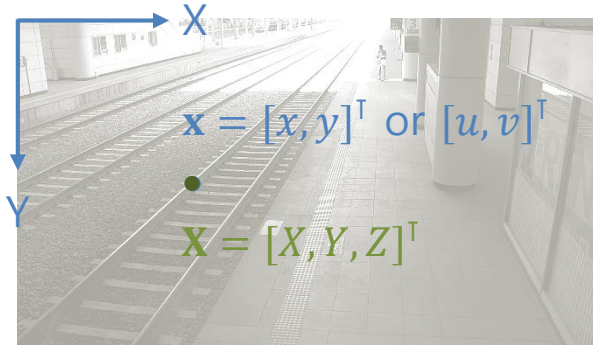


# An Invitation to 3D Vision: Single-View Geometry

Sunglok Choi, Assistant Professor, Ph.D.  
Computer Science and Engineering Department, SEOULTECH  
[sunglok@seoultech.ac.kr](mailto:sunglok@seoultech.ac.kr) | <https://mint-lab.github.io/>

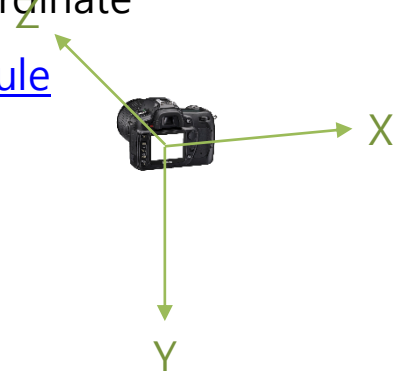
# Getting Started with 2D

- **Image coordinate** (unit: [pixel])



- **Camera coordinate** (unit: [meter])

- Position: [Focal point](#)
- X/Y direction: Image coordinate
- Z direction: [Right-hand rule](#)



# Getting Started with 2D

## ▪ 2D rotation matrix

- Rotational direction: [Right-hand rule](#)



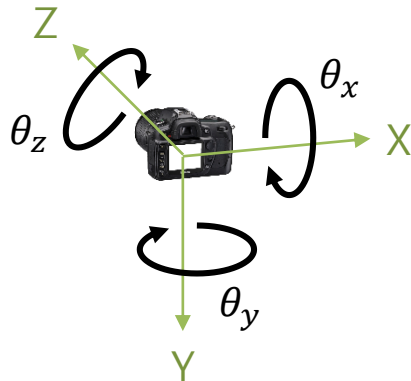
$$R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$$

### Properties of a rotation matrix

- $R^{-1} = R^T$  (orthogonal matrix)
- $\det(R) = 1$

## ▪ 3D rotation matrix

- Rotational direction: [Right-hand rule](#)



	Cameras	Vehicles	Airplanes	Telescopes
$\theta_x$	Tilt	Pitch	Attitude	Elevation
$\theta_y$	Pan	Yaw	Heading	Azimuth
$\theta_z$	Roll	Roll	Bank	Horizon

# Getting Started with 2D

- 3D rotation representation (3 DOF)

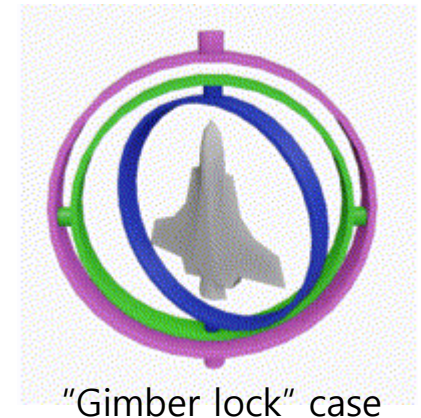
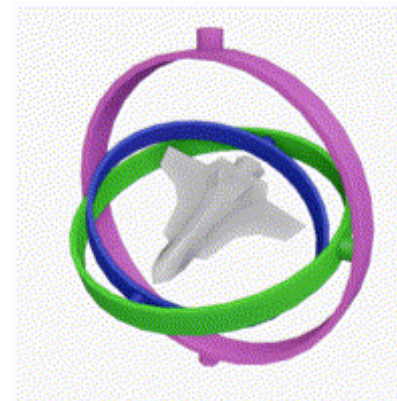
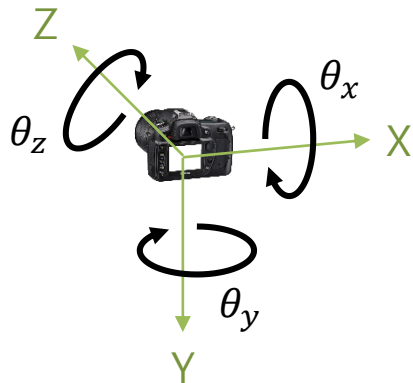
- 3D rotation matrix (9 parameters)

- Notation:  $R = R_z(\theta_z) R_y(\theta_y) R_x(\theta_x) = \begin{bmatrix} \cos \theta_z & -\sin \theta_z & 0 \\ \sin \theta_z & \cos \theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta_y & 0 & \sin \theta_y \\ 0 & 1 & 0 \\ -\sin \theta_y & 0 & \cos \theta_y \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta_x & -\sin \theta_x \\ 0 & \sin \theta_x & \cos \theta_x \end{bmatrix}$

- Properties:  $R^{-1} = R^T$  (orthogonal matrix),  $\det(R) = 1$

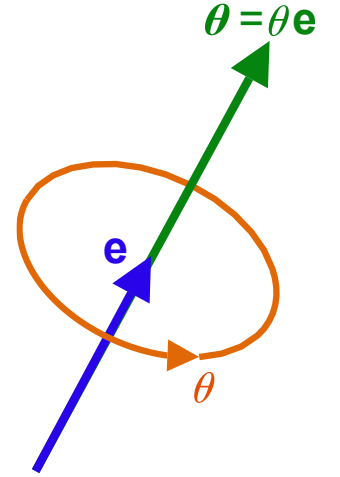
- Euler angle (3 parameters)

- Notation:  $[\theta_x, \theta_y, \theta_z]$
    - Issues: Not unique, not continuous, Gimbar lock (loss of DOF)



# Getting Started with 2D

- 3D rotation representation (3 DOF)
  - Axis-angle representation (3 parameters; a.k.a. *rotation vector*, Rodrigues notation)
    - Notation:  $\boldsymbol{\theta} = \theta \mathbf{e}$ 
      - e.g. Axis:  $\mathbf{e} = [0, 0, 1]$ , angle:  $\theta = \pi/2 \rightarrow \boldsymbol{\theta} = [0, 0, \pi/2]$
    - Properties: Log map of  $\text{SO}(3)$
    - Issues: Not unique ( $-\mathbf{e}$  with  $-\theta$ ), not continuous
    - Note) The standard notation in OpenCV with `cv.Rodrigues()` ( $\mathbf{R} \leftrightarrow \mathbf{rvec}$ )
  - (Unit) Quaternion (4 parameters)
    - Notation:  $[q_x, q_y, q_z, q_w]$  or  $[q_w, q_x, q_y, q_z]$
    - Property:  $q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$
- Note) 3D rotation conversion
  - Python: `scipy.spatial.transform.Rotation`
  - Web apps: NinjaCalc, Glowbuzzer, Andre Gaschler



# Getting Started with 2D

- Example) **3D rotation conversion** [3d\_rotation\_conversion.py]

```
import numpy as np
from scipy.spatial.transform import Rotation

# The given 3D rotation
euler = (45, 30, 60) # Unit: [deg] in XYZ-order

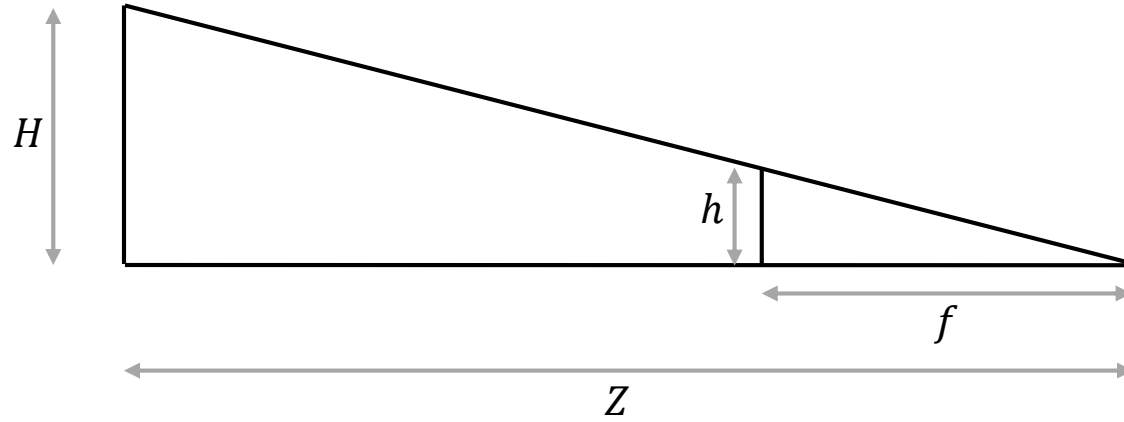
# Generate 3D rotation object
roboj = Rotation.from_euler('zyx', euler[::-1], degrees=True)

# Print other representations
print('\n## Euler Angle (ZYX)')
print(roboj.as_euler('zyx'))
print('\n## Rotation Matrix')
print(roboj.as_matrix())
print('\n## Rotation Vector')
print(roboj.as_rotvec())
print('\n## Quaternion (XYZW)')
print(roboj.as_quat())
```

# Getting Started with 2D

- Similarity

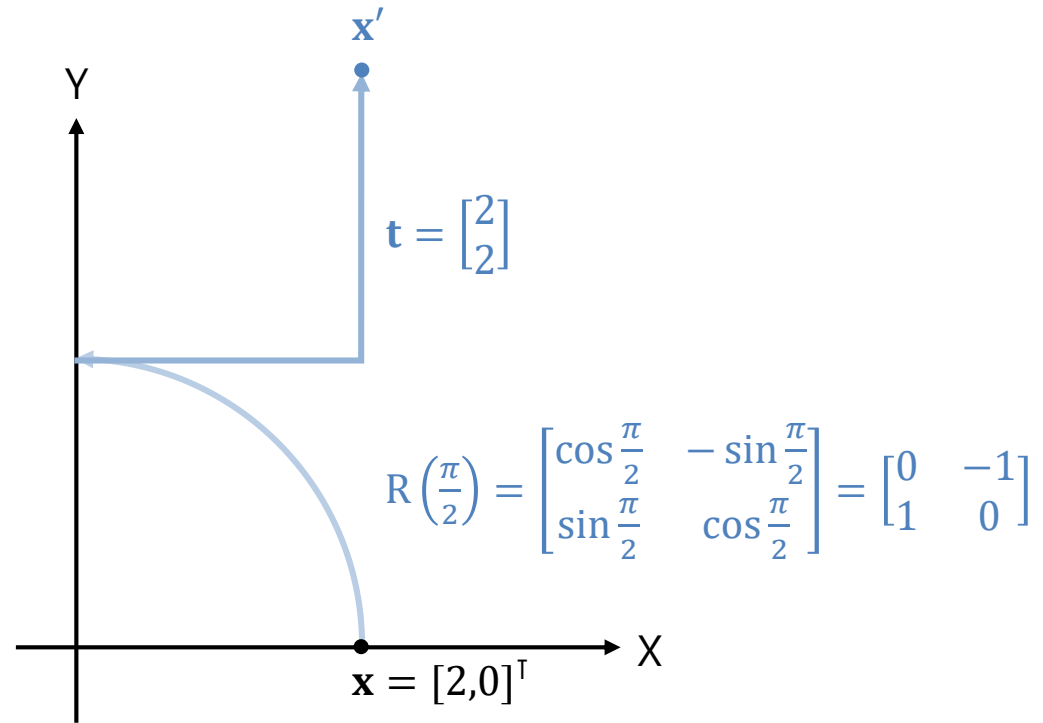
$$\frac{h}{H} = \frac{f}{Z} \quad \text{or} \quad \frac{h}{f} = \frac{H}{Z} \quad \rightarrow \quad h = f \frac{H}{Z}$$



# Getting Started with 2D

- Point transformation

$$\mathbf{x}' = R\mathbf{x} + \mathbf{t} = [R \mid \mathbf{t}] \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$$



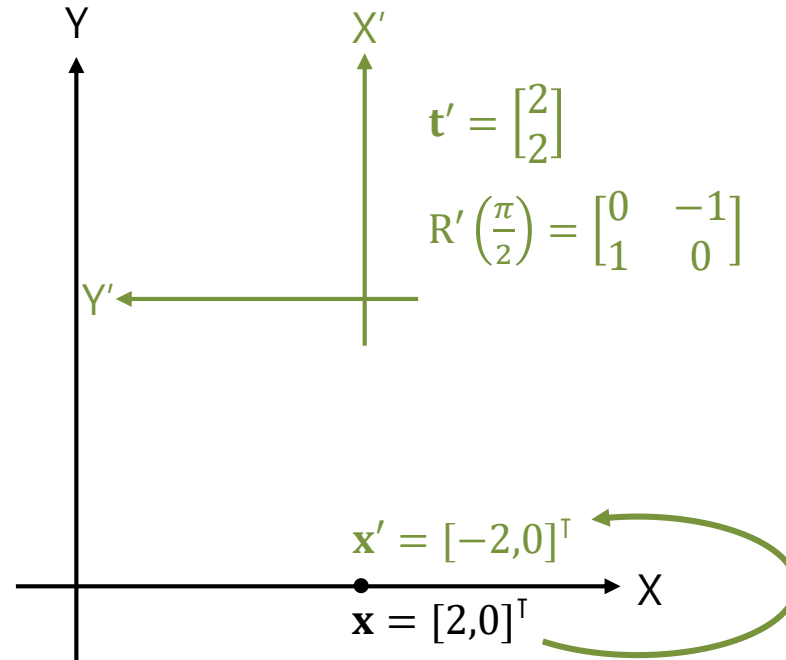


# Getting Started with 2D

- Coordinate transformation

Valid for the following point transformation?

$$\mathbf{x}' = \mathbf{R}'\mathbf{x} + \mathbf{t}' ?$$



# Getting Started with 2D

## Coordinate transformation

Valid for the following point transformation?

$$\mathbf{x}' = \mathbf{R}'\mathbf{x} + \mathbf{t}' ?$$

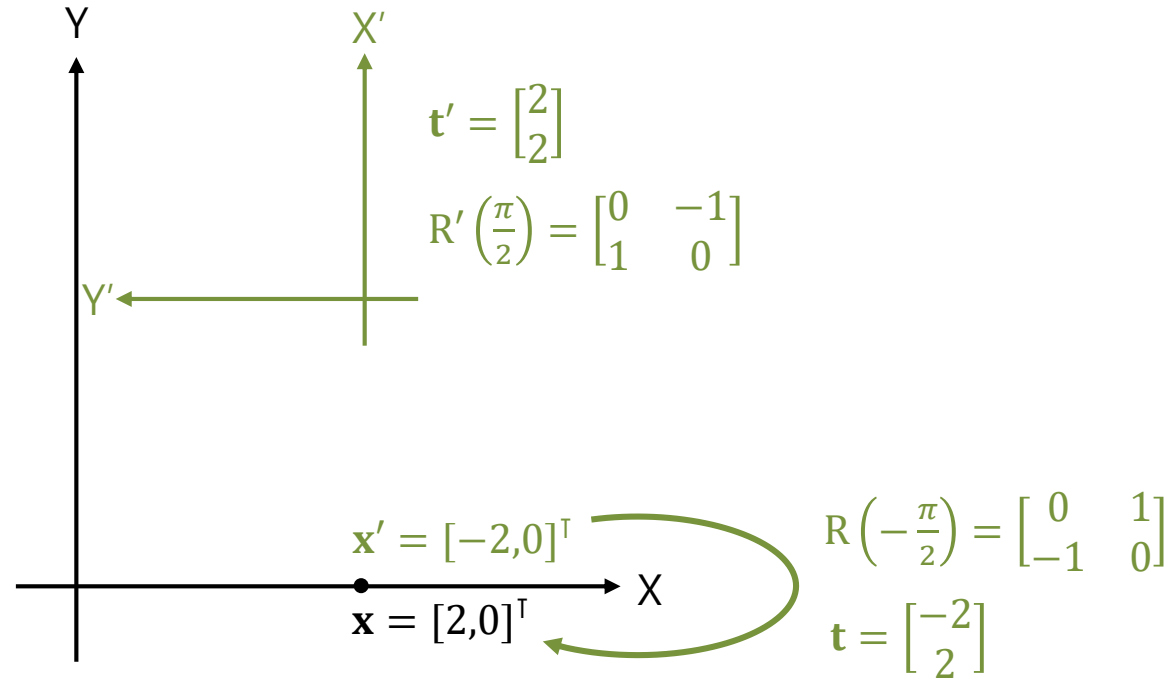
**No!**

The **inverse** of point transformation

$$\mathbf{x} = \mathbf{R}'\mathbf{x}' + \mathbf{t}'$$

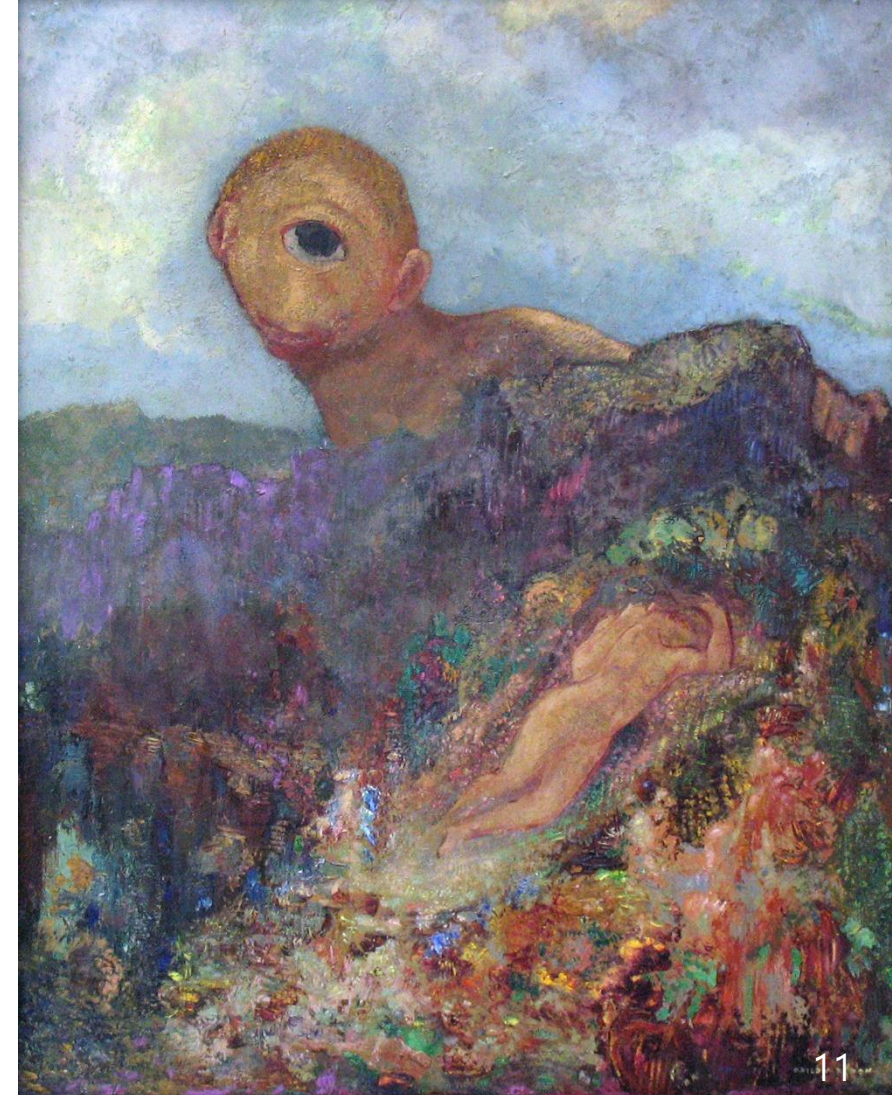
$$\downarrow \mathbf{R}'^T(\mathbf{x} - \mathbf{t}') = \mathbf{x}'$$

$$\mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t} \quad (\mathbf{R} = \mathbf{R}'^T \text{ and } \mathbf{t} = -\mathbf{R}'^T\mathbf{t}')$$



# Table of Contents: **Single-view Geometry**

- **Getting Started with 2D**
  - Coordinate, rotation matrix, 3D rotation representation (rotation vector)
  - Similarity
  - Point transformation, coordinate transformation: **inverse relationship**
- **Camera Projection Models**
  - Pinhole camera model
  - Geometric distortion models
- **Camera Calibration**
- **Absolute Camera Pose Estimation**



*The Cyclops*, gouache and oil by Odilon Redon

# Pinhole Camera Model

- Pinhole camera model



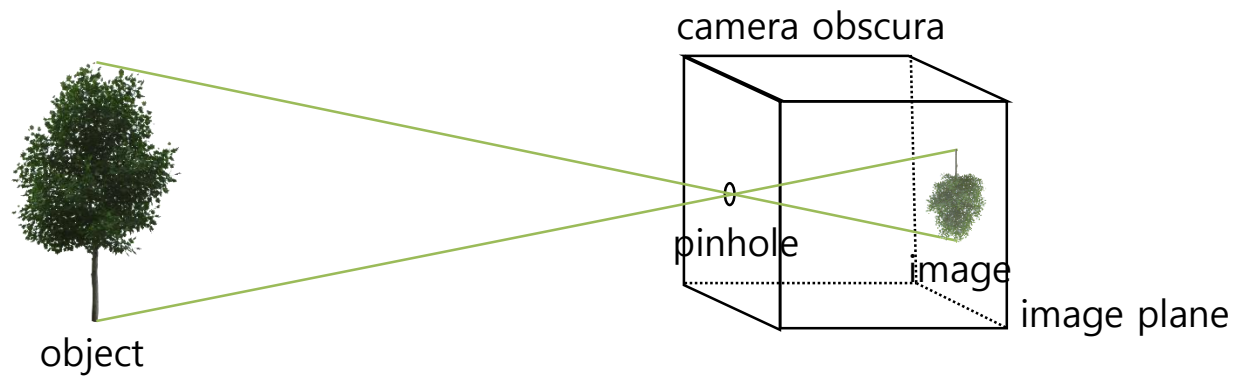
A large-scale camera obscura at San Francisco, California



A modern-day camera obscura



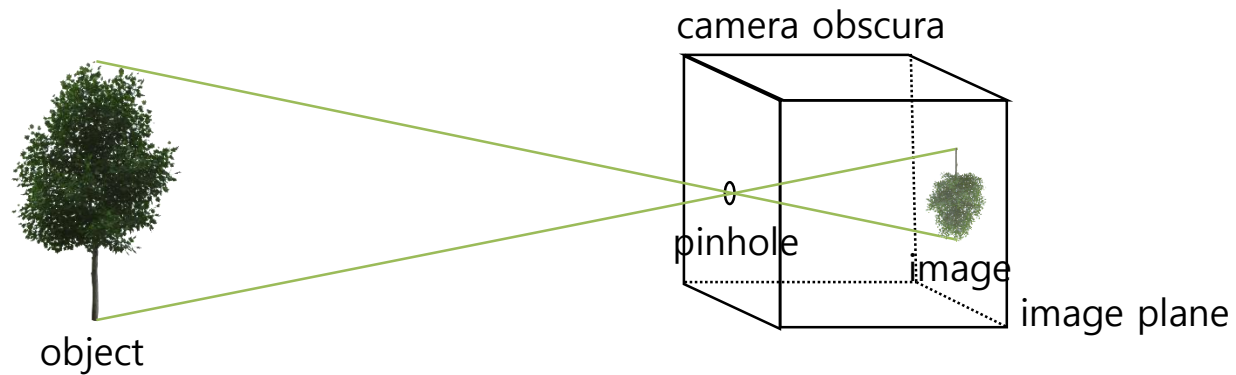
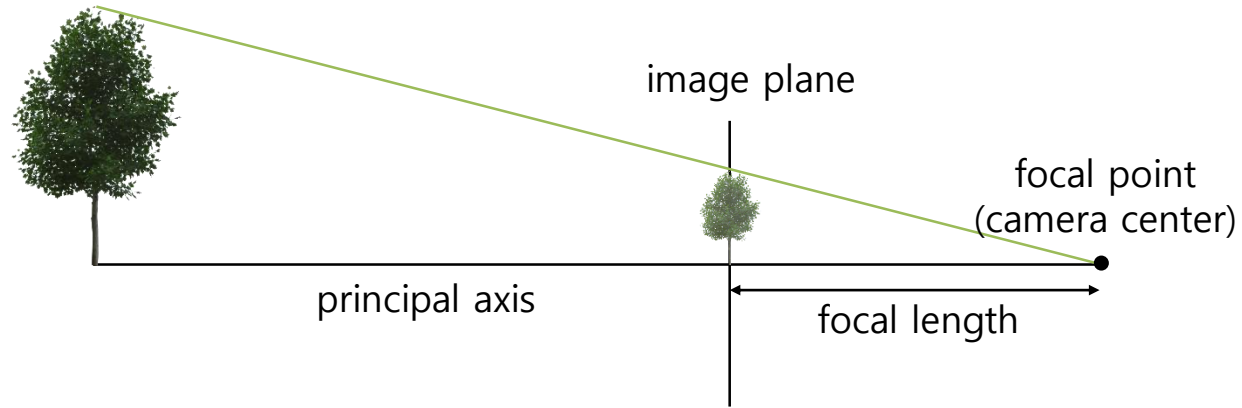
An Image in camera obscura at Portslade, England



# Pinhole Camera Model

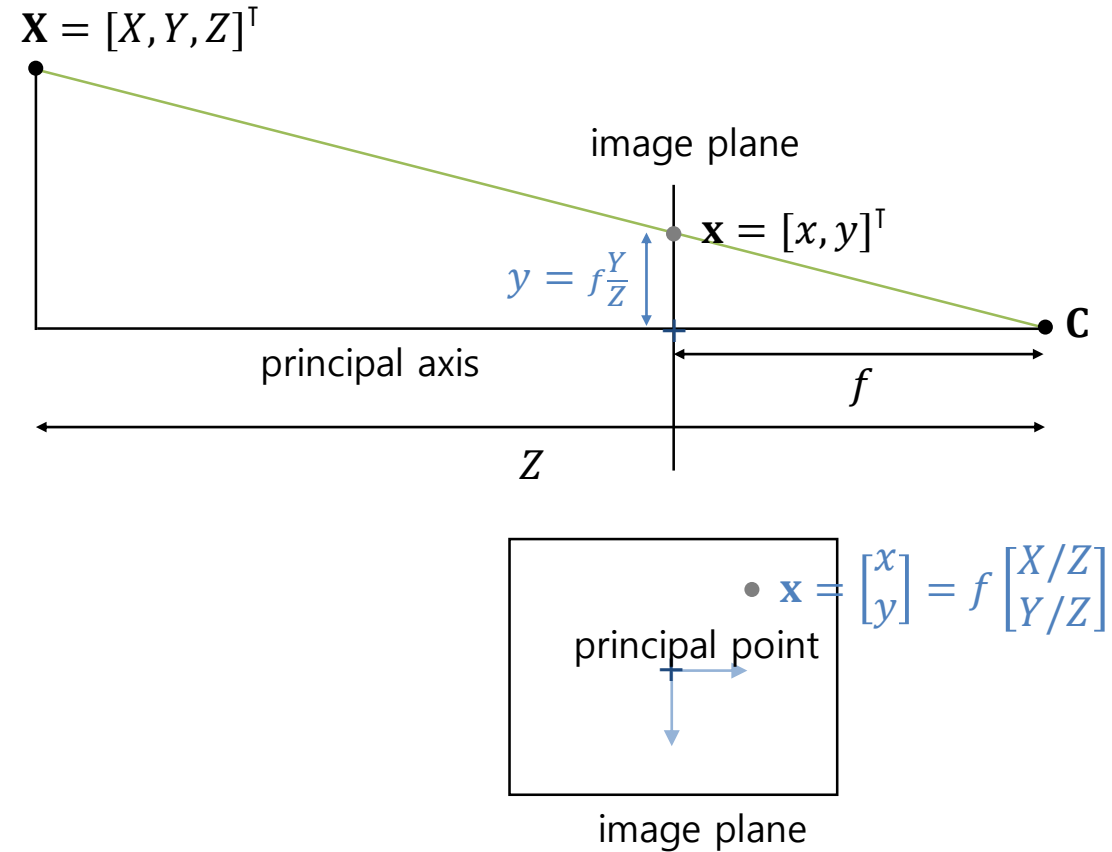
- Pinhole camera model

- In conclusion (without lens distortion),  $\mathbf{x} = \mathbf{P}\mathbf{X}$  ( $\mathbf{P} = \mathbf{K}[\mathbf{R} \mid \mathbf{t}]$ )



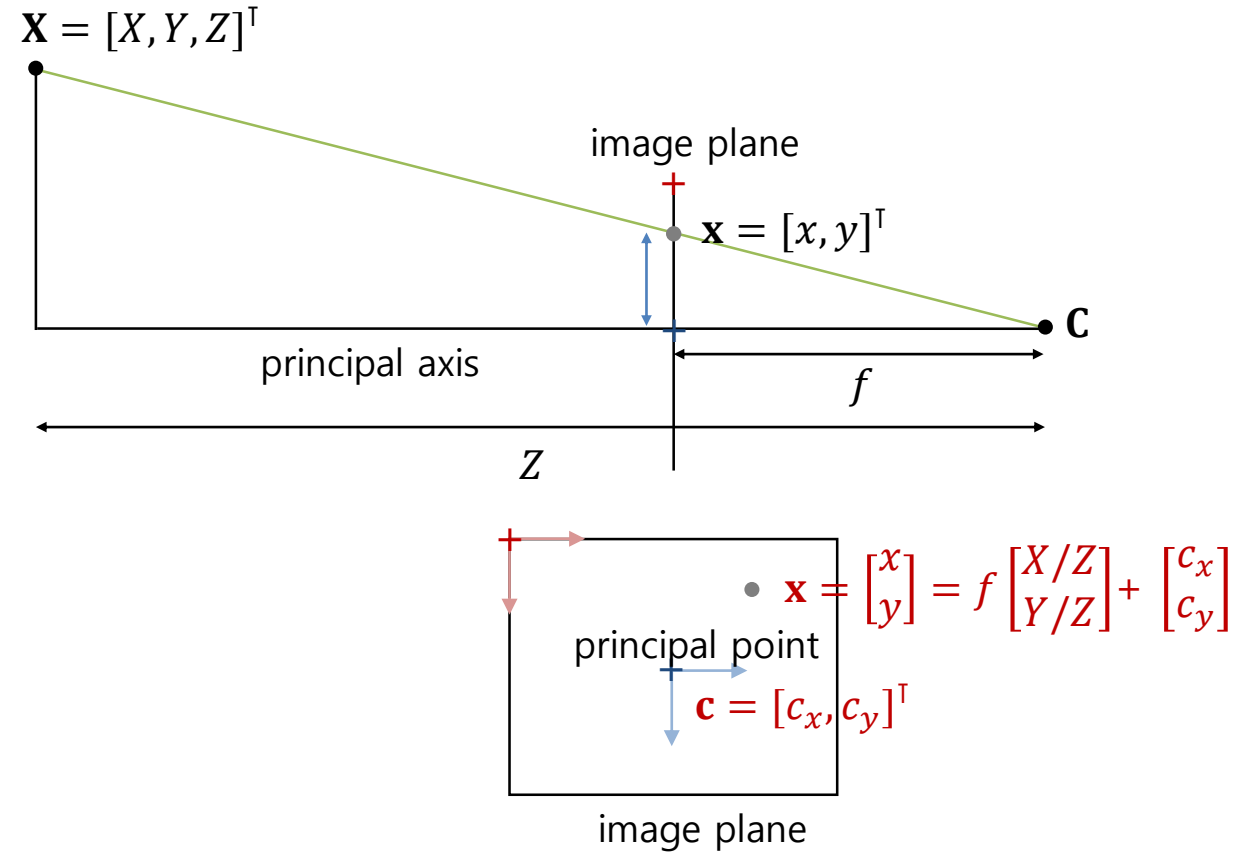
# Pinhole Camera Model

- Pinhole camera model



# Pinhole Camera Model

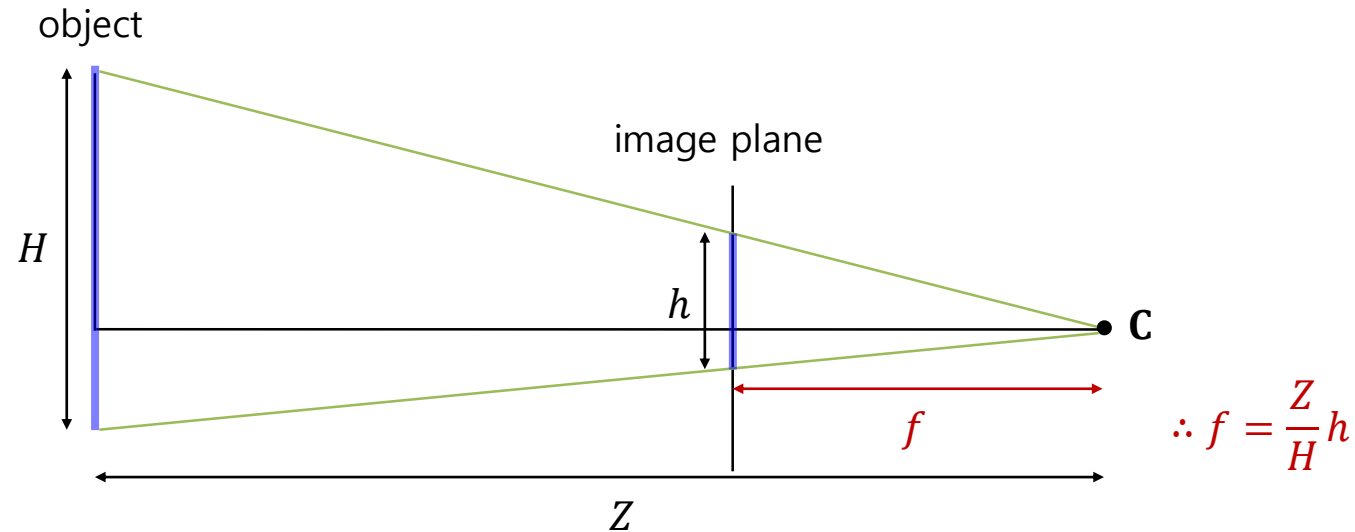
- Pinhole camera model



# Pinhole Camera Model

## ▪ Example) Simple camera calibration

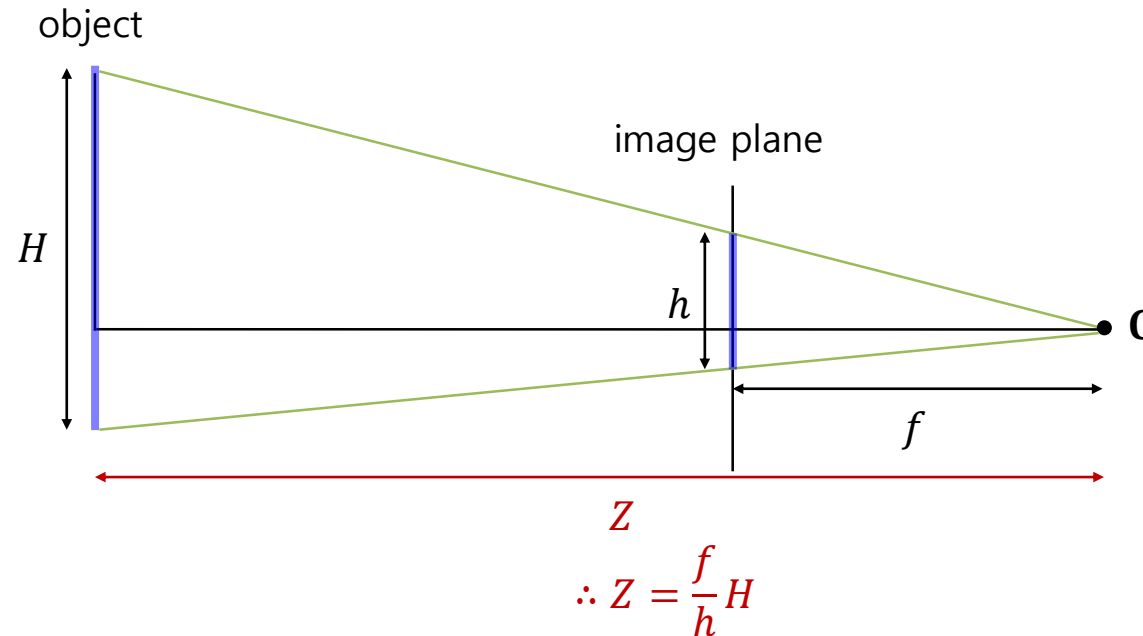
- Unknown: **Focal length ( $f$ )** of the camera (unit: [pixel])
- Given: The observed object height ( $h$ ) on the image plane (unit: [pixel])
- Assumptions
  - The object height ( $H$ ) and distance ( $Z$ ) from the camera are known.
  - The object is aligned with the image plane.





# Pinhole Camera Model

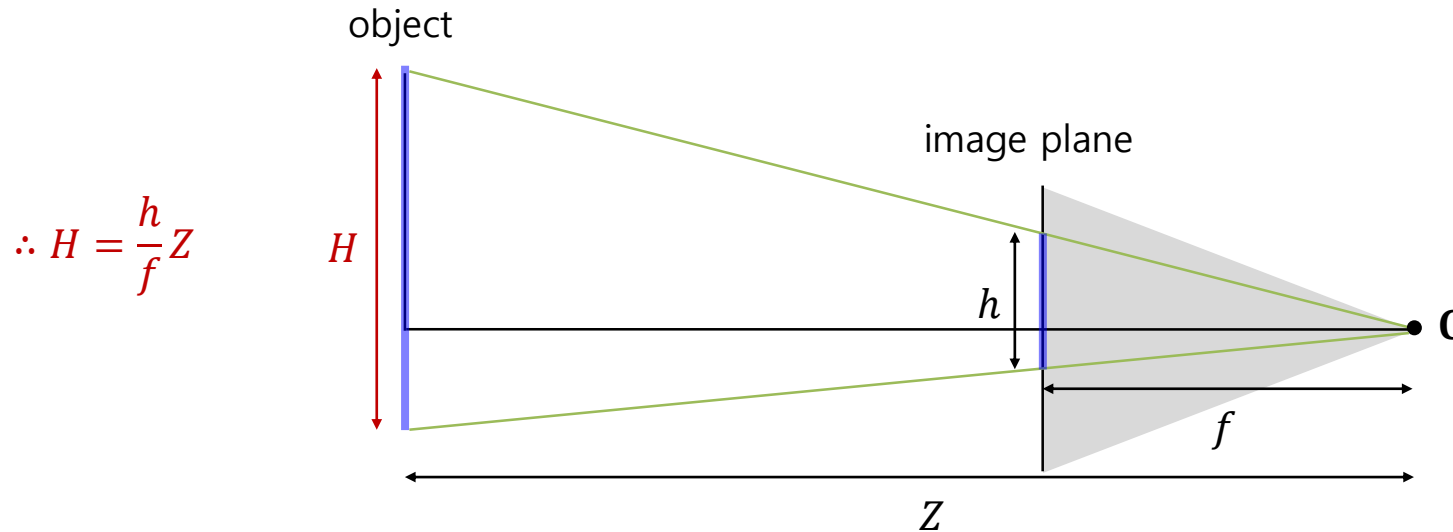
- Example) **Simple depth estimation** (object localization)
  - Unknown: **Object distance ( $Z$ )** from the camera (unit: [m])
  - Given: The observed object height ( $h$ ) on the image plane (unit: [pixel])
  - Assumptions
    - The object height ( $H$ ) and focal length ( $f$ ) are known.
    - The object is aligned with the image plane.



# Pinhole Camera Model

## ▪ Example) **Simple object measurement**

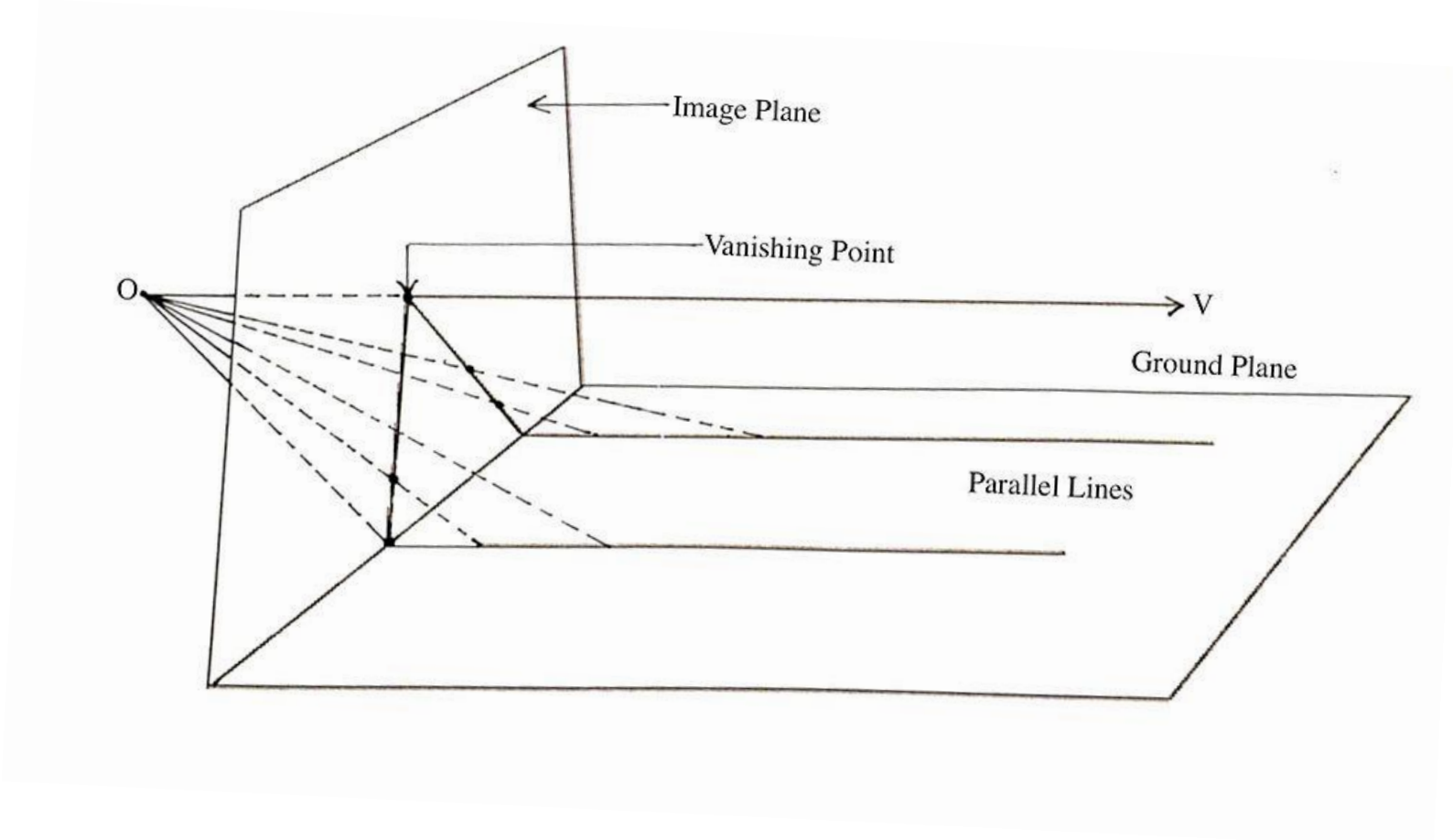
- Unknown: **Object height ( $H$ )** (unit: [m])
- Given: The observed object height ( $h$ ) on the image plane (unit: [pixel])
- Assumptions
  - The object distance ( $Z$ ) from the camera and focal length ( $f$ ) are known.
  - The object is aligned with the image plane.



# Pinhole Camera Model

- Vanishing points

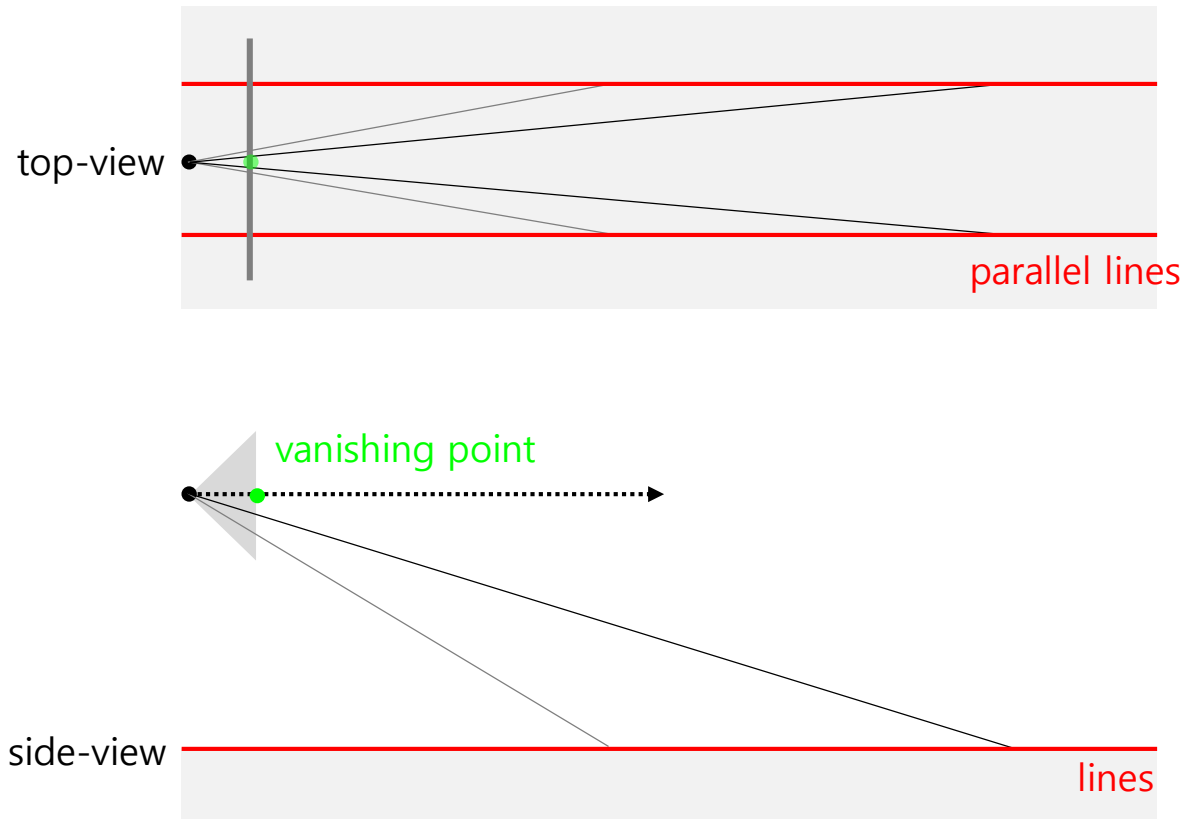
- A point on the image plane where mutually parallel lines in 3D space
  - A vector to the vanishing point is parallel to the lines.
  - A vector to the vanishing point is parallel to the reference plane made by the lines.



# Pinhole Camera Model

## ■ Vanishing points

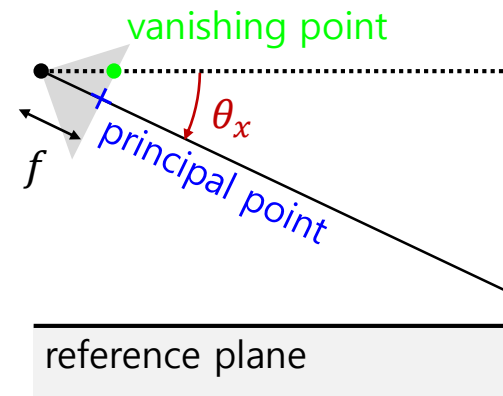
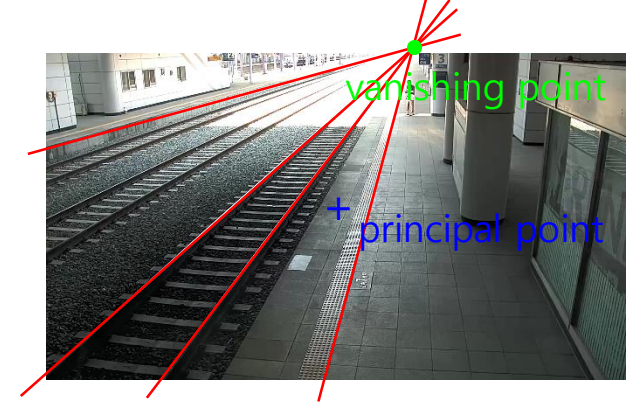
- A point on the image plane where mutually parallel lines in 3D space
  - A vector to the vanishing point is parallel to the lines.
  - A vector to the vanishing point is parallel to the reference plane made by the lines.



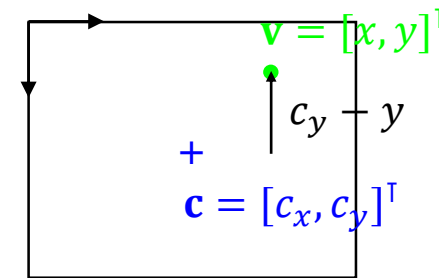
# Pinhole Camera Model

## ▪ Example) Simple camera pose estimation

- Unknown: Tilt angle ( $\theta_x$ ) of the camera w.r.t. the reference plane (unit: [rad])
- Given: A vanishing point ( $x, y$ ) from the reference plane
- Assumptions
  - The focal length ( $f$ ) is known.
  - The principal point ( $c_x, c_y$ ) is known or selected as the center of images.
  - The camera has no roll,  $\theta_z = 0$ .
- Note) The tilt angle in this page is defined as the opposite direction of the common notation.



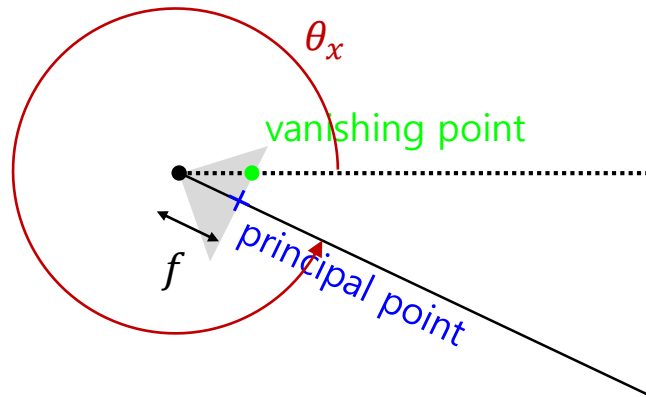
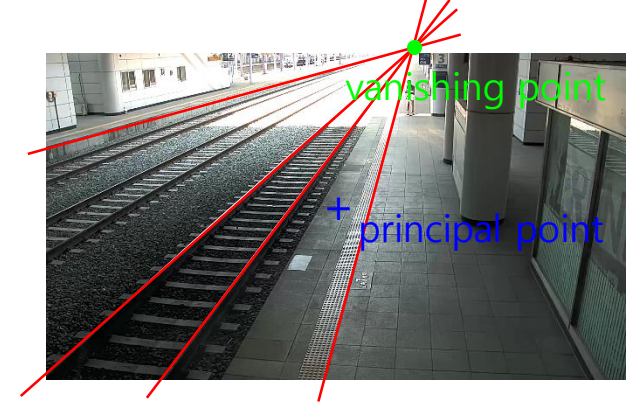
$$\therefore \theta_x = \tan^{-1} \frac{c_y - y}{f}$$



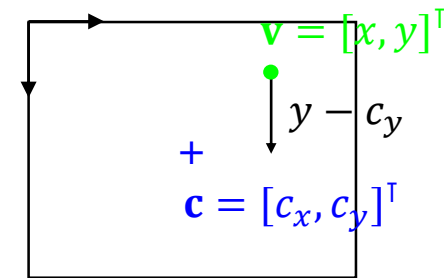
# Pinhole Camera Model

## ▪ Example) Simple camera pose estimation

- Unknown: Tilt angle ( $\theta_x$ ) of the camera w.r.t. the reference plane (unit: [rad])
- Given: A vanishing point ( $x, y$ ) from the reference plane
- Assumptions
  - The focal length ( $f$ ) is known.
  - The principal point ( $c_x, c_y$ ) is known or selected as the center of images.
  - The camera has no roll,  $\theta_z = 0$ .
- Note) The pan angle ( $\theta_y$ ) w.r.t. rails can be calculated similarly using  $x$  instead of  $y$ .



$$\therefore \theta_x = \tan^{-1} \frac{y - c_y}{f}$$





# Pinhole Camera Model

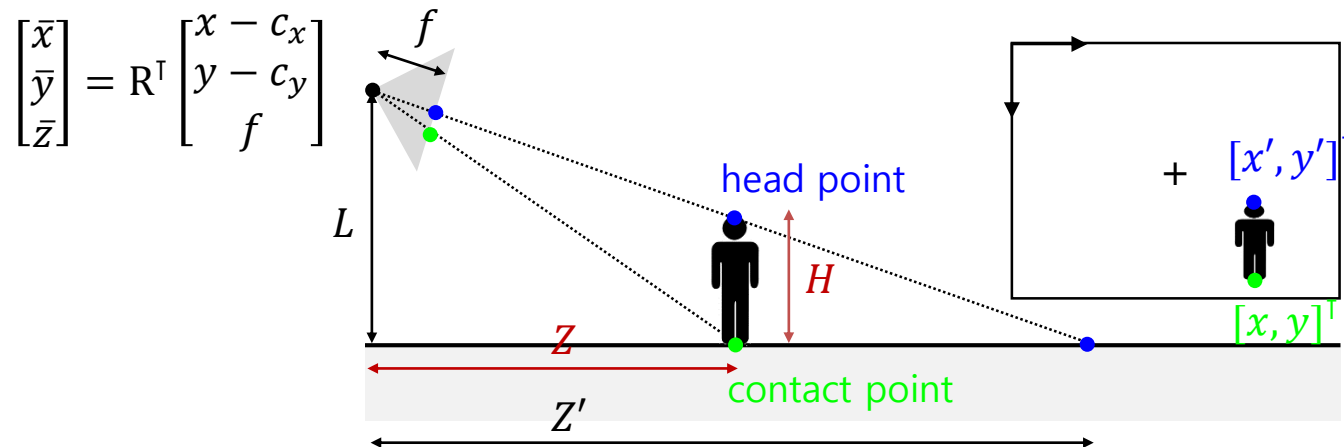
## Example) Object localization #2

- Unknown: **Object position and height** (unit: [m])
- Given: The object's **contact** and **head points** on the image (unit: [pixel])
- Assumptions
  - The focal length, principal points, and camera height are known.
  - ~~The camera is aligned to the reference plane.~~ The camera orientation ( $R$ ) is known.
  - The object is on the reference plane.



$$\therefore Z = \frac{\bar{z}}{\bar{y}} L \quad X = \frac{\bar{x}}{\bar{y}} L \quad H = \left( \frac{\bar{y}}{\bar{z}} - \frac{\bar{y}'}{\bar{z}'} \right) Z$$

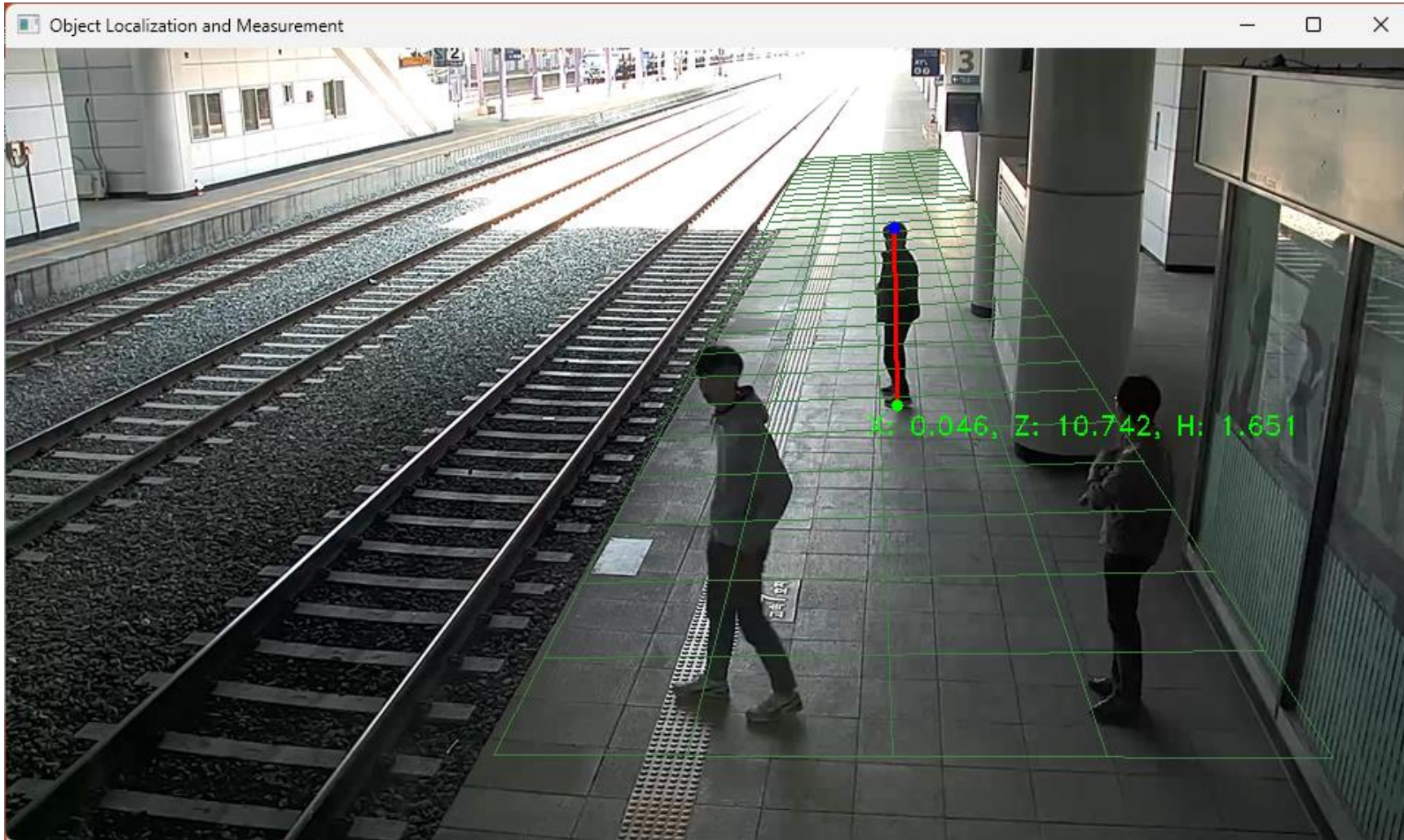
$\therefore$  same camera center





# Pinhole Camera Model

- Example) **Object localization #2** [object\_localization.py]



# Pinhole Camera Model

- Example) **Object localization #2** [object\_localization.py]

```

if __name__ == '__main__':
    ...
    while True:
        img_copy = img.copy()
        if mouse_state['xy_e'][0] > 0 and mouse_state['xy_e'][1] > 0:
            # Calculate object location and height
            c = R.T @ [mouse_state['xy_s'][0] - cx, mouse_state['xy_s'][1] - cy, f]
            h = R.T @ [mouse_state['xy_e'][0] - cx, mouse_state['xy_e'][1] - cy, f]
            if c[1] < 1e-6:
                continue
            X = c[0] / c[2] * L
            Z = c[2] / c[1] * L
            H = (c[1] / c[2] - h[1] / h[2]) * Z
            # Object location X [m]
            # Object location Y [m]
            # Object height [m]

            # Draw the head/contact points and location/height
            cv.line(img_copy, mouse_state['xy_s'], mouse_state['xy_e'], (0, 0, 255), 2)
            cv.circle(img_copy, mouse_state['xy_e'], 4, (255, 0, 0), -1) # Head point
            cv.circle(img_copy, mouse_state['xy_s'], 4, (0, 255, 0), -1) # Contact point
            info = f'X: {X:.3f}, Z: {Z:.3f}, H: {H:.3f}'
            cv.putText(img_copy, info, np.array(mouse_state['xy_s']) + (-20, 20), cv.FONT_HERSHEY_DUPLEX, 0.6, (0, 255, 0))

        cv.imshow('Object Localization and Measurement', img_copy)
        key = cv.waitKey(10)
        if key == 27: # ESC
            break
    
```

$$\begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{bmatrix} = R^T \begin{bmatrix} x - c_x \\ y - c_y \\ f \end{bmatrix}$$

$$X = \frac{\bar{x}}{\bar{y}} L \quad Z = \frac{\bar{z}}{\bar{y}} L \quad H = \left( \frac{\bar{y}}{\bar{z}} - \frac{\bar{y}'}{\bar{z}'} \right) Z$$

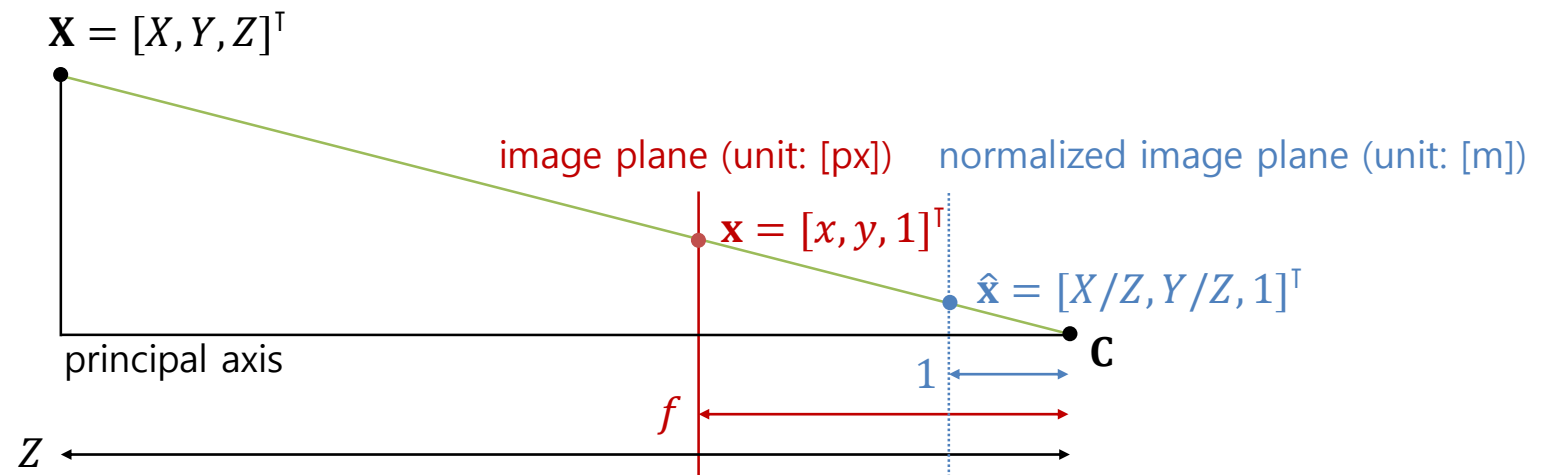
# Pinhole Camera Model

- Camera matrix  $K$

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} = f \begin{bmatrix} X/Z \\ Y/Z \end{bmatrix} + \begin{bmatrix} c_x \\ c_y \end{bmatrix} \rightarrow \mathbf{x} = K\hat{\mathbf{x}} \text{ where } K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \text{ and } \hat{\mathbf{x}} = \begin{bmatrix} X/Z \\ Y/Z \\ 1 \end{bmatrix}$$

$$\text{Simplified as } K = \begin{bmatrix} f & 0 & w/2 \\ 0 & f & h/2 \\ 0 & 0 & 1 \end{bmatrix} \text{ (} w: \text{image width, } h: \text{image height)}$$

$$\text{Generalized as } K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \text{ (} s: \text{skew parameter)}$$

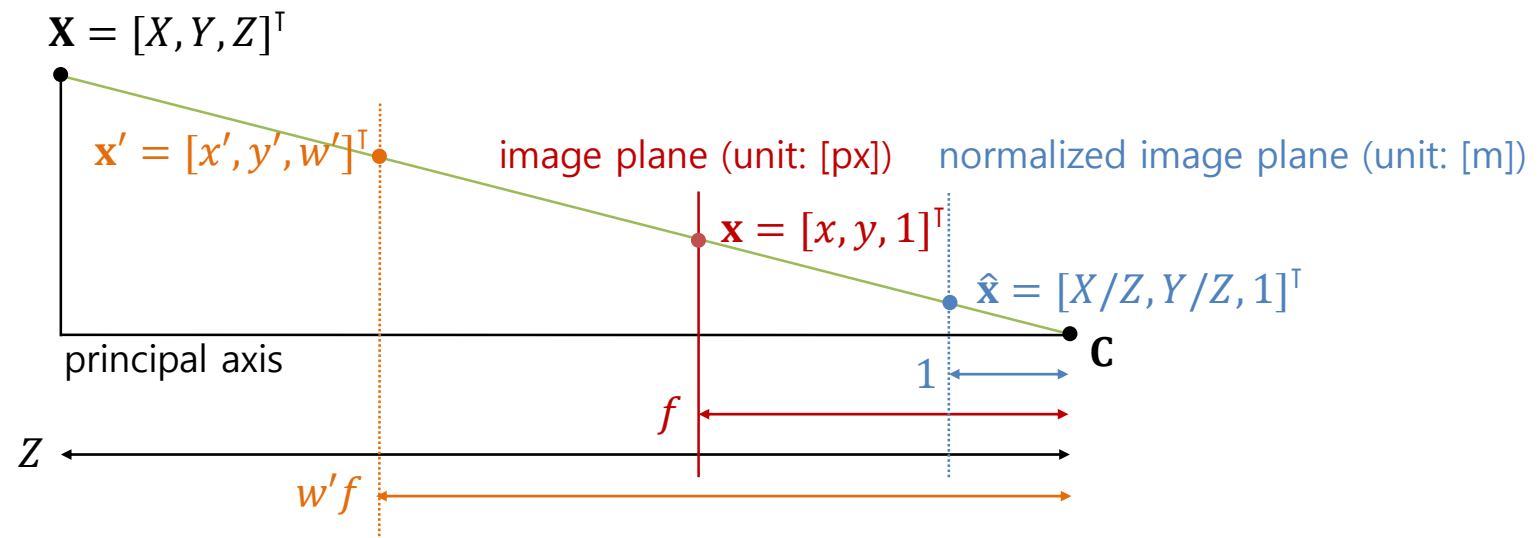


# Pinhole Camera Model

- Homogeneous coordinates (a.k.a. projective coordinates)
  - It describes  $n$ -dimensional project space as  $n + 1$ -dimensional coordinate system.
  - It holds non-conventional equivalence relationship:  $(x_1, x_2, \dots, x_{n+1}) \sim (\lambda x_1, \lambda x_2, \dots, \lambda x_{n+1})$  such that  $(0 \neq \lambda \in \mathbb{R})$ .
    - e.g. (5, 12) is written as (5, 12, 1) which is also equal to (10, 24, 2) or (30, 36, 3) or ...

– On the previous slide,  $\mathbf{x} = K\hat{\mathbf{x}}$  where  $K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$ ,  $\mathbf{x} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$ , and  $\hat{\mathbf{x}} = \begin{bmatrix} X/Z \\ Y/Z \\ 1 \end{bmatrix}$

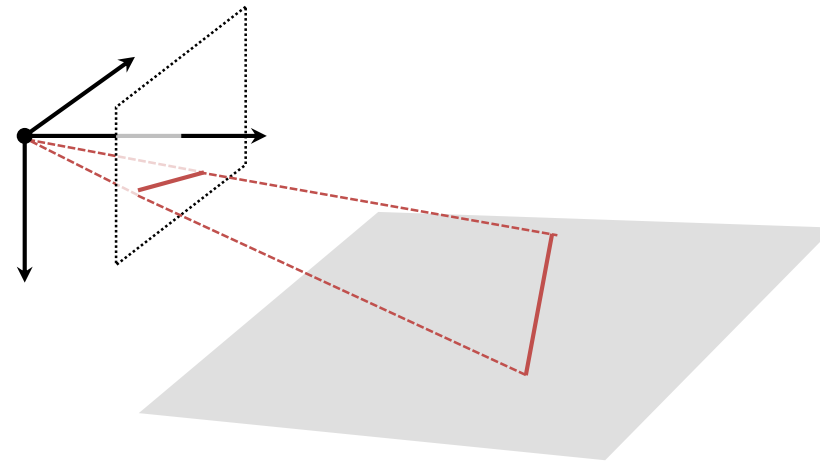
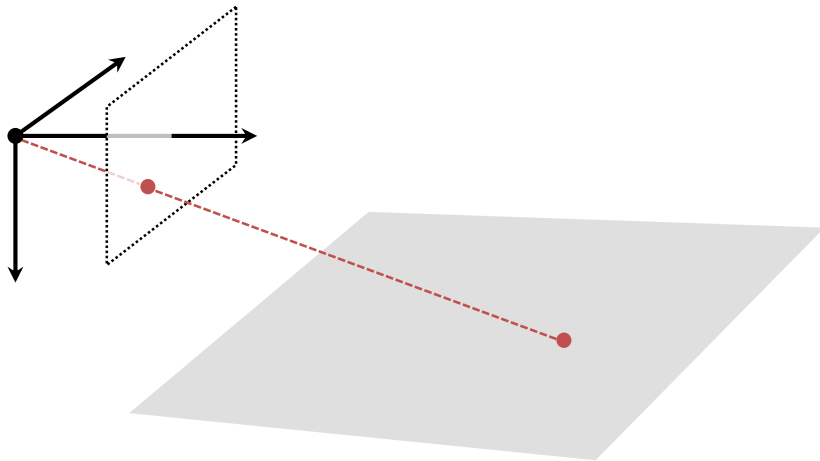
$\mathbf{x}' = K\mathbf{X}$  where  $K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$ ,  $\mathbf{x}' = \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}$ , and  $\mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$  (Note:  $\mathbf{x} = \frac{1}{w'}\mathbf{x}'$ )



# Pinhole Camera Model

## ▪ Why homogeneous coordinates?

- An affine transformation ( $\mathbf{y} = A\mathbf{x} + \mathbf{b}$ ) is formulated by a single matrix multiplication.
- A point at infinity (a.k.a. ideal point) is numerically represented by  $w = 0$ .
- A point and line ( $ax + by + c = 0$ ) are described beautifully as like  $\mathbf{l}^T \mathbf{x} = 0$  or  $\mathbf{x}^T \mathbf{l} = 0$  ( $\mathbf{l} = [a, b, c]^T$ ).
  - Intersection of two lines:  $\mathbf{x} = \mathbf{l}_1 \times \mathbf{l}_2$
  - A line by two points:  $\mathbf{l} = \mathbf{x}_1 \times \mathbf{x}_2$
- A light ray (line at the camera center) is observed as a point on the image plane.
  - A plane at the camera center is observed as a line on the image plane.
  - A conic whose peak is at the camera center is observed as a conic section on the image plane.

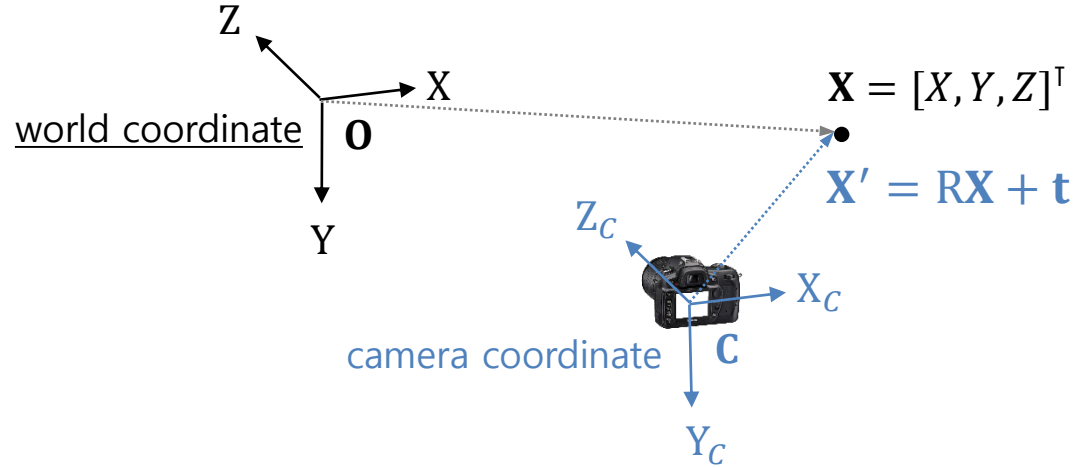


# Pinhole Camera Model

## ▪ Projection matrix P

- Generally, a point  $\mathbf{X}$  is not based on the camera coordinate so that it need be transformed to the camera coordinate.

$$\mathbf{X}' = R\mathbf{X} + \mathbf{t} \rightarrow \mathbf{X}' = [R \mid \mathbf{t}] \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix}$$



- The whole camera projection from the world coordinate to the image coordinate:

$$\mathbf{x} = K\mathbf{X}' = K(R\mathbf{X} + \mathbf{t}) = K [R \mid \mathbf{t}] \begin{bmatrix} \mathbf{X} \\ 1 \end{bmatrix}$$

→  $\mathbf{x} = P\mathbf{X}$  where  $P = K [R \mid \mathbf{t}]$  (3x4 matrix),  $\mathbf{x}$  and  $\mathbf{X}$  in homogenous coordinates

- Note) The camera pose ( $R^T$  and  $-\mathbf{R}^T\mathbf{t}$ ) can be derived from the inverse of point transformation ( $R$  and  $\mathbf{t}$ ).

$$T^{-1} = \begin{bmatrix} R & \mathbf{t} \\ 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} R^T & -R^T\mathbf{t} \\ 0 & 1 \end{bmatrix}$$

# Pinhole Camera Model

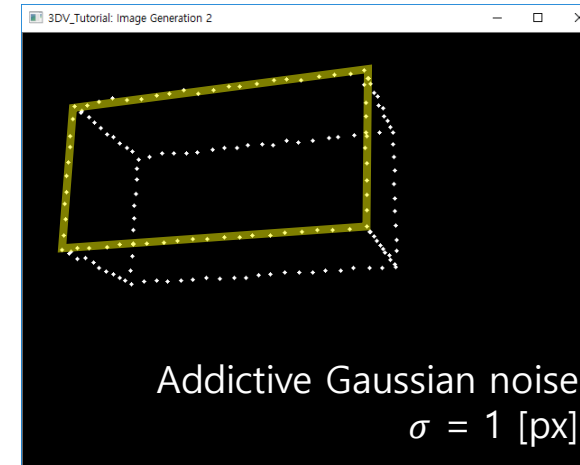
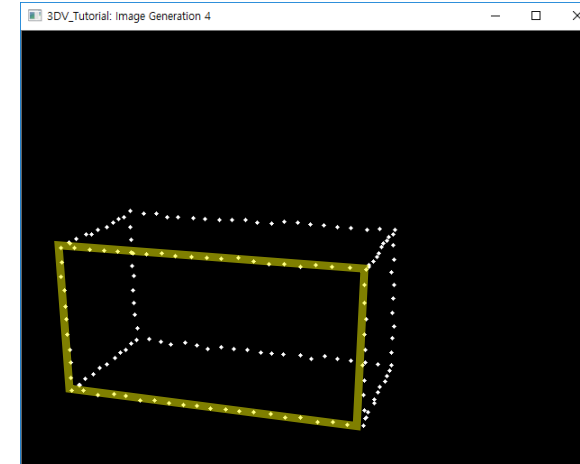
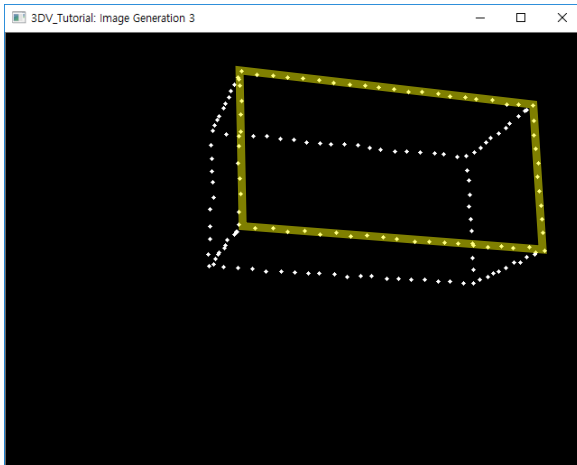
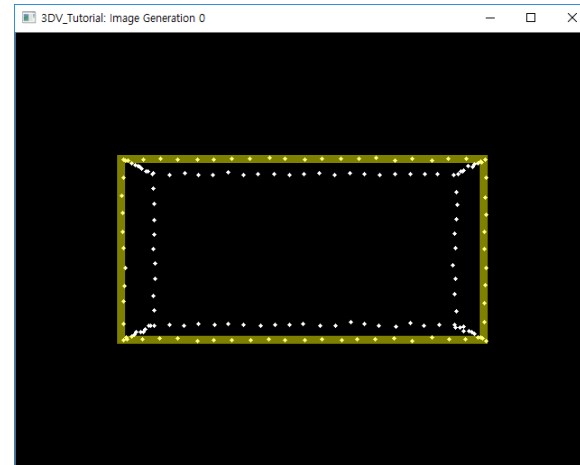
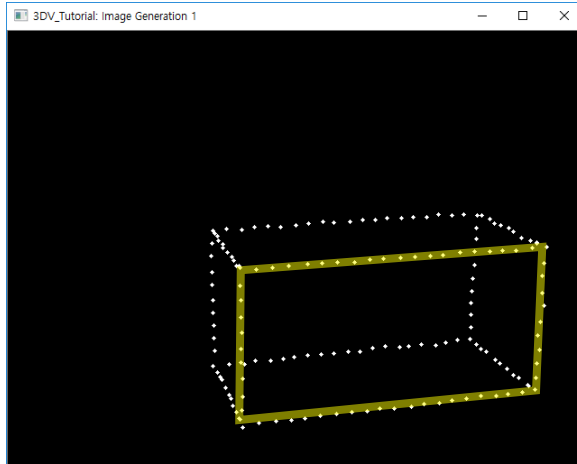
- **Camera parameters** ~ projection matrix  $P$

$$P = K [R \mid t]$$

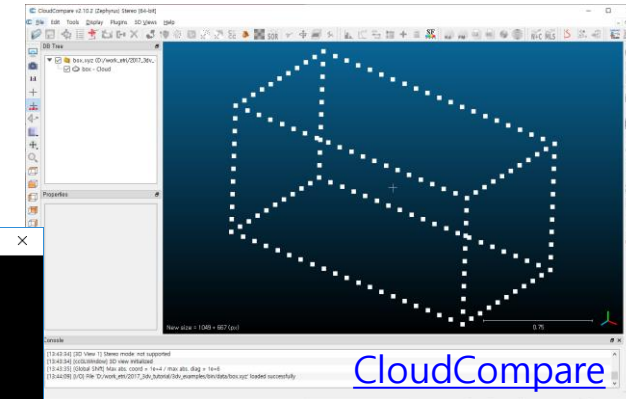
- **Intrinsic parameters** ~ camera matrix  $K$ 
  - e.g. Focal length, principle point, skew, distortion coefficient, ...
- **Extrinsic parameters** ~ point transformation  $R$  and  $t$ 
  - e.g. Rotation and translation

# Pinhole Camera Model

- Example) **Image formation** [image\_formation.py]



A point cloud: data/box.xyz





# Pinhole Camera Model

- Example) **Image formation** [image\_formation.py]

```
from scipy.spatial.transform import Rotation
```

```
# The given camera configuration: Focal length, principal point, image resolution, position, and orientation
```

```
f, cx, cy, noise_std = 1000, 320, 240, 1
```

```
img_res = (640, 480)
```

```
cam_pos = [[0, 0, 0], [-2, -2, 0], [2, 2, 0], [-2, 2, 0], [2, -2, 0]] # Unit: [m]
```

```
cam_ori = [[0, 0, 0], [-15, 15, 0], [15, -15, 0], [15, 15, 0], [-15, -15, 0]] # Unit: [deg]
```

```
# Load a point cloud in the homogeneous coordinate
```

```
X = np.loadtxt('../data/box.xyz') # Size: N x 3
```

```
# Generate images for each camera pose
```

```
K = np.array([[f, 0, cx], [0, f, cy], [0, 0, 1]])
```

```
for i, (pos, ori) in enumerate(zip(cam_pos, cam_ori)):
```

```
    # Derive 'R' and 't'
```

```
    Rc = Rotation.from_euler('zyx', ori[::-1], degrees=True).as_matrix()
```

```
    R = Rc.T
```

```
    t = -Rc.T @ pos
```

```
    # Project the points (Alternative: cv.projectPoints())
```

```
    x = K @ (R @ X.T + t.reshape(-1, 1)) # Size: 3 x N
```

```
    x /= x[-1]
```

```
    # Add Gaussian noise
```

```
    noise = np.random.normal(scale=noise_std, size=(2, len(X)))
```

```
    x[0:2,:] += noise
```

$$\mathbf{K} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \mathbf{R}_c^T \text{ and } \mathbf{t} = -\mathbf{R}_c^T \mathbf{t}_c$$

$$\mathbf{x} = \mathbf{K}(\mathbf{R}\mathbf{X} + \mathbf{t})$$

$$\mathbf{x} = \begin{bmatrix} x/w \\ y/w \\ w/w \end{bmatrix}$$

# Geometric Distortion Models



Q) How to represent such **geometric distortion**?

## ▪ Geometric distortion models

- A camera lens generates geometric distortion, which can be approximated (modeled) as a **nonlinear function**  $f_d$ .
- Geometric distortion models  $f_d$  are mostly defined on **the normalized image plane**.
- Camera projection with **geometric distortion**:  $\mathbf{x} = \text{proj}(\mathbf{X}; \mathbf{K}, \mathbf{R}, \mathbf{t}, d)$  where  $d$  is a set of distortion coefficients.

Note)  $\mathbf{x} = \mathbf{K}(\mathbf{R}\mathbf{X} + \mathbf{t})$  without distortion and normalization

$$\begin{array}{ccccccc} \mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} & \longrightarrow & \mathbf{X}' = \mathbf{R}\mathbf{X} + \mathbf{t} & \longrightarrow & \hat{\mathbf{x}} = \begin{bmatrix} \hat{x} \\ \hat{y} \end{bmatrix} = \begin{bmatrix} X'/Z' \\ Y'/Z' \end{bmatrix} & \longrightarrow & \hat{\mathbf{x}}_d = f_d(\hat{\mathbf{x}}) & \longrightarrow & \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} f_x \hat{x}_d + c_x \\ f_y \hat{y}_d + c_y \end{bmatrix} \\ \text{3D point} & & \text{3D point} & & \text{2D point} & & \text{geometric distortion} & & \text{pinhole camera projection} \\ \text{(the world coordinate)} & & \text{(the camera coordinate)} & & \text{(the normalized image plane)} & & & & \end{array}$$

# Geometric Distortion Models

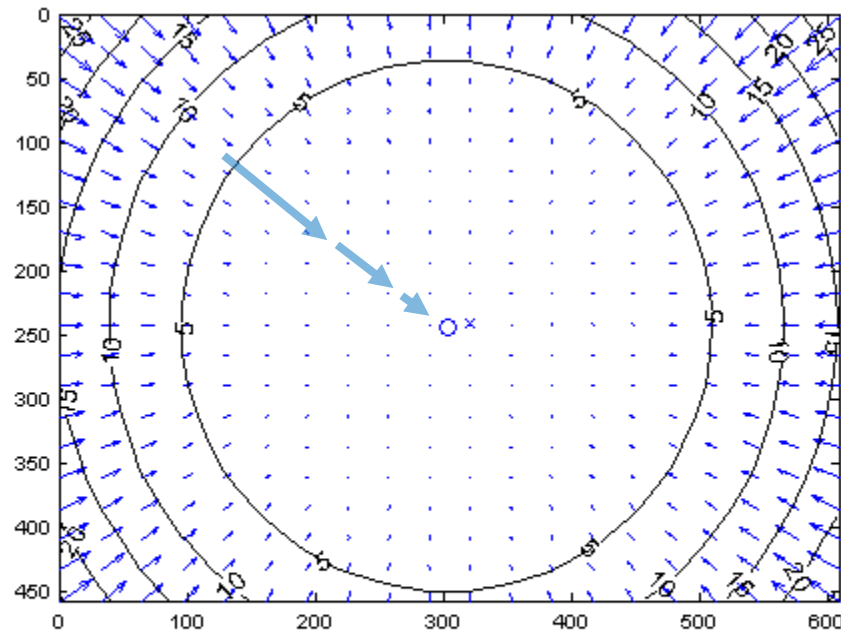
## Geometric distortion models

- Polynomial distortion model (a.k.a. Brown-Conrady model; 1919)

$$\begin{bmatrix} \hat{x}_d \\ \hat{y}_d \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + \dots) \begin{bmatrix} \hat{x} \\ \hat{y} \end{bmatrix} + (1 + p_3 r^2 + p_4 r^4 + \dots) \begin{bmatrix} 2p_1 \hat{x}\hat{y} + p_2(r^2 + 2\hat{x}^2) \\ 2p_2 \hat{x}\hat{y} + p_1(r^2 + 2\hat{y}^2) \end{bmatrix} \quad \text{where } r^2 = \hat{x}^2 + \hat{y}^2$$

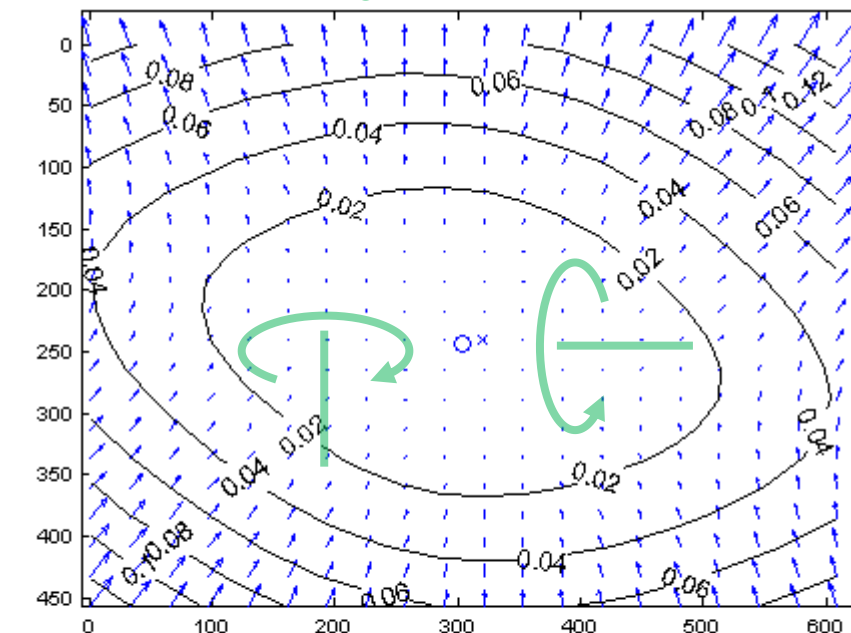
- OpenCV (default): `cv.projectPoints()` ↔ `cv.undistortPoints()`

Radial distortion

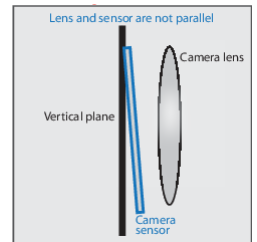


Pixel error = [0.1174, 0.1159]  
 Focal Length = (657.303, 657.744) ± [0.2849, 0.2894]  
 Principal Point = (302.717, 242.334) ± [0.5912, 0.5571]  
 Skew = 0.0004198 ± 0.0001905  
 Radial coefficients = (-0.2535, 0.1187, 0) ± [0.00231, 0.009418, 0]  
 Tangential coefficients = (-0.0002789, 5.174e-005) ± [0.0001217, 0.0001208]

Tangential distortion



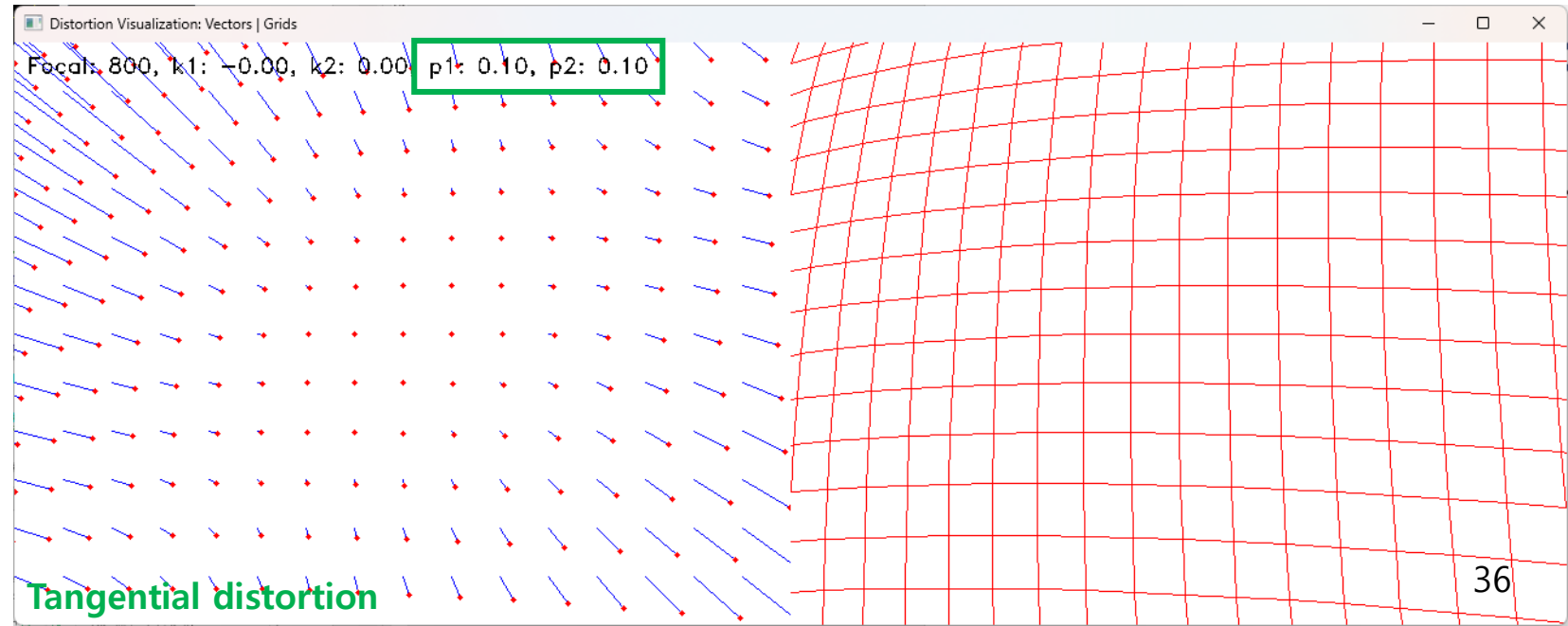
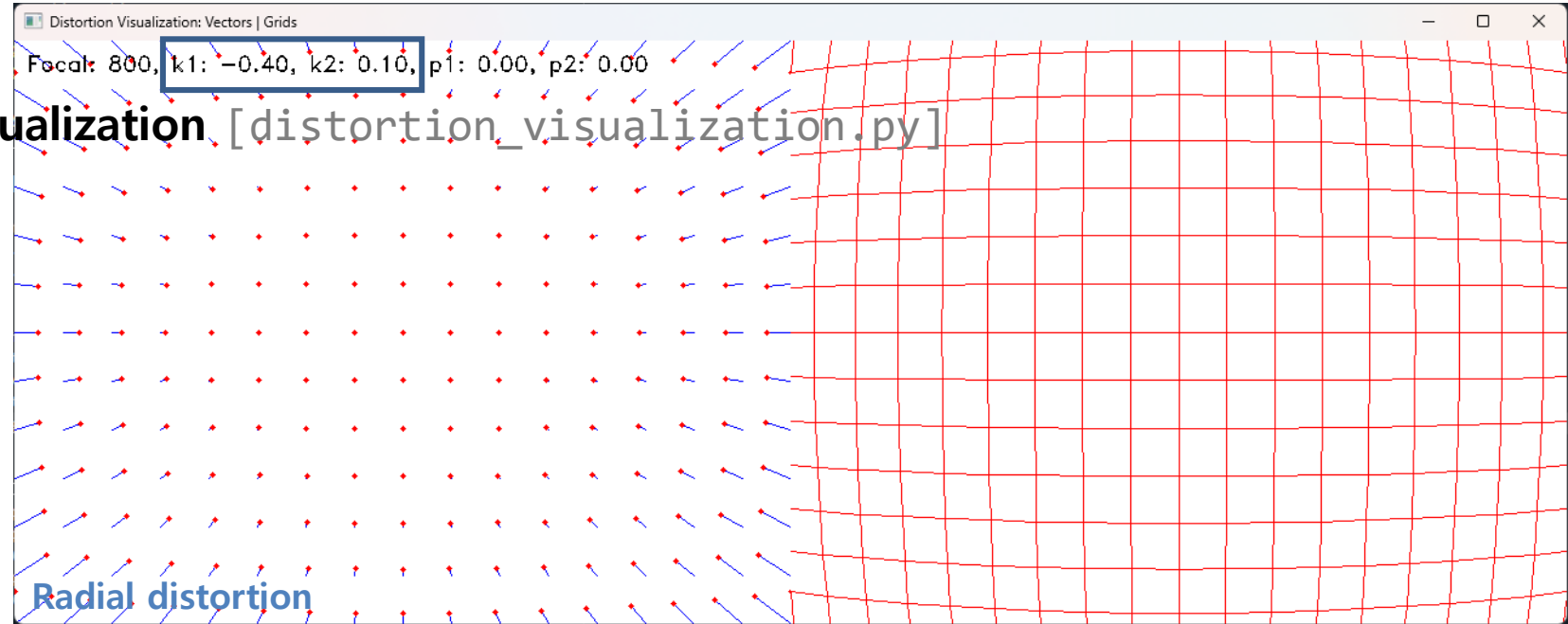
Pixel error = [0.1174, 0.1159]  
 Focal Length = (657.303, 657.744) ± [0.2849, 0.2894]  
 Principal Point = (302.717, 242.334) ± [0.5912, 0.5571]  
 Skew = 0.0004198 ± 0.0001905  
 Radial coefficients = (-0.2535, 0.1187, 0) ± [0.00231, 0.009418, 0]  
 Tangential coefficients = (-0.0002789, 5.174e-005) ± [0.0001217, 0.0001208]



Why?  
 Lens and sensor  
 are not parallel.  
 (usually negligible)

# Geometric Distortion Models

- Example) **Geometric distortion visualization** [distortion\_visualization.py]



# Geometric Distortion Models

- Example) **Geometric distortion visualization** [distortion\_visualization.py]

```
# The initial camera configuration
```

```
img_w, img_h = (640, 480)
```

```
K = np.array([[800, 0, 320],  
              [0, 800, 240],  
              [0, 0, 1.]])
```

```
dist_coeff = np.array([-0.2, 0.1, 0, 0])
```

```
grid_x, grid_y, grid_z = (-18, 19), (-15, 16), 20
```

```
obj_pts = np.array([[x, y, grid_z] for y in range(*grid_y) for x in range(*grid_x)], dtype=np.float32)
```

```
while True:
```

```
    # Project 3D points with/without distortion
```

```
    dist_pts, _ = cv.projectPoints(obj_pts, np.zeros(3), np.zeros(3), K, dist_coeff)
```

```
    zero_pts, _ = cv.projectPoints(obj_pts, np.zeros(3), np.zeros(3), K, np.zeros(4))
```

```
    # Draw vectors
```

```
    img_vector = np.full((img_h, img_w, 3), 255, dtype=np.uint8)
```

```
    for zero_pt, dist_pt in zip(zero_pts, dist_pts):
```

```
        cv.line(img_vector, np.int32(zero_pt.flatten()), np.int32(dist_pt.flatten()), (255, 0, 0))
```

```
    for pt in dist_pts:
```

```
        cv.circle(img_vector, np.int32(pt.flatten()), 1, (0, 0, 255), -1)
```

```
    # Draw grids
```

```
    img_grid = np.full((img_h, img_w, 3), 255, dtype=np.uint8)
```

```
    dist_pts = dist_pts.reshape(len(range(*grid_y)), -1, 2)
```

```
    for pts in dist_pts:
```

```
        cv.polylines(img_grid, [np.int32(pts)], False, (0, 0, 255))
```

# Camera Projection Model

- **Geometric distortion models**

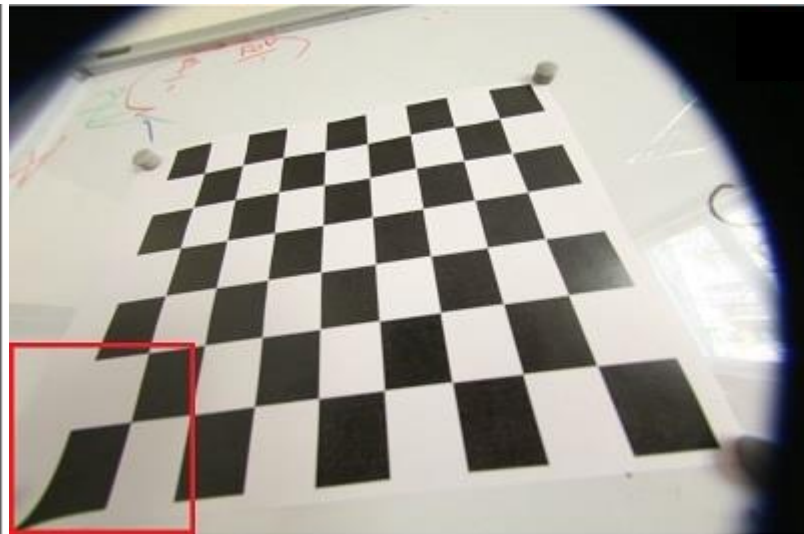
- **Fisheye lens model** (a.k.a. Kannala-Brandt model; [T-PAMI 2006](#))

$$\begin{bmatrix} \hat{x}_d \\ \hat{y}_d \end{bmatrix} = (\theta + k_1\theta^3 + k_2\theta^5 + \dots) \frac{1}{r} \begin{bmatrix} \hat{x} \\ \hat{y} \end{bmatrix} \quad \text{where } r^2 = \hat{x}^2 + \hat{y}^2 \quad \text{and } \theta = \tan^{-1} r$$

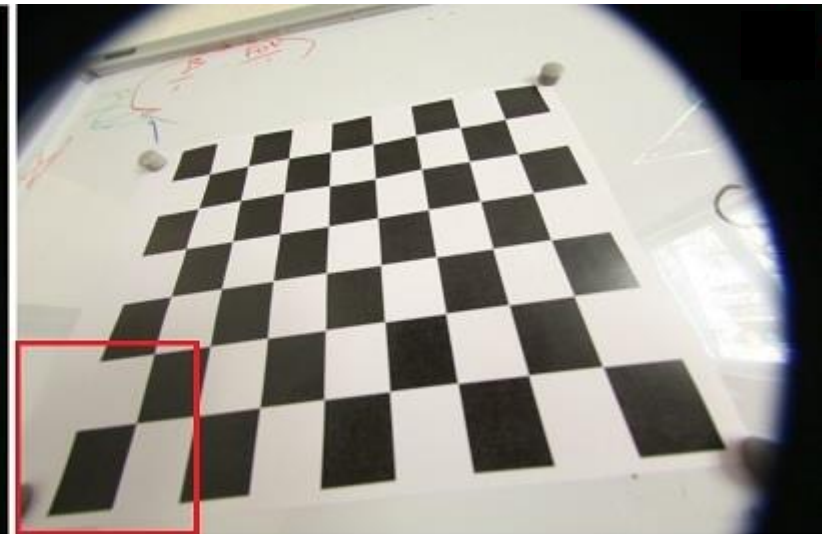
- The fisheye lens model can describe strong barrel distortion **especially around image boundary**.
- OpenCV: `cv.fisheye.projectPoints()` ↔ `cv.fisheye.undistortPoints()`



Original fisheye image



Polynomial distortion model



Fisheye lens model

# Geometric Distortion Models

## ▪ Geometric distortion correction

- Input: The original image
- Output: Its rectified image (without geometric distortion)
- Given: Its camera matrix and distortion coefficient
- Solutions for the polynomial distortion model
  - OpenCV `cv.undistort()` and `cv.undistortPoints()` (Note: included in `imgproc` module)  
↔ `cv.projectPoints()` (Note: included in `calib3d` module)



distortion correction →

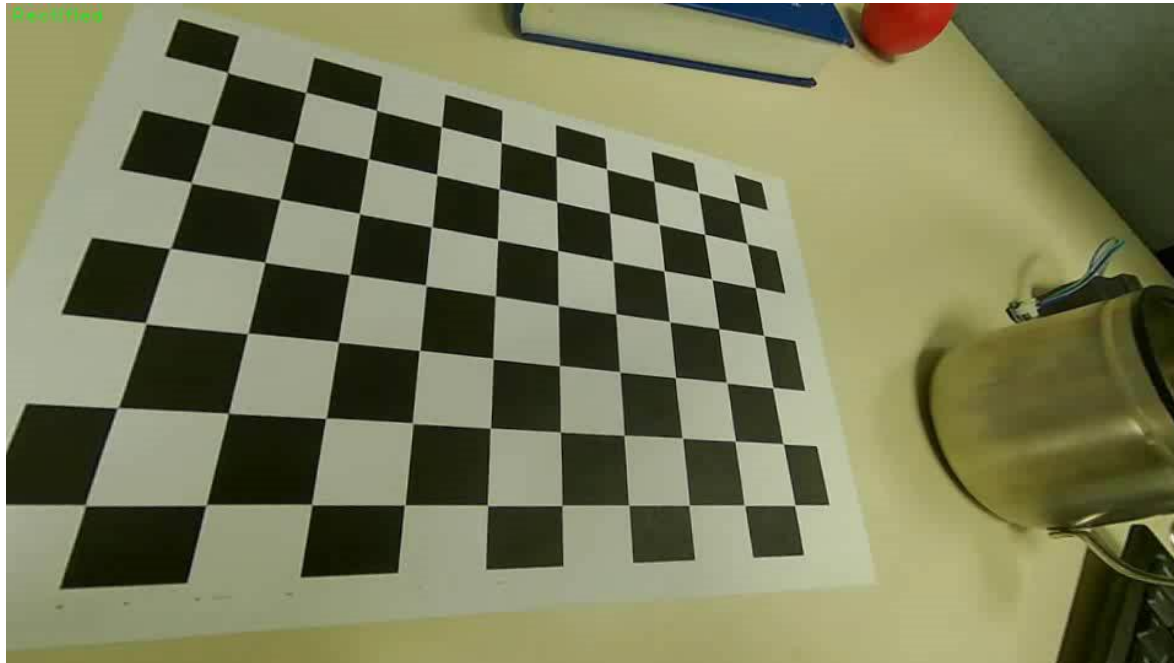
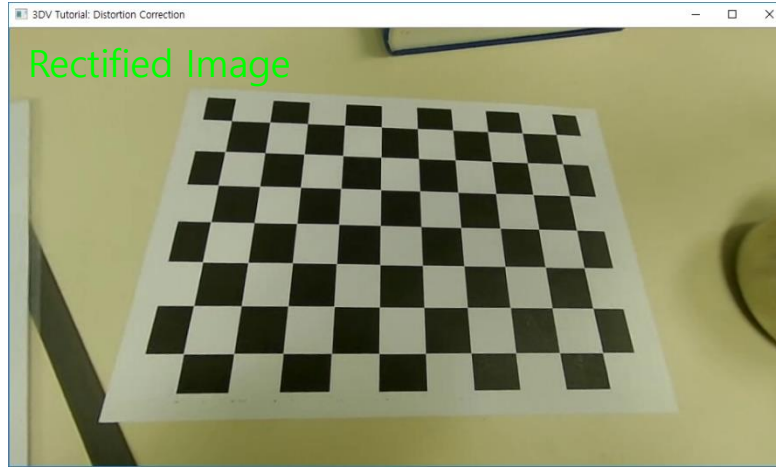
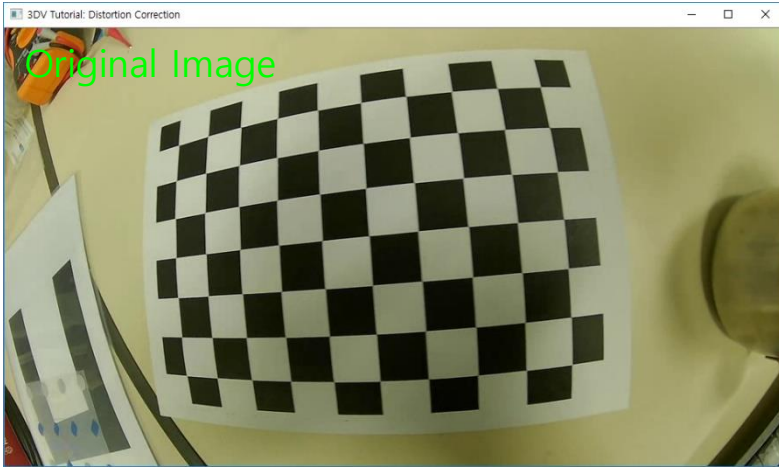
K1: 1.105763E-01  
K2: 1.886214E-02  
K3: 1.473832E-02  
P1: -8.448460E-03  
P2: -7.356744E-03





# Geometric Distortion Models

- Example) **Geometric distortion correction** [distortion\_correction.py]





# Geometric Distortion Models

- Example) **Geometric distortion correction** [distortion\_correction.py]

```
# The given video and calibration data
```

```
input_file = '../data/chessboard.avi'
```

```
K = np.array([[432.7390364738057, 0, 476.0614994349778],  
              [0, 431.239555913084, 288.7602152621297],  
              [0, 0, 1]])
```

```
dist_coeff = np.array([-0.2852754904152874, 0.1016466459919075, -0.0004420196146339175, ...])
```

```
# Open a video
```

```
video = cv.VideoCapture(input_file)
```

```
# Run distortion correction
```

```
show_rectify = True
```

```
map1, map2 = None, None
```

```
while True:
```

```
    # Read an image from the video
```

```
    valid, img = video.read()
```

```
    ...
```

```
    # Rectify geometric distortion (Alternative: cv.undistort())
```

```
    info = "Original"
```

```
    if show_rectify:
```

```
        if map1 is None or map2 is None:
```

```
            map1, map2 = cv.initUndistortRectifyMap(K, dist_coeff, None, None, (img.shape[1], img.shape[0]), cv.CV_32F)
```

```
            img = cv.remap(img, map1, map2, interpolation=cv.INTER_LINEAR)
```

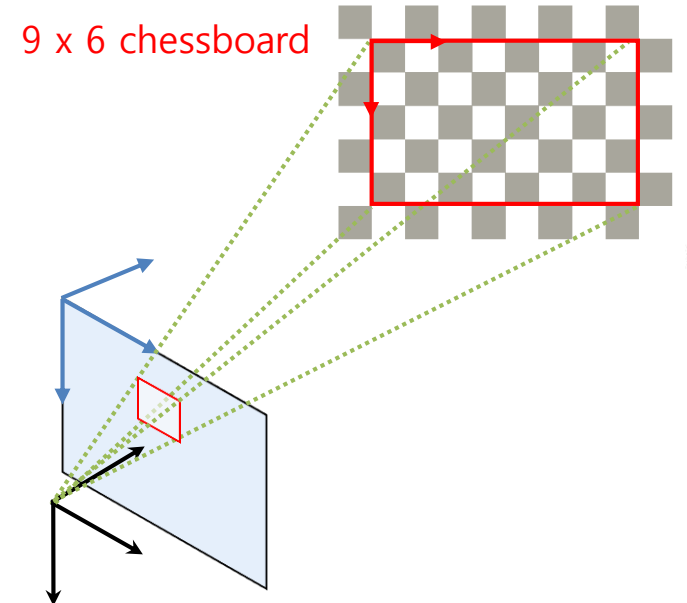
```
            info = "Rectified"
```

```
    cv.putText(img, info, (10, 25), cv.FONT_HERSHEY_DUPLEX, 0.6, (0, 255, 0))
```

# Camera Calibration

## ▪ Camera calibration

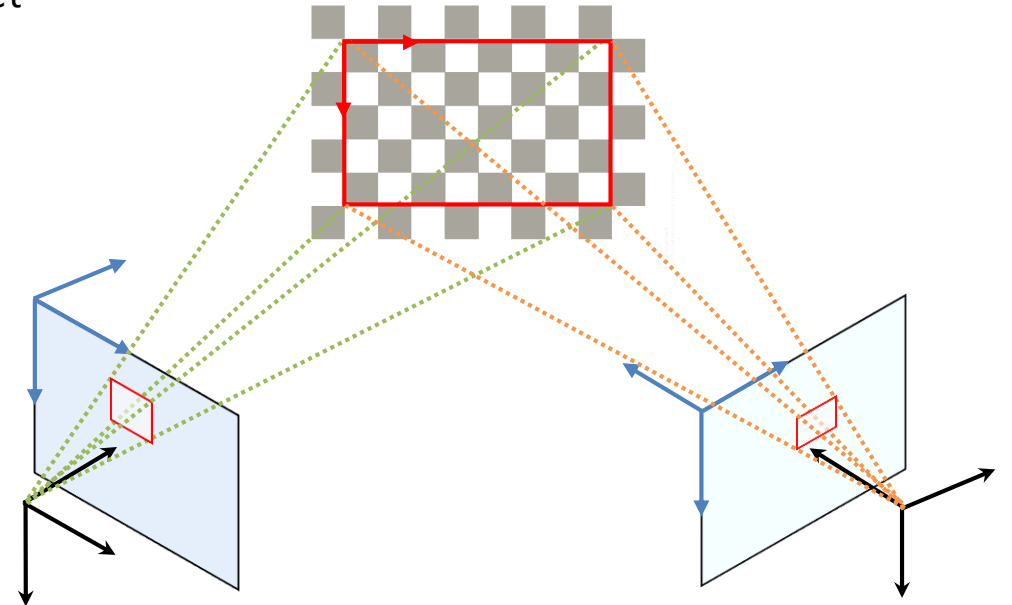
- Unknown: **Intrinsic + extrinsic parameters** ( $5^* + 6$  DOF)
  - Note) The number of intrinsic parameters\* can be varied according to user configuration.
- Given: 3D points  $\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_n$  and their projected points  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$
- Constraints:  $n$  x projection  $\mathbf{x}_i = \mathbf{K} [\mathbf{R} | \mathbf{t}] \mathbf{X}_i$



# Camera Calibration

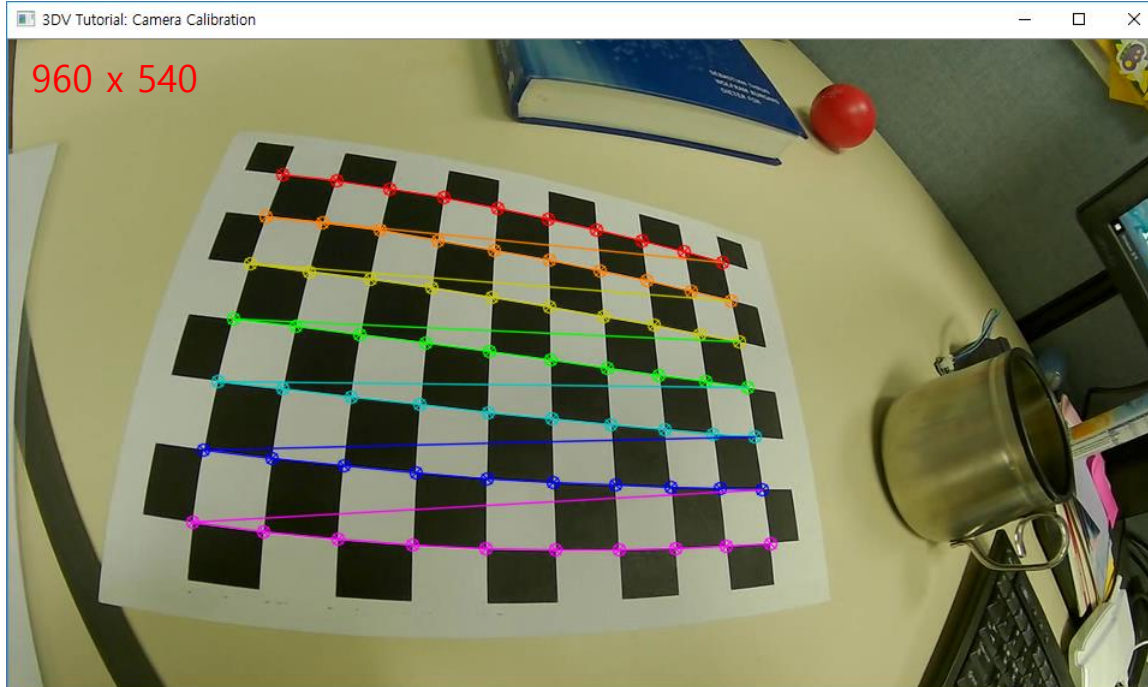
## ▪ Camera calibration

- Unknown: **Intrinsic +  $m$  x extrinsic parameters ( $5^* + m \times 6$  DOF)**
- Given: 3D points  $\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_n$  and their projected points,  $\mathbf{x}_i^j$ , on the  $j$ th image
  - Note)  $m$ : the number of images,  $n$ : the number of 3D points
- Constraints:  $m \times n$  x projection  $\mathbf{x}_i^j = \mathbf{K} \begin{bmatrix} \mathbf{R}_j & \mathbf{t}_j \end{bmatrix} \mathbf{X}_i$
- Solutions [\[Tools\]](#)
  - OpenCV `cv.calibrateCamera()` and `cv.initCameraMatrix2D()`
  - [Camera Calibration Toolbox for MATLAB](#), Jean-Yves Bouguet
  - [DarkCamCalibrator](#), 다크 프로그래머
- Note) How to get calibration boards
  - Print out the pattern and stick it on a hard board
  - [Calibration Checkerboard Collection](#), Mark Jones
  - [Pattern Generator](#), calib.io



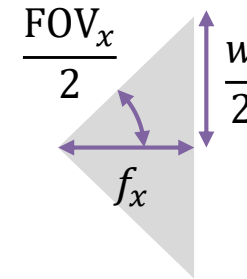
# Camera Calibration

- Example) **Camera calibration** [camera\_calibration.py]



Note) **Field-of-view (FOV)** = focal length (w/o distortion)

- Horizontal:  $2 \times \tan^{-1} \frac{w/2}{f_x} = 96^\circ$
- Vertical:  $2 \times \tan^{-1} \frac{h/2}{f_y} = 64^\circ$



## ## Camera Calibration Results

\* The number of applied images = 22

\* RMS error = 0.473353

\* Camera matrix (K) =

[432.7390364738057, 0, 476.0614994349778] Note) Close to the center of the image, (480, 270)

[0, 431.2395555913084, 288.7602152621297]

[0, 0, 1]

\* Distortion coefficient (k1, k2, p1, p2, k3, ...) =

[-0.2852754904152874, 0.1016466459919075, -0.0004420196146339175, 0.0001149909868437517, -0.01803978785585194]

Note) Close to zero (usually negligible)

# Camera Calibration

- Example) **Camera calibration** [camera\_calibration.py]

```
def select_img_from_video(input_file, board_pattern, select_all=False, wait_msec=10):  
    # Open a video  
    video = cv.VideoCapture(input_file)
```

```
    # Select images  
    img_select = []  
    ...  
    return img_select
```

```
def calib_camera_from_chessboard(images, board_pattern, board_cellsize, K=None, dist_coeff=None, calib_flags=None):  
    # Find 2D corner points from given images
```

$\mathbf{x}_i^j$

```
    img_points = []  
    for img in images:  
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)  
        complete, pts = cv.findChessboardCorners(gray, board_pattern)  
        if complete:  
            img_points.append(pts)
```

```
    assert len(img_points) > 0, 'There is no set of complete chessboard points!'
```

$\mathbf{X}_i$

```
    # Prepare 3D points of the chess board
```

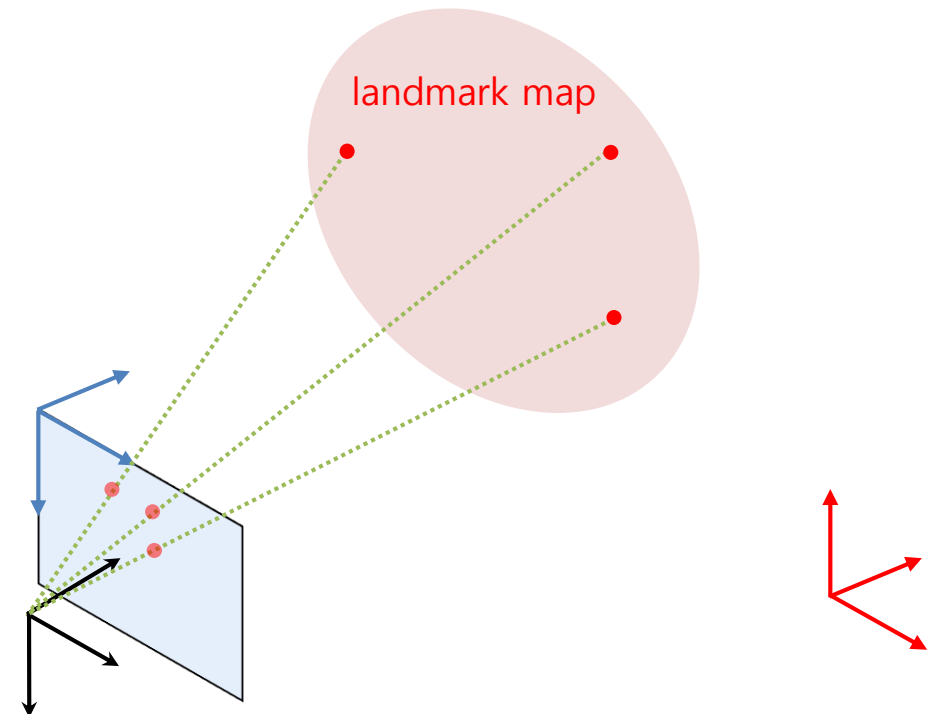
```
    obj_pts = [[c, r, 0] for r in range(board_pattern[1]) for c in range(board_pattern[0])]  
    obj_points = [np.array(obj_pts, dtype=np.float32) * board_cellsize] * len(img_points) # Must be 'np.float32'
```

```
    # Calibrate the camera
```

```
    return cv.calibrateCamera(obj_points, img_points, gray.shape[::-1], K, dist_coeff, flags=calib_flags)
```

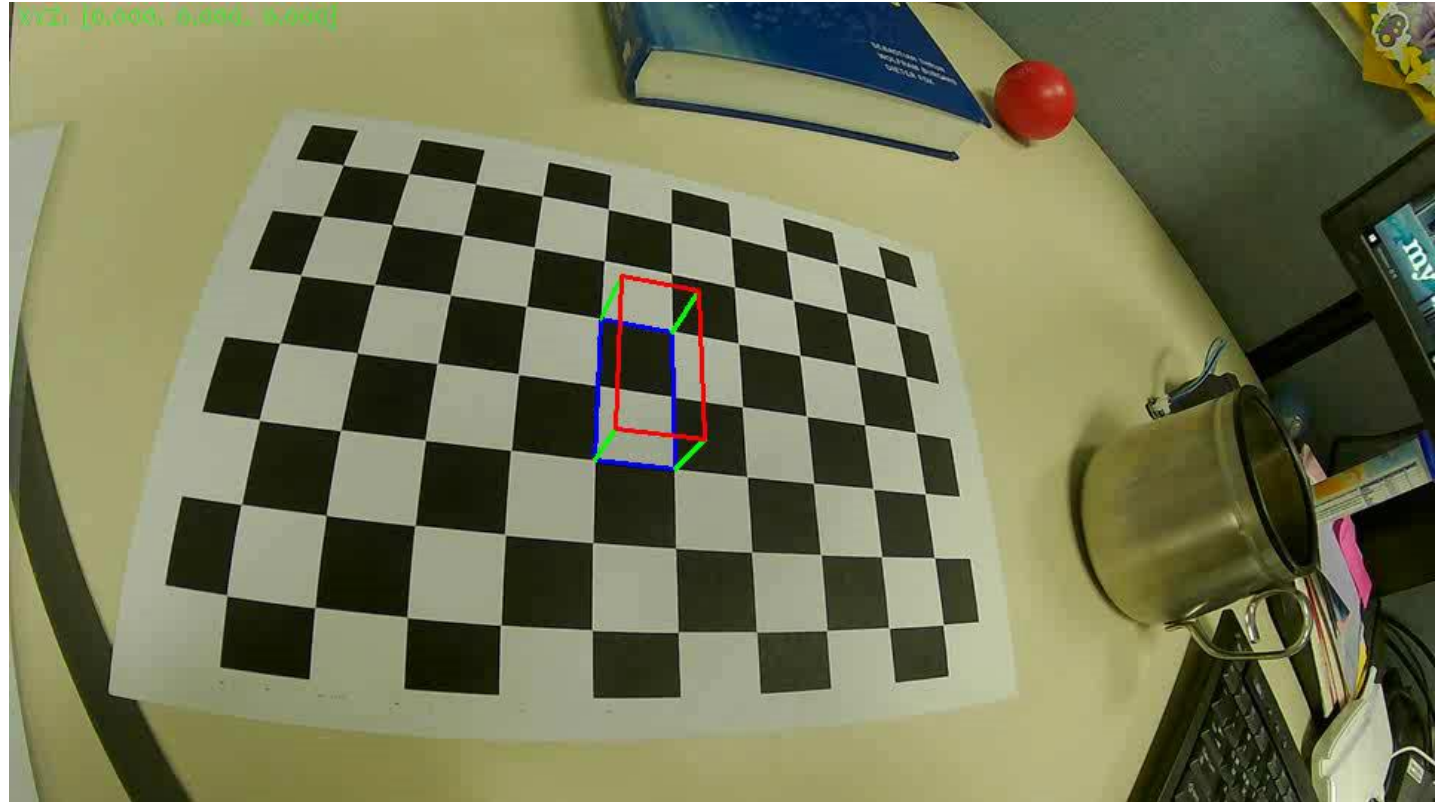
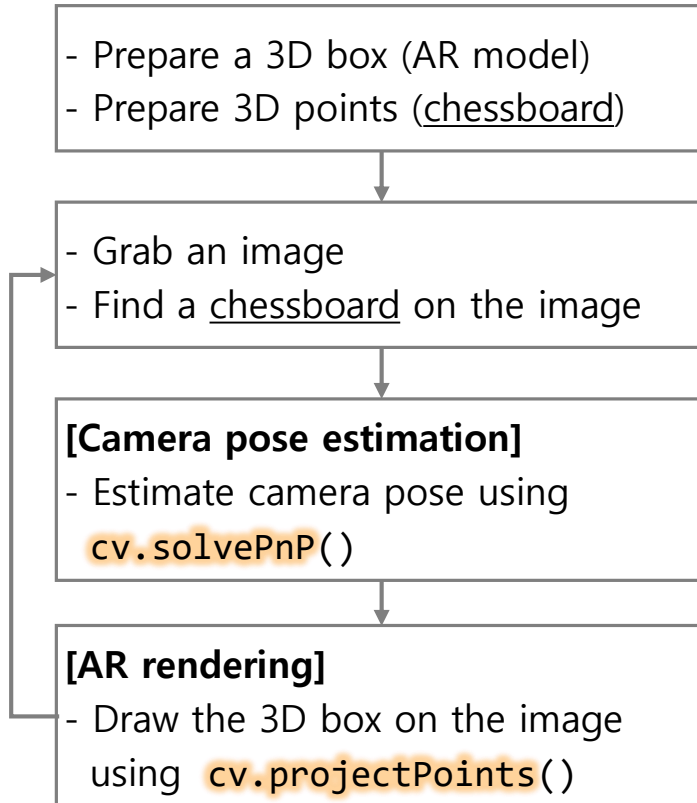
# Absolute Camera Pose Estimation

- **Absolute camera pose estimation** (perspective-n-point; PnP)
  - Unknown: Camera pose  $R$  and  $t$  (6 DOF)
  - Given: 3D points  $\mathbf{X}_1, \mathbf{X}_2, \dots, \mathbf{X}_n$ , their projected points  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$ , and camera matrix  $K$
  - Constraints:  $n$  x projection  $\mathbf{x}_i = K [R | t] \mathbf{X}_i$
  - Solutions ( $n \geq 3$ )  $\rightarrow$  3-point algorithm
    - OpenCV: `cv.solvePnP()` and `cv.solvePnP Ransac()`



# Absolute Camera Pose Estimation

- Example) **Pose estimation (chessboard)** [pose\_estimation\_chessboard.py]



# Absolute Camera Pose Estimation

- Example) **Pose estimation (chessboard)** [pose\_estimation\_chessboard.py]

```
# Open a video
```

```
video = cv.VideoCapture(input_file)
```

```
# Prepare a 3D box for simple AR
```

```
box_lower = board_cellsize * np.array([[4, 2, 0], [5, 2, 0], [5, 4, 0], [4, 4, 0]])
```

```
box_upper = board_cellsize * np.array([[4, 2, -1], [5, 2, -1], [5, 4, -1], [4, 4, -1]])
```

```
# Prepare 3D points on a chessboard
```

```
obj_points = board_cellsize * np.array([[c, r, 0] for r in range(board_pattern[1]) for c in range(board_pattern[0])])
```

```
# Run pose estimation
```

```
while True:
```

```
    # Read an image from the video
```

```
    valid, img = video.read()
```

```
    if not valid:
```

```
        break
```

```
    # Estimate the camera pose
```

```
    complete, img_points = cv.findChessboardCorners(img, board_pattern, board_criteria)
```

```
    if complete:
```

```
        ret, rvec, tvec = cv.solvePnP(obj_points, img_points, K, dist_coeff)
```

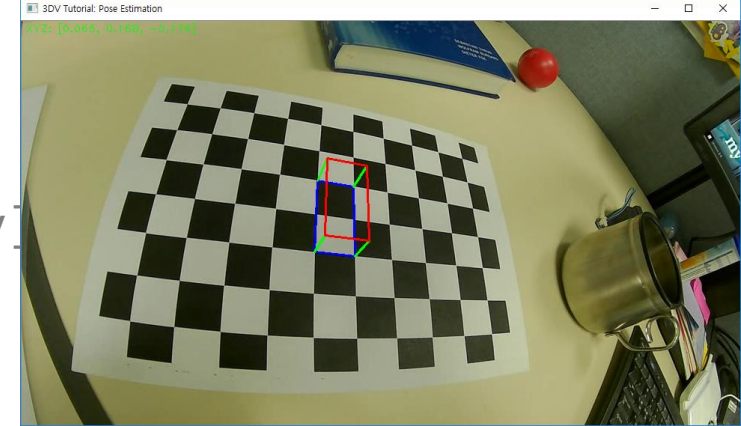
```
    # Draw the box on the image
```

```
    line_lower, _ = cv.projectPoints(box_lower, rvec, tvec, K, dist_coeff)
```

```
    line_upper, _ = cv.projectPoints(box_upper, rvec, tvec, K, dist_coeff)
```

```
    cv.polylines(img, [np.int32(line_lower)], True, (255, 0, 0), 2)
```

```
    cv.polylines(img, [np.int32(line_upper)], True, (0, 0, 255), 2)
```





# Absolute Camera Pose Estimation

- Example) **Pose estimation (chessboard)** [pose\_estimation\_chessboard.py]

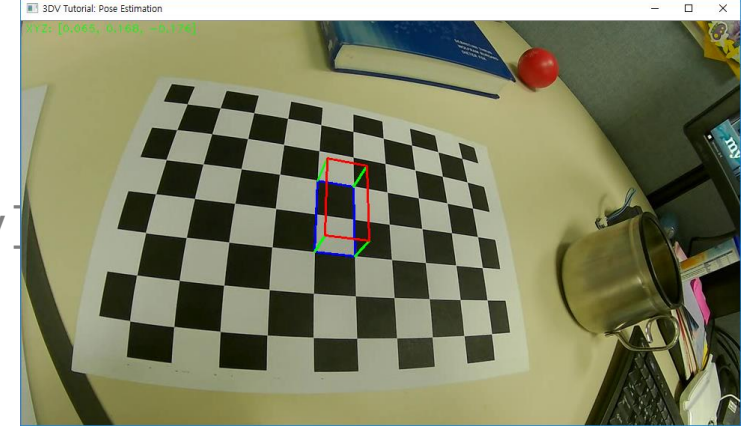
```
# Open a video
# Prepare a 3D box for simple AR
# Prepare 3D points on a chessboard

# Run pose estimation
while True:
    # Read an image from the video

    # Estimate the camera pose
    complete, img_points = cv.findChessboardCorners(img, board_pattern, board_criteria)
    if complete:
        ret, rvec, tvec = cv.solvePnP(obj_points, img_points, K, dist_coeff)

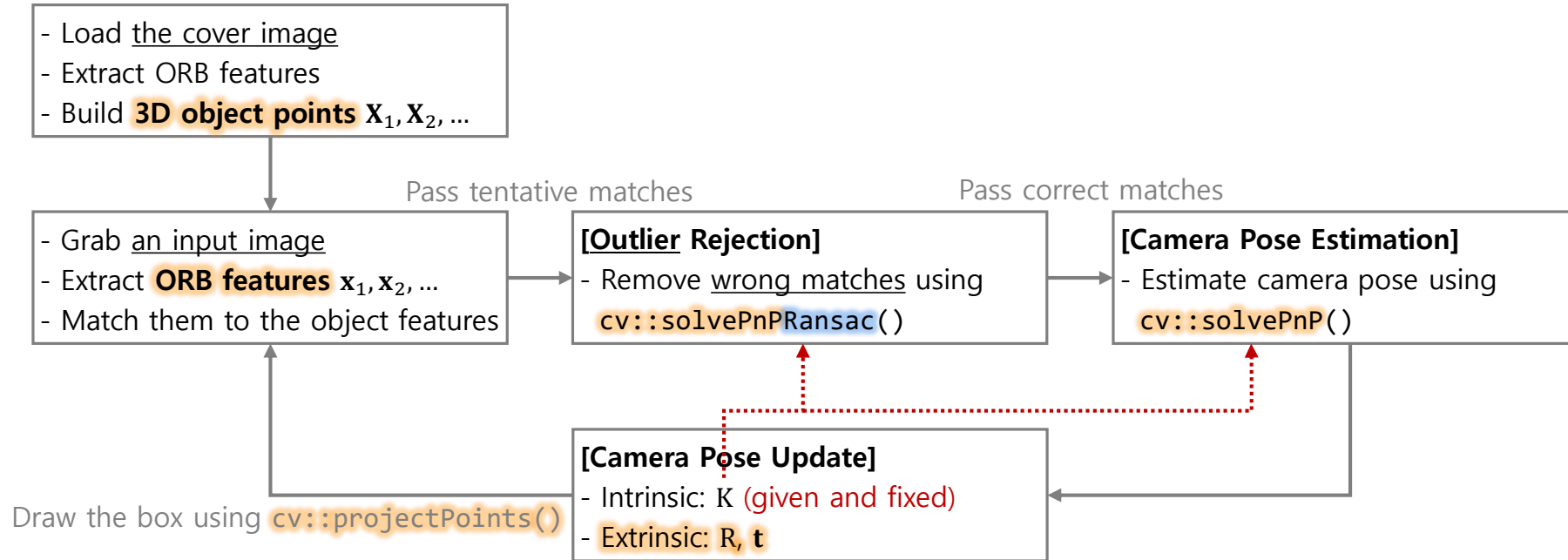
        # Draw the box on the image
        line_lower, _ = cv.projectPoints(box_lower, rvec, tvec, K, dist_coeff)
        line_upper, _ = cv.projectPoints(box_upper, rvec, tvec, K, dist_coeff)
        cv.polylines(img, [np.int32(line_lower)], True, (255, 0, 0), 2)
        cv.polylines(img, [np.int32(line_upper)], True, (0, 0, 255), 2)
        for b, t in zip(line_lower, line_upper):
            cv.line(img, np.int32(b.flatten()), np.int32(t.flatten()), (0, 255, 0), 2)

        # Print the camera position
        R, _ = cv.Rodrigues(rvec) # Alternative) scipy.spatial.transform.Rotation
        p = (-R.T @ tvec).flatten()
        info = f'XYZ: [{p[0]:.3f} {p[1]:.3f} {p[2]:.3f}]'
        cv.putText(img, info, (10, 25), cv.FONT_HERSHEY_DUPLEX, 0.6, (0, 255, 0))
```



# Absolute Camera Pose Estimation

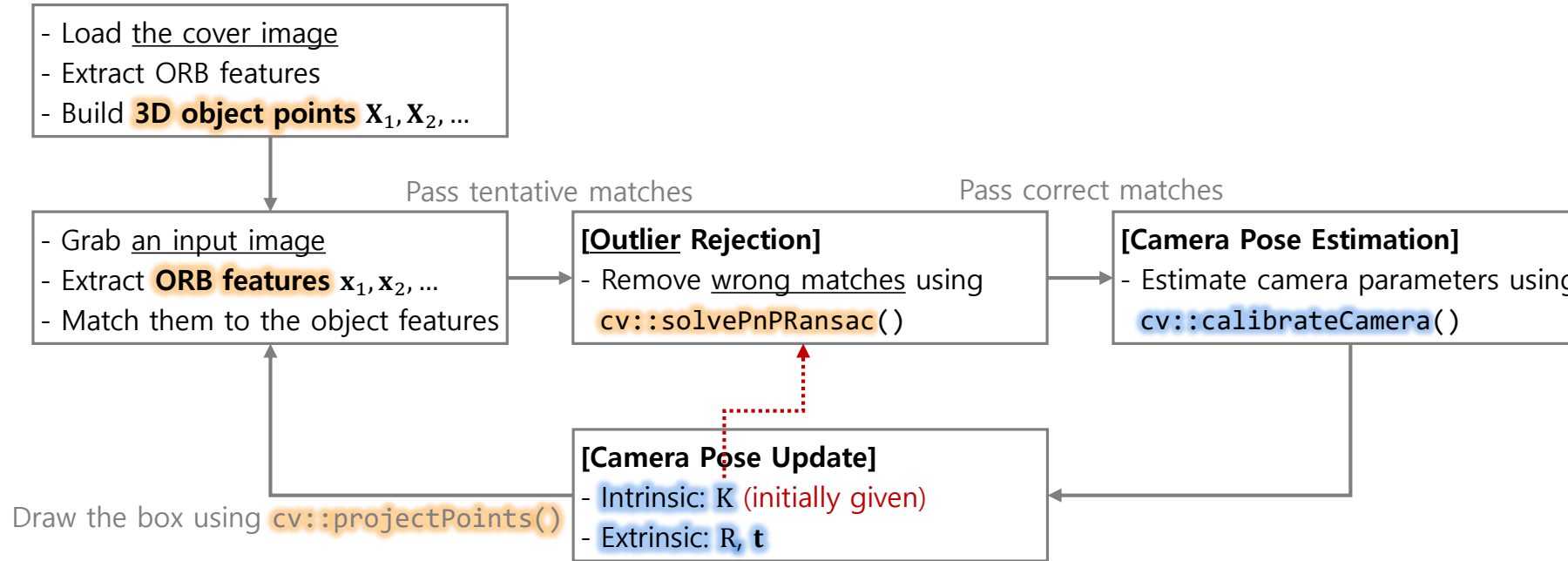
- Example) **Pose estimation (book)** [pose\_estimation\_book1.py]



(due to unknown and changing K; autofocus)

# Absolute Camera Pose Estimation

- Example) **Pose estimation (book) + camera calibration** [pose\_estimation\_book2.py]



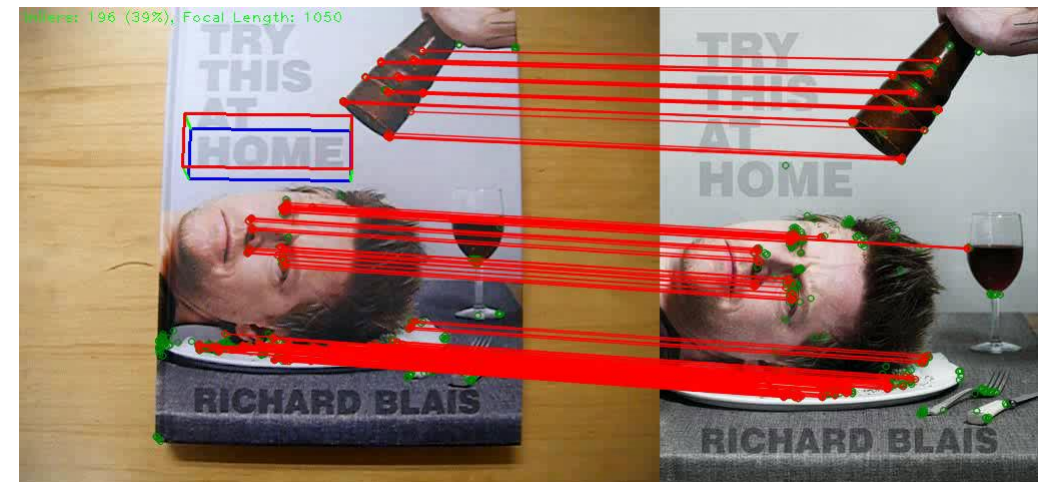
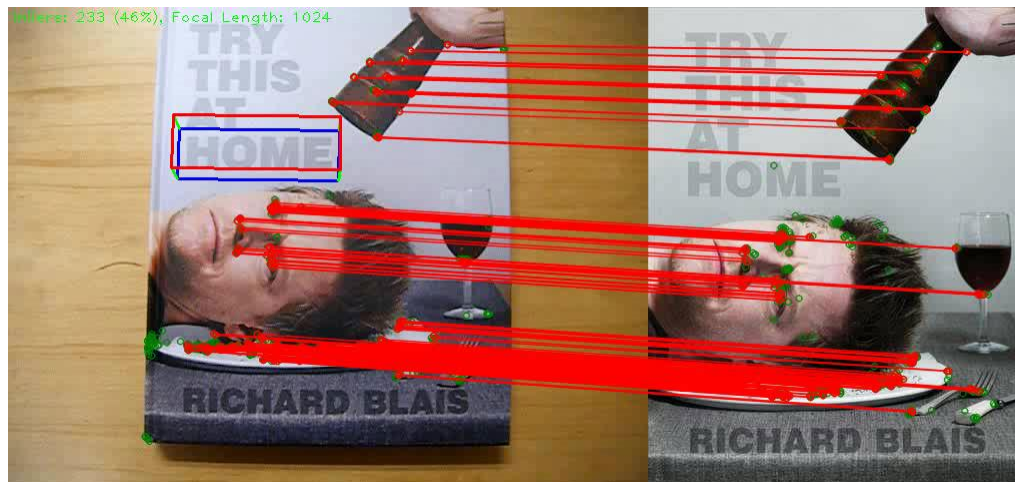
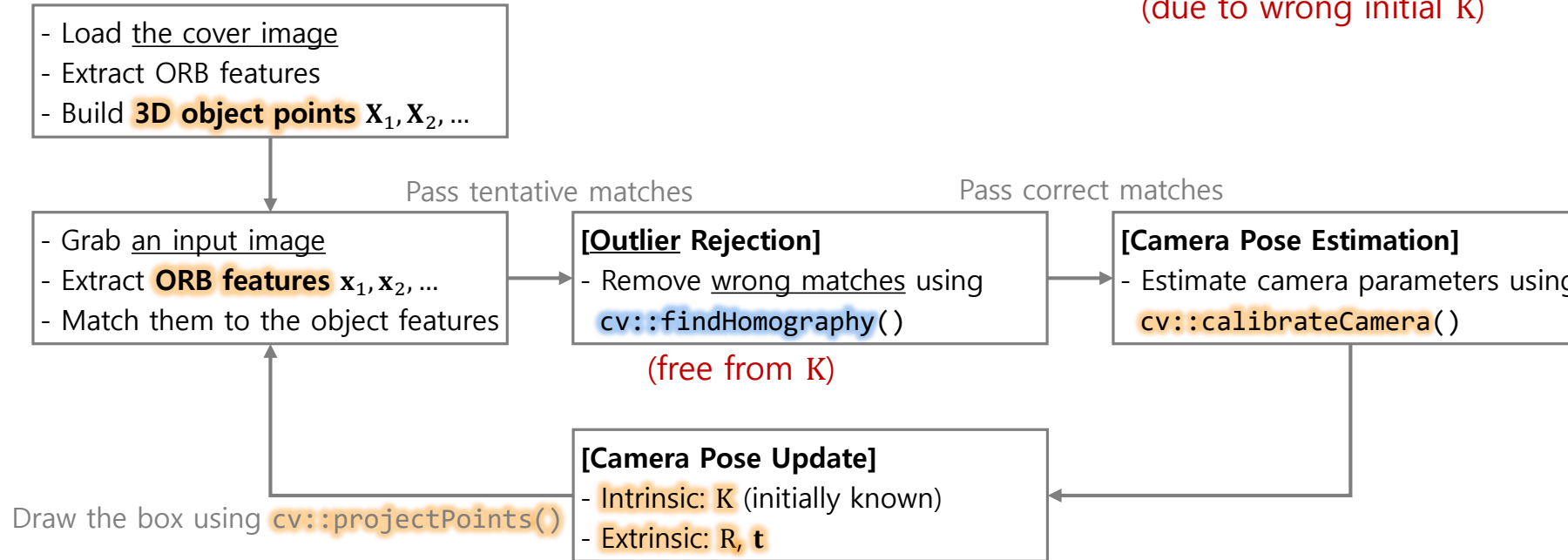
(due to wrong initial K)



# Absolute Camera Pose Estimation

- Example) **Pose estimation (book) + camera calibration** – initially given  $K$  [pose\_estimation\_book3.py]

(due to wrong initial  $K$ )



# Summary

- **Camera Projection Models:**  $\mathbf{x} = \text{proj}(\mathbf{X}; K, R, \mathbf{t}, d)$ 
  - Pinhole camera model:  $\mathbf{x} = K(R\mathbf{X} + \mathbf{t})$ 
    - Note) Homogeneous coordinate
    - Example) Object localization / image formation
  - Geometric distortion models:  $\hat{\mathbf{x}}_d = f_d(\hat{\mathbf{x}})$  on the normalized image plane ( $\hat{\mathbf{x}}$ )
    - e.g. Polynomial distortion model: Radial distortion and tangential distortion
    - Example) Distortion visualization / distortion correction
- **Camera Calibration**
  - To estimate camera intrinsic parameters ( $K$ , distortion coefficients) and extrinsic parameters ( $R$  and  $\mathbf{t}$ )
  - Example) Camera calibration
- **Absolute Camera Pose Estimation (PnP)**
  - To estimate camera extrinsic parameters ( $R$  and  $\mathbf{t}$ )  $\rightarrow$  camera pose ( $R^T$  and  $-R^T\mathbf{t}$ )
  - Example) Pose estimation with a chessboard / ... with a book
    - Q) What is RANSAC? What is homography?