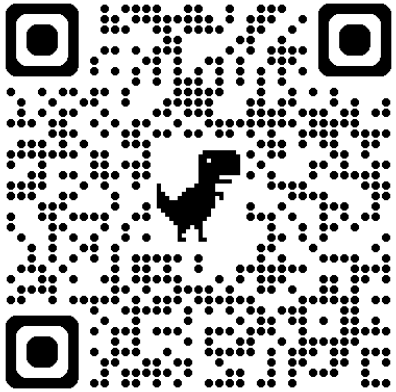


Image Filtering, Warping, and Morphing

Michael Kazhdan

(601.457/657)

Outline

- Image Filtering
- Image Warping
- Image Morphing

Image Filtering

We discussed image-processing where the only thing affecting the change at a pixel is the pixel's (own) value:

- Luminance
- Brightness
- Contrast (modulo a global average luminance)
- Saturation

What about when the changes depends on the values of adjacent pixels?

- Blurring
- Edge Detection
- Etc.

Multi-Pixel Operations

Stationary/Local Filtering

A general approach is to:

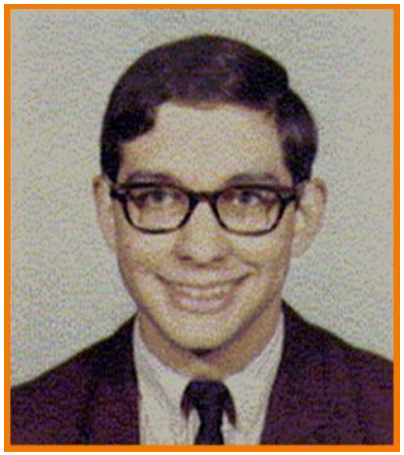
1. Define a *mask* of weights describing how values at adjacent pixels should be combined to generate the new value.
2. Apply the mask at every pixel.*

*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

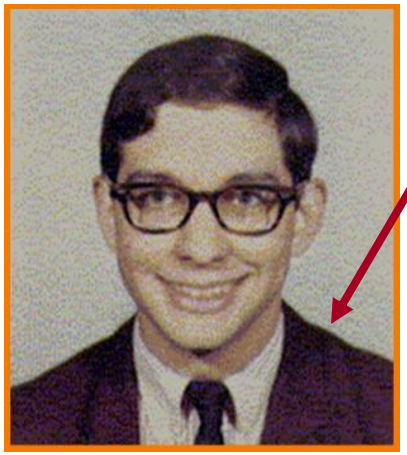


Original

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

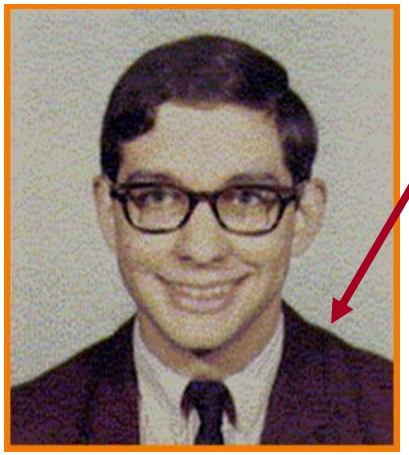


Original

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

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

Blurring



Original

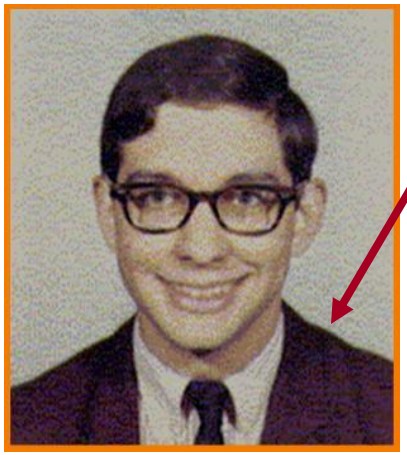
Pixel(x,y): red = 36
green = 36
blue = 0

	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



Original

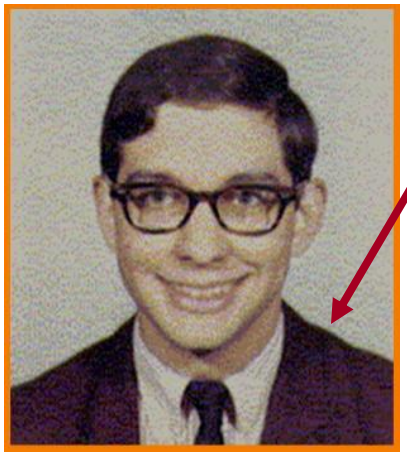
New value for Pixel(x,y).red =
(36 * 1/16) + (109 * 2/16) + (146 * 1/16)
(32 * 2/16) + (36 * 4/16) + (109 * 2/16)
(32 * 1/16) + (36 * 2/16) + (73 * 1/16)

	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



Original

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

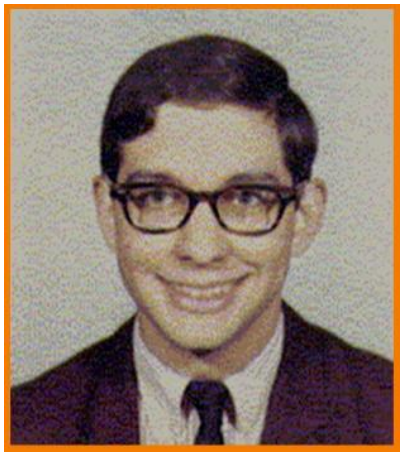
	x - 1	x	x + 1
y - 1	36	109	146
y	32	36	109
y + 1	32	36	73

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

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

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

Blurring

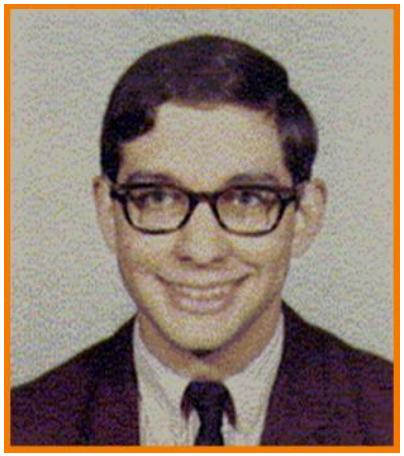
- Perform for each color channel.
- Keep source and destination separate to avoid drift.
- For boundary pixels, not all neighbors are used:
  - » Normalize the mask so the values sum to one, or
  - » Assume that the exterior values are black, or
  - » Assume the exterior values can be obtained by reflecting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

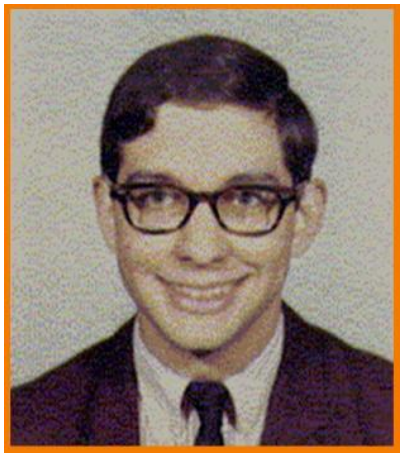
Narrow Blur

Blurring

Masks can have arbitrary size:

- Can expand smaller masks by zero-padding
- Can use more non-zero entries to get a wider blur

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



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
- $0 \leq i \leq 2r$ is the horizontal position in the mask
- $0 \leq j \leq 2r$ is the vertical position in the mask
- **Don't forget to normalize!**

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

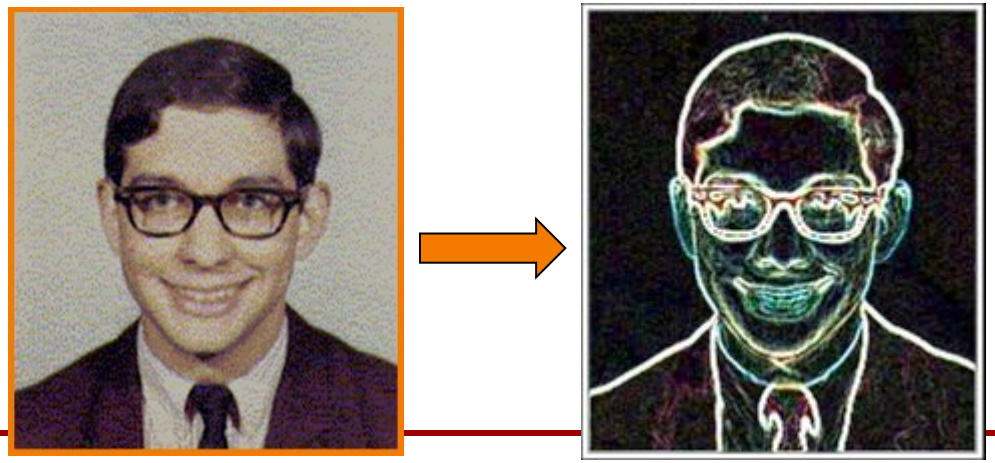
Edge Detection

An edge is where the image is far from constant:

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

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

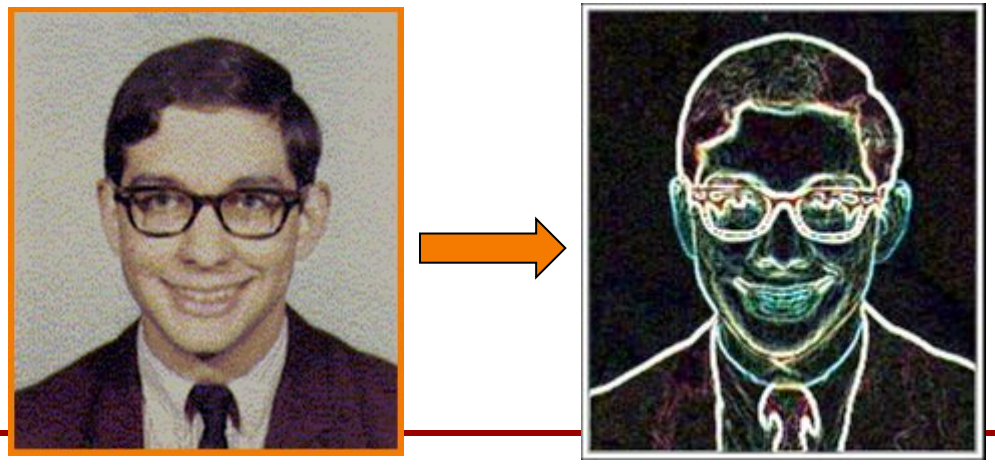
This is sometimes referred to as a **Laplacian** filter.

Pixels with large absolute values correspond to edges:

- Positive values: “upper” edges
- Negative values: “lower” edges

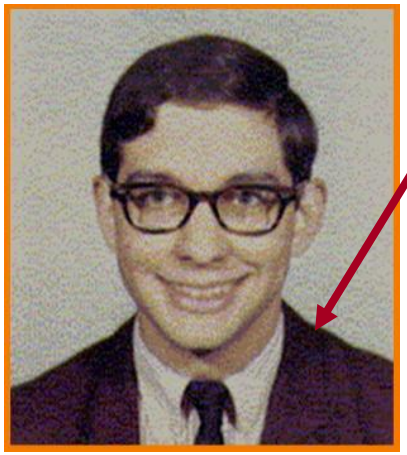
Define a mask whose:

- 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

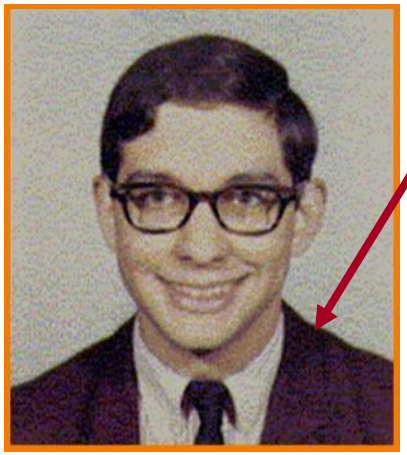


Original

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

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

Edge Detection



Original

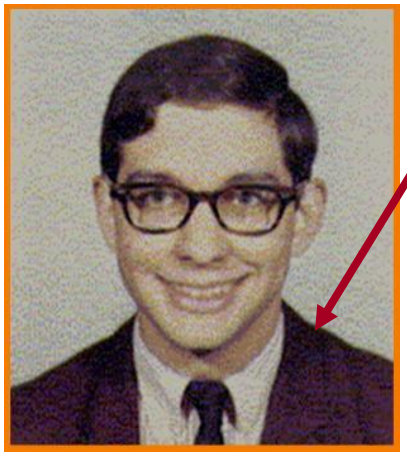
Pixel(x,y): red = 36
green = 36
blue = 0

	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

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

Edge Detection



Original

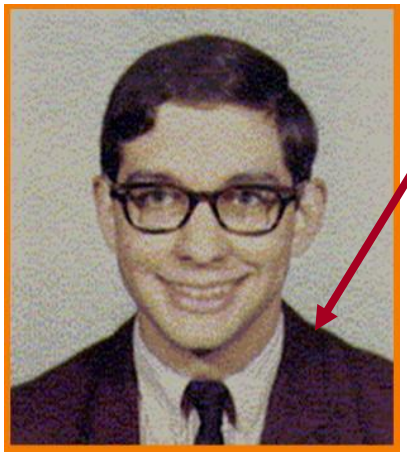
New value for Pixel(x,y).red =
(36 * -1/8) + (109 * -1/8) + (146 * -1/8)
(32 * -1/8) + (36 * 1) + (109 * -1/8)
(32 * -1/8) + (36 * -1/8) + (73 * -1/8)

	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



Original

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

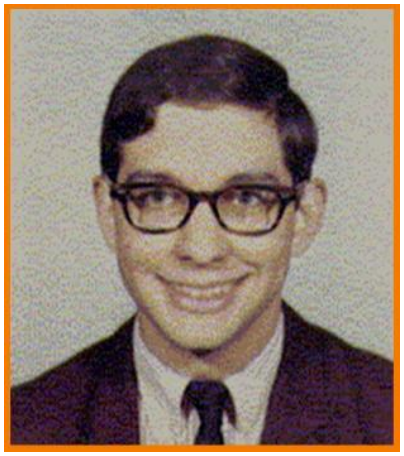
	x - 1	x	x + 1
y - 1	36	109	146
y	32	36	109
y + 1	32	36	73

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.

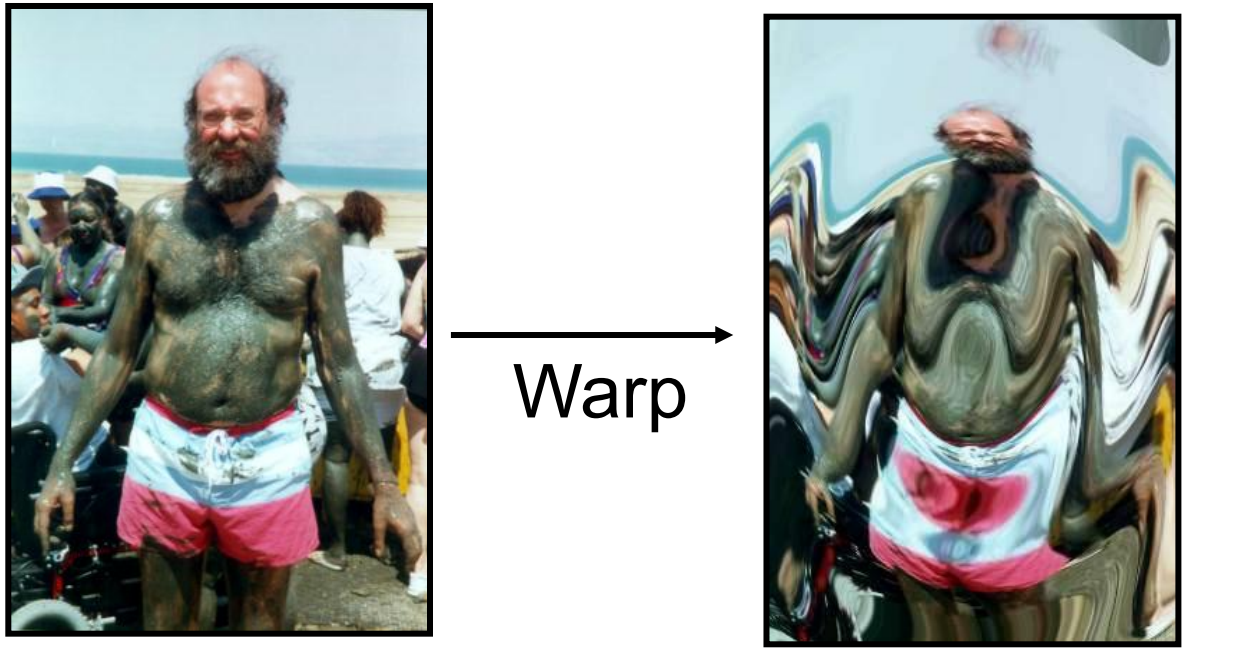
Outline

- Image Filtering
- **Image Warping**
- Image Morphing

Image Warping

"Move" pixels of image

- Mapping
- Resampling



Source image

Destination image

Overview

Mapping

- Forward
- Inverse

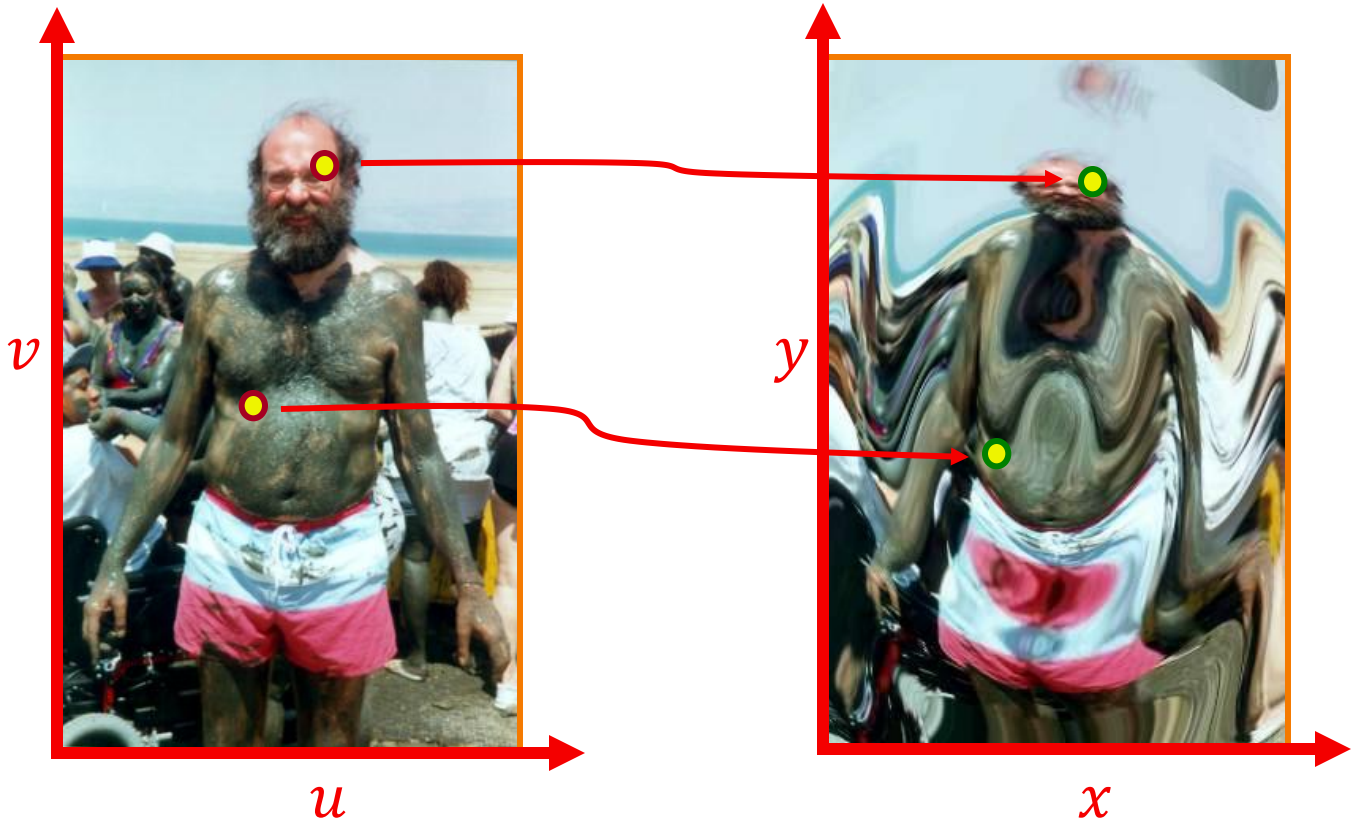
Resampling

- Point sampling
- Triangle filter
- Gaussian filter

Mapping

Define mapping

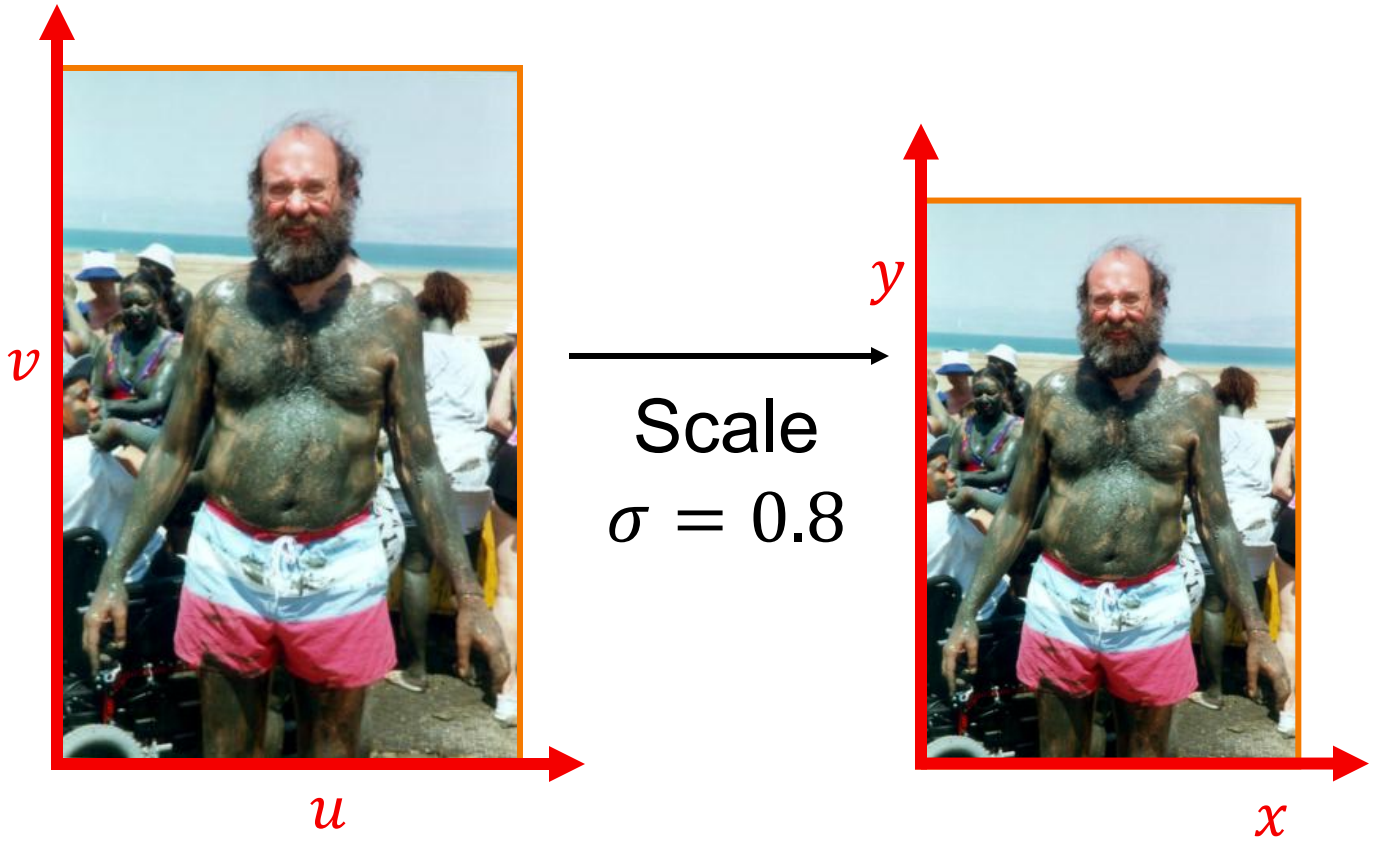
- Describe the destination $(x, y) = \Phi(u, v)$ for every location (u, v) in the source



Example Mappings

Scale by σ:

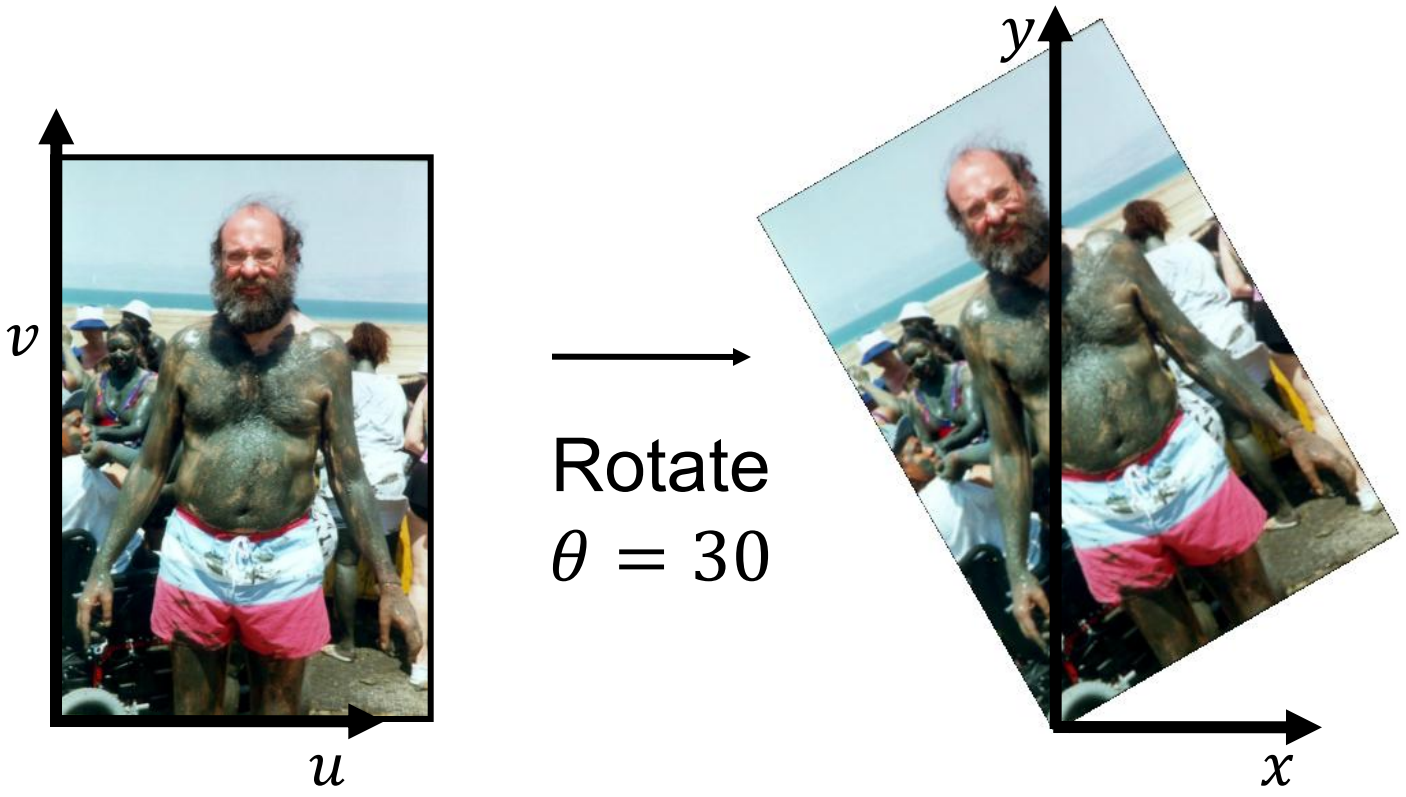
- $\Phi(u, v) = (\sigma u, \sigma v)$



Example Mappings

Rotate by θ degrees:

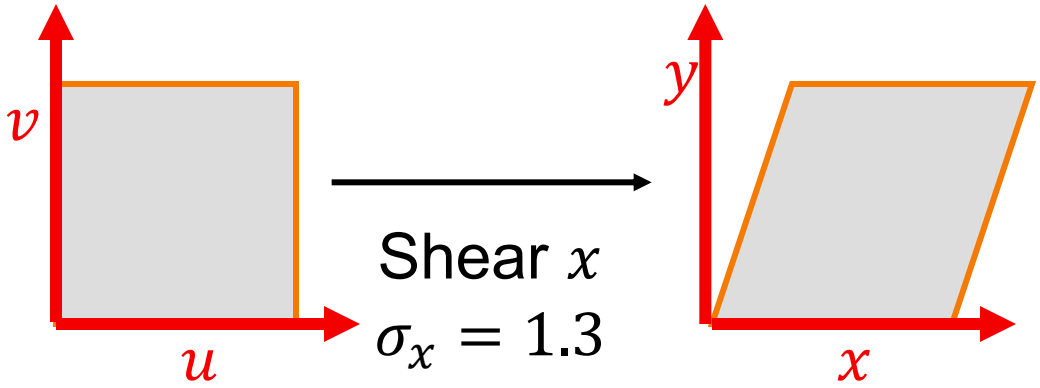
- $\Phi(u, v) = (u\cos\theta - v\sin\theta\,, u\sin\theta + v\cos\theta)$



Example Mappings

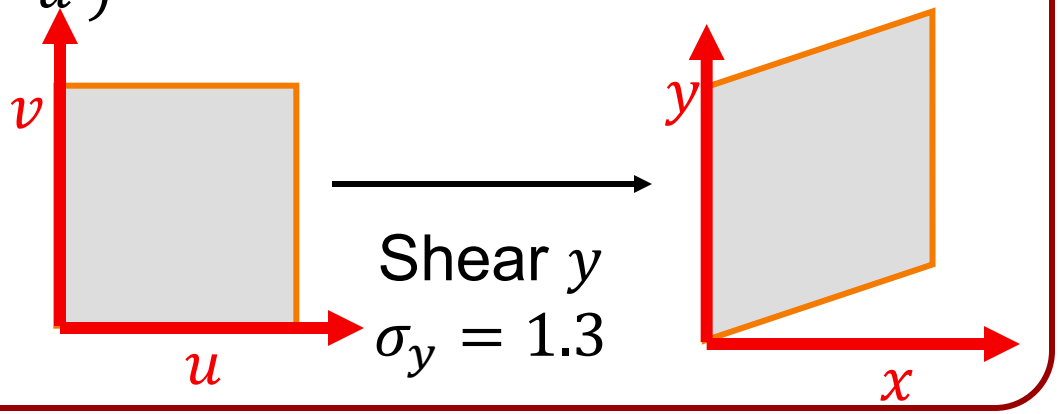
Shear in x by σ_x:

- $\Phi(u, v) = (u + \sigma_x \cdot v, v)$



Shear in y by σ_y:

- $\Phi(u, v) = (u, v + \sigma_y \cdot u)$

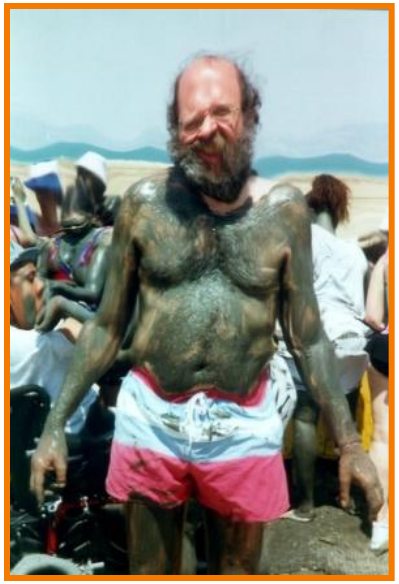


Other Mappings

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

Fish-eye

"Swirl"

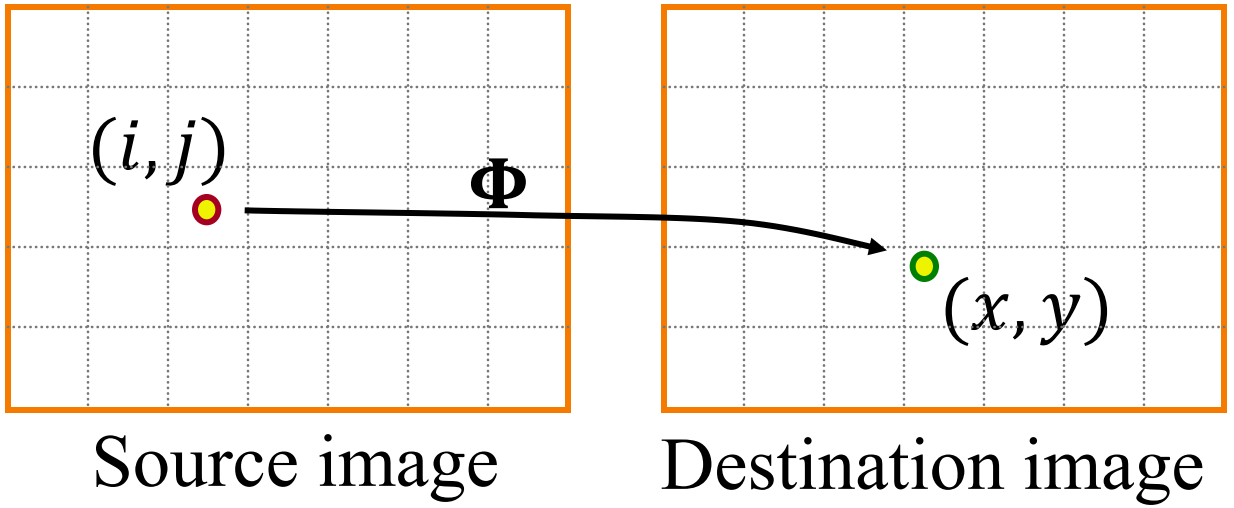


"Rain"

Image Warping Implementation I

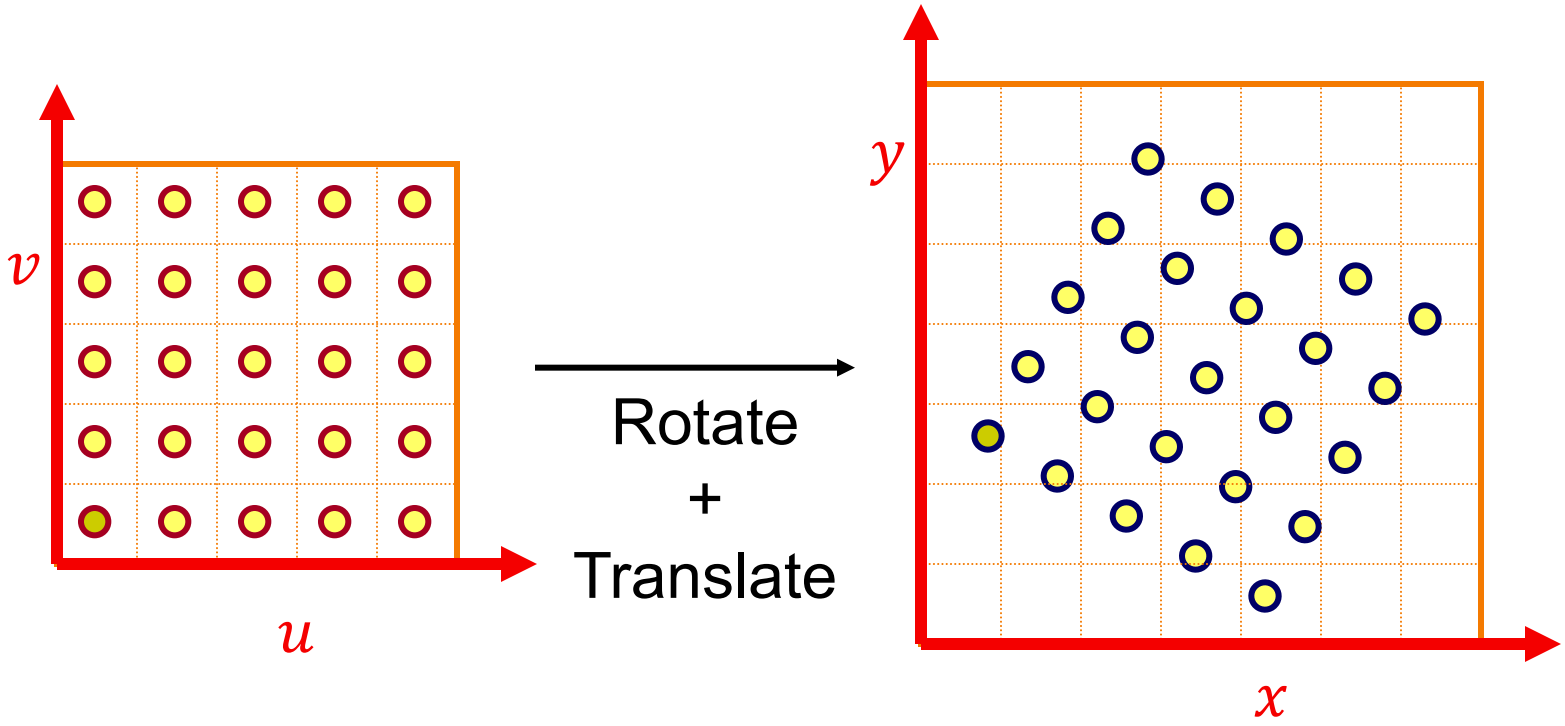
Forward mapping:

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



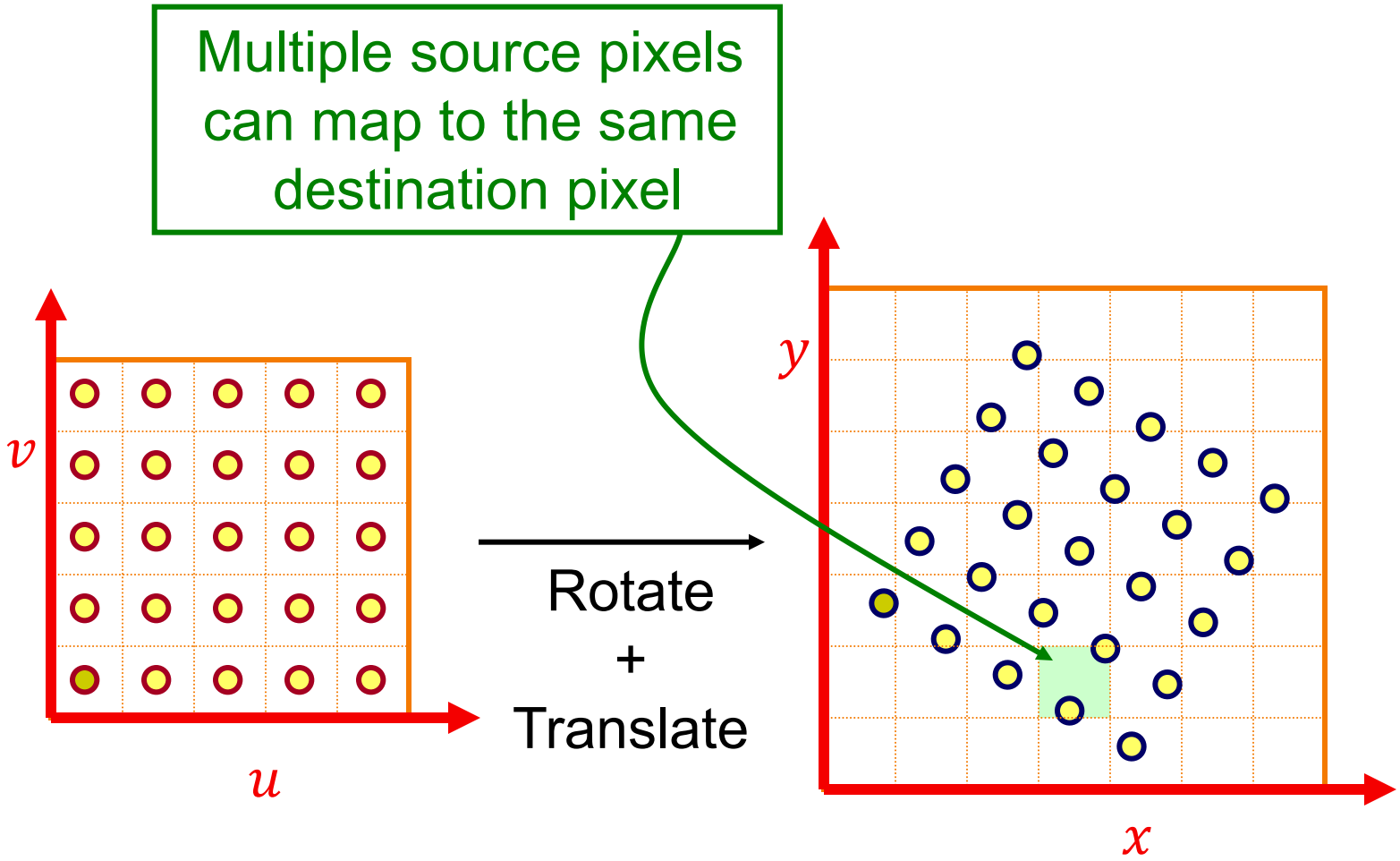
Forward Mapping

Iterate over source



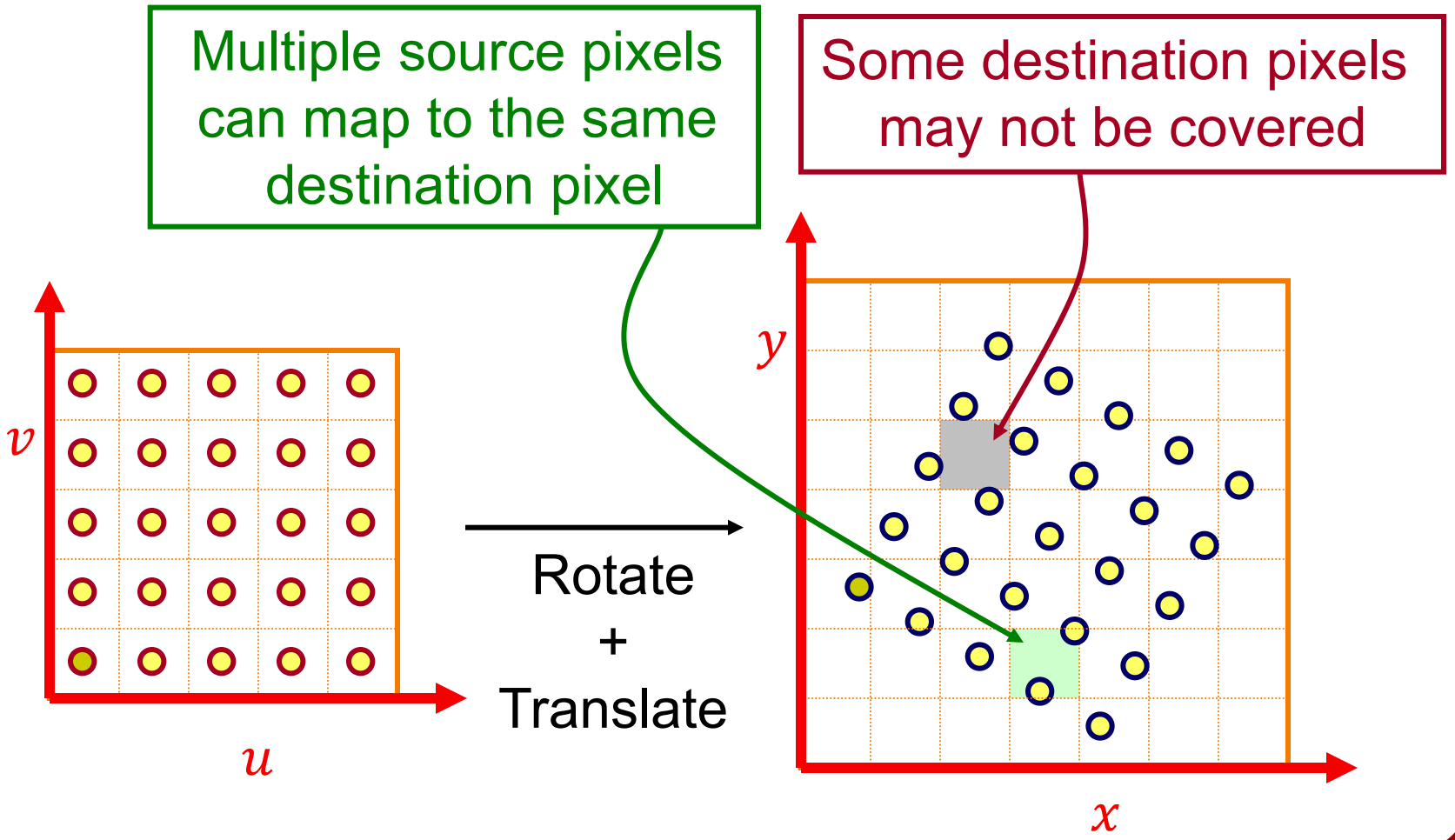
Forward Mapping – BAD!

Iterate over source



Forward Mapping – BAD!

Iterate over source



Forward Mapping – BAD!

Iterate over source

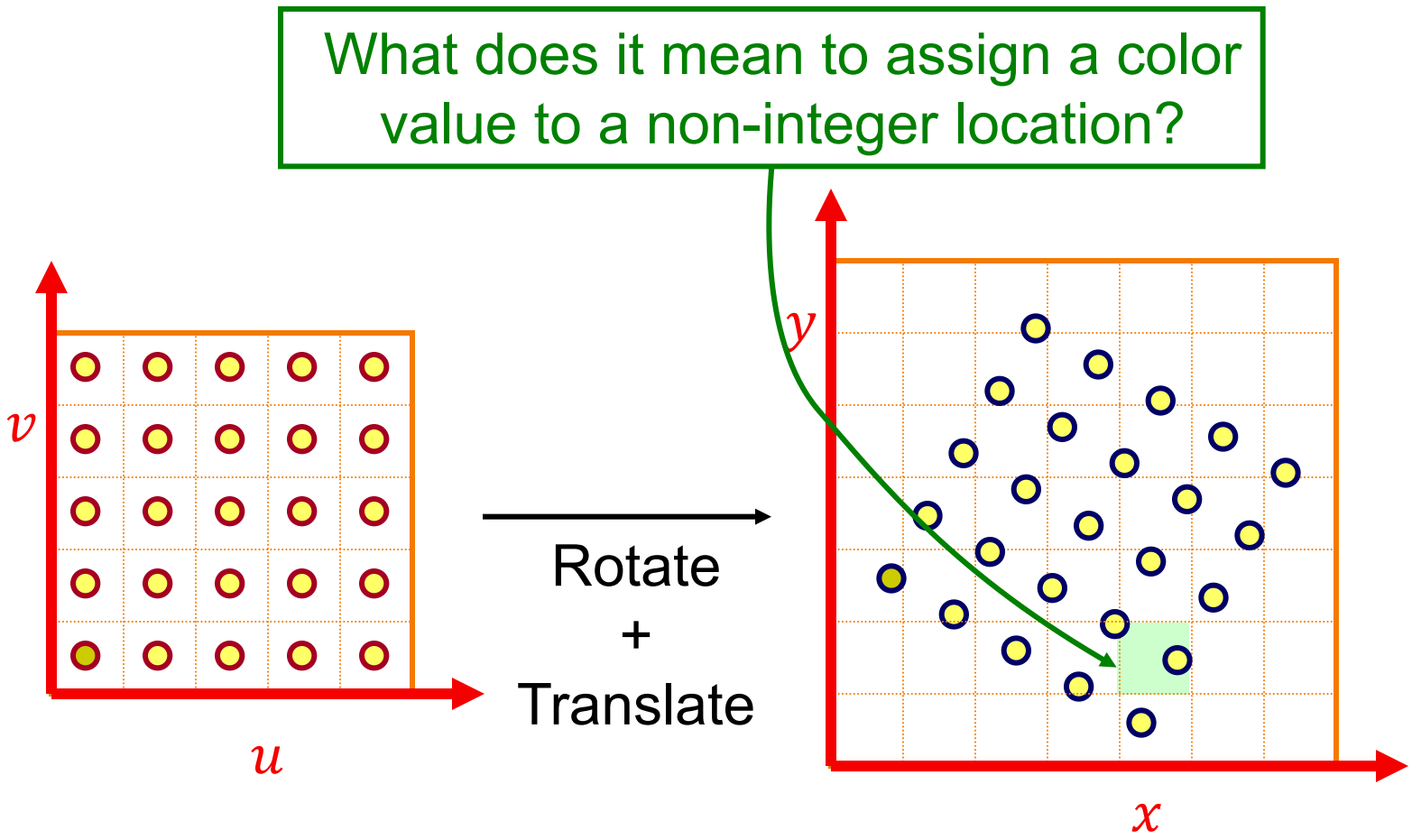


Image Warping Implementation II

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);
```

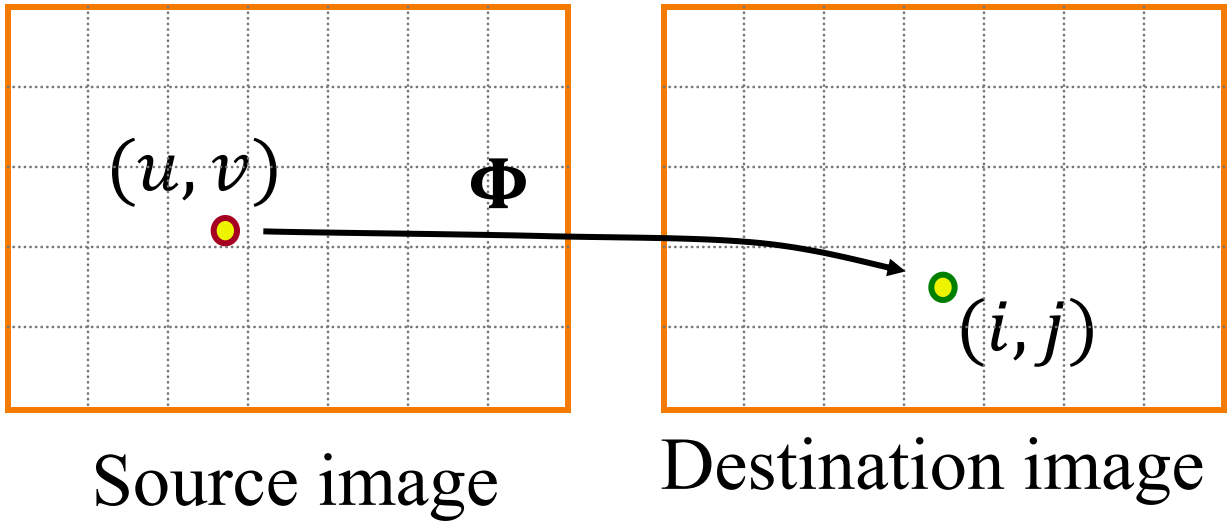
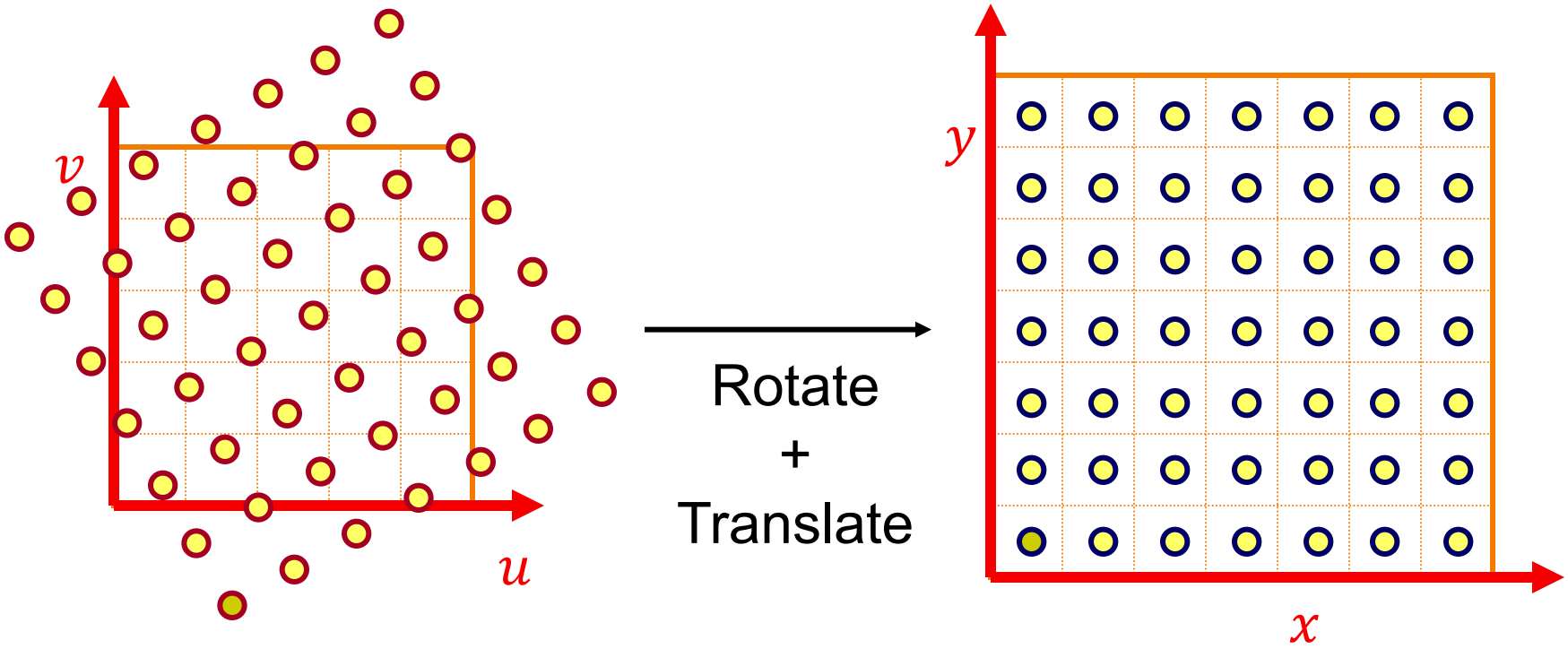


Image Warping Implementation II

Inverse mapping:

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

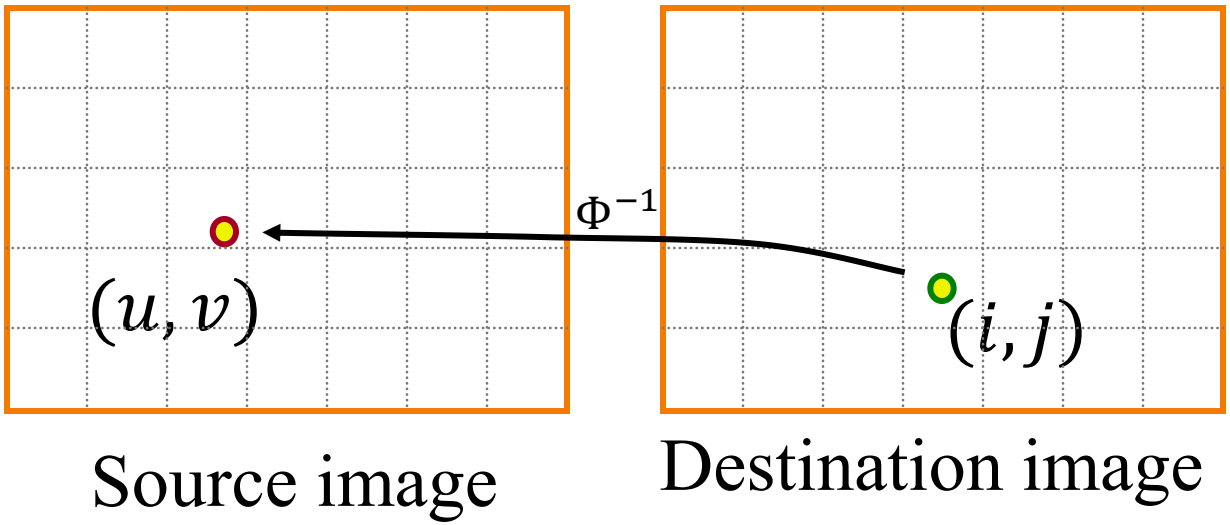
Need to check that source location is in bounds.



Resampling

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

(u, v) does not usually have integer coordinates



Overview

Mapping

- Forward
- Inverse

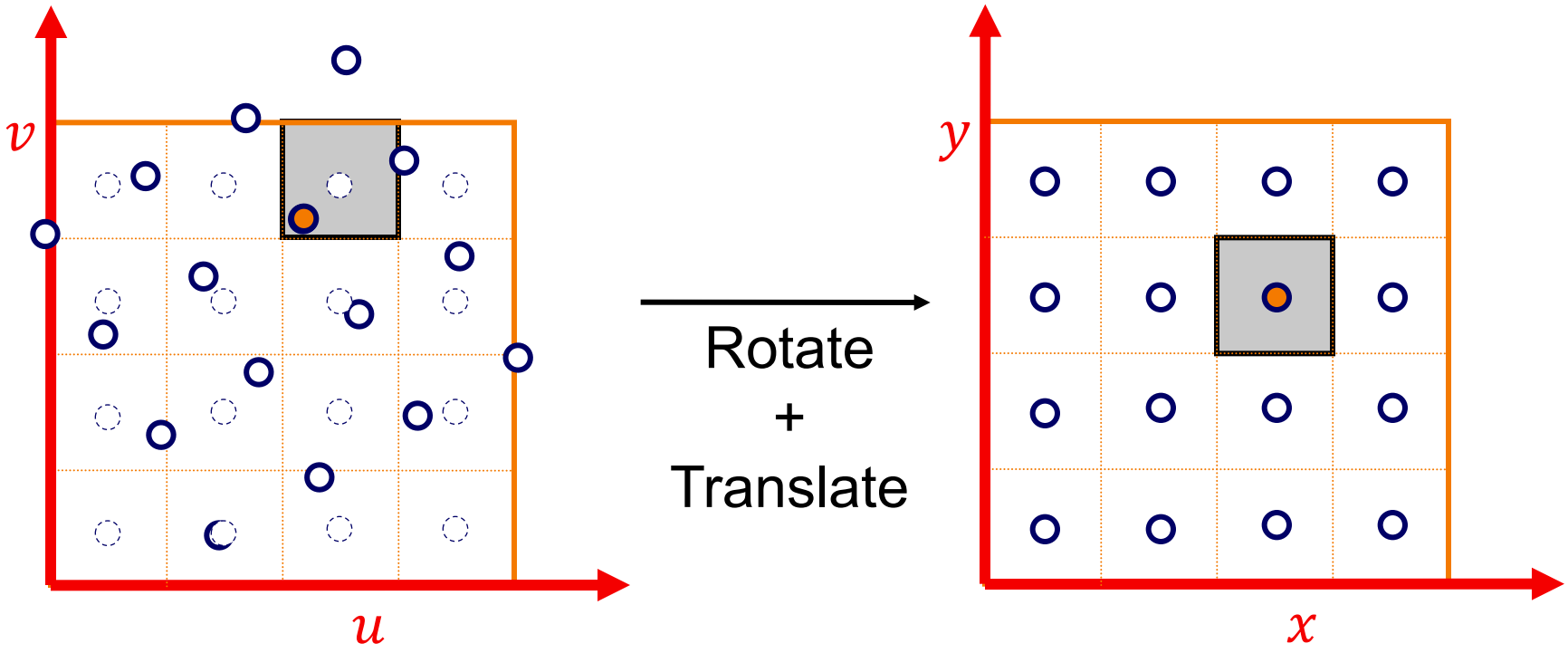
Resampling

- Nearest Point Sampling
- Bilinear Sampling
- Gaussian Sampling

Nearest Point Sampling

Take value at closest pixel:

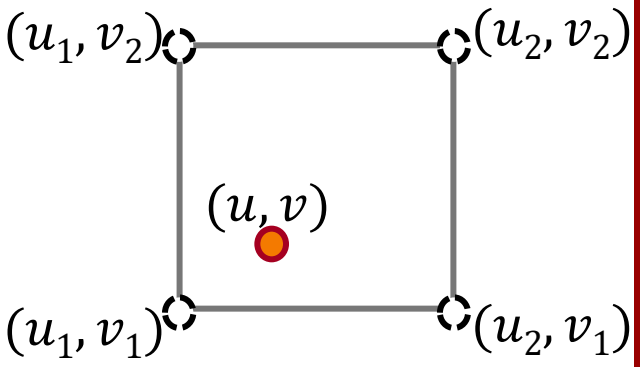
```
int intU = std::round(u);
int intV = std::round(v);
dst(i,j) = src(intU,intV);
```

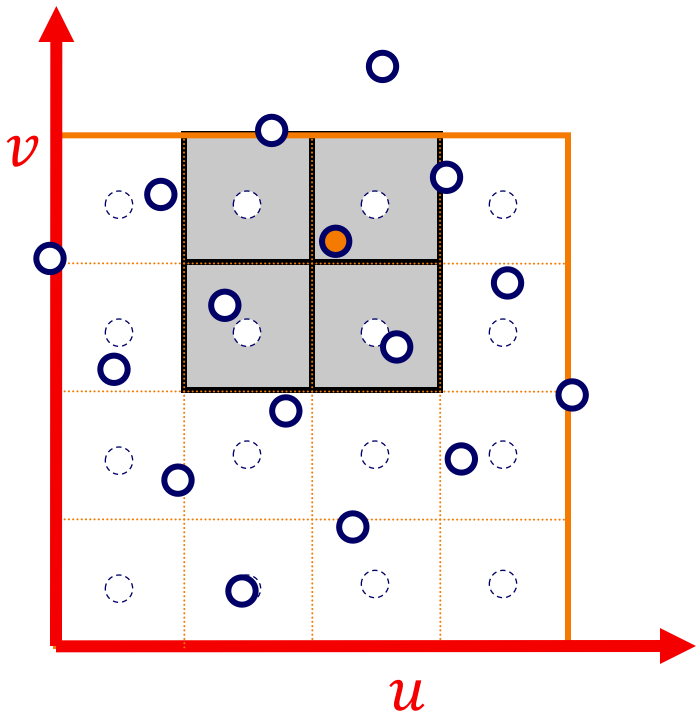


Bilinear Sampling

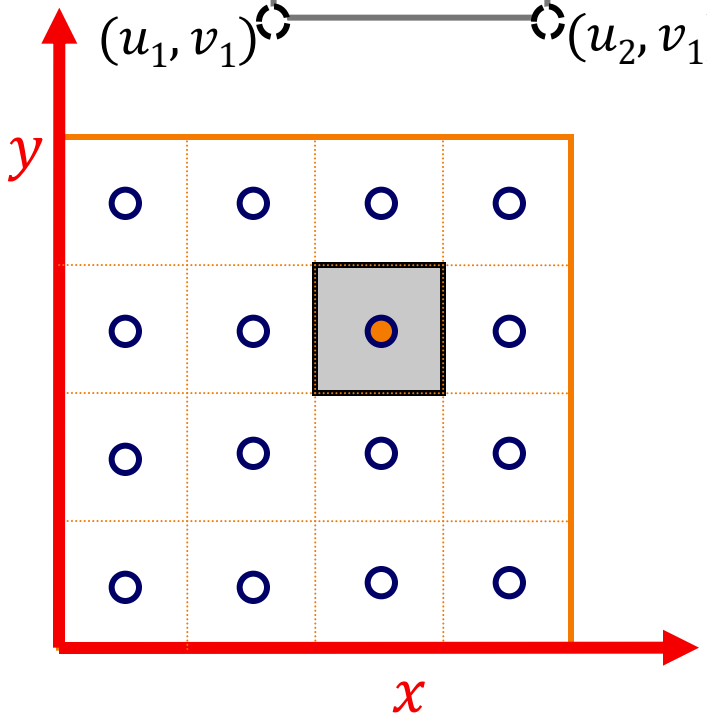
Bilinearly interpolate four closest source pixels:

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





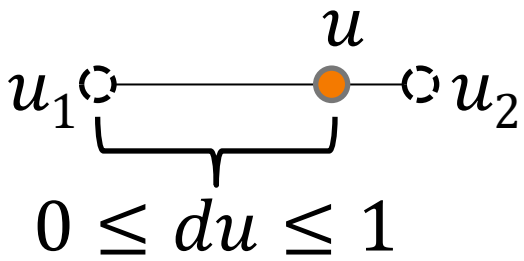
Rotate
+
Translate



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;
```

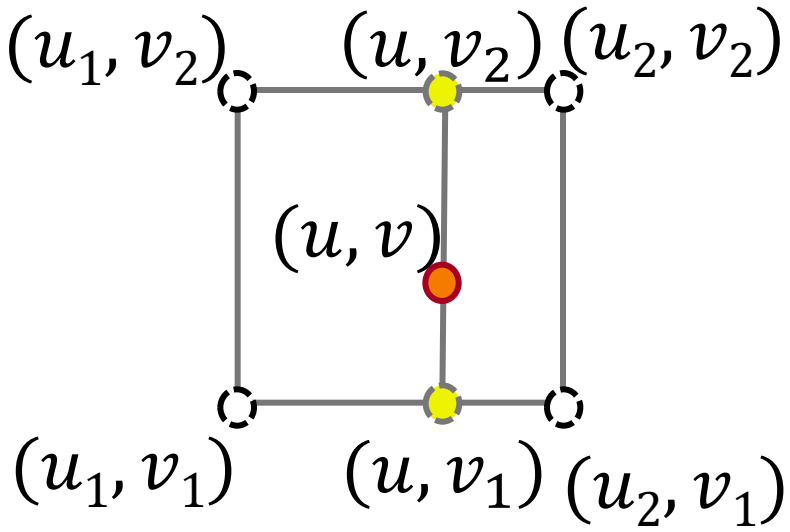
Bilinear Sampling

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




```
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_1, v_2) (u, v_2) (u_2, v_2)

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.

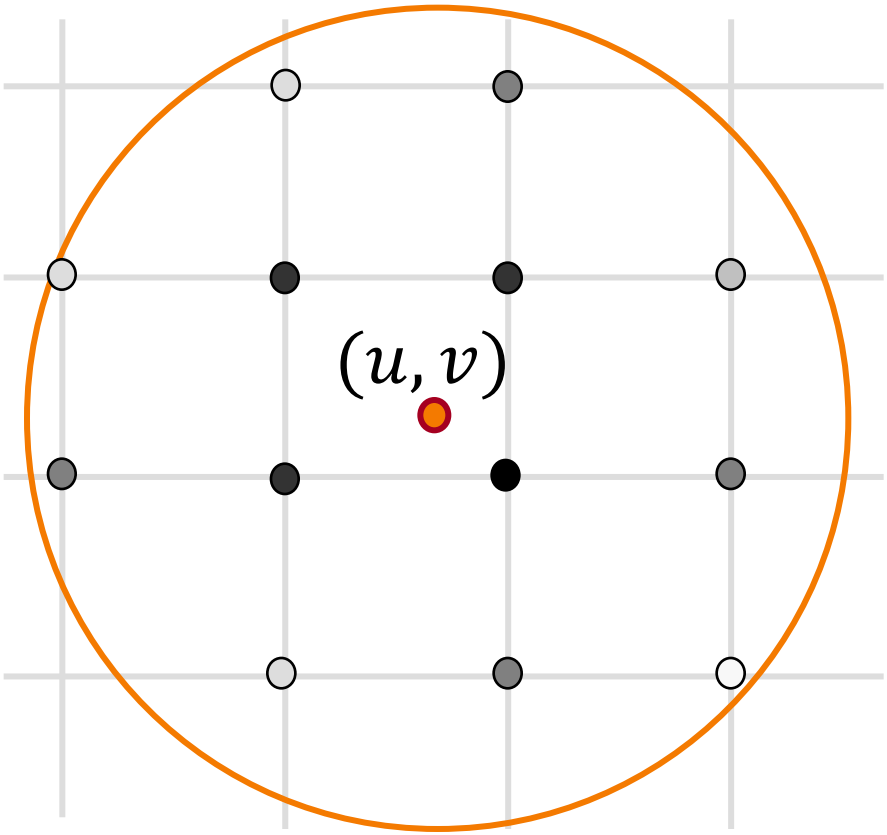
(u_1, v_1) (u, v_1) (u_2, v_1)

```
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;
```

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, the source can't be assumed to be at an integer location.

⇒ Can't precompute a stencil in advance

Image Warping Implementation

Inverse mapping with Gaussian sampling:

```
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);
```

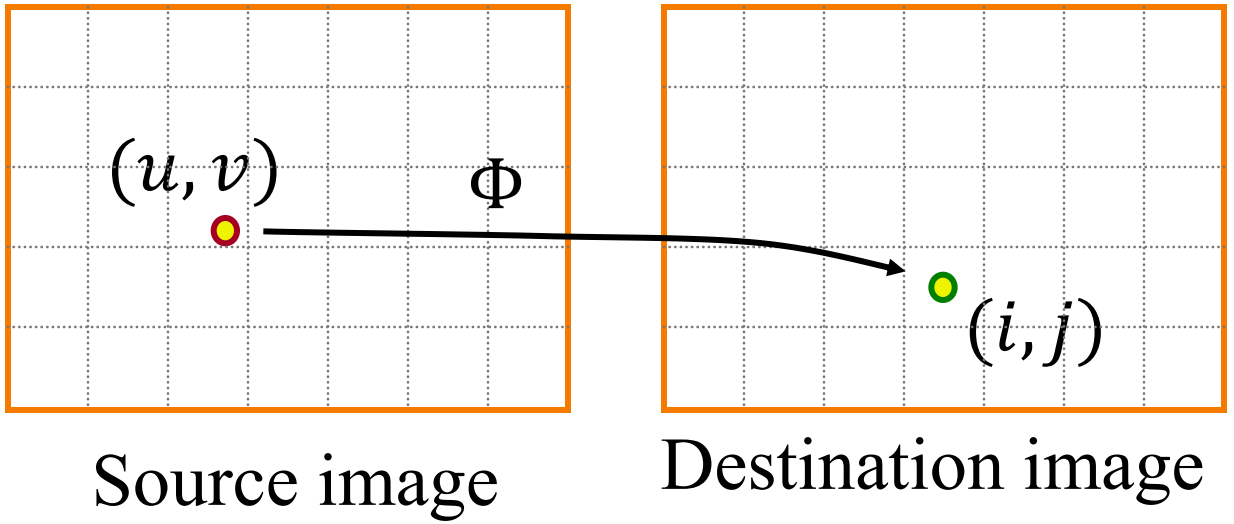


Image Warping Implementation

Inverse mapping with Gaussian sampling:

```
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);
```

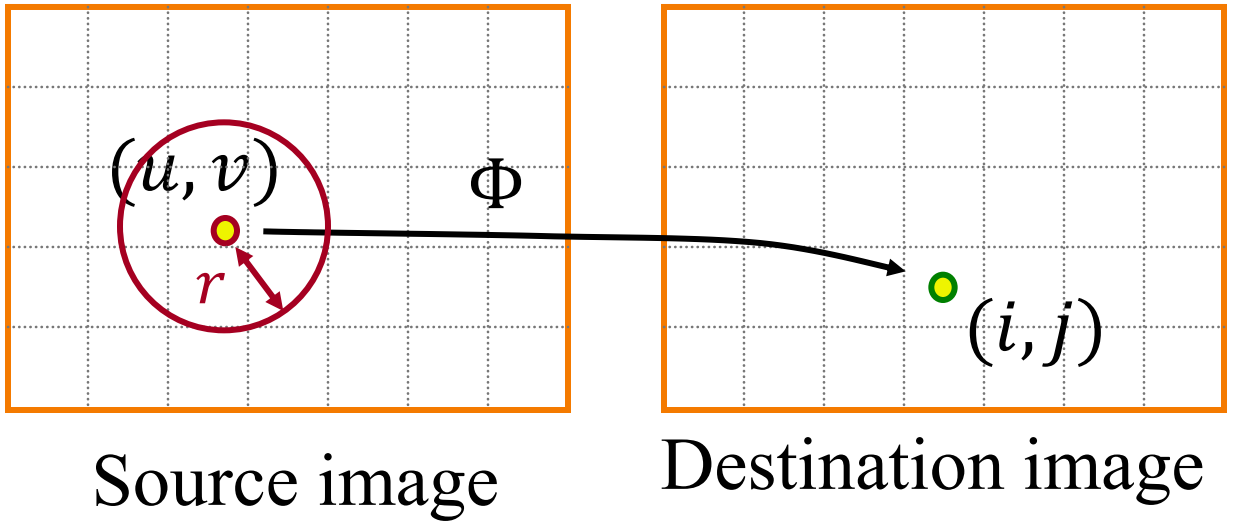


Image Warping Implementation

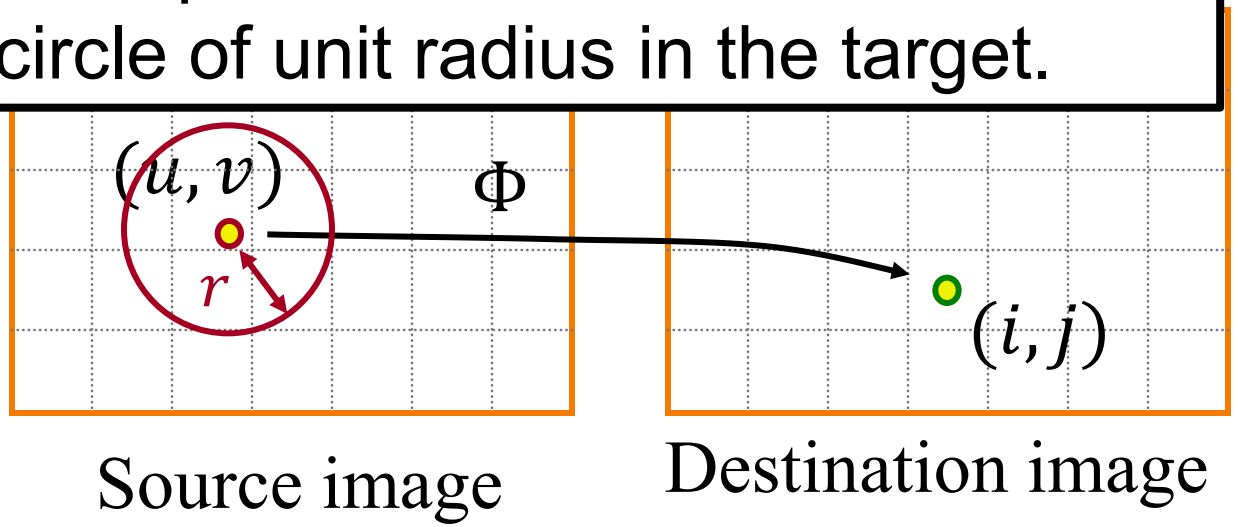
Inverse mapping with Gaussian sampling:

```
for(
  fo
```

Informally:
A target pixel "owns" the region in the circle of radius one around it.

⇓

The circle we sample from on the source should map to a circle of unit radius in the target.

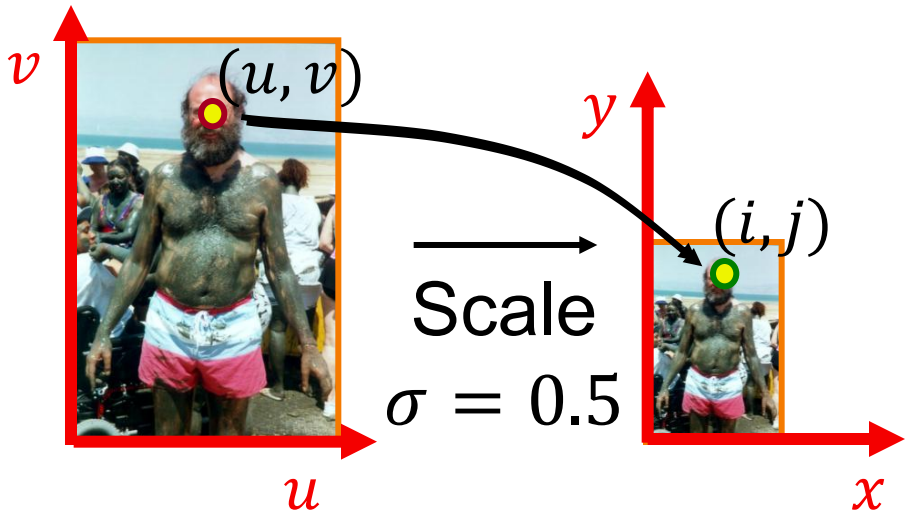


Example: Scale

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);
```

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

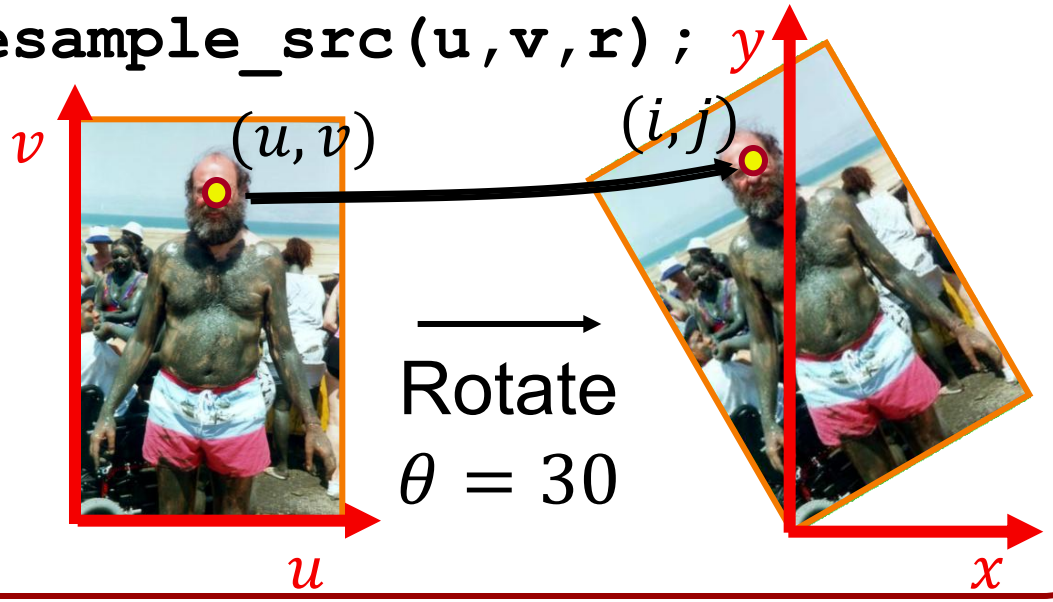


Example: Rotate

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);
```

$r = 1$

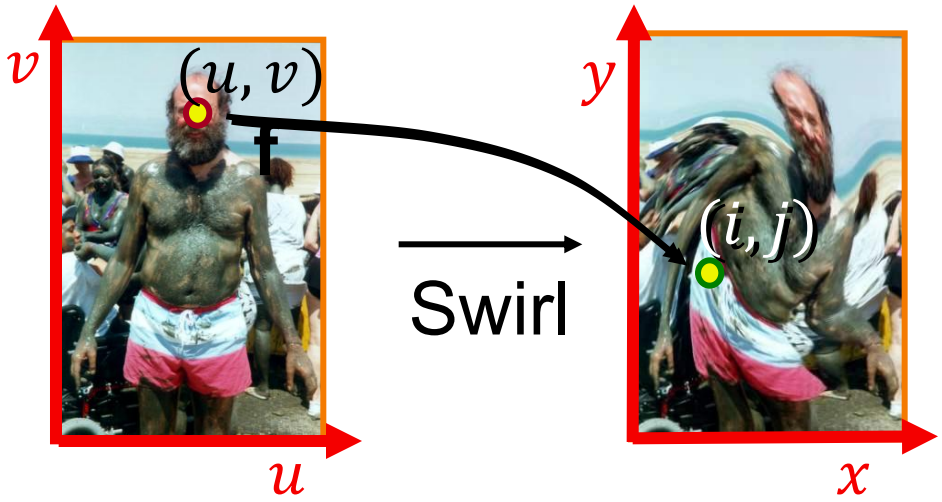


Example:

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);
```

$r = ?$



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:

- Have the radius change,
- Use an ellipse.

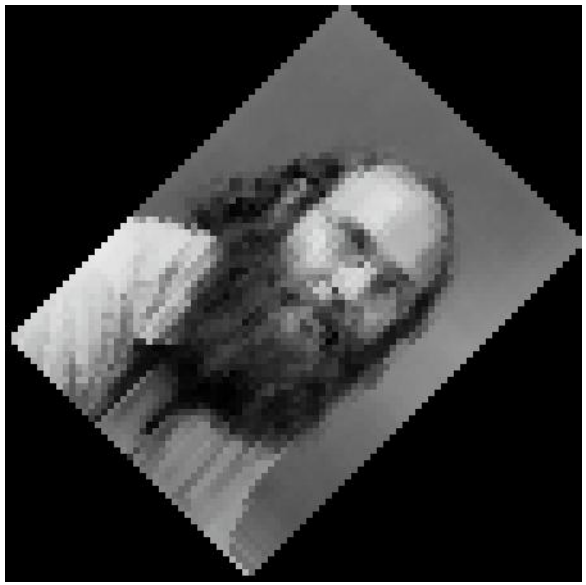
This can be done by looking at the derivative/Jacobean of Φ.

Filtering Methods Comparison

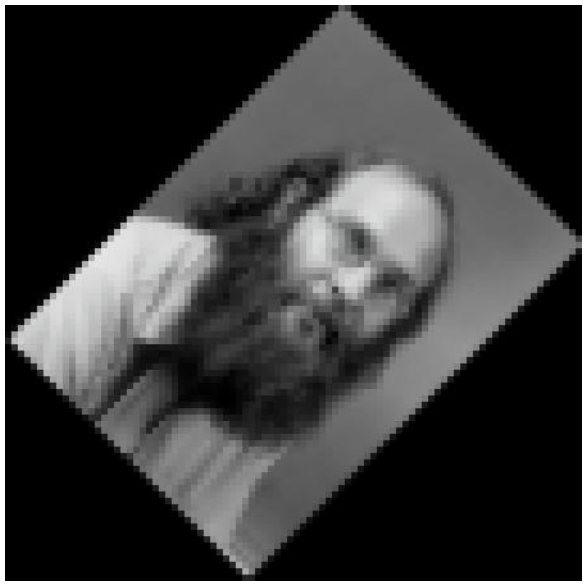
Trade-offs

- Jagged edges versus blurring
- Computation speed

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



Nearest



Bilinear

Gaussian

Outline

- Image Filtering
- Image Warping
- **Image Morphing**

Image Morphing

Animate transition between two images:

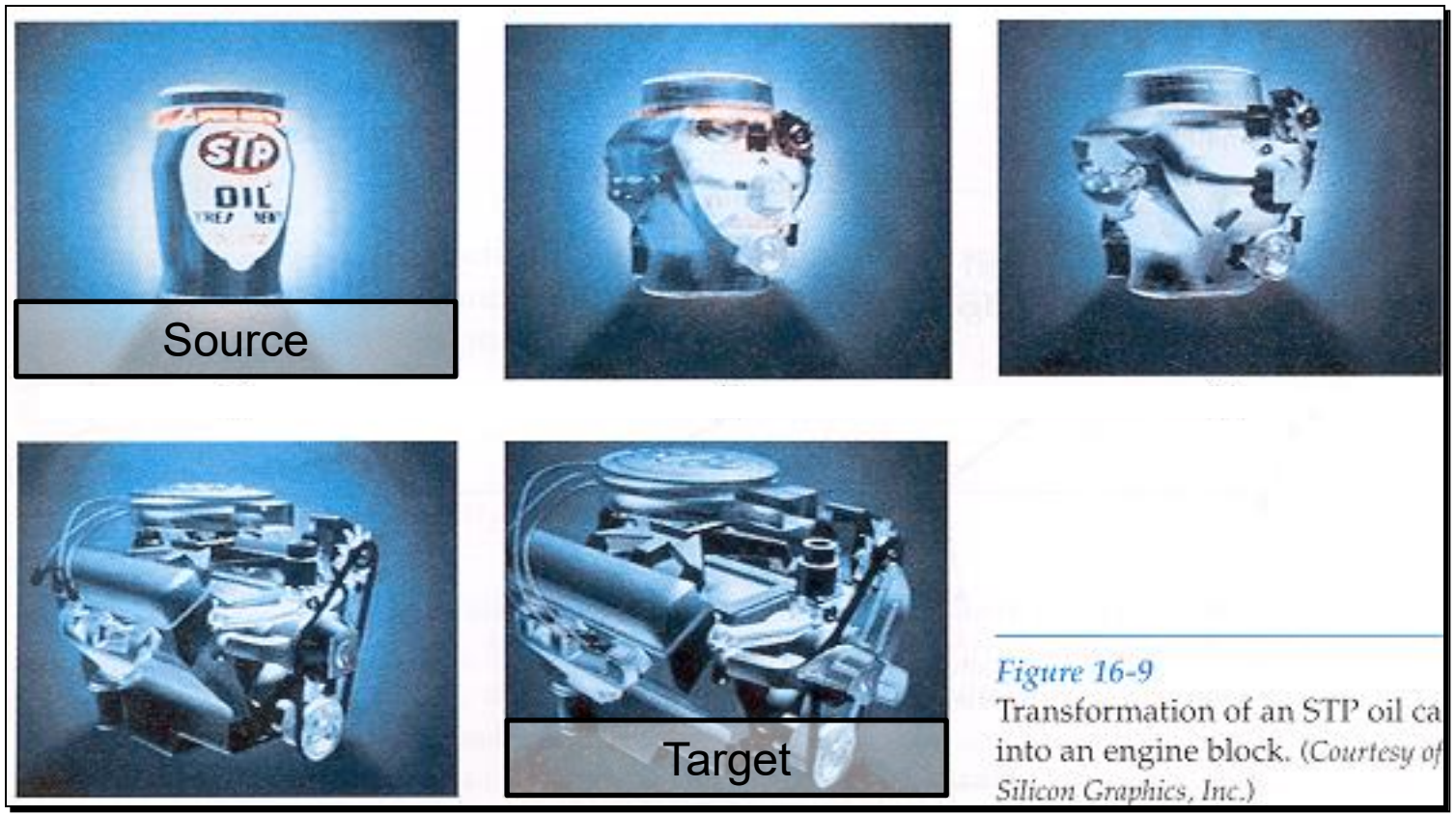
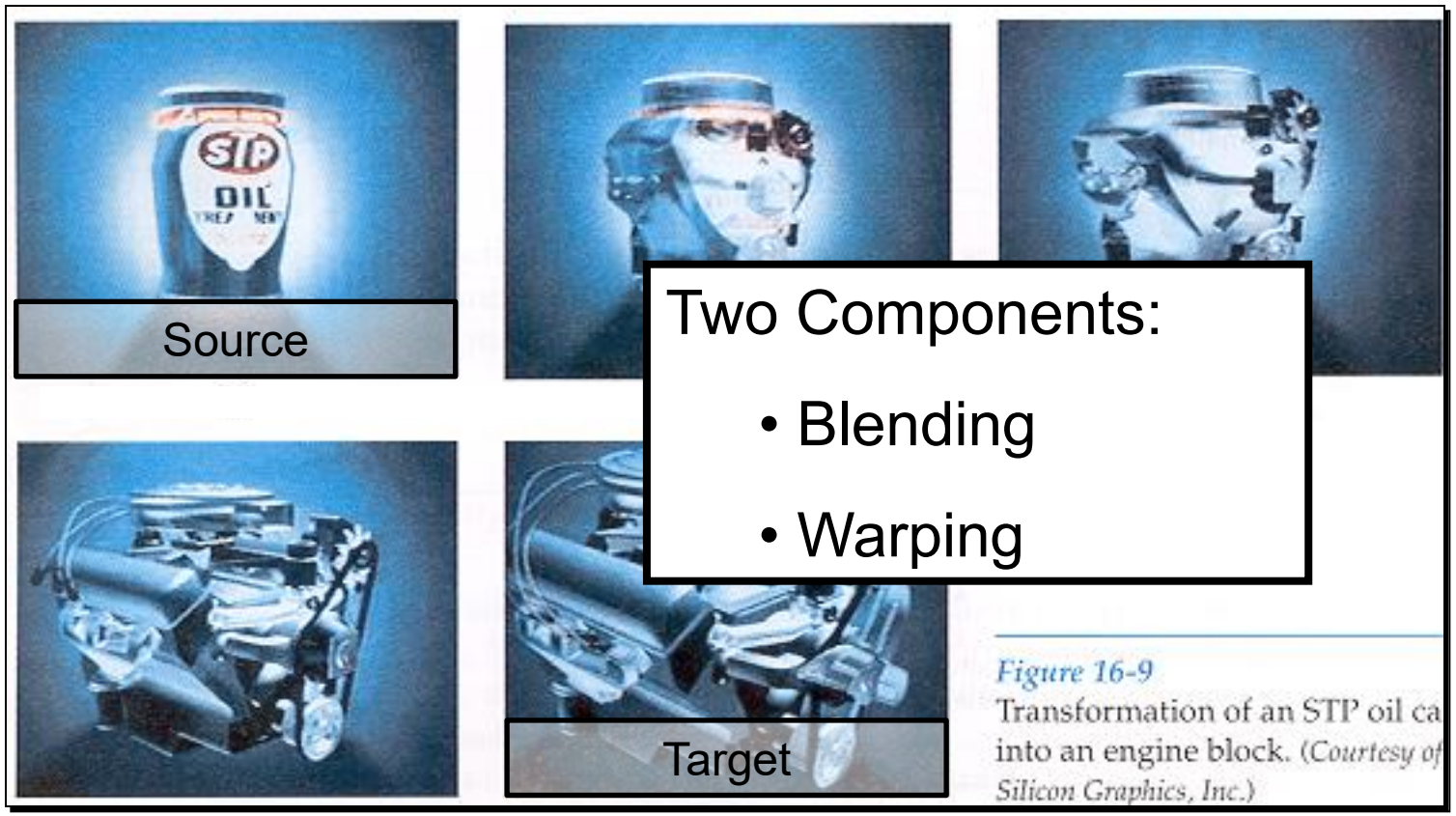


Figure 16-9
Transformation of an STP oil ca
into an engine block. (*Courtesy of*
Silicon Graphics, Inc.)

H&B Figure 16.9

Image Morphing

Animate transition between two images:



H&B Figure 16.9

Image Morphing

Recall (inner-product):

1. For 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:

$$\langle \vec{v}_1, \vec{v}_2 \rangle \equiv x_1 \cdot x_2 + y_1 \cdot y_2 = \|\vec{v}_1\| \cdot \|\vec{v}_2\| \cdot \cos\theta$$

⇒ Dividing by the length of v_2:

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

gives the (signed) length of the base of the triangle.

⇒ This is the projection of v_1 onto the line through v_2.

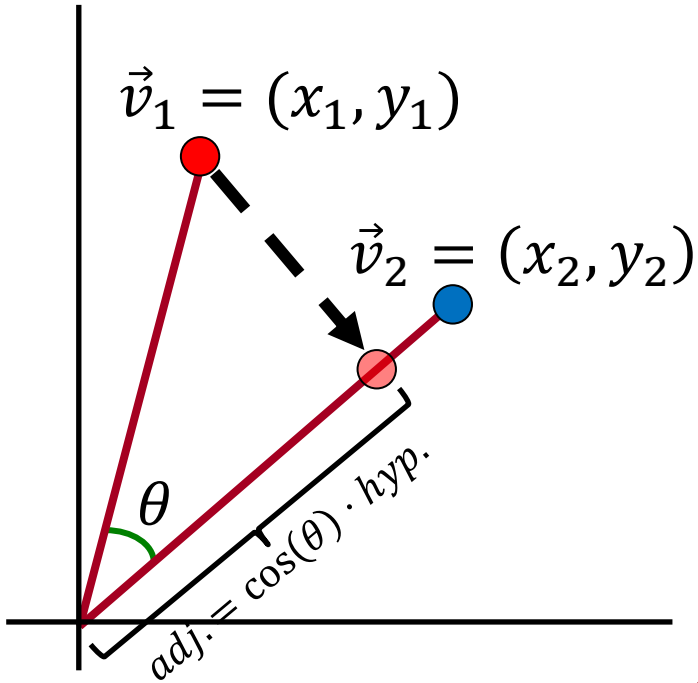


Image Morphing

Recall (inner-product):

1. For 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 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:
$$\langle \vec{v}_1^{\perp}, \vec{v}_2^{\perp} \rangle = \langle \vec{v}_1, \vec{v}_2 \rangle$$

Image Morphing: Blending

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

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

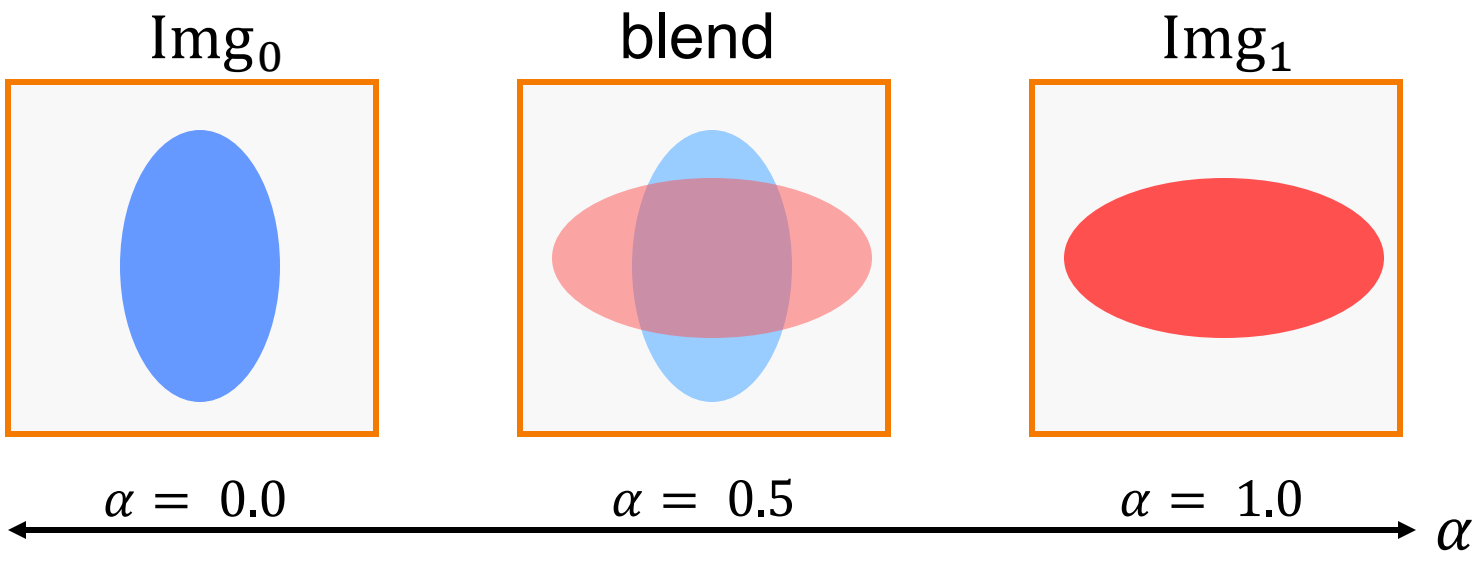
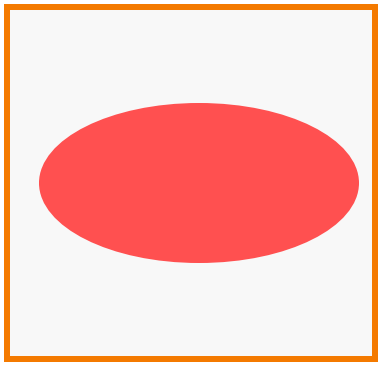


Image Morphing: Warping

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

Img_0



Img_1

$\alpha = 0.0$ $\alpha = 0.5$ $\alpha = 1.0$ α

Image Morphing: Warping

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

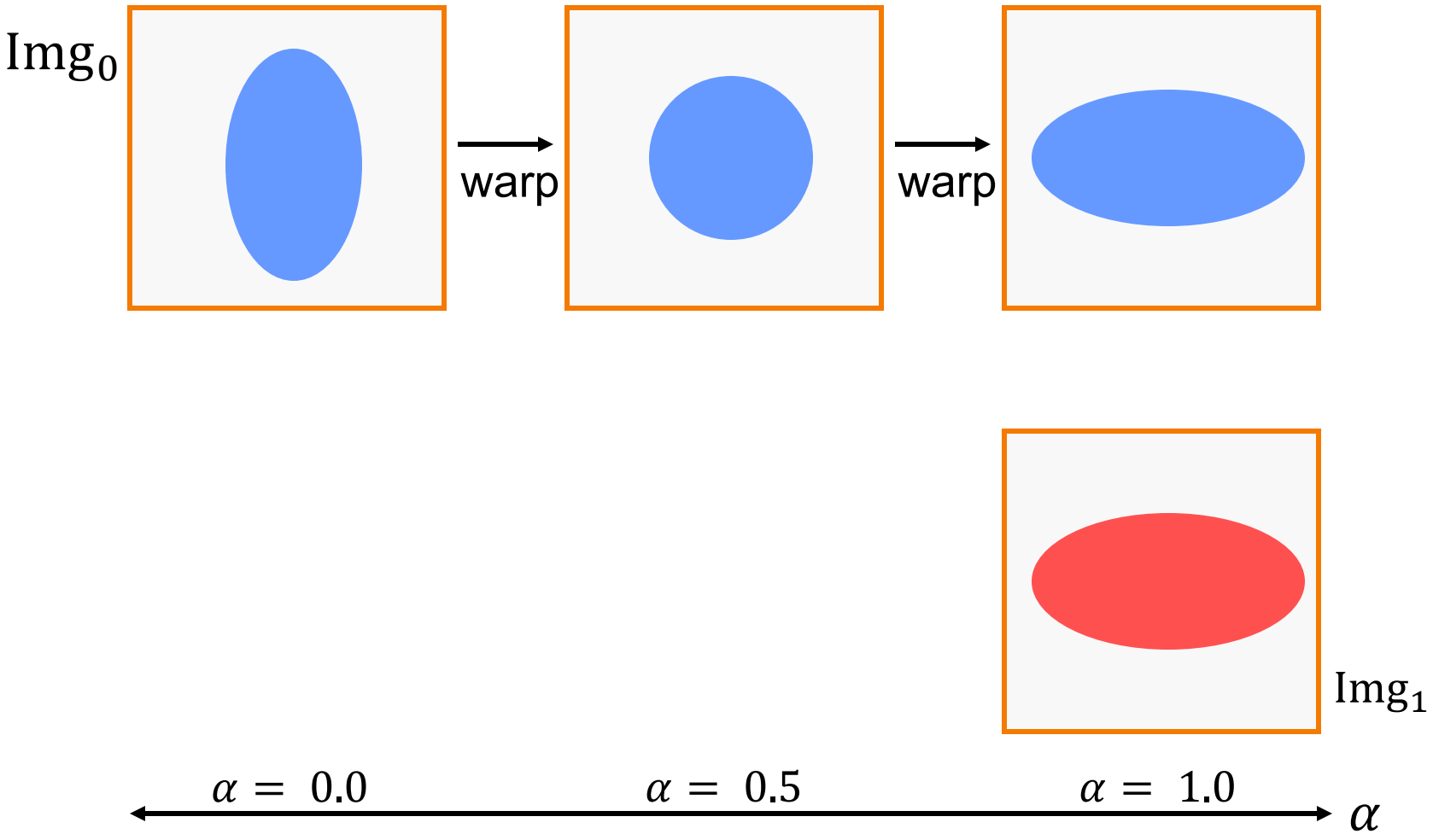


Image Morphing: Warping

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

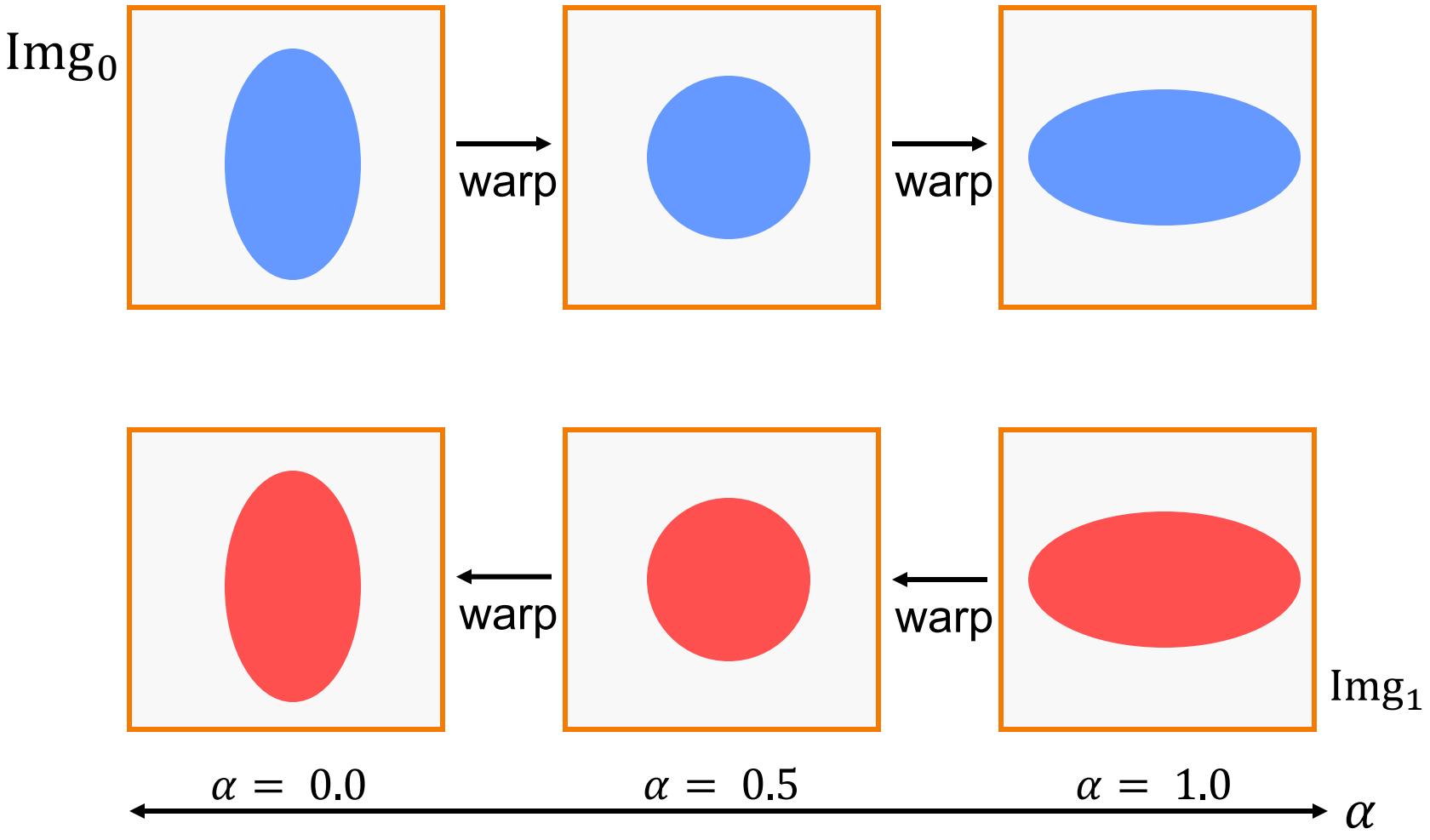


Image Morphing: Warping + Blending

… then blend **colors**

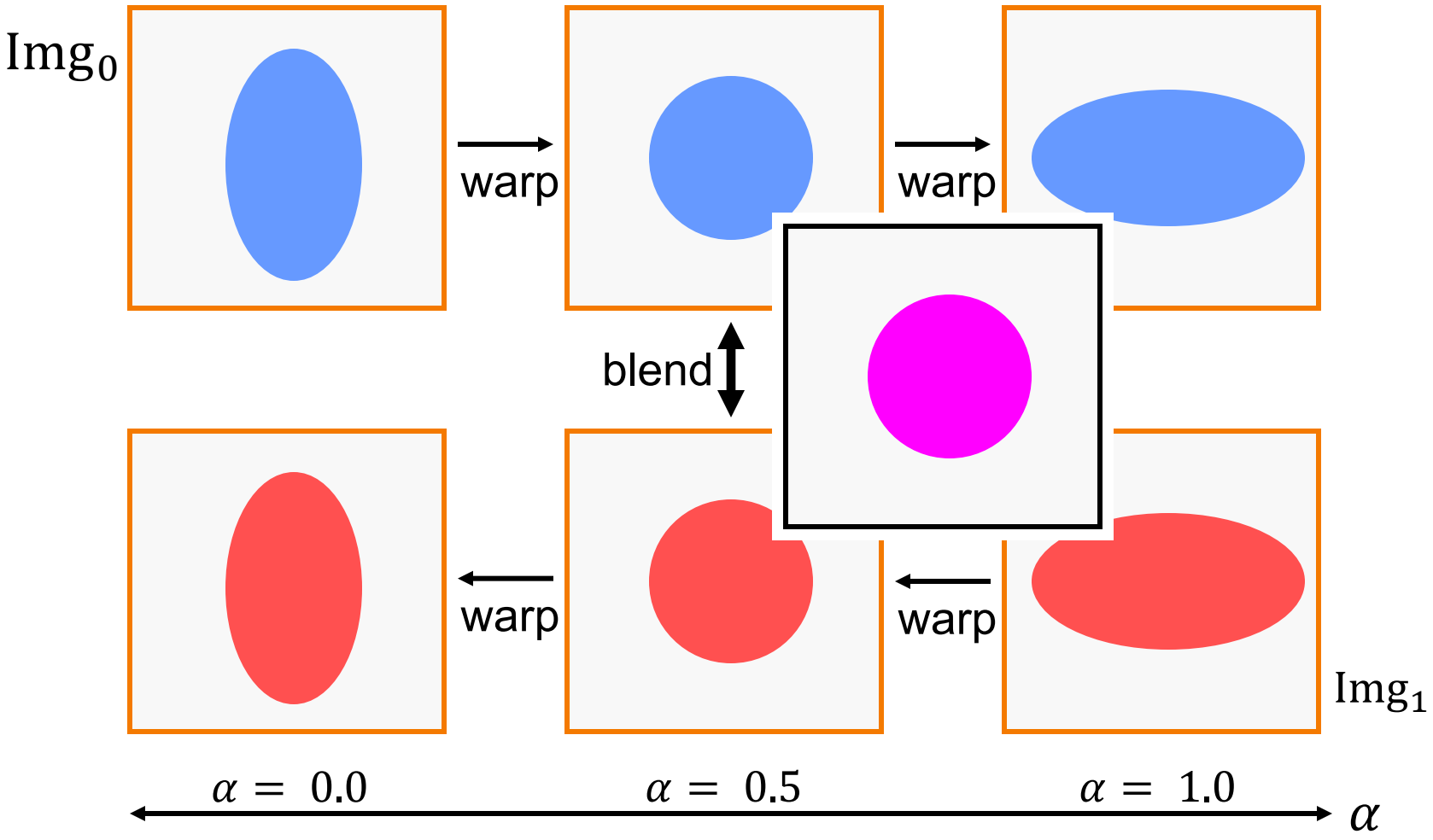
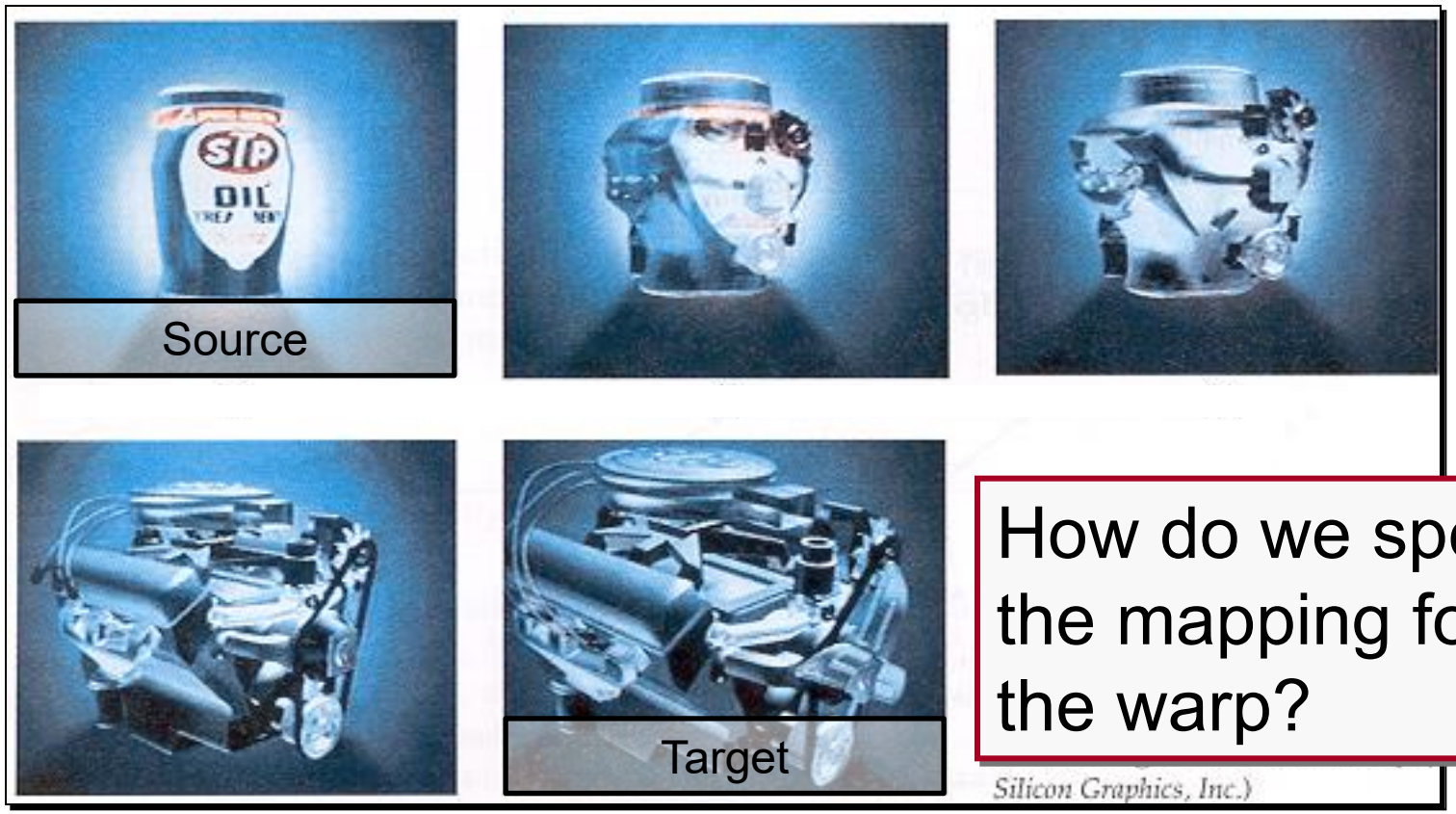


Image Morphing: Warping

The warping step is the hard one

- Aim to align features in images



How do we specify the mapping for the warp?

H&B Figure 16.9

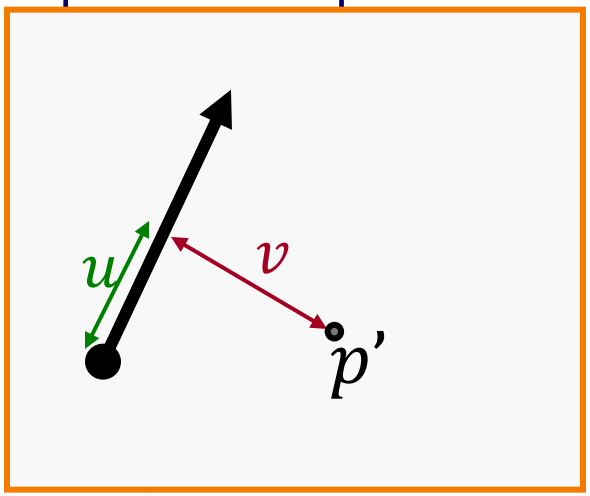
Feature-Based Warping [Beier & Neeley, 1992]

Challenge:

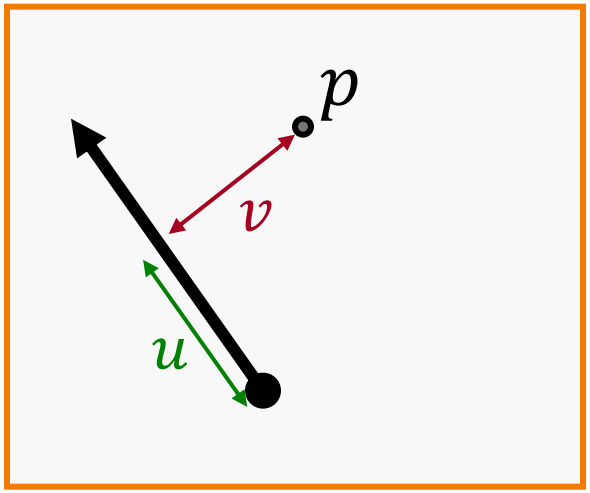
- Given p in the destination, what is the corresponding source location?

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:

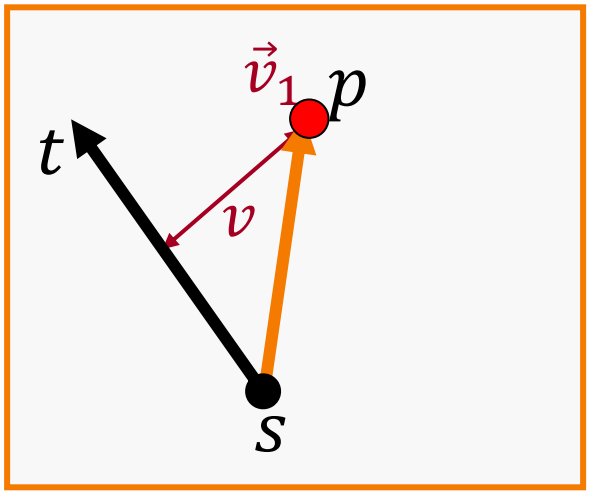
The signed length of $\vec{v}_1$ projected onto $\vec{v}_2$ is:

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

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:

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

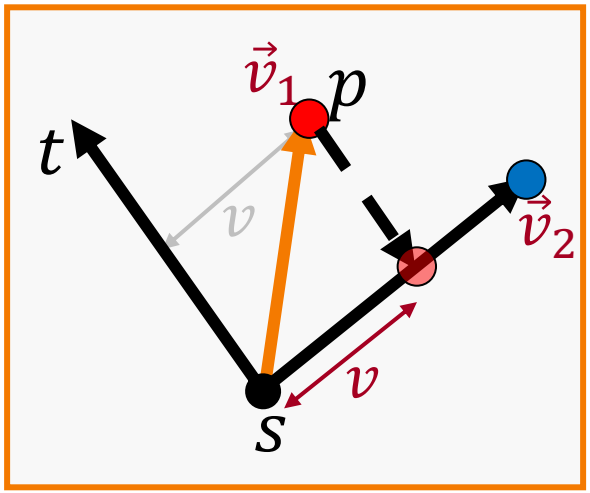
In our case we want:

- 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:

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

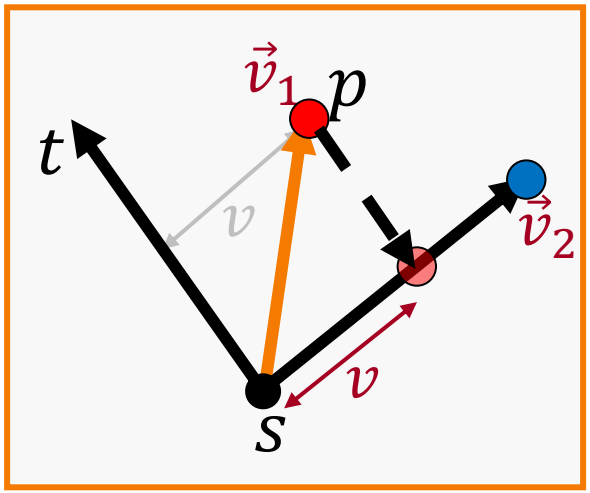


Feature-Based Warping [Beier & Neeley, 1992]

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

This gives:

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



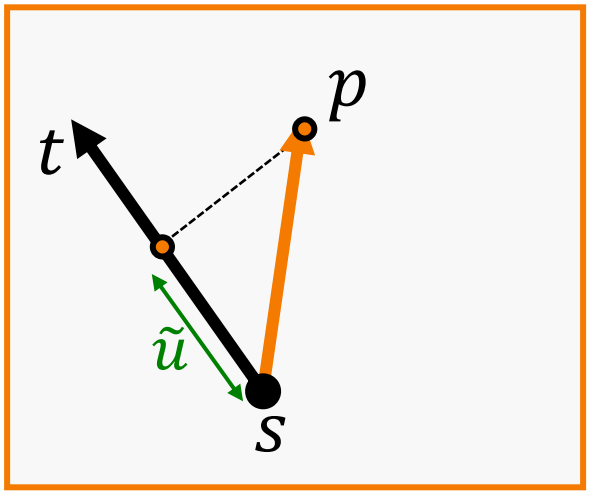
Feature-Based Warping [Beier & Neeley, 1992]

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

Similarly:

- The signed projected length is:

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



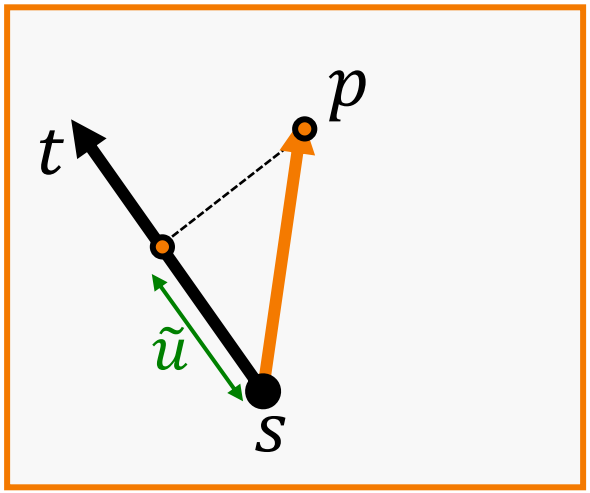
Feature-Based Warping [Beier & Neeley, 1992]

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

Similarly:

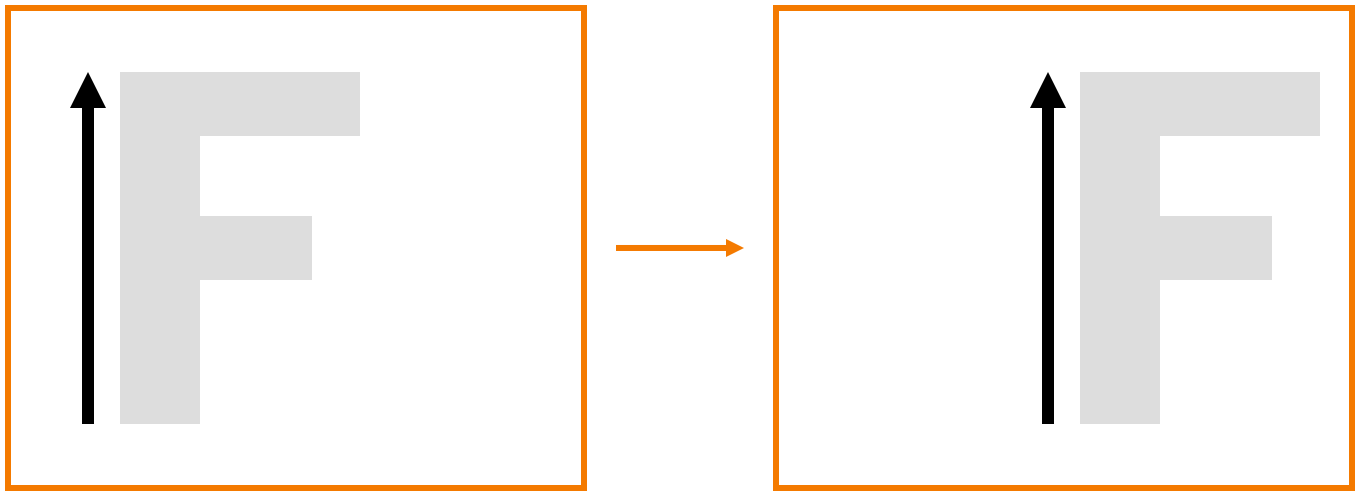
- The **fractional** signed projected length is obtained by dividing by the length of the segment:

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



Warping with One Line Pair

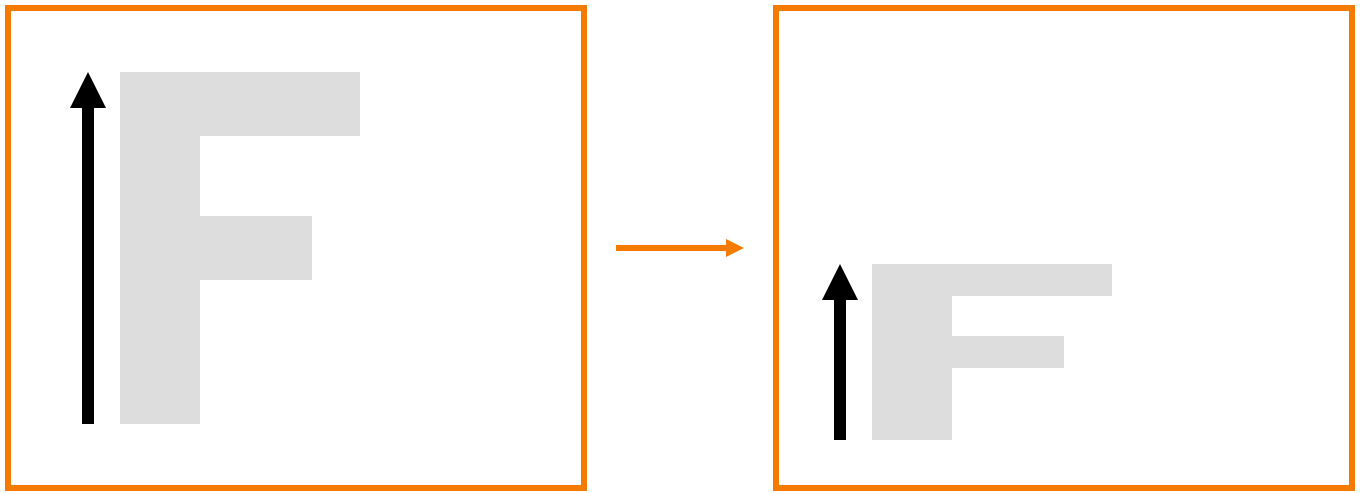
What happens to the “F”?



Translation!

Warping with One Line Pair

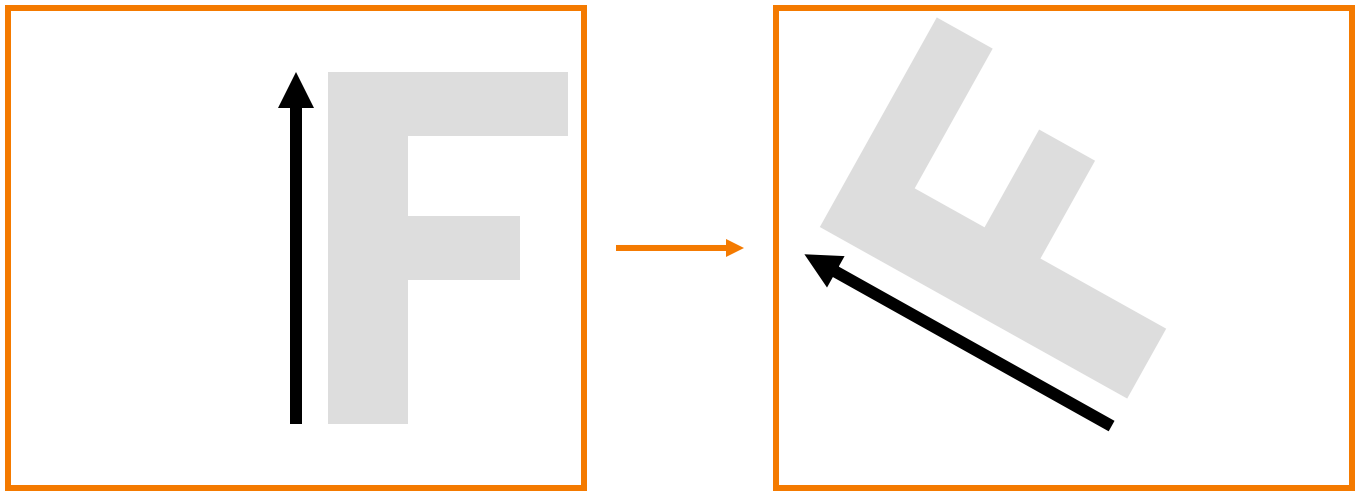
What happens to the “F”?



Non-uniform scale!

Warping with One Line Pair

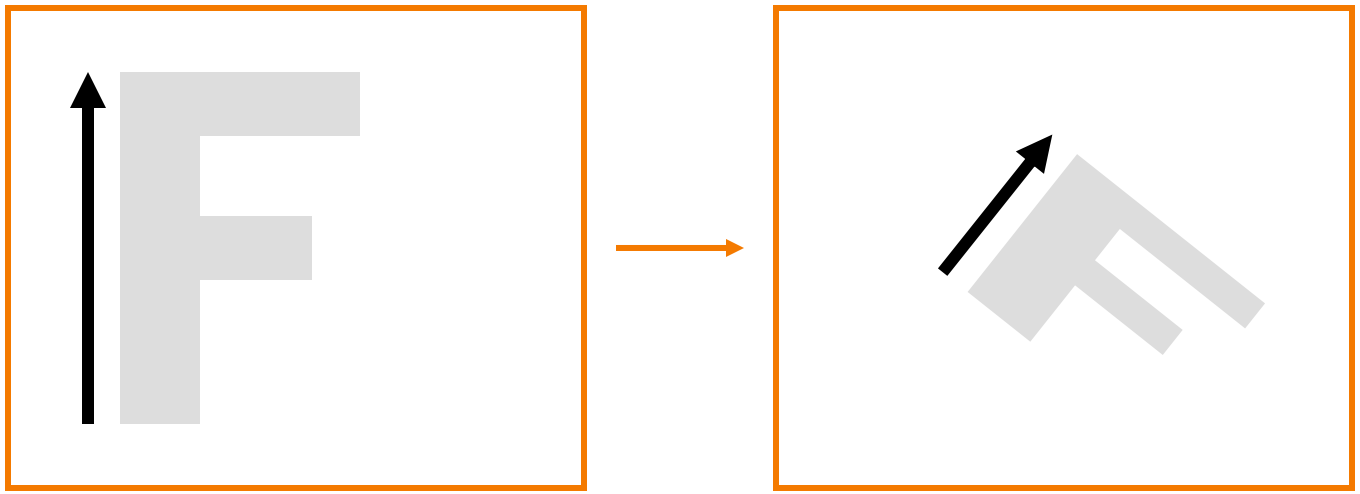
What happens to the “F”?



Rotation!

Warping with One Line Pair

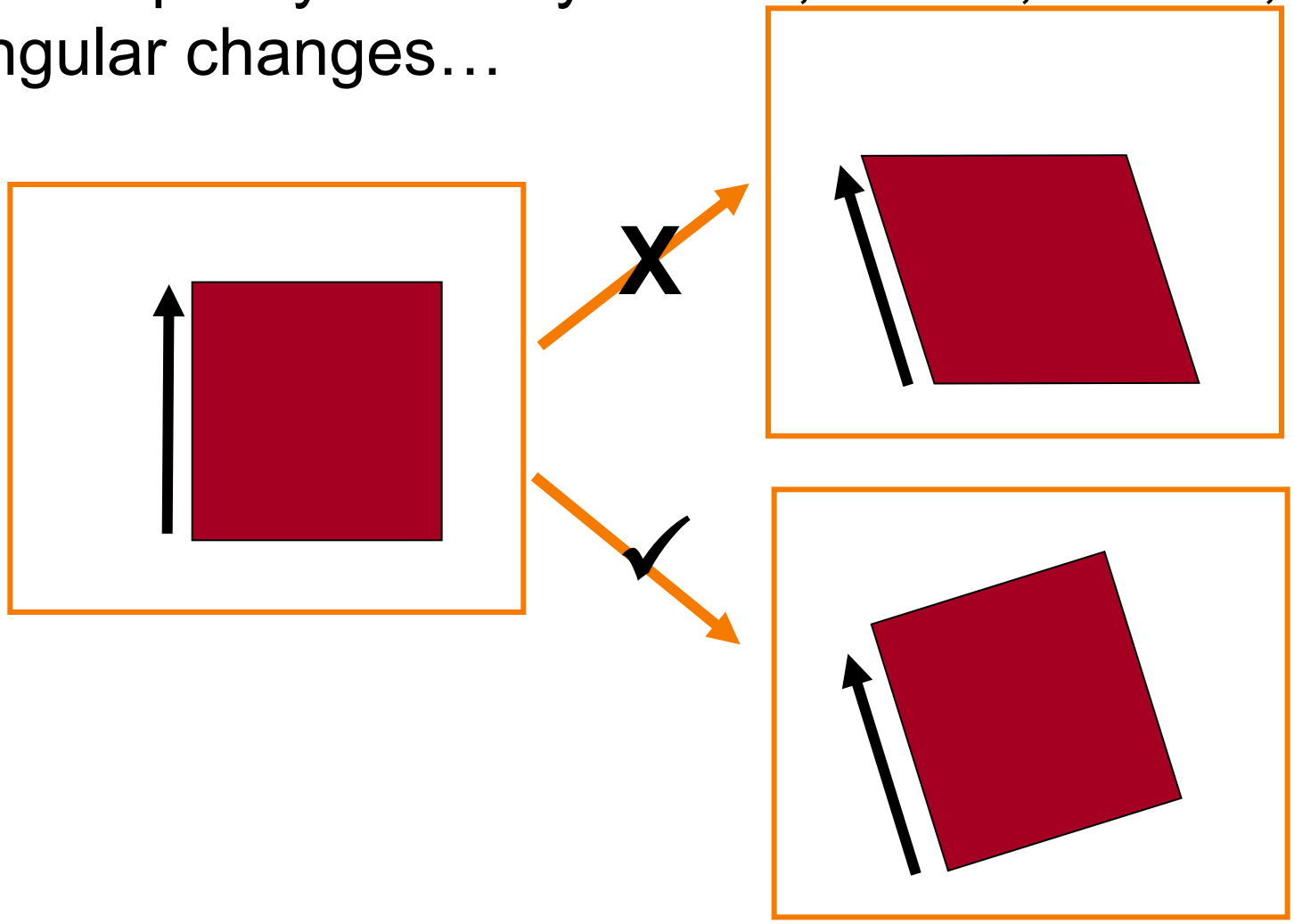
What happens to the “F”?



What types of affine transformations can’t be specified?

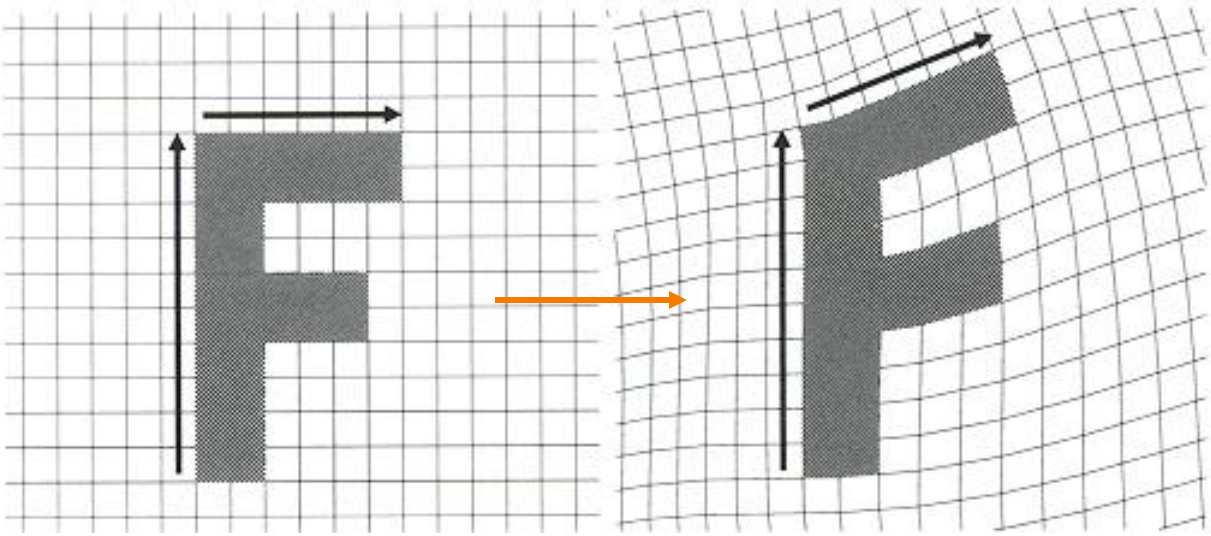
Warping with One Line Pair

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



Warping with Multiple Line Pairs

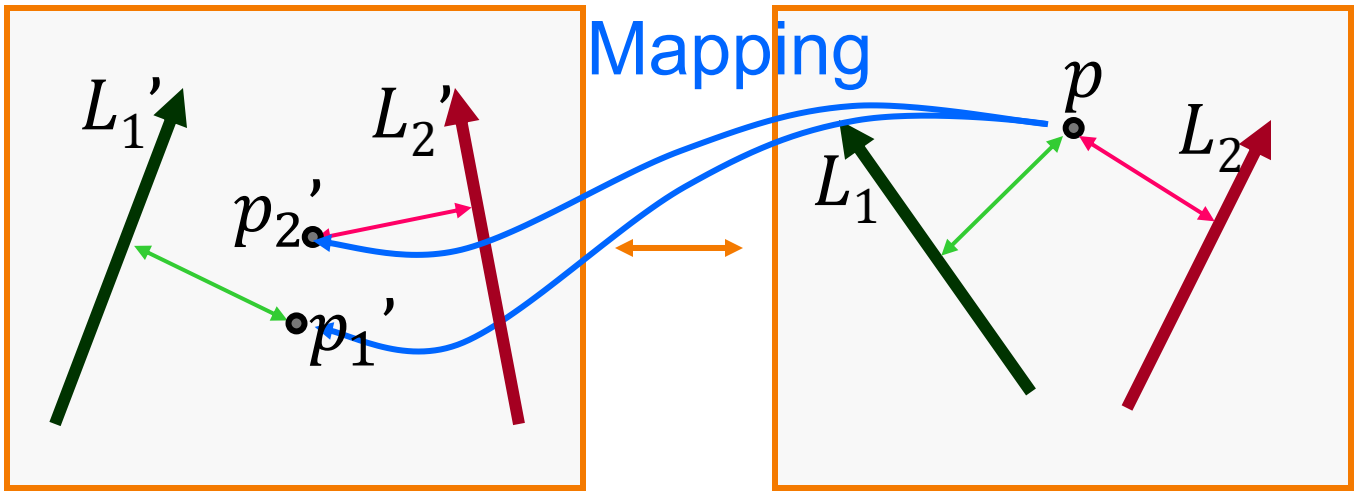
Use weighted combination of points defined by each pair of corresponding lines



Beier & Neeley, Figure 4

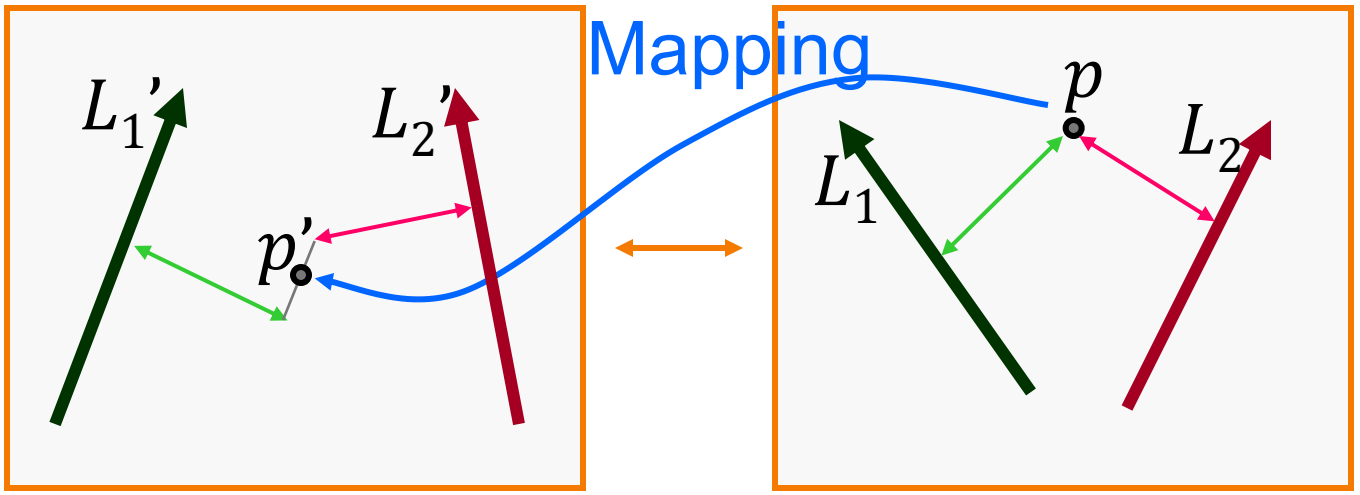
Warping with Multiple Line Pairs

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



Source image

Destination image

p' is a weighted average

Weighting Effect of Each Line Pair

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

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

- $\text{length}[i]$ 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

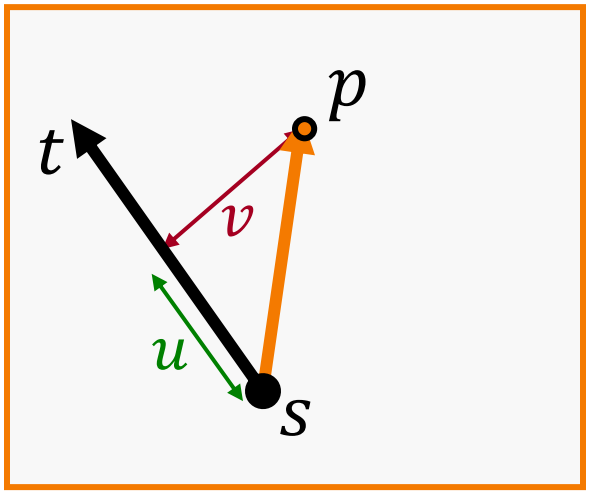
Note: The "~" indicates "up to a constant". Need to normalize so weights sum to 1.

Feature-Based Warping

- $\text{dist}[i](p)$ is the distance from p to $L_{out}[i]$

How do we calculate the distance from a point p to the nearest point on 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 the source and target

Goal:

- Warp the source image to the target

Solution:

- Iterate 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 source position
  - Sample the source (at the averaged position) to get the target color

Warping

Input:

- Source image
- Set of correspon

Goal:

- Warp the source

```
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
```

Solution:

Iterate 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 source position
- Sample the source (at the averaged position) to get the target color

Morphing

Input:

- Two source images
- Set of corresponding line segment pairs in the sources
- Interpolation time $\alpha \in [0,1]$

Goal:

- The morph at time α

Solution:

- α-average the end-points of the segments from the two sources to get the target segments
- **Warp** the 1st image using the 1st source and target segments
- **Warp** the 2nd image using the 2nd source and target segments

⇒ Gives the source images warped to the same target segments

- Compute the α-**blend** of the warped images

Morphing

Input:

- Two source ima
- Set of correspon
- Interpolation time $\alpha \in [0,1]$

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

Goal:

- The morph at time α

Solution:

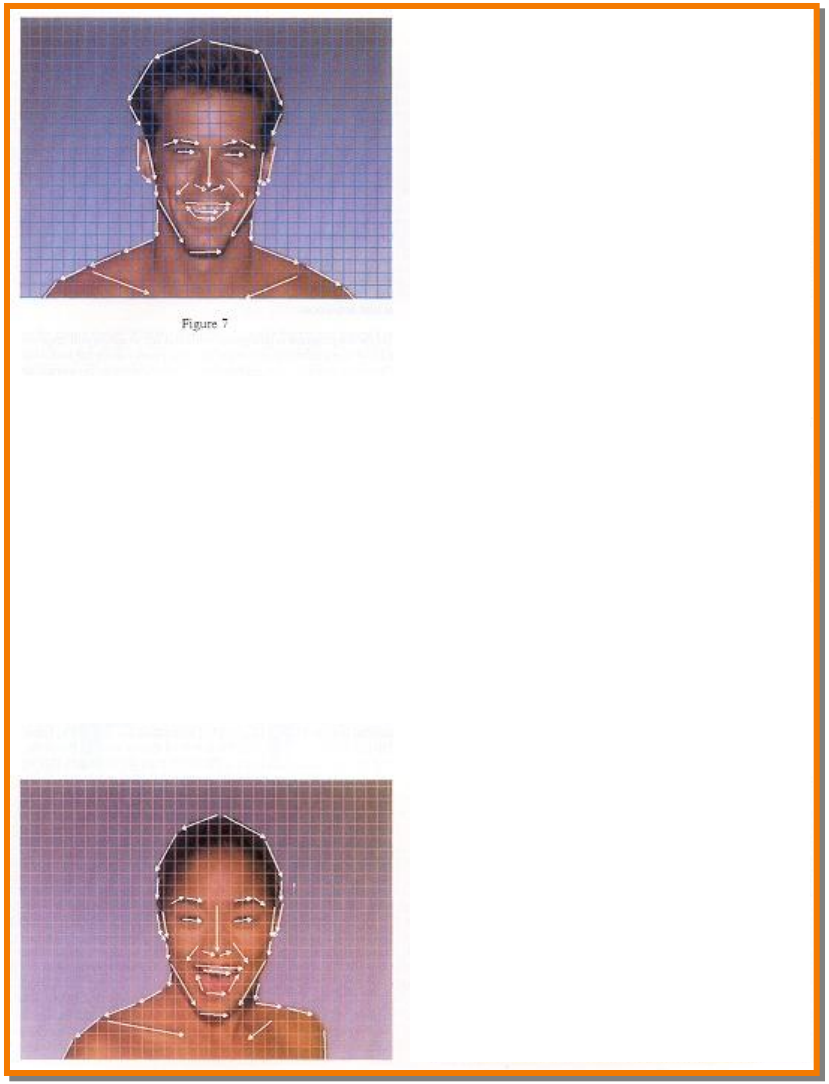
- α-average the end-points of the segments from the two sources to get the target segments
- **Warp** the 1st image using the 1st source and target segments
- **Warp** the 2nd image using the 2nd source and target segments

⇒ Gives the source images warped to the same target segments

- Compute the α-**blend** of the warped images

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

Img_0



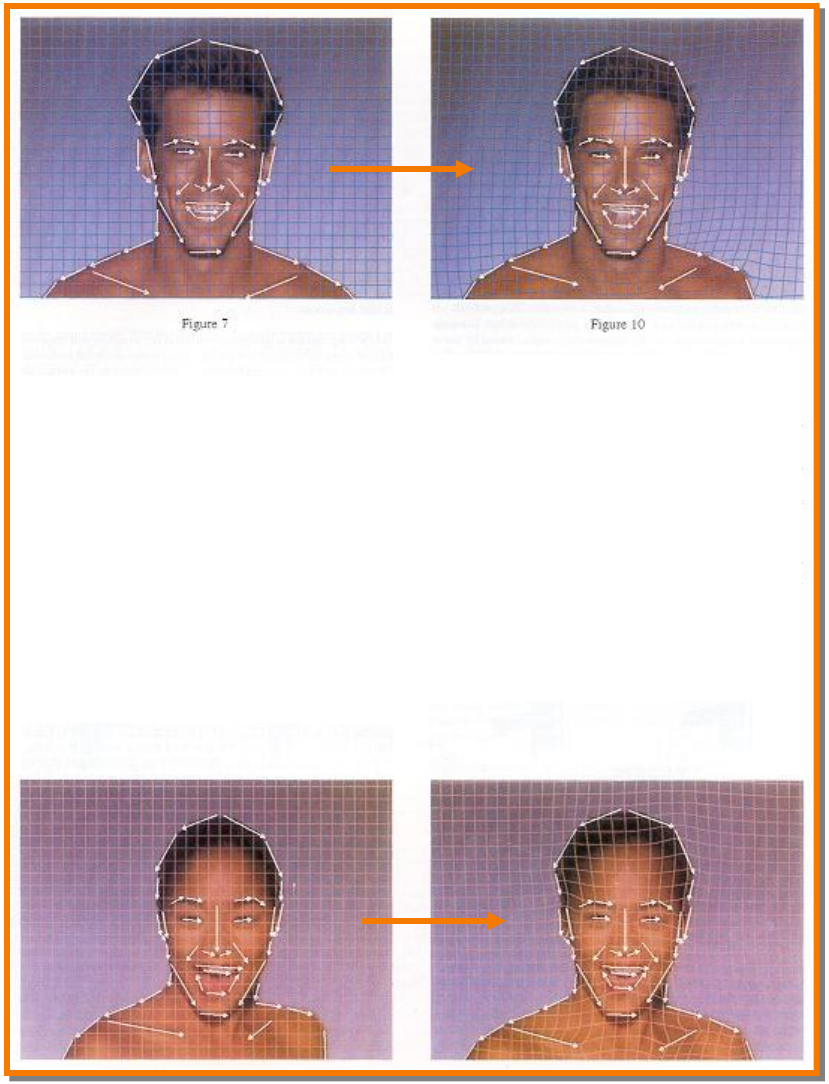
$Warp_0$

Img_1

$Warp_1$

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

Img_0

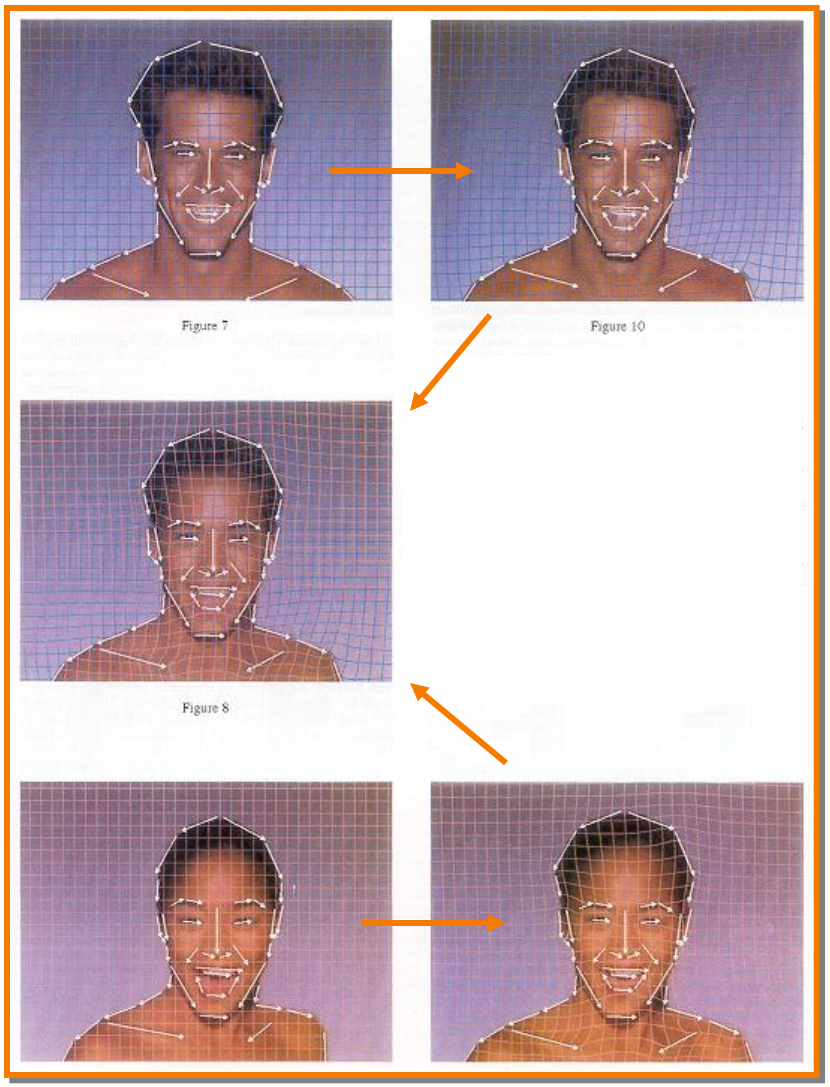


$Warp_0$

Img_1

$Warp_1$

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



Img_0

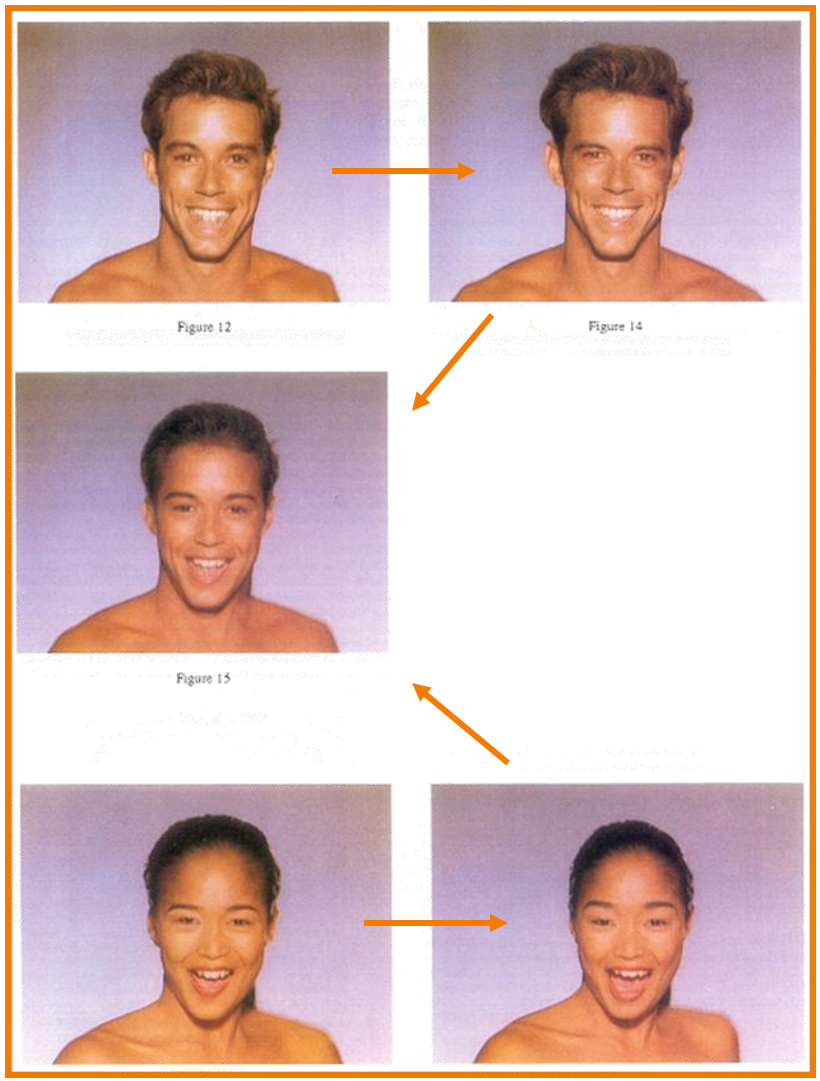
$Warp_0$

Result

Img_1

$Warp_1$

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



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