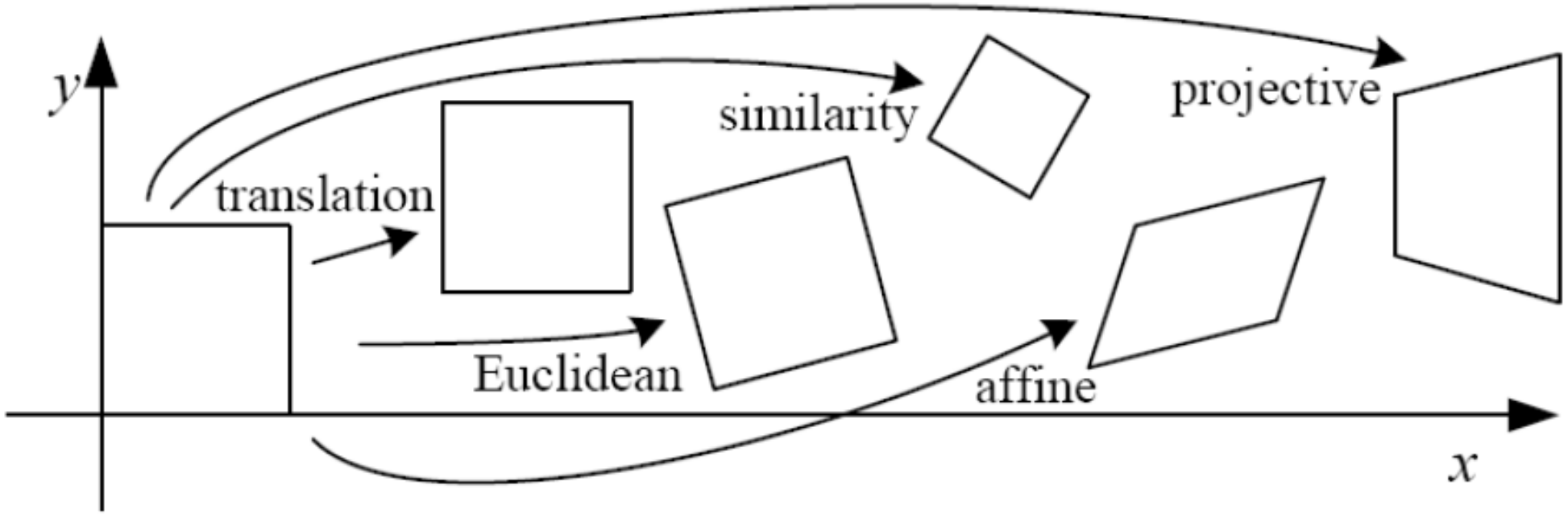
# Image Transformations
# &
# Camera Calibration

Mašinska vizija, 2018.

# Image transformations



# What've we learnt so far?

| Transformation | Matrix | # DoF | Preserves | Icon |
|---|---|---|---|---|
| translation | $\left[\, \boldsymbol{I} \mid \boldsymbol{t} \,\right]_{2\times 3}$ | 2 | orientation |  |
| rigid (Euclidean) | $\left[\, \boldsymbol{R} \mid \boldsymbol{t} \,\right]_{2\times 3}$ | 3 | lengths |  |
| similarity | $\left[\, s\boldsymbol{R} \mid \boldsymbol{t} \,\right]_{2\times 3}$ | 4 | angles |  |
| affine | $\left[\, \boldsymbol{A} \,\right]_{2\times 3}$ | 6 | parallelism |  |
| projective | $\left[\, \tilde{\boldsymbol{H}} \,\right]_{3\times 3}$ | 8 | straight lines |  |

# Image transformations and mapping

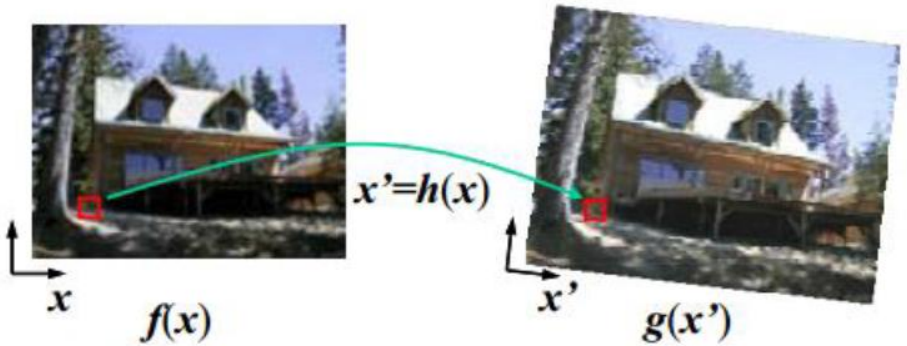
- There are two types of image transformations:
  - uniform
    - uniform resize (scaling)
    - translation
    - rotation
  - non uniform
    - nonuniform resize (stretching) (aspect ratio is not preserved)
    - affine transform
    - perspective transform
    - different polynomial transforms that are pixel position dependent
- These operations are a little less trivial than you might think, because resizing immediately implies questions about how pixels are interpolated (for enlargement) or merged (for reduction)

# Image mapping - forward warping

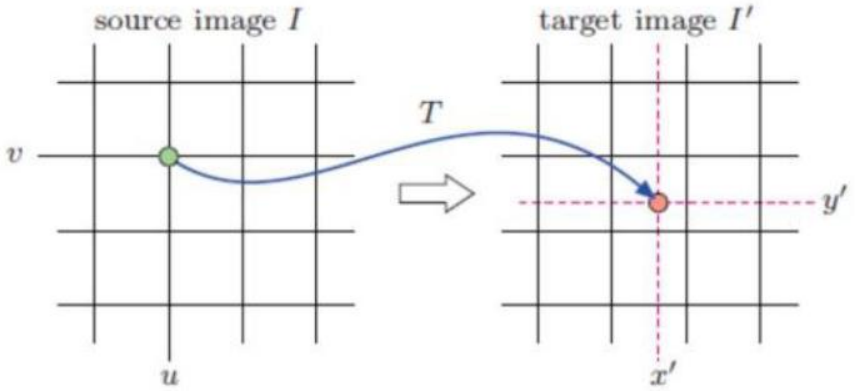
**procedure** *forwardWarp*$(f, \boldsymbol{h}, \textbf{out}\ g)$:

For every pixel $\boldsymbol{x}$ in $f(\boldsymbol{x})$

1. Compute the destination location $\boldsymbol{x}' = \boldsymbol{h}(\boldsymbol{x})$.
2. Copy the pixel $f(\boldsymbol{x})$ to $g(\boldsymbol{x}')$.




(a)

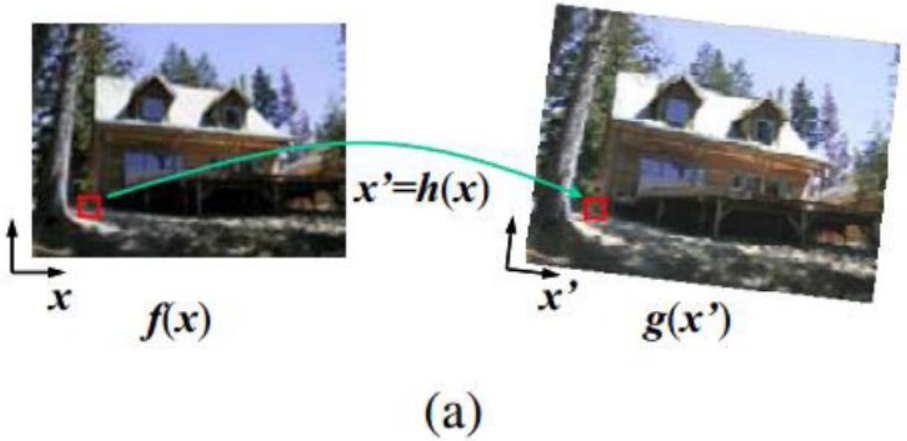


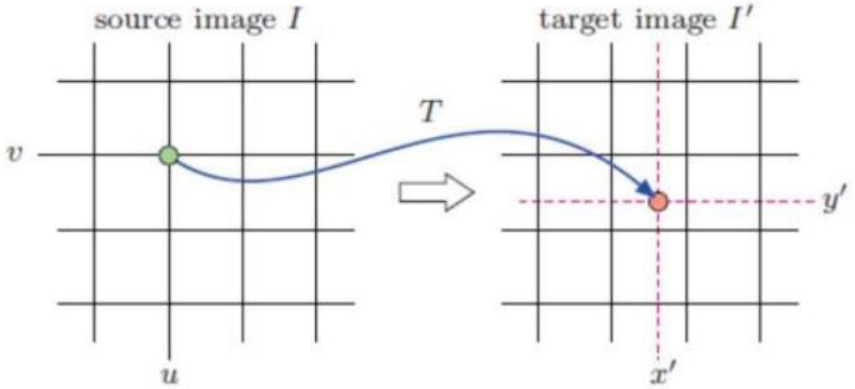
# Image mapping - forward warping

**procedure** *forwardWarp*$(f, \boldsymbol{h}, \textbf{out}\ g)$:

For every pixel $\boldsymbol{x}$ in $f(\boldsymbol{x})$

1. Compute the destination location $\boldsymbol{x}' = \boldsymbol{h}(\boldsymbol{x})$.
2. Copy the pixel $f(\boldsymbol{x})$ to $g(\boldsymbol{x}')$.



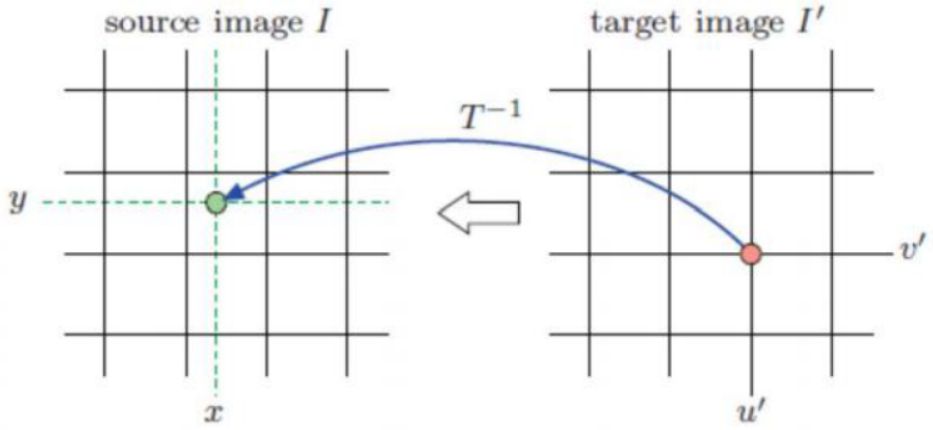

- Issue: In case of upscaling some pixels in resulting image will be left blank. Also, in case of downscaling it will occur that more than one pixel will be projected into the same output pixel. **-> Reverse logic**

# Image mapping – backward warping

**procedure** *inverseWarp*$(f, \boldsymbol{h}, \textbf{out}\ g)$:

For every pixel $\boldsymbol{x}'$ in $g(\boldsymbol{x}')$

1. Compute the source location $\boldsymbol{x} = \hat{\boldsymbol{h}}(\boldsymbol{x}')$
2. Resample $f(\boldsymbol{x})$ at location $\boldsymbol{x}$ and copy to $g(\boldsymbol{x}')$



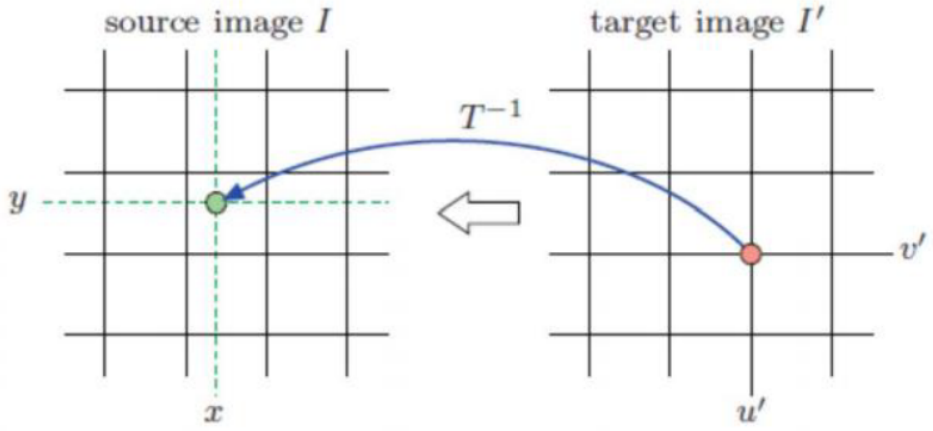

- Each pixel in the resulting image now looks for its source
- All resulting pixels are now valid
- Issue: Source coordinated might not be integer

# Image mapping – backward warping

**procedure** *inverseWarp*$(f, \boldsymbol{h}, \textbf{out}\ g)$:

For every pixel $\boldsymbol{x}'$ in $g(\boldsymbol{x}')$

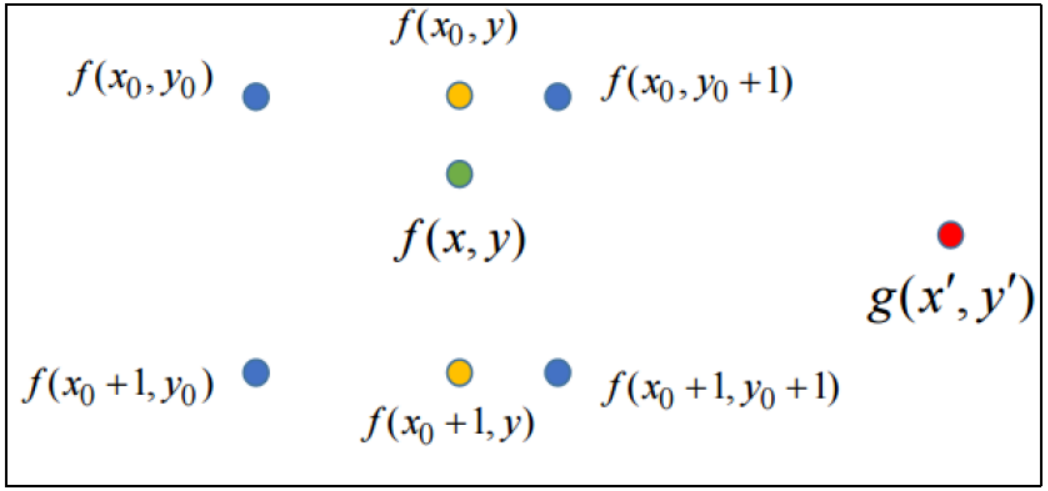
1. Compute the source location $\boldsymbol{x} = \hat{\boldsymbol{h}}(\boldsymbol{x}')$
2. Resample $f(\boldsymbol{x})$ at location $\boldsymbol{x}$ and copy to $g(\boldsymbol{x}')$

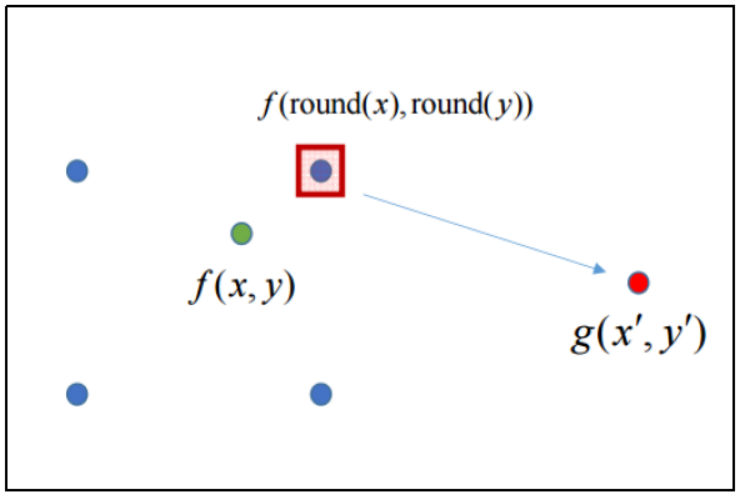


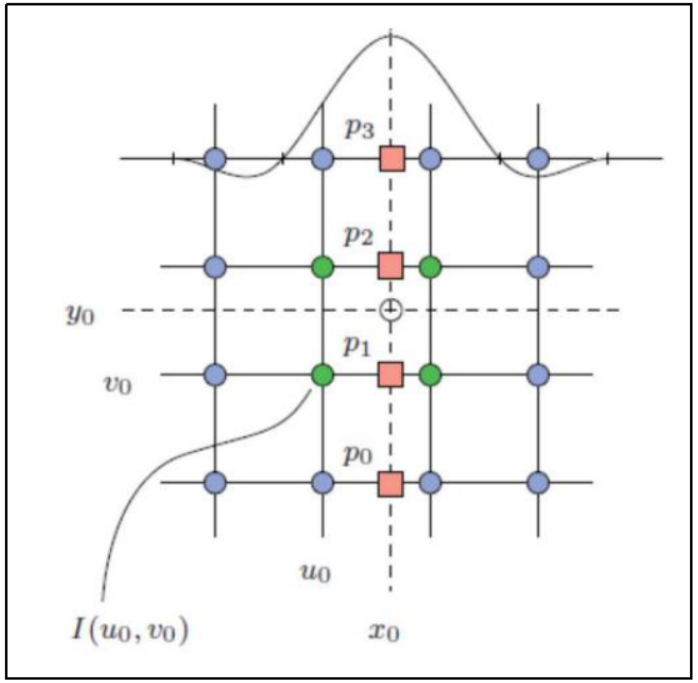

- Each pixel in the resulting image now looks for its source
- All resulting pixels are now valid
- Issue: Source coordinated might not be integer
- -> Interpolation

# Interpolation

- Nearest neighbor
- Bilinear
- Bicubic



# Interpolation - example

ULAZ

NEAREST

BILINEAR

BICUBIC

# Uniform mapping - RESIZE

- void cv: : **resize**(
  cv: : InputArray src, // Input image
  cv: : OutputArray dst, // Result image
  cv: : Size dsize, // New size
  double fx = 0, // x-rescale
  double fy = 0, // y-rescale
  int interpolation = CV: : INTER_LINEAR // interpolation method
  ) ;
- We can specify the size of the output image in two ways:
  - use *absolute sizing* - the dsize argument directly sets the size
  - use *relative sizing* - set dsize to cv: : Size( 0, 0), and set fx and fy to the scale factors to be applied to the x- and y-axes, respectively.
- IMPORTANT: don't mix cv::resize with cv::Mat::resize

# Non-uniform mappings – geometric transforms

- **geometric transforms** - The functions that can stretch, shrink, warp, and/or rotate an image.
- There are two flavors of geometric transforms in 3D to 2D space:
  - transforms that use a 2 × 3 matrix, which are called *affine transforms*;
  - and transforms based on a 3 × 3 matrix, which are called *perspective transforms* or *homographies*.

# Affine transform - cv: : warpAffine( )

- void cv: : **warpAffine**(
  cv: : InputArray src, // Input image
  cv: : OutputArray dst, // Result image
  cv: : InputArray M, // 2-by-3 transform mtx
  cv: : Size dsize, // Destination image size
  int flags = cv: : INTER_LINEAR, // Interpolation, inverse
  int borderMode = cv: : BORDER_CONSTANT, // Pixel extrapolation
  const cv: : Scalar& borderValue = cv: : Scalar() // For constant borders
  ) ;
- cv::invertAffineTransform(): Inverting an affine transformation

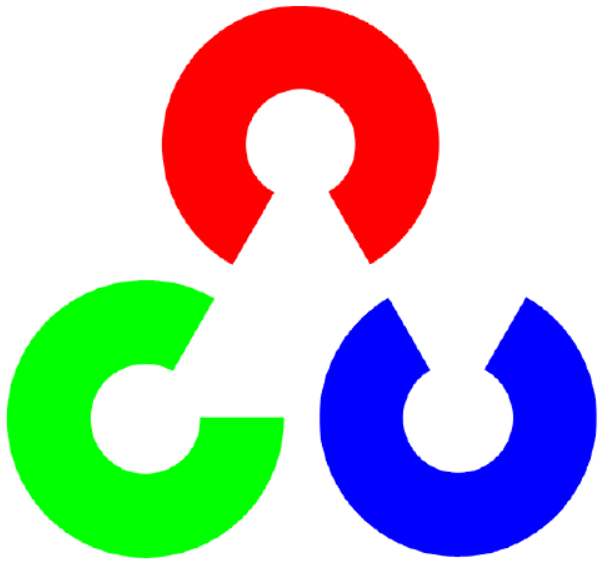
# How to create the M transformation matrix?

- cv: : Mat cv: : **getAffineTransform**( // Return 2-by-3 matrix
  const cv: : Point2f* src, // Coordinates *three* of vertices
  const cv: : Point2f* dst // Target coords, three vertices
  ) ;

  cv: : Mat cv: : **getRotationMatrix2D**( // Return 2-by-3 matrix
  cv: : Point2f center, // Center of rotation
  double angle, // Angle of rotation
  double scale // Rescale after rotation
  ) ;

# Example 1 – resize and rotate

- Open warp_affine_template.cpp
- Perform simple resize transformation
- Perform image rotation for a predefined angle



# Perspective transform - cv: : warpPerspective()

- void cv: : **warpPerspective**(
  cv: : InputArray src, // Input image
  cv: : OutputArray dst, // Result image
  cv: : InputArray M, // 3-by-3 transform mtx
  cv: : Size dsize, // Destination image size
  int flags = cv: : INTER_LINEAR, // Interpolation, inverse
  int borderMode = cv: : BORDER_CONSTANT, // Extrapolation method
  const cv: : Scalar& borderValue = cv: : Scalar() // For constant borders
  ) ;

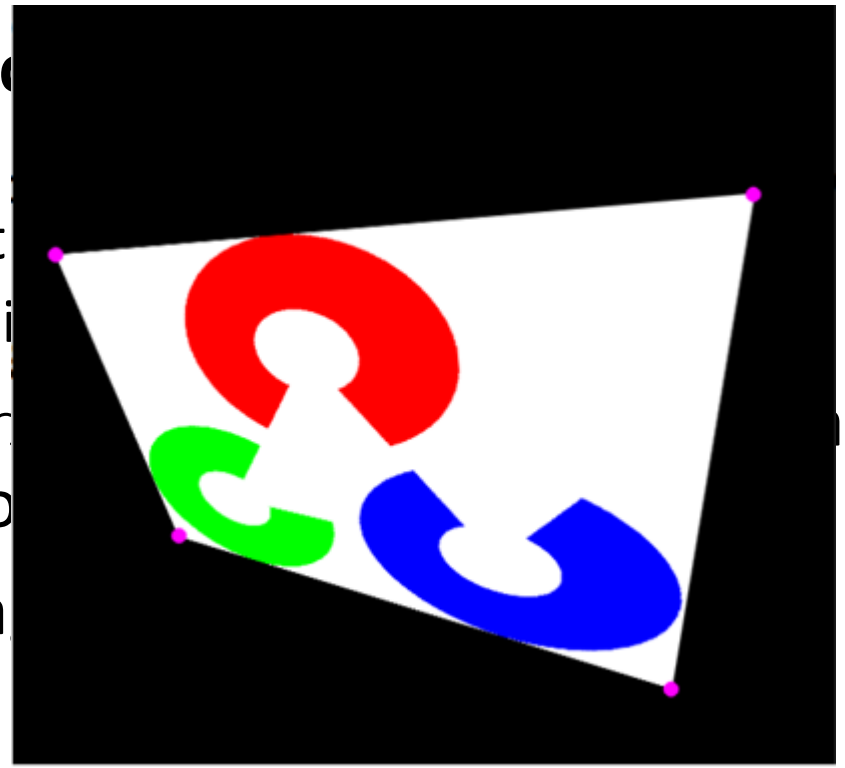
- cv: : Mat cv: : **getPerspectiveTransform**( // Return 3-by-3 matrix
  const cv: : Point2f* src, // Coordinates of *four* vertices
  const cv: : Point2f* dst // Target coords, four vertices
  ) ;

# Example 2 – perspective transform

- Open perspective_warp_template.cpp
- Define two sets of corner points **srcQuad** and **dstQuad**
  - Points are `cv::Point2f` type
  - Use `.cols` nad `.rows` to define the original image corners
  - Define arbitrary corners for the new image
- Use `cv::getPerspetiveTransform` with generated pairs of points to acquire the transformation matrix.
- Apply the matrix and warp the image

# Example 2 – perspective transform

- Open perspective_warp_template.cpp
- 
  points **sr**
  type
  o define t
  the new i
- veTran
  irs of
  formatio
- the ima



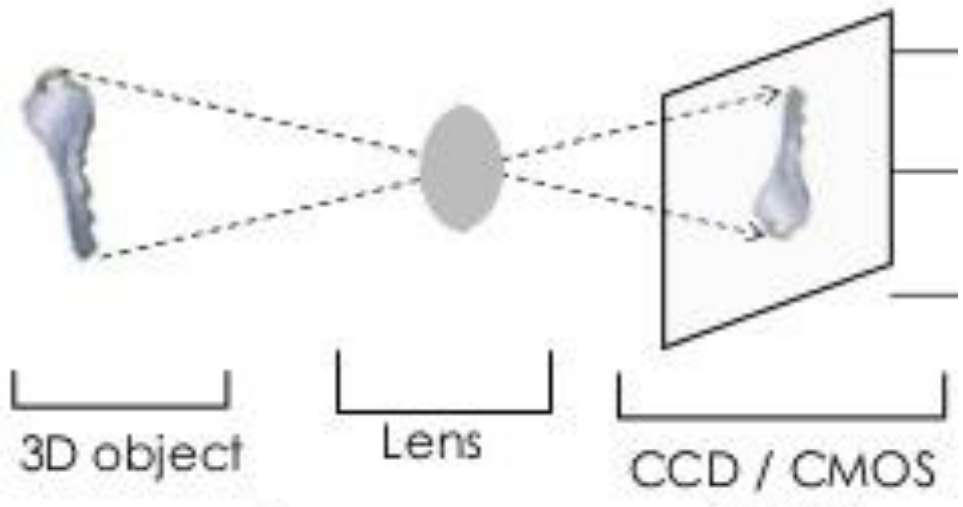
# Image formation

- What 3D world information is stored in each pixel?
- Image formation is a process that produces a particular image given:
  - A set of lighting conditions
  - **Scene geometry**
  - Surface properties
  - **Camera optics**

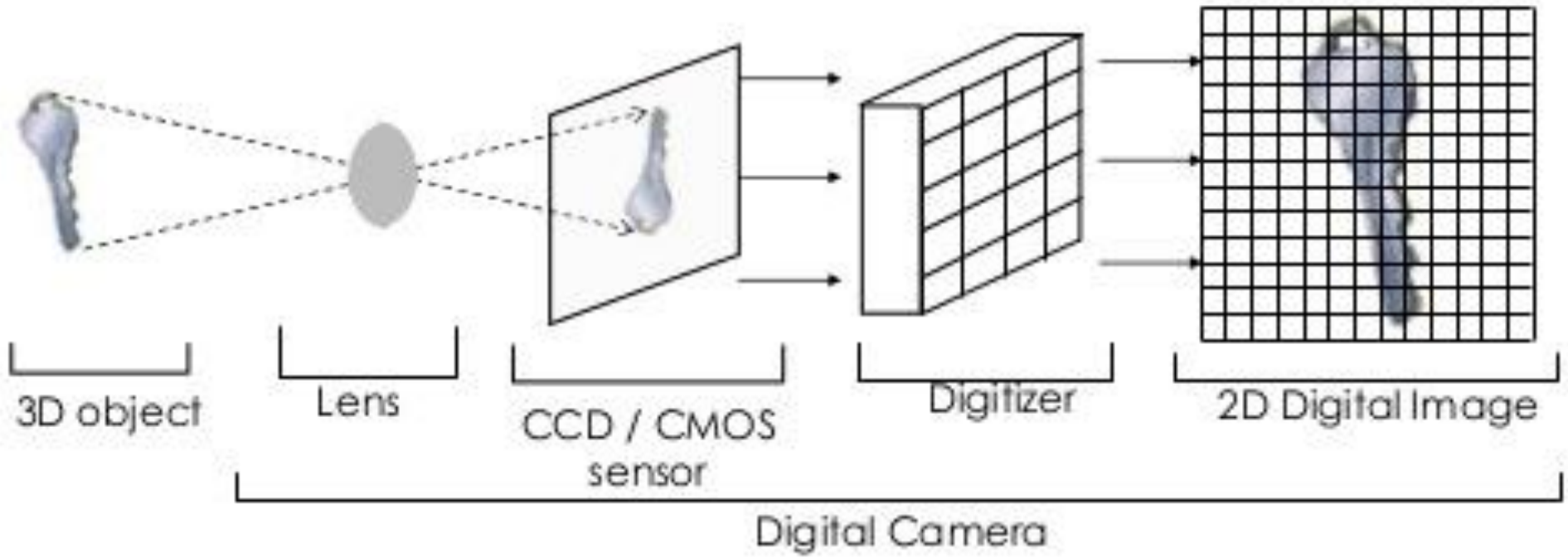
- Are parallel lines in the real world parallel in the image?
- Are straight lines in the real world straight in the image?

# Visual image formation-Digital Version

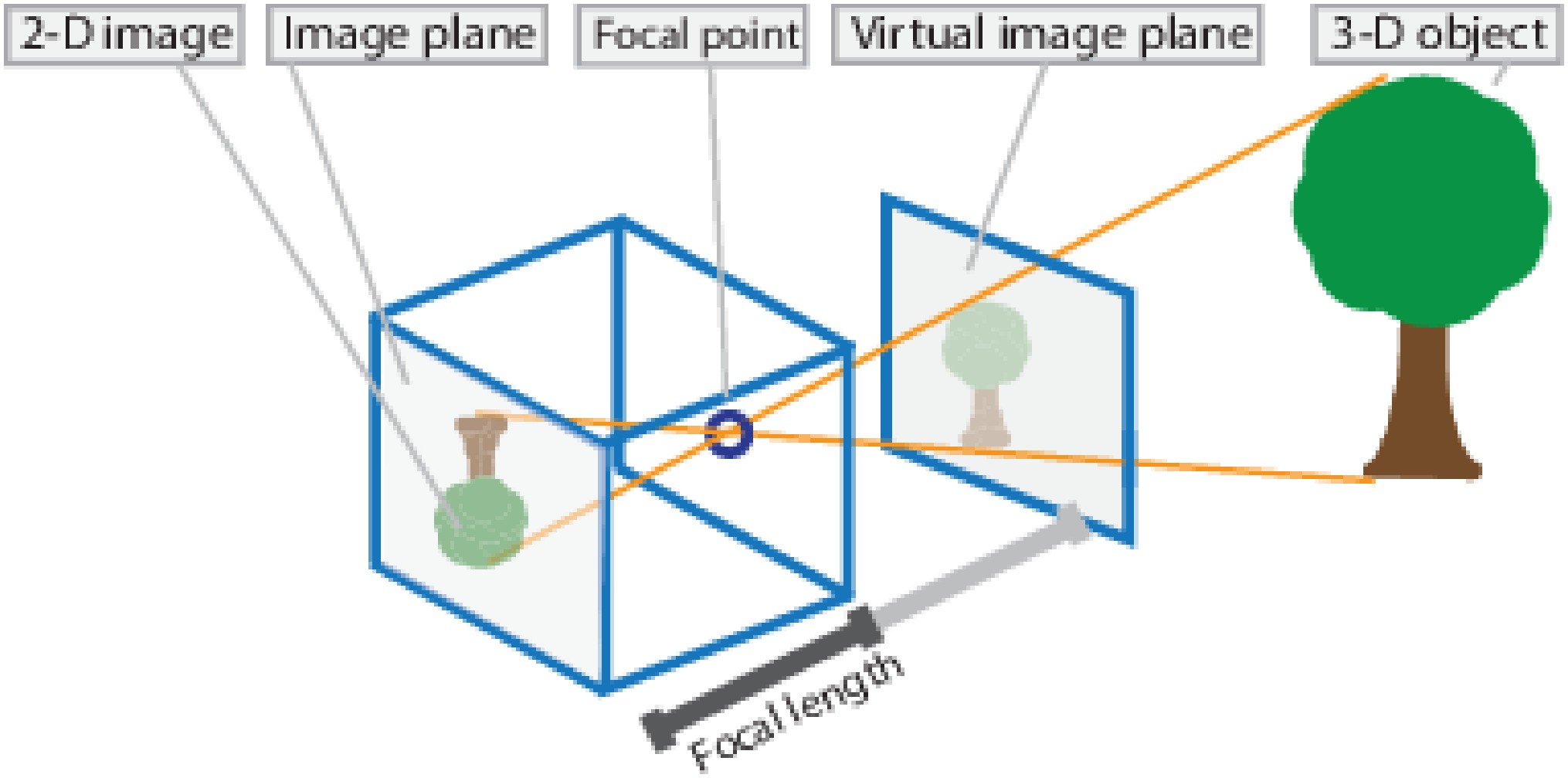
# Visual image formation-Digital Version



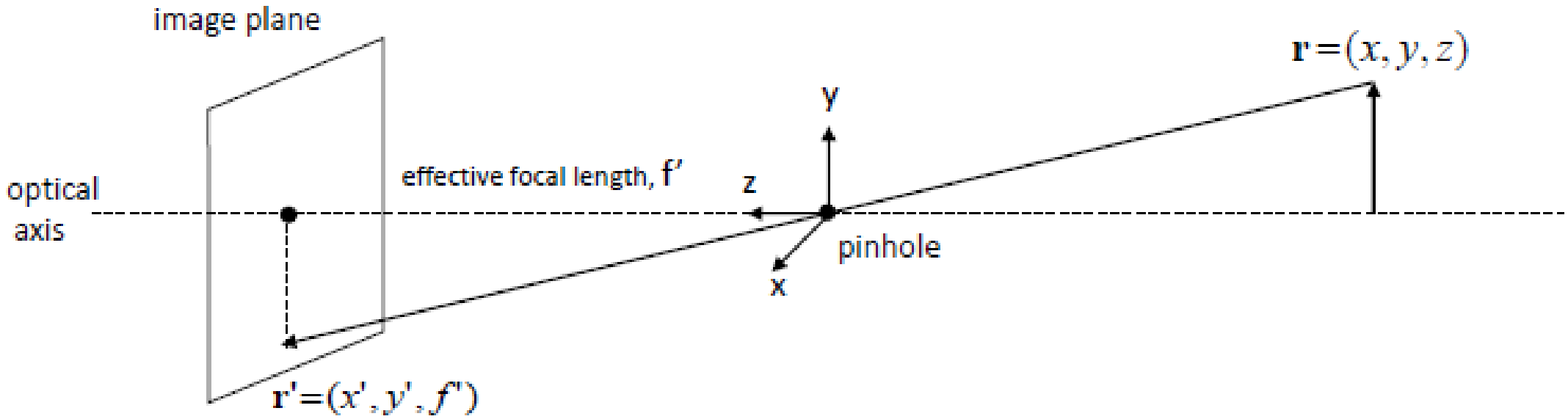
# Visual image formation-Digital Version



# Pinhole camera model

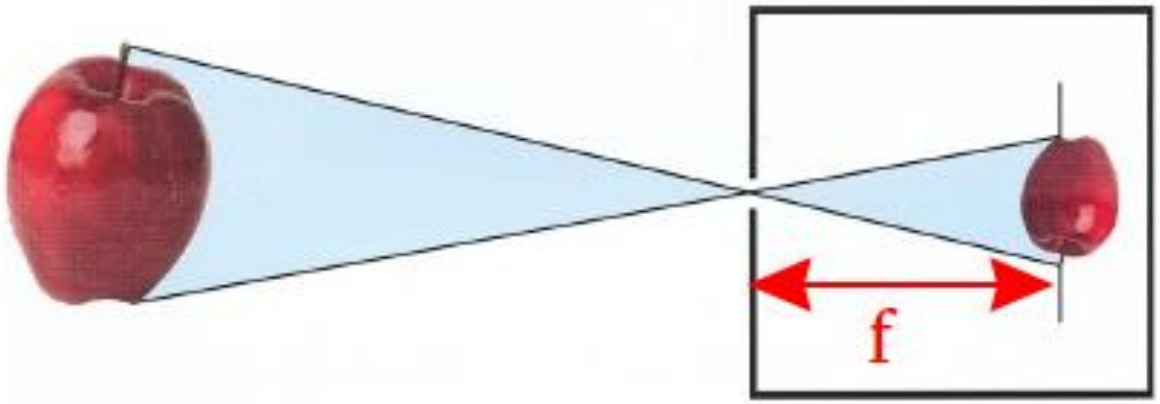


# Pinhole camera model

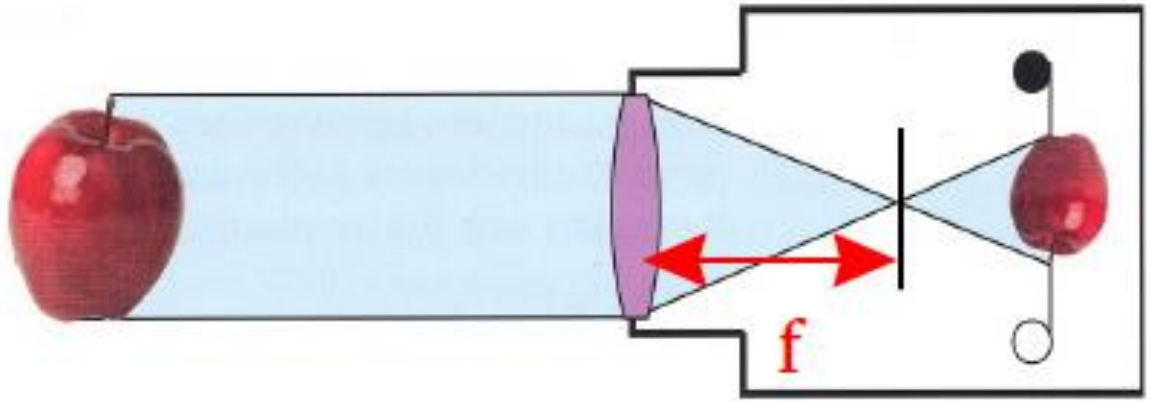



$$\frac{\mathbf{r}'}{f'}=\frac{\mathbf{r}}{z} \quad \Rightarrow \quad \frac{x'}{f'}=\frac{x}{z} \qquad \frac{y'}{f'}=\frac{y}{z}$$

# Pinhole vs. real camera model

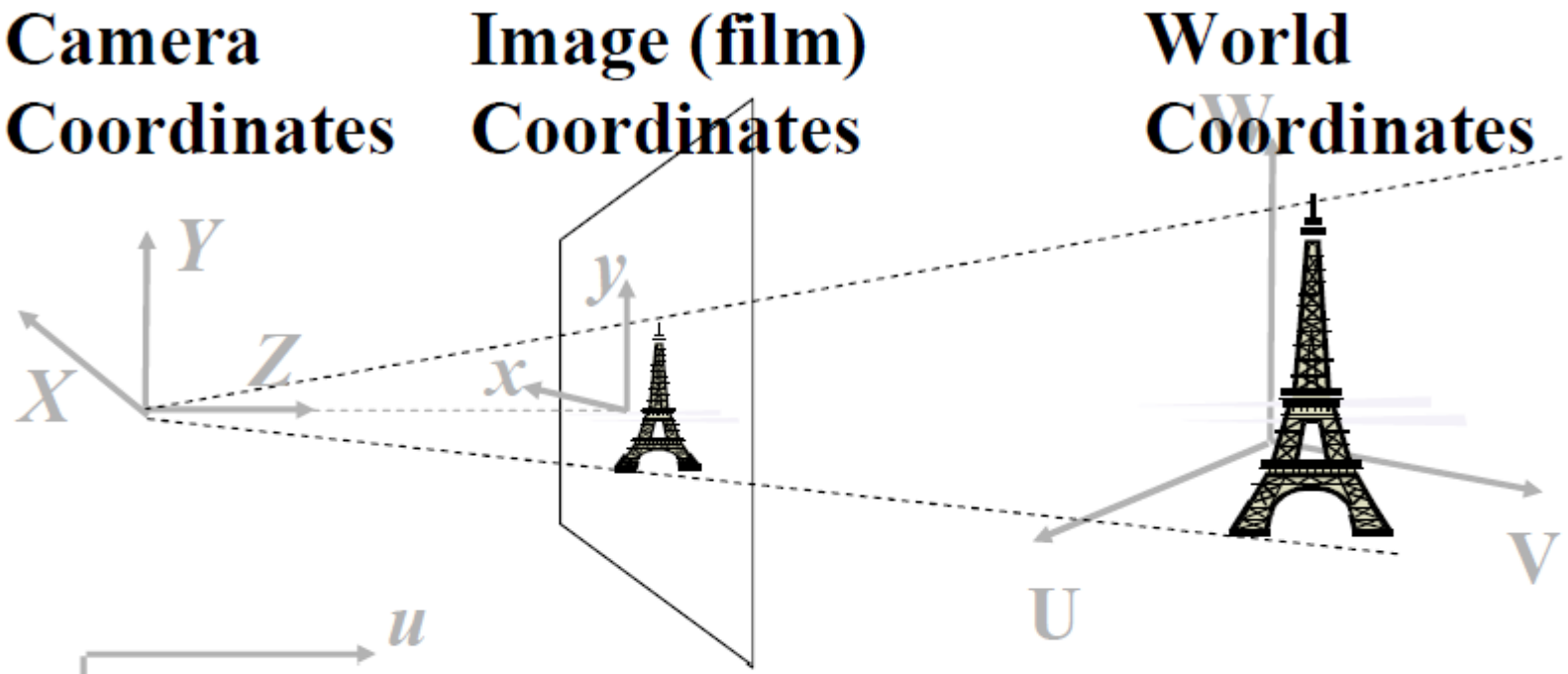


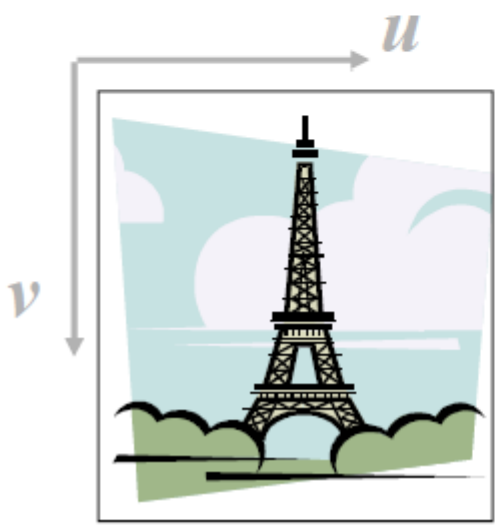

*Pinhole Camera*



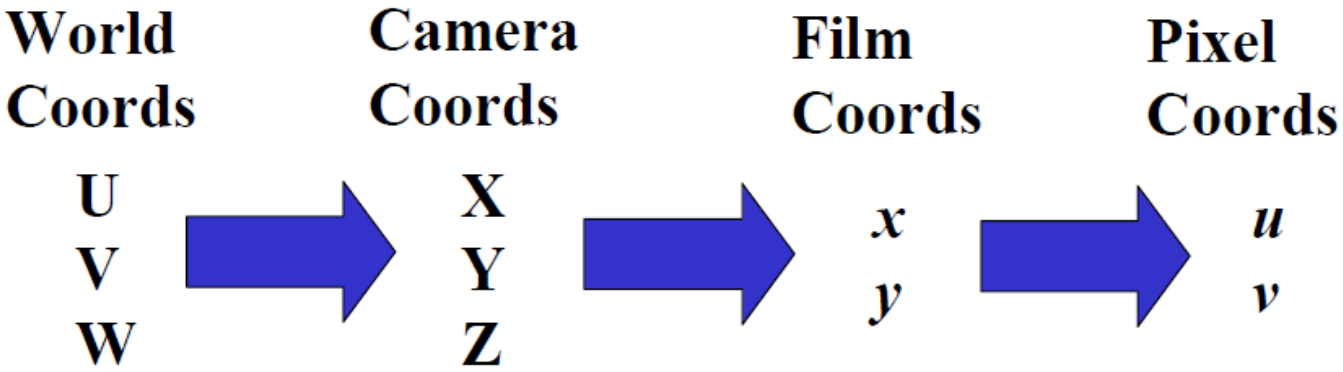

*Camera with lens to focus image.*

# 3D coordinates to pixel coordinates?

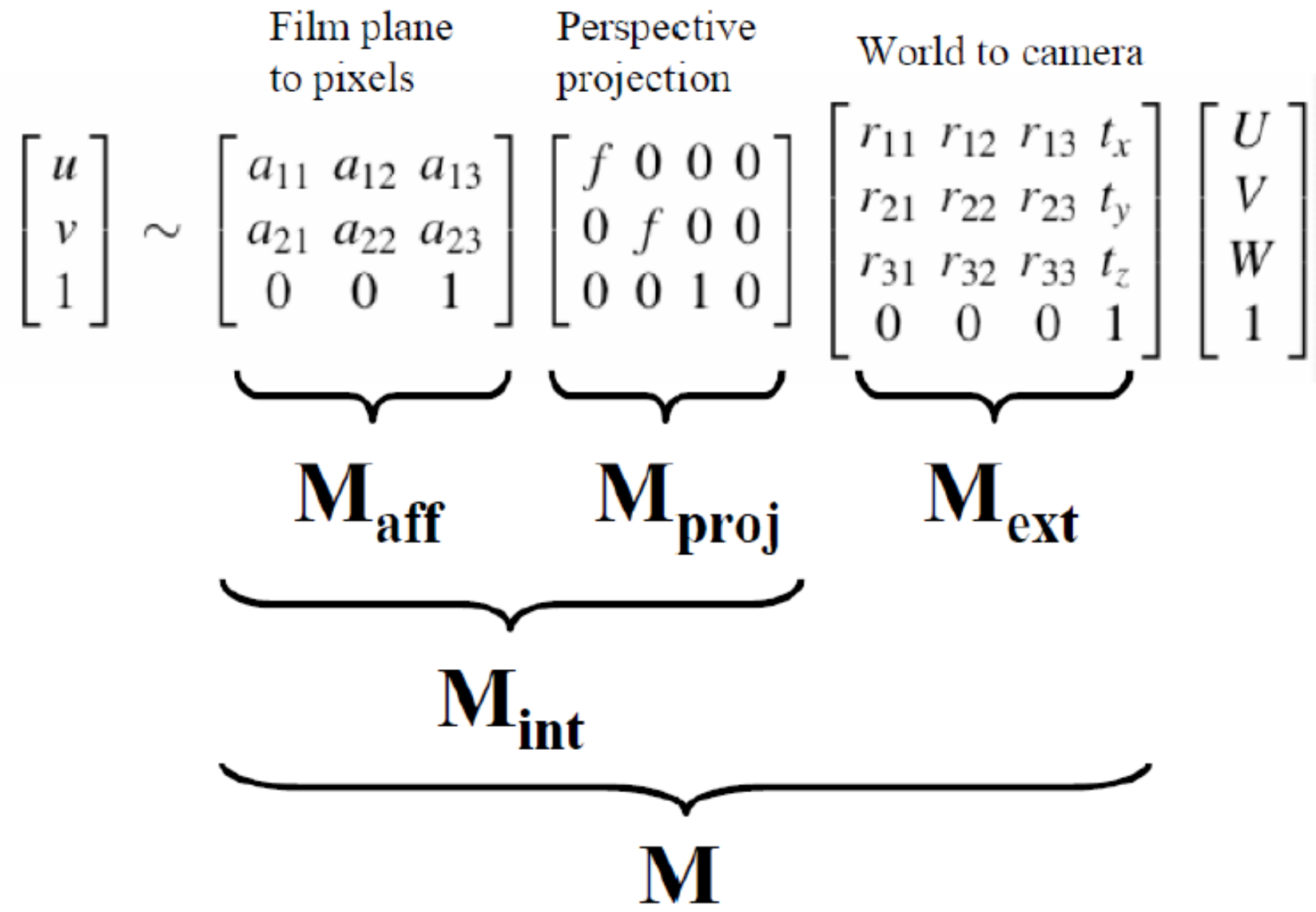


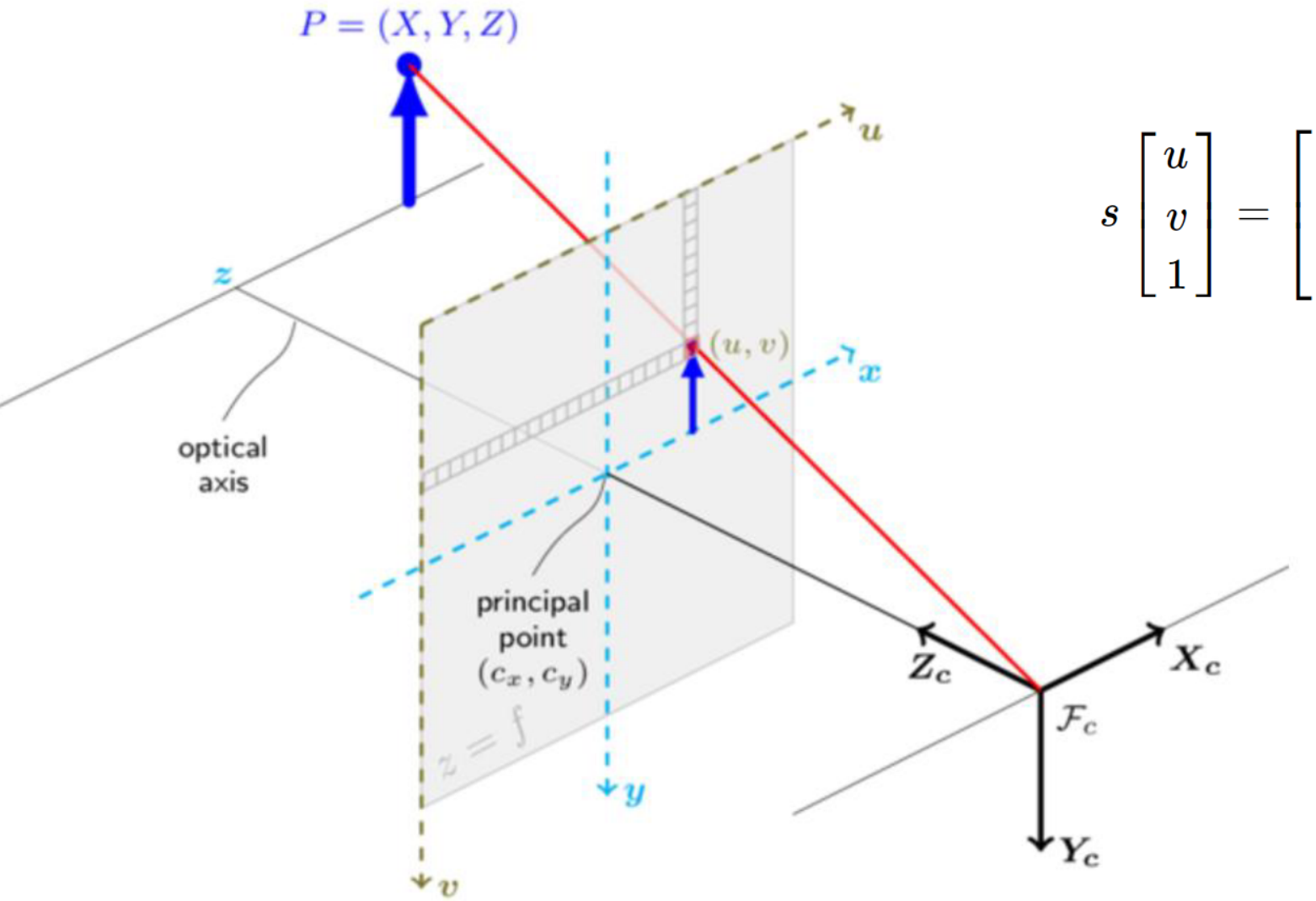




Pixel Coordinates



# Camera matrix






Pinhole camera model

Why two different focal lengths?

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$s\ m' = A[R|t]M'$$

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

- $(X, Y, Z)$ are the coordinates of a 3D point in the world coordinate space
- $(u, v)$ are the coordinates of the projection point in pixels
- $A$ is a camera matrix, or a matrix of intrinsic parameters
- $(cx, cy)$ is a principal point that is usually at the image center
- $fx, fy$ are the focal lengths expressed in pixel units.

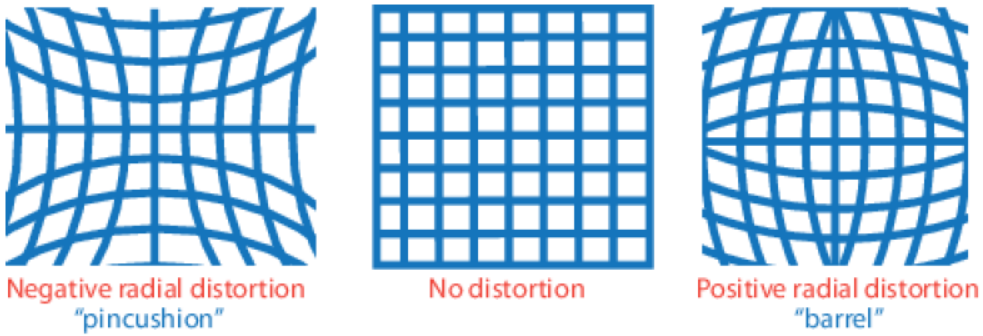
# Distortion

# Distortion example

# Distortion in camera calibration

- The camera matrix does not account for lens distortion because an ideal pinhole camera does not have a lens. To accurately represent a real camera, the camera model includes the radial and tangential lens distortion.
- Radial
- Tangential
- Prism

# Radial distortion




- Radial distortion occurs when light rays bend more near the edges of a lens than they do at its optical center. The smaller the lens, the greater the distortion.
- The distortion is 0 at the (optical) center of the imager and increases as we move toward the periphery
- If, for example, a camera has been calibrated on images of 320 x 240 resolution, absolutely the same distortion coefficients can be used for 640 x 480 images from the same camera **while fx, fy, cx, and cy need to be scaled appropriately**.

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

# Radial distortion

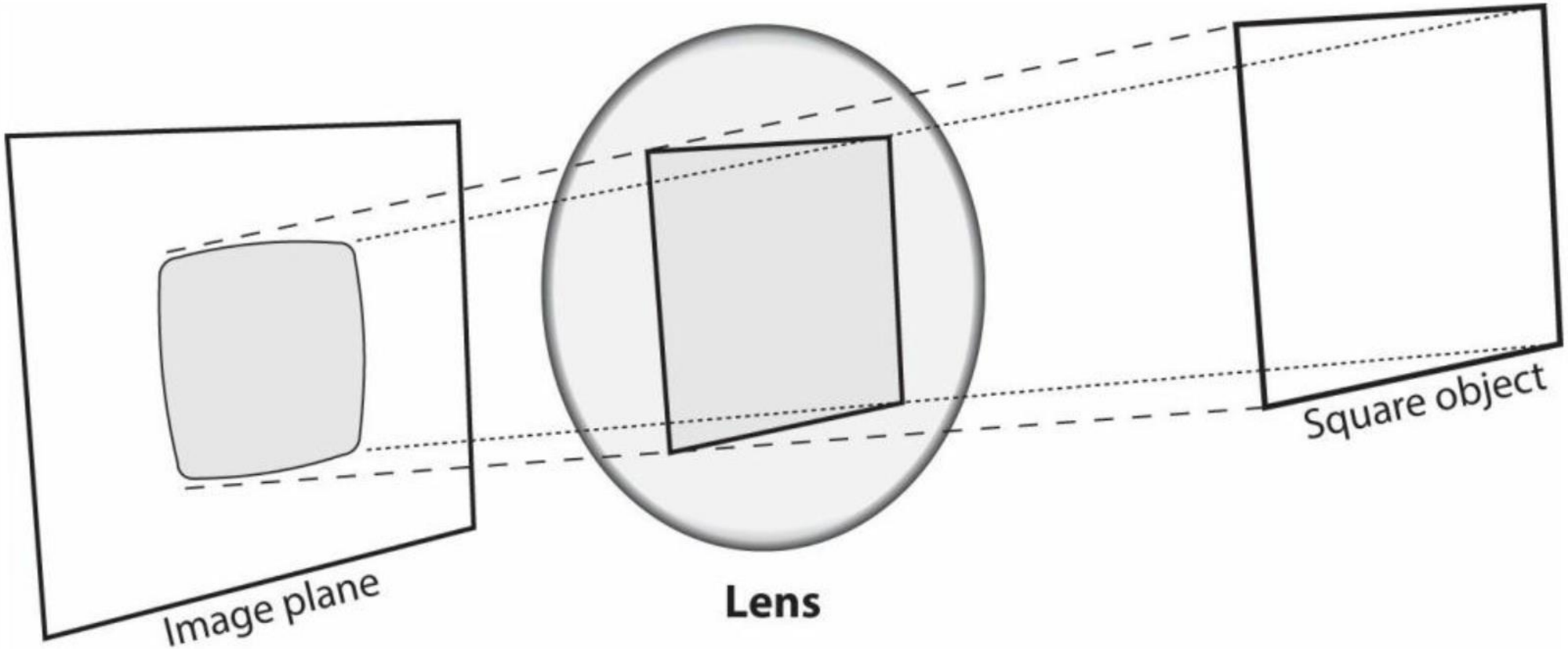
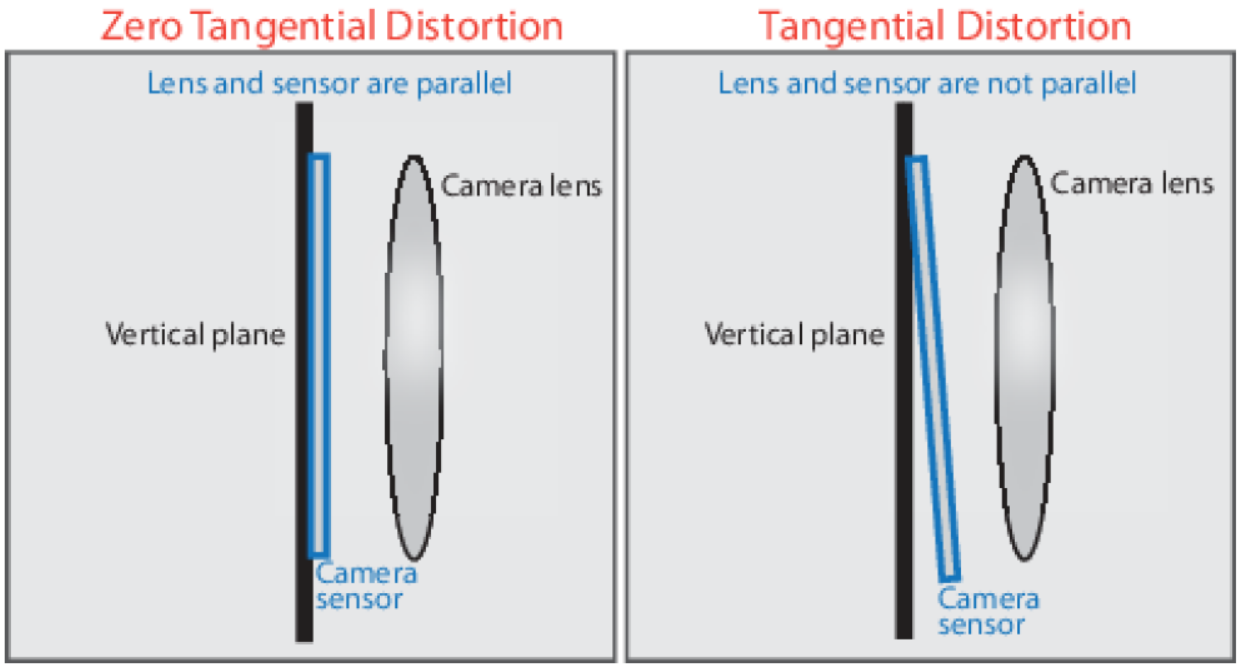



*Figure 18-3. Radial distortion: rays farther from the center of a simple lens are bent too much compared to rays that pass closer to the center; thus, the sides of a square appear to bow out on the image plane (this is also known as barrel distortion)*

# Tangential distortion

Tangential distortion occurs when the lens and the image plane are not parallel. The tangential distortion coefficients model this type of distortion.

$$x_{corrected} = x + [2p_1xy + p_2(r^2 + 2x^2)]$$
$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2xy]$$



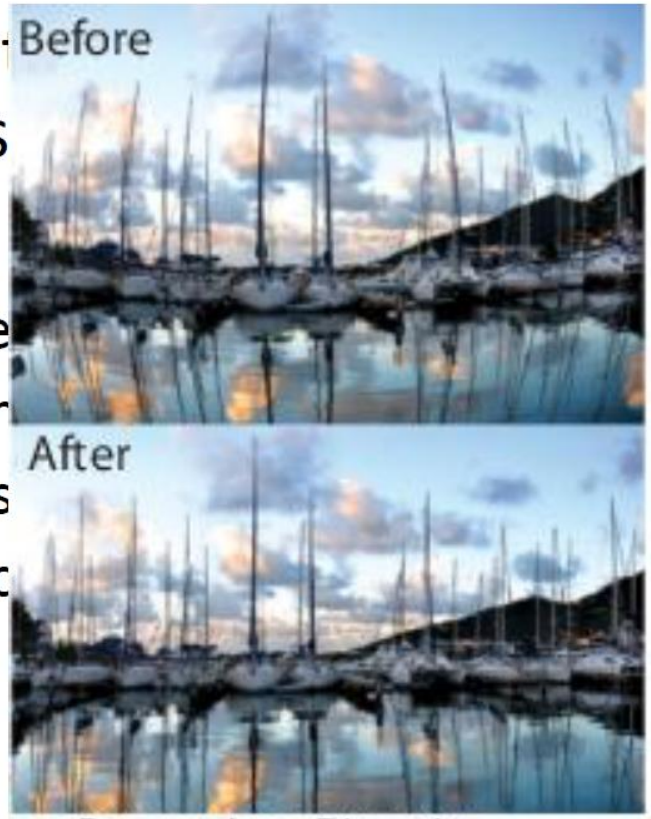
# Camera calibration

# Camera Calibration, Why?

- Camera calibration is a necessary step in 3D computer vision.
- A calibrated camera can be used as a **quantitative sensor**
- It is essential in many applications to recover 3D quantitative measures about the observed scene from 2D images. Such as 3D Euclidean structure.
- From a calibrated camera we can measure how far an object is from the camera, or the height of the object, etc. e.g., object avoidance in robot navigation.

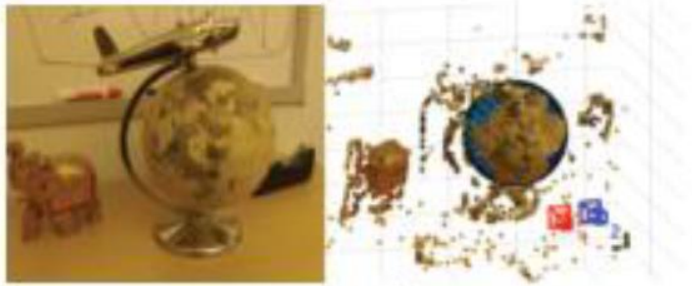
# Camera calibration

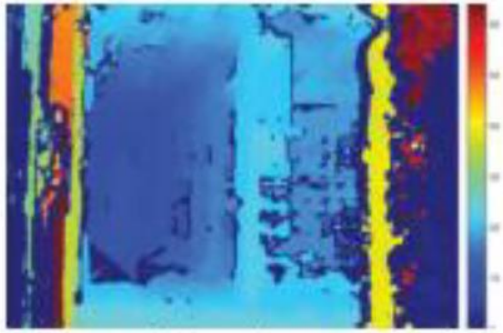
- Geome
  image s
- Typical
  - corre
  - estim
  - meas
  - and c



Remove Lens Distortion

lens and


Estimate 3-D Structure from Camera Motion


Estimate Depth
Using a Stereo Camera

Measure Planar Objects

# Camera model

- The calibration algorithm uses the camera model proposed by Jean-Yves Bouguet. The model includes:
  - The pinhole camera model
  - Lens distortion
- The pinhole camera model does not account for lens distortion because an ideal pinhole camera does not have a lens.
- To accurately represent a real camera, the full camera model used by the algorithm includes the radial and tangential lens distortion.

# Camera calibration

- Find the intrinsic and extrinsic parameters of a camera
  - **Extrinsic** parameters: the camera's location and orientation in the world.
  - **Intrinsic** parameters: the relationships between pixel coordinates and camera coordinates.
- Find the distortion coefficients of the camera
  - Radial
  - Tangential
- VERY large literature on the subject
- Good calibration is important when we need to:
  - Reconstruct a world model.
  - Interact with the world: Robot, hand-eye coordination

# Basic idea

1. Given a set of world points *Pi* and their image coordinates *(ui,vi)*
2. Find the projection matrix *M*
3. and then find intrinsic and extrinsic parameters.

- To estimate the camera parameters, we need to have 3-D world points and their corresponding 2-D image points.
- These correspondences are acquired using multiple images of a calibration pattern, such as a checkerboard.
- Using the correspondences, we can solve for the camera parameters.

# How many parameters are we looking for?

$$\begin{bmatrix} u \\ v \\ w \end{bmatrix} = K_{3\times3}\left({}^{C}_{W}R_{3\times3} \quad {}^{C}O_{W\,3\times1}\right)_{3\times4}\begin{pmatrix} {}^{W}P \\ 1 \end{pmatrix} \Rightarrow p = MP$$

*Intrinsic parameters*

*Rotation* *Translation*

*Extrinsic parameters*

*Projection matrix*
3x4

$$M = K_{3\times3}\left({}^{C}_{W}R_{3\times3} \quad {}^{C}O_{W\,3\times1}\right)_{3\times4} = K[r_1 \quad r_2 \quad r_3 \quad t]$$

Only 11 free parameters (not 12):
5 intrinsic, 3 for rotation, 3 for translation
*Perspective Projection Matrix*

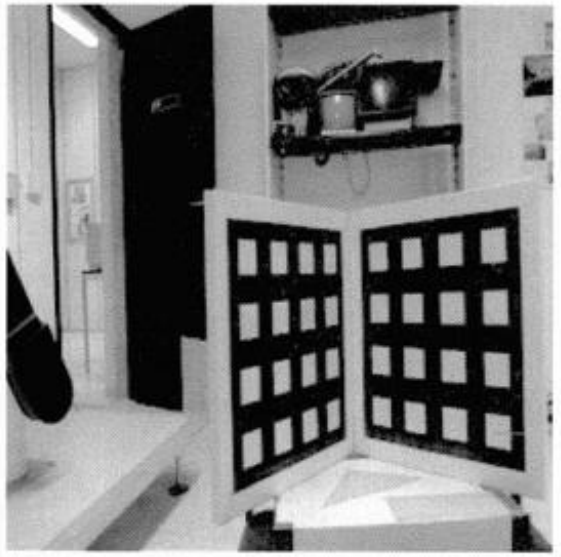
# Calibration Techniques

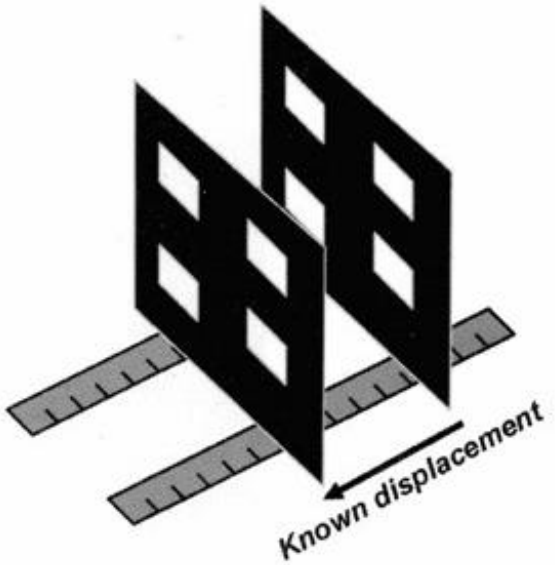
- Calibration using 3D calibration object
- Calibration using 2D planer pattern
- Calibration using 1D object (line-based calibration)
- Self Calibration: no calibration objects
- Vanishing points from for orthogonal direction
- Many other smart ideas

# Calibration Techniques

Calibration using 3D calibration object:

- Calibration is performed by observing a calibration object whose geometry in 3D space is known with very good precision.
- Calibration object usually consists of two or three planes orthogonal to each other, e.g. calibration cube
- Calibration can also be done with a plane undergoing a precisely known translation (Tsai approach)
- (+) most accurate calibration, simple theory
- (-) more expensive, more elaborate setup

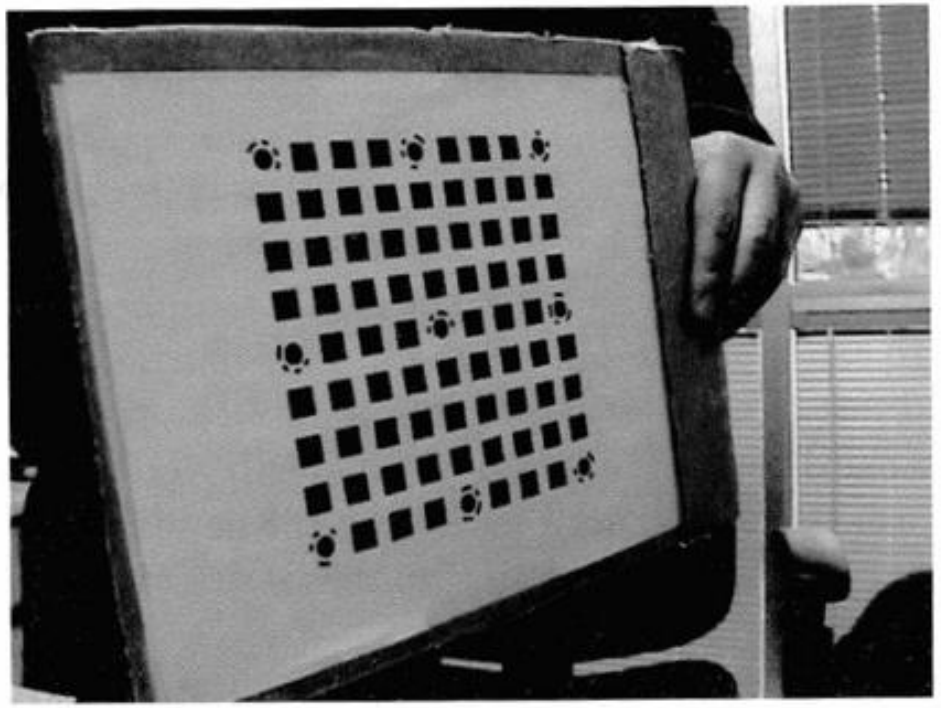




# Calibration Techniques

2D plane-based calibration

- Requires observation of a planar pattern shown at few different orientations
- No need to know the plane motion
- Set up is easy, the most popular approach.



# Calibration Techniques

Self-calibration:

- Techniques in this category do not use any calibration object
- Only image point correspondences are required
- Just move the camera in a static scene and obtain multiple images
- Correspondences between three images are sufficient to recover both the internal and external parameters

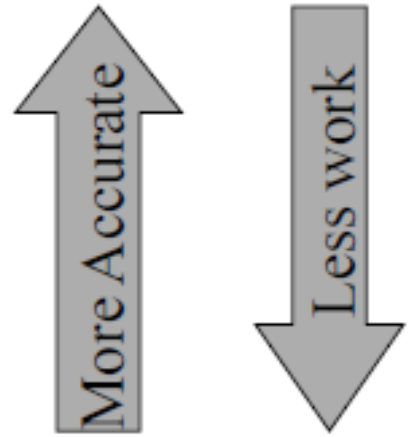
# Calibration Techniques

Which calibration technique to use ?

- Calibration with apparatus versus self-calibration:
- Whenever possible, if you can pre-calibrate the camera, you should do it with a calibration object
- Self-calibration cannot usually achieve the same accuracy as calibration with an object.
- Sometimes pre-calibration is impossible (e.g., a scene reconstruction from an old movie), self-calibration is the only choice.

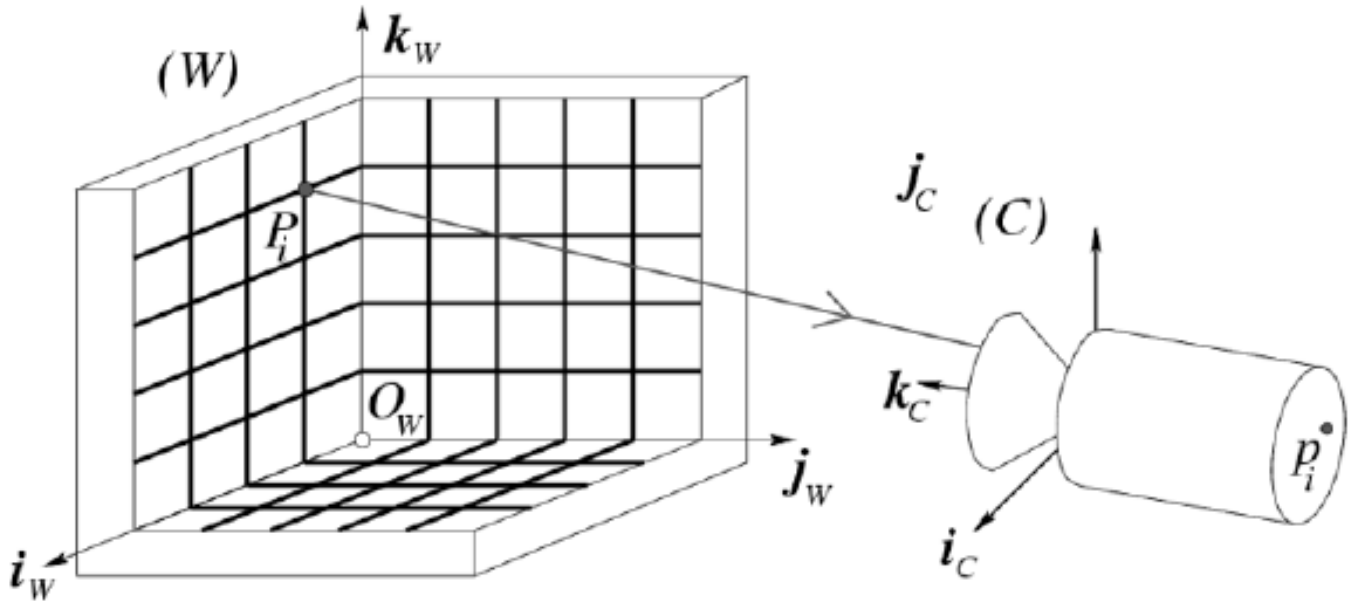
# Calibration Techniques

- Calibration using 3D calibration object
- Calibration using 2D planer pattern
- Calibration using 1D object (line-based calibration)
- Self Calibration: no calibration objects




- 2D planer pattern approaches seems to be a good compromise: **good accuracy with simple setup**
- 1D object is suitable for calibrating multiple cameras at once.

# Calibration Problem




Given $n$ points $P_1, \ldots, P_n$ with *known* positions and their images $p_1, \ldots, p_n$

Find $\boldsymbol{i}$ and $\boldsymbol{e}$ such that

$$\begin{cases} u_i = \dfrac{\boldsymbol{m}_1(\boldsymbol{i},\boldsymbol{e}) \cdot \boldsymbol{P}_i}{\boldsymbol{m}_3(\boldsymbol{i},\boldsymbol{e}) \cdot \boldsymbol{P}_i} \\ \\ v_i = \dfrac{\boldsymbol{m}_2(\boldsymbol{i},\boldsymbol{e}) \cdot \boldsymbol{P}_i}{\boldsymbol{m}_3(\boldsymbol{i},\boldsymbol{e}) \cdot \boldsymbol{P}_i} \end{cases} \quad \text{for} \quad i = 1, \ldots, n$$

*i* intrinsic parameters
*e* extrinsic parameters

$$\sum_{i=1}^{n} \left[ \left( u_i - \frac{\boldsymbol{m}_1(\boldsymbol{i},\boldsymbol{e}) \cdot \boldsymbol{P}_i}{\boldsymbol{m}_3(\boldsymbol{i},\boldsymbol{e}) \cdot \boldsymbol{P}_i} \right)^2 + \left( v_i - \frac{\boldsymbol{m}_2(\boldsymbol{i},\boldsymbol{e}) \cdot \boldsymbol{P}_i}{\boldsymbol{m}_3(\boldsymbol{i},\boldsymbol{e}) \cdot \boldsymbol{P}_i} \right)^2 \right] \quad \text{is minimized}$$

## Camera Calibration and least-squares

- Camera Calibration can be posed as least-squares parameter estimation problem.
- Estimate the intrinsic and extrinsic parameters that minimize the mean-squared deviation between predicted and observed image features.
- Least-squares parameter estimation is a fundamental technique that is used extensively in computer vision.
- You can formulate many problems as error minimization between observed and predicted values.
  - Linear Least-Squares
  - Nonlinear Least-Squares
- Typically, use linear least-squares to obtain a solution and then use nonlinear least square to refine the solution and estimate more parameters (e.g. radial distortion)

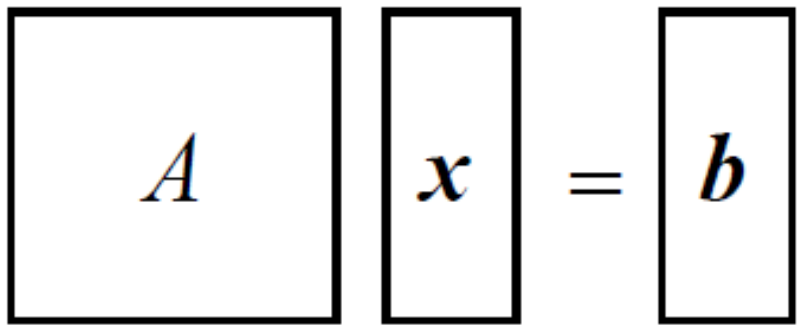
# Linear Least-Squares

- Linear system of equations

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \quad \cdots \quad + a_{1q}x_q &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \quad \cdots \quad + a_{2q}x_q &= b_2 \\ \cdots & \\ a_{p1}x_1 + a_{p2}x_2 + \quad \cdots \quad + a_{pq}x_q &= b_p \end{aligned} \quad \Leftrightarrow Ax = b$$

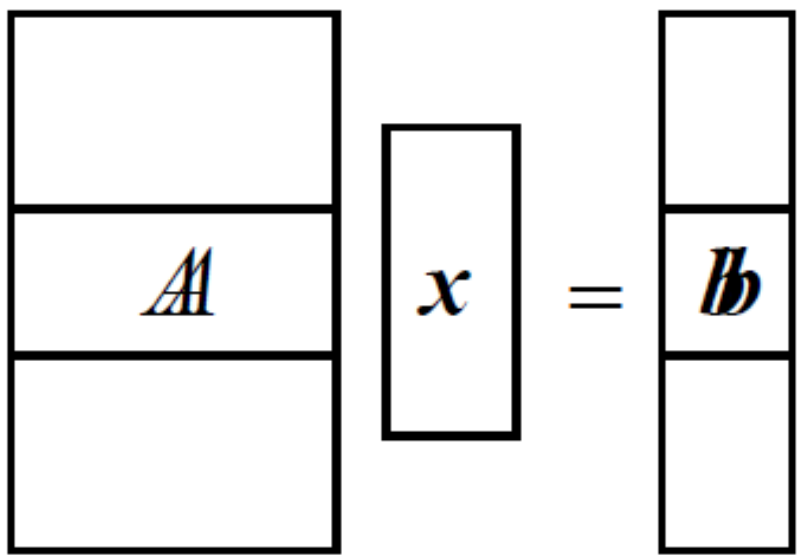
$$A_{p\times q} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1q} \\ a_{21} & a_{22} & \cdots & a_{2q} \\ \cdots & \cdots & \cdots & \cdots \\ a_{p1} & a_{p2} & \cdots & a_{pq} \end{pmatrix} \qquad x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_q \end{pmatrix} \qquad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix}$$

# Linear Systems



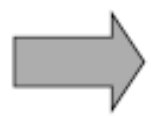

Square system:

- unique solution
- Gaussian elimination




Rectangular system ??

- under-constrained: infinity of solutions
- over-constrained: no solution



Minimize $||Ax-b||^2$

# Camera Calibration with a 2D planer object

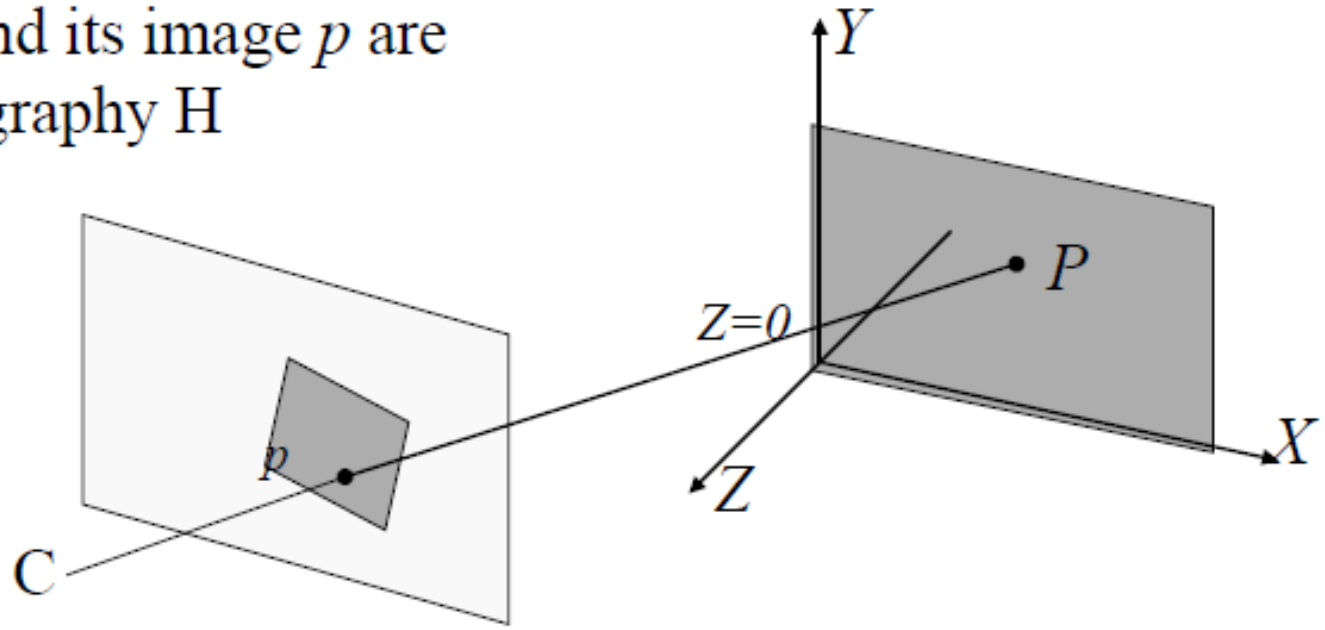
- Assume model plane is on Z=0 of the world coordinate system

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K\begin{bmatrix} r_1 & r_2 & r_3 & t \end{bmatrix}\begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix} = K\begin{bmatrix} r_1 & r_2 & t \end{bmatrix}\begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} \Rightarrow sp = \mathrm{H}_{3\mathrm{x}3}P$$

A model point $P$ and its image $p$ are related by a homography H

$$\mathrm{H} = K\begin{bmatrix} r_1 & r_2 & t \end{bmatrix}$$

H is a 3x3 matrix defined up to a scale factor (only 8 degrees of freedom)



- Given an image of the model plane, a homography can be estimated, Let's denoted by H=[$h_1$ $h_2$ $h_3$]

$$[\mathrm{h}_1 \quad \mathrm{h}_2 \quad \mathrm{h}_3] = \lambda K [r_1 \quad r_2 \quad t]$$

$$K^{-1}[\mathrm{h}_1 \quad \mathrm{h}_2 \quad \mathrm{h}_3] = \lambda [r_1 \quad r_2 \quad t]$$

$$K^{-1}\mathrm{h}_1 = \lambda r_1$$

$$K^{-1}\mathrm{h}_2 = \lambda r_2$$

Since $r_1$ and $r_2$ are orthonormal, we can obtain the following two important constraints relating $K$ and H

$$h_1^T K^{-T} K^{-1} h_2 = 0$$

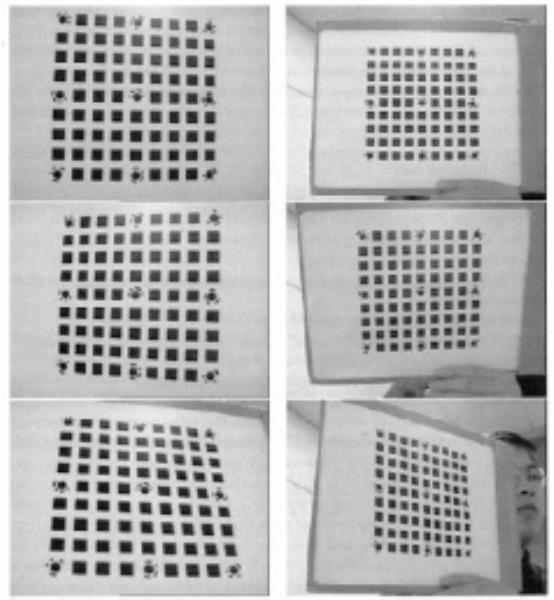
$$h_1^T K^{-T} K^{-1} h_1 = h_2^T K^{-T} K^{-1} h_2$$

- Given a set of images of the same plane we can establish a set of homographies $H_i$, one homography from each image.
- Given the constraints between H and $K$, we can solve for $K$

$$h_1^T K^{-T} K^{-1} h_2 = 0$$

$$h_1^T K^{-T} K^{-1} h_1 = h_2^T K^{-T} K^{-1} h_2$$

- Note $K^{-T}K^{-1}$ is a 3x3 symmetric matrix; we have six unknowns.
- Each homography gives two linear equations in 6 unknowns
- We need at least 3 images of the plane to estimate the parameters in $K$
- Once K is known, we can estimate the extrinsic parameters:



$$r_1 = \lambda K^{-1}\mathrm{h}_1, \qquad r_2 = \lambda K^{-1}\mathrm{h}_2, \qquad r_3 = r_1 \times r_2, \qquad t = \lambda K^{-1}\mathrm{h}_3$$

# Calibration

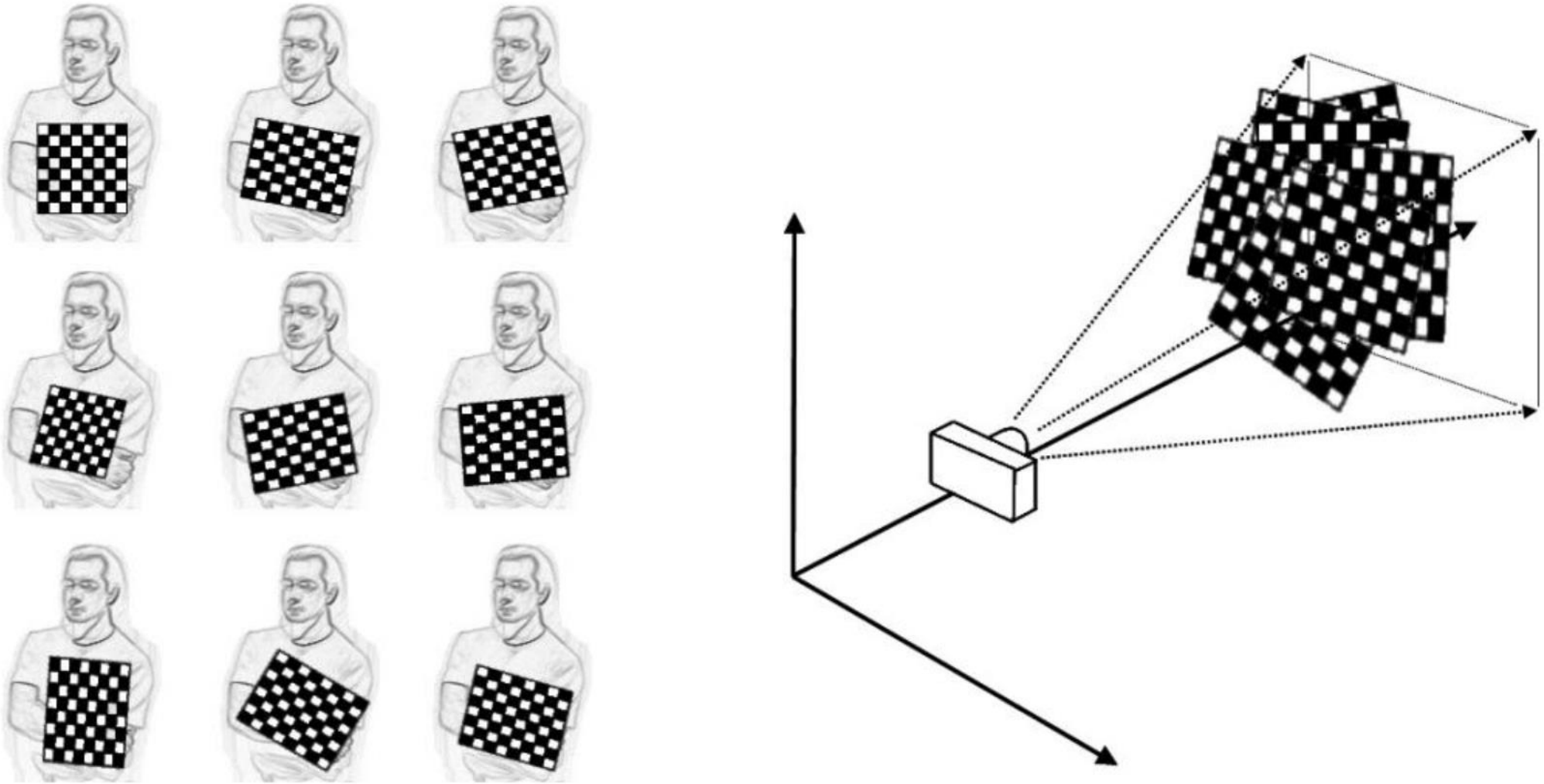


*Figure 18-10. Images of a chessboard being held at various orientations (left) provide enough information to completely solve for the locations of those images in global coordinates (relative to the camera) and the camera intrinsics*

# Example 3 – perform camera calibration

https://docs.opencv.org/3.2.0/d4/d94/tutorial_camera_calibration.html

- Create New Project (New Project -> Win32 console application -> Next -> Empty project -> Finish)
- Add OpenCV Property pages
- Add `camera_calibration_with_debug.cpp` to Source files
- Adjust the path to the config file **in_VID5.xml**

```
246  int main(int argc, char* argv[])
247  {
248      help();
249      Settings s;
250      const string inputSettingsFile = argc > 1 ? argv[1] : "../../kod/in_VID5.xml";
251      FileStorage fs(inputSettingsFile, FileStorage::READ); // Read the settings
```

- Open the config file

```
<Settings>
  <!-- Number of inner corners per a item row and column. (square, circle) -->
  <BoardSize_Width> 14</BoardSize_Width>
  <BoardSize_Height>9</BoardSize_Height>

  <!-- The size of a square in some user defined metric system (pixel, millimeter)-->
  <Square_Size>16</Square_Size>

  <!-- The type of input used for camera calibration. One of: CHESSBOARD CIRCLES_GRID A
  <Calibrate_Pattern>"CHESSBOARD"</Calibrate_Pattern>

  <!-- The input to use for calibration.
       To use an input camera -> give the ID of the camera, like "1"
       To use an input video  -> give the path of the input video, like "/tmp/x.avi"
       To use an image list   -> give the path to the XML or YAML file containing the
       -->
  <!-- <Input>"../VID5.xml"</Input> -->
  <Input>"0"</Input>
  <!--  If true (non-zero) we flip the input images around the horizontal axis.-->
  <Input_FlipAroundHorizontalAxis>0</Input_FlipAroundHorizontalAxis>

  <!-- Time delay between frames in case of camera. -->
  <Input_Delay>500</Input_Delay>

  <!-- How many frames to use, for calibration. -->
  <Calibrate_NrOfFrameToUse>10</Calibrate_NrOfFrameToUse>
  <!-- Consider only fy as a free parameter, the ratio fx/fy stays the same as in the i
       Use or not setting. 0 - False Non-Zero - True-->
```

Size of the board in numbers of chessboard corners, NOT SQUARES

Camera ID – enumeration starts from zero

```
<Calibrate_FixAspectRatio> 1 </Calibrate_FixAspectRatio>
<!-- If true (non-zero) tangential distortion coefficients  are set to zeros and stay z
<Calibrate_AssumeZeroTangentialDistortion>1</Calibrate_AssumeZeroTangentialDistortion>
<!-- If true (non-zero) the principal point is not changed during the global optimizati
<Calibrate_FixPrincipalPointAtTheCenter> 1 </Calibrate_FixPrincipalPointAtTheCenter>

<!-- The name of the output log file. -->
<Write_outputFileName>"out_camera_data.xml"</Write_outputFileName>
<!-- If true (non-zero) we write to the output file the feature points.-->
<Write_DetectedFeaturePoints>1</Write_DetectedFeaturePoints>
<!-- If true (non-zero) we write to the output file the extrinsic camera parameters.-->
<Write_extrinsicParameters>1</Write_extrinsicParameters>
<!-- If true (non-zero) we show after calibration the undistorted images.-->
<Show_UndistortedImage>1</Show_UndistortedImage>
```

# Example 3 - continued

- Change config file to fit your chessboard chart and the source you are using (images, video file, camera)
- Build and run!

- What are the steps in the algorithm?

# Algorithm

- read Settings
- for (i=0;;i++)
  - acquire next frame/image
  - check if we have enough images already processed?
  - NO
    - examine the image and detect chessboard corners
    - save chessboard corners into ImagePoints
  - YES
    - perform camera calibration using cv::calibrateCamera()
    - using the newly generated camera matrix and distortion matrix and rvecs and tvecs display undistorted images
    - undistort() is used for remapping from distorted to undistorted
- end

# OpenCV calib3d module

- OpenCV provides several algorithms to help us compute these intrinsic parameters.
- The actual calibration is done via cv: : **calibrateCamera**( ).
- In this routine, the method of calibration is to target the camera on a known structure that has many individual and identifiable points. By viewing this structure from a variety of angles, we can then compute the (relative) location and orientation of the camera at the time of each image as well as the intrinsic parameters of the camera
- To provide multiple views, we rotate and translate the object

# cv::calibrateCamera()

```
double cv::calibrateCamera    (        InputArrayOfArrays          objectPoints,
                                       InputArrayOfArrays          imagePoints,
                                       Size                        imageSize,
                                       InputOutputArray            cameraMatrix,
                                       InputOutputArray            distCoeffs,
                                       OutputArrayOfArrays         rvecs,
                                       OutputArrayOfArrays         tvecs,
                                       OutputArray                 stdDeviationsIntrinsics,
                                       OutputArray                 stdDeviationsExtrinsics,
                                       OutputArray                 perViewErrors,
                                       int                         flags = 0,
                                       TermCriteria                criteria =
TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, DBL_EPSILON)
)
```

# cv::calibrateCamera()

- The algorithm performs the following steps:
  - Compute the initial intrinsic parameters (the option only available for planar calibration patterns) or read them from the input parameters. The distortion coefficients are all set to zeros initially unless some of CV_CALIB_FIX_K? are specified.
  - Estimate the initial camera pose as if the intrinsic parameters have been already known. This is done using solvePnP .
  - Run the global Levenberg-Marquardt optimization algorithm to minimize the reprojection error, that is, the total sum of squared distances between the observed feature points imagePoints and the projected (using the current estimates for camera parameters and the poses) object points objectPoints.
- The function returns the final re-projection error.

# cv::undistort()

- The function transforms an image to compensate radial and tangential lens distortion.
- The function is simply a combination of cv::**initUndistortRectifyMap** and cv::**remap** (with bilinear interpolation).
- Those pixels in the destination image, for which there is no correspondent pixels in the source image, are filled with zeros (black color).
- A particular subset of the source image that will be visible in the corrected image can be regulated by newCameraMatrix. You can use cv::**getOptimalNewCameraMatrix** to compute the appropriate newCameraMatrix depending on your requirements.
- The camera matrix and the distortion parameters can be determined using cv::calibrateCamera. If the resolution of images is different from the resolution used at the calibration stage, fx, fy, cx and cy need to be scaled accordingly, while the distortion coefficients remain the same.

# initUndistortRectifyMap()

- Takes calculated camera matrix, dist coeffs, rvec and tvecs and calculates the mapping between the wanted undistorted pixel and original distorted pixel.
- This is necessary for backward warping of the image.
- Beside the camera matrix, we provide new camera matrix we want if we changed the alpha in cv::**getOptimalNewCameraMatrix**
- Generated maps are then input for remap function which performs the actual warping

# initUndistortRectifyMap() - algorithm

$$x \leftarrow (u - {c'}_x)/{f'}_x$$
$$y \leftarrow (v - {c'}_y)/{f'}_y$$
$$[X\,Y\,W]^T \leftarrow R^{-1} * [x\,y\,1]^T$$
$$x' \leftarrow X/W$$
$$y' \leftarrow Y/W$$
$$r^2 \leftarrow x'^2 + y'^2$$
$$x'' \leftarrow x' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + 2p_1 x' y' + p_2(r^2+2x'^2) + s_1 r^2 + s_2 r^4$$
$$y'' \leftarrow y' \frac{1+k_1 r^2+k_2 r^4+k_3 r^6}{1+k_4 r^2+k_5 r^4+k_6 r^6} + p_1(r^2+2y'^2) + 2p_2 x' y' + s_3 r^2 + s_4 r^4$$
$$s\begin{bmatrix} x''' \\ y''' \\ 1 \end{bmatrix} = \begin{bmatrix} R_{33}(\tau_x,\tau_y) & 0 & -R_{13}((\tau_x,\tau_y) \\ 0 & R_{33}(\tau_x,\tau_y) & -R_{23}(\tau_x,\tau_y) \\ 0 & 0 & 1 \end{bmatrix} R(\tau_x,\tau_y) \begin{bmatrix} x'' \\ y'' \\ 1 \end{bmatrix}$$
$$map_x(u,v) \leftarrow x''' f_x + c_x$$
$$map_y(u,v) \leftarrow y''' f_y + c_y$$

# cv::getOptimalNewCameraMatrix

- The function computes and returns the optimal new camera matrix based on the free scaling parameter.
- By varying this parameter, you may retrieve only sensible pixels alpha=0 , keep all the original image pixels if there is valuable information in the corners alpha=1 , or get something in between.
- When alpha>0 , the undistortion result is likely to have some black pixels corresponding to "virtual" pixels outside of the captured distorted image.
- The original camera matrix, distortion coefficients, the computed new camera matrix, and newImageSize should be passed to initUndistortRectifyMap to produce the maps for remap .

# computeReprojectionErrors

```
408
409  static double computeReprojectionErrors( const vector<vector<Point3f> >& objectPoints,
410                                           const vector<vector<Point2f> >& imagePoints,
411                                           const vector<Mat>& rvecs, const vector<Mat>& tvecs,
412                                           const Mat& cameraMatrix , const Mat& distCoeffs,
413                                           vector<float>& perViewErrors)
414  {
415      vector<Point2f> imagePoints2;
416      int i, totalPoints = 0;
417      double totalErr = 0, err;
418      perViewErrors.resize(objectPoints.size());
419
420      for( i = 0; i < (int)objectPoints.size(); ++i )
421      {
422          projectPoints( Mat(objectPoints[i]), rvecs[i], tvecs[i], cameraMatrix,
423                         distCoeffs, imagePoints2);
424          err = norm(Mat(imagePoints[i]), Mat(imagePoints2), CV_L2);
425
426          int n = (int)objectPoints[i].size();
427          perViewErrors[i] = (float) std::sqrt(err*err/n);
428          totalErr        += err*err;
429          totalPoints     += n;
430      }
431
432      return std::sqrt(totalErr/totalPoints);
433  }
```

# Example 3 – look at the results

- Open out_camera_data
- What can we conclude from intrinsic matric?
  - cx, cy = ?
  - what did we expect?
  - focal length?
- How do the parameters change with resolution?
  - inputCapture.set(CV_CAP_PROP_FRAME_WIDTH, 1280);
  - inputCapture.set(CV_CAP_PROP_FRAME_HEIGHT, 720);
- How big are distortion coefficients?

# Camera spec

- The C270 has a pixel size of 2.8 µm square, and a resolution of 1280x720, so the sensor size is 3.58 mm x 2.02 mm.
- http://support.logitech.com/en_us/article/17556

# References


[1] http://docs.opencv.org/master/– OpenCV 3.2.0-dev Documentation

[2] Adrian Kaehler and Gary Bradski. 2016. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library* (1st ed.). O'Reilly Media, Inc..

[3] Kenneth Dawson-Howe. 2014. *A Practical Introduction to Computer Vision with OpenCV* (1st ed.). Wiley Publishing.

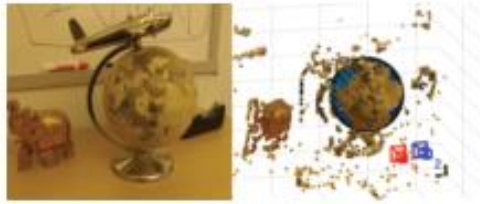
[4] Zhang, Z. "A Flexible New Technique for Camera Calibration." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 22, No. 11, 2000, pp. 1330–1334.

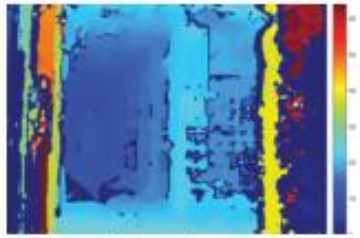
[5] https://www.mathworks.com/help/vision/ug/camera-calibration.html

[6] http://tnt.etf.bg.ac.rs/~oe4dos/Predavanja/OE4DOS%20-%20Poboljsanje%20kvaliteta%20slike%20-%20prostorne%20operacije.pdf

- Geometric camera calib
resectioning, estimates ensor of an image or video came
- You can use these parar neasure the size of an object in of the camera in the scene.
- These tasks are used in applications such as machine vision to detect and measure objects. They are also used in robotics, for navigation systems, and 3-D scene reconstruction.
- Examples of what you can do after calibrating your camera:

Remove Lens Distortion

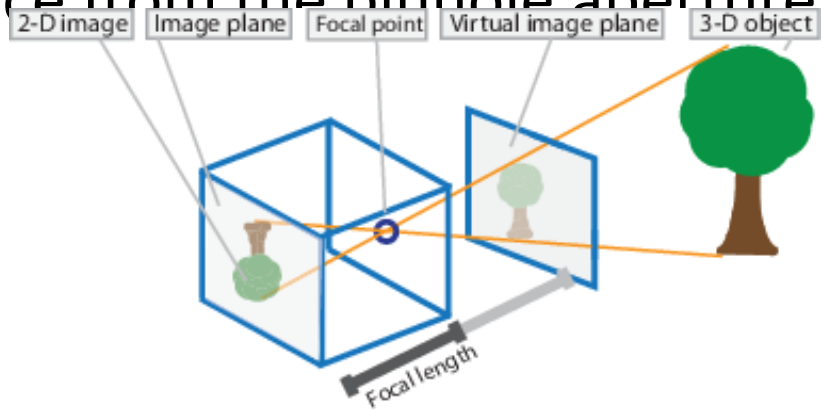

Estimate 3-D Structure from Camera Motion


Estimate Depth
Using a Stereo Camera

Measure Planar Objects

# Pinhole camera model

- A pinhole camera is a simple camera without a lens and with a single small aperture. Light rays pass through the aperture and project an inverted image on the opposite side of the camera. Think of the virtual image plane as being in front of the camera and containing the upright image of the scene.
- As a result, the image on this *image plane* (also called the *projective plane*) is always in focus, and the size ofthe image relative to the distant object is given by a single parameter ofthe camera: its *focal length*. For our idealized pinhole camera, the distance from the pinhole aperture to the screen is precisely the focal length



# Reference


[1] http://docs.opencv.org/master/– OpenCV 3.2.0-dev Documentation

[2] Adrian Kaehler and Gary Bradski. 2016. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library* (1st ed.). O'Reilly Media, Inc..

[3] https://www.cs.rutgers.edu/~elgammal/classes/cs534/lectures/Calibration.pdf

[4] Kenneth Dawson-Howe. 2014. *A Practical Introduction to Computer Vision with OpenCV* (1st ed.). Wiley Publishing.

[5] Zhang, Z. "A Flexible New Technique for Camera Calibration." *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Vol. 22, No. 11, 2000, pp. 1330–1334.

[6] https://www.mathworks.com/help/vision/ug/camera-calibration.html

[7] http://tnt.etf.bg.ac.rs/~oe4dos/Predavanja/OE4DOS%20-%20Poboljsanje%20kvaliteta%20slike%20-%20prostorne%20operacije.pdf