Image Filtering, Warping, and Morphing

Michael Kazhdan

(601.457/657)

Outline

- Image Filtering
- Image Warping
- Image Morphing

Image Filtering

- What about the case when the modification that we would like to make to a pixel depends on the pixels around it?
  - Blurring
  - Edge Detection
  - Etc.

Multi-Pixel Operations

Stationary/Local Filtering

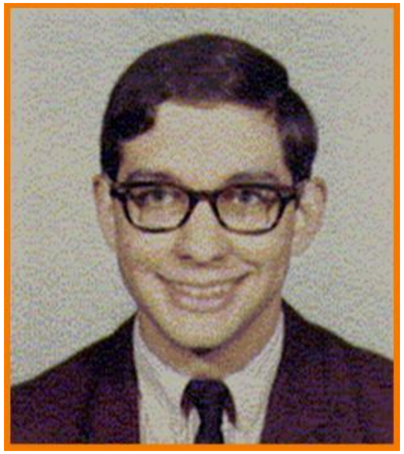
A general approach is to:

1. Define a *mask* of weights telling us how values at adjacent pixels should be combined to generate the new value.
2. Apply the (same) mask at every pixel.*

*Care is needed around at boundary pixels.

Blurring

- To blur across pixels, define a mask:
  - Whose entries sum to one
  - Whose value is larger near the center of the mask
  - Whose values are non-negative



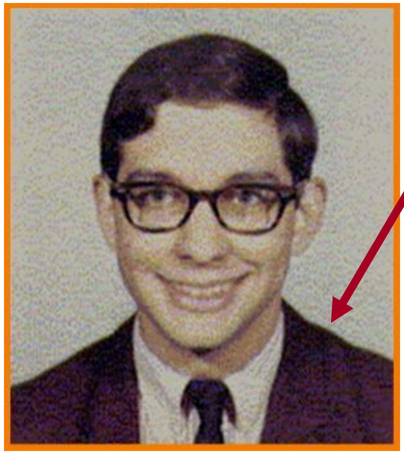
Original

Blur

$$\text{Filter} = \begin{bmatrix} 1/16 & 2/16 & 1/16 \\ 2/16 & 4/16 & 2/16 \\ 1/16 & 2/16 & 1/16 \end{bmatrix}$$

Blurring

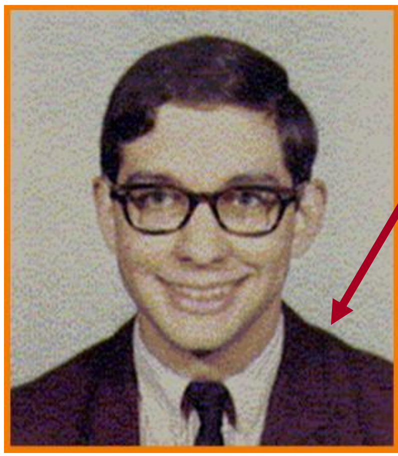
Pixel(x,y): red = 36
green = 36
blue = 0



Original

$$\text{Filter} = \begin{bmatrix} 1/16 & 2/16 & 1/16 \\ 2/16 & 4/16 & 2/16 \\ 1/16 & 2/16 & 1/16 \end{bmatrix}$$

Blurring



Original

Pixel(x,y): red = 36
green = 36
blue = 0

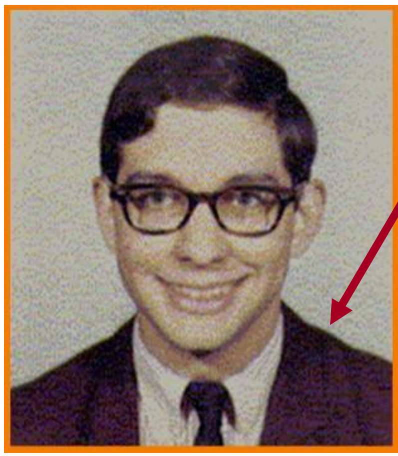
	x - 1	x	x + 1
y - 1	36	109	146
y	32	36	109
y + 1	32	36	73

Pixel(x,y).red and its red neighbors

$$\text{Filter} = \begin{bmatrix} 1/16 & 2/16 & 1/16 \\ 2/16 & 4/16 & 2/16 \\ 1/16 & 2/16 & 1/16 \end{bmatrix}$$

Blurring

New value for Pixel(x,y).red =
(36 * 1/16) + (109 * 2/16) + (146 * 1/16)
(32 * 2/16) + (36 * 4/16) + (109 * 2/16)
(32 * 1/16) + (36 * 2/16) + (73 * 1/16)



Original

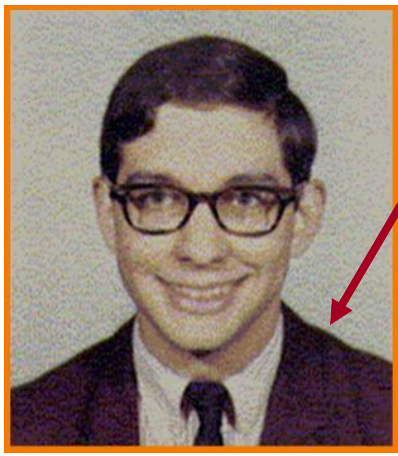
	x - 1	x	x + 1
y - 1	36	109	146
y	32	36	109
y + 1	32	36	73

Pixel(x,y).red and its red neighbors

$$\text{Filter} = \begin{bmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{bmatrix}$$

Blurring

New value for Pixel(x,y).red = 62.69



Original

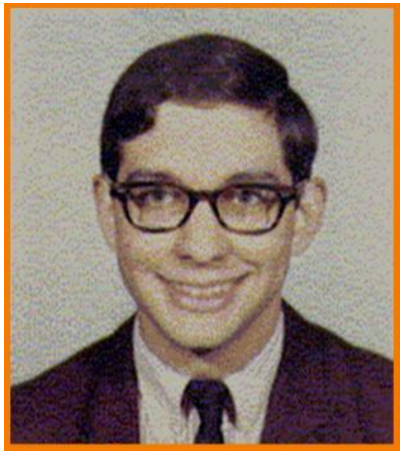
	x - 1	x	x + 1
y - 1	36	109	146
y	32	36	109
y + 1	32	36	73

Pixel(x,y).red and its red neighbors

$$\text{Filter} = \begin{bmatrix} 1/16 & 2/16 & 1/16 \\ 2/16 & 4/16 & 2/16 \\ 1/16 & 2/16 & 1/16 \end{bmatrix}$$

Blurring

New value for Pixel(x,y).red = 63



Original

Blur

$$\begin{bmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{bmatrix}$$

Note: Though we quantized, we could have used another techniques (e.g. dithering)

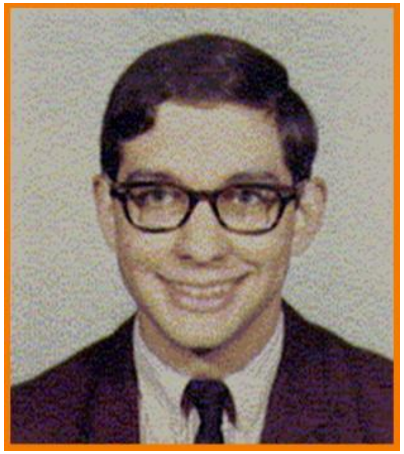
Blurring

- Repeat for each color channel of each pixel.
- Keep source and destination separate to avoid drift.
- For boundary pixels, not all neighbors are used:
  - » Normalize the mask so the values sum to one, or
  - » Assume that the exterior values are black, or
  - » Assume the exterior values can be obtained by reflecting the image across the boundary, or
  - » Assume…

Blurring

- Masks can have arbitrary size:
  - Can expand smaller masks by zero-padding

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 2 & 4 & 2 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} /16 \Leftrightarrow \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} /16$$



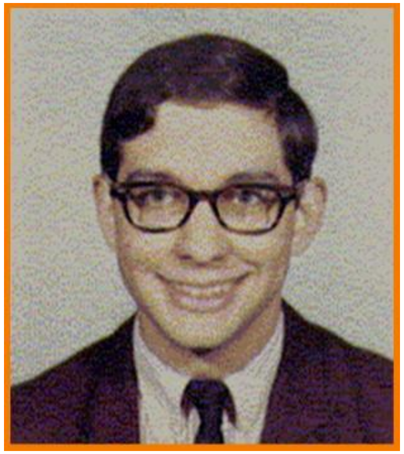
Original

Narrow Blur

Blurring

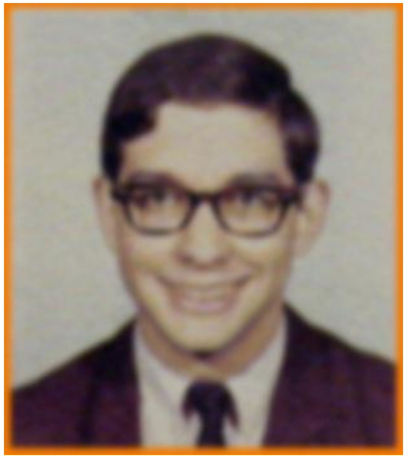
- Masks can have arbitrary size:
  - Can expand smaller masks by zero-padding
  - Can use more non-zero entries to get a wider blur

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 2 & 4 & 2 & 0 \\ 0 & 1 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} /16 \qquad \begin{bmatrix} 0 & 1 & 2 & 1 & 0 \\ 1 & 2 & 4 & 2 & 1 \\ 2 & 4 & 8 & 4 & 2 \\ 1 & 2 & 4 & 2 & 1 \\ 0 & 1 & 2 & 1 & 0 \end{bmatrix} /48$$



Original

Narrow Blur



Wide Blur

Blurring

A general way for defining the entries of a mask of size $(2r+1) \times (2r+1)$ is to evaluate a Gaussian:

$$\text{GaussianMask}[i][j] \sim e^{-\frac{(i-r)^2+(j-r)^2}{4r^2}}$$
$$i,j \in [0,2r]$$

- r is the (integer) mask radius
- $0 \le i \le 2r$ is the horizontal position in the mask
- $0 \le j \le 2r$ is the vertical position in the mask
- **Don't forget to normalize!**

Note:
The center of the mask is at index (r,r).

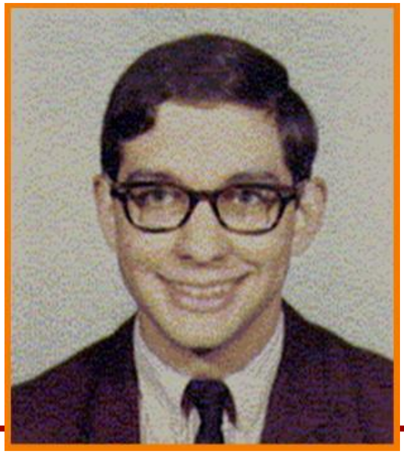
Edge Detection

An edge is where the image is far from constant:

⇒ The difference between the value at the pixel and the average value of neighboring pixels is large (in absolute value)

Define a mask whose:

- Entries sum to zero
- Value is one at the center pixel



$$\text{Filter} = \frac{1}{8}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge Detection

This is sometimes referred to as a **Laplacian** filter.

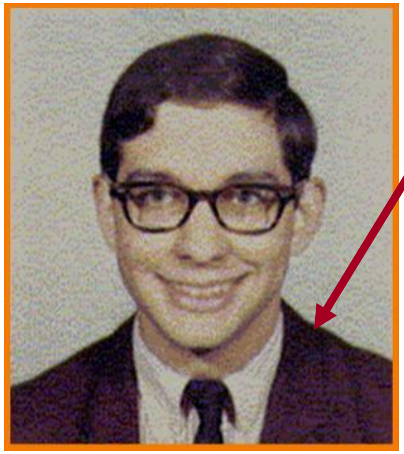
Pixels with large absolute values correspond to edges:
- Positive values: “upper” edges
- Negative values: “lower” edges

Define a mask whose:
- Entries sum to zero
- Value is one at the center pixel

$$\text{Filter} = \frac{1}{8}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge Detection

Pixel(x,y): red = 36
green = 36
blue = 0



Original

$$\text{Filter} = \frac{1}{8}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge Detection

Pixel(x,y): red = 36
green = 36
blue = 0

Original

	x - 1	x	x + 1
y - 1	36	109	146
y	32	36	109
y + 1	32	36	73

Pixel(x,y).red and its red neighbors

$$\text{Filter} = \frac{1}{8}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge Detection

New value for Pixel(x,y).red =
(36 * -1/8) + (109 * -1/8) + (146 * -1/8)
(32 * -1/8) + (36 * 1) + (109 * -1/8)
(32 * -1/8) + (36 * -1/8) + (73 * -1/8)

Original

	x - 1	x	x + 1
y - 1	36	109	146
y	32	36	109
y + 1	32	36	73

Pixel(x,y).red and its red neighbors

$$\text{Filter} = \frac{1}{8}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge Detection

New value for Pixel(x,y).red = -285/8

Original

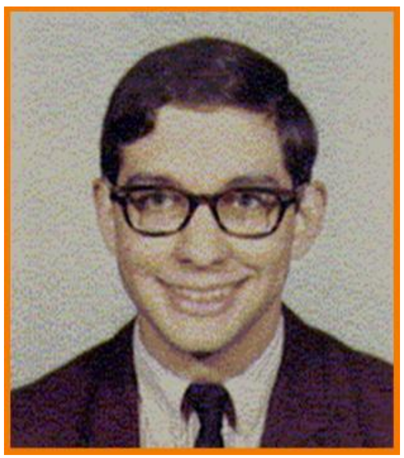
	x - 1	x	x + 1
y - 1	36	109	146
y	32	36	109
y + 1	32	36	73

Pixel(x,y).red and its red neighbors

$$\text{Filter} = \frac{1}{8}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Edge Detection

New value for Pixel(x,y).red = -35.625



Original

Detected Edges

$$\text{Filter} = \frac{1}{8}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Note:

Output values **are not colors**.

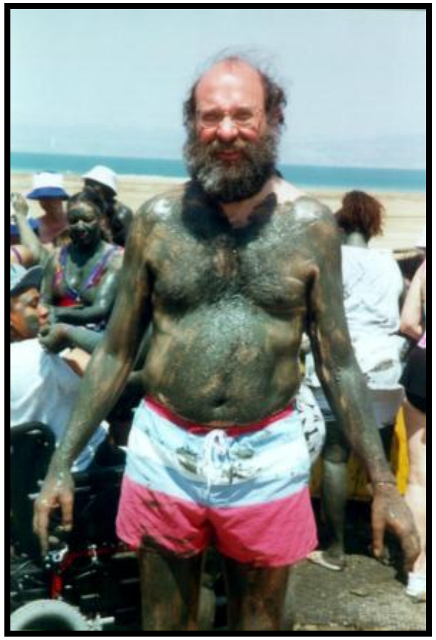
⇒ Need to find a way to remap for visualization.

Outline

- Image Filtering
- Image Warping
- Image Morphing

Image Warping

- Move pixels of image
  - Mapping
  - Resampling



Warp

Source image

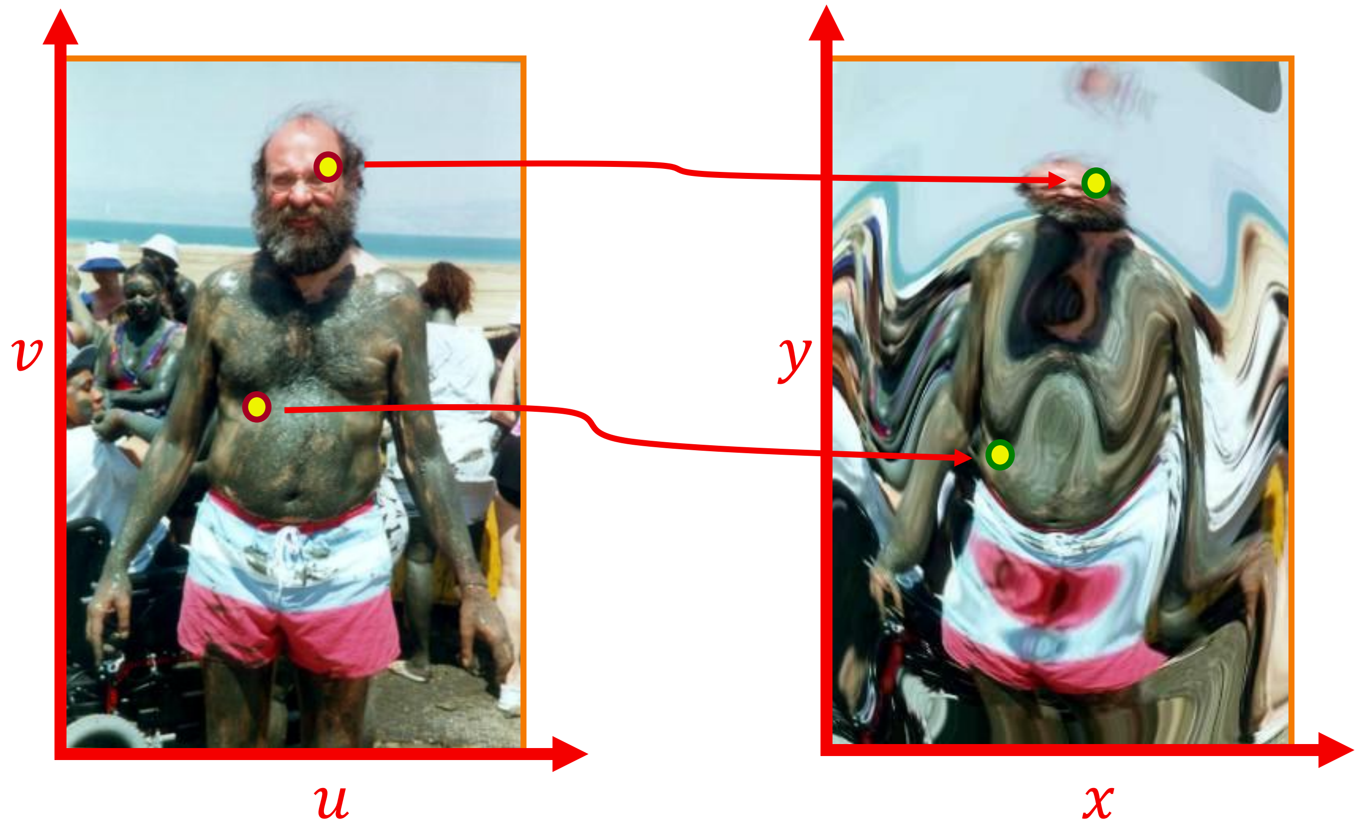
Destination image

Overview

- Mapping
  - Forward
  - Inverse
- Resampling
  - Point sampling
  - Triangle filter
  - Gaussian filter

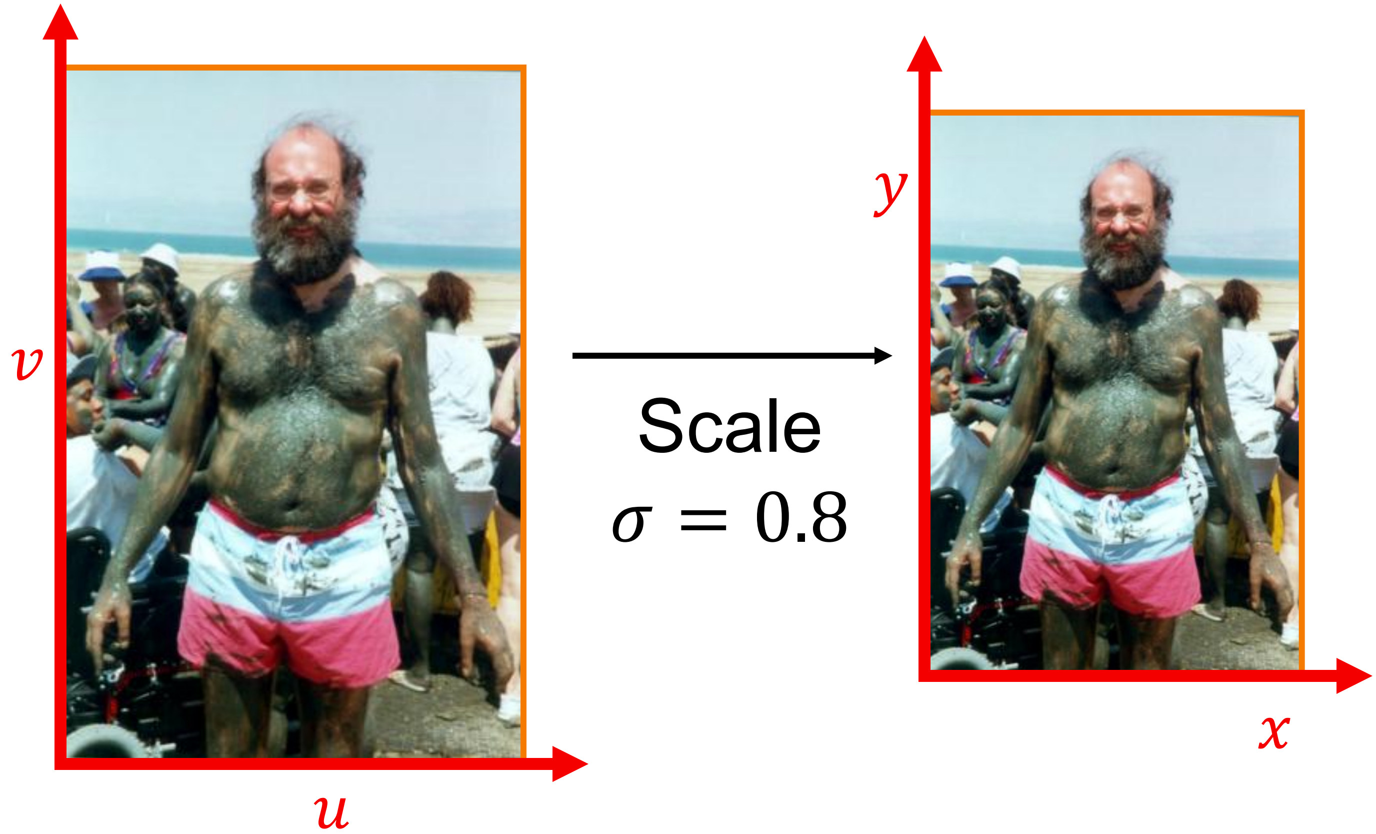
Mapping

- Define transformation
  - Describe the destination $(x, y) = \Phi(u, v)$ for every location (u, v) in the source



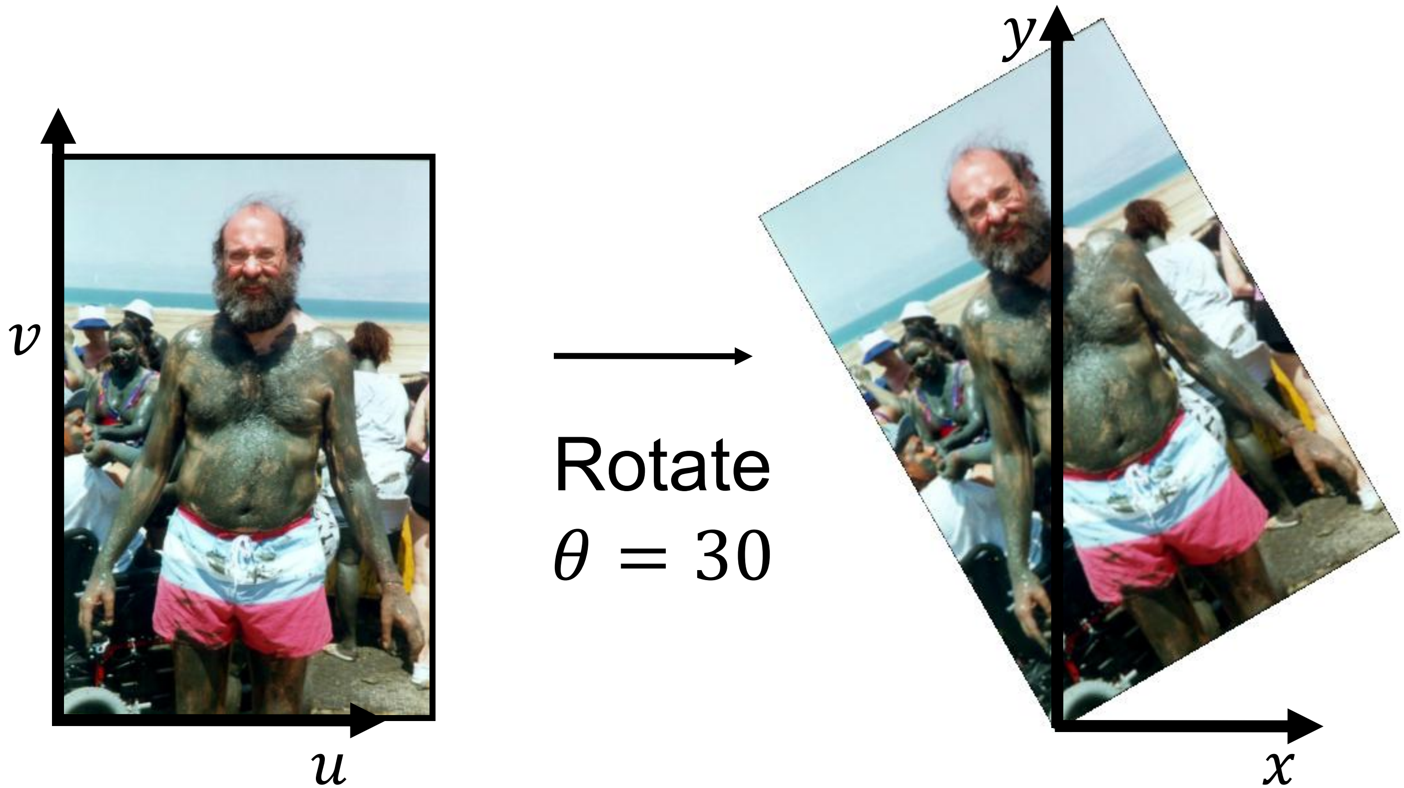
Example Mappings

- Scale by σ:
  - $\Phi(u, v) = (\sigma u, \sigma v)$



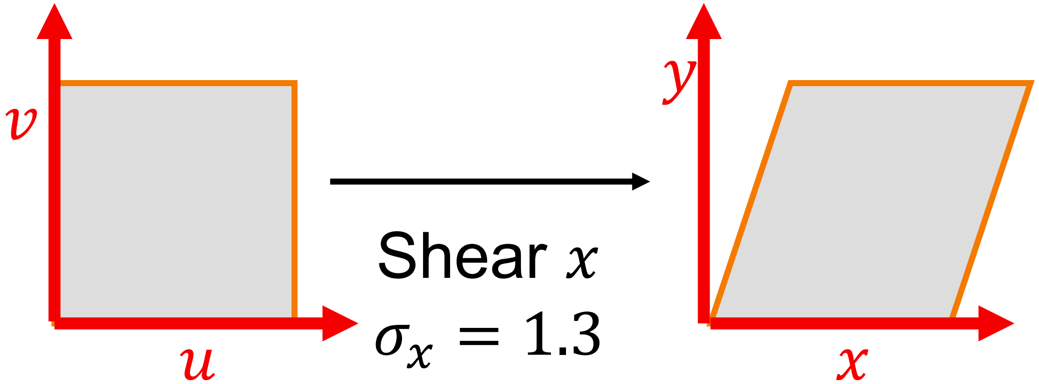
Example Mappings

- Rotate by θ degrees:
  - $\Phi(u, v) = (u\cos\theta - v\sin\theta, u\sin\theta + v\cos\theta)$

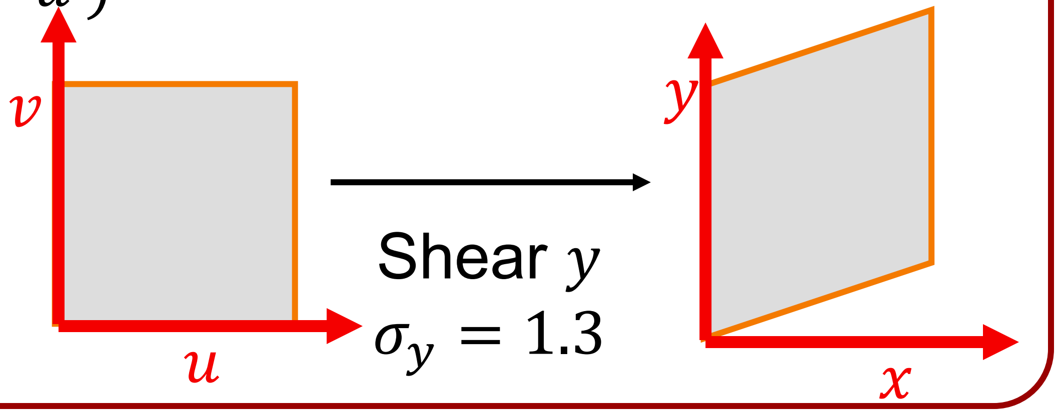


Example Mappings

- Shear in x by σ_x:
  - $\Phi(u,v) = (u + \sigma_x \cdot v, v)$

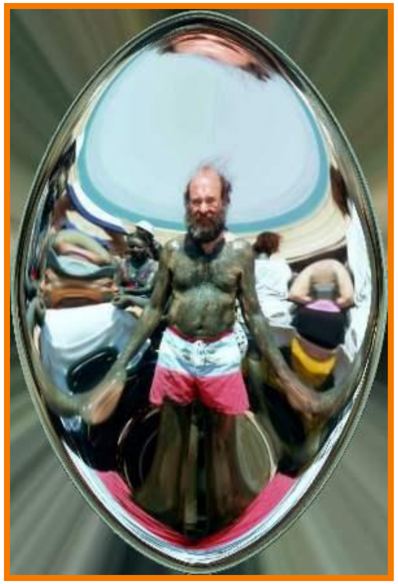


- Shear in y by σ_y:
  - $\Phi(u,v) = (u, v + \sigma_y \cdot u\ )$



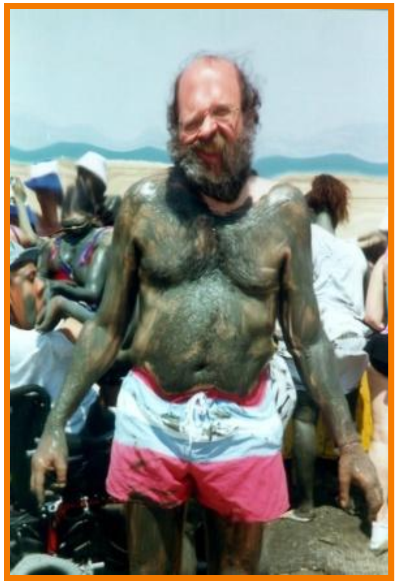
Other Mappings

- Any function of u and v:
  - $\Phi(u, v) = \cdots$



Fish-eye

"Swirl"

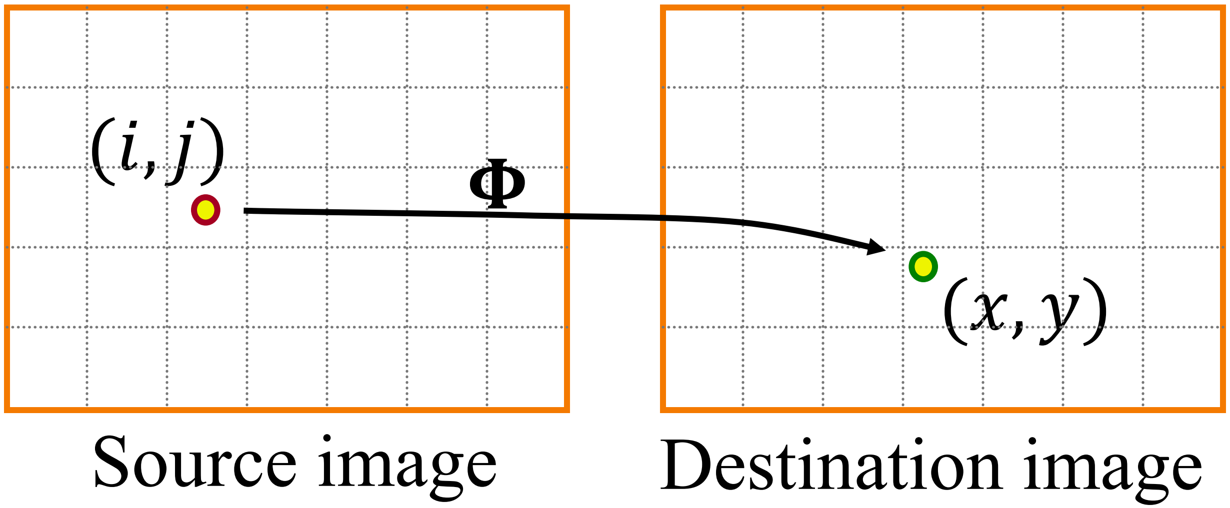


"Rain"

Image Warping Implementation I

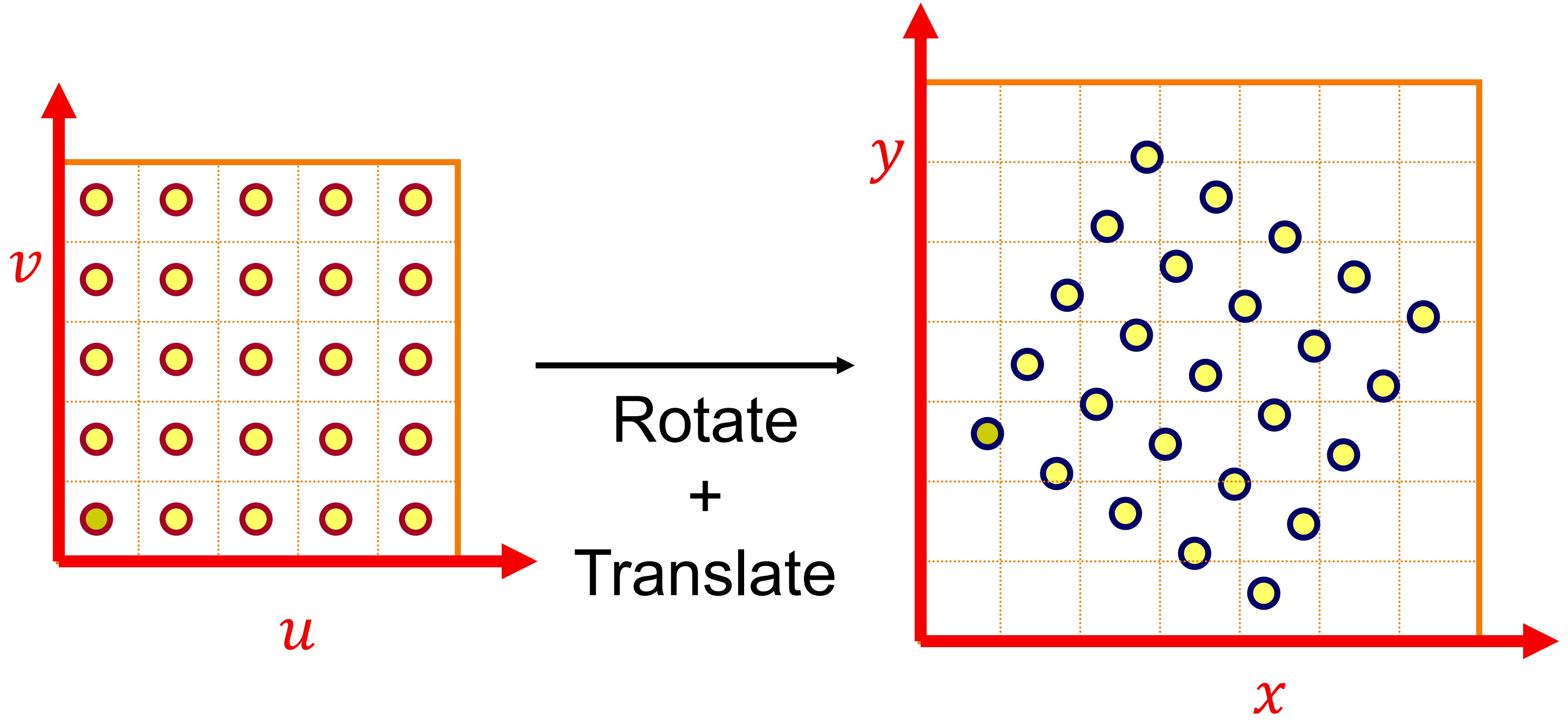
- Forward mapping:

```
for( j=0 ; j<srcHeight ; j++ )
  for( i=0 ; i<srcWidth ; i++ )
    (x,y) = Φ(i,j);
    dst(x,y) = src(i,j);
```



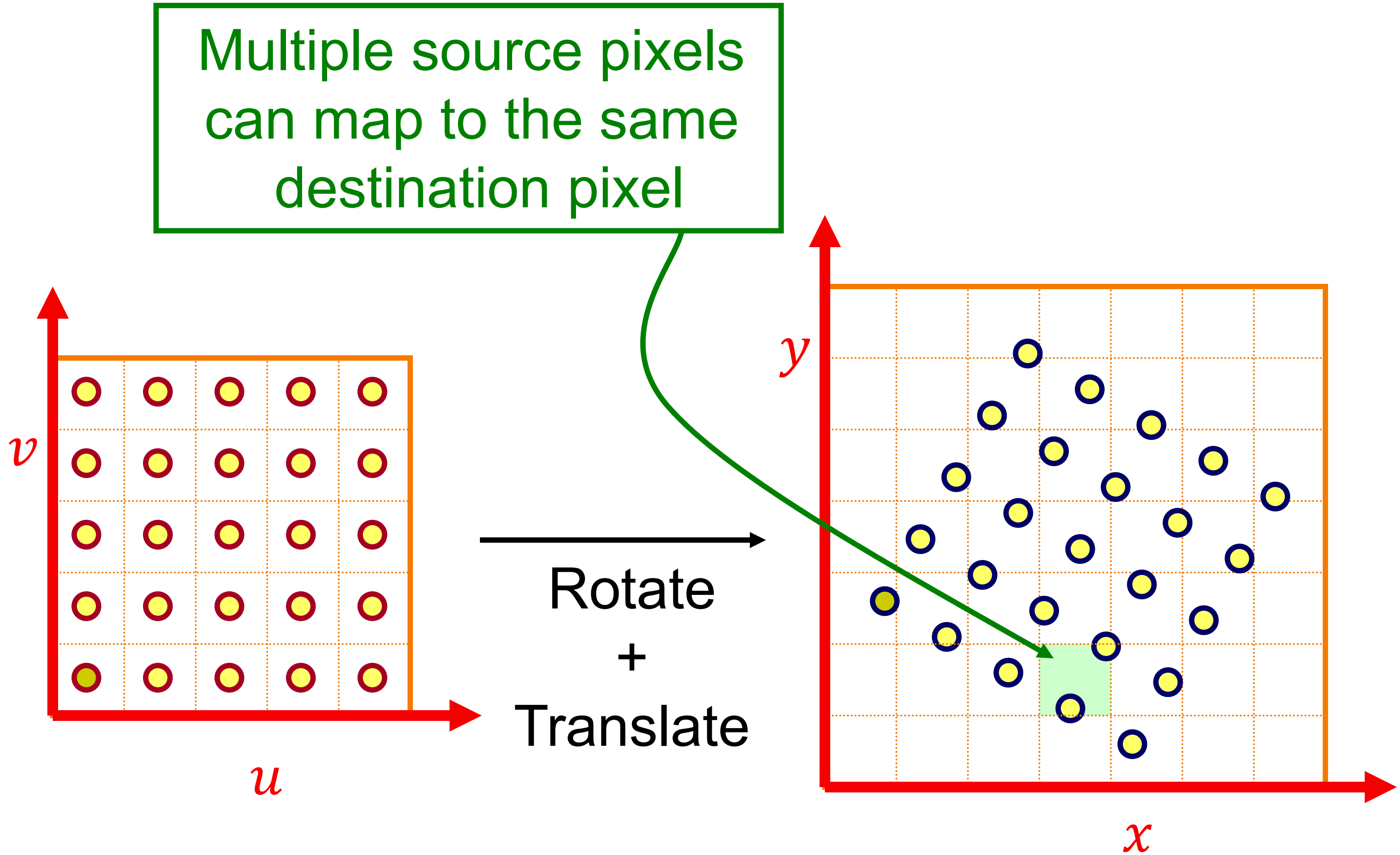
Forward Mapping

- Iterate over source image



Forward Mapping – BAD!

- Iterate over source image



Forward Mapping – BAD!

- Iterate over source image

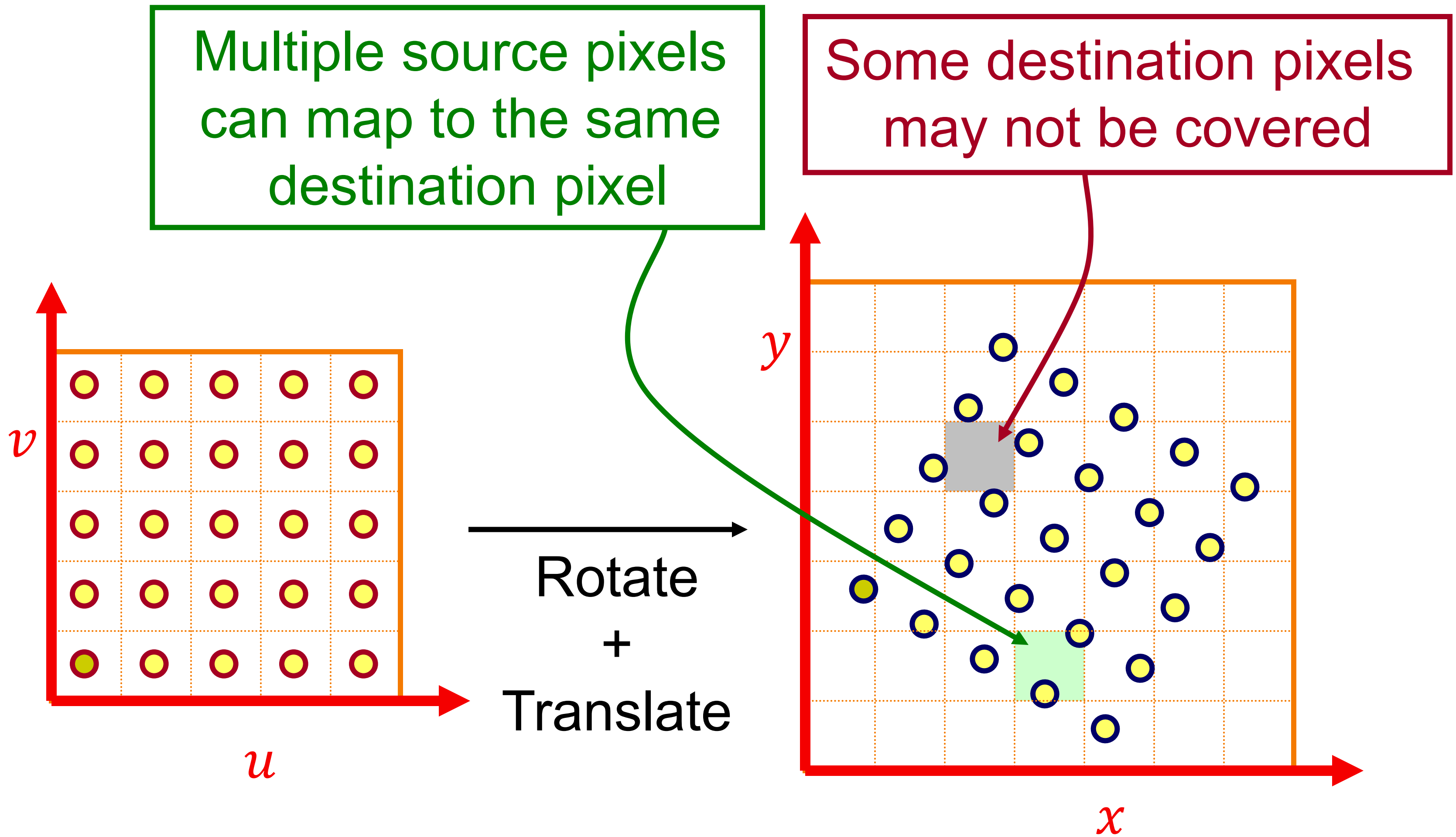


Image Warping Implementation II

- Inverse mapping:

```
for( j=0 ; j<dstHeight ; j++ )
  for( i=0 ; i<dstWidth ; i++ )
    (u,v) = Φ⁻¹(i,j);
    dst(i,j) = src(u,v);
```

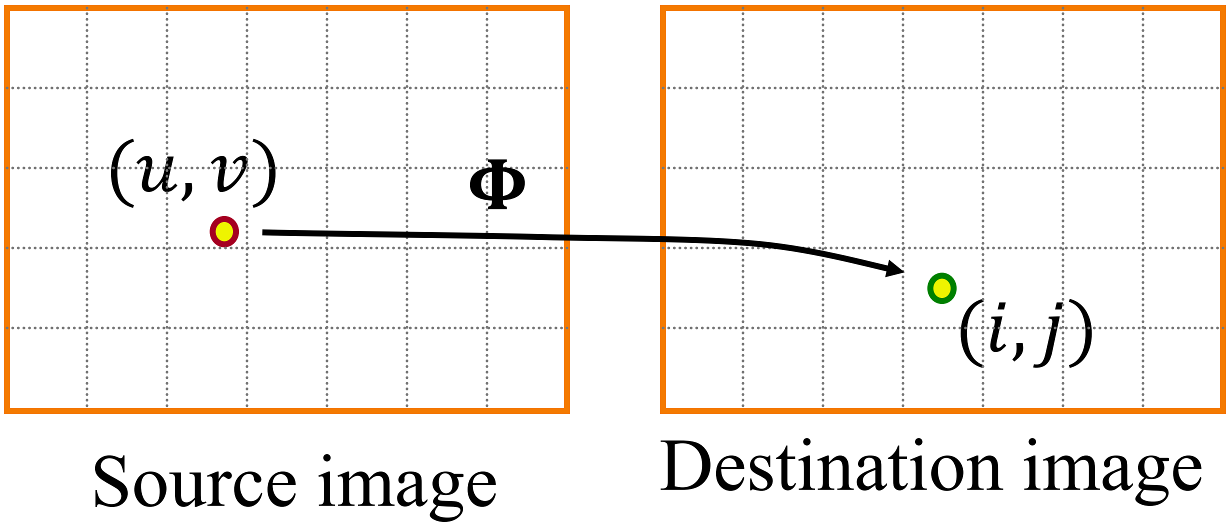
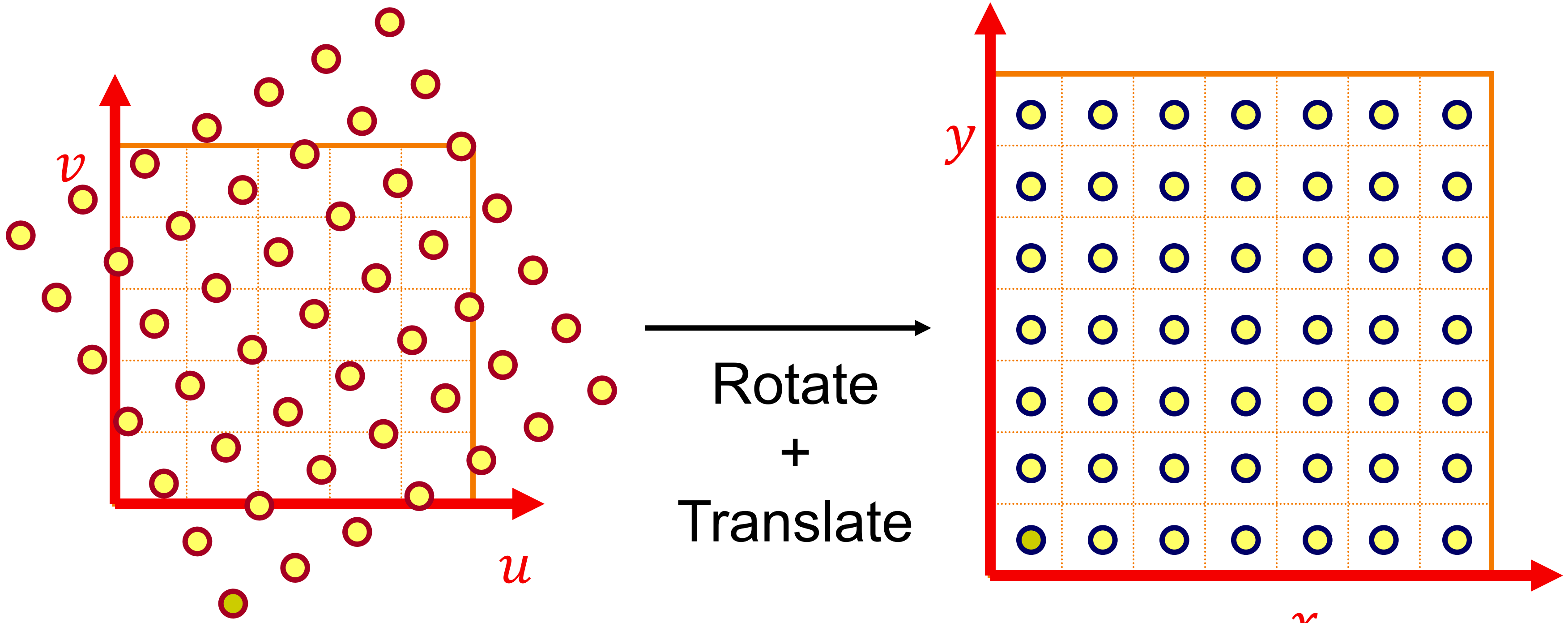


Image Warping Implementation II

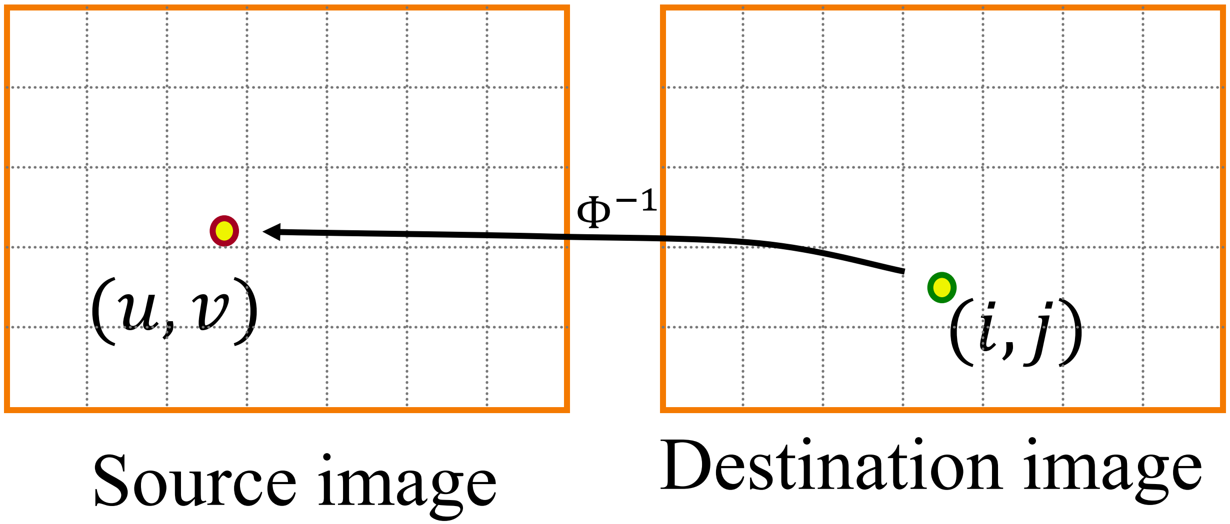
- Inverse mapping:
  - ✓ A single value assigned to each destination pixel
  - ✗ Must resample source



Resampling

- Evaluate source image at $(u, v) = \Phi^{-1}(i, j)$

(u, v) does not usually have integer coordinates



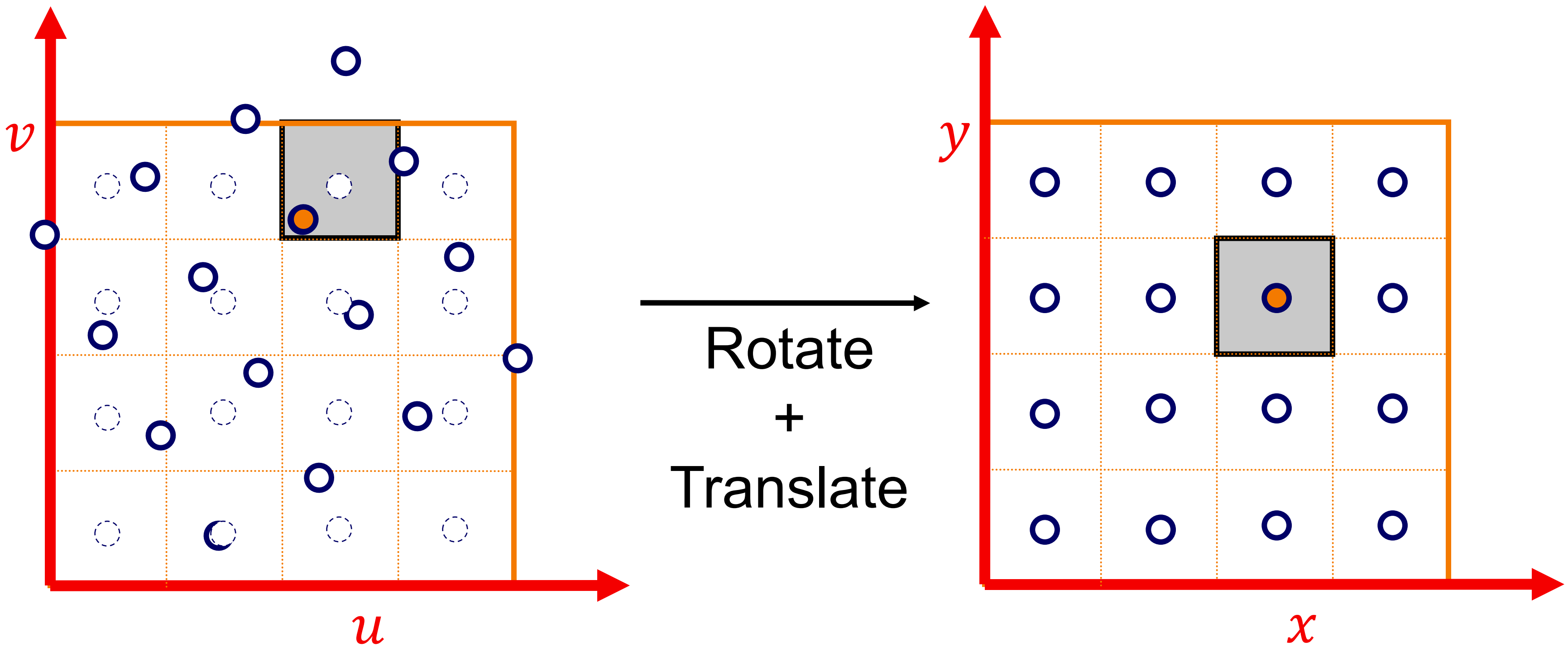
Overview

- Mapping
  - Forward
  - Inverse
- Resampling
  - Nearest Point Sampling
  - Bilinear Sampling
  - Gaussian Sampling

Nearest Point Sampling

- Take value at closest pixel (indexed by center):

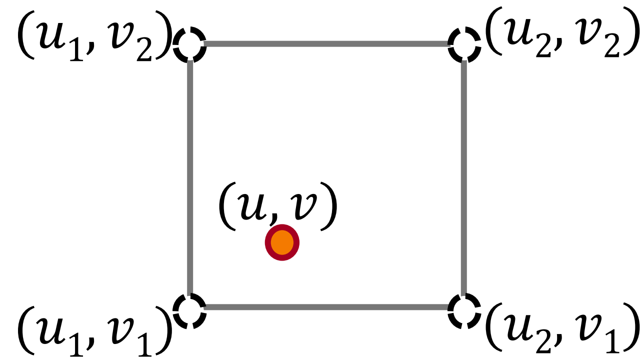
```
int intU = floor(u+0.5);
int intV = floor(v+0.5);
dst(i,j) = src(intU,intV);
```

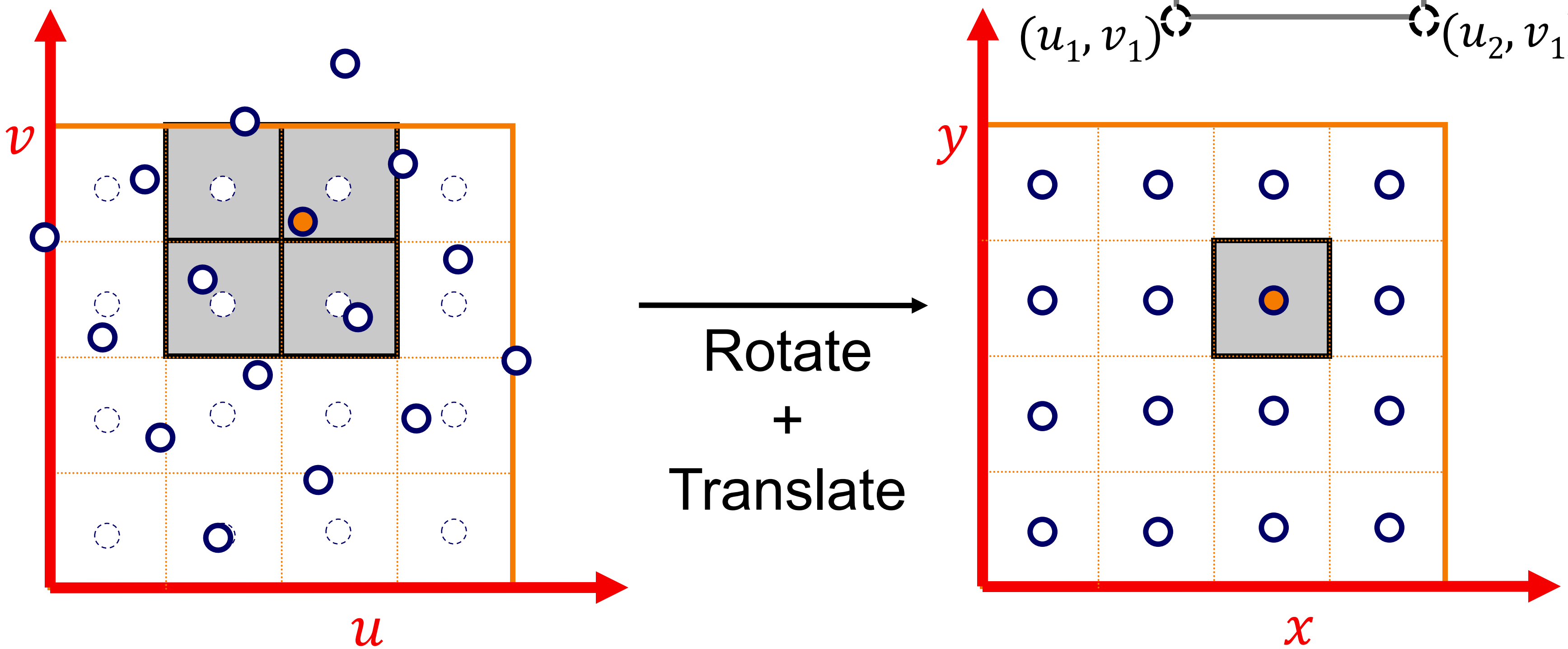


Bilinear Sampling

- Bilinearly interpolate four closest source pixels

$\mathrm{dst}(i,j)$ = Weighted average of source at (u_1, v_1), (u_2, v_1), (u_1, v_2), and (u_2, v_2)

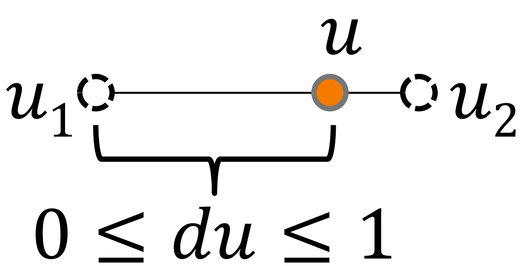




Linear Sampling

- Linearly interpolate two closest source pixels

  $\mathrm{dst}(i)$ = linear interpolation of u_1 and u_2




```
Given i, the pixel location in the target:
    u = Φ^-1(i)
    u1 = floor(u);
    u2 = u1 + 1;
    du = u - u1;
    dst(i) = src(u1)*(1-du) + src(u2)*du;
```

Bilinear Sampling

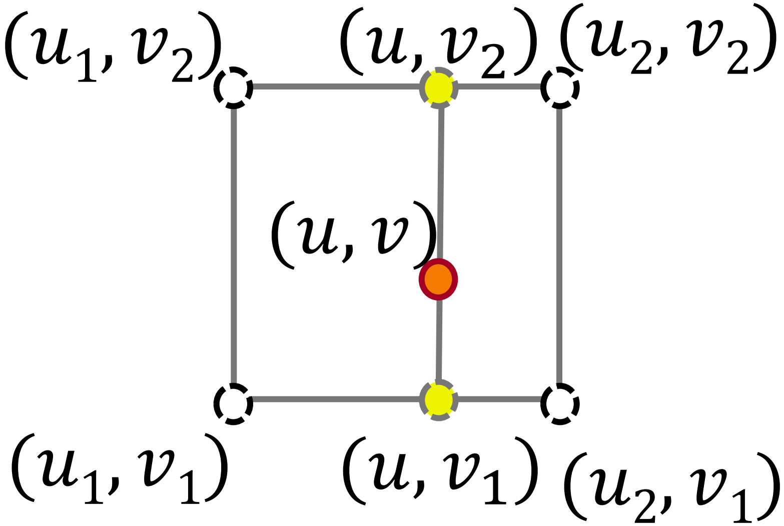
- Bilinearly interpolate four closest source pixels

  a = linear interpolation of $\mathrm{src}(u_1, v_1)$ and $\mathrm{src}(u_2, v_1)$

  b = linear interpolation of $\mathrm{src}(u_1, v_2)$ and $\mathrm{src}(u_2, v_2)$

  $\mathrm{dst}(i, j)$ = linear interpolation of a and b

```
(u,v) = Φ^{-1}(i,j)

u1 = floor( u ) , u2 = u1 + 1;
v1 = floor( v ) , v2 = v1 + 1;

du = u - u1;

a = src(u1,v1)*(1-du)
  + src(u2,v1)*(  du);

b = src(u1,v2)*(1-du)
  + src(u2,v2)*du;

dv = v - v1;
dst(i,j) = a*(1-dv) + b*dv;
```



Bilinear Sampling

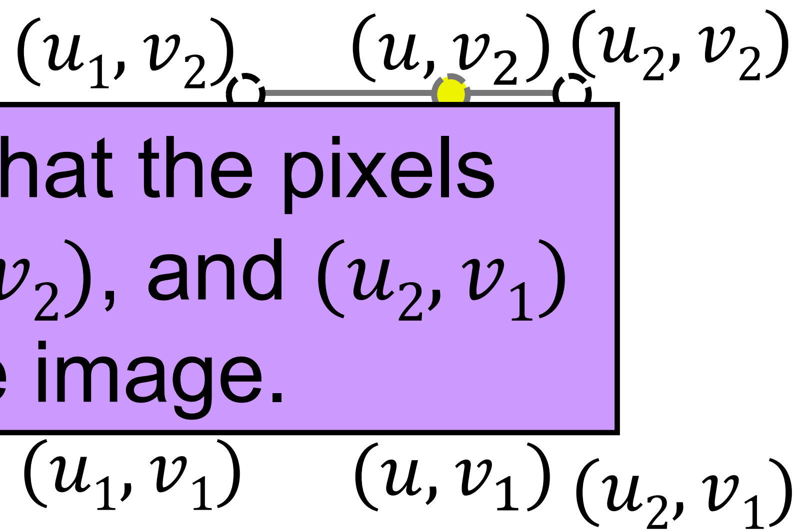
- Bilinearly interpolate four closest source pixels

  a = linear interpolation of $\mathrm{src}(u_1, v_1)$ and $\mathrm{src}(u_2, v_1)$

  b = linear interpolation of $\mathrm{src}(u_1, v_2)$ and $\mathrm{src}(u_2, v_2)$

  $\mathrm{dst}(i, j)$ = linear interpolation of a and b

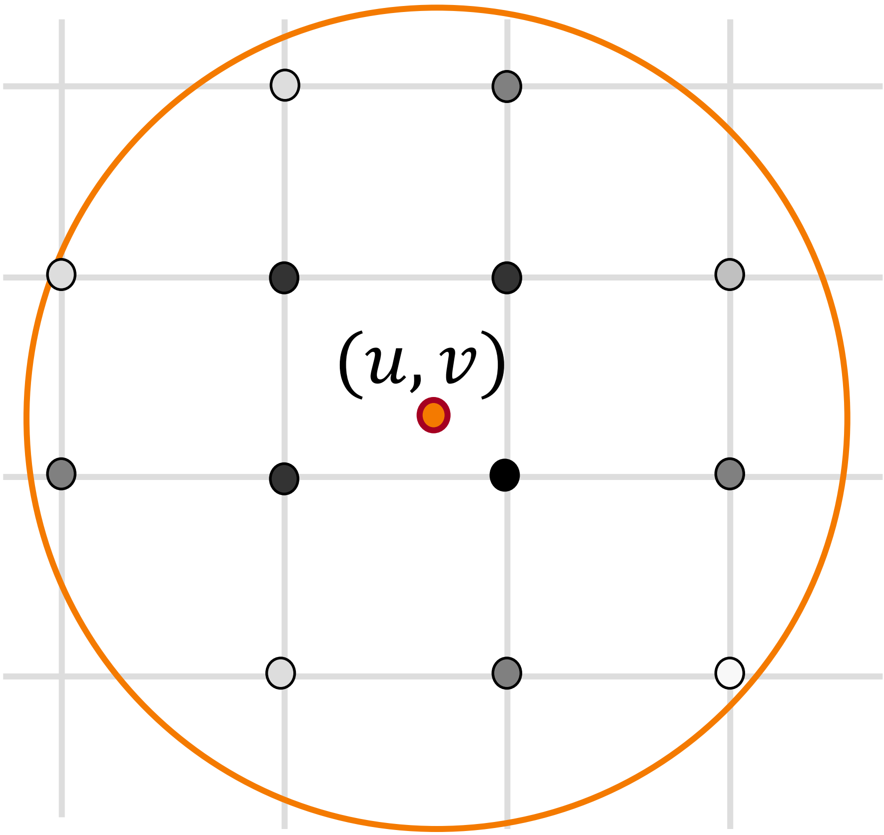
```
(u,v) = Φ^-1(i,j)

u1 =
v1 =

du =

a =
  + src(u2,v1)*(  du);

b = src(u1,v2)*(1-du)
  + src(u2,v2)*du;

dv = v - v1;
dst(i,j) = a*(1-dv) + b*dv;
```



Make sure to test that the pixels (u_1, v_1), (u_2, v_2), (u_1, v_2), and (u_2, v_1) are within the image.

Gaussian Sampling

- Compute weighted sum of pixel neighborhood:
  - The blending weights are the normalized values of a Gaussian function.



Note:

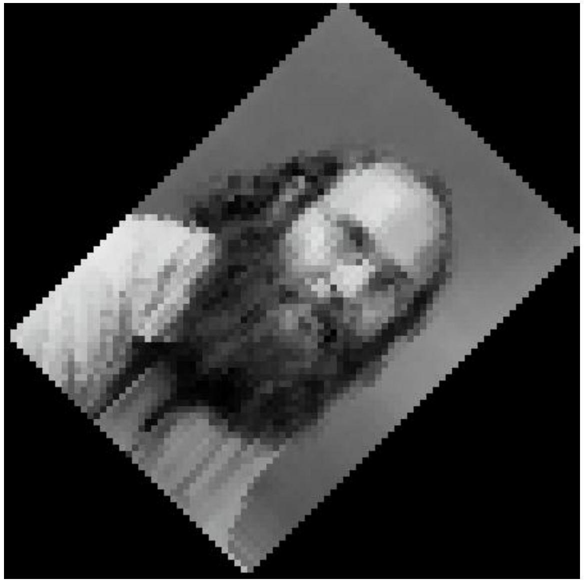
In contrast to the blurring filter, this doesn't assume that the center is at an integer location.

⇒ Can't precompute a stencil in advance

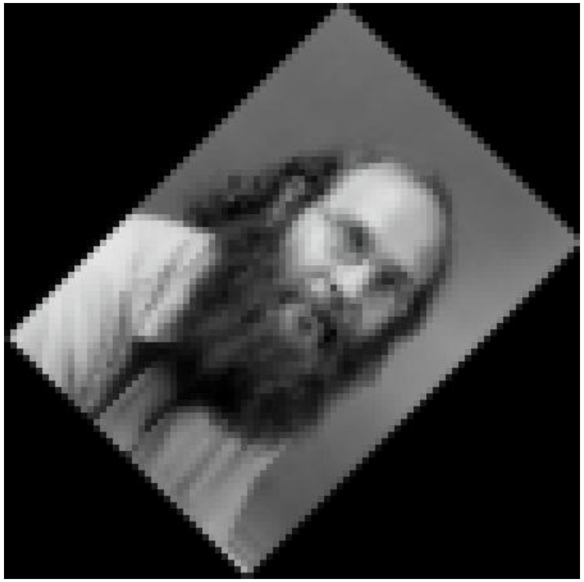
Filtering Methods Comparison

- Trade-offs
  - Jagged edges versus blurring
  - Computation speed

We'll talk more about trade-offs next time.



Nearest



Bilinear

Gaussian

Image Warping Implementation

- Inverse mapping:

```
for( j=0 ; j<dstHeight ; j++ )
  for( i=0 ; i<dstWidth ; i++ )
    (u,v) = Φ⁻¹(i,j);
    dst(i,j) = resample_src(u,v,r);
```

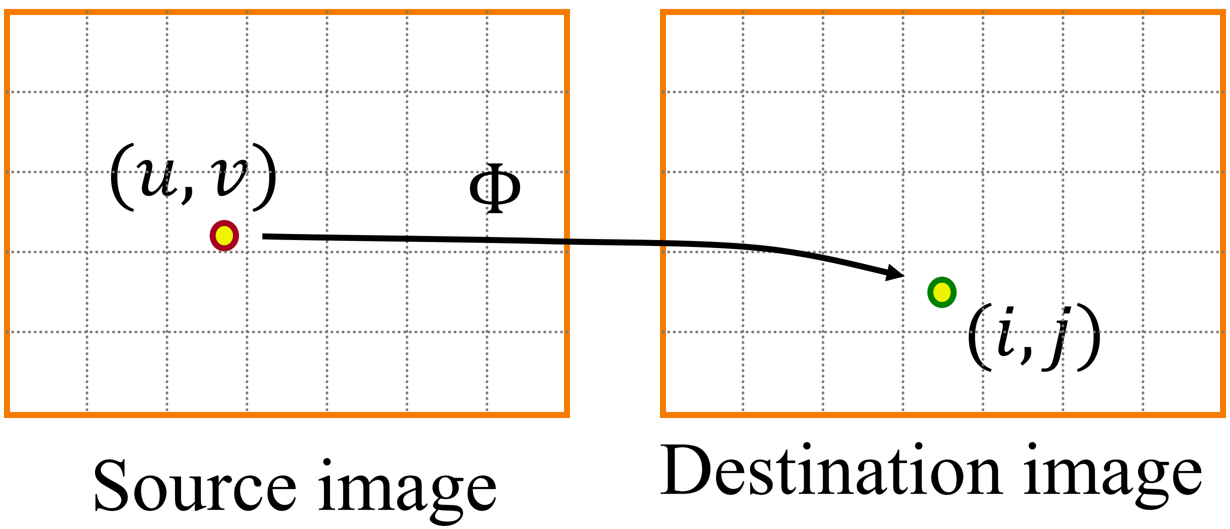
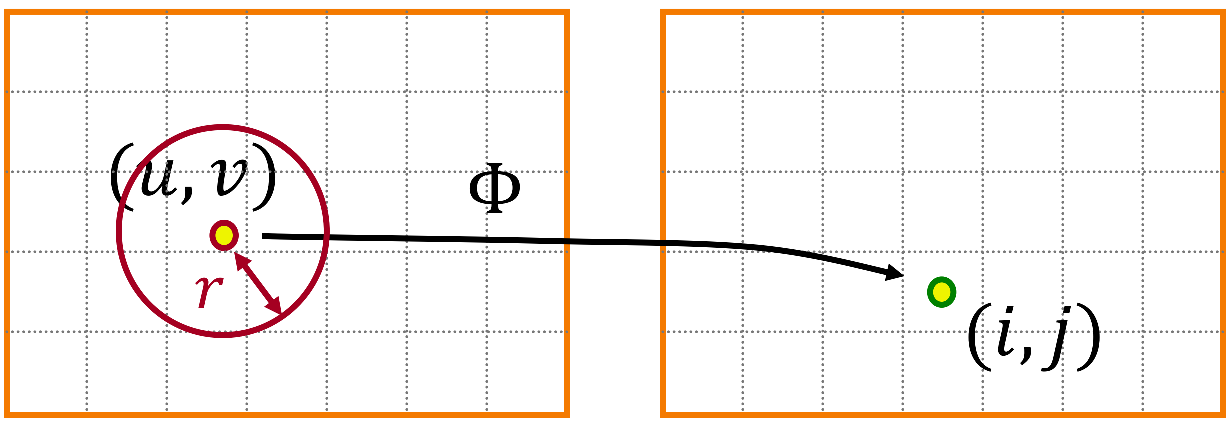


Image Warping Implementation

- Inverse mapping:

```
for( j=0 ; j<dstHeight ; j++ )
  for( i=0 ; i<dstWidth ; i++ )
    (u,v) = Φ^-1(i,j);
    dst(i,j) = resample_src(u,v,r);
```



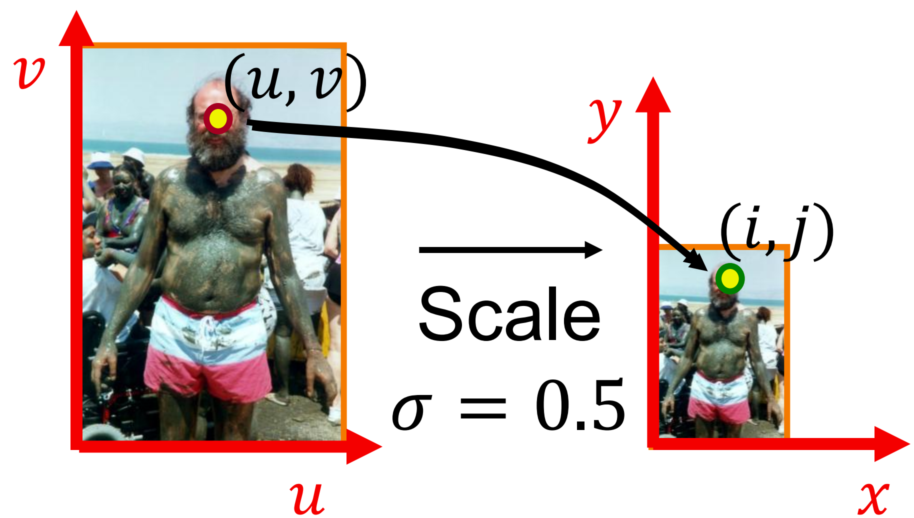
Source image Destination image

Example: Scale

Scale(src , dst , σ):

```
r ≅ ?;
for( j=0 ; j<dstHeight ; j++ )
  for( i=0 ; i<dstWidth ; i++ )
    (u,v) = (i,j) / σ;
    dst(i,j) = resample_src(u,v,r);
```

$$r = \frac{1}{\sigma}$$

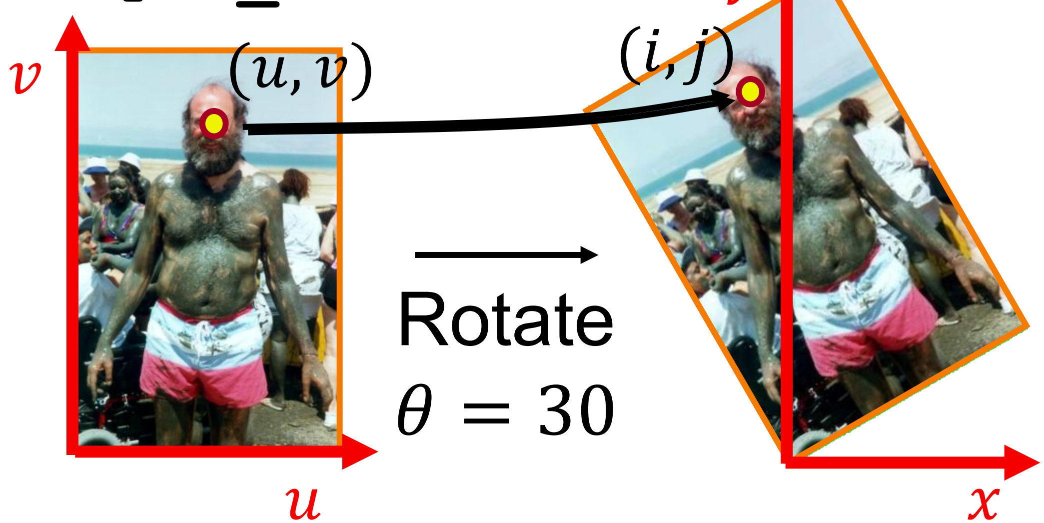


Example: Rotate

Rotate(src , dst , θ):

```
r ≅ ?;
for( j=0 ; j<dstHeight ; j++ )
   for( i=0 ; i<dstWidth ; i++ )
      (u,v) = ( i*cos(-θ) - j*sin(-θ) ,
                i*sin(-θ) + j*cos(-θ) );
      dst(x,y) = resample_src(u,v,r);
```

$r = 1$

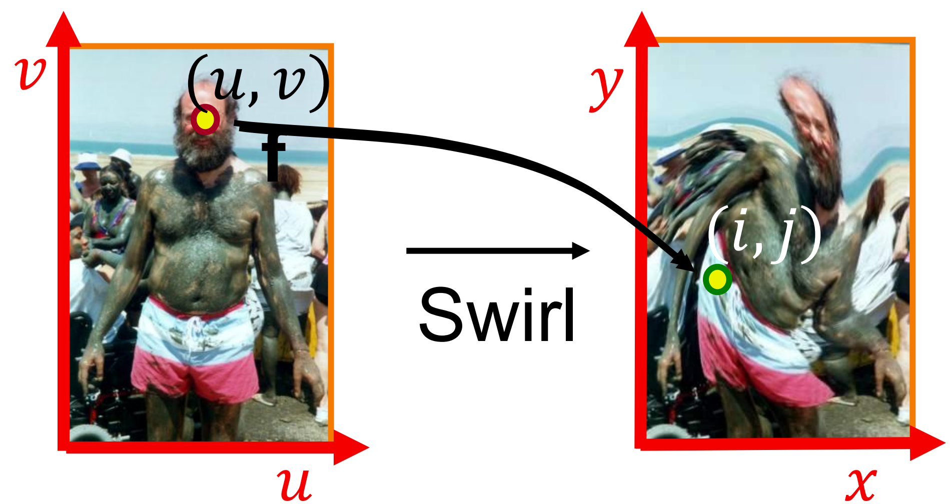


Example:

<u>General(src , dst , Φ):</u>

```
r ≅ ?;
for( j=0 ; j<dstHeight ; j++ )
   for( i=0 ; i<dstWidth ; i++ )
      (u,v) = Φ^-1(i,j);
       dst(i,j) = resample_src(u,v,r);
```

$r = ?$



Example:

General(src , dst , Φ):

```
r ≅ ?;
for( j=0 ; j<dstHeight ; j++ )
   for( i=0 ; i<dstWidth ; i++ )
      (u,v) = Φ^-1(i,j);
       dst(i,j) = resample_src(u,v,r);
```

Instead of using a fixed radius circle to sample the source, we can:

1. Have the radius change,
2. Use an ellipse.

For example, the parameters can be determined by looking at the derivative/Jacobean of Φ.

Outline

- Image Filtering
- Image Warping
- Image Morphing

Image Morphing

- Animate transition between two images

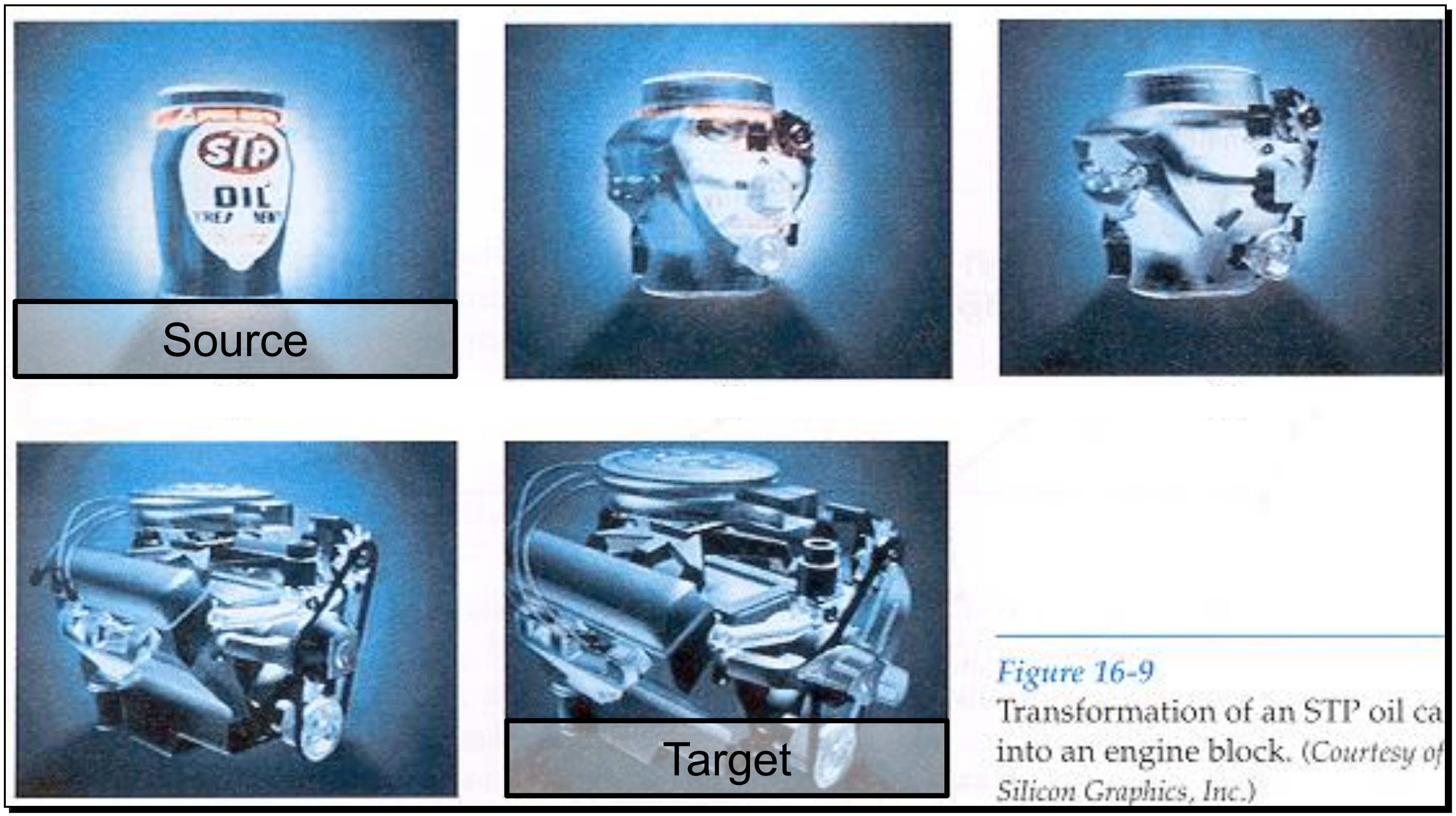
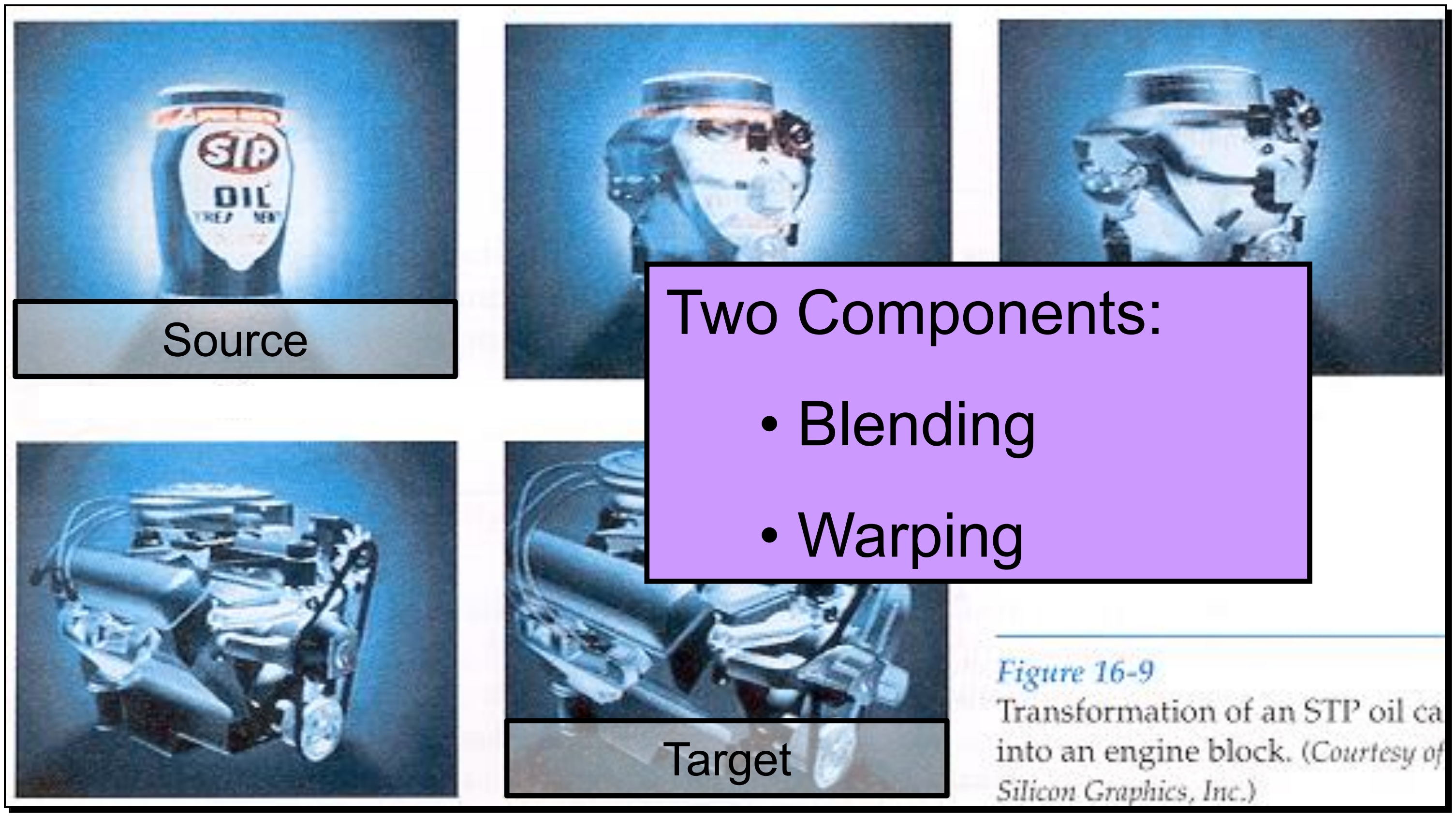


Figure 16-9
Transformation of an STP oil ca
into an engine block. (*Courtesy of*
Silicon Graphics, Inc.)

H&B Figure 16.9

Image Morphing

- Animate transition between two images



Two Components:

- Blending
- Warping

H&B Figure 16.9

Image Morphing

Recall (Inner Product):

1. For two vectors $\vec{v}_1 = (x_1, y_1)$ and $\vec{v}_2 = (x_2, y_2)$, the inner product between $\vec{v}_1$ and $\vec{v}_2$ is:
$$\langle \vec{v}_1, \vec{v}_2 \rangle \equiv x_1 \cdot x_2 + y_1 \cdot y_2 = \|\vec{v}_1\| \cdot \|\vec{v}_2\| \cdot \cos\theta$$

⇒ Gives the cosine of the angle between the two, scaled by the product of their lengths.

⇒ Dividing by the length of v_2 gives the signed length of the projection of v_1 onto the line through v_2:

$$\|v_1\| \cdot \cos\theta = \frac{\langle v_1, v_2 \rangle}{\|v_2\|}$$

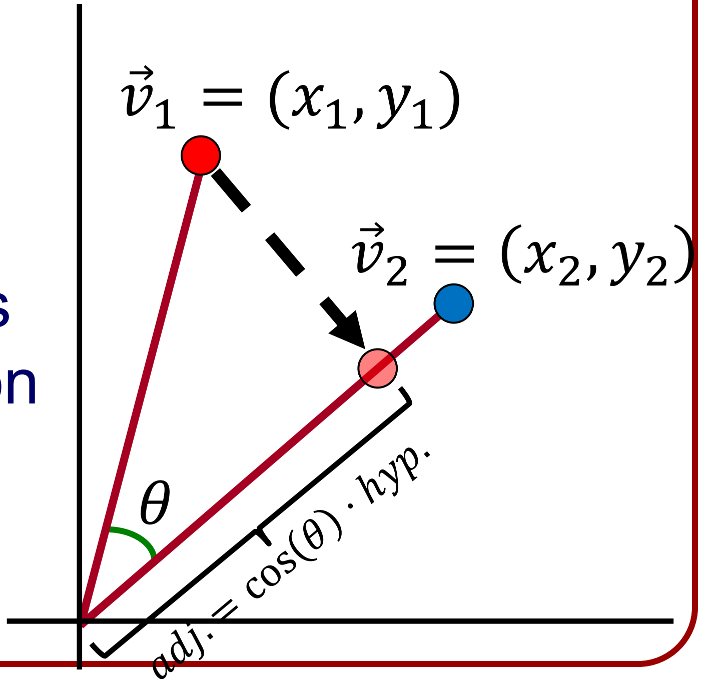


Image Morphing

Recall (Inner Product):

1. For two vectors $\vec{v}_1 = (x_1, y_1)$ and $\vec{v}_2 = (x_2, y_2)$, the inner product between $\vec{v}_1$ and $\vec{v}_2$ is:
$$\langle \vec{v}_1, \vec{v}_2 \rangle \equiv x_1 \cdot x_2 + y_1 \cdot y_2 = \|\vec{v}_1\| \cdot \|\vec{v}_2\| \cdot \cos\theta$$

2. For a vector $\vec{v} = (x, y)$, its $90°$ (CCW) rotation is:
$$\vec{v}^{\perp} = (-y, x)$$
This is the vector:
    - With the same length as $\vec{v}$, and
    - Which is perpendicular to $\vec{v}$:
$$\langle \vec{v}, \vec{v}^{\perp} \rangle = 0$$

In 2D, the perpendicular is unique up to sign (disambiguated with CCW vs CW).

In higher dimensions, there is a continuum of perpendicular vectors.

Image Morphing

Recall (Inner Product):

1. For two vectors $\vec{v}_1 = (x_1, y_1)$ and $\vec{v}_2 = (x_2, y_2)$, the inner product between $\vec{v}_1$ and $\vec{v}_2$ is:
$$\langle \vec{v}_1, \vec{v}_2 \rangle \equiv x_1 \cdot x_2 + y_1 \cdot y_2 = \|\vec{v}_1\| \cdot \|\vec{v}_2\| \cdot \cos\theta$$

2. For a vector $\vec{v} = (x, y)$, its $90°$ (CCW) rotation is:
$$\vec{v}^{\perp} = (-y, x)$$

3. The inner-product of two vectors is *isometry-invariant*: If $\mathbf{R} \in \mathbb{R}^{2\times 2}$ is a rotation/reflection, then:
$$\langle \mathbf{R}(\vec{v}_1), \mathbf{R}(\vec{v}_2) \rangle = \langle \vec{v}_1, \vec{v}_2 \rangle$$

⇒ In particular:
$$\left\langle \vec{v}_1^{\perp}, \vec{v}_2^{\perp} \right\rangle = \langle \vec{v}_1, \vec{v}_2 \rangle$$

Image Morphing: Blending

Blend **colors** using an α-blend ($\alpha \in [0,1]$):

$$\text{blend}(i,j,\alpha) = (1-\alpha) \cdot \text{Img}_0(i,j) + \alpha \cdot \text{Img}_1(i,j)$$

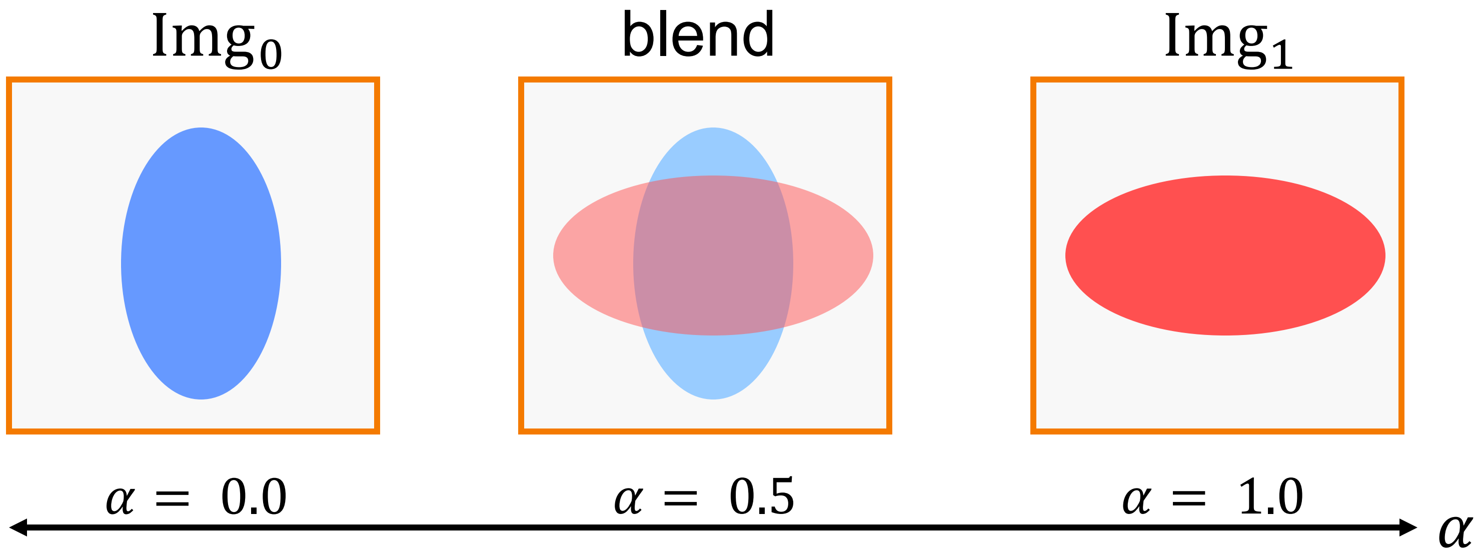


Image Morphing: Warping

Deform Img_0 so its **shape** matches that of Img_1…

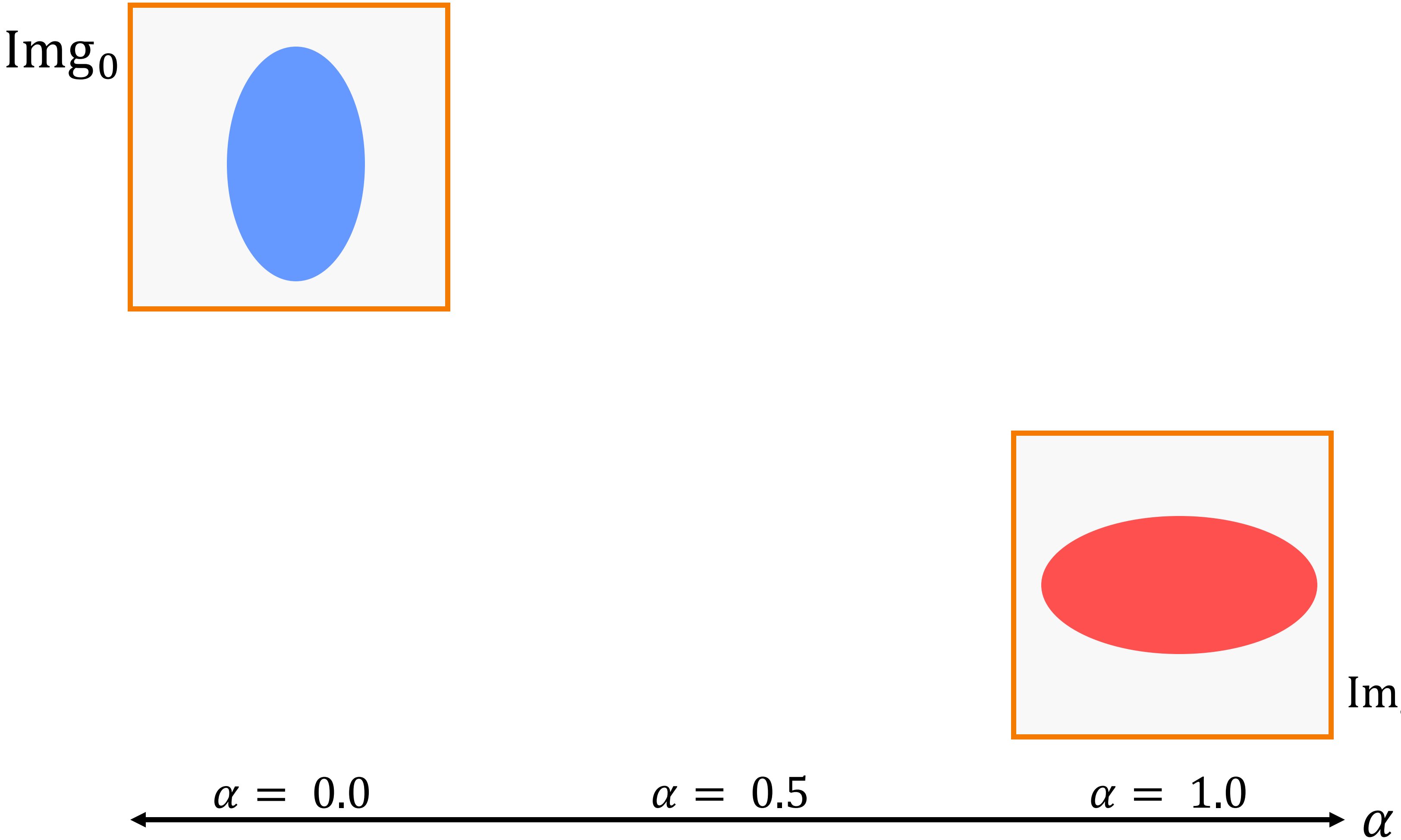


Image Morphing: Warping

Deform Img_0 so its **shape** matches that of Img_1…

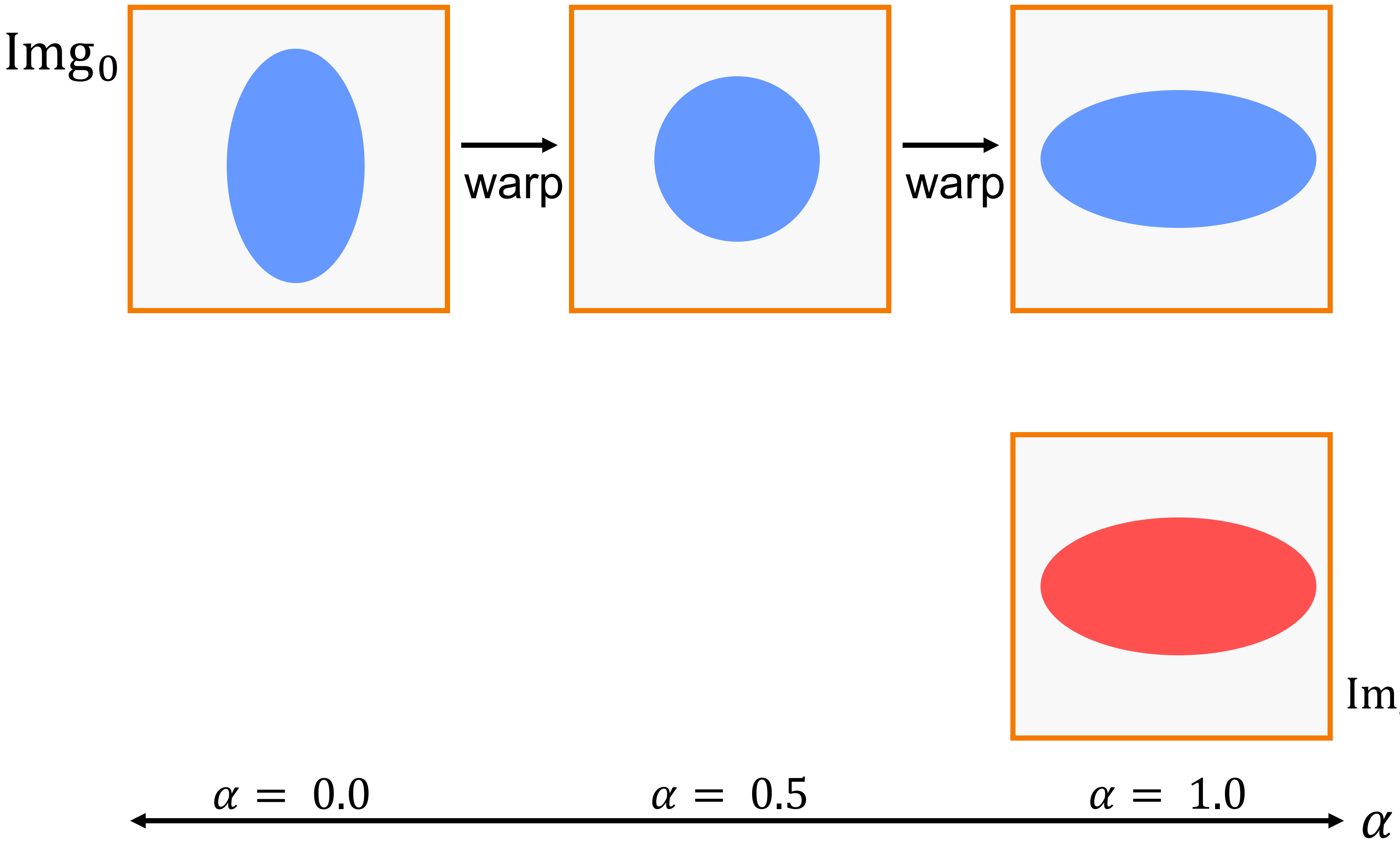


Image Morphing: Warping

Deform Img_1 so its **shape** matches that of Img_0…

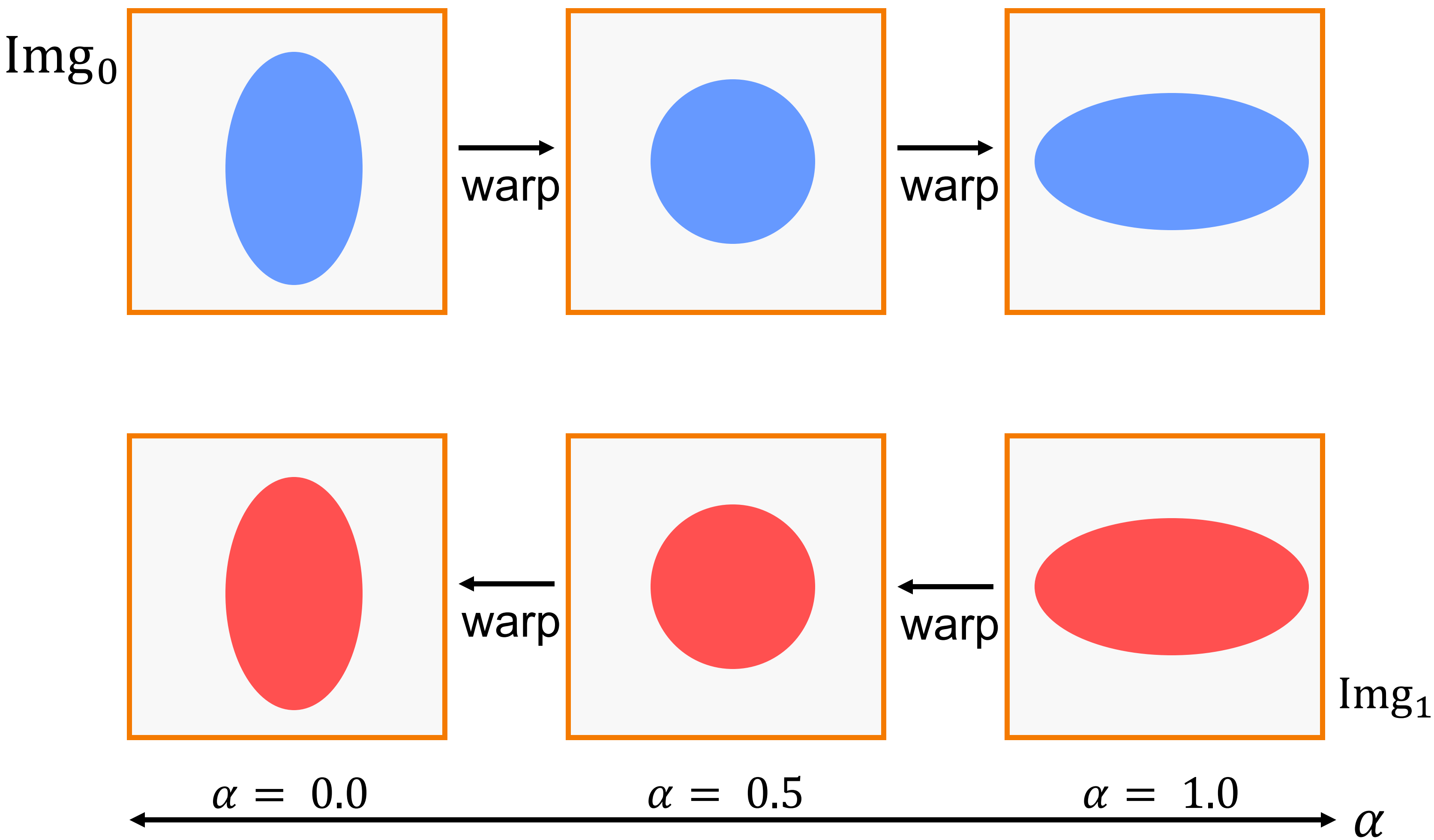


Image Morphing: Warping + Blending

… then blend **colors**

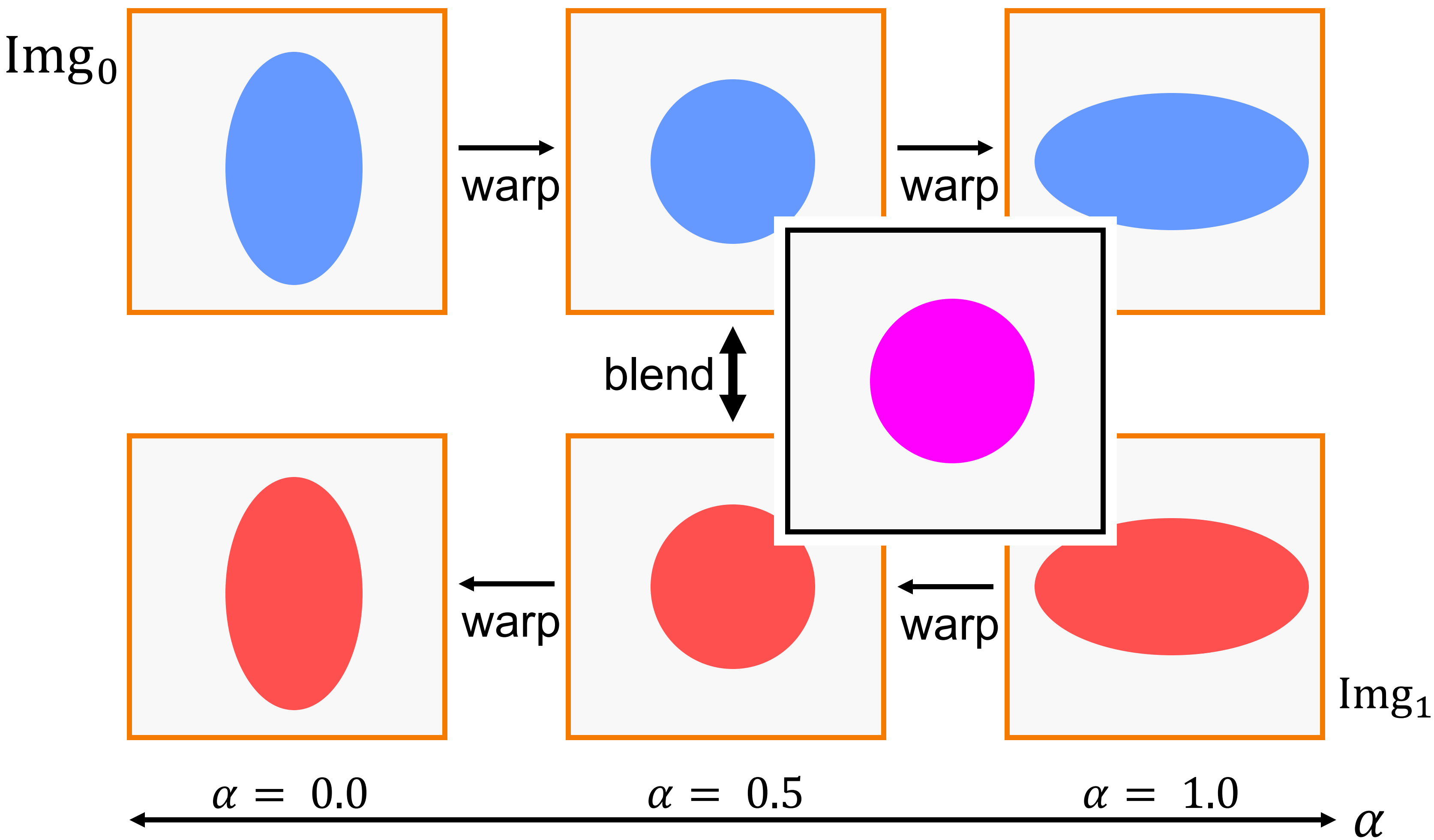
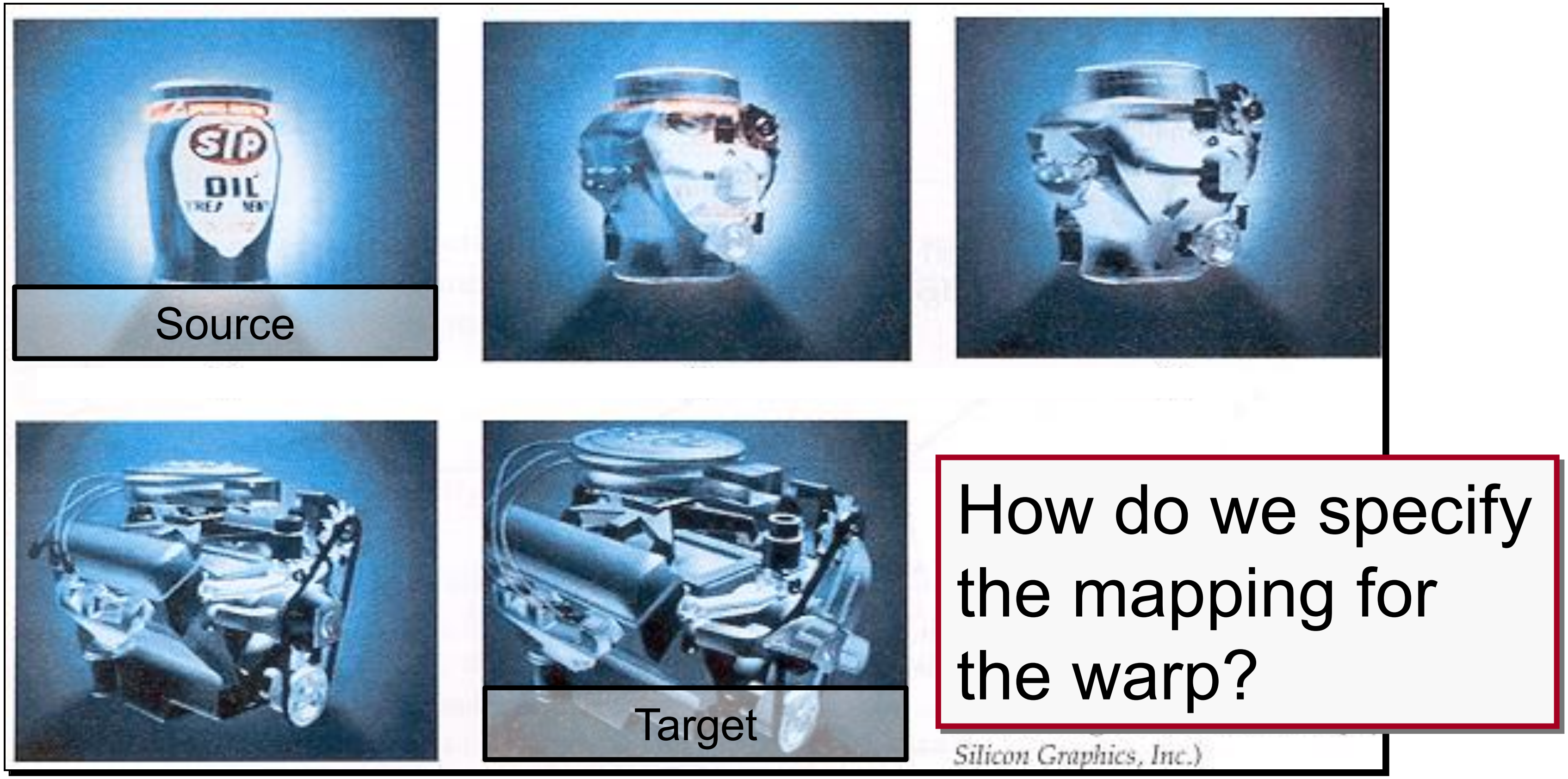


Image Morphing: Warping

- The warping step is the hard one
  - Aim to align features in images



H&B Figure 16.9

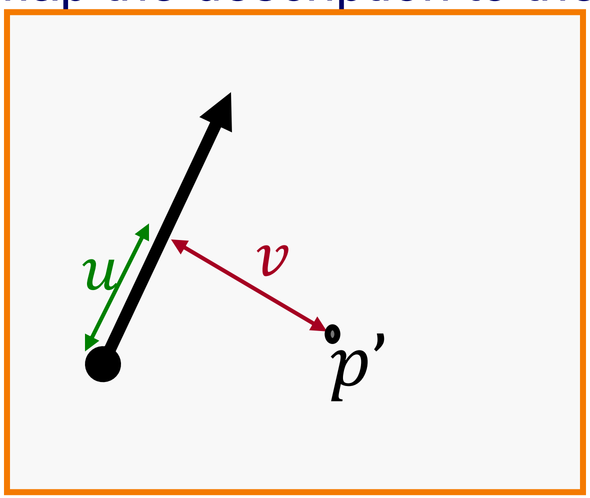
Feature-Based Warping [Beier & Neeley, 1992]

Challenge:

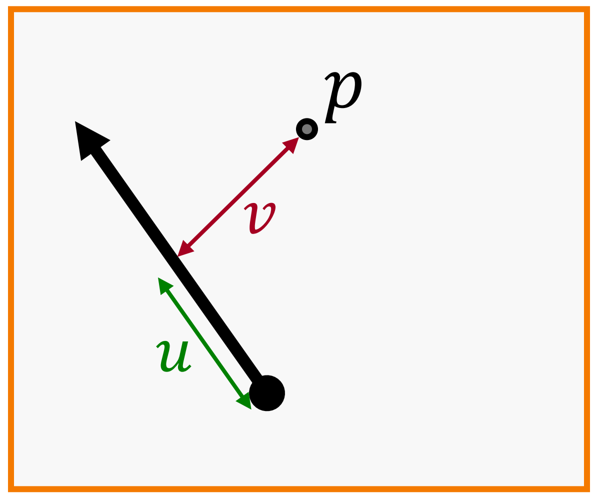
- Given p in the destination, what is the corresponding source location?

Approach:

- Define a pair of corresponding lines in the source and target
- Describe p relative to the destination line
- Map the description to the source



Source image



Destination image

u is a signed fraction
v is a signed length (in pixels)

Feature-Based Warping [Beier & Neeley, 1992]

How do we calculate v (perp. pixel distance)?

Recall:

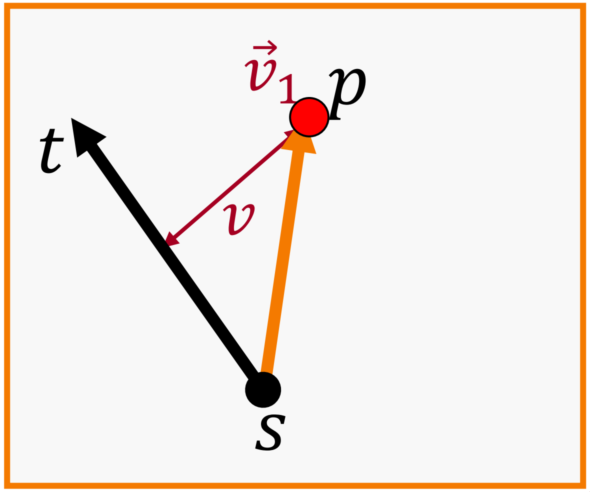
The signed length of $\vec{v}_1$ projected onto $\vec{v}_2$ is:

$$\frac{\langle \vec{v}_1, \vec{v}_2 \rangle}{||\vec{v}_2||}$$

In our case we want:

- The signed length of the direction from the start of the segment to p:

$$\vec{v}_1 = p - s$$



Feature-Based Warping [Beier & Neeley, 1992]

How do we calculate v (perp. pixel distance)?

Recall:

The signed length of $\vec{v}_1$ projected onto $\vec{v}_2$ is:

$$\frac{\langle \vec{v}_1, \vec{v}_2 \rangle}{||\vec{v}_2||}$$

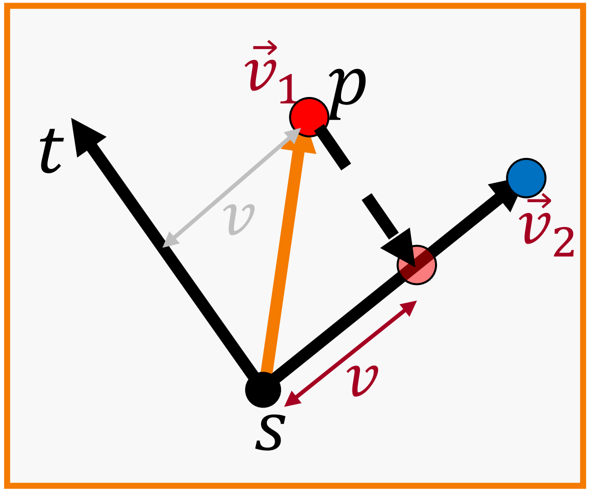
In our case we want:

- The signed length of the direction from the start of the segment to p:

$$\vec{v}_1 = p - s$$

- Projected onto the perpendicular of the direction from the start of the segment to the end:

$$\vec{v}_2 = (t - s)^{\perp}$$



Feature-Based Warping [Beier & Neeley, 1992]

How do we calculate v (perp. pixel distance)?

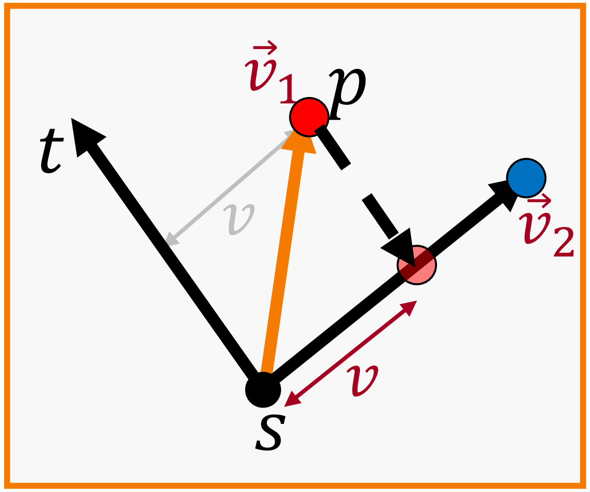
Recall:

The signed length of $\vec{v}_1$ projected onto $\vec{v}_2$ is:

$$\frac{\langle \vec{v}_1, \vec{v}_2 \rangle}{||\vec{v}_2||}$$

In our case we want:

- The signed length of the direction from the start of the segment to p:
  $$\vec{v}_1 = p - s$$
- Projected onto the perpendicular of the direction from the start of the segment to the end:
  $$\vec{v}_2 = (t - s)^\perp$$

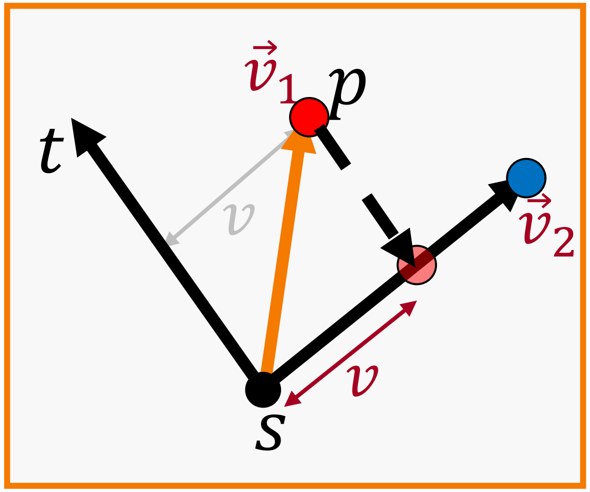


Feature-Based Warping [Beier & Neeley, 1992]

How do we calculate v (perp. pixel distance)?

This gives:

$$v = \frac{\langle \overbrace{p - s}^{\vec{v}_1}, \overbrace{(t-s)^{\perp}}^{\vec{v}_2} \rangle}{\|(t-s)^{\perp}\|}$$



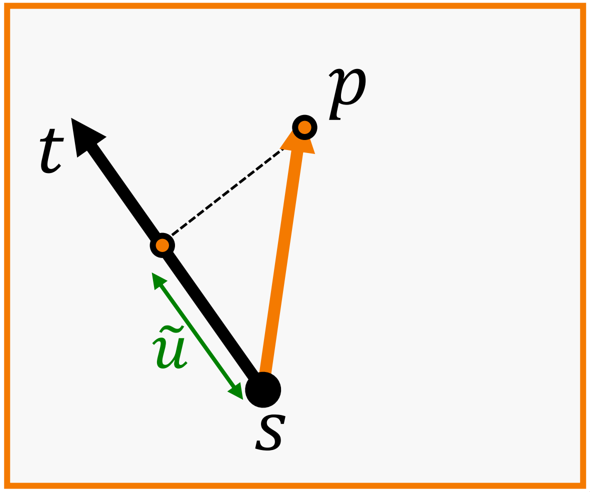
Feature-Based Warping [Beier & Neeley, 1992]

How do we calculate u (par. **fractional** distance)?

Similarly:

- The signed projected length is:

$$\tilde{u} = \frac{\langle \overbrace{p - s}^{\vec{v}_1}, \overbrace{t - s}^{\vec{v}_2} \rangle}{\|t - s\|}$$



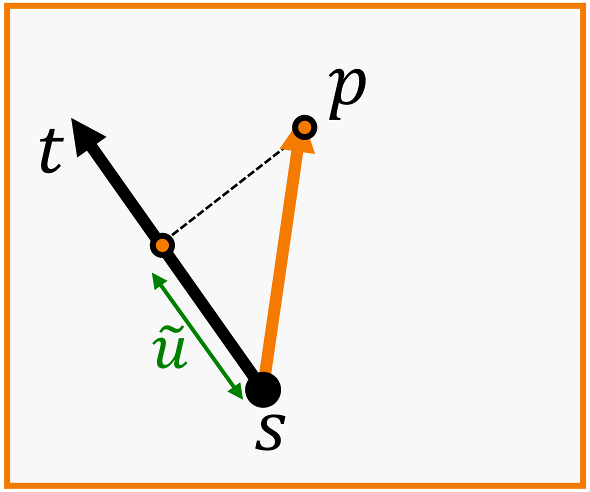
Feature-Based Warping [Beier & Neeley, 1992]

How do we calculate u (par. **fractional** distance)?

Similarly :

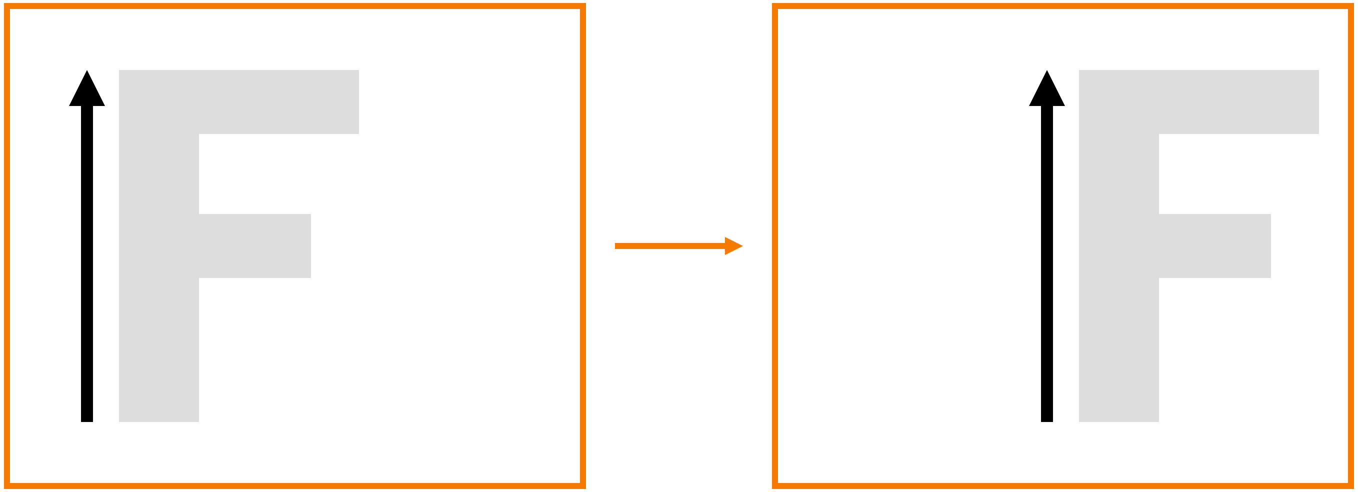
- The **fractional** signed projected length is obtained by dividing by the length of the segment:

$$u = \frac{\langle \overbrace{p-s}^{\vec{v}_1}, \overbrace{t-s}^{\vec{v}_2} \rangle}{\|t-s\|} \cdot \frac{1}{\|t-s\|}$$



Warping with One Line Pair

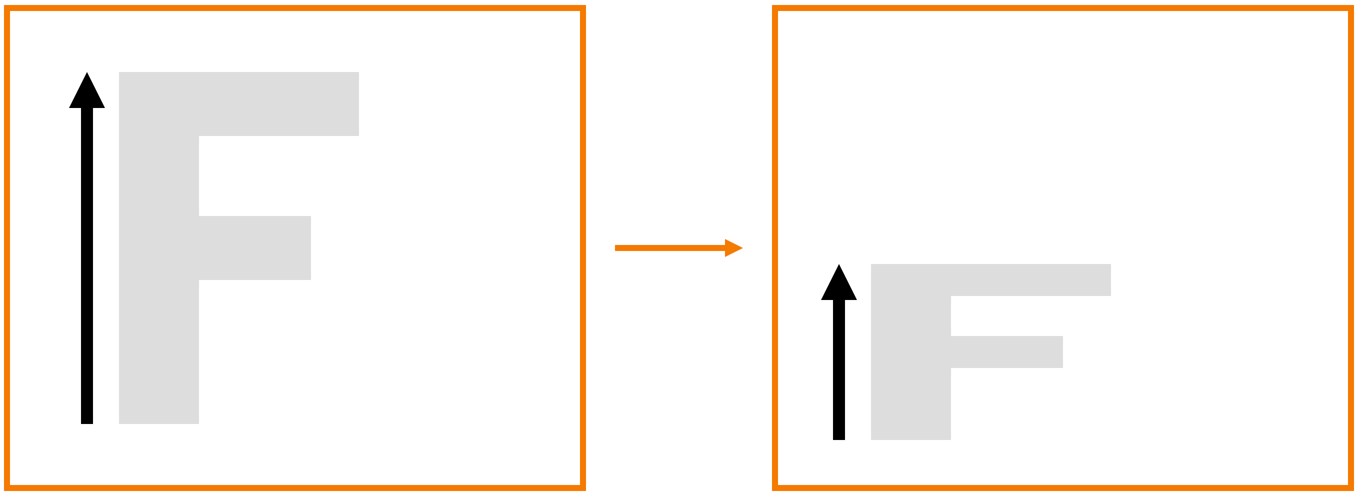
What happens to the "F"?



Translation!

Warping with One Line Pair

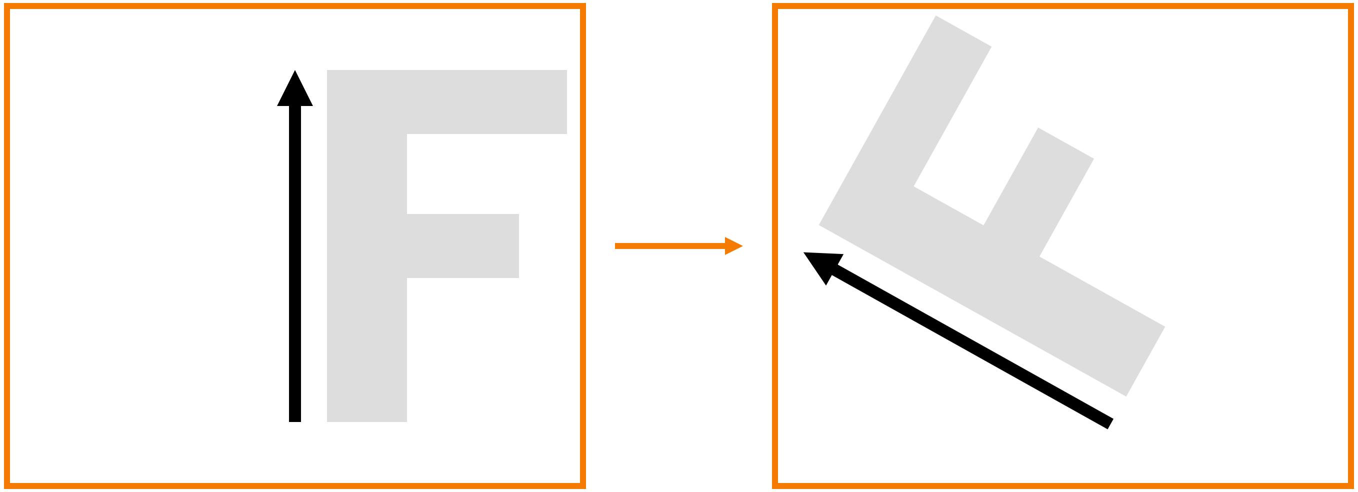
What happens to the "F"?



Non-uniform scale!

Warping with One Line Pair

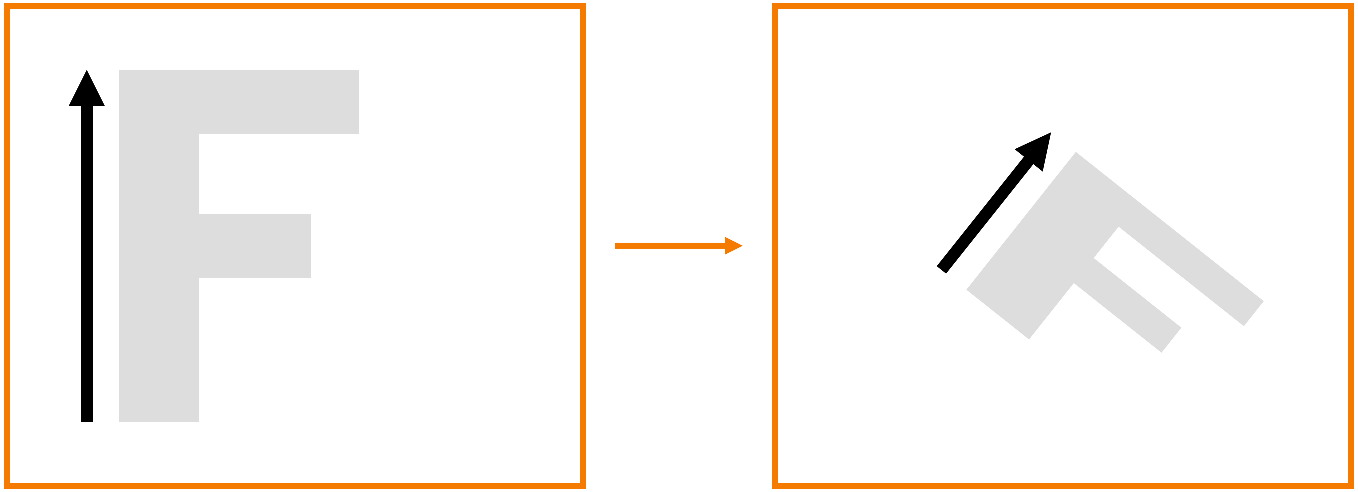
What happens to the “F”?



Rotation!

Warping with One Line Pair

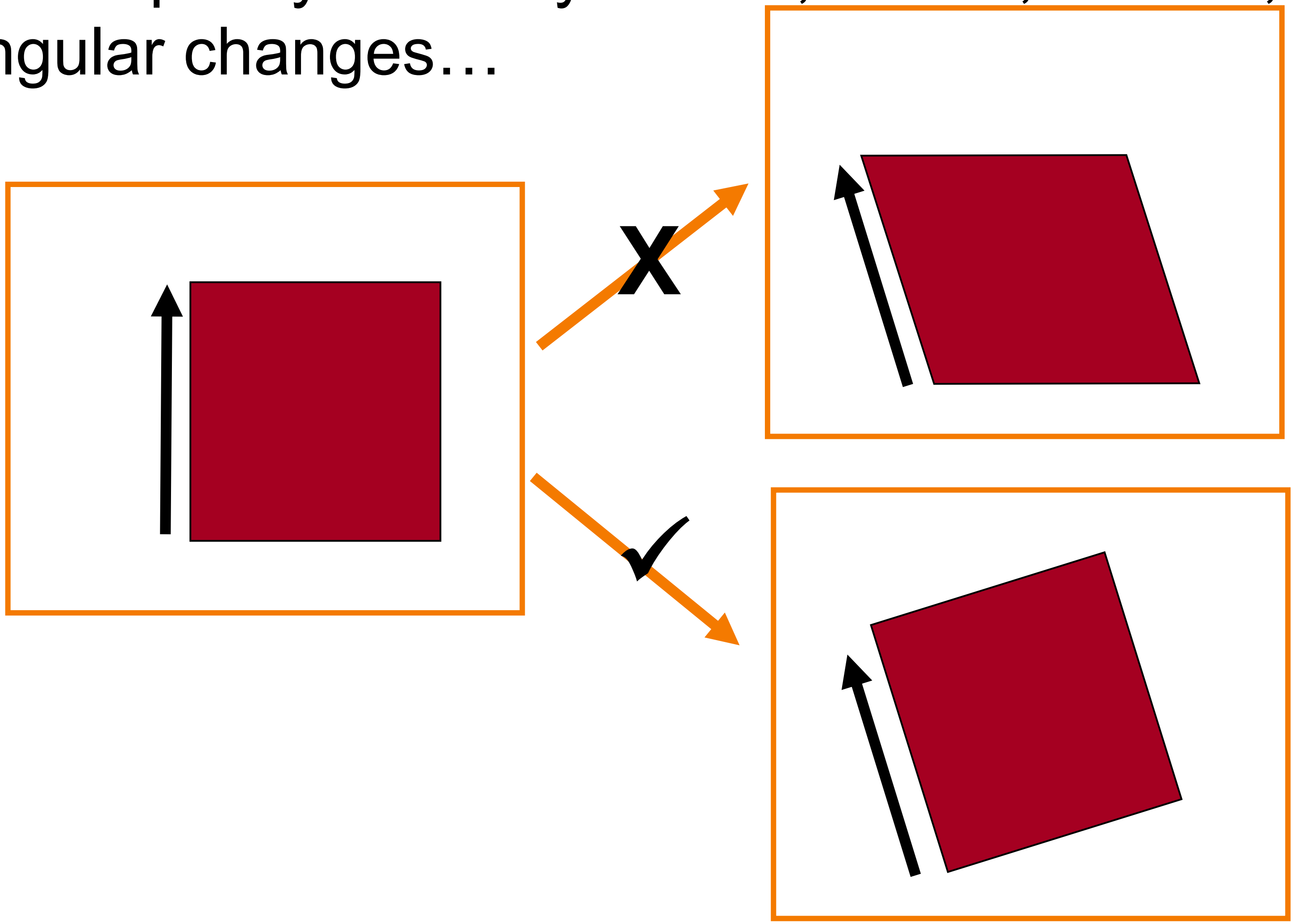
What happens to the "F"?



What types of affine transformations can't be specified?

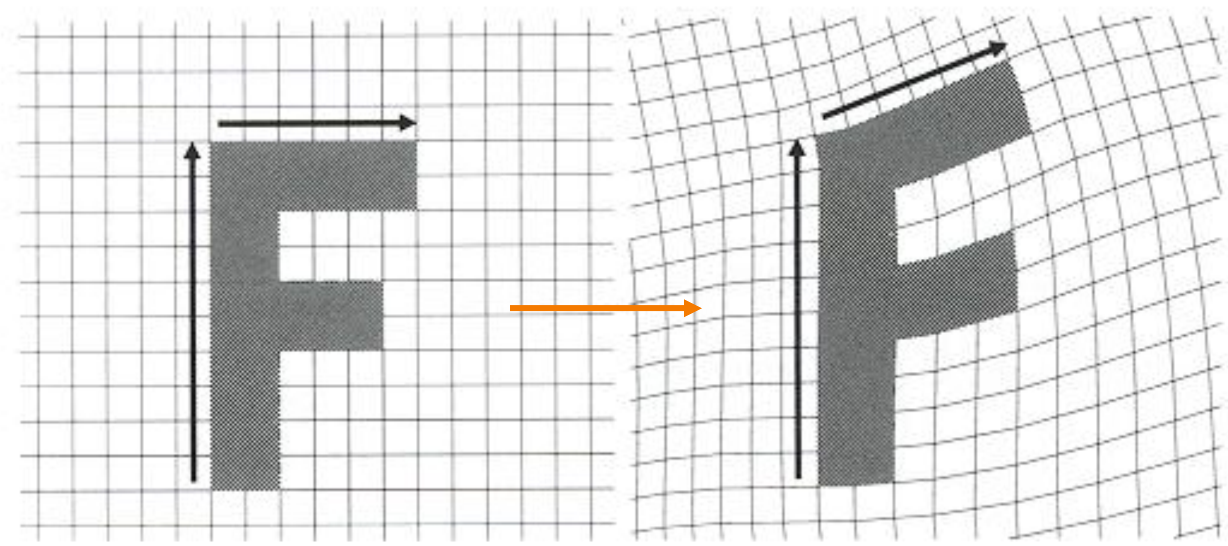
Warping with One Line Pair

Can’t specify arbitrary scales, skews, mirrors, angular changes…



Warping with Multiple Line Pairs

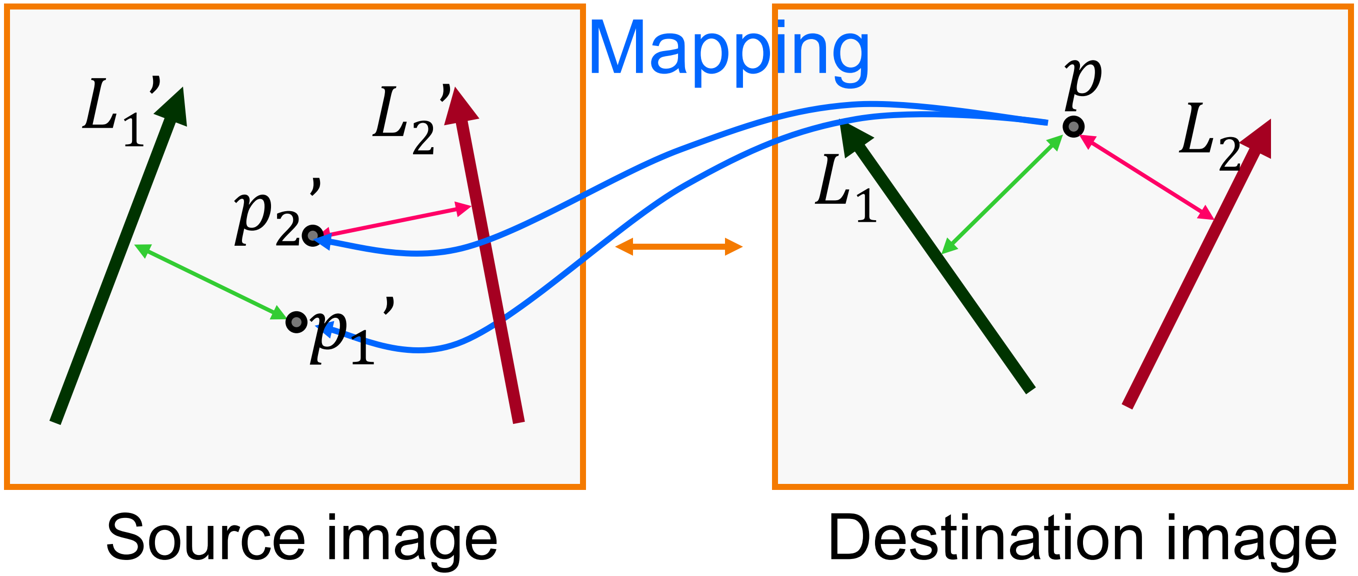
Use weighted combination of points defined by each pair of corresponding lines



Beier & Neeley, Figure 4

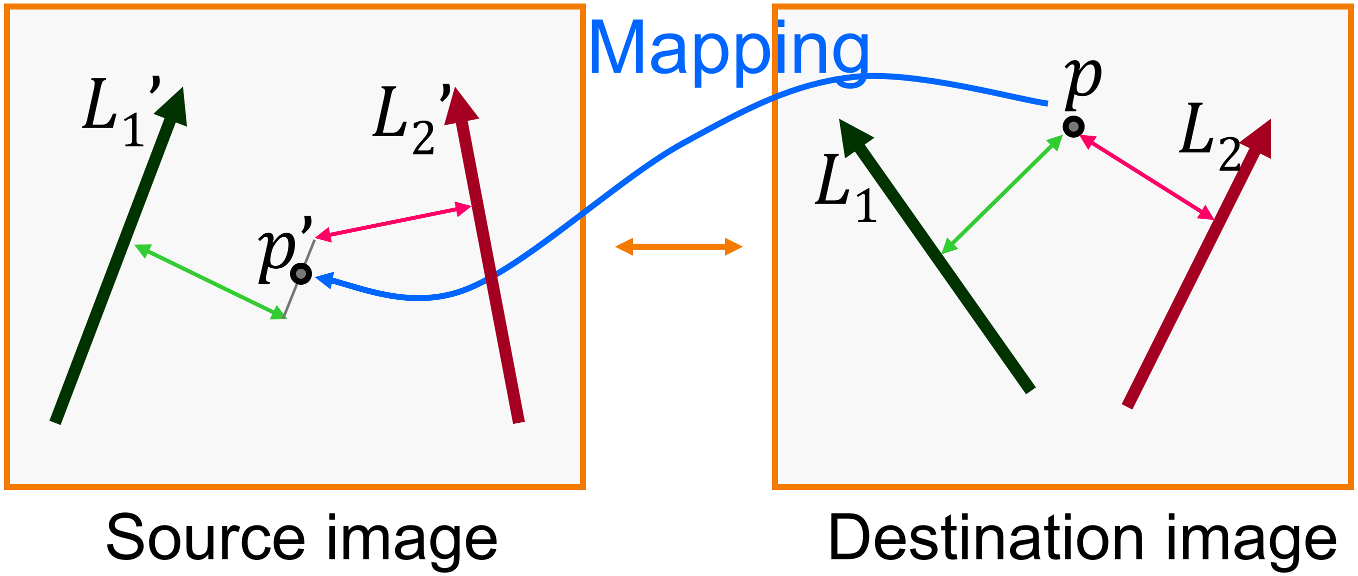
Warping with Multiple Line Pairs

Use weighted combination of points defined by each pair of corresponding lines



Warping with Multiple Line Pairs

Use weighted combination of points defined by each pair of corresponding lines



p' is a weighted average

Weighting Effect of Each Line Pair

Given a set of line pairs $\{L_{in}[0], \ldots, L_{in}[N]\}$ and $\{L_{out}[0], \ldots, L_{out}[N]\}$, to weight the contribution of each line pair, [Beier & Neely, 1992] use:

$$\text{weight}[i](p) \sim \left(\frac{\text{length}[i]^c}{a + \text{dist}[i](p)} \right)^b$$

where:

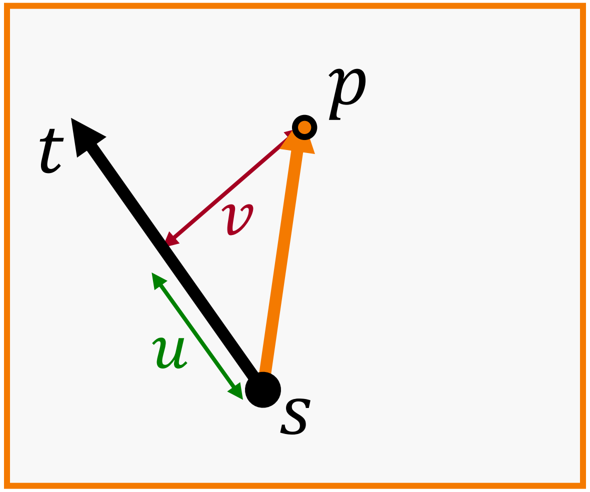
- $\text{length}[i]$ is the length of line $L_{out}[i]$
- $\text{dist}[i](p)$ is the distance from p to $L_{out}[i]$
- a (small), $b \in [0.5, 2.0]$, $c \in [0.0, 1.0]$ are constants that control the warp

Note: The "~" indicates "up to a constant". Need to normalize so weights sum to 1.

Feature-Based Warping

How do we calculate the unsigned distance from a point p to the line segment from s to t?

$$\text{dist}(p) = \begin{cases} |v| & \text{if } u \in [0,1] \\ \|p - s\| & \text{if } u < 0 \\ \|p - t\| & \text{if } u > 1 \end{cases}$$



Warping

Input:

Source image

Set of corresponding line segment pairs in source and target

Goal:

Warp the source image to a target image

Solution:

Iterating over each pixel in the target

- For each pair of line segments
  - » Compute the corresponding position in the source
  - » Compute the weights
- Average to get the final source position
- Sample the source (at the source position) to get the target color

Warping: Pseudocode

Warp(Img_{src} , L_{src}[N] , L_{tgt}[N])
{
    foreach target pixel p_{trgt}:
        $p_{src} = (0,0)$
        $sum = 0$
        for i = 0 to N:
            $q_{src} = p_{trgt}$ transformed by ($L_{src}[i]$, $L_{trgt}[i]$)
            p_{src} += q_{src} * $weight[i](p_{trgt})$
            sum += $weight[i](p_{trgt})$
        p_{src} /= sum
        $Img_{trgt}(p_{trgt}) = Img_{src}(p_{src})$
    return Img_{trgt}
}

Morphing

Input:

Source and target images

Set of corresponding line segment pairs in source and target

Interpolation time $\alpha \in [0,1]$

Goal:

The morph of the source to the target at time α

Solution:

- Compute the α-weighted average of the of line segments (by averaging the end-points)
- **Warp** the source using the source and averaged line segments
- **Warp** the target using the target and averaged line segments
- Compute the α-**blend** of the warped images

Morphing: Pseudocode

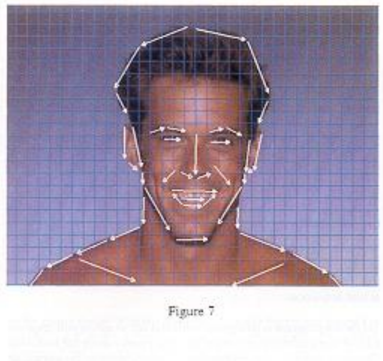
Morph(Img_0 , $L_0[N]$, Img_1 , $L_1[N]$, α)
{
    foreach $i \in \{1,\ldots,N\}$:
        $L_\alpha[i]$ = line α-th of the way from $L_0[i]$ to $L_1[i]$

    $Warp_0$ = Warp(Img_0 , $L_0[]$, $L_\alpha[]$)
    $Warp_1$ = Warp(Img_1 , $L_1[]$, $L_\alpha[]$)   } warp

    return $(1-\alpha)*Warp_0 + \alpha*Warp_1$ } α-blend
}

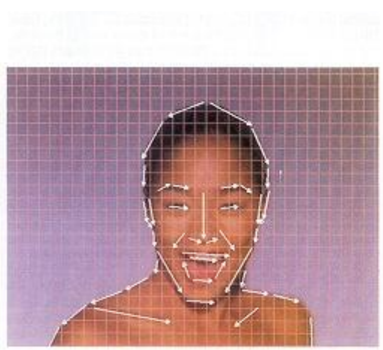
[Beier & Neely, 1992] Example $(\alpha = 0.5)$

Img_0



Warp_0

Img_1

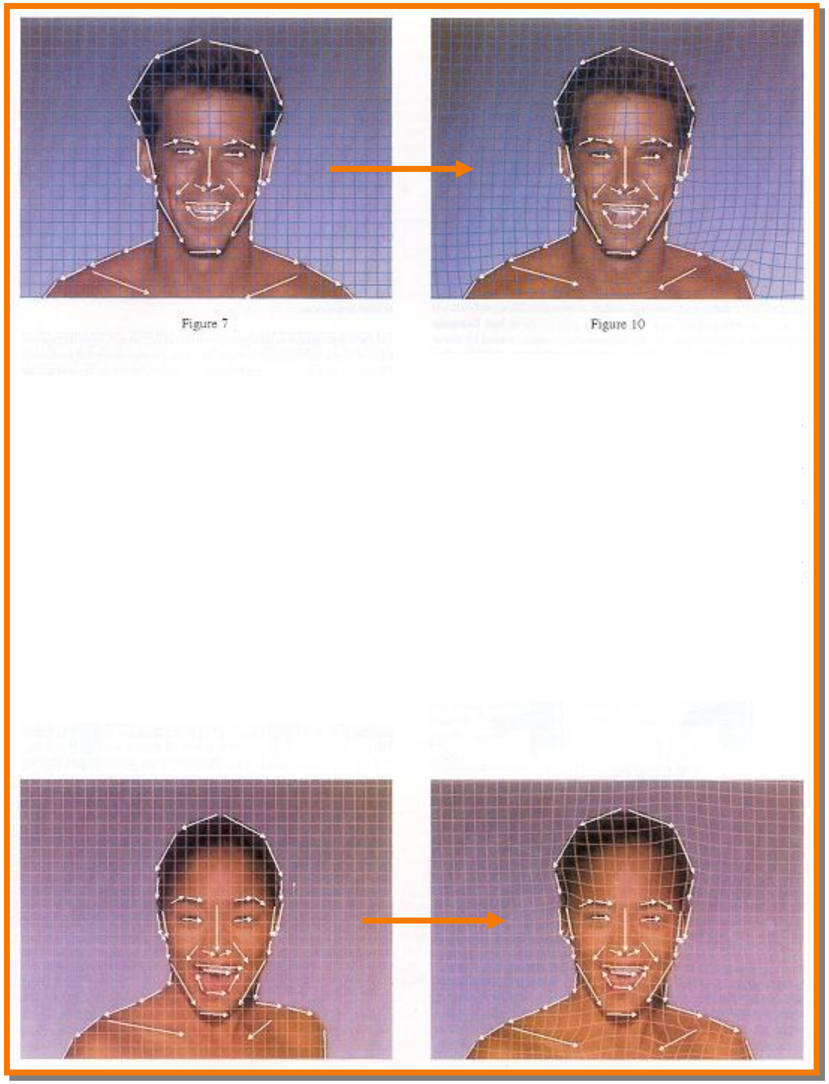


Warp_1

[Beier & Neely, 1992] Example $(\alpha = 0.5)$

Img$_0$

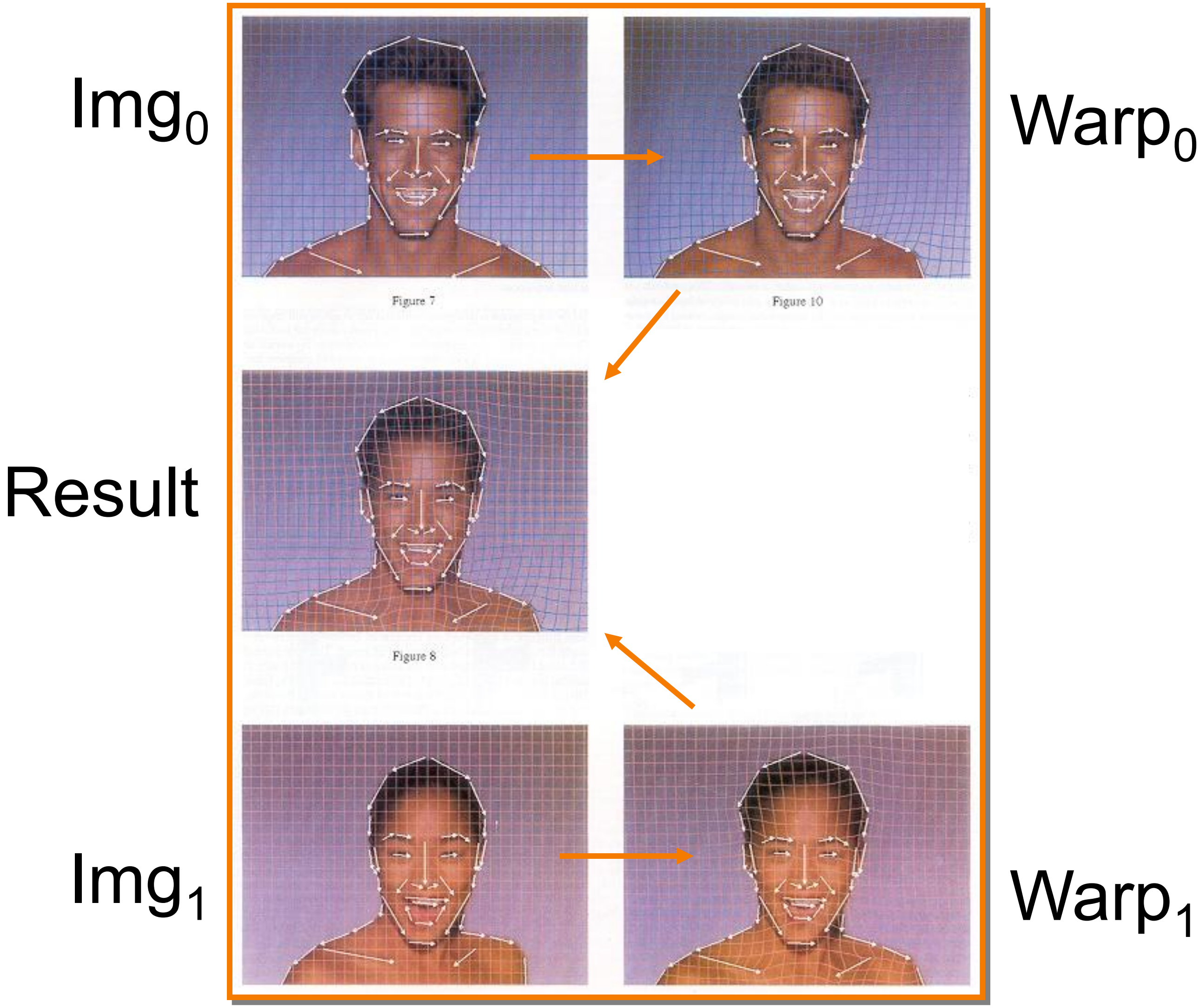
Warp$_0$



Img$_1$

Warp$_1$

[Beier & Neely, 1992] Example ($\alpha = 0.5$)

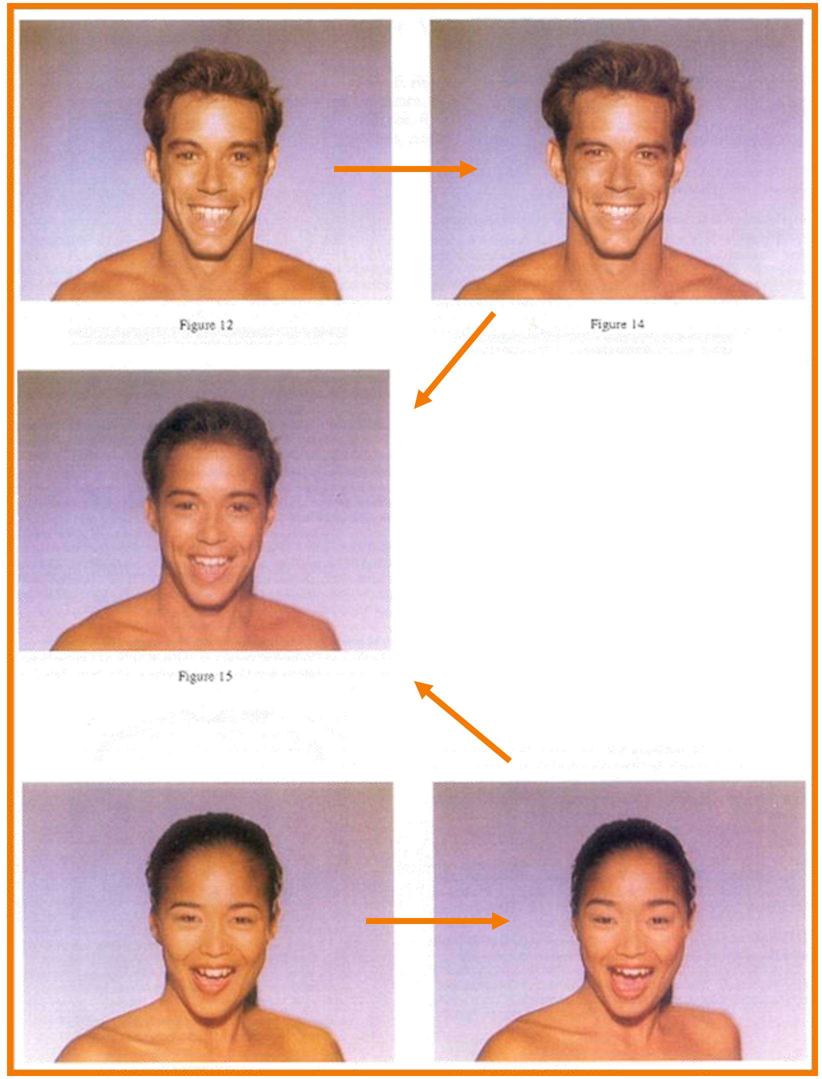


[Beier & Neely, 1992] Example $(\alpha = 0.5)$

Img_0

Warp_0

Result



Img_1

Warp_1

Full Morph Animation: Pseudocode

```
Animate( Img_0 , L_0[N] , Img_1 , L_1[N], Imgs_out[T+1] )
{
    foreach t∈{0,…,T}:
        Imgs_out[t] = Morph(Img_0 , L_0[N] , Img_1 , L_1[N] , t/T )
}
```

Morphing

Check out Michael Jackson's "Black or White" video at:

https://www.youtube.com/watch?v=pTFE8cirkdQ

Or the earlier Plymouth Voyager commercial at:

https://www.youtube.com/watch?v=0b939O7dGqQ