



디지털영상처리, 컴퓨터비전

# Ch. 5: Geometric Transformations

`cvGetAffineTransform()`, `cvWarpAffine()` 함수  
`cv2DRotationMatrix()`, `invertAffineTransform()` 함수  
`cvGetPerspectiveTransform()`, `cvWarpPerspective` 함수

2024년 1학기

서경대학교 김진헌

# 차례

1

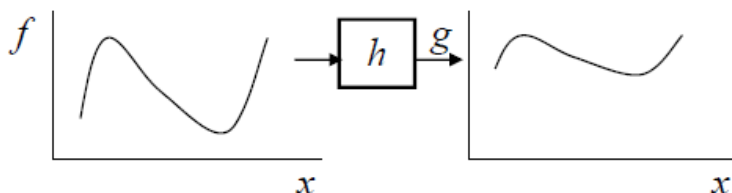
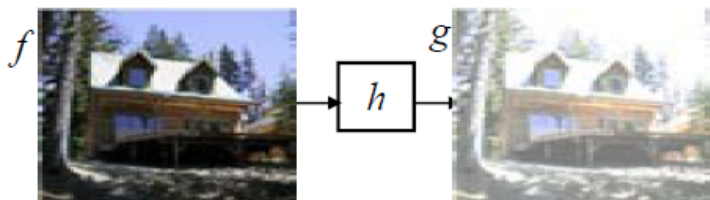
- 개요
- 이론적 분석
- 개념 요약
- OpenCV 함수
- 예제 분석

# 1. 개요

2

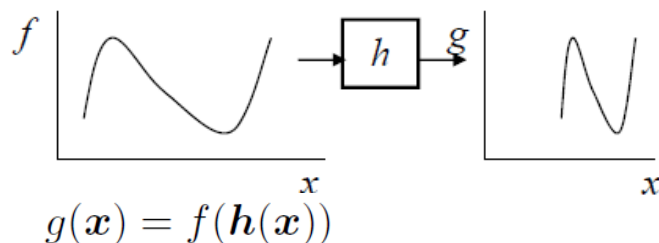
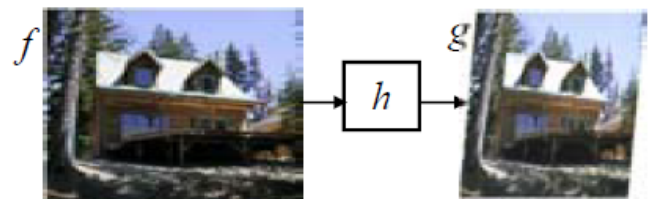
Richard Szeliski, Computer Vision: Algorithms and Applications, 2010

## □ 명암변환 vs 기하학적 변환



$$g(x) = h(f(x)) \quad x = (x, y) \in \mathcal{R}^2$$

**Intensity transform(명암 변환):** 화소 값을 변환  
특정 위치  $(x, y)$ 의 화소 값을 알면 그 값을  
바탕으로 변환할 화소 값을 만들어 낸다.  
 $h()$ 는  $(x, y)$  위치의 화소 값을 변환한다.



**Geometric Transform(기하학적 변환):** 영상의  
좌표축을 변환하여 그것에 따라 영상을 변환  
 $f()$ 는 영상 함수.  $(x, y)$ 에 따라 결정된다.  
 $h()$ 는  $(x, y)$  좌표축을  $(x', y')$ 로 변환한다.

기하학적 변환의 문제=위치  $\mathbf{x}$ 에 있는 화소가  
변환될 영상  $\mathbf{x}'$ 로 갈 수 있는 맵핑 함수  $h(\mathbf{x})$ 를  
찾는 문제

Image warping involves modifying the **domain** of an image function rather than its range.

# 1.1 기학적 변환의 구현 방식

3

## □ Forward Warping (전방 변환):

- 입력 영상의 각 화소를 출력 영상으로 직접 매핑하는 방식.
- 입력 영상의 모든 화소에 대해 출력 영상에 대응되는 위치를 계산하고, 이동, 회전 또는 크기 조정 등의 변환을 적용한다.
- 이 방법은 비교적 직관적이지만 출력 영상에서 입력 영상으로의 화소 값에 대한 매핑이 분명하지 않을 수 있다.
- 출력 영상의 화소 값은 입력 영상의 화소 값의 조합으로 얻어지므로, 특정한 출력 화소는 여러 입력 화소로부터 영향을 받을 수 있다.

## □ Inverse Warping (역방향 변환):

- 출력 영상의 각 픽셀을 입력 이미지로부터 직접 매핑하는 방식.
- 출력 영상의 각 화소 위치에 대해 입력 영상에서 해당 위치로의 역변환을 적용하여 대응되는 입력 영상의 픽셀 값을 결정한다.
- 이 방법은 출력 영상의 각 화소에 대한 입력 영상의 위치를 명확하게 알 수 있지만, 이를 위해서는 출력 영상의 모든 화소에 대해 입력 영상으로의 역변환을 계산해야 한다.
- 이는 계산적으로 비용이 많이 들 수 있습니다.

# 1.2 Forward/Inverse 장단점 비교

## 4

### □ Forward Warping의 장단점:

#### □ 장점:

- 비교적 직관적이며 간단합니다. 입력 이미지의 모든 픽셀을 출력 이미지로 직접 매핑하는 방식이기 때문에 구현하기 쉽습니다.
- 처리 속도가 빠를 수 있습니다. 특히, 출력 이미지의 해상도가 낮을 경우에는 빠른 계산이 가능합니다.

#### □ 단점:

- 출력 이미지의 일부 영역에서 입력 이미지로부터의 정보 손실이 발생할 수 있습니다. 특히, 출력 이미지의 특정 픽셀이 여러 입력 픽셀로부터 영향을 받을 수 있으므로 정보의 왜곡이 발생할 수 있습니다.
- 입력 이미지와 출력 이미지 사이의 대응 관계가 분명하지 않을 수 있습니다.

### □ Inverse Warping의 장단점:

#### □ 장점:

- 출력 이미지의 각 픽셀이 입력 이미지의 명확한 대응점을 가집니다. 따라서, 변환된 이미지의 픽셀 값이 정확하게 계산됩니다.
- 변환된 이미지의 특정 영역에서 입력 이미지의 정보 손실이 적습니다.

#### □ 단점:

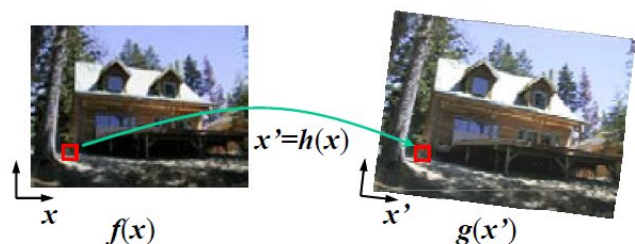
- 계산 비용이 많이 들 수 있습니다. 출력 이미지의 각 픽셀에 대해 입력 이미지로의 역변환을 계산해야 하므로 처리 속도가 느릴 수 있습니다.
- 정확한 역변환을 위해서는 입력 이미지와 출력 이미지 사이의 변환 관계를 명확하게 이해해야 합니다. 때로는 이 변환 관계를 정확하게 파악하기 어려울 수 있습니다.

- 따라서, Forward warping은 구현이 간단하고 처리 속도가 빠를 수 있지만, 정보의 왜곡이 발생할 수 있습니다. 반면에 Inverse warping은 출력 이미지의 정확성과 정보 손실을 최소화할 수 있지만, 계산 비용이 높을 수 있고 변환 관계를 이해하는 것이 중요합니다.

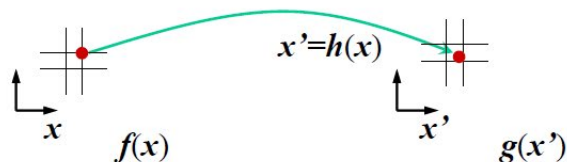
# 1.3 Forward/Inverse warping

5

Richard Szeliski, Computer Vision: Algorithms and Applications, 2010



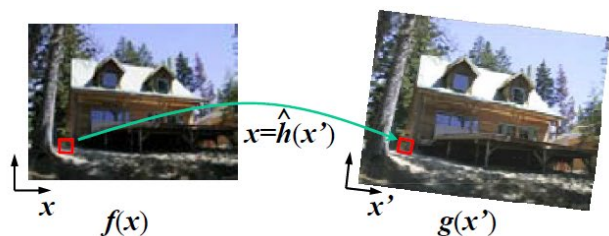
(a)



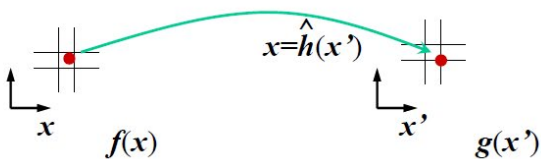
(b)

Forward warping algorithm: (a) a pixel  $f(x)$  is copied to its corresponding location  $x' = h(x)$  in image  $g(x')$ ; (b) detail of the source and destination pixel locations.

단점: crack/hole 존재(변환된 영상에 빈 공간이 발생할 수 있음) => interpolation으로 한계.



(a)



(b)

주의!!!: 녹색 화살표 방향을 바꾸어야 함.

Inverse warping algorithm : (a) a pixel  $g(x')$  is sampled from its corresponding location  $x = \hat{h}(x')$  in image  $f(x)$ ; (b) detail of the source and destination pixel locations.

참고: 역방향 맵핑 예제

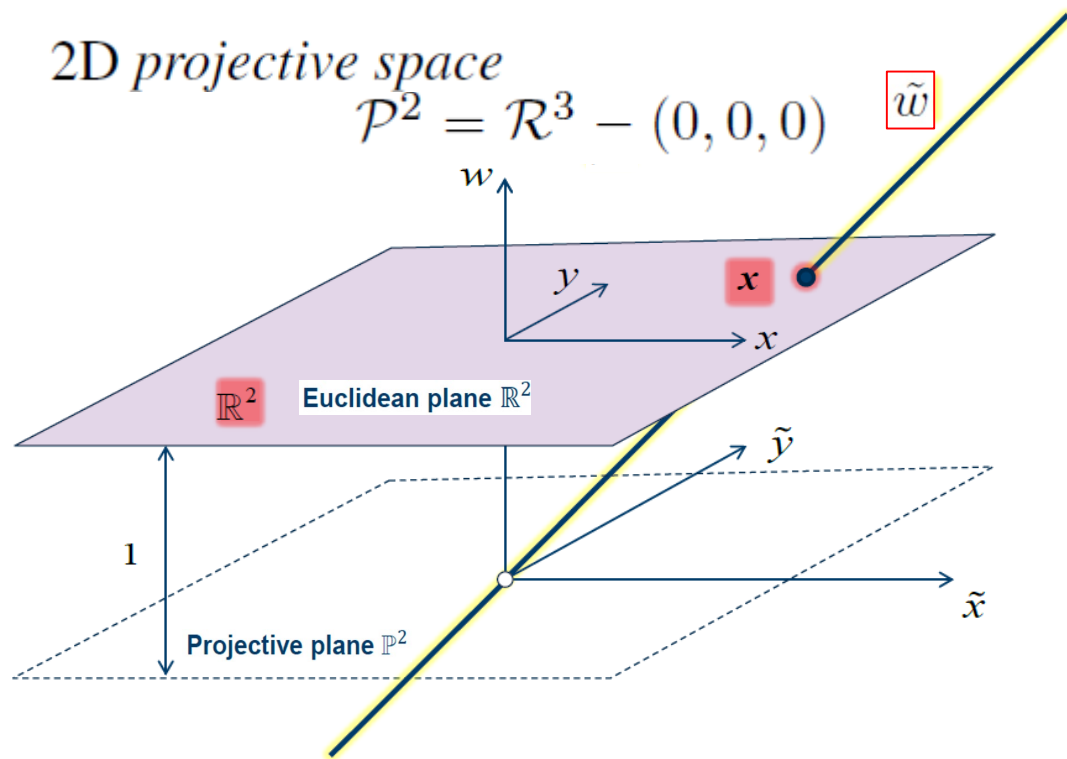
<https://stackoverflow.com/questions/34989513/how-to-reverse-warpperspective>

# 2. 매트릭스 정의에 의한 변환

들어가기에 앞서: Projective Plane

6

$f: a \mapsto b$ 는 함수  $f$ 는 원소  $a$ 를 원소  $b$ 에 대응시킨다는 것을 의미한다.



How to describe points in the plane?

**Euclidean plane  $\mathbb{R}^2$**

- Choose a 2D coordinate frame
- Each point corresponds to a unique pair of Cartesian coordinates

$$x = (x, y) \in \mathbb{R}^2 \mapsto x = \begin{bmatrix} x \\ y \end{bmatrix}$$

**Projective plane  $\mathbb{P}^2$**

- Expand coordinate frame to 3D
- Each point corresponds to a triple of homogeneous coordinates

$$\tilde{x} = (\tilde{x}, \tilde{y}, \tilde{w}) \in \mathcal{P}^2 \mapsto \tilde{x} = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{w} \end{bmatrix}$$

A homogeneous vector  $\tilde{x}$  can be converted back into an *inhomogeneous* vector  $x$  by dividing through by the last element  $\tilde{w}$ , i.e.,

$$\tilde{x} = (\tilde{x}, \tilde{y}, \tilde{w}) = \tilde{w}(x, y, 1) = \tilde{w}\bar{x}, \quad \text{where } \bar{x} = (x, y, 1) \text{ is the augmented vector.}$$

Homogenous Vector

Inhomogenous Vector(augmented vector)

# 2.1 Transformation Matrix [ ]

매트릭스 연산으로 scale, shear, rotation 문제를 해결해 보자.

7

Affine transformations in 5 minutes

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \text{scale\_horizontal} & \text{shear\_horizontal} \\ \text{shear\_vertical} & \text{scale\_vertical} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

we'll start with the identity matrix  
this does nothing



$$\begin{bmatrix} 1.2 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$$

now if we change the upper left element  
it performs a

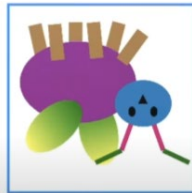


$$\begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.3 \end{bmatrix}$$

changing the bottom right element  
performs a scale in y

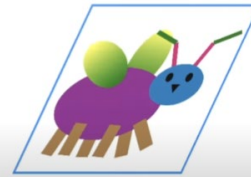


$$\begin{bmatrix} 1.0 & 0.0 \\ 0.0 & -1.0 \end{bmatrix}$$



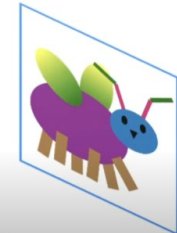
$$\begin{bmatrix} 1.0 & 0.5 \\ 0.0 & 1.0 \end{bmatrix}$$

the upper right entry corresponds to a  
horizontal shear



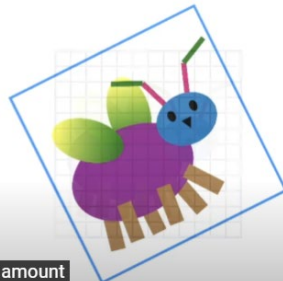
$$\begin{bmatrix} 1.0 & 0.0 \\ -0.5 & 1.0 \end{bmatrix}$$

and the lower left is a vertical shear



$$\begin{bmatrix} 1.0 & -0.5 \\ 0.5 & 1.0 \end{bmatrix}$$

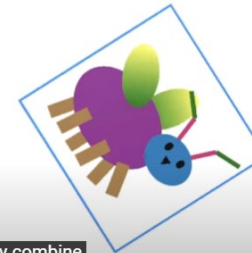
and then vertically by the same amount  
cool that creates a rotation



회전은  
shear를 x, y  
방향으로 같은  
양으로  
shearing 하면  
된다. 그런데  
크기가 조금씩  
달라진다.

$$\begin{bmatrix} \cos(1.7\pi) & -\sin(1.7\pi) \\ \sin(1.7\pi) & \cos(1.7\pi) \end{bmatrix}$$

when we put these together they combine  
to produce a perfect rotation



Scale 성분은  $\cos()$ 을  
적용하면 같은 크기로  
완벽한 회전을 구현할  
수 있다.  
→ 문제는 이제  
이동(translation) 만  
남았다.

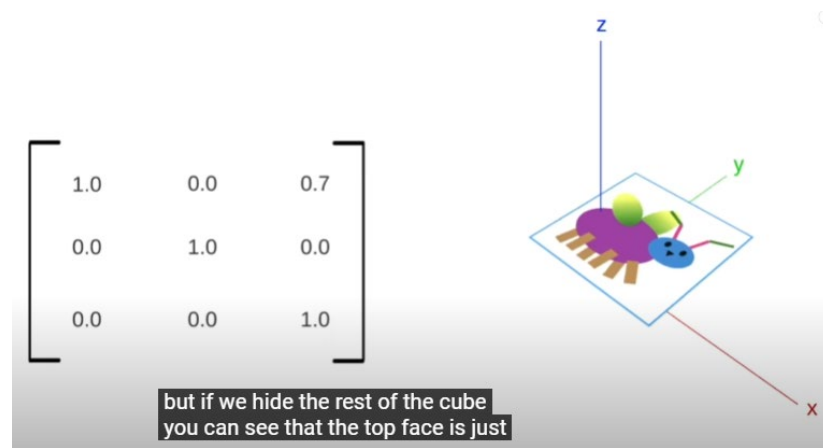
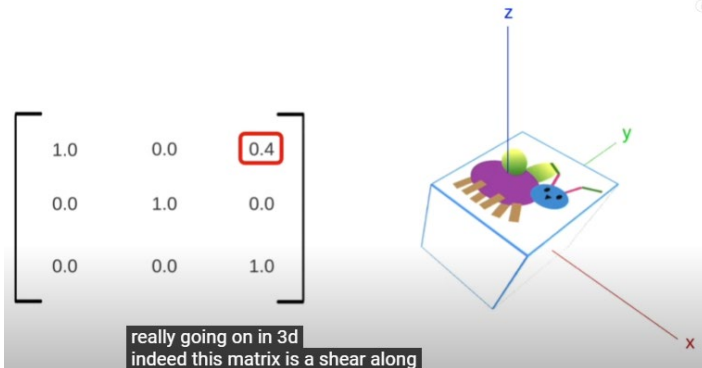
Scale, shear, rotation은 2x2 matrix로 충분하다. 이동은 이것만으로는 안된다



이동(translation) 문제를 매트릭스로 해결하기 위해 z 축을 도입하고  
 Homogeneous coordinates를 도입한다.  
 Translation 성분은 3D 평면의 shearing처럼 작동한다.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \text{scale}_{\text{horizontal}} & \text{shear}_{\text{horizontal}} & \text{translation}_{\text{horizontal}} \\ \text{shear}_{\text{vertical}} & \text{scale}_{\text{vertical}} & \text{translation}_{\text{vertical}} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

실제로는 z축에 대한 연산은 실행하지 않는다.  
 맨 위 평면이 translation 영상이다.



3D 평면의 shearing이 맨 윗면은 translating한 것이 된다.

## 2.2 Affine Transform의 매트릭스

9

**Identity** 
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

**Reflection** 
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

**Translation** 
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

**Scale** 
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

**Rotation** 
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

**Shear-X** 
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \lambda_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

**Shear-Y** 
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \lambda_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

## 2.3 참고: 이론적 분석

10

- 대다수 Homogenous 좌표와 Inhomogenous 좌표와의 관계를 수식으로 표현

$$x' = \underbrace{\begin{bmatrix} I & t \end{bmatrix}}_{2 \times 2 \text{ matrix}} \bar{x} \quad \text{or} \quad \bar{x}' = \underbrace{\begin{bmatrix} I & t \\ 0^T & 1 \end{bmatrix}}_{2 \times 3 \text{ matrix}} \bar{x}$$

- 투영 변환은 Homogenous 좌표계로 표현된다.

$$\tilde{x}' = \underbrace{\tilde{H}}_{\substack{3 \times 3 \\ \text{matrix}}} \tilde{x}$$

# 2D Transforms의 이론적 개념

11

Richard Szeliski , Computer Vision: Algorithms and Applications, 2010

**Translation.** 2D translations can be written as  $x' = x + t$  or

$$x' = \begin{bmatrix} I & t \end{bmatrix} \bar{x} \quad I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

where  $I$  is the  $(2 \times 2)$  identity matrix or

where  $0$  is the zero vector. 
$$\bar{x}' = \begin{bmatrix} I & t \\ 0^T & 1 \end{bmatrix} \bar{x}$$

**Rotation + translation.** = Rigid/Euclidean transform

$x' = Rx + t$  or

$$x' = \begin{bmatrix} R & t \end{bmatrix} \bar{x}$$

where

$$R = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

is an orthonormal rotation matrix with  $RR^T = I$  and  $|R| = 1$ .

**Scaled rotation.**

$$\mathbf{x}' = \begin{bmatrix} s\mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{x}} = \begin{bmatrix} a & -b & t_x \\ b & a & t_y \end{bmatrix} \bar{\mathbf{x}},$$

where we no longer require that  $a^2 + b^2 = 1$ .

**Affine.** The affine transformation is written as  $\mathbf{x}' = \mathbf{A}\bar{\mathbf{x}}$ , where  $\mathbf{A}$  is an arbitrary  $2 \times 3$  matrix, i.e.,

$$\mathbf{x}' = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} \bar{\mathbf{x}}.$$

Parallel lines remain parallel under affine transformations.

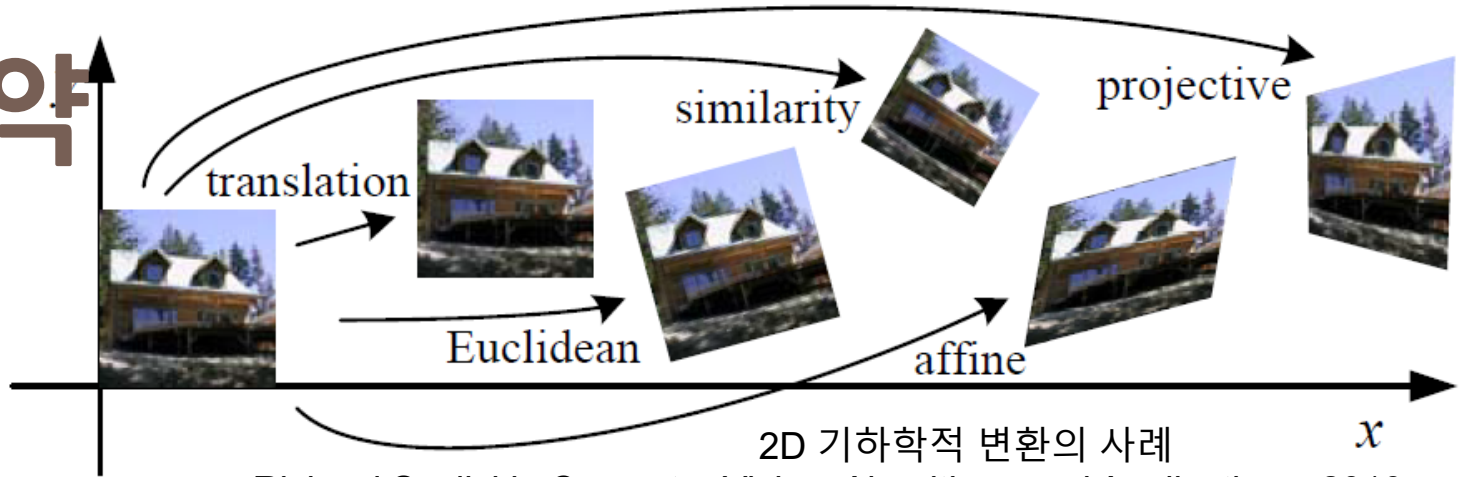
**Projective.** This transformation, also known as a *perspective transform* or *homography*, operates on homogeneous coordinates,

Homogenous coordinates 임에 유의

$$\tilde{\mathbf{x}}' = \tilde{\mathbf{H}}\tilde{\mathbf{x}},$$

# 3. 요약

13



2D 기하학적 변환의 사례  
Richard Szeliski , Computer Vision: Algorithms and Applications, 2010

[warpAffine\(\)](#) 함수로 변환

Transformation of $\mathbb{P}^2$	Matrix	#DoF	Preserves	Visualization
<b>Translation</b> $(x, y)$ 좌표이동.	$2 \times 3 \begin{bmatrix} I & t \\ 0^T & 1 \end{bmatrix}$	2 $(x, y)$	Orientation	
<b>Euclidean</b> 강체(rigid) 변환 translation + rotation	$2 \times 3 \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix}$	3 +회전 $\theta$	+ Lengths	
<b>Similarity</b> scaling	$2 \times 3 \begin{bmatrix} sR & t \\ 0^T & 1 \end{bmatrix}$	4 +scale	+ Angles	
<b>Affine</b> <span style="float: right;"><math>2 \times 3</math></span> 직선과 평형성을 그대로 유지하는 변환 Affine 변환의 일반식은 $Wx+b$	$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix}$	6	+ Parallelism	
<b>Homography /projective</b> <i>Perspective</i> <span style="float: right;"><math>3 \times 3</math></span> <a href="#">warpPerspective()</a> 함수로 변환	$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$	8	Straight lines	

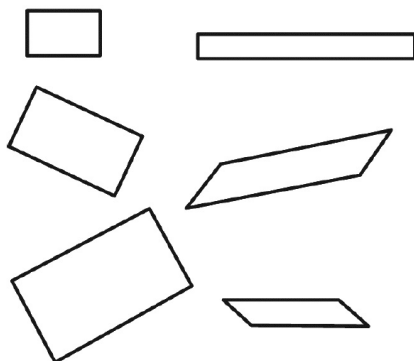
# 요약- Affine vs Perspective

14

이동이 들어간다면  
변환 매트릭스는  
(2x3)가 된다.

**Affine (2x2)**

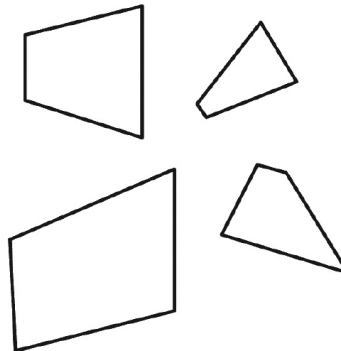
**Parallelograms**



warpAffine() 함수로 변환

**Perspective (3x3)**  
(or "Homography")

**Trapezoids**  
(Includes all of Affine)



warpPerspective() 함수로 변환

- 강체(Rigid-Body), Euclidean 변환 : 크기 및 4각형의 각도가 보존(ex; Translation, Rotation)
- 유사(Similarity) 변환 : 크기는 변하고 각도는 보존(ex; Scaling)
- Affine 변환: 선형변환과 이동변환까지 포함. 선의 평행성은 유지.(ex; 사각형->평행사변형)

Perspective 변환: Affine 변환 특징 포함 + 평행성도 유지되지 않음. 원근 변환

# 4. OpenCV 함수

15

종류	함수명	주요 입력	출력	기능
Affine Transform	<a href="#"><u>warpAffine()</u></a>	변환매트릭스 (2x3)	변환된 영상	어파인 변환시행
	<a href="#"><u>getAffineTransform()</u></a>	3개의 좌표 정보 쌍	변환 매트릭스	Affine 변환을 위한 매트릭스 생성
	<a href="#"><u>getRotationMatrix2D()</u></a>	중심점, 각도, 스케일	변환 매트릭스	회전 변환을 위한 매트릭스 생성
	<a href="#"><u>invertAffineTransform()</u></a>	변환매트릭스	역변환 매트릭스	Affine 변환의 역변환을 위한 변환 매트릭스 생성
Perspective Transform	<a href="#"><u>warpPerspective()</u></a>	변환매트릭스 (3x3)	변환된 영상	투영변환 시행
	<a href="#"><u>getPerspectiveTransform()</u></a>	4개의 좌표 정보 쌍	변환매트릭스	투영 변환을 위한 매트릭스 생성

\* 기하학적 변환의 다른 함수도 더 있는데 주로 여러 영상 데이터 셋트에 대한 변환 매트릭스를 생성하는데 관계된 함수들이다. 예: perspectiveTransform(), findHomography() 등.. Ch. 09에서 다룬다.



# warpAffine()

```
dst=cv.warpAffine(src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]])
```

16

- Applies an affine transformation to an image. The function warpAffine transforms the source image using the specified matrix, when the flag **WARP\_INVERSE\_MAP** is set.

$$\text{dst}(x, y) = \text{src}(M_{11}x + M_{12}y + M_{13}, M_{21}x + M_{22}y + M_{23})$$

- Otherwise, the transformation is first inverted with **invertAffineTransform** and then put in the formula above instead of M. The function cannot operate in-place.

**src** input image.

**dst** output image that has the size dsize and the same type as src .

**M**  $2 \times 3$  transformation matrix.

**dsize** size of the output image.

**Flags** Default = **INTER\_LINEAR**. combination of interpolation methods (see **InterpolationFlags**) and the optional flag **WARP\_INVERSE\_MAP** that means that M is the inverse transformation ( dst→src ).

**borderMode** Default=**BORDER\_CONSTANT**. pixel extrapolation method (see **BorderTypes**); when borderMode=**BORDER\_TRANSPARENT**, it means that the pixels in the destination image corresponding to the "outliers" in the source image are not modified by the function.

**borderValue** value used in case of a constant border; by default, it is 0.

# getAffineTransform()

retval=cv.getAffineTransform(src, dst)

17

- Calculates an affine transform from three pairs of the corresponding points.
- The function calculates the  $2 \times 3$  matrix of an affine transform so that:

$$\begin{bmatrix} x'_i \\ y'_i \end{bmatrix} = \text{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2$$

## Parameters

**src** Coordinates of triangle vertices in the source image.

**dst** Coordinates of the corresponding triangle vertices in the destination image.

# getRotationMatrix2D()

retval = cv.getRotationMatrix2D(center, angle, scale)

18

- Calculates an affine matrix of 2D rotation.
- The transformation maps the rotation center to itself. If this is not the target, adjust the shift.

<b>center</b>	Center of the rotation in the source image. (가로, 세로)
<b>angle</b>	Rotation angle in degrees. Positive values mean counter-clockwise rotation (the coordinate origin is assumed to be the top-left corner).
<b>scale</b>	Isotropic scale factor.

# invertAffineTransform()

`iM = cv.invertAffineTransform(M[, iM])`

19

- Inverts an affine transformation.
- The function computes an inverse affine transformation represented by  $2 \times 3$  matrix  $M$ :

$$\begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \end{bmatrix}$$

- The result is also a  $2 \times 3$  matrix of the same type as  $M$ .

## Parameters

**M** Original affine transformation.

**iM** Output reverse affine transformation.

# warpPerspective()

`dst = cv.warpPerspective(src, M, dsize[, dst[, flags[, borderMode[, borderValue]]]])`

20

- Applies a perspective transformation to an image.
- The function `warpPerspective` transforms the source image using the specified matrix:
- when the flag `WARP_INVERSE_MAP` is set.

$$\text{dst}(x, y) = \text{src} \left( \frac{M_{11}x + M_{12}y + M_{13}}{M_{31}x + M_{32}y + M_{33}}, \frac{M_{21}x + M_{22}y + M_{23}}{M_{31}x + M_{32}y + M_{33}} \right)$$

- Otherwise, the transformation is first inverted with `invert` and then put in the formula above instead of `M`. The function cannot operate in-place.

<b>src</b>	input image.
<b>dst</b>	output image that has the size <code>dsize</code> and the same type as <code>src</code> .
<b>M</b>	3×3 transformation matrix.
<b>dsize</b>	size of the output image.
<b>flags</b>	combination of interpolation methods ( <code>INTER_LINEAR</code> or <code>INTER_NEAREST</code> ) and the optional flag <code>WARP_INVERSE_MAP</code> , that sets <code>M</code> as the inverse transformation ( <code>dst</code> → <code>src</code> ).
<b>borderMode</b>	pixel extrapolation method ( <code>BORDER_CONSTANT</code> or <code>BORDER_REPLICATE</code> ).
<b>borderValue</b>	value used in case of a constant border; by default, it equals 0.

# getPerspectiveTransform()

`retval = cv.getPerspectiveTransform(src, dst[, solveMethod])`

21

- Calculates a perspective transform from four pairs of the corresponding points.
- The function calculates the  $3 \times 3$  matrix of a perspective transform so that:

$$\begin{bmatrix} t_i x'_i \\ t_i y'_i \\ t_i \end{bmatrix} = \text{map\_matrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

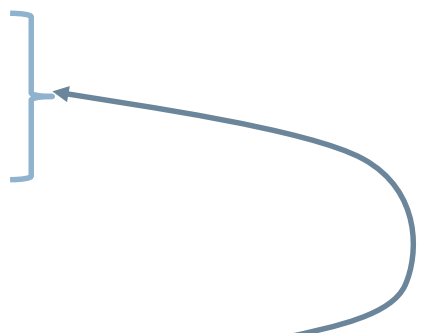
$$dst(i) = (x'_i, y'_i), src(i) = (x_i, y_i), i = 0, 1, 2, 3$$

## Parameters

- src** Coordinates of quadrangle vertices in the source image.
- dst** Coordinates of the corresponding quadrangle vertices in the destination image.
- solveMethod** method passed to [`cv::solve`](#) ([DecompTypes](#))

# 5. 예제 분석

22

- 변환 매트릭스는 어떻게 생성하는가?
  - 변환하는 함수는 무엇인가?
  - 이 두 함수들의 파라미터의 용법은 무엇인가?
  - 역변환 관계는 어떻게 분석하는가?
  - 4개 변환에 대해 개별 예제로 작성하였음
- 

# 5.1 Translation

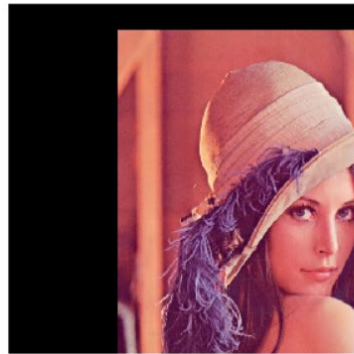
23

GeoTrans\_1\_Translation.py

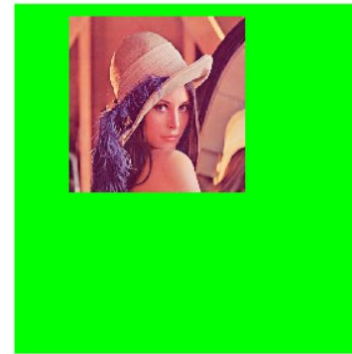
1) (512, 512)  
Original



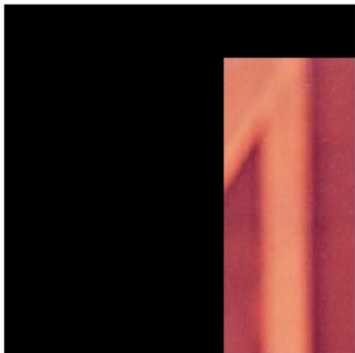
2) (512, 512)  
Translation



3) (1024, 1024)  
Translation



4) (256, 256)  
Translation



5) (512, 512)  
Inv Trans



6) (1024, 1024)  
Inv Trans

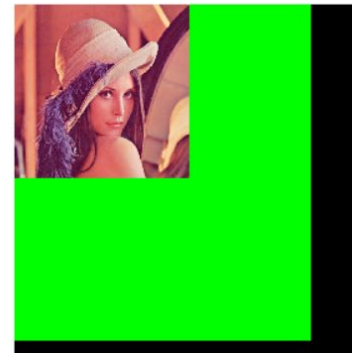


Fig.2: 1) identity= (2, 2)

```
[[1. 0.]  
 [0. 1.]]
```

Fig.2: 2) translation.shape= (2, 1)

```
[[160]  
 [ 40]]
```

Fig.2: 3) translation\_matrix=(2, 3)

```
[[ 1.  0. 160.]  
 [ 0.  1.  40.]]
```

t\_matrix = np.hstack((identity, translation))

Fig.5: inverse translation\_matrix=(2, 3)

```
[[ 1. -0. -160.]  
 [-0.  1. -40.]]
```

Fig.6: inverse translation\_matrix=(2, 3)

```
[[ 1. -0. -160.]  
 [-0.  1. -40.]]
```



# 5.2 Rotation

24

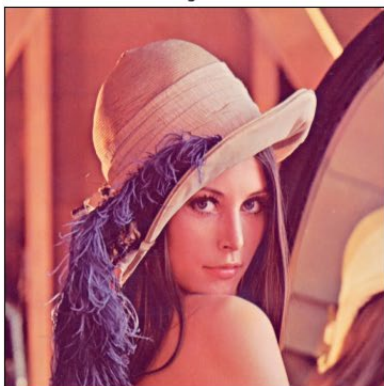
GeoTrans\_2\_Rotation.py

타이틀 표기법    출력 영상의 크기, dsize(가로, 세로)  
회전 각도, scale, center(가로, 세로)

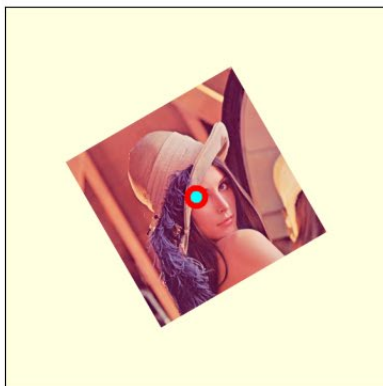
회전 중심점 표시:  
(반시계 방향 회전)



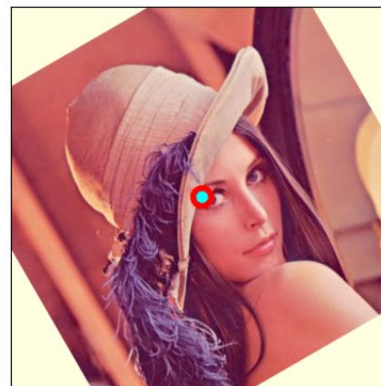
1) size=(512, 512)  
Original



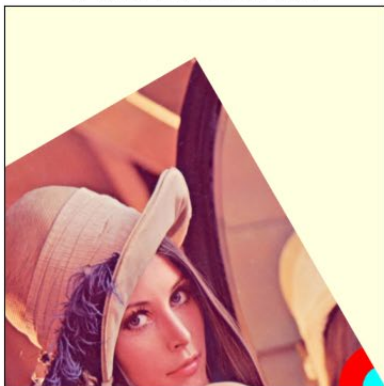
2) size=(512, 512)  
 $\theta=30$ ,  $s=0.5$ ,  $c=(256, 256)$



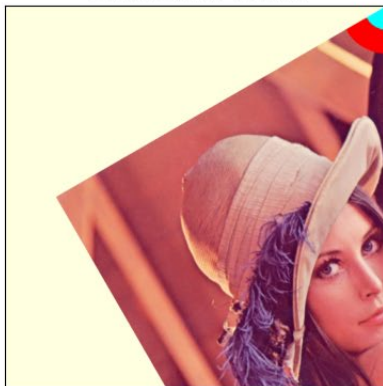
3) size=(512, 512)  
 $\theta=30$ ,  $s=1.0$ ,  $c=(256, 256)$



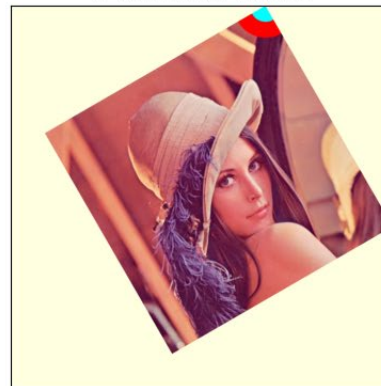
4) size=(512, 512)  
 $\theta=30$ ,  $s=1.0$ ,  $c=(512, 512)$



5) size=(512, 512)  
 $\theta=30$ ,  $s=1.0$ ,  $c=(512, 0)$



6) size=(768, 768)  
 $\theta=30$ ,  $s=1.0$ ,  $c=(512, 0)$

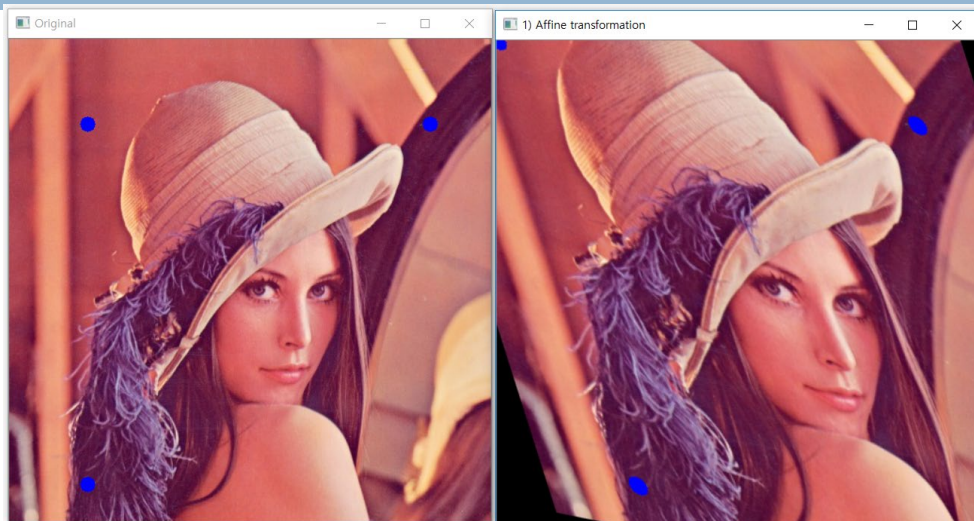


# 5.3 Affine

25

GeoTrans\_3\_Affine.py

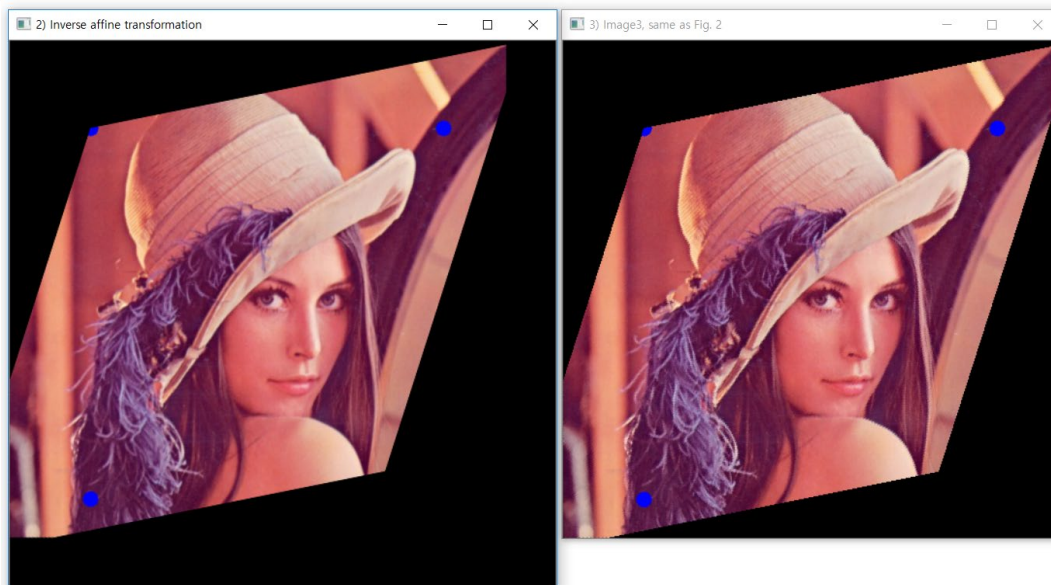
원본



1) 어파인  
변환 결과

```
matrix.shape=(2, 3) matrix.shape=  
[[ 1.22802198  0.39267016 -137.26613831]  
 [ 0.24725275  1.23560209 -131.7261665 ]]
```

2) 어파인  
역변환

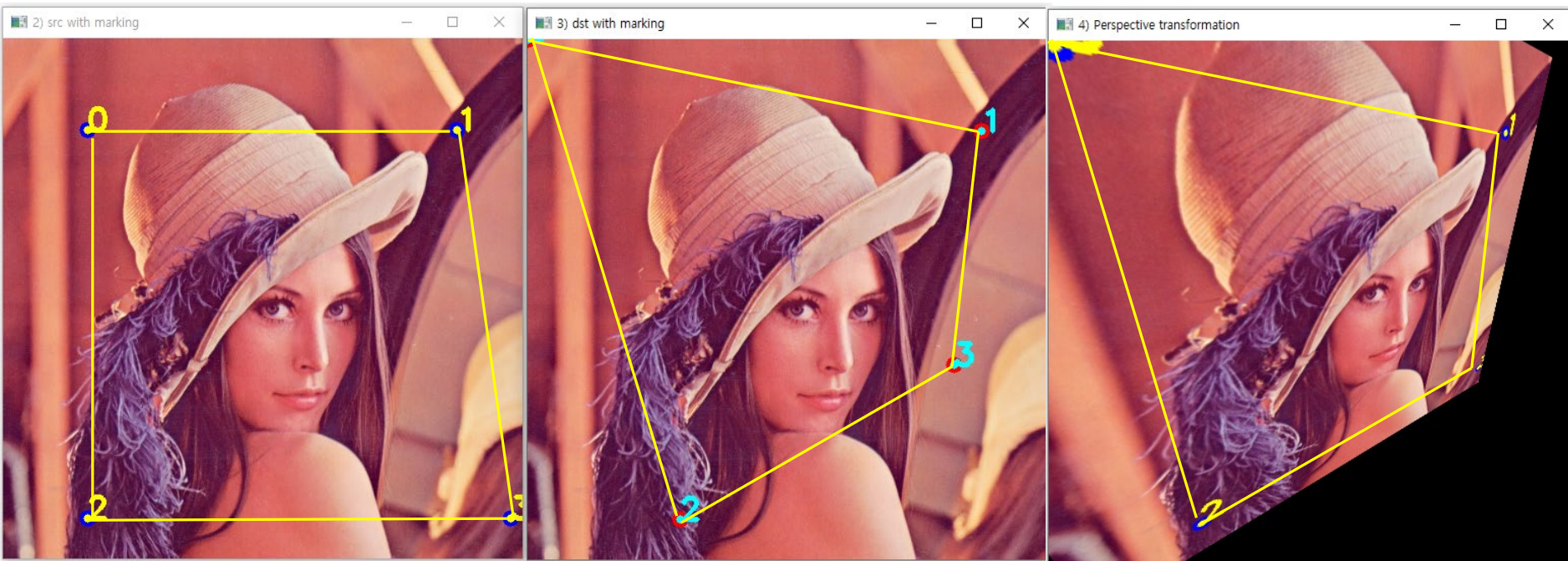


3) 어파인 역변환 결과  
- 1)의 매트릭스를 가지고  
1)의 영상에 대해  
WARP\_INVERSE\_MAP  
플래그로 어파인 변환한  
결과

# 5.4 Perspective

26

GeoTrans\_4\_Perspective.py



```
pts4_src = [(83, 90), (447, 90), (83, 472), (500, 472)]
```

2) Src 영상  
투영할 4개의 포인트

```
pts4_dst = [[0, 0], [447, 90], [150, 472], [420, 320]]
```

3) Dst 영상  
투영될 4개의 포인트

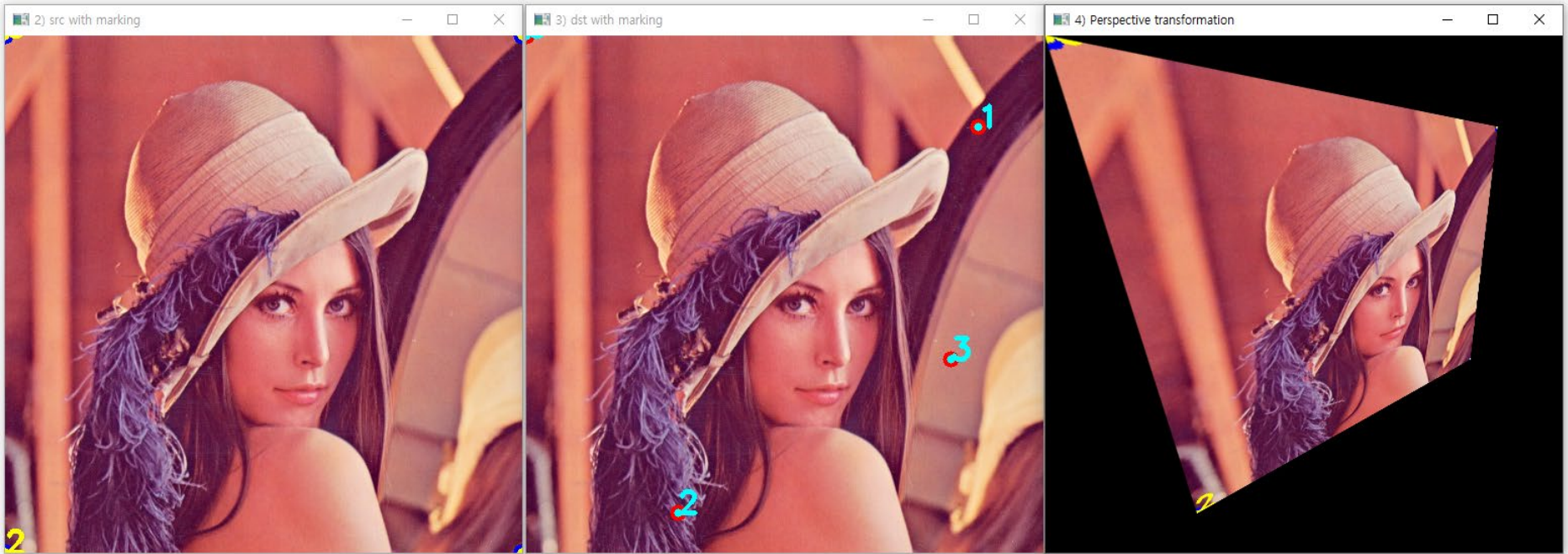
4) 투영변환 결과 영상  
2)의 영상의 포인트를 3)의 포인트로 매핑하는  
투영 매트릭스를 구하여 투영 변환을 실시한  
결과



27

Src를 전 영역을 지정하였을 경우

GeoTrans\_4\_Perspective.py



```
pts4_src = [(0, 0), (cols-1, 0), (0, rows-1), (cols-1, rows-1)] # src, (x, y), 원본 전 영역
```

2) Src 영상  
투영할 4개의 포인트

```
pts4_dst = [[0, 0], [447, 90], [150, 472], [420, 320]]
```

3) Dst 영상  
투영될 4개의 포인트

4) 투영 변환 결과 영상  
2)의 영상의 포인트를 3)의 포인트로 매핑하는  
투영 매트릭스를 구하여 투영 변환을 실시한  
결과

□ 끝...