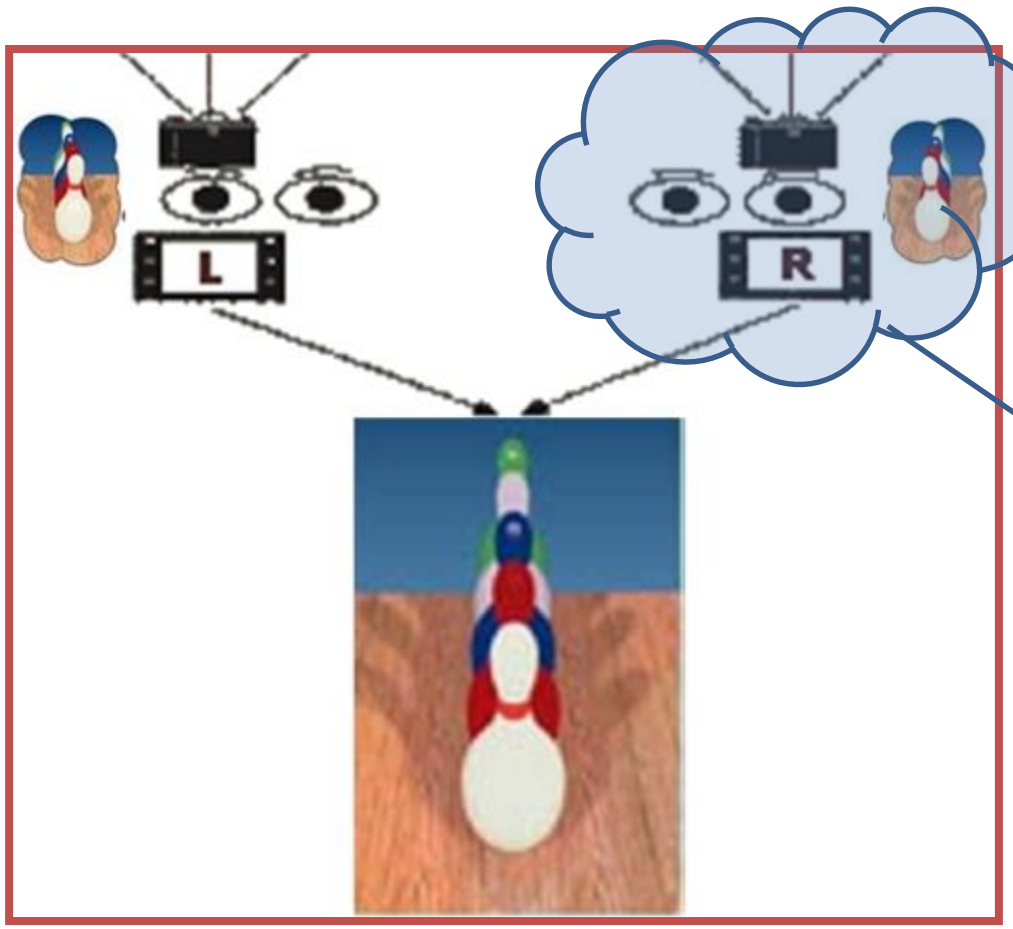
Camera Calibration
(Single-View Geometry)

Gang Pan

gpan@zju.edu.cn

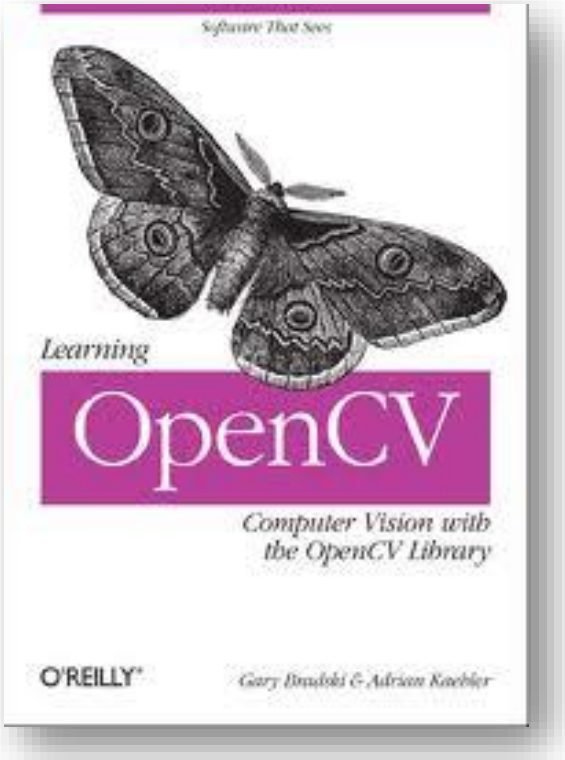
Outline



- **Single-view geometry**
- **Camera projection**

- Camera model
- Camera calibration

Chapter 11 Camera Models and Calibration
Chapter 12 Projection and 3D Vision



Projection

Projection

Julian Beever **http://users.skynet.be/J.Beever/pave.htm**

In this class

- Camera model

  Learning OpenCV: pp.371-377

  – Modeling projection

  – Intrinsic parameters

  – Distortions

  Learning OpenCV: pp.379-381

  – View transformation

  – Extrinsic parameters

- Camera calibration

Modeling Projection

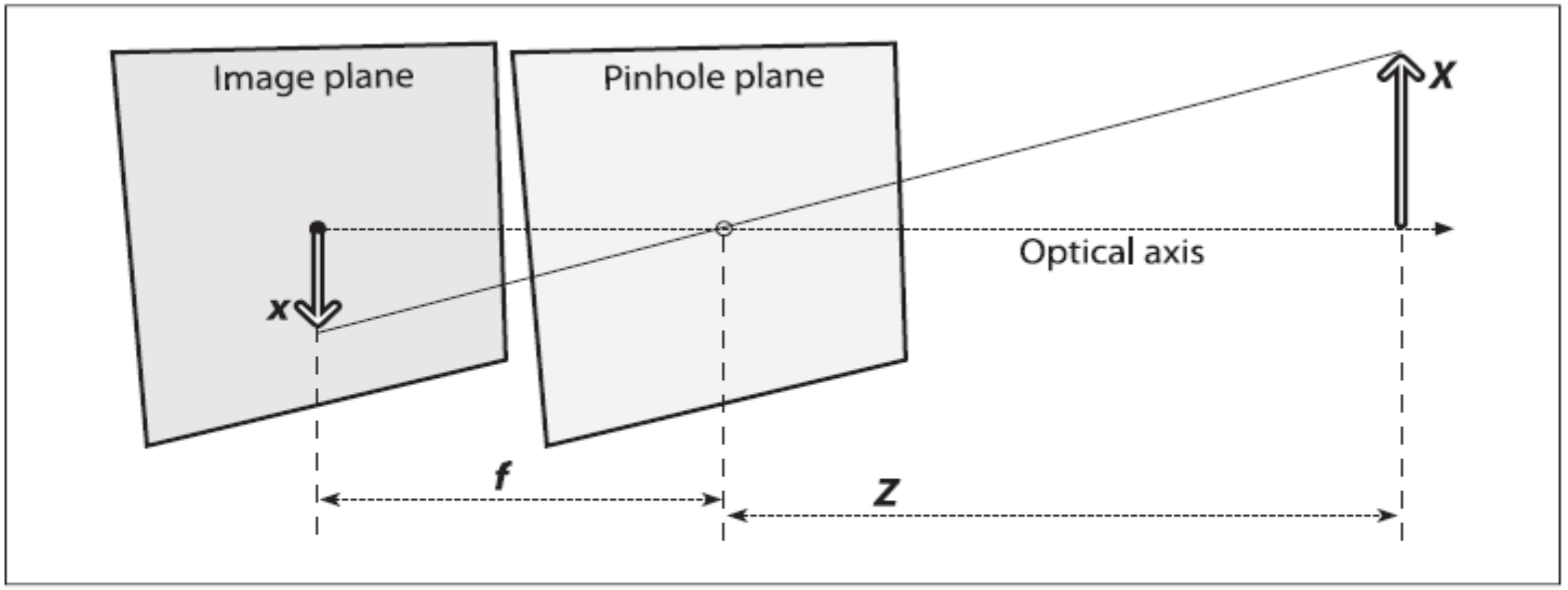


Figure 11-1. Pinhole camera model: a pinhole (the pinhole aperture) lets through only those light rays that intersect a particular point in space; these rays then form an image by "projecting" onto an image plane

Pinhole camera model

$$\frac{-x}{f} = \frac{X}{Z} \longrightarrow -x = f\frac{X}{Z}$$

Modeling Projection

- All the parameters inside

$$\begin{cases} x_{screen} = f_x\left(\frac{Y}{Z}\right) + c_x, \\ y_{screen} = f_y\left(\frac{X}{Z}\right) + c_y \end{cases}$$

$$f_x = Fs_x,\ f_y = Fs_y$$

The units

- ***F***: mm
- s_x, s_y: pixel/mm
- f_x, f_y: pixel

Homogeneous coordinates

Silven [Heikkila97]). The projection of the points in the physical world into the camera is now summarized by the following simple form:

$$q = MQ, \quad \text{where} \quad q = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, \quad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}, \quad Q = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

- *M* contains the **intrinsic parameters**

Matrix operations!

- **Intrinsic parameters**

$$\left(f_x,\ f_y,\ c_x,\ c_y\right)$$

Summery

- **Pinhole model** is the simplest one, but very effective.
- We use 4 parameters to describe the projection and imaging.
- Thanks to homogenous coordinates, all the operations are integrated into a single MATRIX!

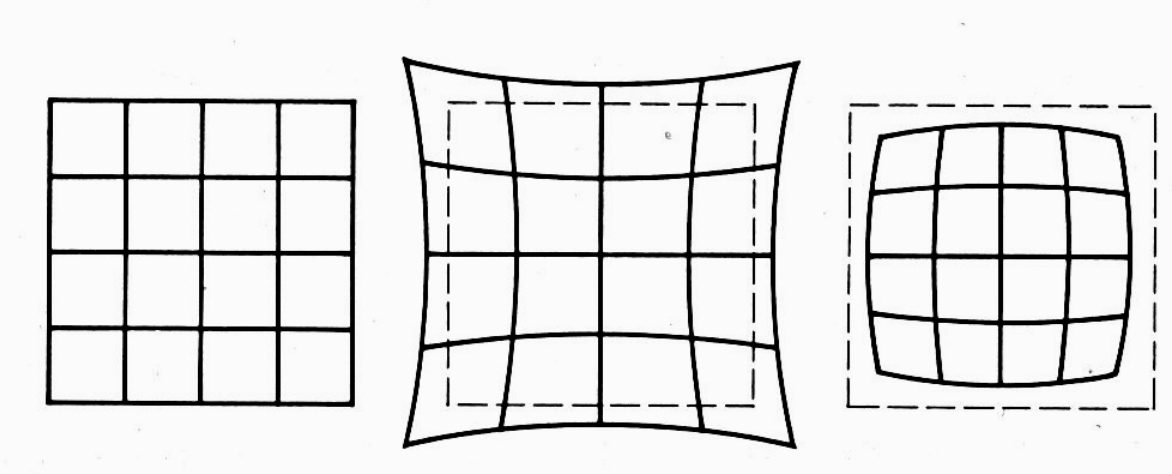
Lens Distortion

- Remind that why we introduce lens
  - Collect more light
  - No free lunch: lens troubles
- Two types of distortion
  - **Radial distortion**
  - **Tangential distortion**

Troubles with Lens

- Beyond the simple geometry of pinhole camera
- Introducing distortions from the lens itself

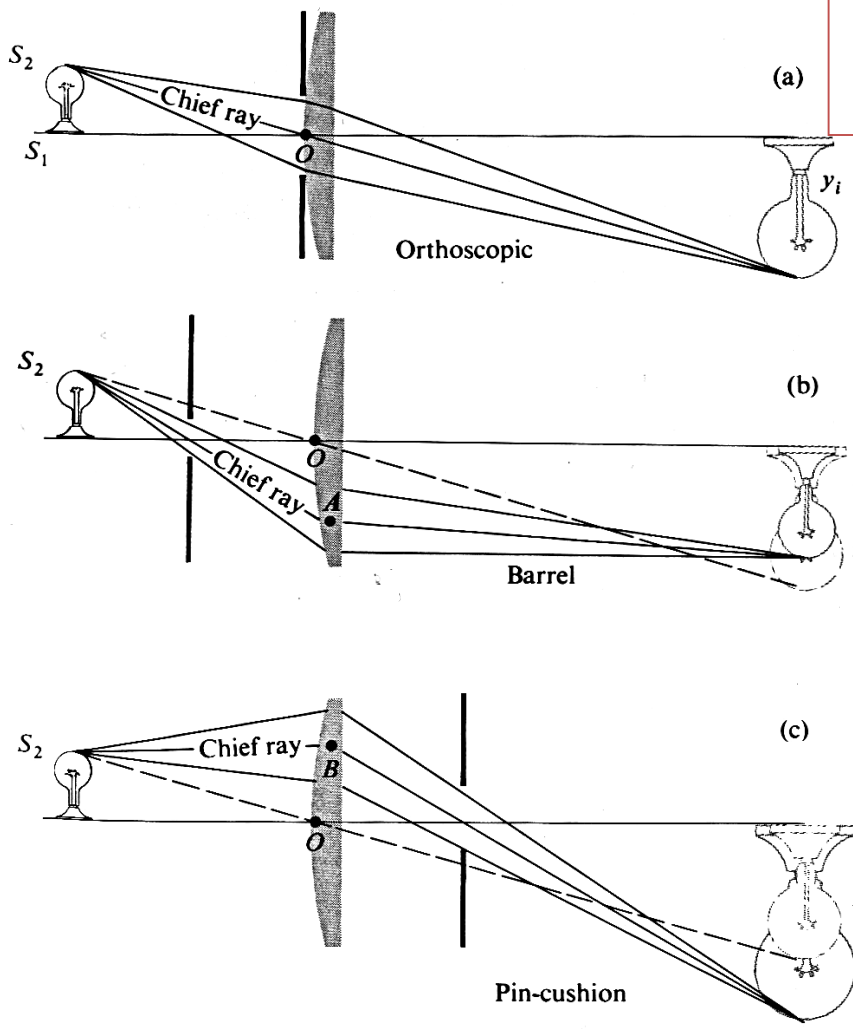
Radial Distortion



- Radial distortion of the image
  - Caused by imperfect lenses (shape of lens)
  - Deviations are most noticeable for rays that pass through the edge of the lens

Mystery Behind Radial Distortion

- **Radial distortion caused by**
  - **The geometry of the lens**
  - **Aperture position**



- Orthoscopic
- Barrel
- Pin-cushion

Modeling Radial Distortion

$$x_{\text{corrected}} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

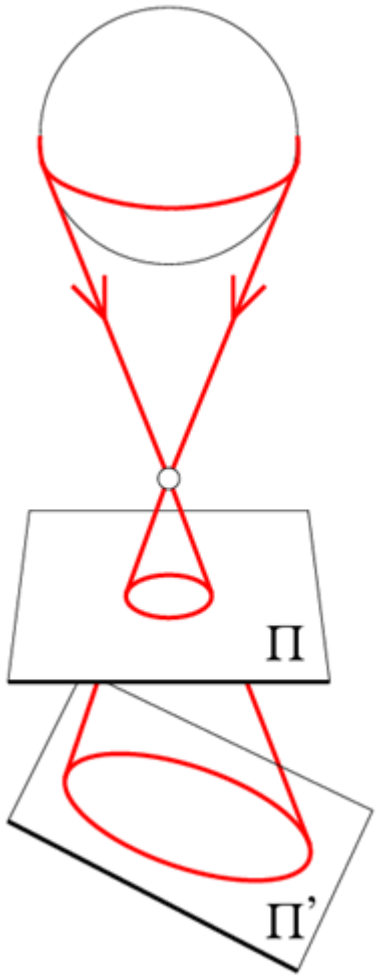
$$y_{\text{corrected}} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

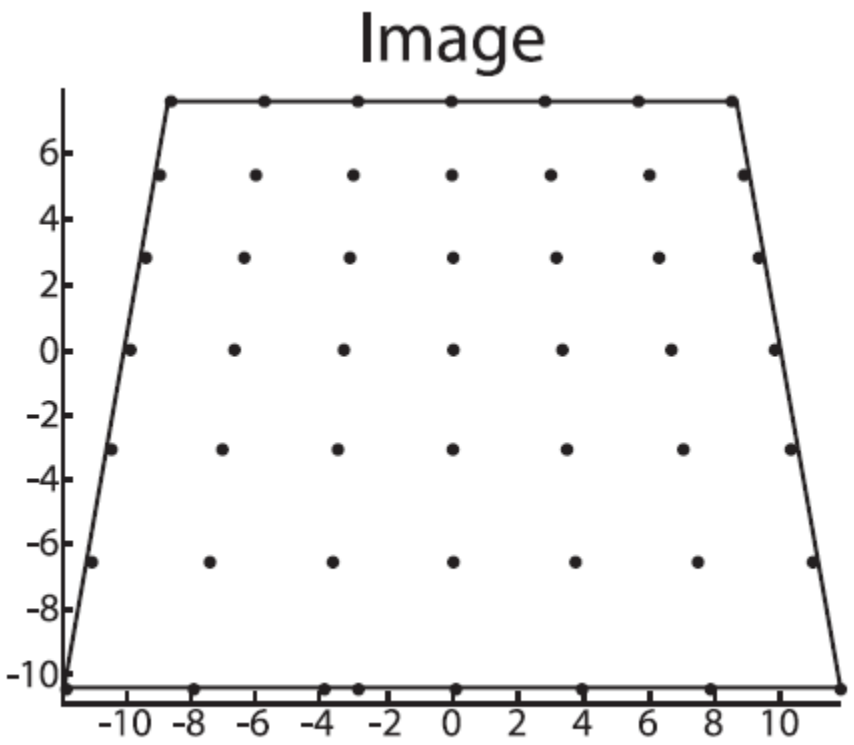
- To model lens distortion
  - Use above projection operation instead of standard projection matrix multiplication

Correcting Radial Distortion

from [Helmut Dersch]

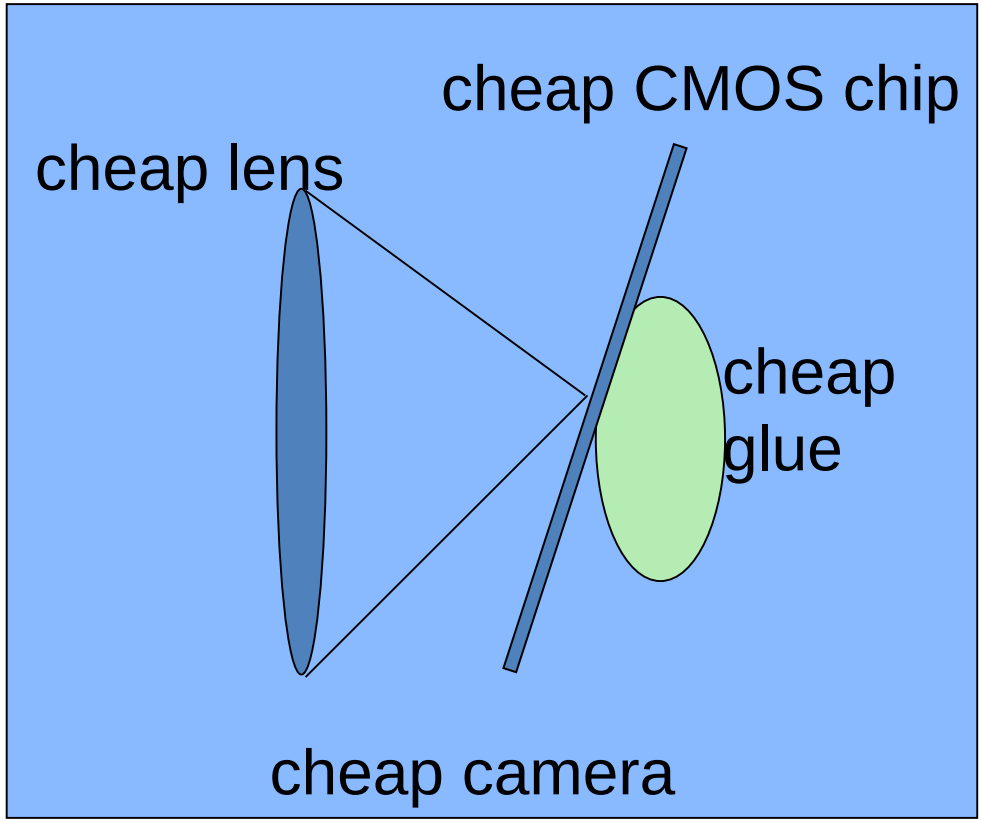
Tangential Distortion



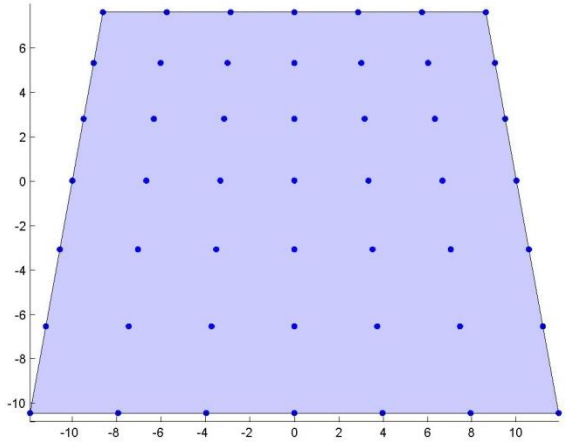


Mystery Behind Tangential Distortion

- **Tangential distortion caused by**
  - **The decentering of the optical component (assembly process)**



image



Modeling Tangential Distortion

$$x_{\text{corrected}} = x + [2p_1 y + p_2(r^2 + 2x^2)]$$

$$y_{\text{corrected}} = y + [p_1(r^2 + 2y^2) + 2p_2 x]$$

- To model lens distortion
  - Use above projection operation instead of standard projection matrix multiplication

Summarize Distortion

- Called **distortion parameters**

$$(k_1,\ k_2,\ p_1,\ p_2,\ k_3)$$

- Called **intrinsic parameters**

$$(f_x,\ f_y,\ c_x,\ c_y)$$

OpenCV Func

- cvUndistort2()
  - cvInitUndistortMap()
    - computes the distortion map
    - Input: Intrinsic matrix & distoration coefficients are from cvCalibrateCamera2()
  - cvRemap()
    - apply this map to an arbitrary image
    - cvUndistortPoints()
- <Learning OpenCV>: pp.396-397

```
// Undistort images
void cvInitUndistortMap(
  const CvMat*   intrinsic_matrix,
  const CvMat*   distortion_coeffs,
  cvArr*         mapx,
  cvArr*         mapy
);
void cvUndistort2(
  const CvArr*   src,
  CvArr*         dst,
  const cvMat*   intrinsic_matrix,
  const cvMat*   distortion_coeffs
);
// Undistort a list of 2D points only
void cvUndistortPoints(
  const CvMat* _src,
  CvMat*         dst,
  const CvMat*  intrinsic_matrix,
  const CvMat*  distortion_coeffs,
  const CvMat*  R  = 0,
  const CvMat*  Mr = 0;
);
```

In this class

- Camera model

  Learning OpenCV: pp.371-377

  – Modeling projection

  – Intrinsic parameters

  – Distortions

  Learning OpenCV: pp.379-381

  – View transformation

  – Extrinsic parameters

- Camera calibration

View Transformation

- Projecting a point *Q(x, y, z)*

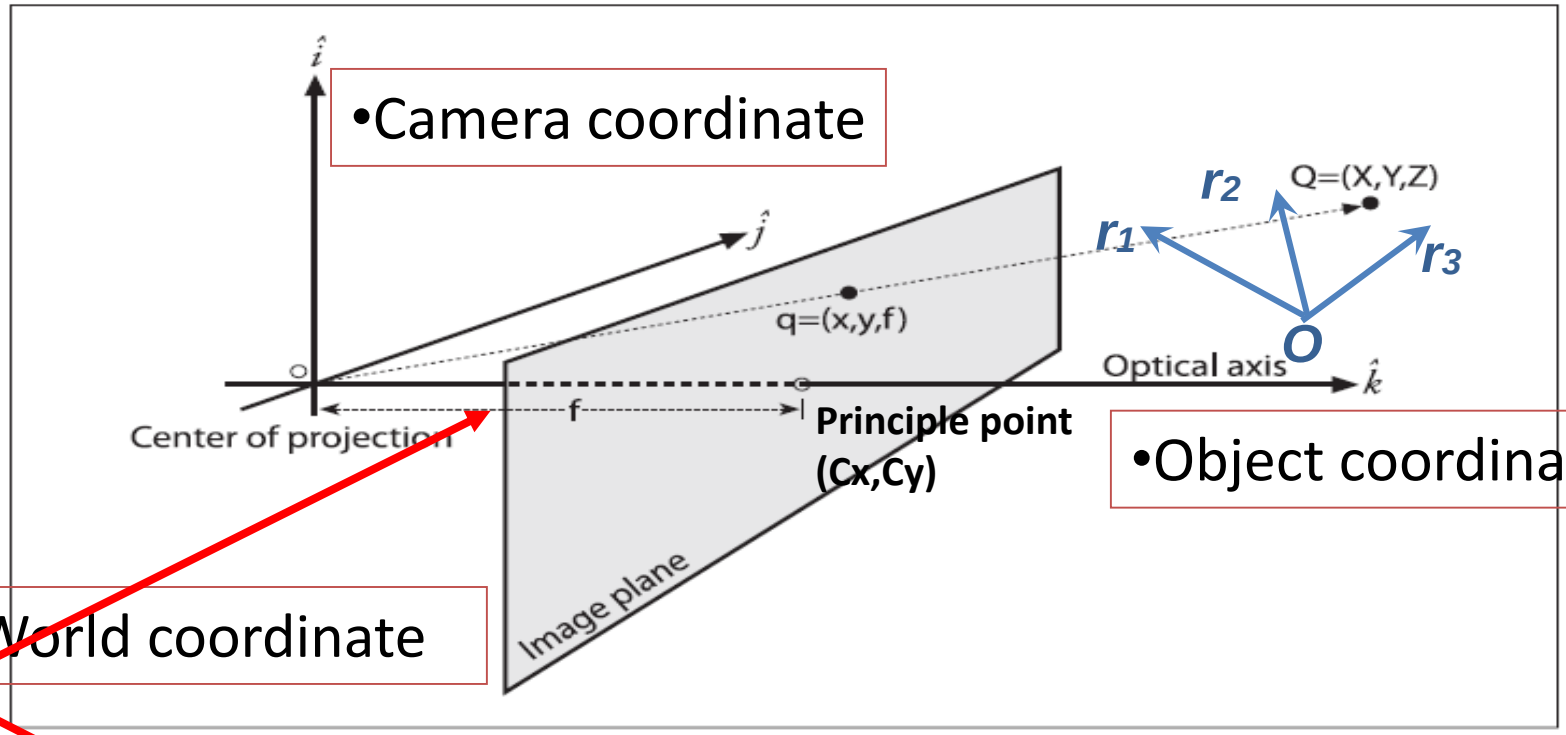


Figure 11-2. A point Q = (X, Y, Z) is projected onto the image plane by the ray passing through the center of projection, and the resulting point on the image is q = (z, y, f); the image plane is really just the projection screen "pushed" in front of the pinhole (the math is equivalent but simpler this way)

Rotation and Translation

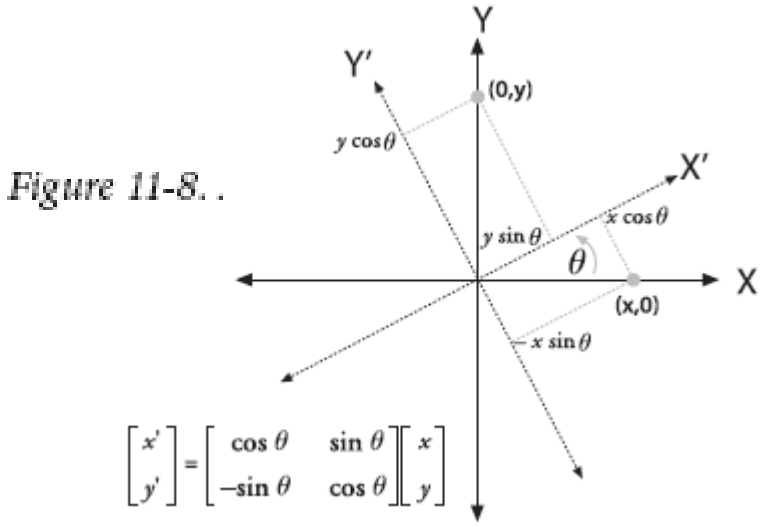


Figure 11-8. .

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & \sin\psi \\ 0 & -\sin\psi & \cos\psi \end{bmatrix} \quad R_y(\varphi) = \begin{bmatrix} \cos\varphi & 0 & -\sin\varphi \\ 0 & 1 & 0 \\ \sin\varphi & 0 & \cos\varphi \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$P' = \begin{bmatrix} \mathbf{R}_{3\times3} & \mathbf{t}_{3\times1} \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix} P$$

$$R = R_z(\theta),\ R_y(\varphi),\ R_x(\psi)$$

$$\mathbf{t}_{3\times1} = (t_x, t_y, t_z)'$$

View Transformation

- Transformation between camera and object
- Called **extrinsic parameters**

$$(\theta, \varphi, \psi, t_x, t_y, t_z)$$

- Called **intrinsic parameters**

$$(f_x,\ f_y,\ c_x,\ c_y)$$

- Called **distortion parameters**

$$(k_1,\ k_2,\ p_1,\ p_2,\ k_3)$$

Extrinsic Parameters

$$(\theta, \varphi, \psi, t_x, t_y, t_z)$$

$$Q_{cam} = \begin{bmatrix} \mathbf{R}_{3\times3} & \mathbf{t}_{3\times1} \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix} Q_{obj}$$

$$M_{ext} = \begin{bmatrix} \mathbf{R}_{3\times3} & \mathbf{t}_{3\times1} \\ 0 \quad 0 \quad 0 & 1 \end{bmatrix}$$

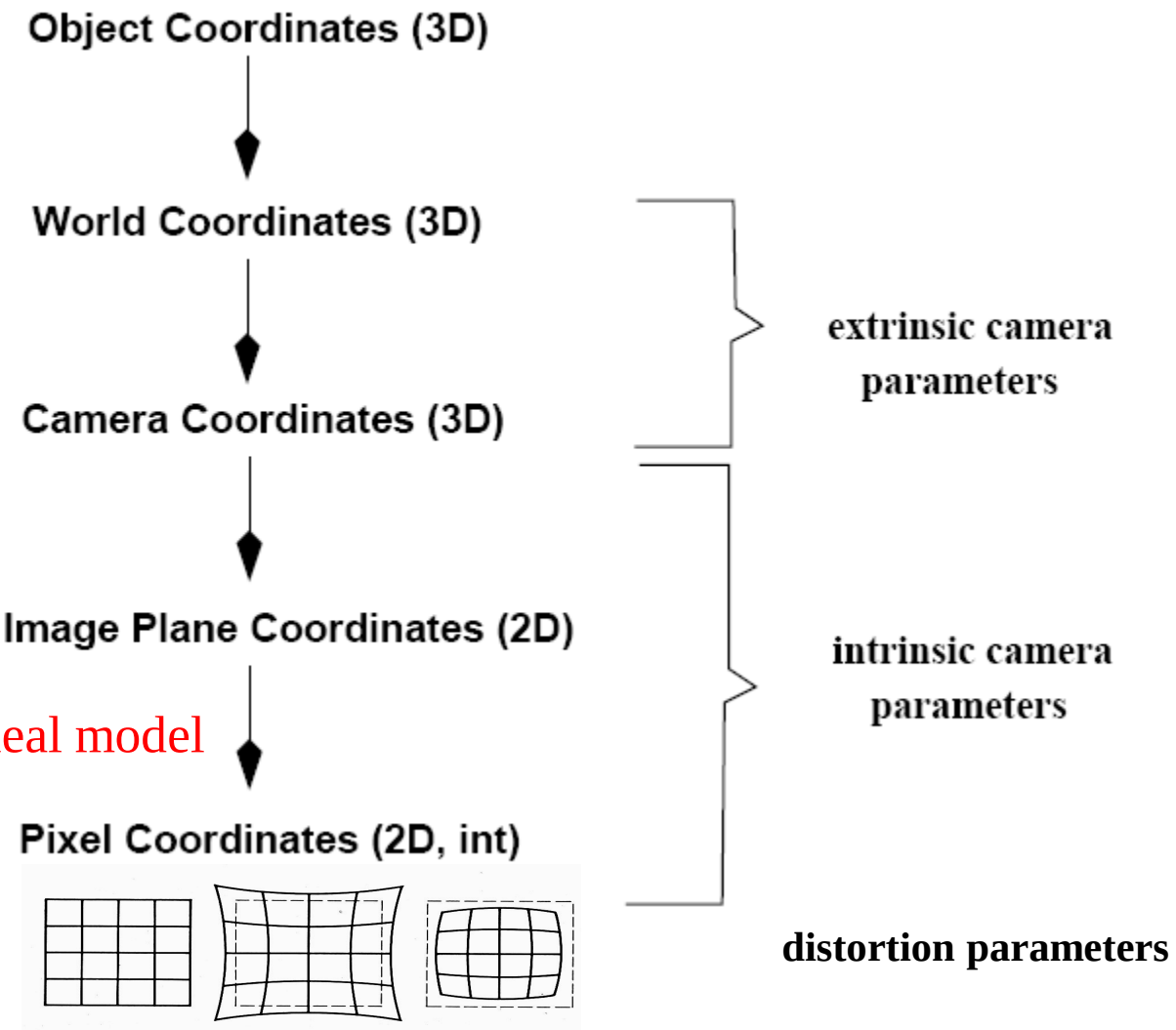
Transformation: camera <-> object

$$Q_{cam} = M_{ext} Q_{obj}$$

Projection and A\D imaging (Pinhole camera)

$$q_{image} = M_{int} Q_{cam}$$

Camera parameters

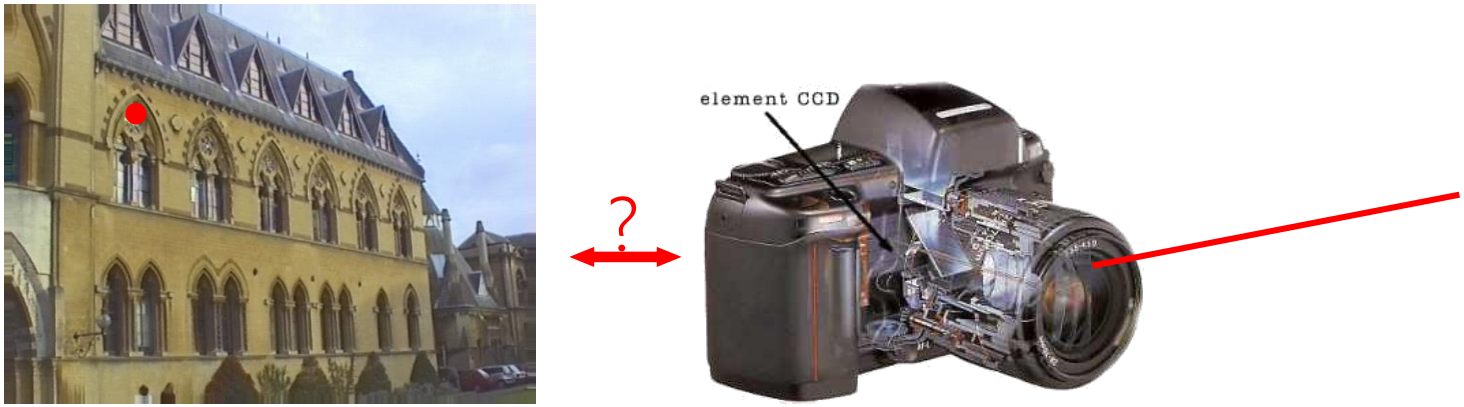


What to Study?

- Camera model
- Camera calibration
  - What?
  - Why?
  - How?(in openCV)

What is Camera Calibration?

Compute relation between pixels and rays in space.



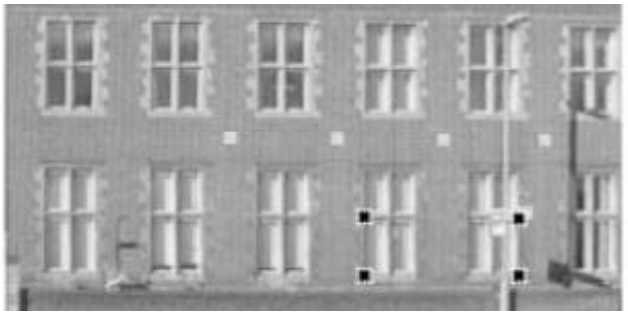
Why to Calibrate a Camera?

- In order to work in 3D, we need to know the parameters of the particular camera setup.
- Good calibration is important when we need to
  - Reconstruct a world model
    - e.g., VirtualEarth project
  - Interact with the world
    - Robot, hand-eye coordination
    - Wii like?!

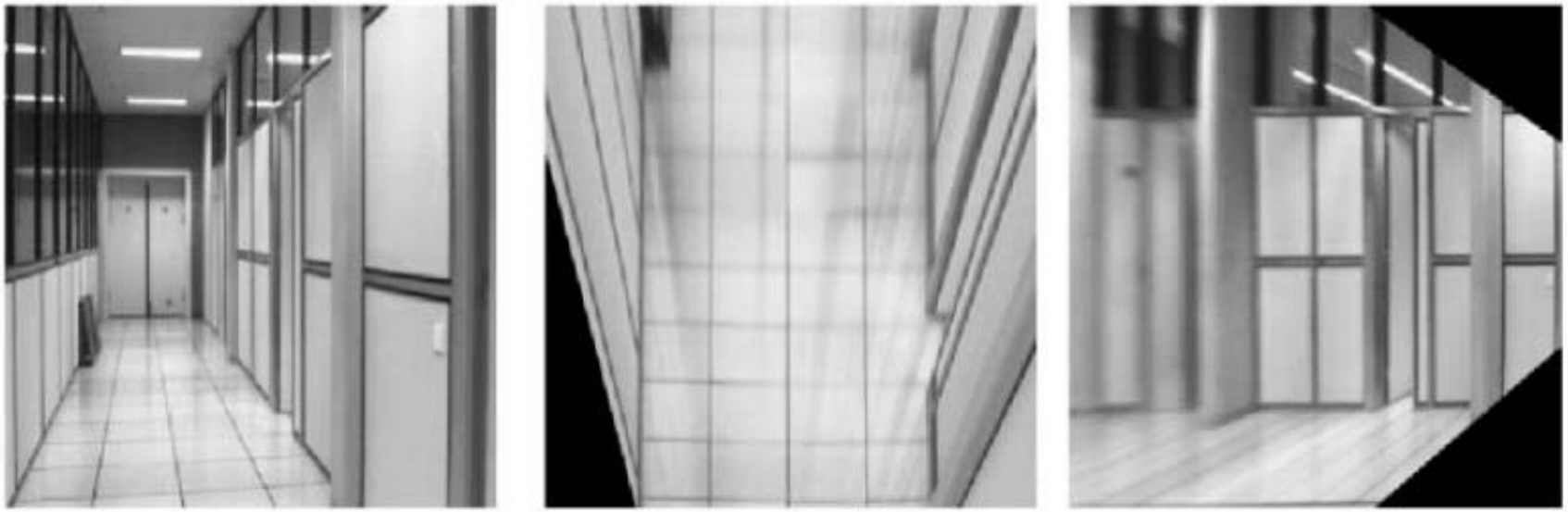
Example 1: Removing Perspective Distortion

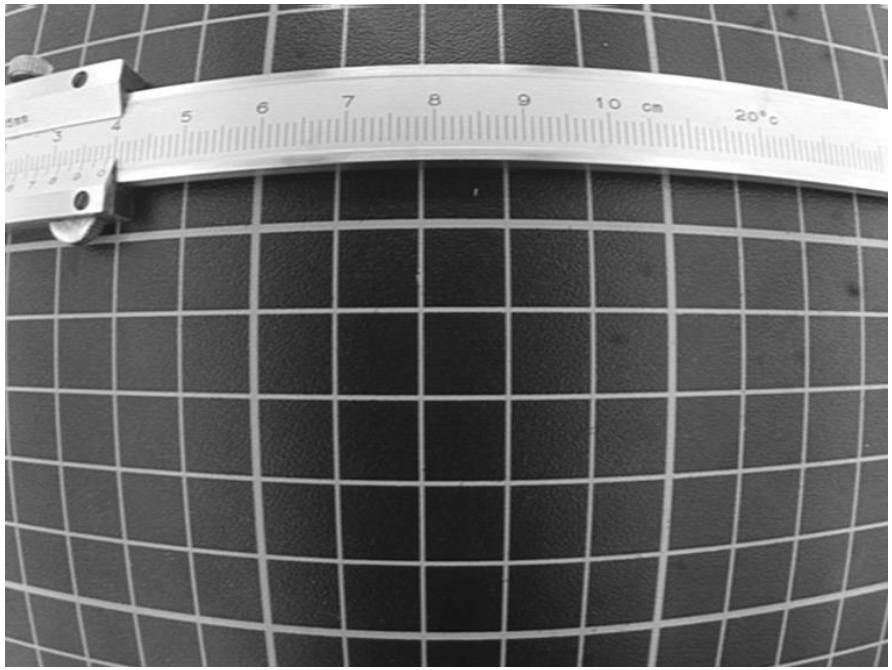
Given: the coordinates of four points on the scene plane

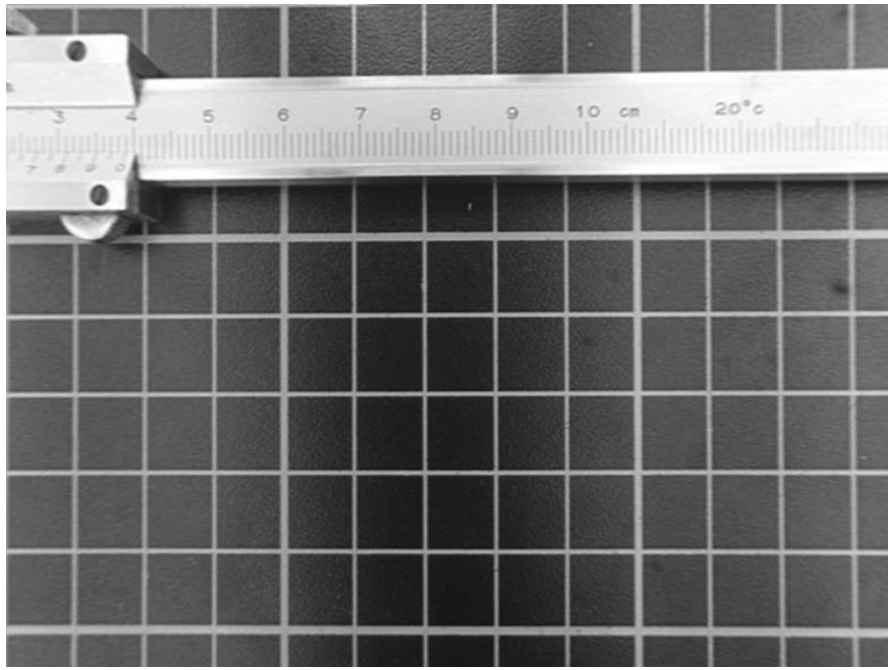
Find: a projective rectification of the p



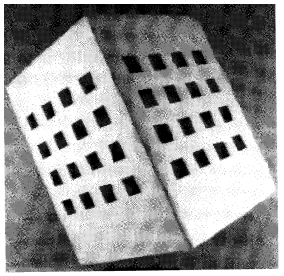
Example 2: Synthetic Rotations



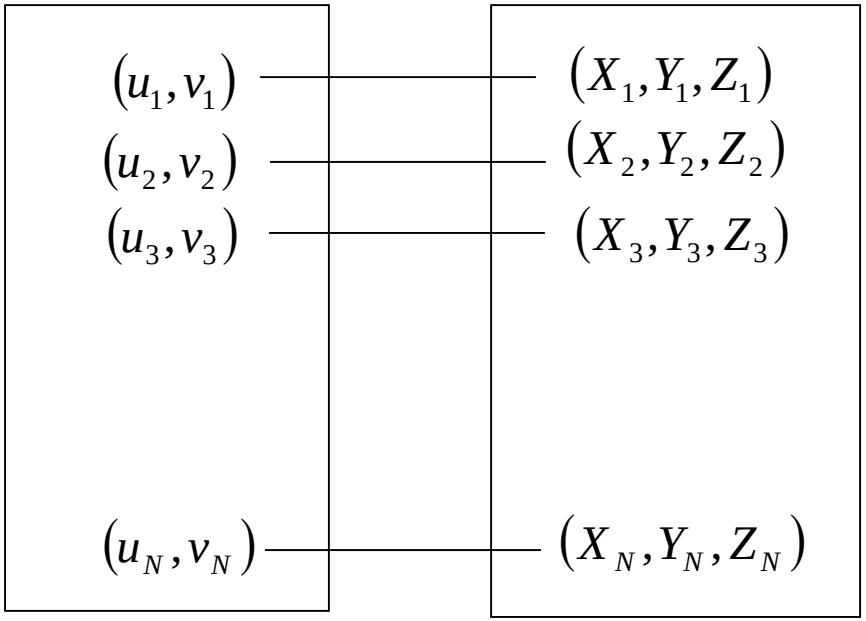

5 6 7 8 9 10 cm 20°c

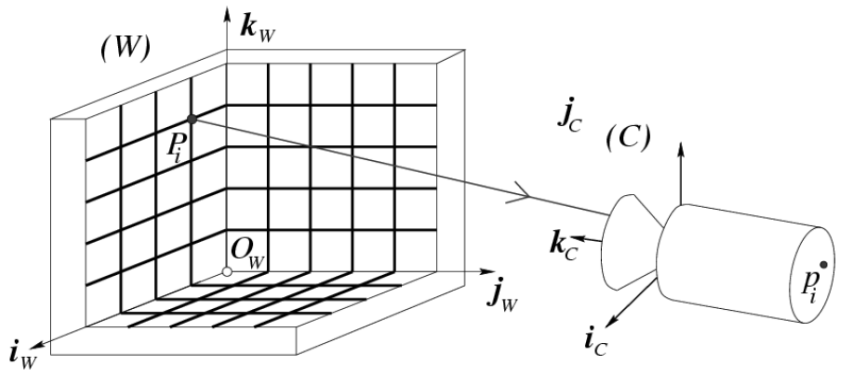

3 4 5 6 7 8 9 10 cm 20°c

Calibration Problem



- **Given**
  - ***N*** correspondences b/w scene and images
- **Recover the camera parameters**
  - Distortion coeffients, intrinsic para., extrinsic para.





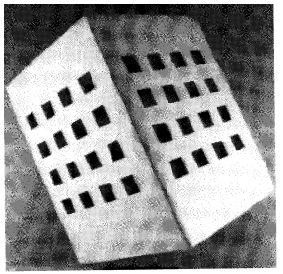
Euclidean Geometry

1.Self-Calibration

- Do not use any calibration object
- Moving camera in static scene
- The rigidity of the scene provides constraints on camera's internal parameters
- Correspondences b/w images are sufficient to recover both internal and external parameters

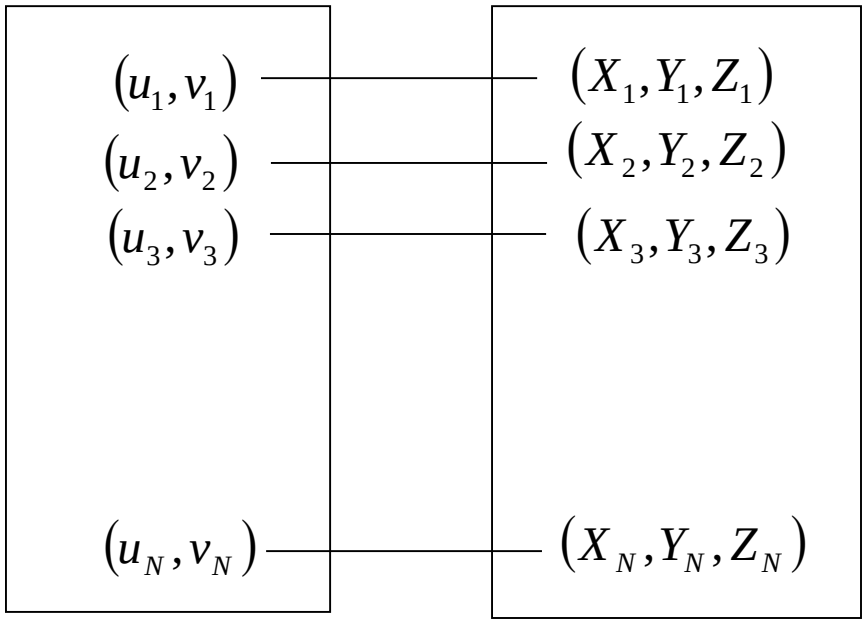
- Very flexible, but not reliable
  - Cannot always obtain reliable results due to many parameters to estimate

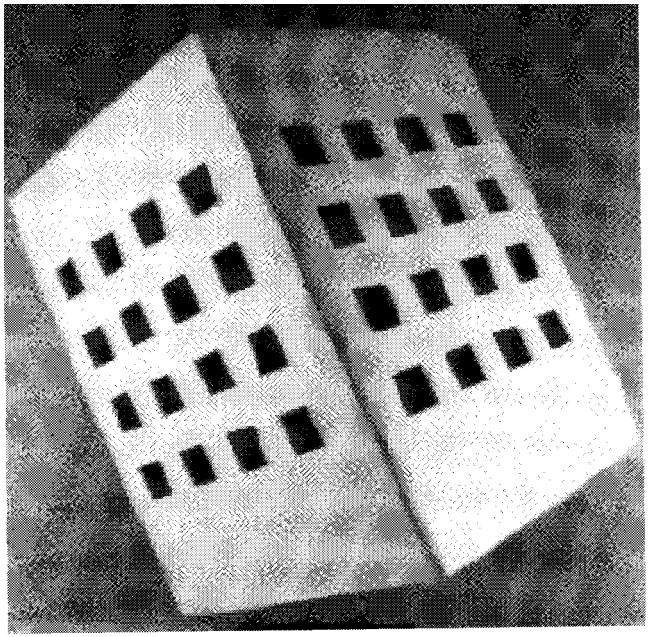
2. 3D reference object-based Calibration



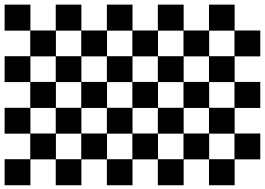
- Observing a calibration object with known geometry in 3D space
- Calibration object usually consists of two or three planes orthogonal to each other
- Pros:
  - Can be done very efficiently
- Cons:
  - Expensive calibration apparatus and elaborate setup required

Calibration object: Box (Calibration pattern)





3. Plane-based Calibration (Calibration by Homography)

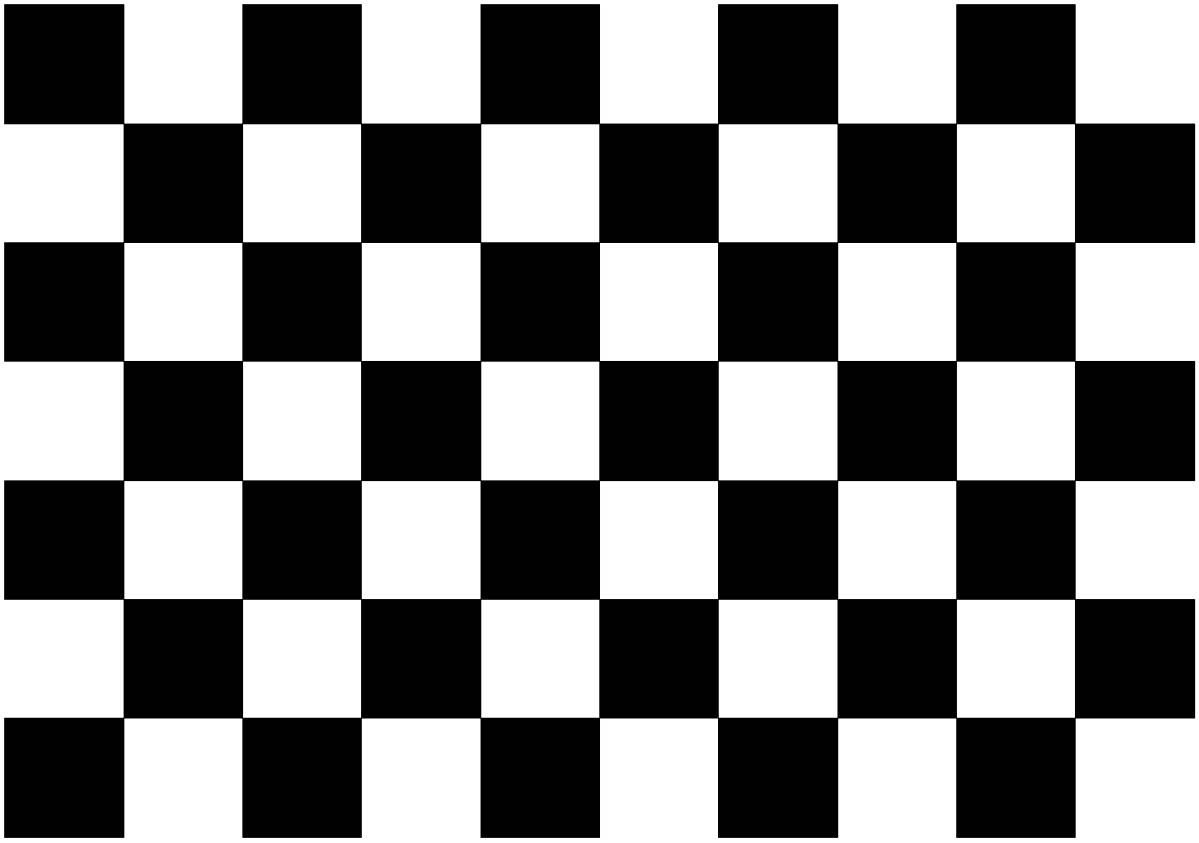


- Considered flexibility, robustness, and low cost

- Only require the camera to observe a planar pattern shown at a few (minimum 2) different orientations
  - Pattern can be printed and attached on planer surface
  - Either camera or planar pattern can be moved by hand

- More flexible and robust than traditional techniques
  - Easy setup
  - Anyone can make calibration pattern

Calibration object: Chessboard



This is a 9x6
OpenCV chessboard
http://sourceforge.net/projects/opencvlibrary/

from Zhang [Zhang99; Zhang00] and Sturm [Sturm99].

Calibrations with a Chessboard

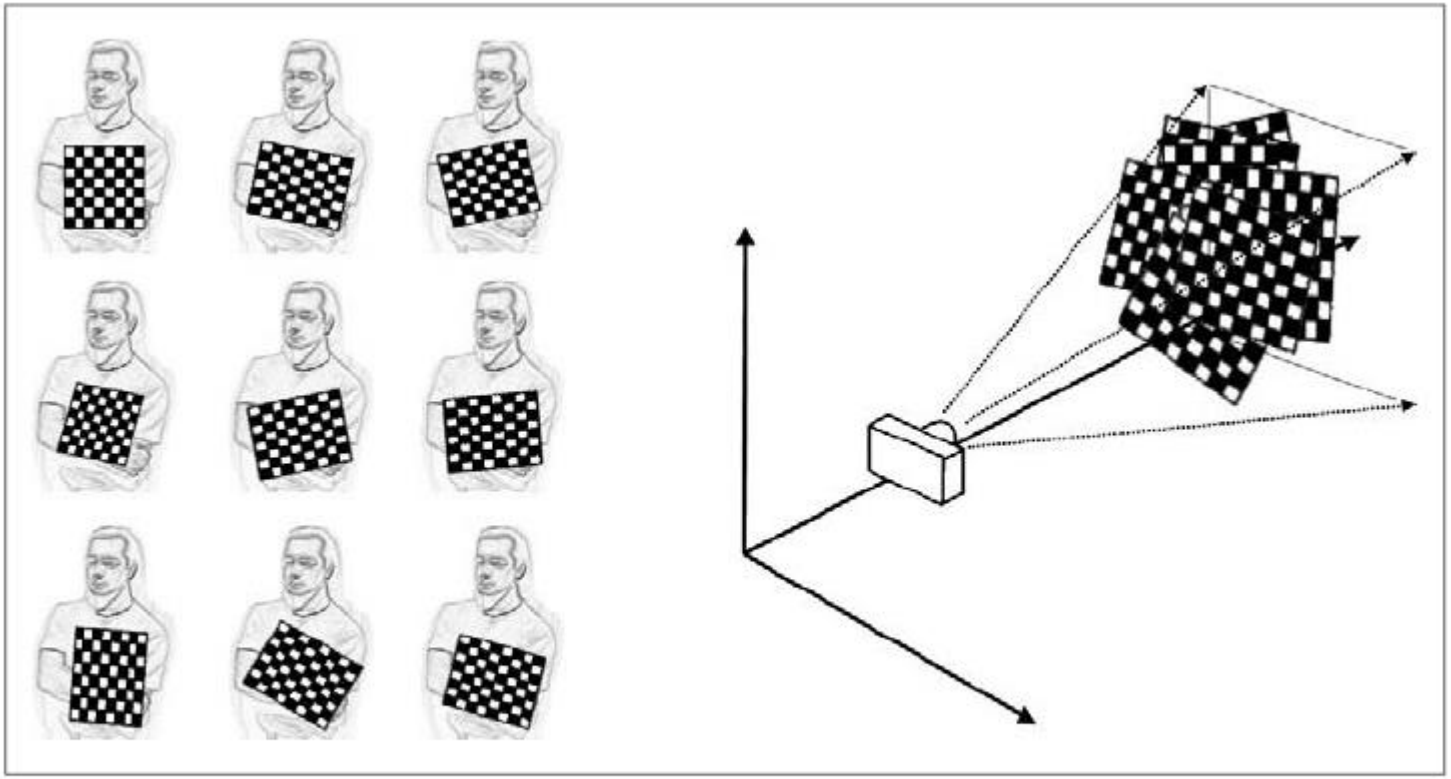
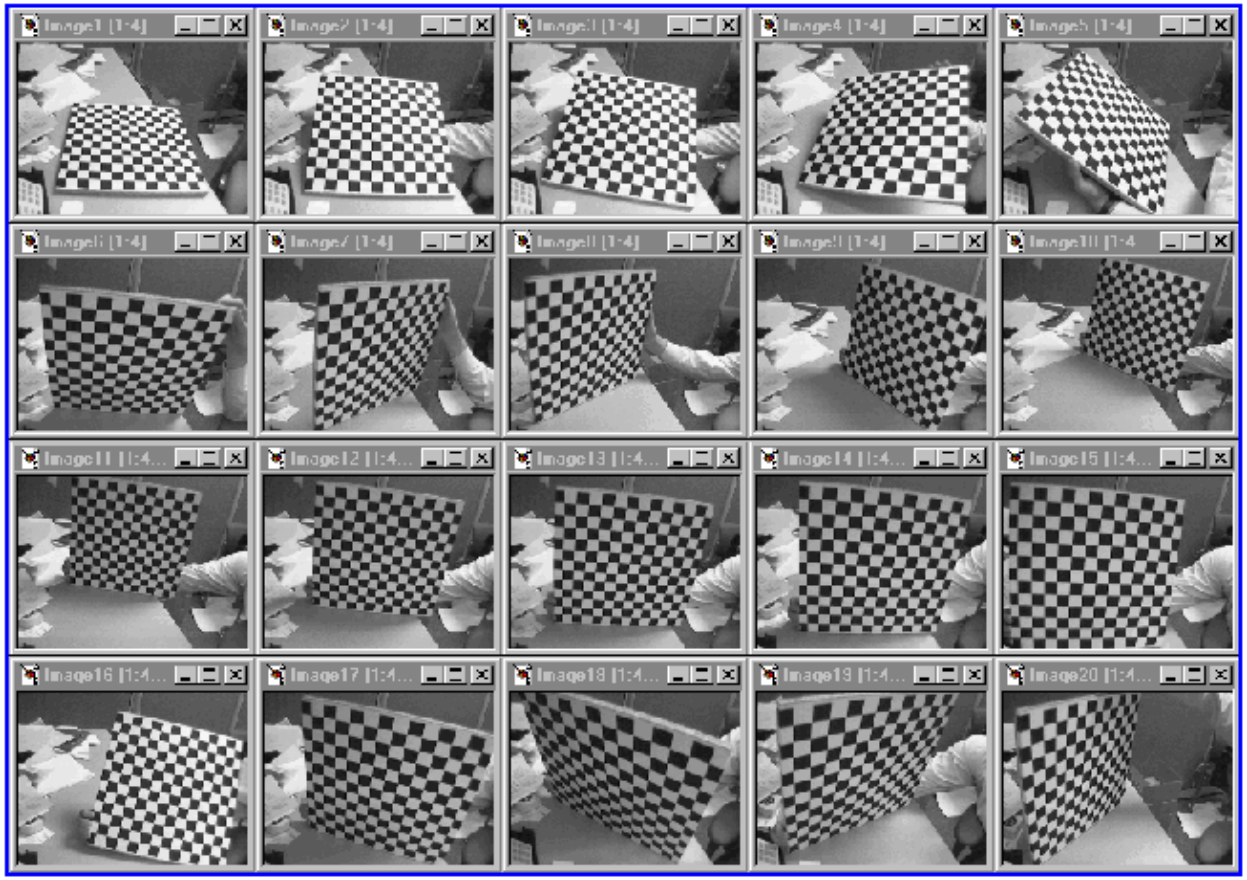
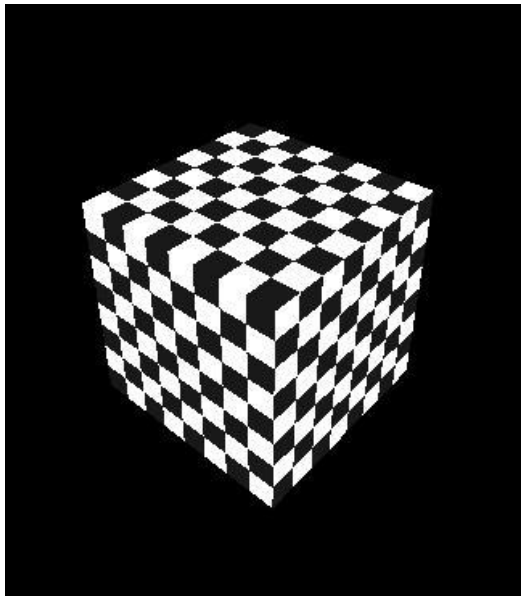


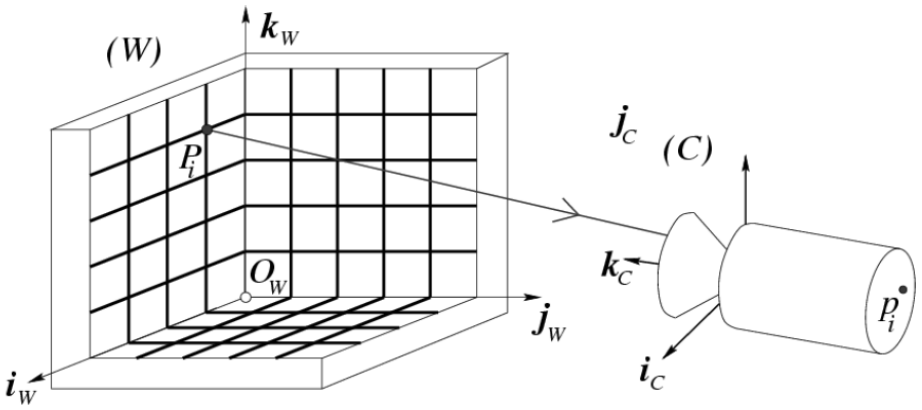
Figure 11-9. Images of a chessboard being held at various orientations (left) provide enough information to completely solve for the locations of those images in global coordinates (relative to the camera) and the camera intrinsics





Calibration Problem: Pattern-based

- **Given**
  - Calibration object with ***N*** corners
  - ***K*** views of this calibration object
- **Recover the camera parameters**
  - Distortion coeffients, intrinsic para., extrinsic para.



Calibration Procedure

- **Calibration object:**
  - we know positions of corners of grid with respect to a coordinate system.
- **Find the corners from images.**
- **Construct the equations**
  - relating image coordinates to world coordinates
- **Solve the equations** to get the camera parameters

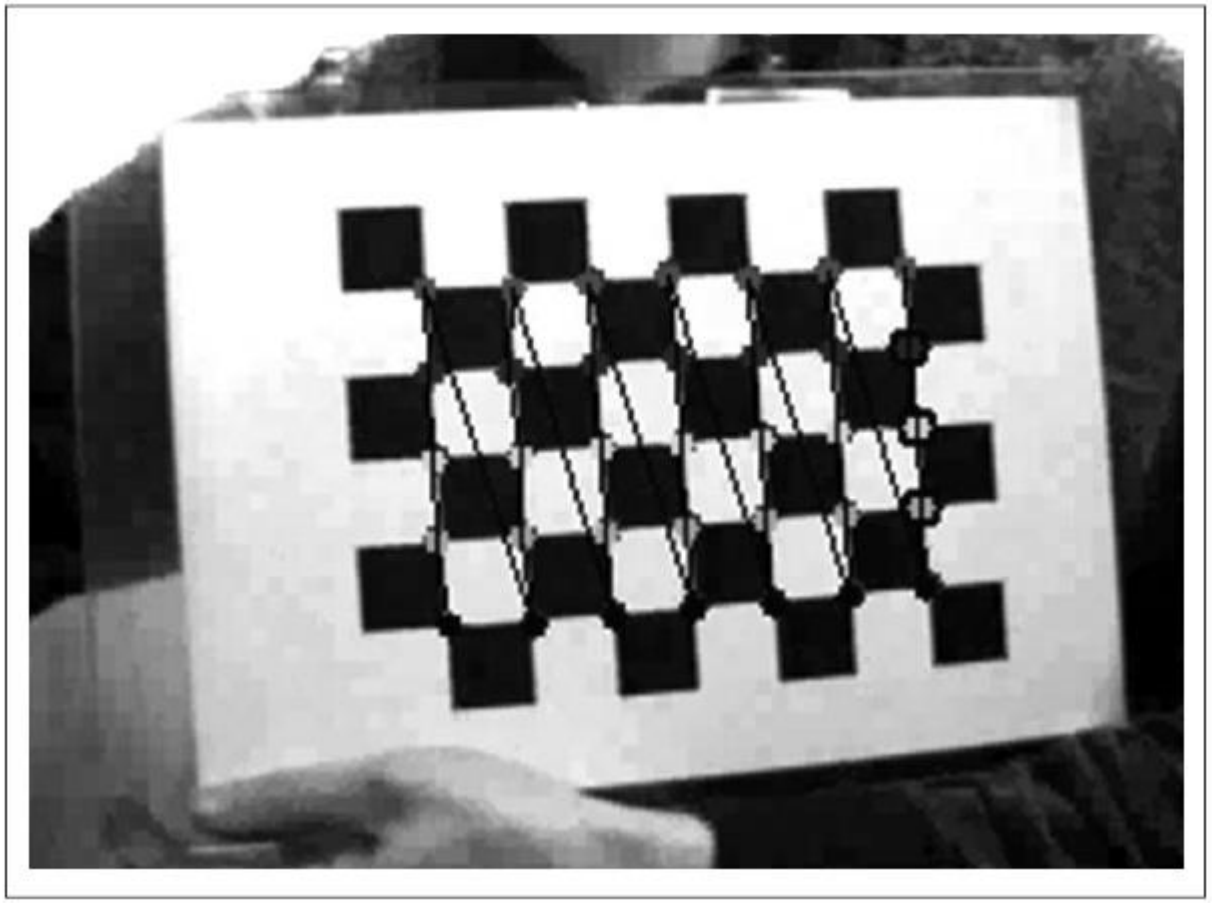
Finding the Corners

- OpenCV provides the function
  - pay attention to the parameters *pattern_size*
    - DO count the **interior** corners

```
int cvFindChessboardCorners(
  const void*   image,
  CvSize        pattern_size,
  CvPoint2D32f* corners,
  int*          corner_count = NULL,
  int           flags        = CV_CALIB_CB_ADAPTIVE_THRESH
);
```

  - cvFindCornerSubPix() to refine the detection

cvDrawChessboardCorners()



Least-Squares Parameter Estimation

- **Linear Least-Squares Methods(线性最小二乘法)**
  - A system of *p* linear equations in *q* unknowns:

$$\begin{cases} u_{11}x_1 + u_{12}x_2 + \ldots + u_{1q}x_q = y_1 \\ u_{21}x_1 + u_{22}x_2 + \ldots + u_{2q}x_q = y_2 \\ \ldots \\ u_{p1}x_1 + u_{p2}x_2 + \ldots + u_{pq}x_q = y_p \end{cases} \Leftrightarrow \mathrm{U}\mathbf{x} = \mathbf{y} \qquad \mathrm{U} \equiv \begin{pmatrix} u_{11} & u_{12} & \ldots & u_{1q} \\ u_{21} & u_{22} & \ldots & u_{2q} \\ \ldots & \ldots & \ldots & \ldots \\ u_{p1} & u_{p2} & \ldots & u_{pq} \end{pmatrix} \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_q \end{pmatrix} \text{ and } \mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ \ldots \\ y_q \end{pmatrix}$$

when $p < q$, the set of solutions to this equation forms a $(q - p)$-dimansional vector subspace of R^q

when $p = q$, there is unique solution

when $p > q$, there is no solution

We focuse on the overconstrained case $p > q$ and assumes that U has maximal rank q

Least-Squares Parameter Estimation

- **Linear Least-Squares Methods**
  - Normal Equations and the Pseudoinverse (正则方程和伪逆)
  - To find the vector ***x*** minimizing error measure ***E***:

$$E \overset{def}{=} \sum_{i=1}^{p} (u_{i1}x_1 + u_{i2}x_2 + \ldots + u_{iq}x_q - y_1)^2 = |\mathrm{U}\mathbf{x} - \mathbf{y}|^2$$

$$E = \mathbf{e}\cdot\mathbf{e} \text{ where } \mathbf{e} \overset{def}{=} \mathrm{U}\mathbf{x} - \mathbf{y} \quad \Rightarrow \quad \frac{\partial E}{\partial x_i} = 2\frac{\partial \mathbf{e}}{\partial x_i}\cdot\mathbf{e} = 0 \quad \text{for } i = 1,\ldots,q$$

The columns of U are the vectors $\mathbf{c}_j = (u_{1j},\ldots,u_{mj})^T \ (j = 1,\ldots,q)$

$$\frac{\partial \mathbf{e}}{\partial x_i} = \frac{\partial}{\partial x_i}\left[\begin{pmatrix} \mathbf{c}_1 & \ldots & \mathbf{c}_q \end{pmatrix} \begin{pmatrix} x_1 \\ \ldots \\ x_q \end{pmatrix} - \mathbf{y} \right] = \frac{\partial}{\partial x_i}\left(x_1\mathbf{c}_1 + \ldots + x_q\mathbf{c}_q - \mathbf{y} \right) = \mathbf{c}_i$$

Pseudoinverse of U

$$\mathbf{0} = \begin{pmatrix} \mathbf{c}_1^T \\ \ldots \\ \mathbf{c}_q^T \end{pmatrix} (\mathrm{U}\mathbf{x} - \mathbf{y}) = \mathrm{U}^T(\mathrm{U}\mathbf{x} - \mathbf{y}) \Leftrightarrow \mathrm{U}^T\mathrm{U}\mathbf{x} = \mathrm{U}^T\mathbf{y} \quad \Rightarrow \quad \mathbf{x} = \left[\left(\mathrm{U}^T\mathrm{U}\right)^{-1}\mathrm{U}^T \right]\mathbf{y} = \mathrm{U}^{\dagger}_{q\times q}\mathbf{y}$$

When U has maximal rank q, the U^TU is invertible

Least-Squares Parameter Estimation

- **Linear Least-Squares Methods**
  - Homogeneous Systems and Eigenvalue Problems (齐次系统和特征值问题)
  - A system of *p* linear equations in q unknowns, but the vector y is zero:

$$\begin{cases} u_{11}x_1 + u_{12}x_2 + \ldots + u_{1q}x_q = 0 \\ u_{21}x_1 + u_{22}x_2 + \ldots + u_{2q}x_q = 0 \\ \qquad\qquad \ldots \\ u_{p1}x_1 + u_{p2}x_2 + \ldots + u_{pq}x_q = 0 \end{cases} \Leftrightarrow \mathrm{U}\mathbf{x} = \mathbf{0}$$

Homogeneous Equation:

1. ***if x is a solution, so is λx (λ!=0)***
2. ***p=q & U is nonsignular, unique solution x=0;***
3. ***P>=q & U is signular (rank<q), nontrivial(nonzero)***

$$E = \left|\mathrm{U}\mathbf{x} - \mathbf{y}\right|^2 \Rightarrow E = \left|\mathrm{U}\mathbf{x}\right|^2$$

Constraint: $\left|\mathbf{x}\right|^2 = 1$

$$\Rightarrow E = \left|\mathrm{U}\mathbf{x}\right|^2 = \mathbf{x}^T\left(\mathrm{U}^T\mathrm{U}\right)\mathbf{x} = \mathbf{x}^T\mathrm{U}^*_{q\times q}\mathbf{x}$$

U^* - symmetric positive semi-definite, its eigenvalues>=0

$$\Rightarrow \mathrm{U}^*_{q\times q}\mathbf{e}_i = \lambda_i\mathbf{e}_i \quad \text{where } i = 1,\ldots,q \text{ and } 0 \le \lambda_1 \le \cdots \le \lambda_q$$

$$\mathbf{x} = \mu_1\mathbf{e}_1 + \cdots + \mu_q\mathbf{e}_q \quad \text{with} \quad \mu_1^2 + \cdots + \mu_q^2 = 1$$

When i=1, $E_{min} = \lambda_1^2$

$$E(\mathbf{x}) - E(\mathbf{e}_1) = \mathbf{x}^T\left(\mathrm{U}^T\mathrm{U}\right)\mathbf{x} - \mathbf{e}_1^T\left(\mathrm{U}^T\mathrm{U}\right)\mathbf{e}_1 = \sum_{j=1}^{q}\lambda_j^2\mu_j^2 - \lambda_1^2 \ge \lambda_1^2\left(\sum_{j=1}^{q}\mu_j^2 - 1\right) = 0$$

Least-Squares Parameter Estimation

- **Nonlinear Least-Squares Methods**
  - **A general system of *p* equations in *q* unknowns:**

$$\begin{cases} f_1(x_1, x_2, \cdots, x_q) = 0 \\ f_2(x_1, x_2, \cdots, x_q) = 0 \\ \cdots \\ f_p(x_1, x_2, \cdots, x_q) = 0 \end{cases} \Leftrightarrow \mathbf{f}(\mathbf{x}) = \mathbf{0} \quad \Rightarrow \quad E(\mathbf{x}) \overset{def}{=} |\mathbf{f}(\mathbf{x})|^2 = \sum_{i=1}^{p} f_i^2(\mathbf{x}) \quad \text{when } p > q$$

There is no general method to find E_{min}

when $p < q$, the solutions form a $(q - p)$-dimansional subset of R^q

when $p = q$, there is a finite set of solutions

when $p > q$, there is no solution

$$f_i(\mathbf{x} + \delta\mathbf{x}) = f_i(\mathbf{x}) + \delta x_1 \frac{\partial f_i}{\partial x_1}(\mathbf{x}) + \cdots + \delta x_q \frac{\partial f_i}{\partial x_q}(\mathbf{x}) + O\left(|\delta\mathbf{x}|^2\right) \approx f_i(\mathbf{x}) + \nabla f_i(\mathbf{x}) \cdot \delta\mathbf{x},$$

The gradient of f_i at the point x

$$\nabla f_i(\mathbf{x}) = \left(\frac{\partial f_i}{\partial x_1}, \ldots, \frac{\partial f_i}{\partial x_q} \right)^T$$

$$\Rightarrow \mathbf{f}(\mathbf{x} + \delta\mathbf{x}) \approx \mathbf{f}(\mathbf{x}) + \mathrm{J}_{\mathbf{f}}(x)\delta\mathbf{x}, \quad \mathrm{J}_{\mathbf{f}}(x) \overset{def}{=} \begin{pmatrix} \nabla f_1^T(\mathbf{x}) \\ \cdots \\ \nabla f_p^T(\mathbf{x}) \end{pmatrix} = \begin{pmatrix} \frac{\partial f_i}{\partial x_q}(\mathbf{x}) & \cdots & \frac{\partial f_i}{\partial x_q}(\mathbf{x}) \\ \cdots & \cdots & \cdots \\ \frac{\partial f_i}{\partial x_q}(\mathbf{x}) & \cdots & \frac{\partial f_i}{\partial x_q}(\mathbf{x}) \end{pmatrix}$$

Jacobian matrix of f(x)

Least-Squares Parameter Estimation

- **Nonlinear Least-Squares Methods**
  - **Newton's method:**
    - Square Systems of Nonlinear Equations ($p=q$)

$$\mathbf{f}(\mathbf{x}+\delta\mathbf{x}) \approx \mathbf{f}(\mathbf{x}) + \mathrm{J}_{\mathbf{f}}(x)\delta\mathbf{x} \xrightarrow{\mathbf{f}(\mathbf{x}+\delta\mathbf{x})\approx\mathbf{0}} \mathrm{J}_{\mathbf{f}}(x)\delta\mathbf{x} = -\mathbf{f}(\mathbf{x})$$

    - Overconstrained Systems of Nonlinear Equations ($p>q$)

$$\left.\begin{aligned} \mathbf{F}(\mathbf{x}) &= \frac{1}{2}\nabla E(\mathbf{x}) = \mathrm{J}_{\mathbf{f}}(x)\delta\mathbf{x} \\ \mathrm{J}_{\mathbf{F}}(x) &= \mathrm{J}_{\mathbf{f}}^{T}(x)\mathrm{J}_{\mathbf{f}}(x) + \sum_{i=1}^{p} f_i(\mathbf{x})\mathrm{H}_{f_i}(\mathbf{x}) \end{aligned}\right\}$$

$$\Rightarrow \left[\mathrm{J}_{\mathbf{f}}^{T}(x)\mathrm{J}_{\mathbf{f}}(x) + \sum_{i=1}^{p} f_i(\mathbf{x})\mathrm{H}_{f_i}(\mathbf{x})\right]\delta\mathbf{x} = -\mathrm{J}_{\mathbf{f}}^{T}(x)\mathbf{f}(\mathbf{x})$$

$$\mathrm{H}_{f_i}(\mathbf{x}) \overset{def}{=} \begin{pmatrix} \frac{\partial^2 f_i}{\partial x_1 \partial x_1}(\mathbf{x}) & \cdots & \frac{\partial^2 f_i}{\partial x_1 \partial x_q}(\mathbf{x}) \\ \cdots & \cdots & \cdots \\ \frac{\partial^2 f_i}{\partial x_1 \partial x_q}(\mathbf{x}) & \cdots & \frac{\partial^2 f_i}{\partial x_q \partial x_q}(\mathbf{x}) \end{pmatrix}$$

The Hessian of $f_i(X)$

Least-Squares Parameter Estimation

- **Nonlinear Least-Squares Methods**
  - **The Gauss-Newton and Levenberg-Marquardt Algorithms**
    - Computing Hessians of f_i is difficult and/or expensive

$$E(\mathbf{x}) = \left|\mathbf{f}(\mathbf{x})\right|^2 \xrightarrow{\mathbf{f}(\mathbf{x}+\delta\mathbf{x})\approx\mathbf{f}(\mathbf{x})+\mathrm{J}_{\mathbf{f}}(x)\delta\mathbf{x}}$$

$$\rightarrow E(\mathbf{x}+\delta\mathbf{x}) = \left|\mathbf{f}(\mathbf{x}+\delta\mathbf{x})\right|^2 \approx \left|\mathbf{f}(\mathbf{x})+\mathrm{J}_{\mathbf{f}}(x)\delta\mathbf{x}\right|^2$$

    - Gauss-Newton Algorithm

$$\mathrm{J}_{\mathbf{f}}^{\dagger}(x)\delta\mathbf{x} = -\mathbf{f}(\mathbf{x})$$

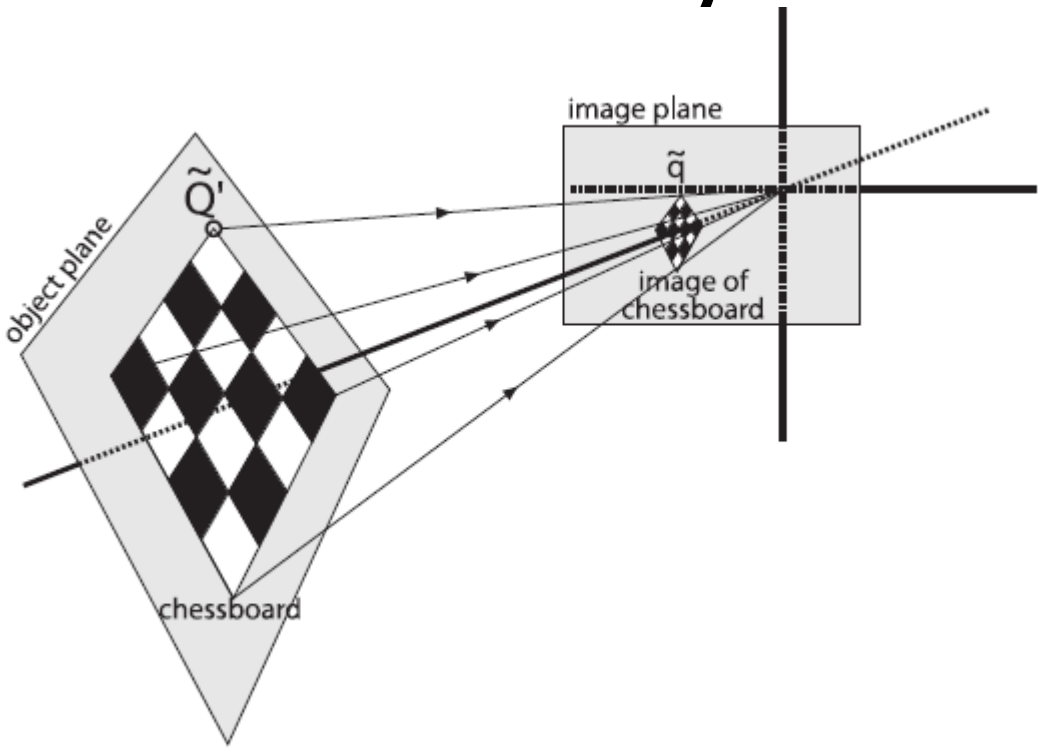
$$\mathrm{J}_{\mathbf{f}}^{T}(x)\mathrm{J}_{\mathbf{f}}(x)\delta\mathbf{x} = -\mathrm{J}_{\mathbf{f}}^{T}(x)\mathbf{f}(\mathbf{x})$$

    - Levenberg-Marquardt Algorithm

$$\left[\mathrm{J}_{\mathbf{f}}^{T}(x)\mathrm{J}_{\mathbf{f}}(x) + \mu\mathrm{Id}\right]\delta\mathbf{x} = -\mathrm{J}_{\mathbf{f}}^{T}(x)\mathbf{f}(\mathbf{x})$$

Calibration by Homography

Plane → Plane



[Zhang99, Zhang00]

$$\tilde{Q} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix}^t$$

$$\tilde{q} = \begin{bmatrix} x & y & 1 \end{bmatrix}^t$$

Express the action of the homography simply as:

$$\tilde{q} = sH_{3\times 4}\tilde{Q}$$

H has two parts:

i.physical transformation

locates the object plane we are viewing $\boldsymbol{M}_{\mathbf{ext}}$

i.Projection

introduces the camera intrinsic matrix $\boldsymbol{M}_{\boldsymbol{int}}$

Calibration via Homography

[Zhang99, Zhang00]

Zhengyou Zhang, **A Flexible New Technique for Camera Calibration**
IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), 2000.

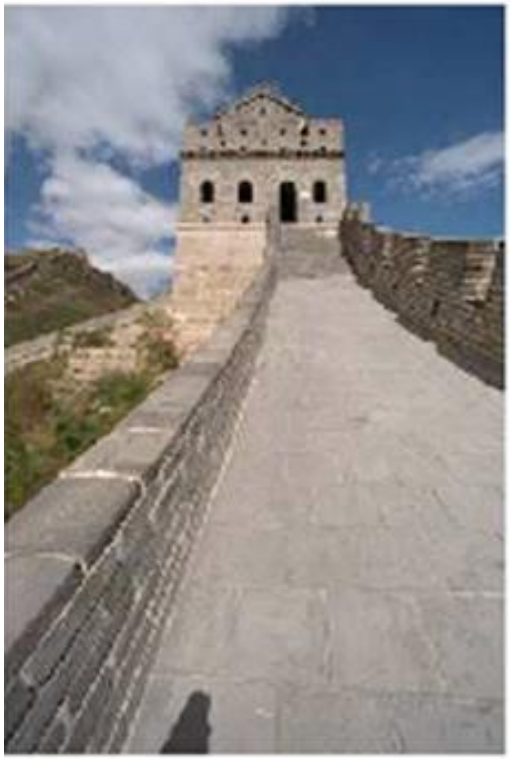
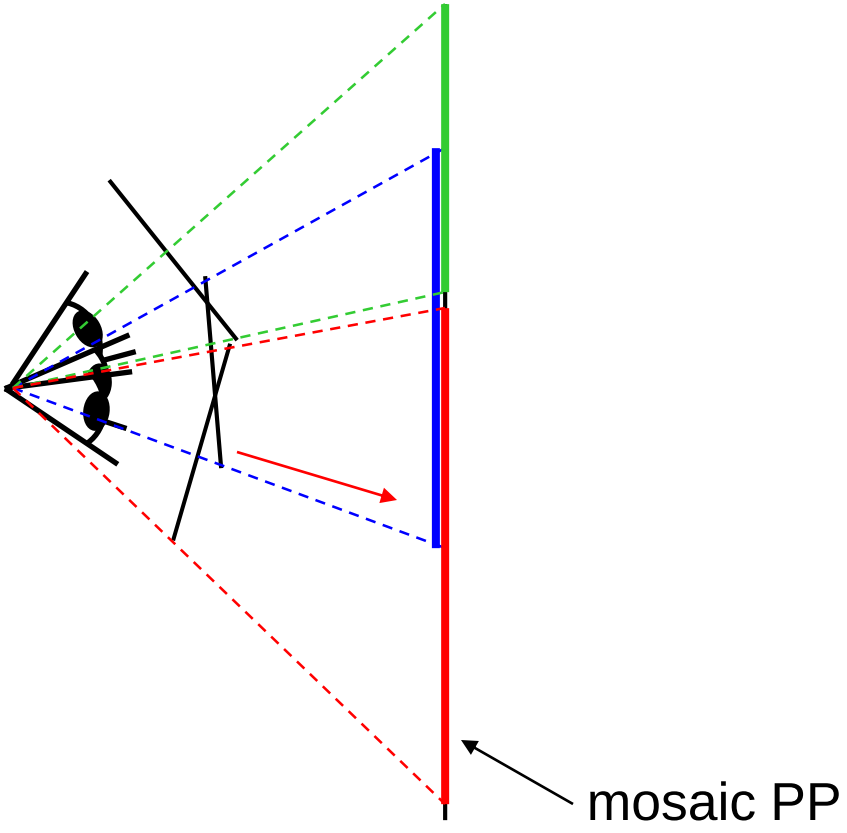


Image reprojection: mosaic



The mosaic has a natural interpretation in 3D

- The images are reprojected onto a common plane
- The mosaic is formed on this plane
- Mosaic is a *synthetic wide-angle camera*

Source: Steve Seitz

Homography

How to relate two images from the same camera center?

- how to map a pixel from PP1 to PP2?

Think of it as a 2D **image warp** from one image to another.

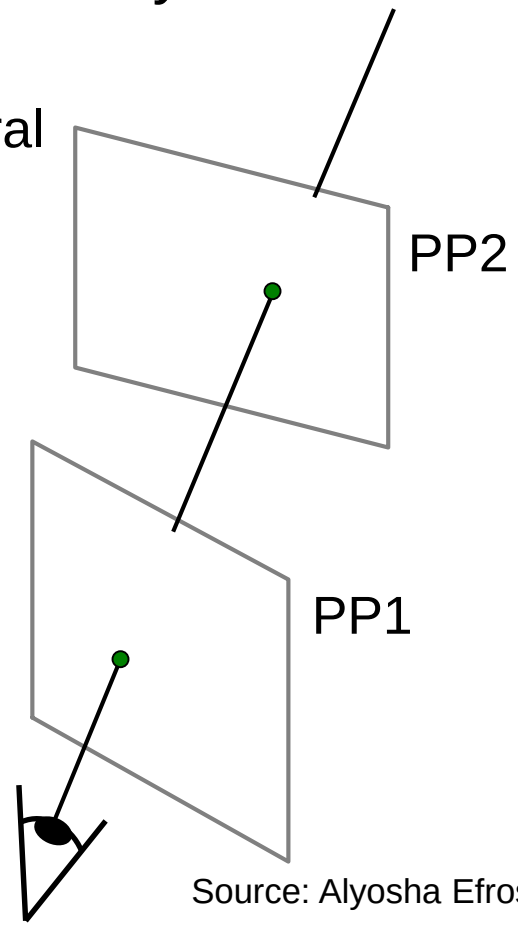
A projective transform is a mapping between any two PPs with the same center of projection

- rectangle should map to arbitrary quadrilateral
- parallel lines aren't
- but must preserve straight lines

called **Homography**

$$\begin{bmatrix} wx' \\ wy' \\ w \end{bmatrix} = \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\mathbf{p'} \qquad \mathbf{H} \qquad \mathbf{p}$$



Source: Alyosha Efros

Calibration via Homography

$$M_{int} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad M_{ext} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \longrightarrow \tilde{q} = sM_{int}M_{ext}\tilde{Q}$$

Focus on the target plane, and set the plane as Z=0

$$\tilde{Q}' = \begin{bmatrix} X & Y & 0 & 1 \end{bmatrix}^t$$

$$\tilde{q}' = sH_{3\times 3}\tilde{Q}'$$

$$\tilde{q} = sM_{int}\begin{bmatrix} r_1 & r_2 & r_3 & \mathbf{t} \end{bmatrix}\begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = sM_{int}\begin{bmatrix} r_1 & r_2 & \mathbf{t} \end{bmatrix}\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

Homography Matrix

Suggested readings

- **Learning OpenCV**, pp.389-392

- Zhengyou Zhang, **A Flexible New Technique for Camera Calibration**, IEEE Transactions on Pattern Analysis and Machine Intelligence (T-PAMI), 2000.

Homography

- ***H*** *relates the positions of the points on a source image plane* to the points on the destination image plane.

$$p_{\text{dst}} = Hp_{\text{src}}, \quad p_{\text{src}} = H^{-1}p_{\text{dst}}$$

$$p_{\text{dst}} = \begin{bmatrix} x_{\text{dst}} \\ y_{\text{dst}} \\ 1 \end{bmatrix}, \quad p_{\text{src}} = \begin{bmatrix} x_{\text{src}} \\ y_{\text{src}} \\ 1 \end{bmatrix}$$

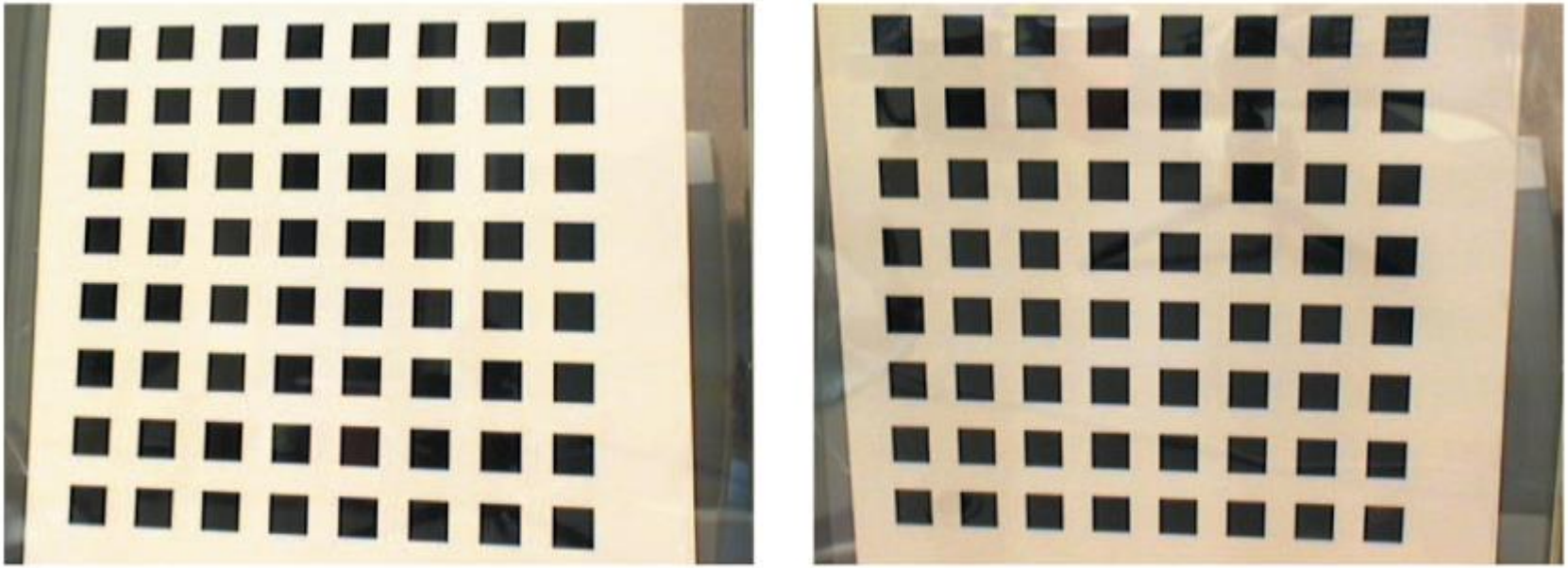


Figure 5: First and second images after having corrected radial distortion

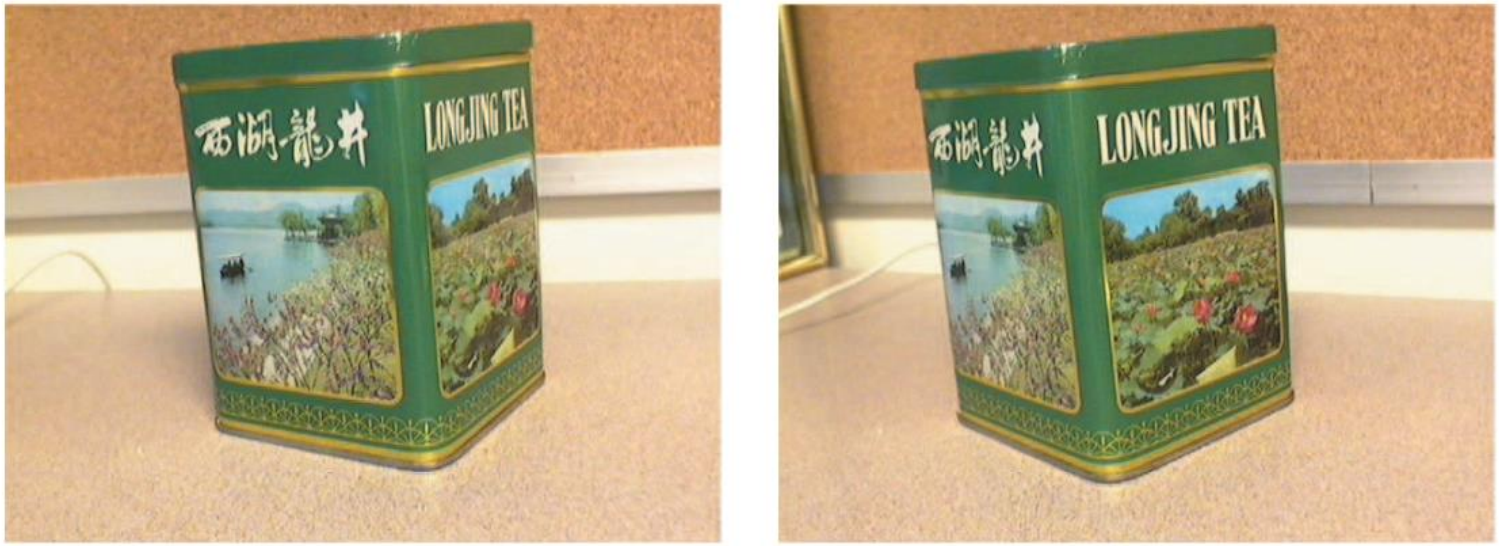


Figure 6: Two images of a tea tin

Figure 7: Three rendered views of the reconstructed tea tin

OpenCV Func

```
void cvFindHomography(
    const CvMat* src_points,
    const CvMat* dst_points,
    CvMat*       homography
);
```

- Homogeneous matrix: $H_{33}=1$
  - Degree of freedom for H is **8**
- At lease **4** points needed.
  - More is helpful.

How Many Chess Corners (*pattern_size*) for Calibration*?*

- How many parameters we have?
  - 6 extrinsic parameters $(\theta, \varphi, \psi, t_x, t_y, t_z)$
  - 4 intrinsic parameters $(f_x,\ f_y,\ c_x,\ c_y)$
  - 5 distortion parameters $(k_1,\ k_2,\ p_1,\ p_2,\ k_3)$
- 2D geometry -- 5
  - 3 points yield 6 constraints (in principle)
  - Enough for 5 parameters!
  - More for robustness

3D geometry

2D geometry

How Many Chess Corners (*pattern_size*) for Calibration*?*

- Each view:
  - Gives **8** equations, because a square can described by **4** points.
  - **Six** individual parameters (extrinsic) : R, t
  - Common parameters (intrinsic):
    - intrinsic matrix (4 parameters)
    - distortion coefficients
    - …
- How about giving ***n*** views??

How Many Chess Corners (*pattern_size*) for Calibration*?*

- The number of parameters increases as more views are involved.
  - Suppose *N* corners and *K* views(images), therefore, ***2NK*** corners constraints
  - The parameters: ***K*6+4***
  - 2NK >= K*6+4, that is, (N-3)K>=2, s.t. K>1
  - If K==2, N=4: Two views each with four corners.
  - Again, we use more for robustness.
    - # of views >=**10**, size of chessboard >= **7x8**

Opencv Func

- cvCalibrateCamera2()
  - Must have the corners' position at hand!
  - Note that with intrinsic parameters:
    - We can project sth: 3D->2D ☺
    - But only one line projected to one pixel ☹
  - It is used by stereo calibration
    - Calibrate two cameras at the same time

cvCalibrateCamera2()

```
void cvCalibrateCamera2(
  CvMat*    object_points,
  CvMat*    image_points,
  int*      point_counts,
  CvSize    image_size,
  CvMat*    intrinsic_matrix,
  CvMat*    distortion_coeffs,
  CvMat*    rotation_vectors    = NULL,
  CvMat*    translation_vectors = NULL,
  int       flags               = 0
);
```

cvCalibrateCamera2()

- ***object_points***
  - Describe physical coordinates of the known object points.
  - *N*3* matrix
  - We have *K* points on *M* images: *N* = *K*M*
    - Actually *K* points repeated *M* times
  - The unit and coordinates are based on users.

$$(0,0),(0,1),(0,2),\ldots,(1,0),(2,0),\ldots,(1,1),\ldots,(S_{\text{width}}-1,S_{\text{height}}-1)$$

  - Plane is the simplest way~

cvCalibrateCamera2

- ***Rotation_vectors***
  - *M * 3*
  - V_{1*3} : Euler angle
    - An axis that the chessboard moves around
    - The magnitude represents the angle of rotation
    - Convert to 3*3 matrix
      - Use cvRodrigues2()

cvCalibrateCamera2

- ***Flags***
  - CV_CALIB_USE_INTRINSIC_GUESS
    - Prior knowledge of intrinsic parameters
  - CV_CALIB_FIX_*****
    - If we have known sth, just fix it!
  - CV_CALIB_ZERO_TANGENT_DIST
    - For high end cameras, turn off the (p1 ,p2)

cvFindExtrinsicCameraParams2()

```
void cvFindExtrinsicCameraParams2(
    const CvMat* object_points,
    const CvMat* image_points,
    const CvMat* intrinsic_matrix,
    const CvMat* distortion_coeffs,
    CvMat*       rotation_vector,
    CvMat*       translation_vector
);
```

Do the Calibration Now!

- cvUndistort2()
- cvFindChessboardCorners()
- cvCalibrateCamera2()
  - cvFindExtrinsicCameraParams2()