \end{matrix}$$

Linear numerator & denominator.

If $g=h=0$ then you get affine as a special case.

Allows a square to be distorted into any quadrilateral.

8 degrees of freedom ($a-h$). We can choose $i=1$, arbitrarily.

Inverse is of same form (is also projective). Given by adjoint (or inverse, if it exists) of the matrix above.

Good when controlling a warp with quadrilaterals, since 4 points in 2-D determine the 8 degrees of freedom.

Bilinear Mappings

Have the form:

$$u = axy + bx + cy + d$$

$$v = exy + fx + gy + h$$

If $a=e=0$ then you get affine as a special case.

Allows a square to be distorted into any quadrilateral.

8 degrees of freedom ($a-h$).

Inverse is not of same form (but it requires square root(s) - slow!).
Therefore not recommended if you need the inverse.

Performing an Image Warp, 1

Destination scanning:

```
for y = ymin to ymax
  for x = xmin to xmax
    u = u(x,y)
    v = v(x,y)
    copy pixel at source[u,v] to dest[x,y]
```

this is the simplest method; some aliasing problems, however

Source scanning:

```
for v = vmin to vmax
  for u = umin to umax
    x = x(u,v)
    y = y(u,v)
    copy pixel at source[u,v] to dest[x,y]
```

often leads to “holes” (unwritten pixels), but this can be fixed by drawing “fat pixels”

Performing an Image Warp, 2

Two-pass method: [Catmull-Smith, SIGGRAPH 80]

```
for v                                first, resample rows
  for u
    x = x(u,v)
    copy source[u,v] to temp[x,v]
```

```
for x                                then, resample columns
  for v
    y = y(x,v)
    copy temp[x,v] to dest[x,y]
```

works out cleanly for affine and projective warps

advantage: all filtering is 1-D, so more amenable to hardware implementation

Resampling Filters

Image warping requires resampling: converting a digital signal from one sampling grid to another. It's 2-D signal resampling.

If you copy pixels (“point sampling”), as in the previous pseudocode, results are ugly.

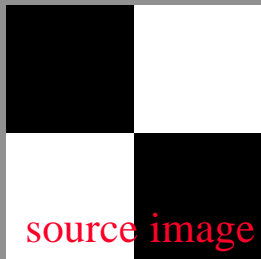
As in 1-D resampling, filtering must be used:

- if the mapping scales up (stretches), the operation is called upsampling or interpolation. *If done poorly, you get rastering (e.g. pixel replication).*
- if the mapping scales down (squeezes), the operation is called downsampling or decimation. *If done poorly, you get aliasing (e.g. moire).*

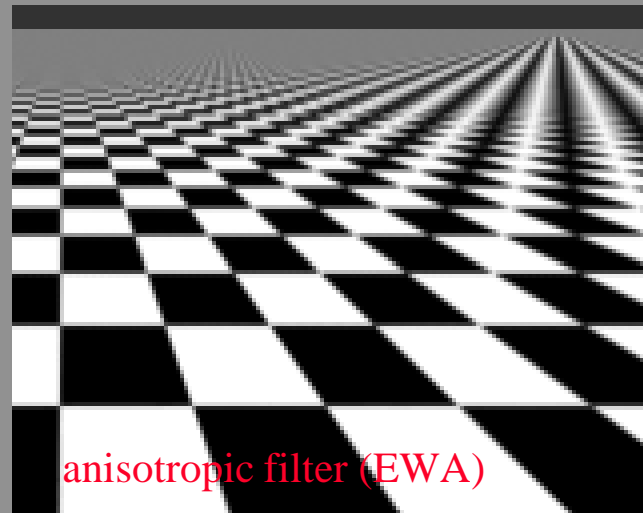
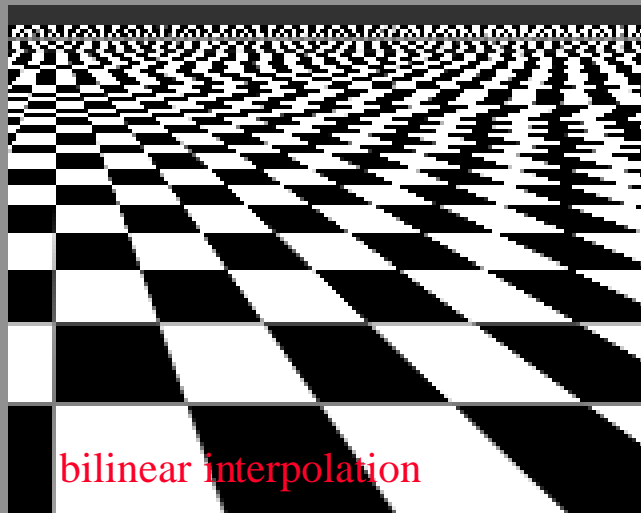
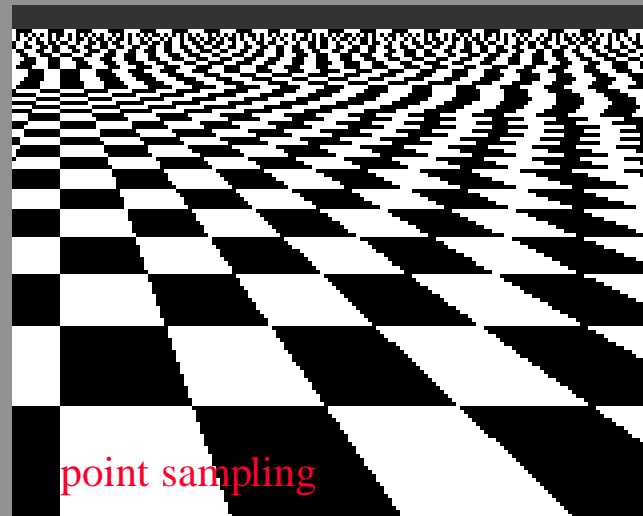
High quality resampling with arbitrary scale factors requires careful use of low pass filters of variable support (width). But for a non-affine warp, a *shift-variant* filter is needed -- these are not widely studied in the signal processing literature.

Good resampling for scale factors near 1 (not scaling up or down much) can be done with bilinear interpolation. OK for most image warping.

Comparison of Resampling Filters



this is a very strenuous test; most image warping doesn't downsample as much as we see near the "horizon" of this image



Bilinear Interpolation

An inexpensive, continuous function that interpolates data on a square grid:

Within each square, if the corner values are p_{00} , p_{10} , p_{01} , p_{11} , at points $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$, respectively, then to interpolate at point (x,y) , where x and y are between 0 and 1:

$$p_{xy} = (1-x)*(1-y)*p_{00} + x*(1-y)*p_{10} \\ + (1-x)*y*p_{01} + x*y*p_{11}$$

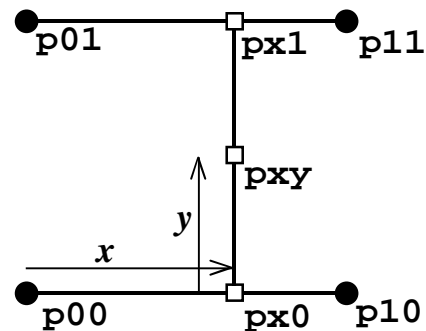
If working with RGB pictures, do the same operation to each of the three channels, independently.

To optimize the above, do the following, which takes 3 multiplies instead of 8:

$$p_{x0} = p_{00} + x*(p_{10}-p_{00})$$

$$p_{x1} = p_{01} + x*(p_{11}-p_{01})$$

$$p_{xy} = p_{x0} + y*(p_{x1}-p_{x0})$$



Morphing

Morphing (short for “metamorphosis”) is the visual transformation of one object into another, usually using 2-D image processing techniques. 3-D metamorphosis is more complex.

You could just cross-dissolve, but that looks artificial, non-physical. Instead:

morph = warp the shape & cross-dissolve the colors.

Usually you do the warp and cross-dissolve simultaneously.

Cross-dissolving is the easy part; warping is the hard part.

To cross-dissolve by a number *dfrac* in the range [0,1] between picA and picB:

```
for y = ymin to ymax
  for x = xmin to xmax
    temp[x,y].r = picA[x,y].r + dfrac*(picB[x,y].r-picA[x,y].r)
    temp[x,y].g = picA[x,y].g + dfrac*(picB[x,y].g-picA[x,y].g)
    temp[x,y].b = picA[x,y].b + dfrac*(picB[x,y].b-picA[x,y].b)
```

Beier & Neely's Morphing Method

That Beier & Shawn Neely's morph method [SIGGRAPH '92] is one of the best in existence. They also warp the shape & cross-dissolve the colors, independently.

First, let's look at their warping method. Then we'll turn to morphing.

Basic idea of **their warping method**, to warp a source image to a dest. image†:

1. Specify the correspondence between source image and destination image interactively using a set of line segment pairs.
2. Concoct a continuous function that maps destination image points to source image points.
 - a. Given a point in destination image, determine “weights” of each line segment based on distance of point from line & length of line in destination image.
 - b. For each line segment, compute a displacement vector to add to dest point.
 - c. Compute weighted average of displacements and add to dest point to compute source point.

†Note: source image is not necessarily “picA” and dest image is not necessarily “picB”

Beier&Neely's Morph: Sequence of Operations

- Read in two picture files, picA and picB, and one lines file.
Lines file contains line segment pairs PQ_{iA} , PQ_{iB} .
- Compute destination line segments by linearly interpolating between PQ_{iA} and PQ_{iB} by *warpfraction*. *These line segments define the “destination shape”*.
- Warp picture A to destination shape, computing a new picture†. We'll call the result “Warped A”.
- Warp picture B to destination shape, computing a new picture†. We'll call the result “Warped B”.
- Cross dissolve between Warped A and Warped B by *dissolvefrac*.
- Write the resulting picture to a file.

†Use bilinear interpolation when reading from picA or picB, to avoid blockiness.

Morphing Beyond Images

Volumetric Morphing [Lerios, SIGGRAPH 95]

Beier-Neely in 3-D

Morphing Implicit Surfaces [Turk, SIGGRAPH 99]

implicit surface is set of points $\mathbf{x} : f(\mathbf{x})=0$ where $\mathbf{x} = (x,y,z)$

take two implicit surfaces, $a(\mathbf{x})=0$ and $b(\mathbf{x})=0$

create a morph between them:

$$m(\mathbf{x},t) = (1-t) a(\mathbf{x}) + t b(\mathbf{x}) \text{ -- pick } t, \text{ find all } \mathbf{x} : m(\mathbf{x},t) = 0$$

Morphing Polygonal Meshes [Lee, SIGGRAPH 99]

parametrize mesh A and mesh B (tricky to do without much distortion!)

establish correspondence between meshes a la Beier-Neely

mutually subdivide so meshes have the same topology

linearly interpolate geometry