Image Warping and Morphing

OUTLINE:

- Image Warping
- Morphing
- Beier and Neely’s Morphing Method

Image Warping

Point processing and filtering don't move pixels around.

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 coordinates, then you need either $x=x(u,v)$ & $y=y(u,v)$ or $u=u(x,y)$ & $v=v(x,y)$. Usually, the latter are more useful, because they allow you to use...

Destination scanning (simplest way to perform image warping):

```
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]
```

Simple Mappings

There are many ways to create useful mappings from the 2-D source space to the 2-D destination space, but the simplest are:

Affine mappings:

$x = au+bv+c$

$y = du+ev+f$

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

Allows a square to be distorted into any parallelogram. 6 degrees of freedom (*a*-*f*).

Inverse is of same form (is also affine). *Good for triangles.*

Projective mappings (a.k.a. “perspective”):

$x = (au+bv+c)/(gu+hv+i)$

$y = (du+ev+f)/(gu+hv+i)$

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). *Good for quads.*

Bilinear mappings:

$x=auv+bu+cv+d$

$y=euv+fu+gv+h$

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 (it requires square root(s) - slow!). *Not recommended.*

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

Thad Beier & Shawn Neely's morph method [SIGGRAPH '92] is probably 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 $\boldsymbol{PQ_{iA}}$, $\boldsymbol{PQ_{iB}}$.
- Compute destination line segments by linearly interpolating between $\boldsymbol{PQ_{iA}}$ and $\boldsymbol{PQ_{iB}}$ 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.

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):

```
pxy = (1-x)*(1-y)*p00 + x*(1-y)*p10
    + (1-x)*y*p01 + x*y*p11
```

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:

```
px0 = p00 + x*(p10-p00)
px1 = p01 + x*(p11-p01)
pxy = px0 + y*(px1-px0)
```