# Final Project

- 依據Michael Jackson的Black & White影片 ，實作全班同學的人臉轉換 Morphing Video.
- 作業要求
  - 拍攝上、下、左、右轉身動作影片上傳Youtube
  - Morphing自己的影片 & 下一位學號順序同學的影片
- 程式要求: 請以Matlab實作下列功能
  - 特徵點標記 (使用Label Me)
  - 計算三角片mesh
  - 對每一個三角片做image warping，完成整張影像的image morphing
  - 對於兩段影片進行morphing
  - 將morphing結果製作動畫 (Matlab)

# Final Project

- 繳交時間:
  - 12/18(三)上傳個人影片
    - 背景白色牆面
    - 肩膀頸部區間保持肉色
    - Follow Black & White影片動作
  - 01/13(一) 程式、報告與成果繳交

# Affine Transformations

# Affine Transformations

| Name | Matrix | # D.O.F. | Preserves: | Icon |
|---|---|---|---|---|
| translation | $\begin{bmatrix} \boldsymbol{I} \mid \boldsymbol{t} \end{bmatrix}_{2\times 3}$ | 2 | orientation $+ \cdots$ | |
| rigid (Euclidean) | $\begin{bmatrix} \boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times 3}$ | 3 | lengths $+ \cdots$ | |
| similarity | $\begin{bmatrix} s\boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times 3}$ | 4 | angles $+ \cdots$ | |
| affine | $\begin{bmatrix} \boldsymbol{A} \end{bmatrix}_{2\times 3}$ | 6 | parallelism $+ \cdots$ | |
| projective | $\begin{bmatrix} \tilde{\boldsymbol{H}} \end{bmatrix}_{3\times 3}$ | 8 | straight lines | |

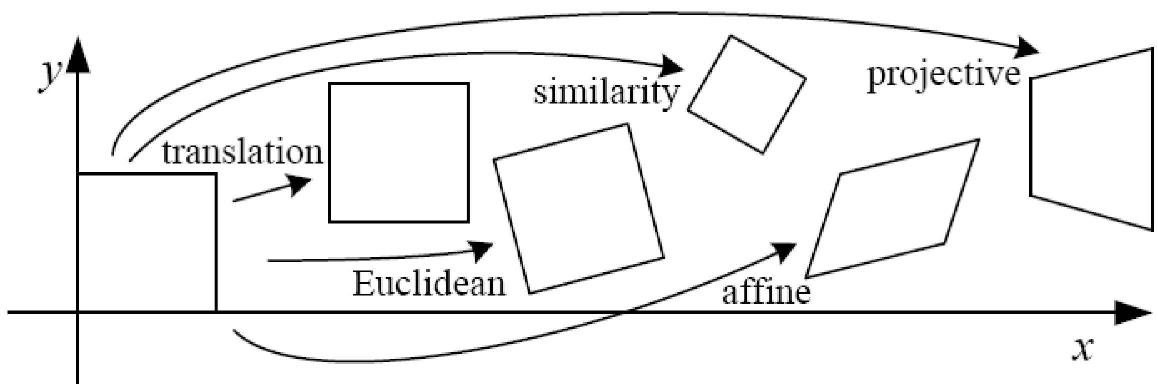



Figure 2.4: *Basic set of 2D planar transformations*

# Geometric transformations

- Geometric transformations will map points in one space to points in another: $(x',y',z') = \boldsymbol{f}(x,y,z)$.
- These transformations can be very simple, such as scaling each coordinate, or complex, such as non-linear twists and bends.
- We'll focus on transformations that can be represented easily with matrix operations.
- We'll start in 2D...

# Representation

- We can represent a **point**, **p** = (x,y), in the plane
  - as a column vector $$\begin{bmatrix} x \\ y \end{bmatrix}$$
  - as a row vector $$\begin{bmatrix} x & y \end{bmatrix}$$

# Representation, cont.

- We can represent a **2-D transformation M** by a matrix

$$\mathbf{M} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

- If **p** is a column vector, *M* goes on the left:

$$\mathbf{p}' = \mathbf{M}\mathbf{p}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- If **p** is a row vector, $M^T$ goes on the right:

$$\mathbf{p}' = \mathbf{p}\mathbf{M}^{\mathrm{T}}$$

$$\begin{bmatrix} x' & y' \end{bmatrix} = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

- We will use **column vectors**.

# Two-dimensional transformations

- Here's all you get with a 2 x 2 transformation matrix **M**:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

- So:

$$x' = ax + by$$
$$y' = cx + dy$$

- We will develop some intimacy with the elements *a*, *b*, *c*, *d*…

# Identity

- Suppose we choose *a=d=1, b=c=0:*
  - Gives the **identity** matrix:

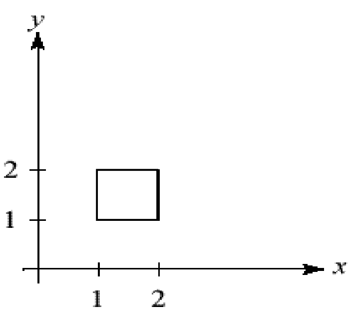
$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

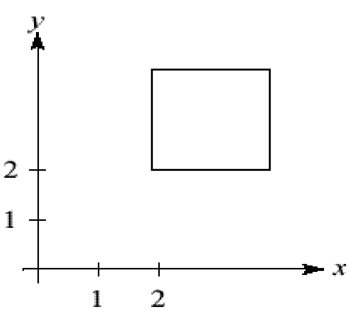
$$x' = ax + by$$
$$y' = cx + dy$$

  - Doesn't move the points at all

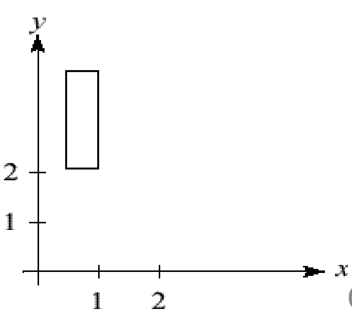
# Scaling

- Suppose *b=c=0*, but let *a* and *d* take on any *positive* value:
  - Gives a **scaling** matrix: $\begin{bmatrix} a & 0 \\ 0 & d \end{bmatrix}$
  - Provides **differential (non-uniform) scaling** in *x* and *y*:




$$\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

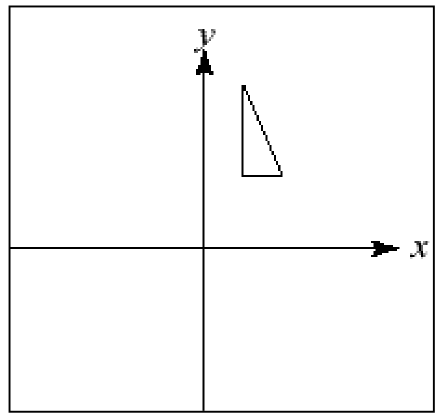



$$\begin{bmatrix} 1/2 & 0 \\ 0 & 2 \end{bmatrix}$$

$$x' = ax$$
$$y' = dy$$

# Reflection

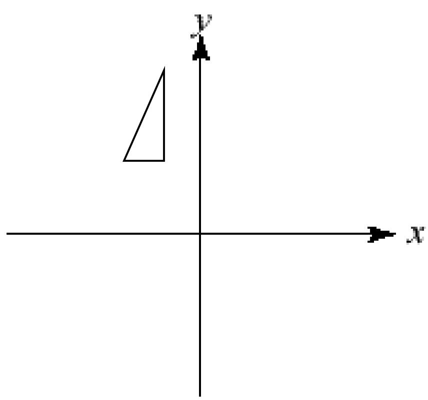
- Suppose *b=c=0*, but let either *a* or *d* go negative.
- Examples:

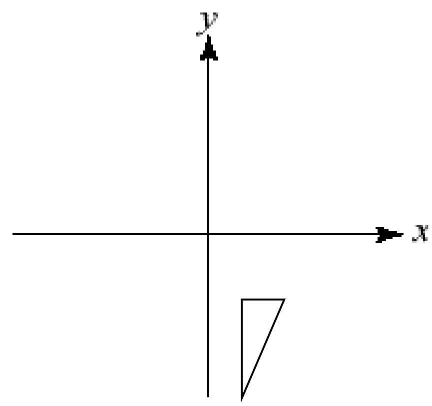



$$x' = x$$

$$y' = -y$$

$$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$$




$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

# Shear

- Now leave *a=d=1* and experiment with *b*
- The matrix

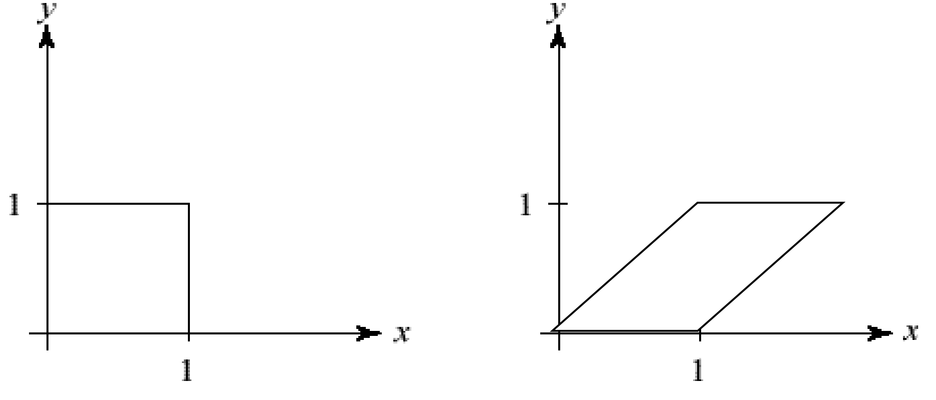
$$\begin{bmatrix} 1 & b \\ 0 & 1 \end{bmatrix}$$

gives:

$$x' = x + y$$

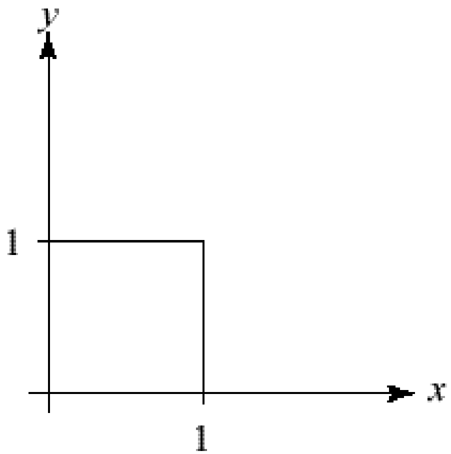
$$y' = y$$

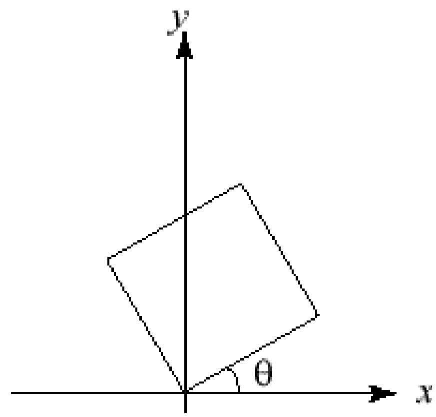
$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$



# Rotation

- From our observations of the effect on the unit square, it should be easy to write down a matrix for "rotation about the origin":




$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix}$$

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix} \rightarrow \begin{bmatrix} -\sin(\theta) \\ \cos(\theta) \end{bmatrix}$$

Thus $$M_R = R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

# Linear transformations

- The unit square observations also tell us the 2x2 matrix transformation implies that we are representing a point in a new coordinate system:

$$\mathbf{p}' = \mathbf{M}\mathbf{p} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \mathbf{u} & \mathbf{v} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = x \cdot \mathbf{u} + y \cdot \mathbf{v}$$

- where **u**=[$a$ $c$]$^T$ and **v**=[$b$ $d$]$^T$ are vectors that define a new **basis** for a **linear space**.
- The transformation to this new basis is a **linear transformation**.

# Limitations of the 2 x 2 matrix

- A 2 x 2 linear transformation matrix allows
  - Scaling
  - Rotation
  - Reflection
  - Shearing

$$\mathbf{p}' = \mathbf{M}\mathbf{p}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

# Affine transformations

- In order to incorporate the idea that both the basis and the origin can change, we augment the linear space **u**, **v** with an origin **t**.
- Note that while **u** and **v** are **basis vectors**, the origin **t** is a **point**.
- We call **u**, **v**, and **t** (basis and origin) a **frame** for an **affine space**.
- Then, we can represent a change of frame as:

$$\mathbf{p}' = x \cdot \mathbf{u} + y \cdot \mathbf{v} + \mathbf{t}$$

- This change of frame is also known as an **affine transformation**.

# Homogeneous Coordinates

- To represent transformations among affine frames, we can loft the problem up into 3-space, adding a third component to every point:

$$\mathbf{p}' = x \cdot \mathbf{u} + y \cdot \mathbf{v} + 1 \cdot \mathbf{t} = \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{t} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

- How to write the linear system? How many corresponding pairs we need to solve the linear system?

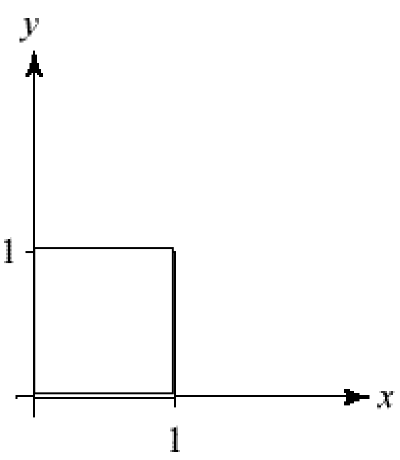
# Homogeneous coordinates

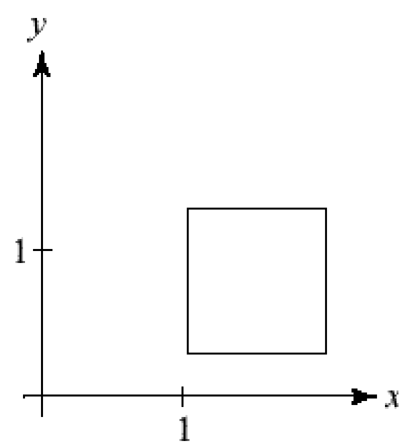
This allows us to perform translation as well as the linear transformations as a matrix operation:

$$\mathbf{p}' = \mathbf{M}_{\mathbf{T}}\mathbf{p} \qquad \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$x' = x + t_x$$

$$y' = y + t_y$$



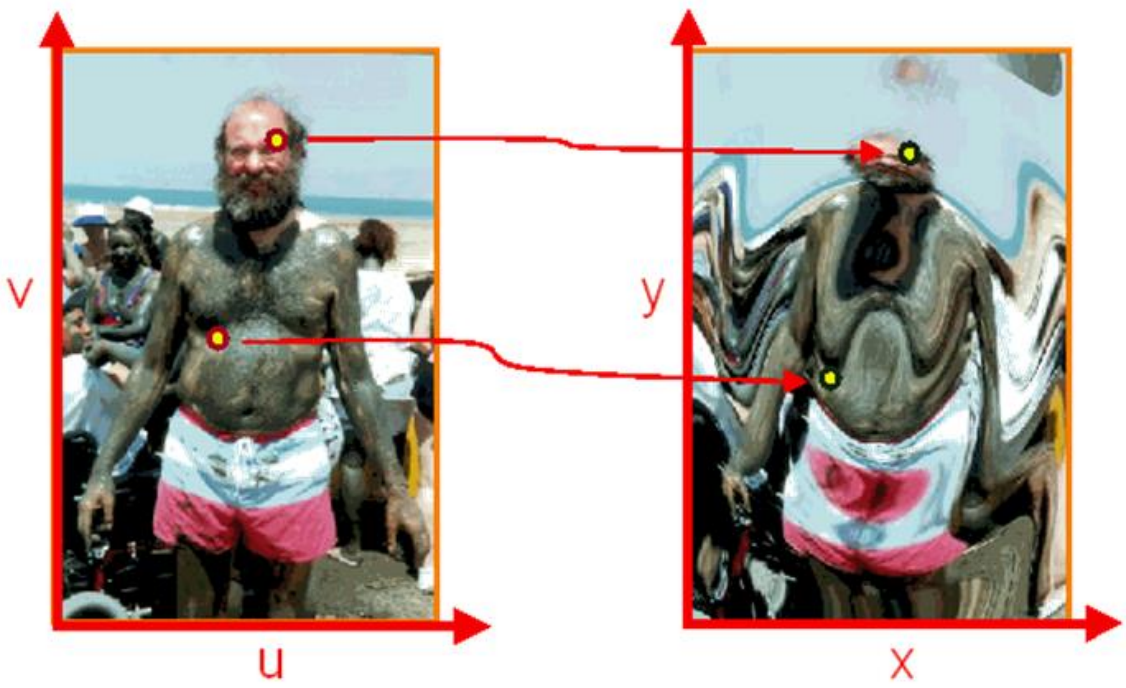




$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1/2 \\ 0 & 0 & 1 \end{bmatrix}$$

# Image Warping

# Image Warping

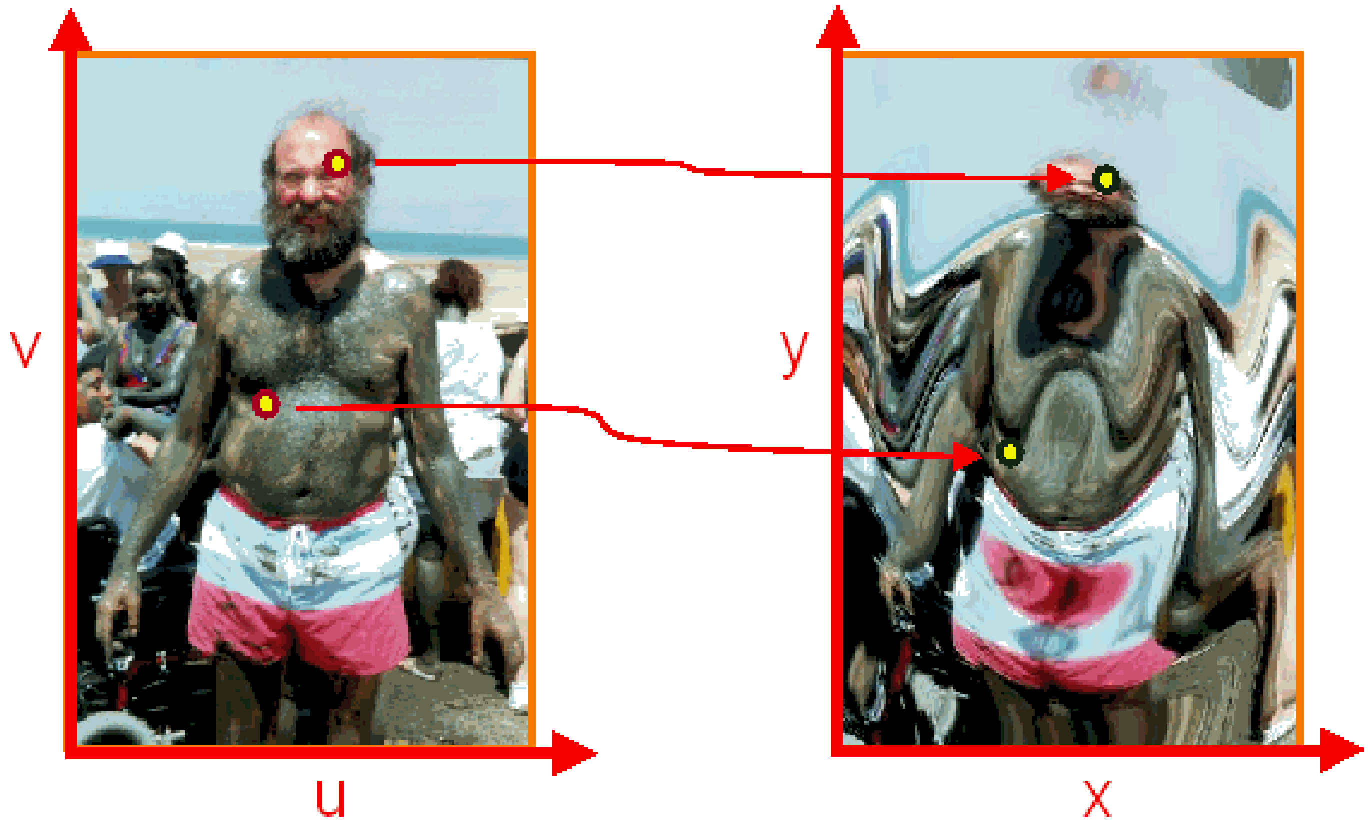
- Moving pixels of image
  - Mapping
  - Resampling




source image destination image

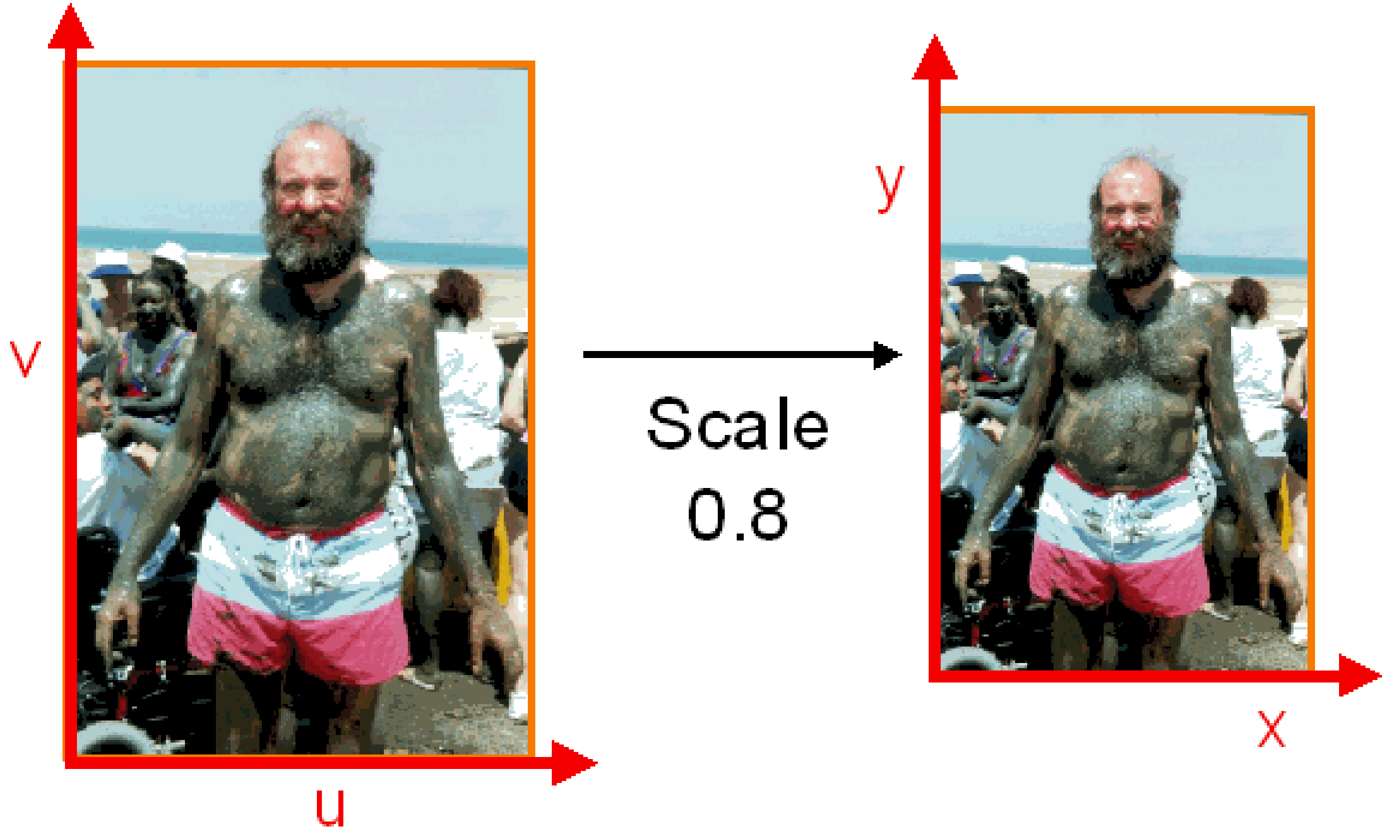
# Mapping

- Define transformation
  - Describe the destination (x,y) for every location (u,v) in the source (or vice-versa, if invertible)



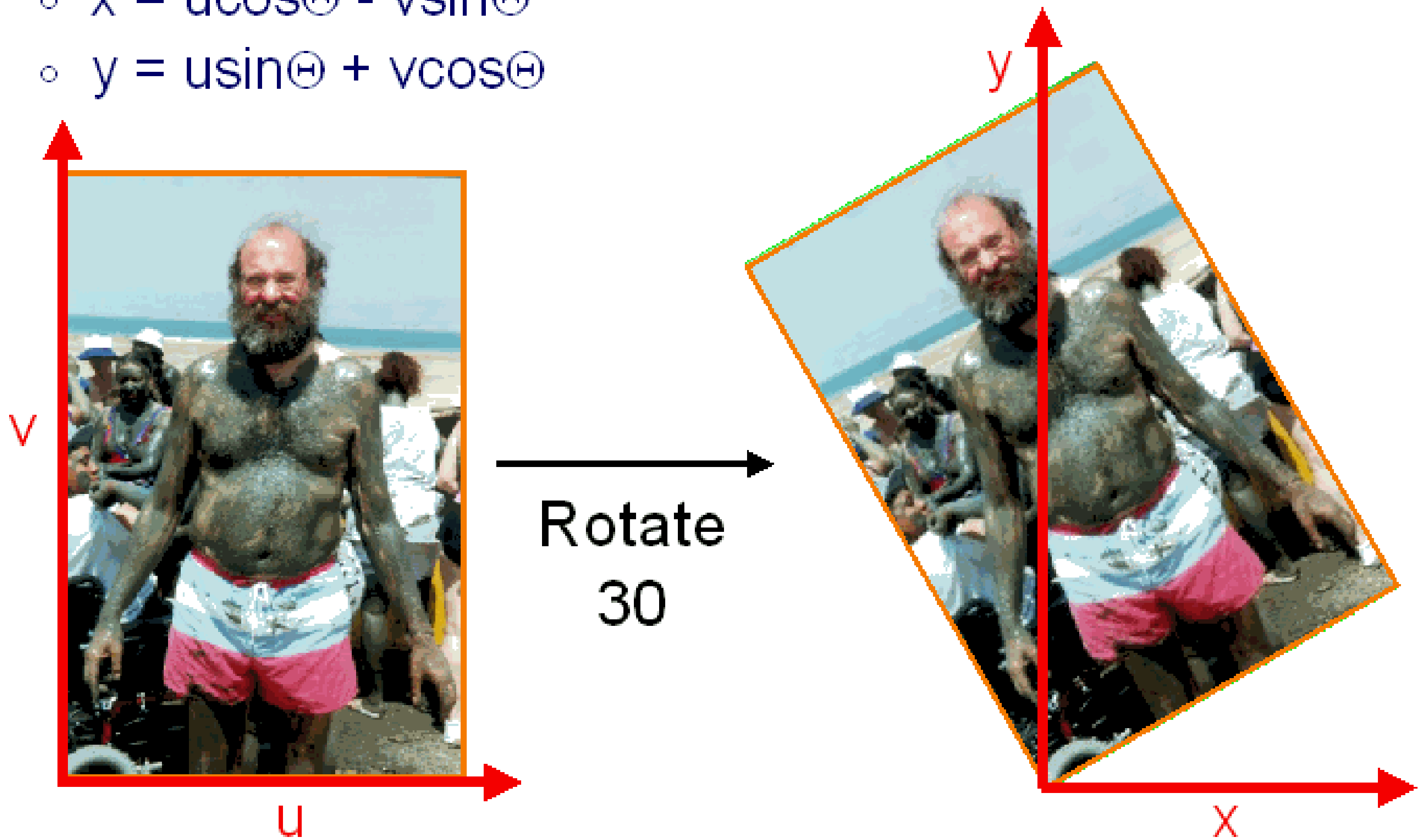
# Example Mappings

- Scale by *factor*:
  - x = *factor* * u
  - y = *factor* * v



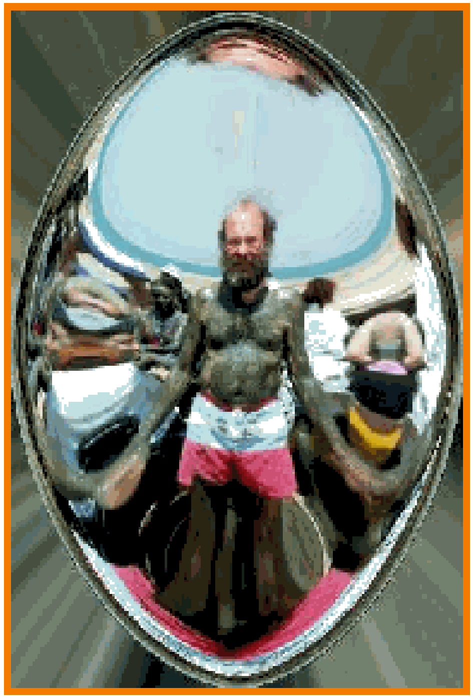
# Example Mappings

- Rotate by $\Theta$ degrees:
  - $x = u\cos\Theta - v\sin\Theta$
  - $y = u\sin\Theta + v\cos\Theta$

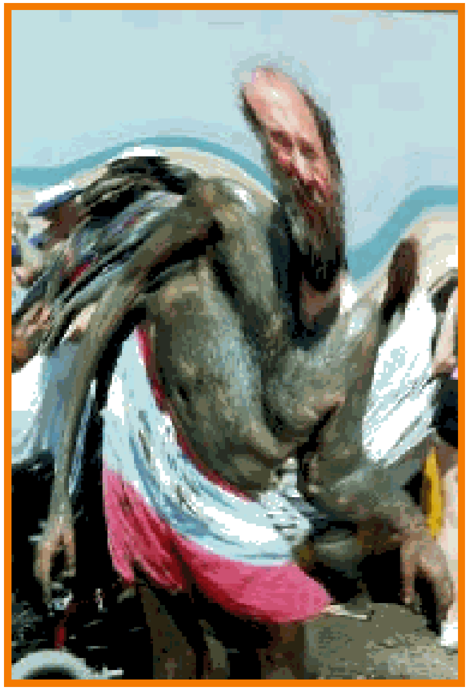


# Other Mappings

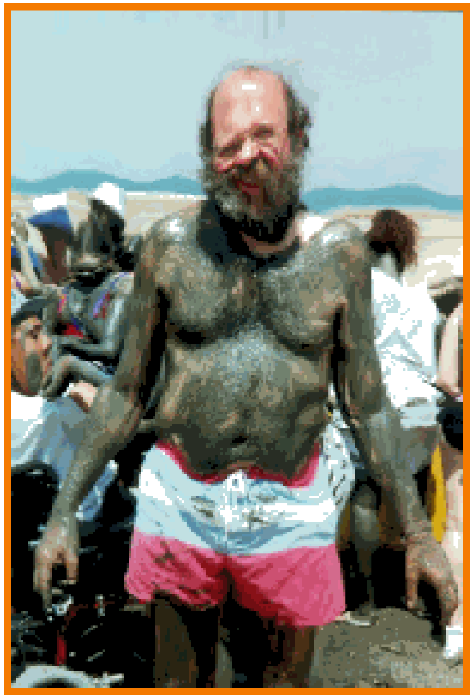
- Any function of u and v:
  - $x = f_x(u,v)$
  - $y = f_y(u,v)$



Fish-eye



"Swirl"



"Rain"

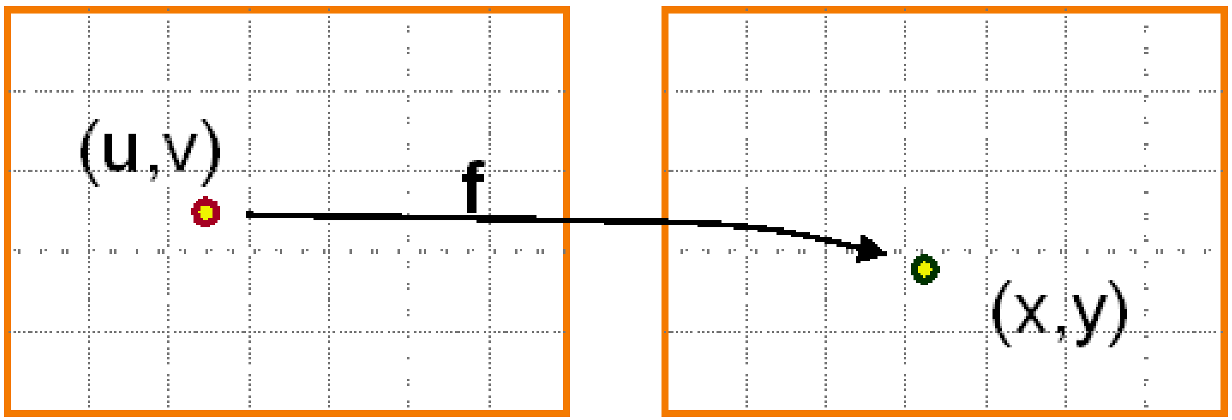
# Image Warping Implementation I

- Forward mapping:

```
for (int u = 0; u < umax; u++) {
  for (int v = 0; v < vmax; v++) {
    float x = fx(u,v);
    float y = fy(u,v);
    dst(x,y) = src(u,v);
  }
}
```

Decide the position

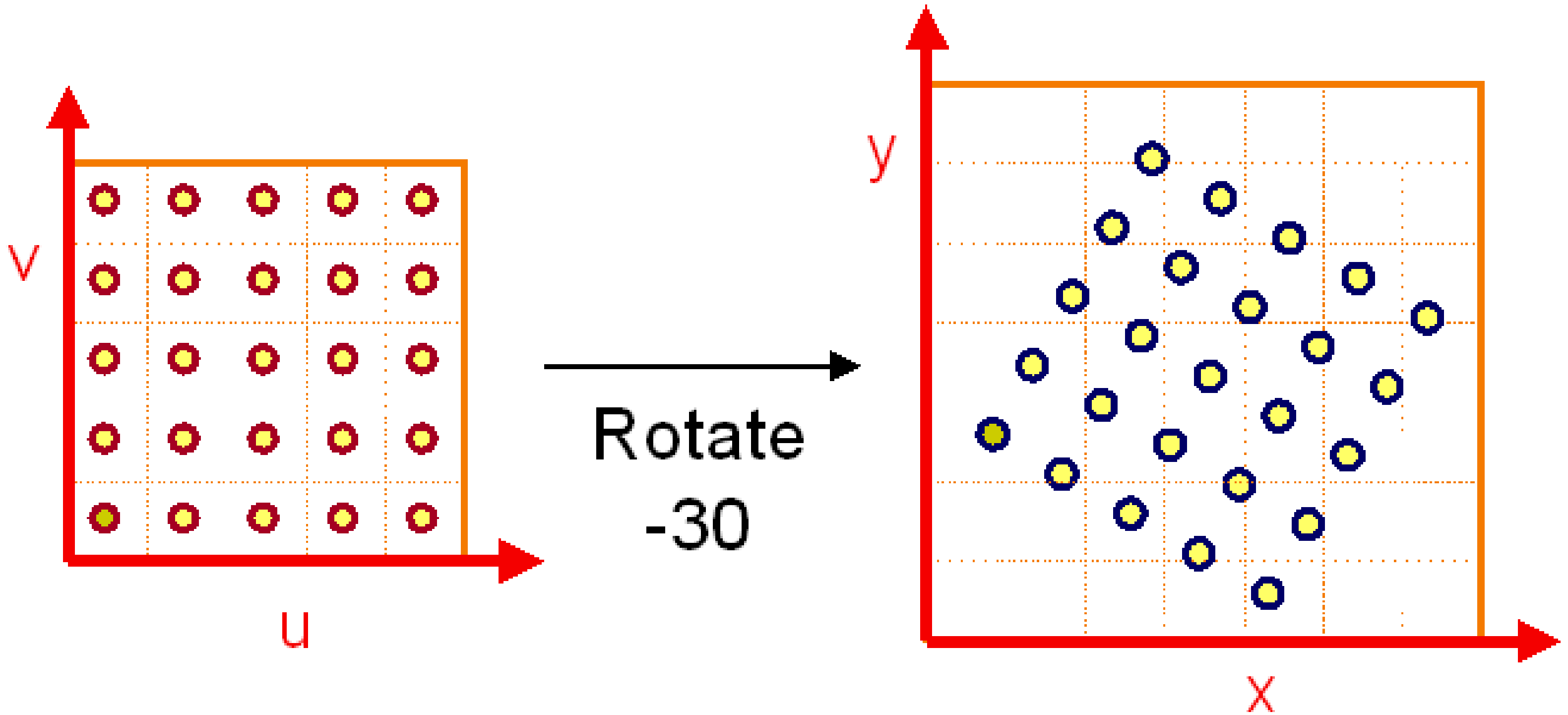
Decide the color




Source image    Destination image

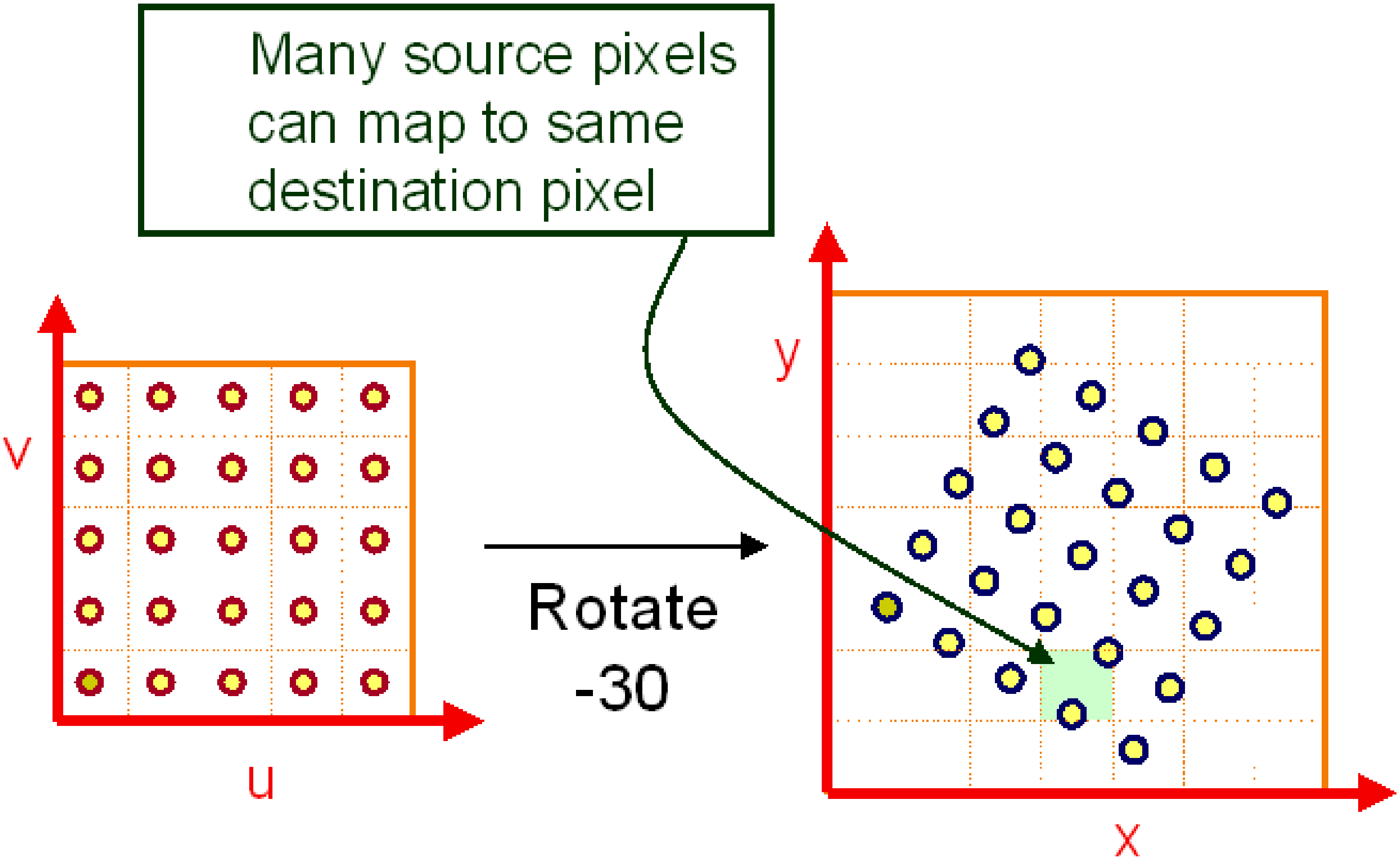
# Forward Mapping

- Iterate over source image



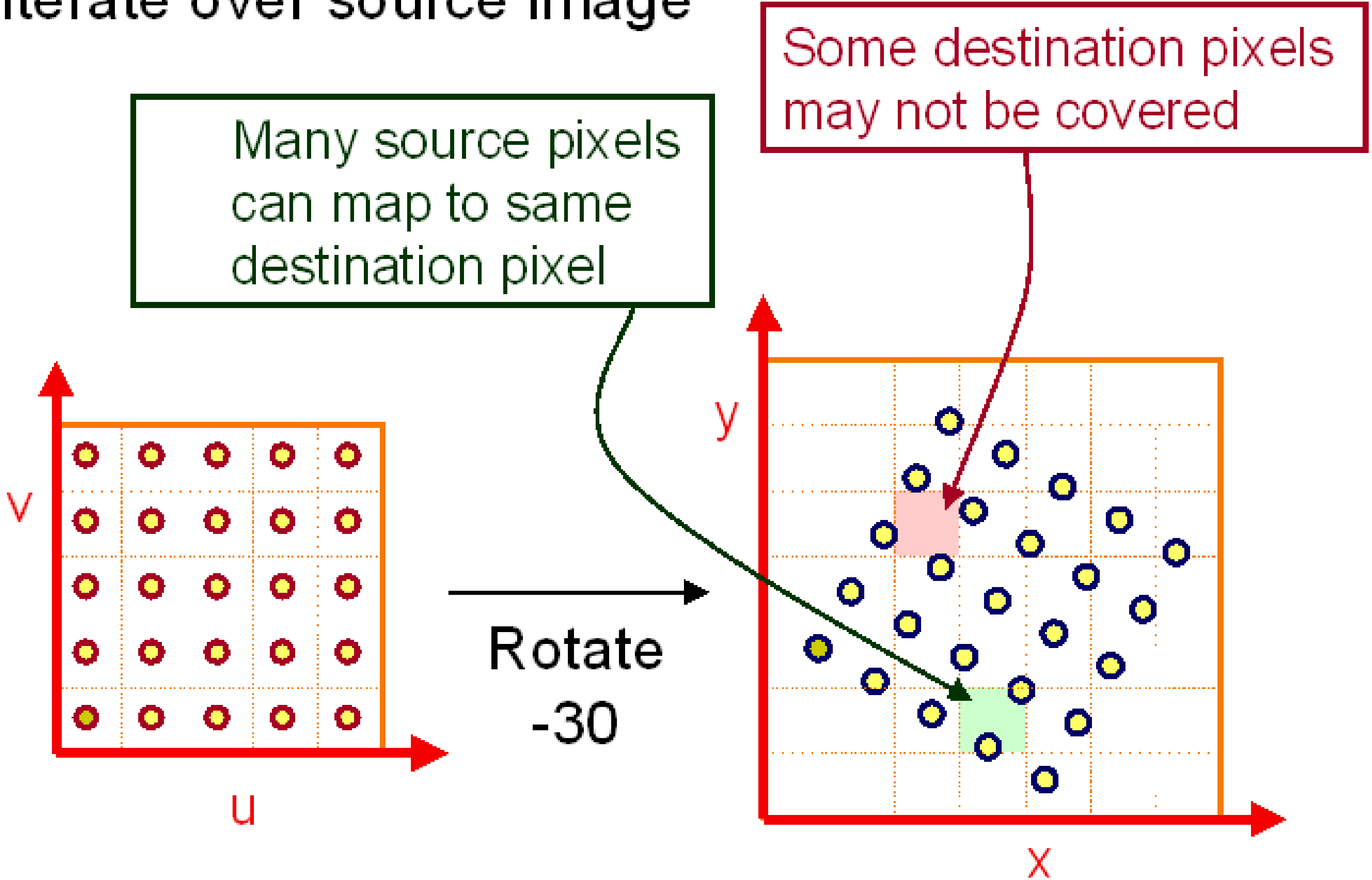
# Forward Mapping - NOT

- Iterate over source image



# Forward Mapping - NOT

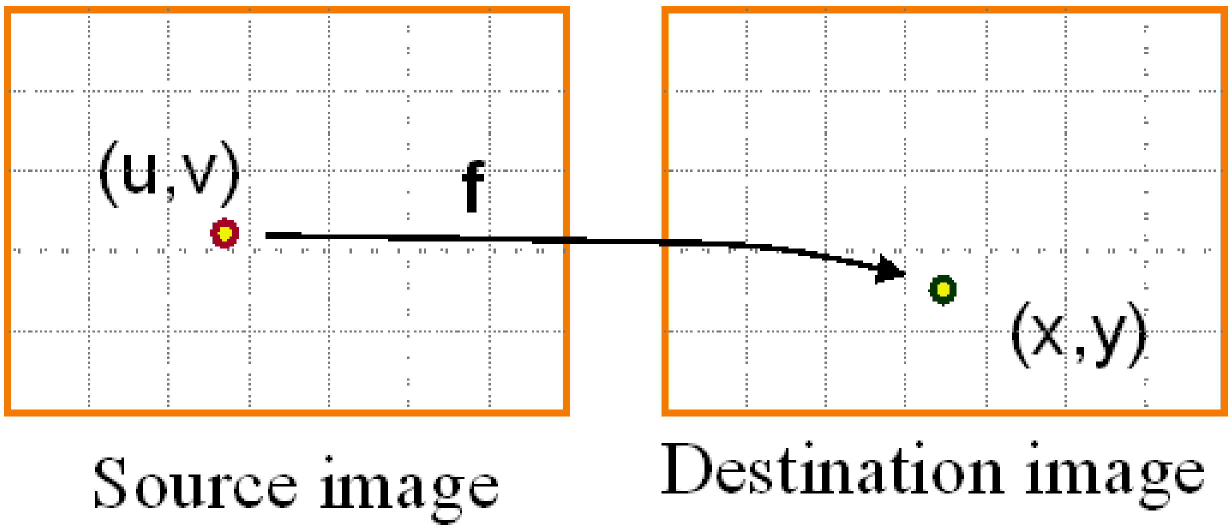
- Iterate over source image



# Image Warping Implementation II

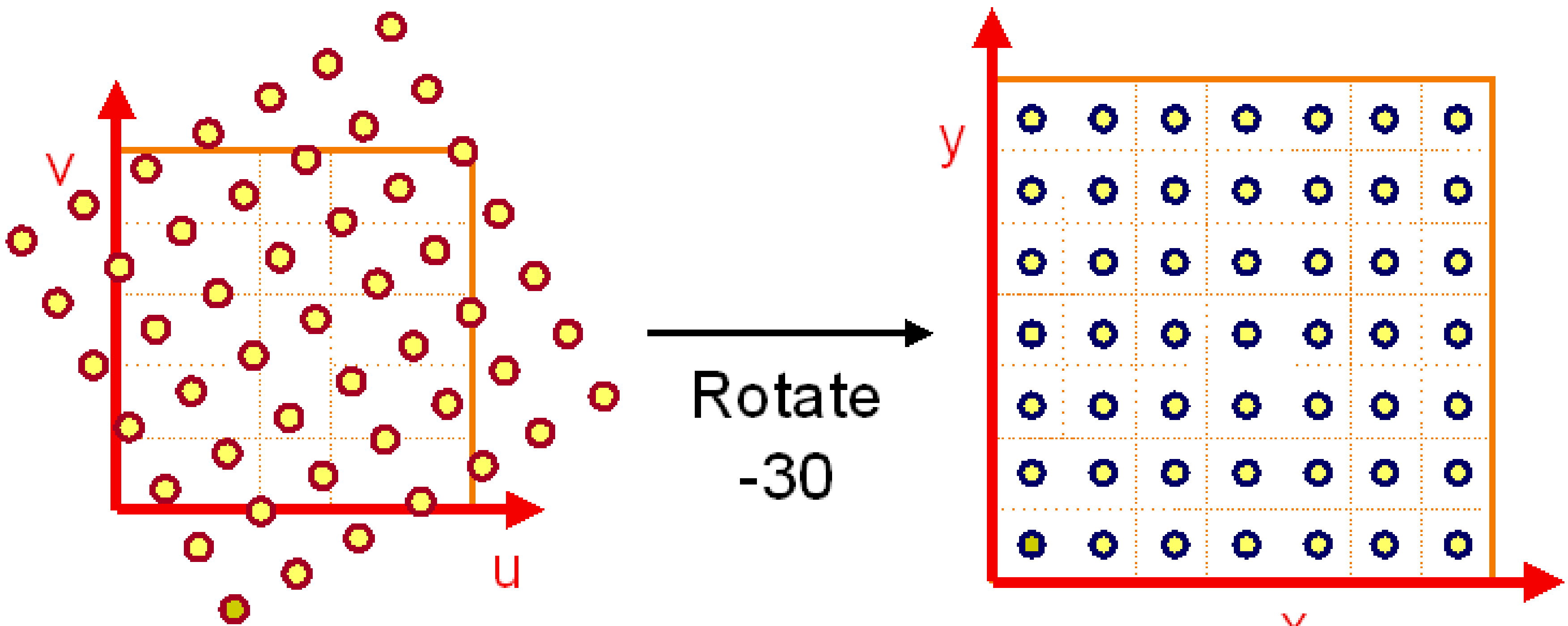
- Reverse mapping: ( Backward mapping )

```
for (int x = 0; x < xmax; x++) {
   for (int y = 0; y < ymax; y++) {
      float u = fx^-1(x,y);
      float v = fy^-1(x,y);
      dst(x,y) = src(u,v);
   }
}
```



# Reverse Mapping

- Iterate over destination image
  - Must resample source
  - May oversample, but much simpler!



# Overview

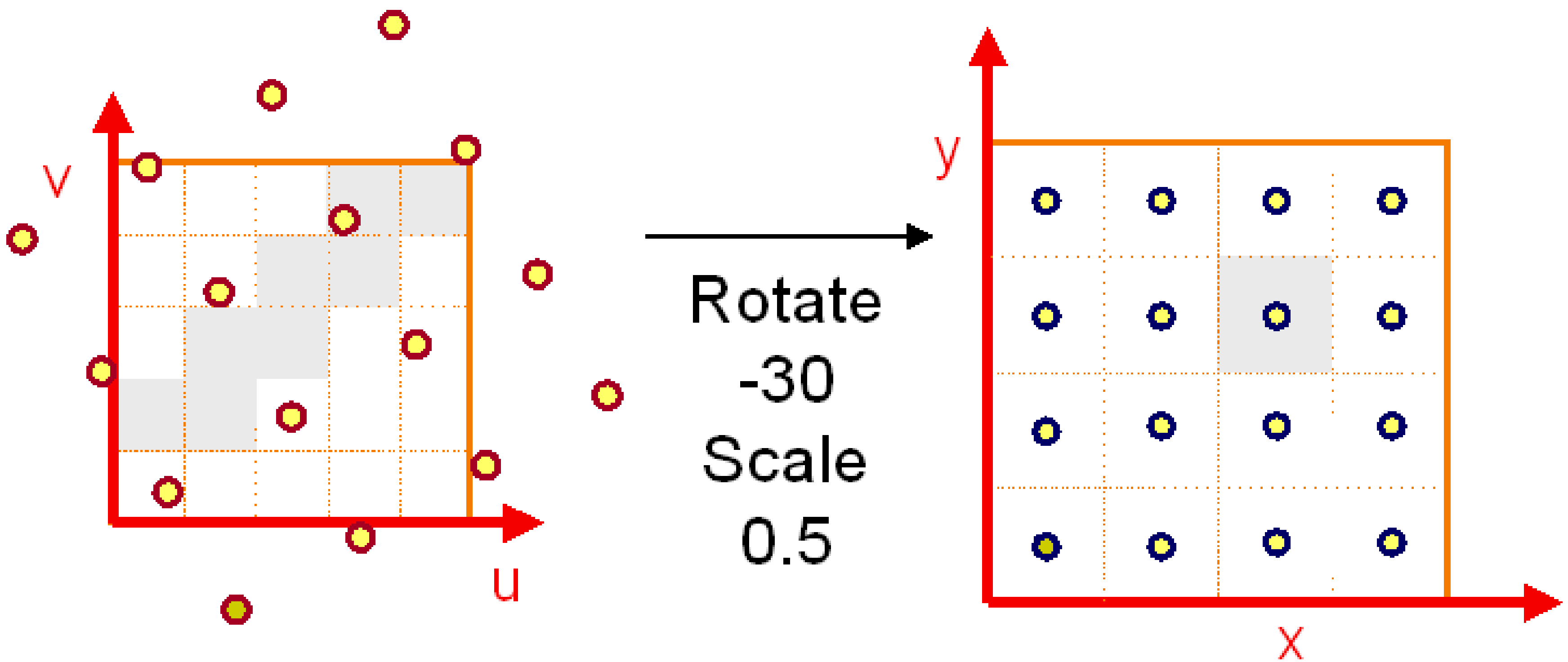
- Mapping
  - Forward
  - Reverse

» **Resampling**
  - Point sampling
  - Triangle filter
  - Gaussian filter

# Point Sampling

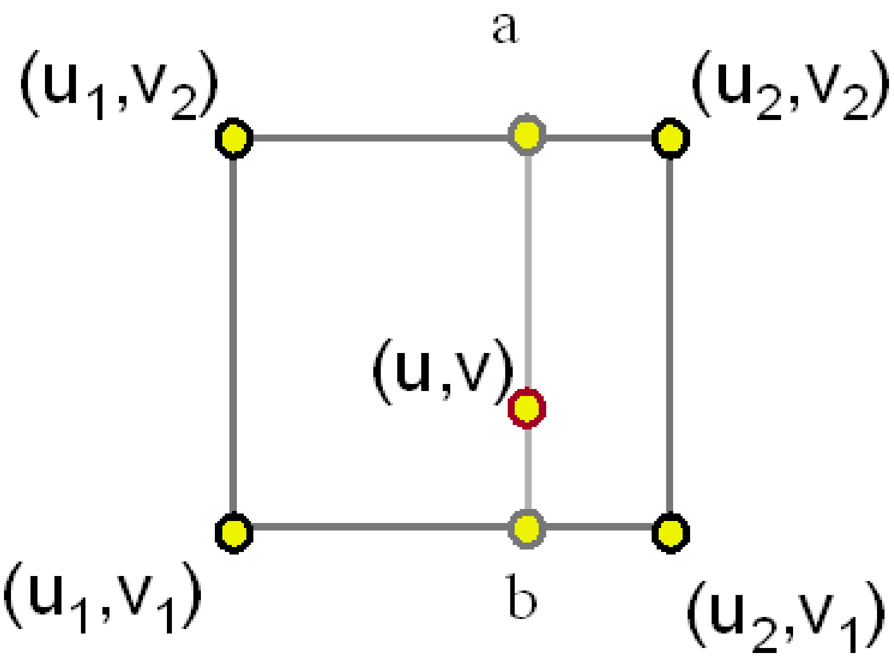
- Take value at closest pixel:
  - int iu = trunc(u+0.5);
  - int iv = trunc(v+0.5);
  - dst(x,y) = src(iu,iv);

This method is simple, but it causes aliasing



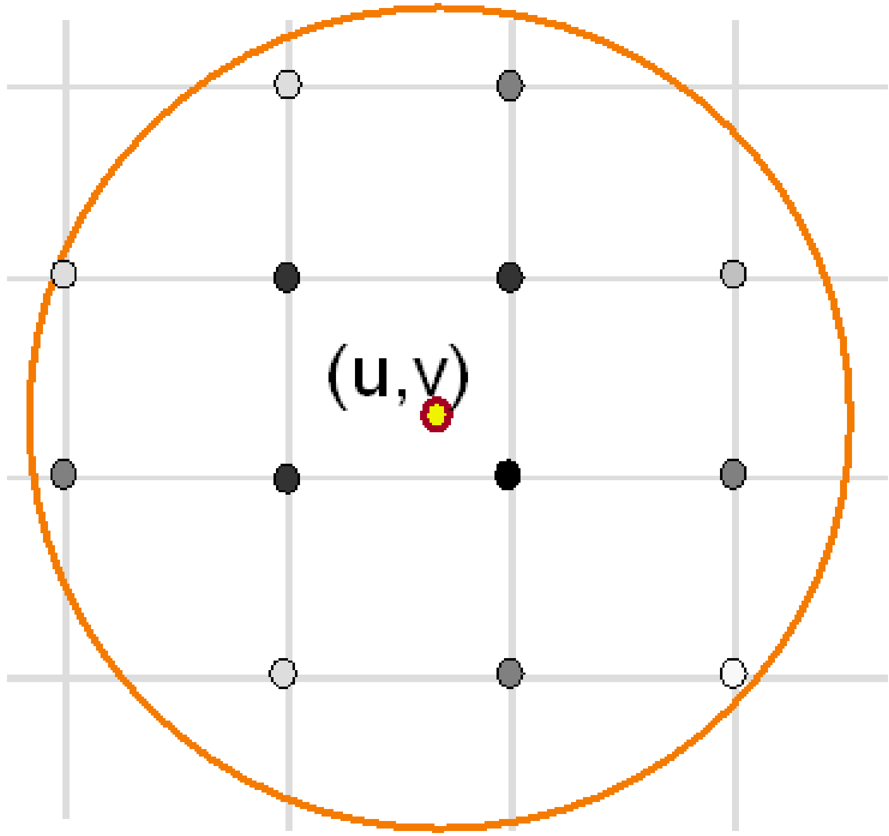
# Triangle Filtering

- Bilinearly interpolate four closest pixels
  - a = linear interpolation of src($u_1$,$v_2$) and src($u_2$,$v_2$)
  - b = linear interpolation of src($u_1$,$v_1$) and src($u_2$,$v_1$)
  - dst(x,y) = linear interpolation of “a” and “b”



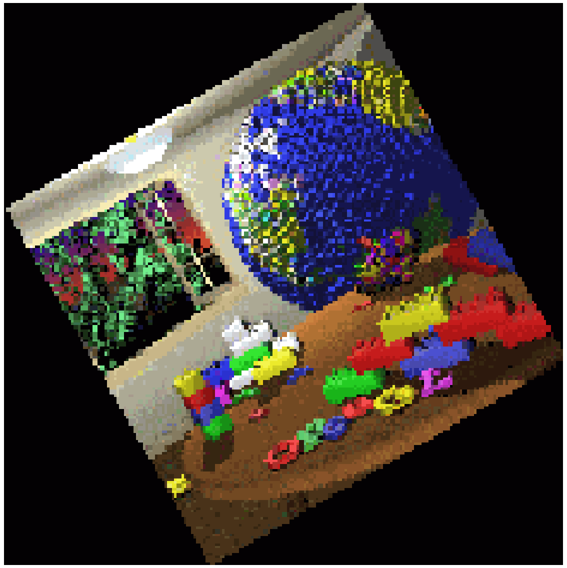
# Gaussian Filtering

- Compute weighted sum of pixel neighborhood:
  - Weights are normalized values of Gaussian function

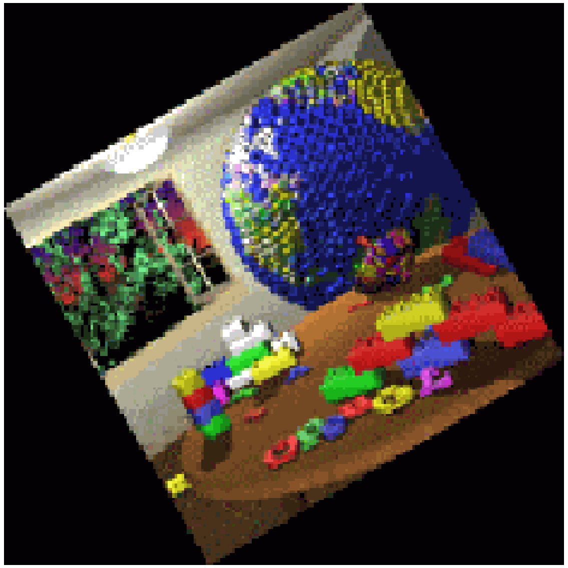


# Filtering Methods Comparison

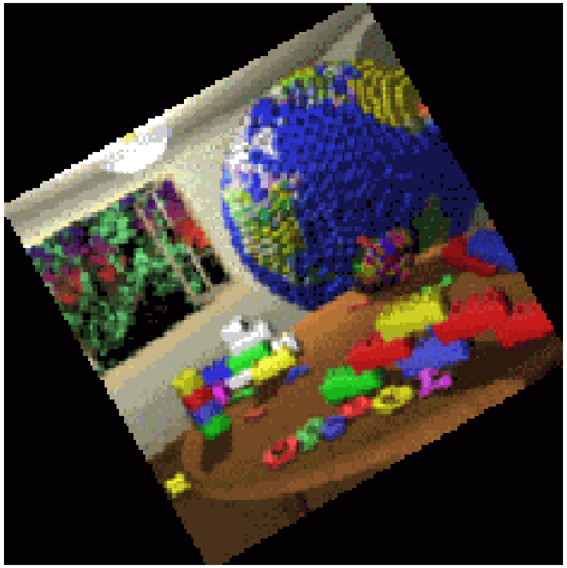
- Trade-offs
  - Aliasing versus blurring
  - Computation speed



Point



Bilinear

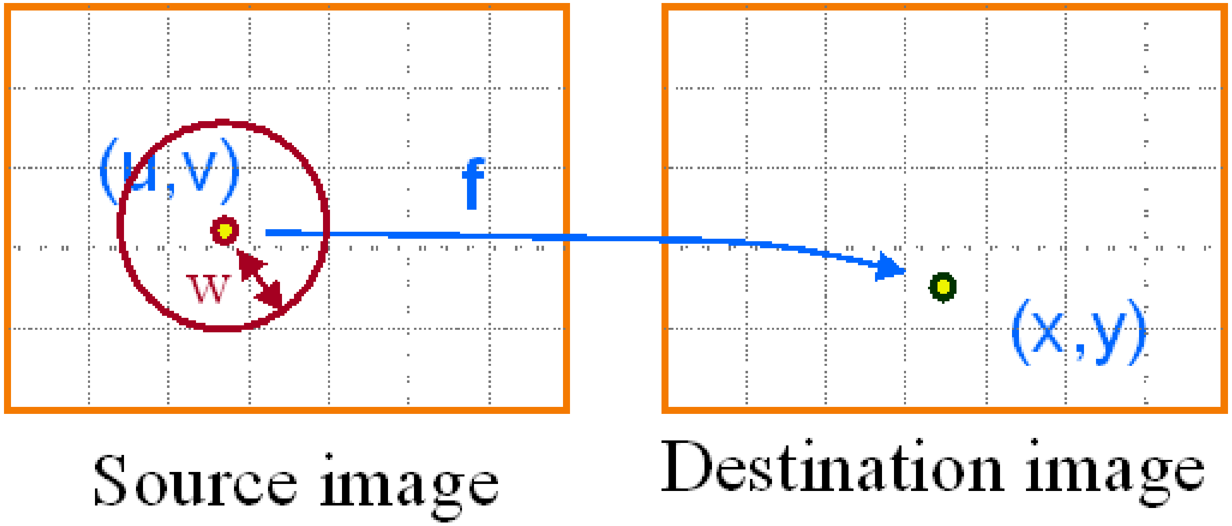


Gaussian

# Image Warping Implementation

- Reverse mapping:

```
for (int x = 0; x < xmax; x++) {
  for (int y = 0; y < ymax; y++) {
    float u = fx^-1(x,y);
    float v = fy^-1(x,y);
    dst(x,y) = resample_src(u,v,w);
  }
}
```



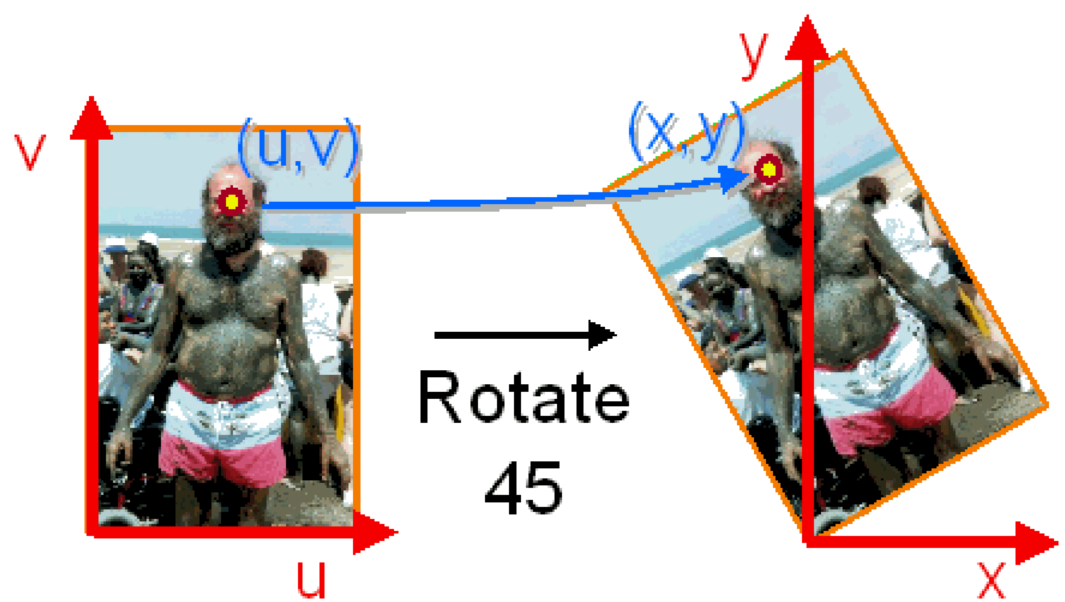
# Example: Rotate

- Rotate (src, dst, theta):

```
for (int x = 0; x < xmax; x++) {
  for (int y = 0; y < ymax; y++) {
    float u = x*cos(-Θ) - y*sin(-Θ);
    float v = x*sin(-Θ) + y*cos(-Θ);
    dst(x,y) = resample_src(u,v,w);
  }
}
```

$x = u\cos\Theta - v\sin\Theta$

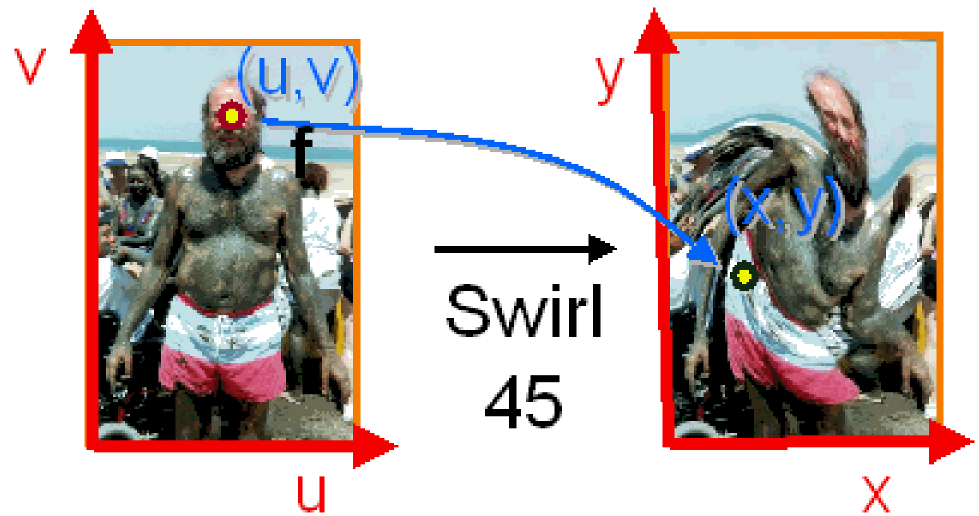
$y = u\sin\Theta + v\cos\Theta$



# Example: Fun

- Swirl (src, dst, theta):

```
for (int x = 0; x < xmax; x++) {
  for (int y = 0; y < ymax; y++) {
    float u = rot(dist(x,xcenter)*theta);
    float v = rot(dist(y,ycenter)*theta);
    dst(x,y) = resample_src(u,v,w);
  }
}
```



# Summary

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

Reverse mapping
is simpler to implement

Different filters trade-off
speed and aliasing/blurring

Fun and creative warps
are easy to implement!

# Image Morphing

# Image Morphing

*Michael Jackson's MTV "Black or White"*

# Image morphing

- The goal is to synthesize a fluid transformation from one image to another.
- Cross dissolving is a common transition between cuts, but it is not good for morphing because of the ghosting effects.

image #1 dissolving image #2

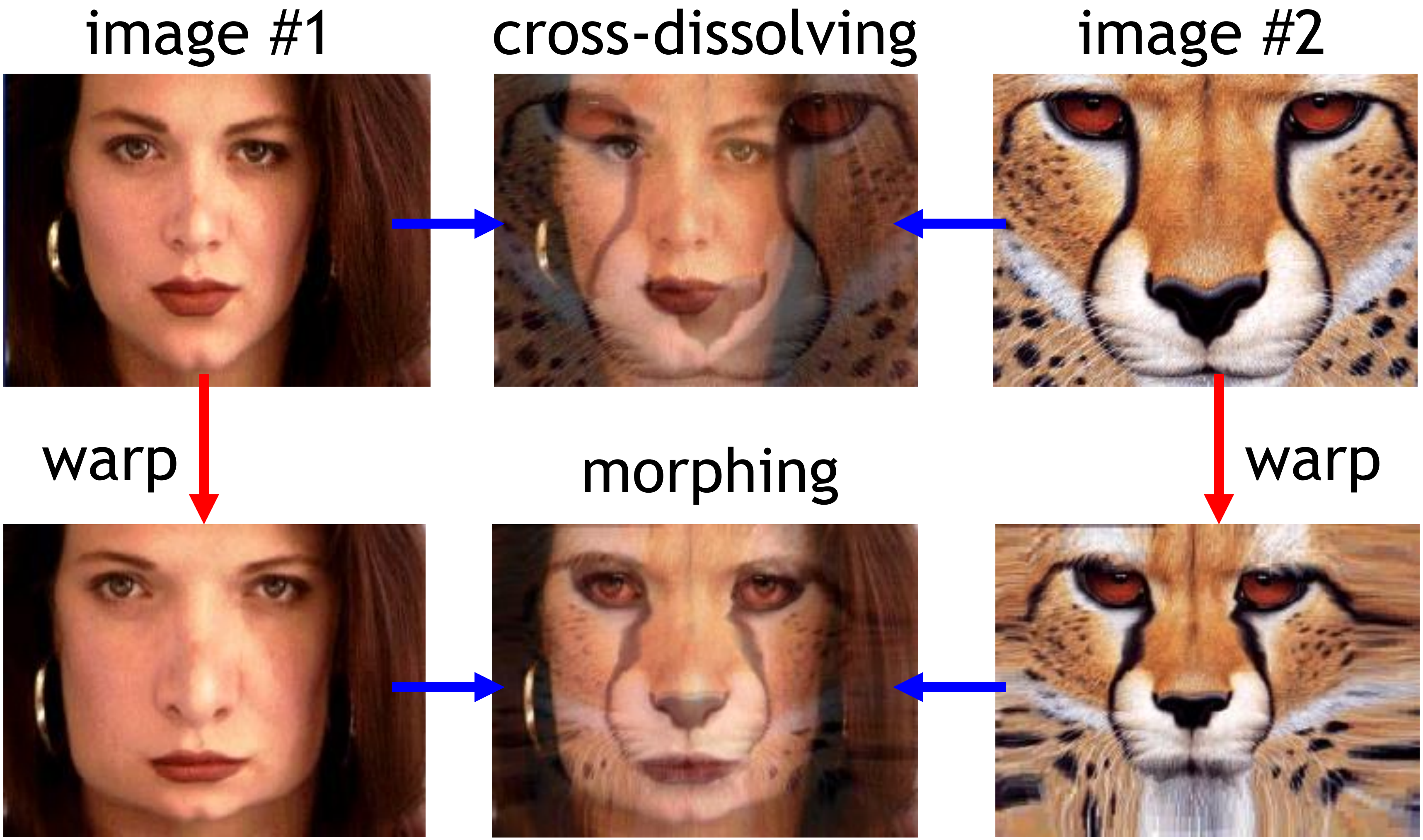
Artifacts of cross-dissolving

# Image morphing

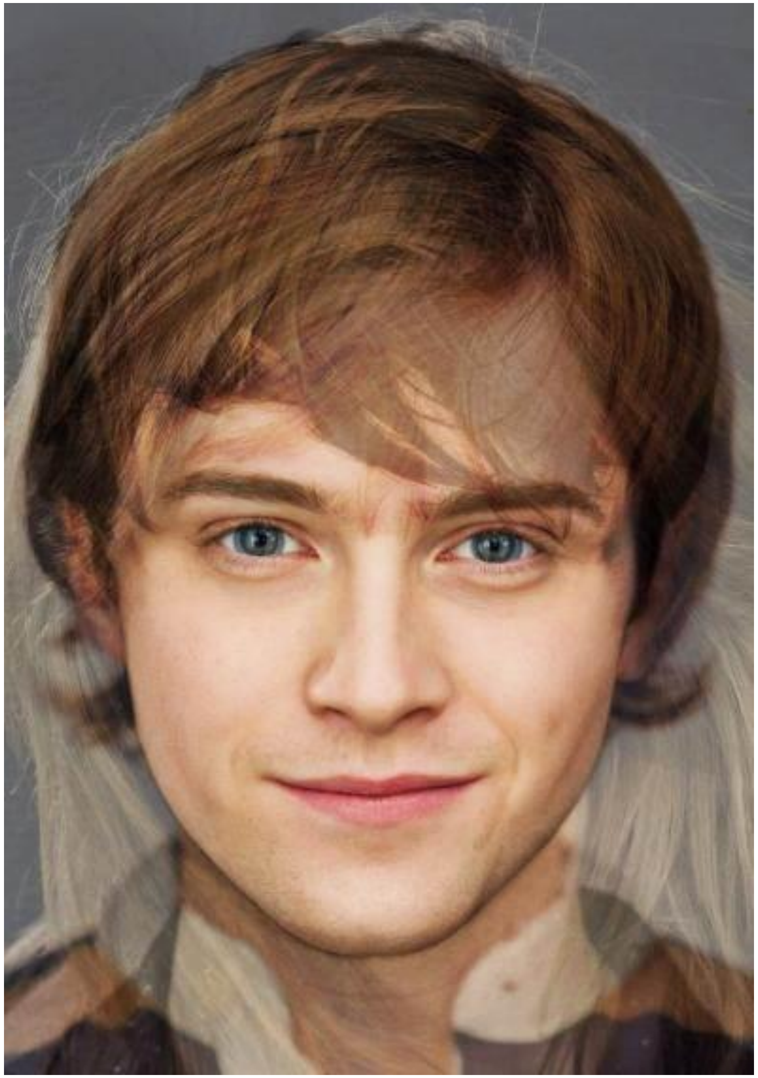
- Why ghosting?
- Morphing = warping + cross-dissolving

shape (geometric) — color (photometric)

# Image morphing



# Face averaging by morphing



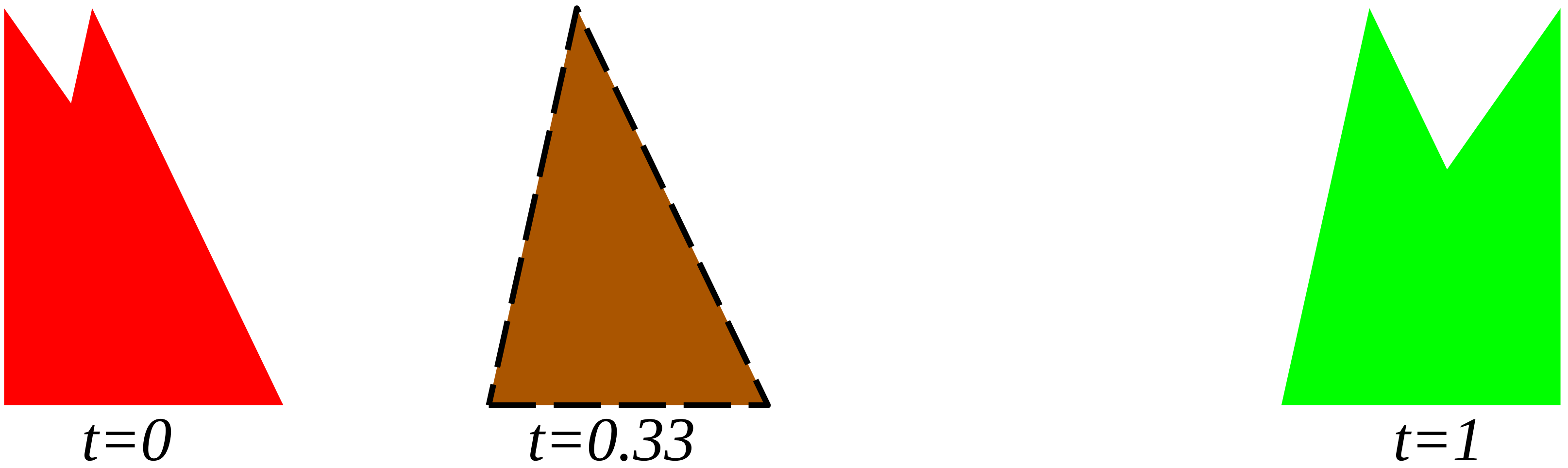
丹尼爾克雷夫、魯柏葛林特、艾瑪華森

克里夫歐文、休傑克曼、伊旺麥奎格

# Image morphing

create a morphing sequence: for each time t

1. Create an intermediate warping field (by interpolation)
2. Warp both images towards it
3. Cross-dissolve the colors in the newly warped images



# An ideal example (in 2004)

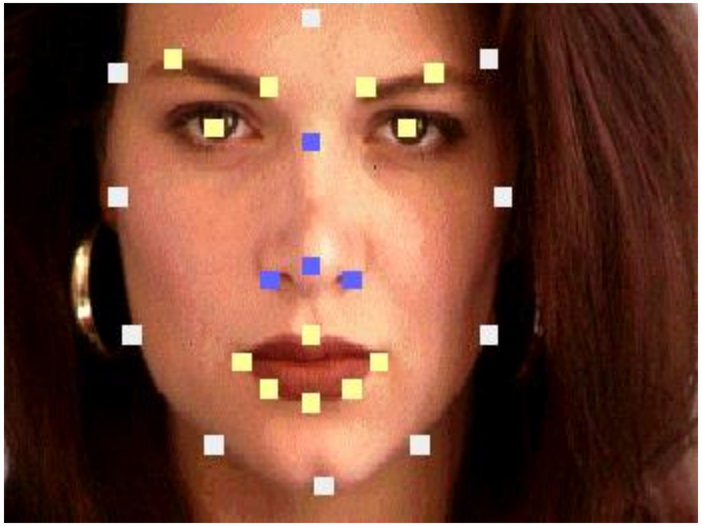
t=0

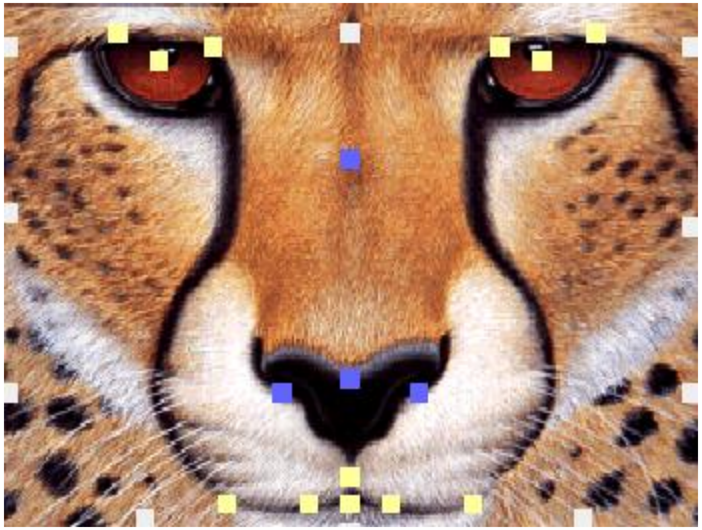
t=0.75

t=1

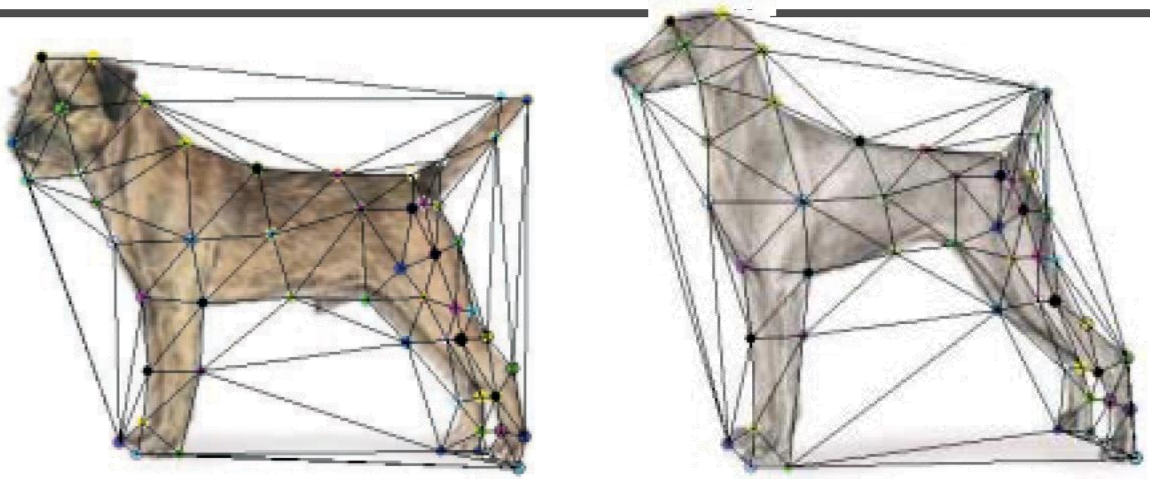
# Warp specification

- How can we specify the warp
  - Specify corresponding points





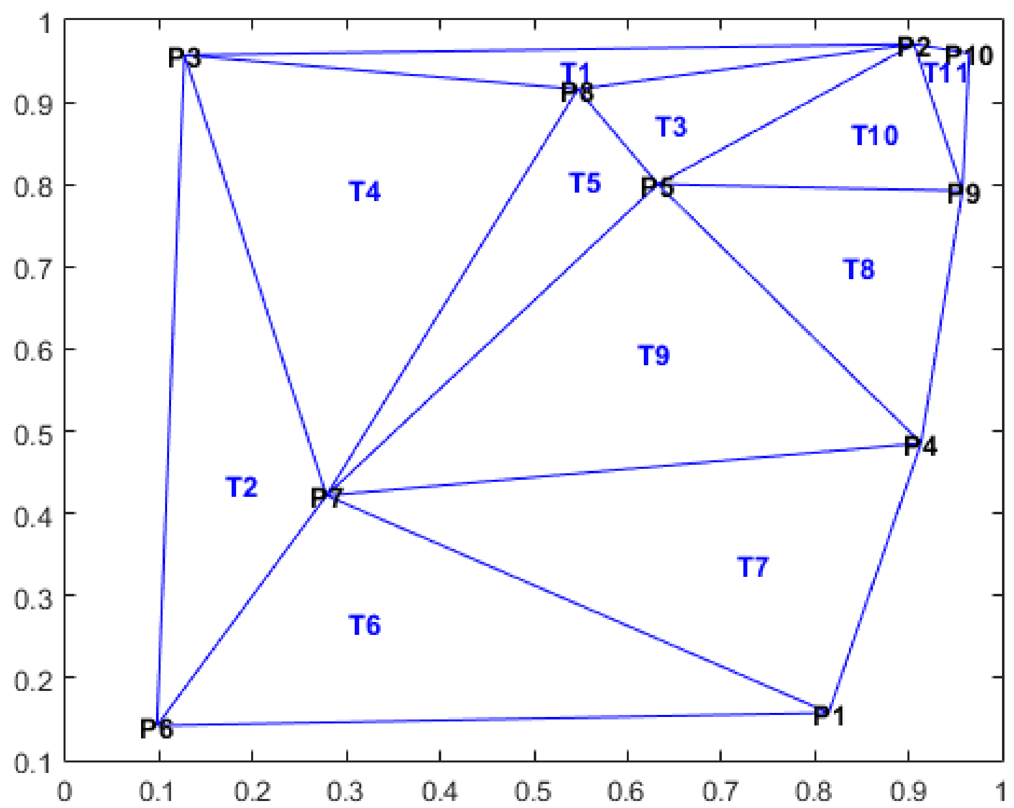
# Solution: convert to mesh warping



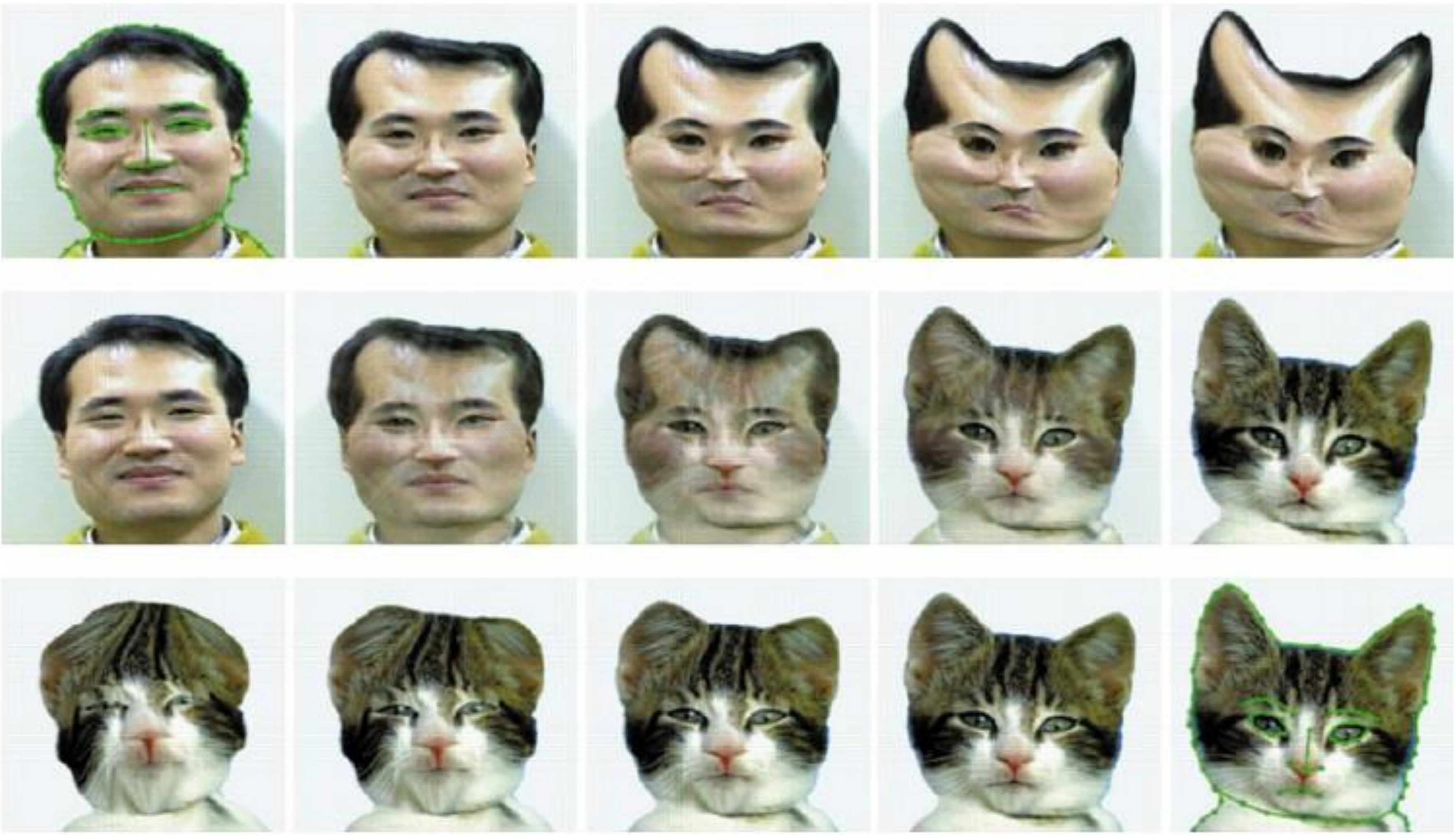
1. Define a triangular mesh over the points
   - Same mesh in both images!
   - Now we have triangle-to-triangle correspondences
2. Warp each triangle separately from source to destination
   - How do we warp a triangle?
   - 3 points = affine warp!
   - Just like texture mapping

# Creating and Editing Delaunay Triangulations

- x = rand(10,1);
- y = rand(10,1);
- dt = delaunayTriangulation(x,y)
- triplot(dt);



# Transition control



# Multi-source morphing