ELEG4701: Intelligent Interactive Robot Practice

Lecture 9: Visual Servoing for Mobile Robots

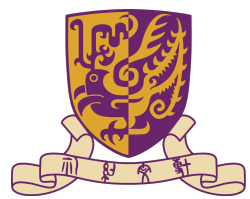
Jiewen Lai

Research Assistant Professor

jwlai@ee.cuhk.edu.hk

EE, CUHK

2024 Spring



Outline

Part 1. What is Visual Servoing?

What is Visual Servoing?

Visual Servoing is also known as **Vision-based Robot Control**

- It uses feedback information extrated from a vision sensor (e.g., camera) to control the robot's motion.

Visual servoing was first proposed by *Gerald* in 1979, referred to "Real Time Control of A Robot with a Mobile Camera."

Part 2. Introduction to Pinhole Camera Model

Introduction to Pinhole Camera Model

Pinhole imaging principle

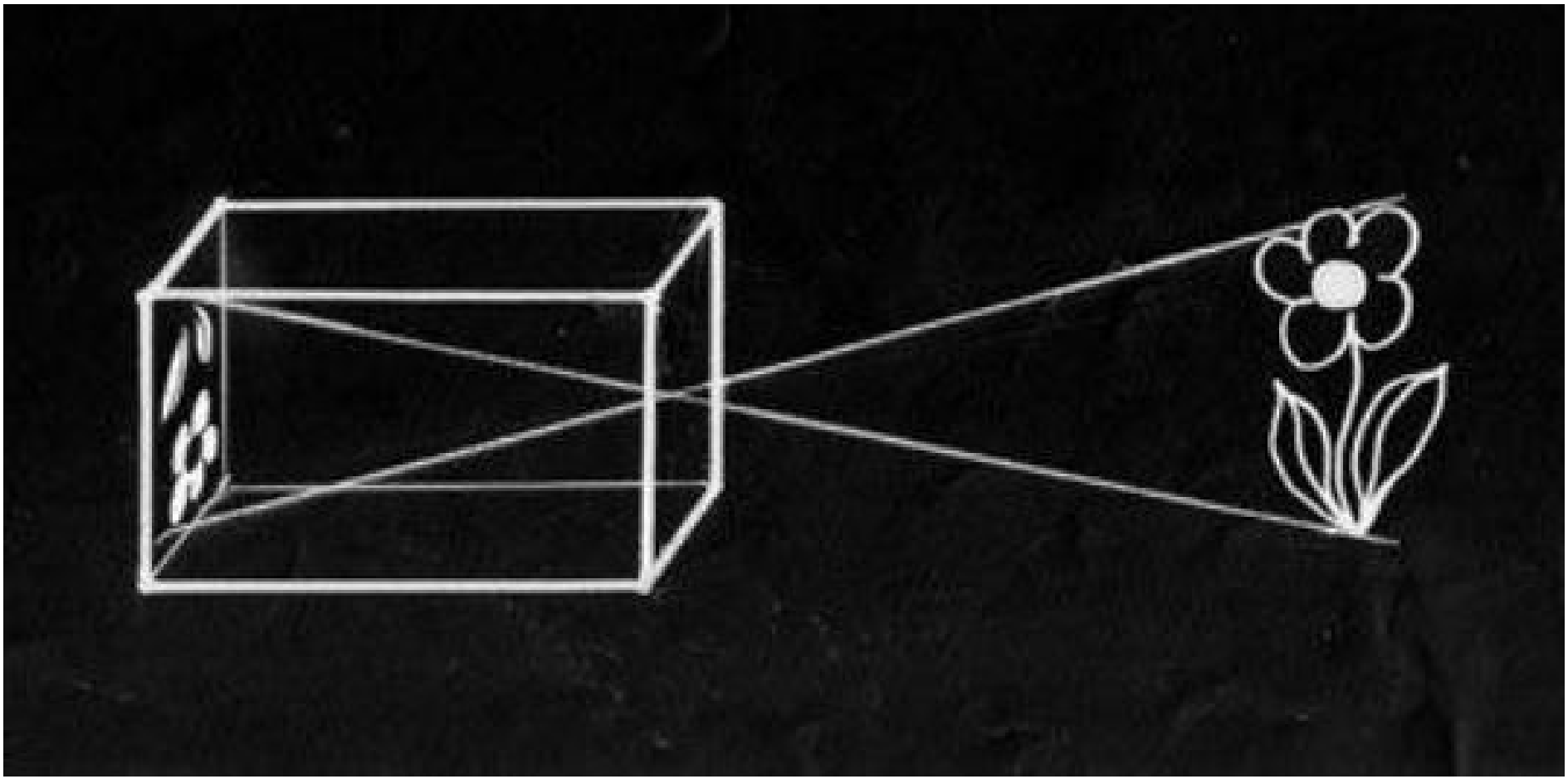
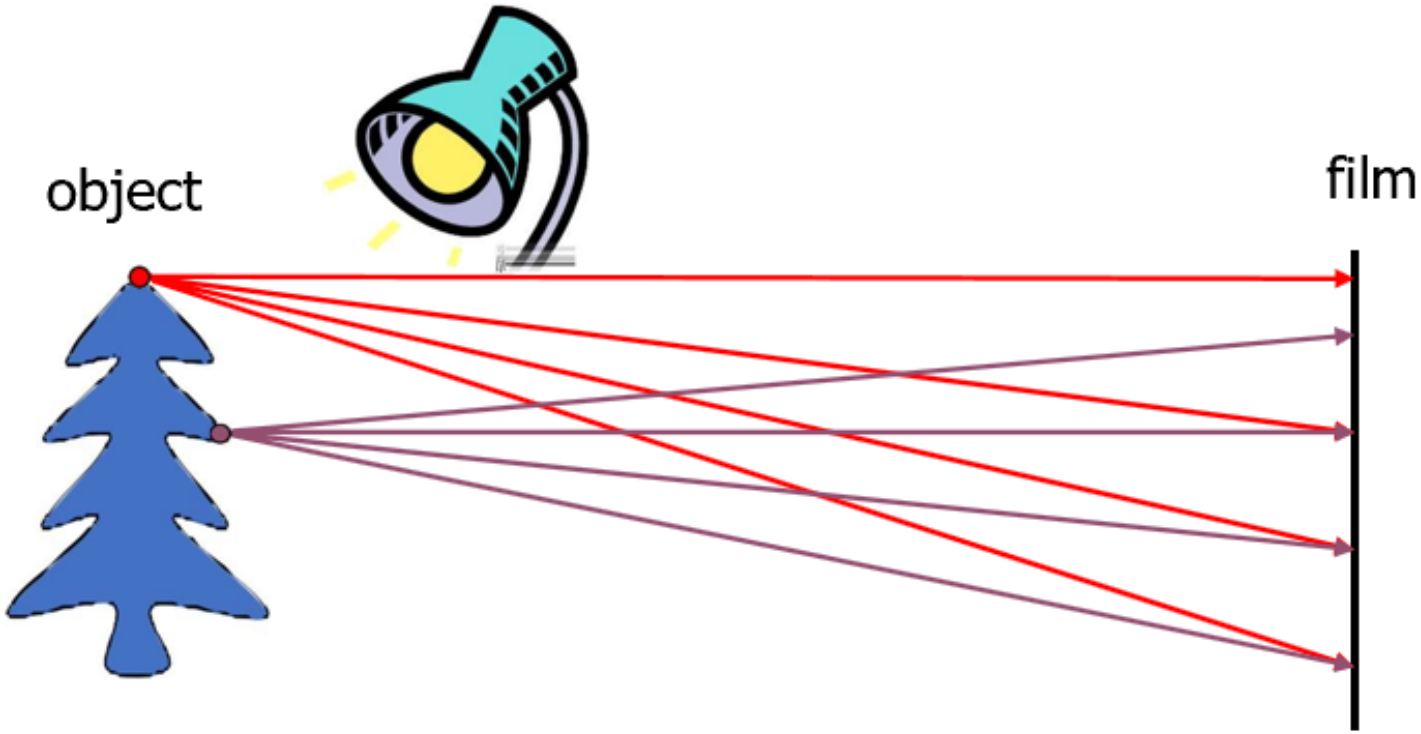


Figure: Camera model can be simplified as a pinhole camera model.

How to form an image?

Place a piece of film in front of an object.

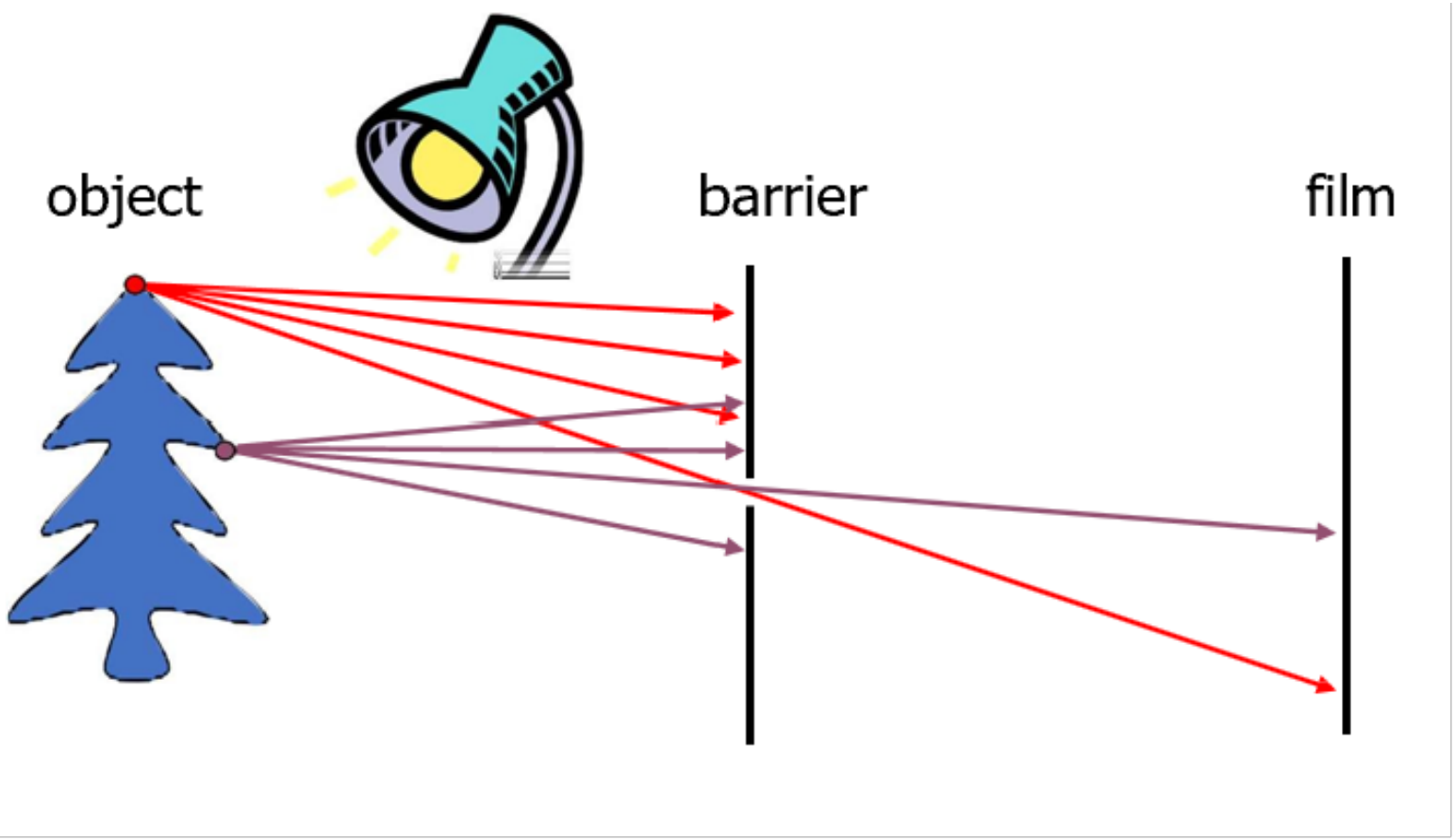


Do we get a reasonable image?

Pinhole Camera Model

Add a barrier to block off most of the rays

- It reduces blurring
- The opening is known as **aperture**

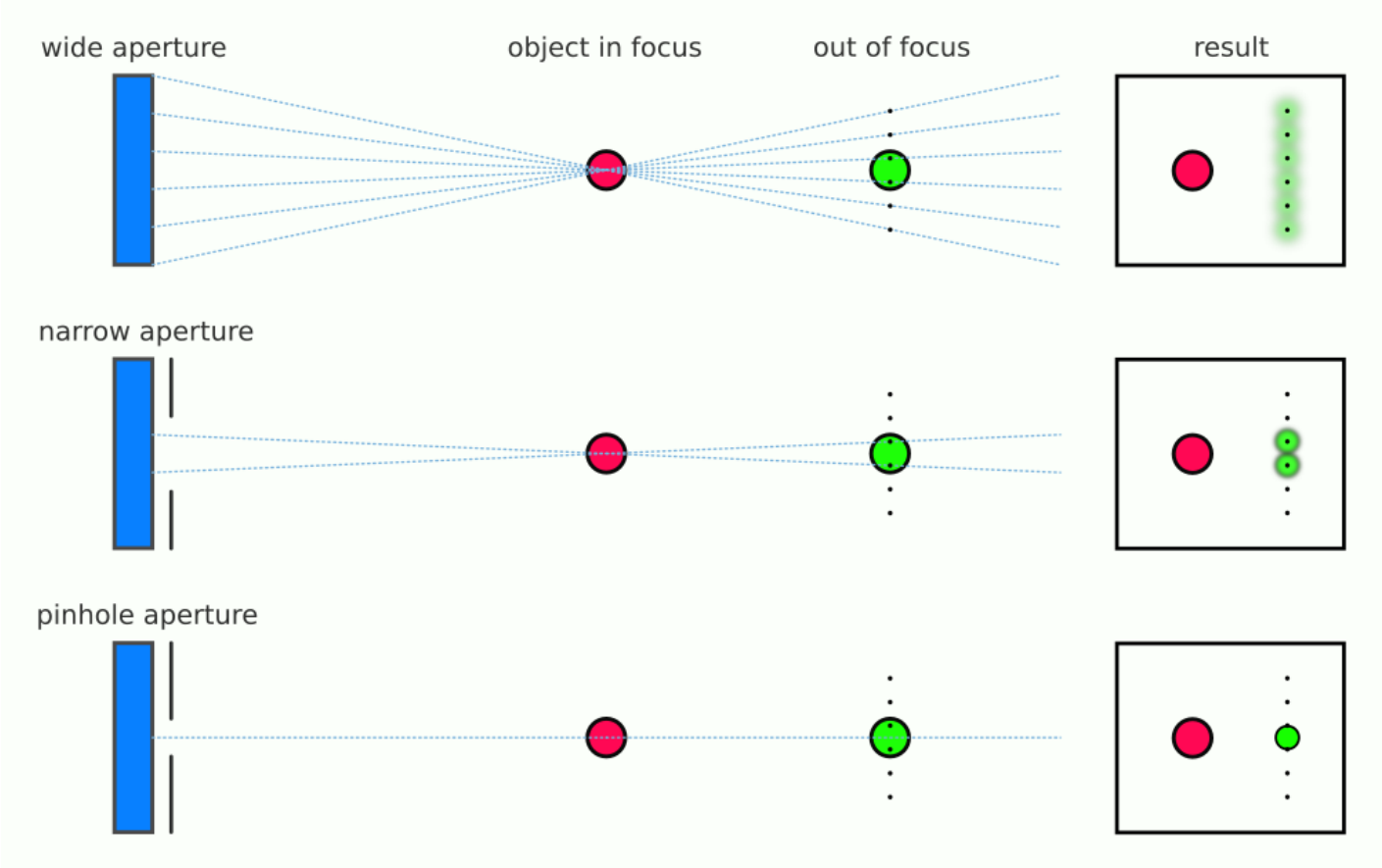


Home-made Pinhole Camera

This is a pinhole camera image - but it's blurred

What can we do to reduce the blur?

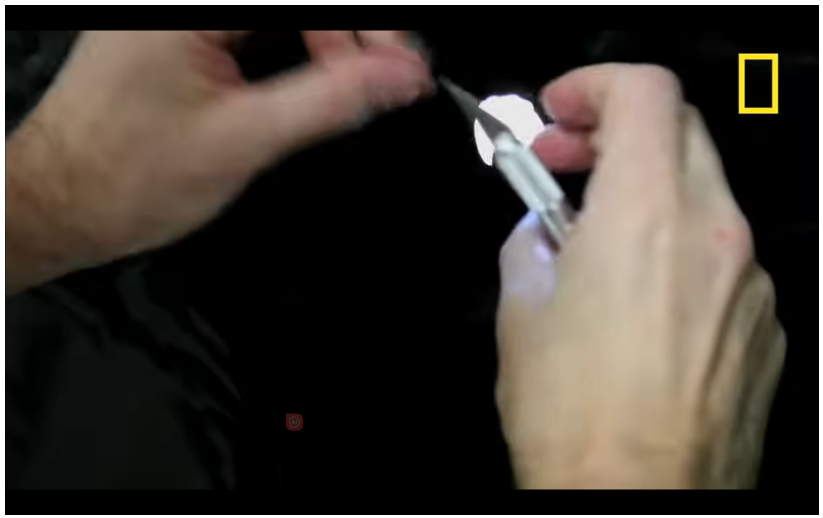
Home-made Pinhole Camera

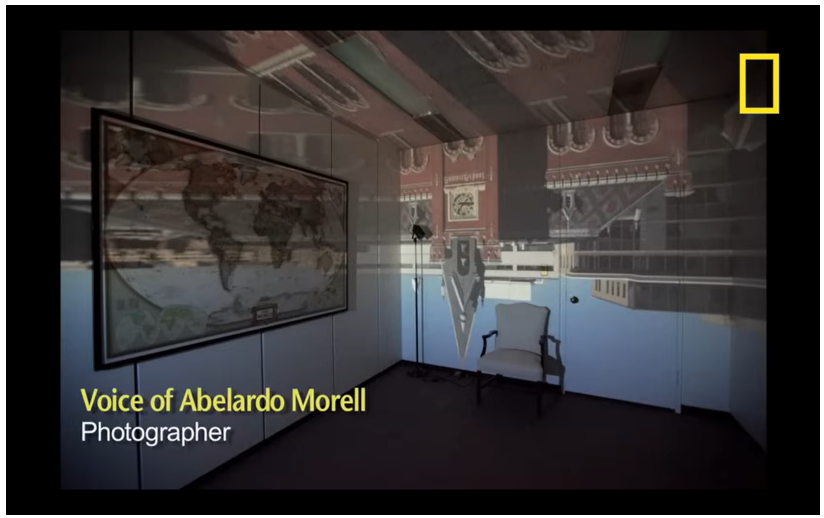


Smaller pinhole aperture

Home-made Pinhole Camera

A ‘room-sized’ pinhole camera

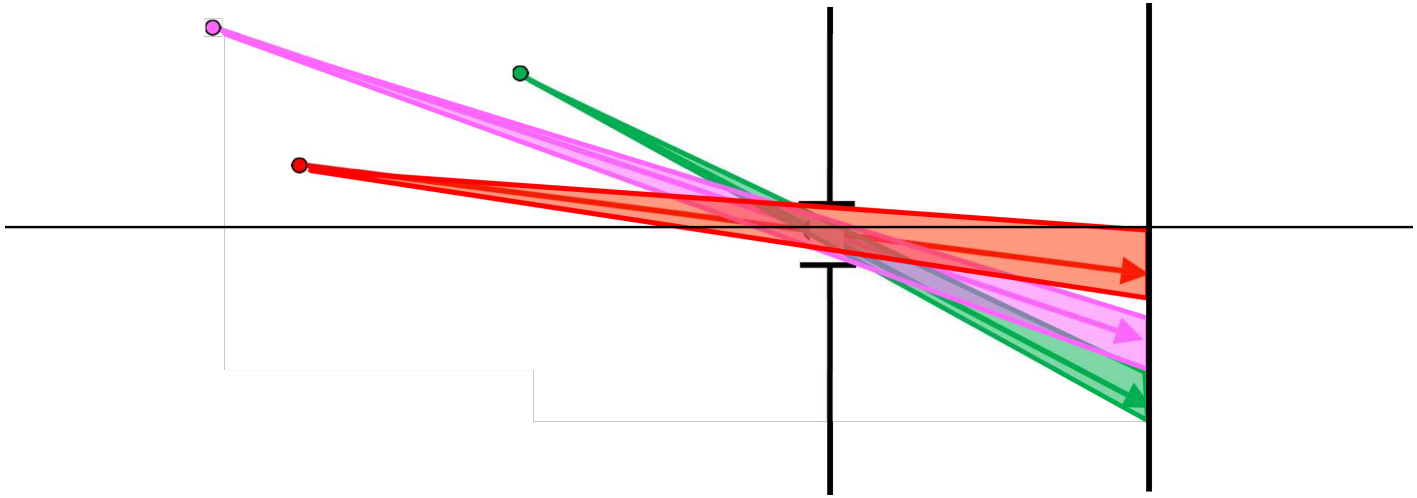




Making Your Own Room With a View, *National Geographic*: https://youtu.be/gvzpu0Q9RTU

Why do we use a lens?

- The ideal pinhole: only one ray of light reaches each point on the film
  –> but the image can be very dim
  –> We need to make the pinhole (i.e. aperture) bigger



- A lens can focus multiple rays coming from the same point

Image formation using a converging lens

- A lens focuses light onto the film
- Rays passing through the optical center are **NOT deviated**

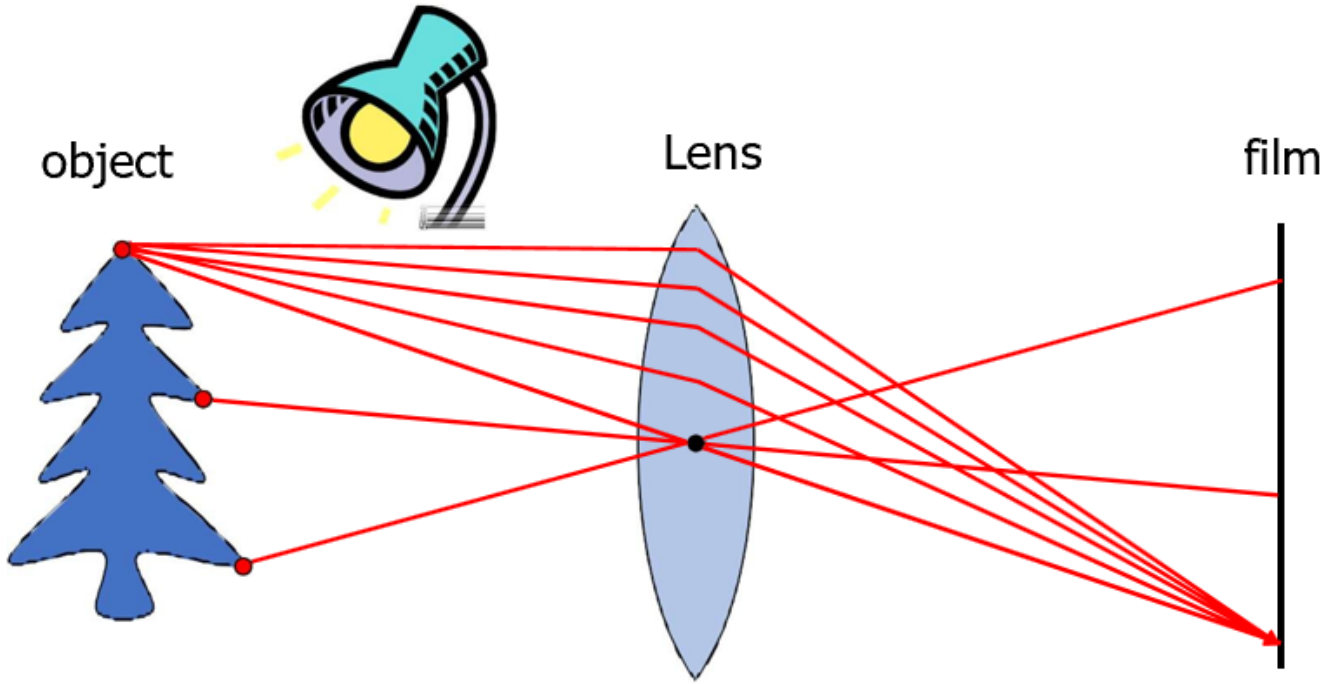
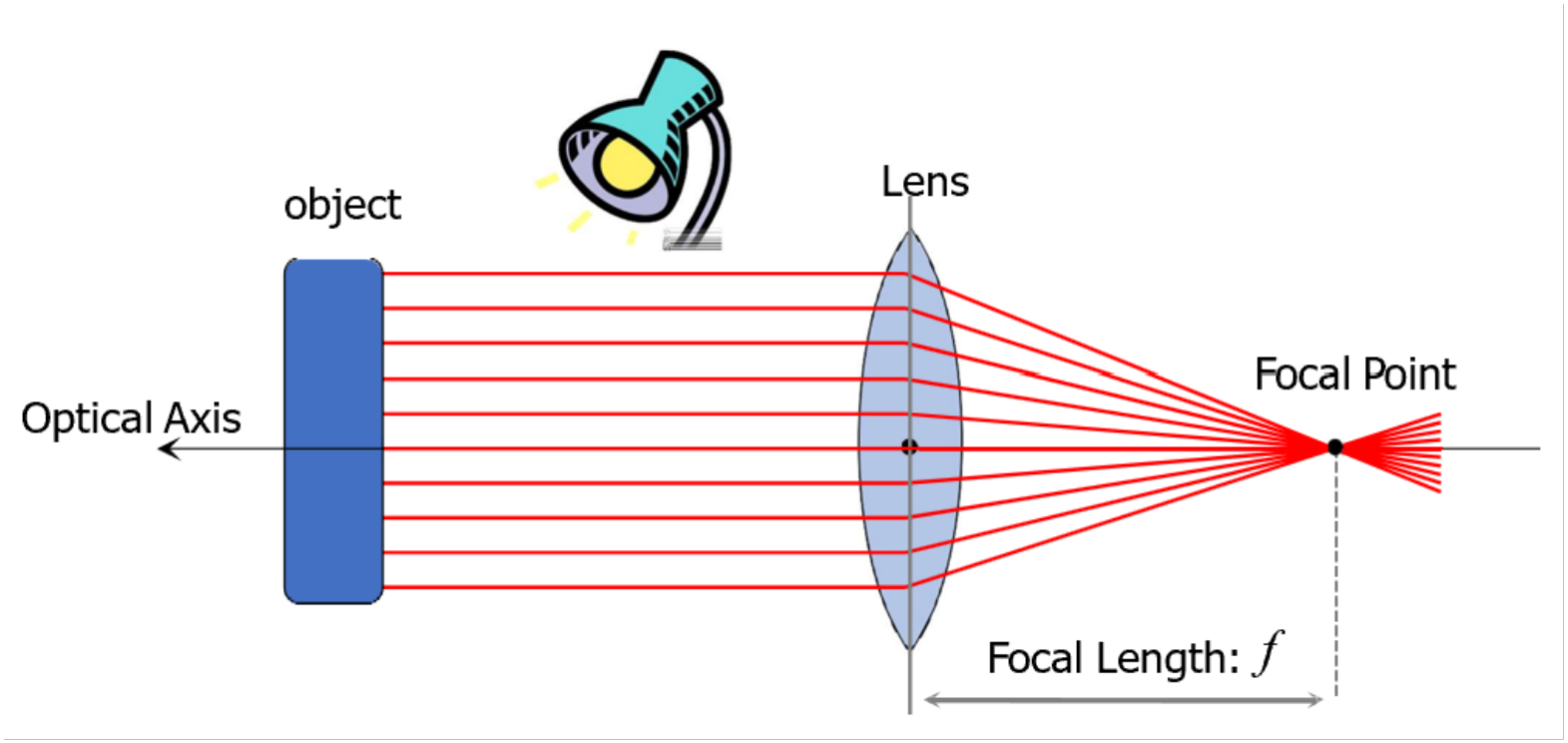


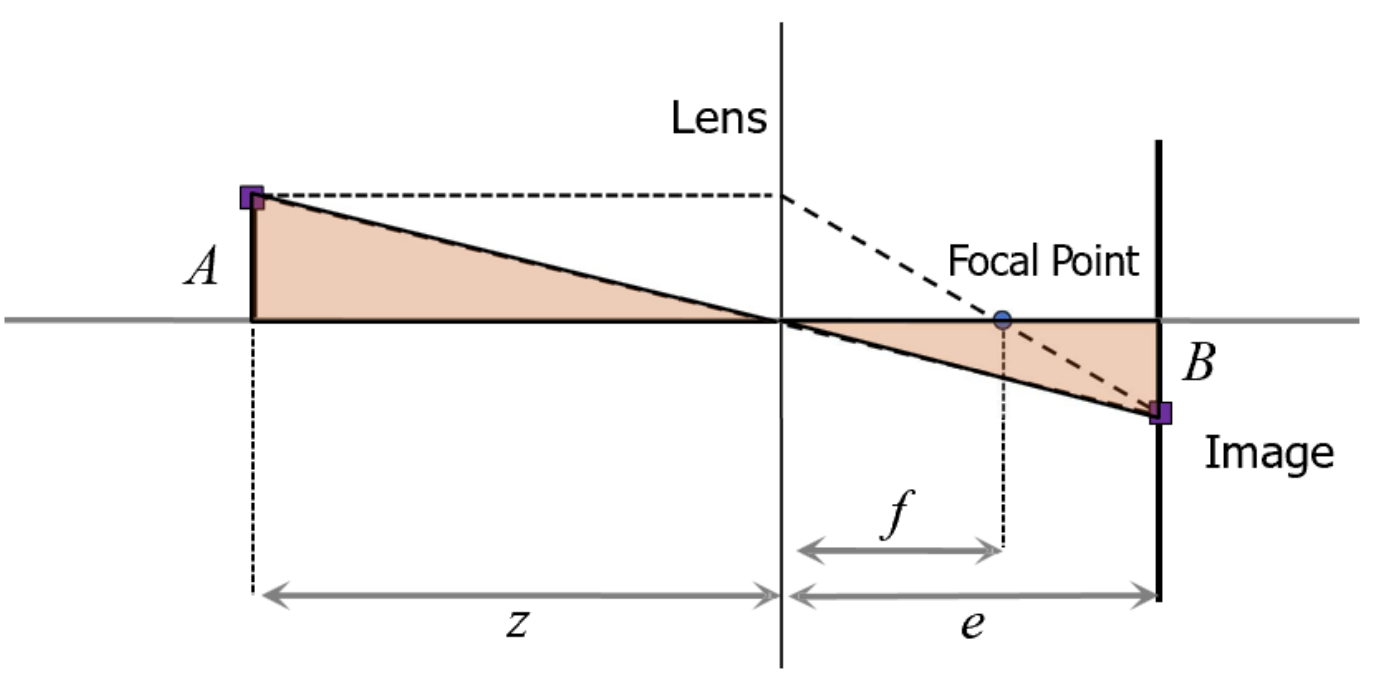
Image formation using a converging lens

- A lens focuses light onto the film
- Rays passing through the optical center are **NOT deviated**
- All rays parallel to the **Optical Axis** will converge at the **Focal Point**



Thin lens equation

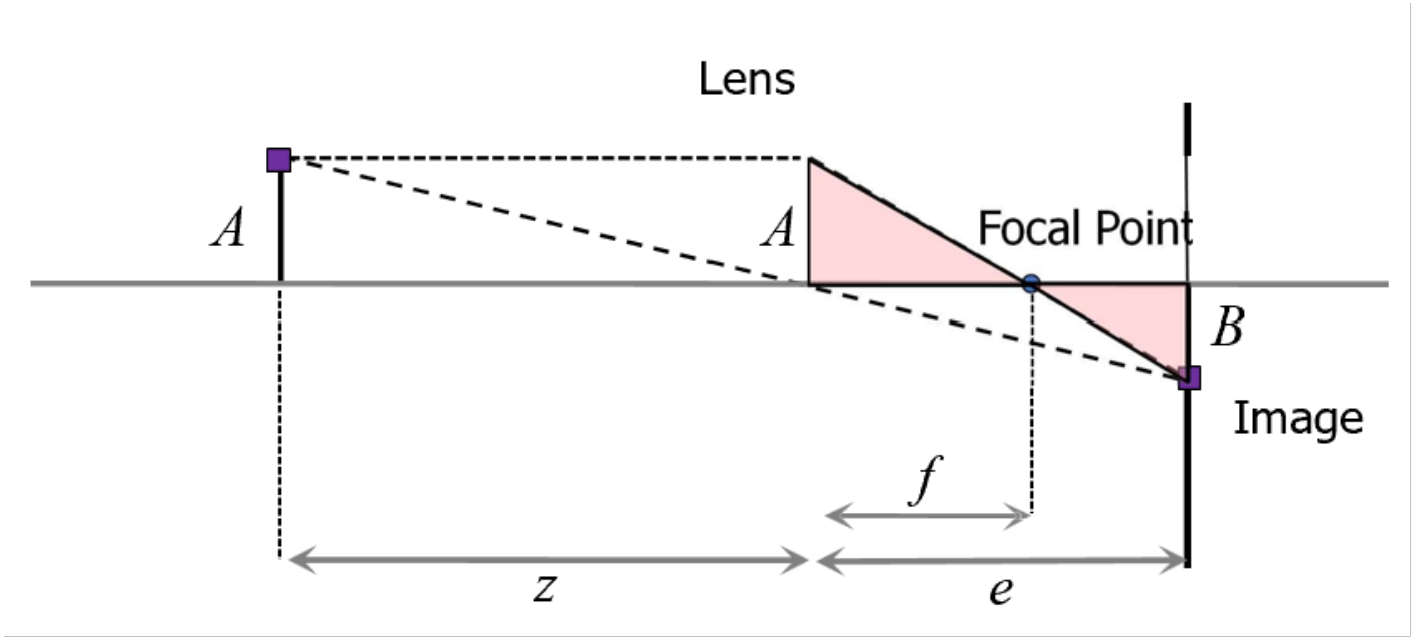
Similar Triangles (相似三角形):



$$\frac{B}{A} = \frac{e}{z}$$

Find a relationship between f, z, and e

Thin lens equation



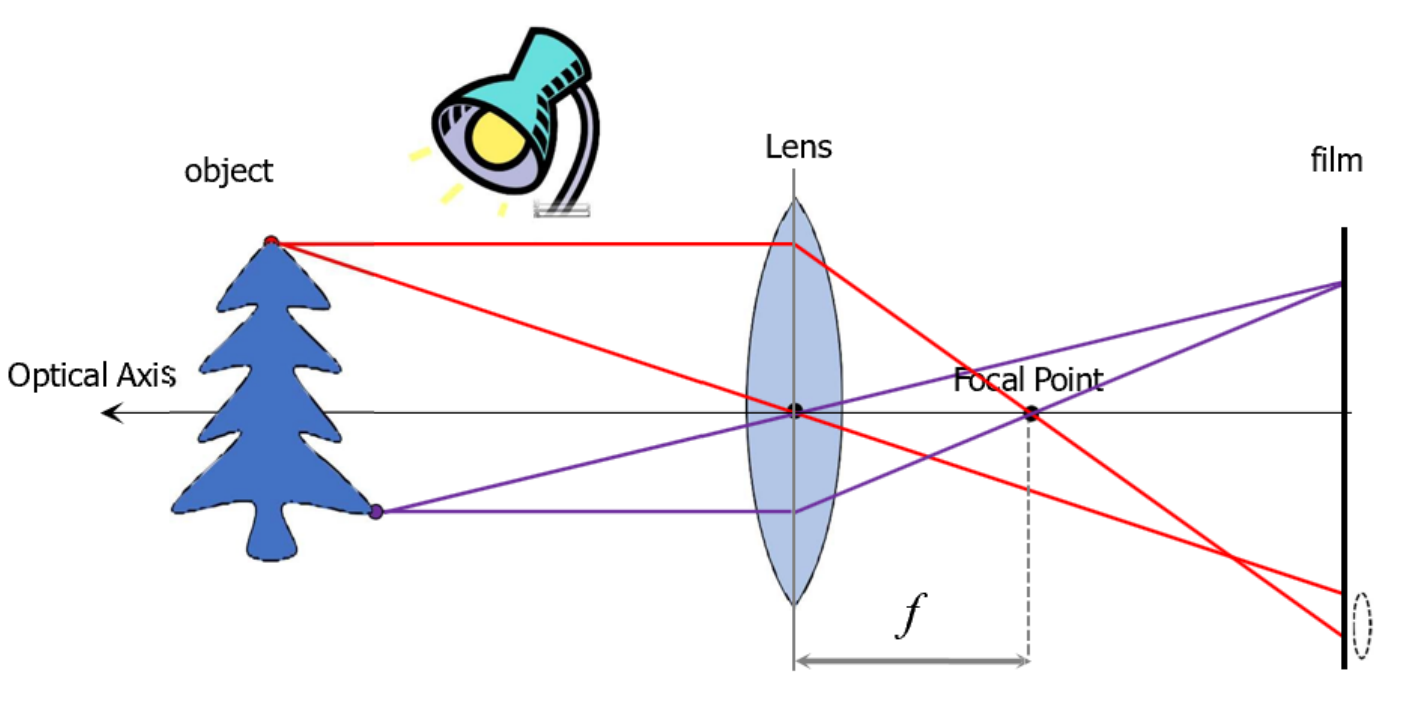
$$\left.\begin{aligned} \frac{B}{A} &= \frac{e-f}{f} = \frac{e}{f} - 1 \\ \frac{B}{A} &= \frac{e}{z} \end{aligned}\right\} \frac{e}{f} - 1 = \frac{e}{z} \rightarrow \frac{1}{f} = \frac{1}{z} + \frac{1}{e}$$

"Thin lens equation"

Any object point satisfying this equation is **in focus**

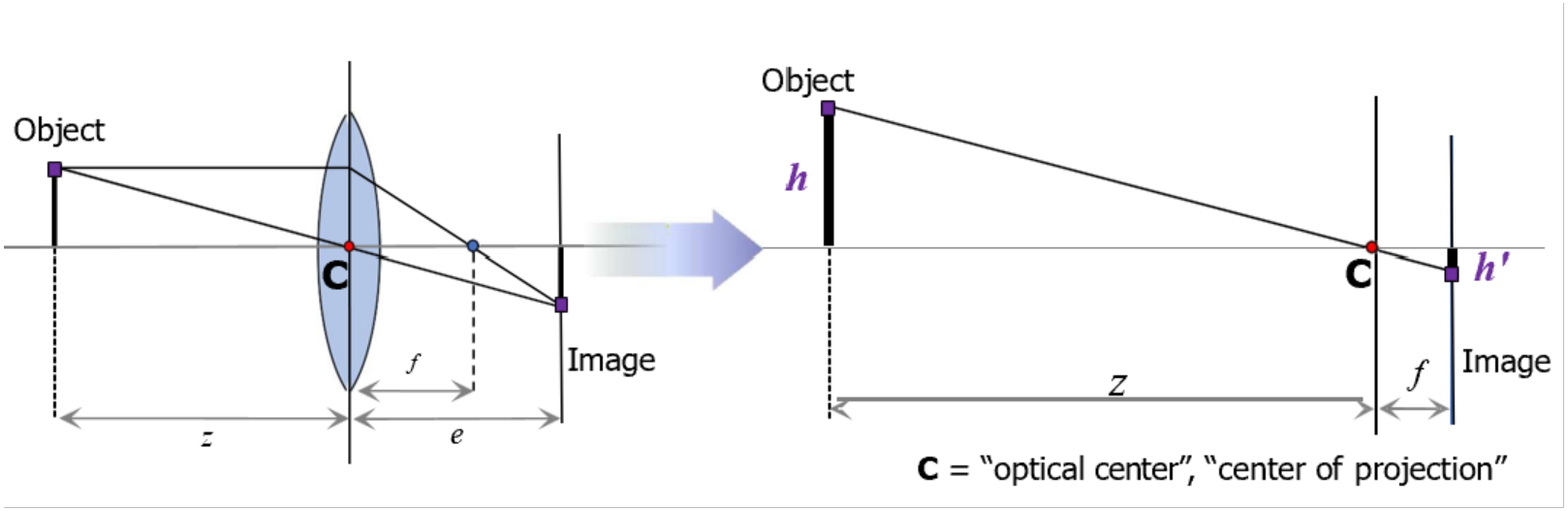
"In Focus"

- There is a specific distance from the lens, at which the **world points** are "in focus" in the image
- Other points project to a "blur circle" in the image



On the right: "Circle of Confusion" or "Blur Circle"

Perspective Camera

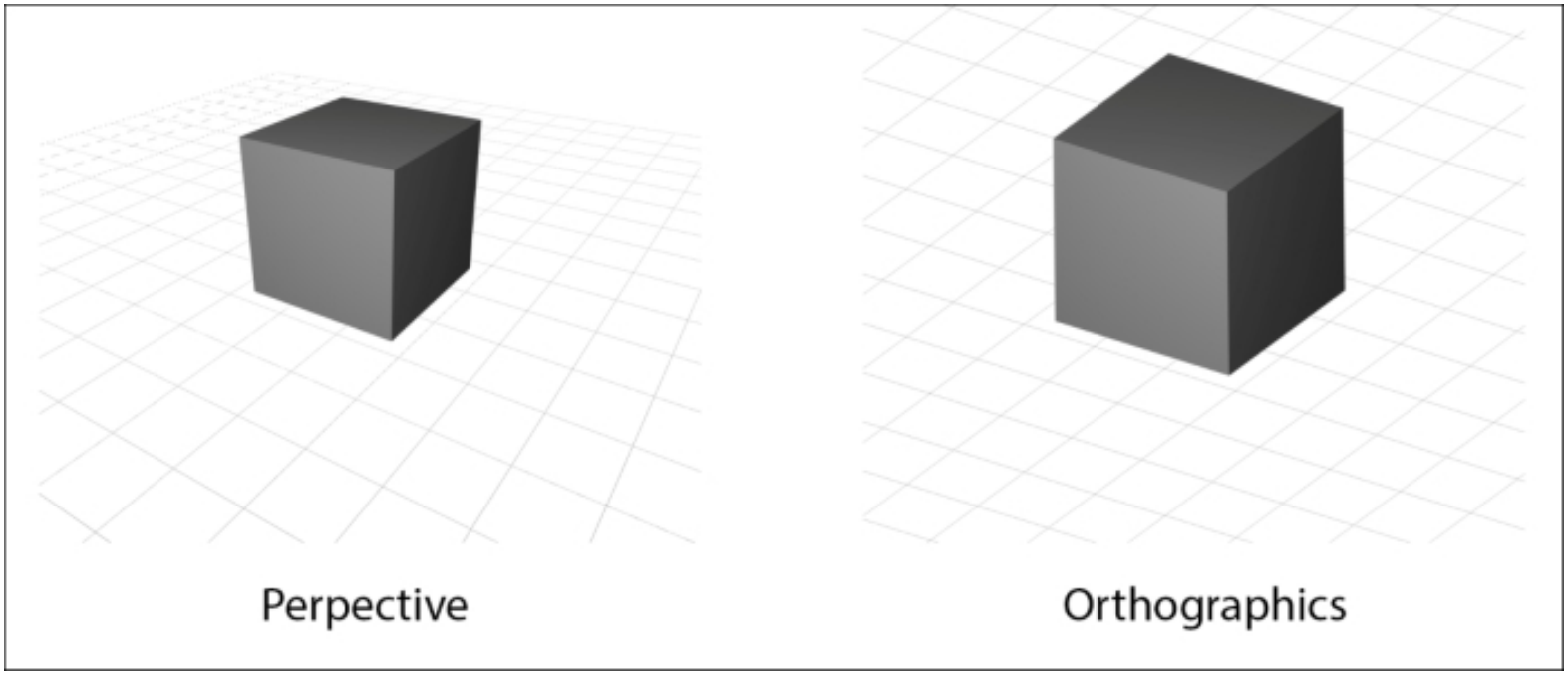


- Adjust the image plane so that the objects at infinity are **in focus**
- So,

$$\left.\begin{array}{l} z >> f, \quad z >> L \\ \dfrac{1}{f} = \dfrac{1}{z} + \dfrac{1}{e} \end{array}\right\} \frac{1}{f} \approx \frac{1}{e} \rightarrow f \approx e$$

- With a constant f, we can express a relationship on perspective: The dependence of the apparent size of an object on its depth (i.e. distance from the camera) is known as perspective

Perspective vs Orthographics

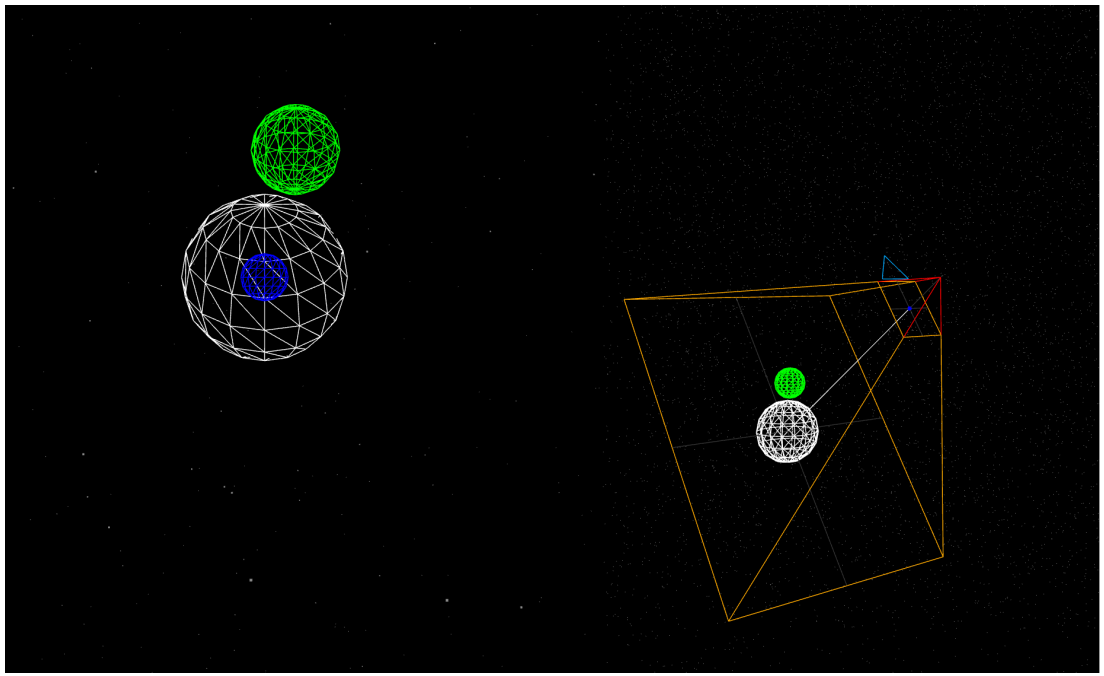


Which one is more natural / intuitive to you?

More reading: https://www.mathworks.com/help/matlab/creating_plots/understanding-view-projections.html

Playing with Perspective

- Perspective gives us very strong **depth** cues
- Hence, we can perceive a 3D scene by viewing its 2D representation (i.e. 2D image)



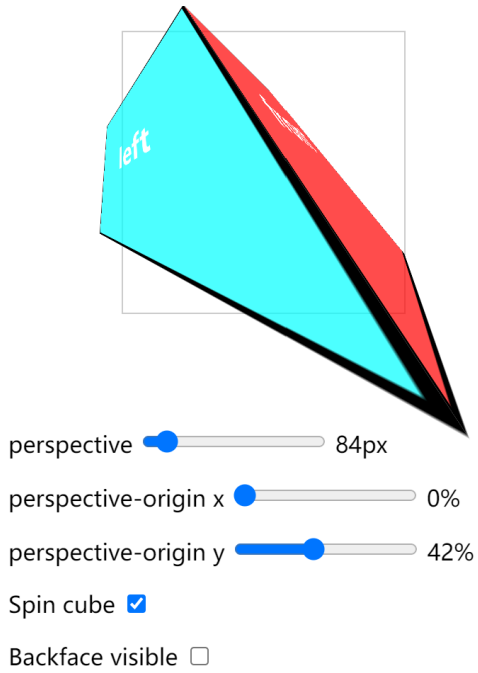
Objects vary in sizes within different distances

Try it yourself: https://threejs.org/examples/webgl_camera.html
(Press 'O' and 'P' to switch between Orthographic and Perspective mode)

Playing with Persepctive

Perspective Effects

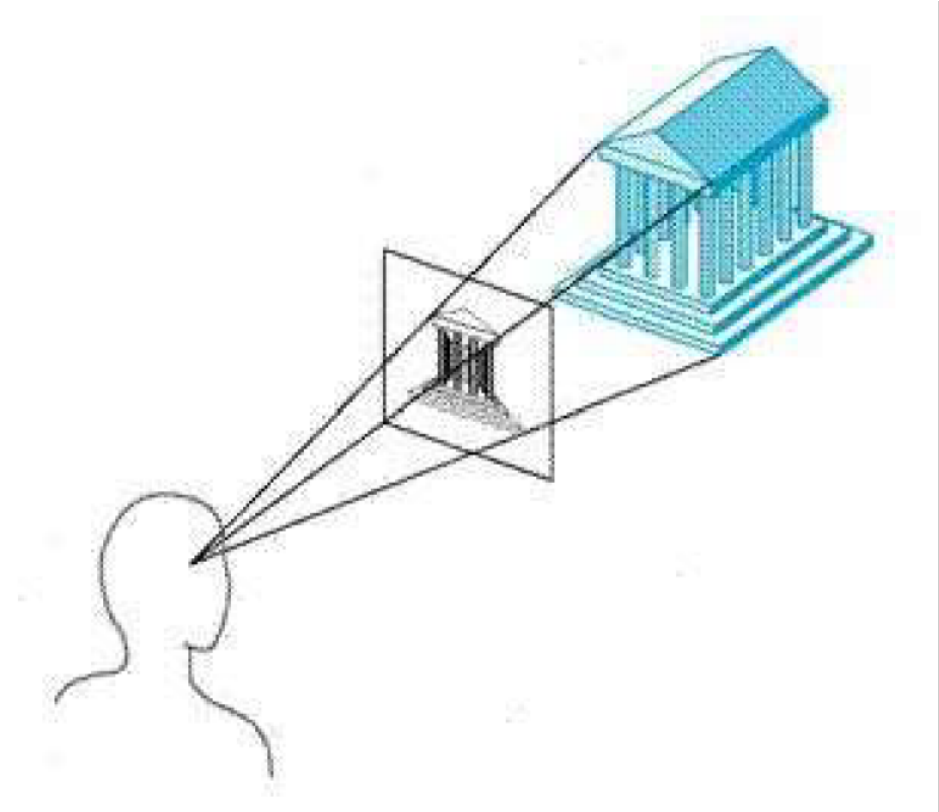
- Parallel lines do not remain parallel
- Our visual system expects all parallel lines to intersect at **one point**



Try this to play with the "perspective-ness": https://codepen.io/desandro/pen/bMqZmr

Perspective Projection

How the **3D world points** map to **pixels** in the 2D image?



From World to Pixel coordinates

Goal: We want to use a 2D pixel coordinates (u, v) to represent a real-world point P_w

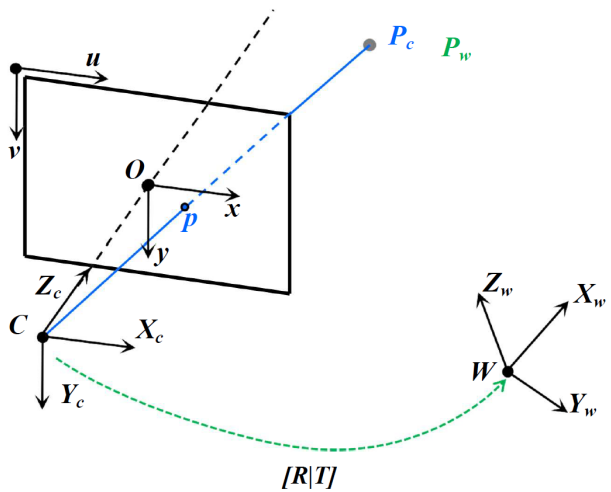
Tools: We have $\{W\}$, $\{C\}$, and $\{O\}$.

Find pixel coordinates (u, v) of point P_w in the world frame $\{W\}$:

1. Convert world point P_w to camera point P_c

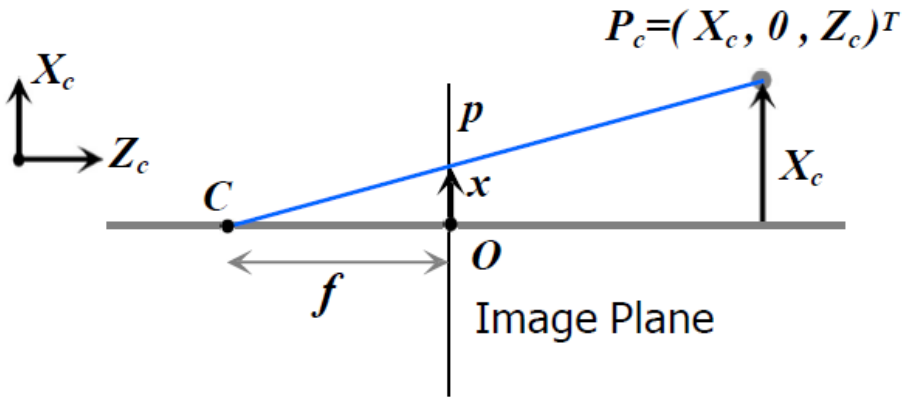
Find pixel coordinates (u, v) of point P_c in the camera frame $\{C\}$:

1. Convert P_c to image-plane coordinates (x, y)
2. Convert P_c to (discrete) pixel coordinates (u, v)



From Camera frame to Image plane

- Camera Point $P_c = [X_c, 0, Z_c]^\top$ projects to $p = [x, y]^\top$ onto the image plane
- According to similar triangles: $\frac{x}{y} = \frac{X_c}{Z_c} \Rightarrow x = \frac{fX_c}{Z_c}$
- Similarly, in the general case: $\frac{y}{f} = \frac{Y_c}{Z_c} \Rightarrow y = \frac{fY_c}{Z_c}$



From Camera frame to ~~Image plane~~ Pixel coordinates

In order to convert p from local image plane coords $p = [x, y]^\top$ to pixel coords $[u, v]^\top$, we need to account for:

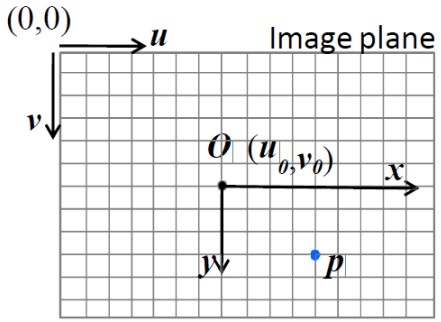
- The pixel coords of the camera optical center $O = [u_0, v_0]^\top$
- Scale factors k_u, k_v for the **pixel-size** in both dimensions, such that:

$$u = u_0 + k_u x \Rightarrow u = u_0 + \frac{k_u f X_c}{Z_c}$$

$$v = v_0 + k_v y \Rightarrow v = v_0 + \frac{k_v f Y_c}{Z_c}$$

- Use **Homogeneous Coordinates** for linear mapping from 3D to 2D, by introducing an **extra scale** λ:

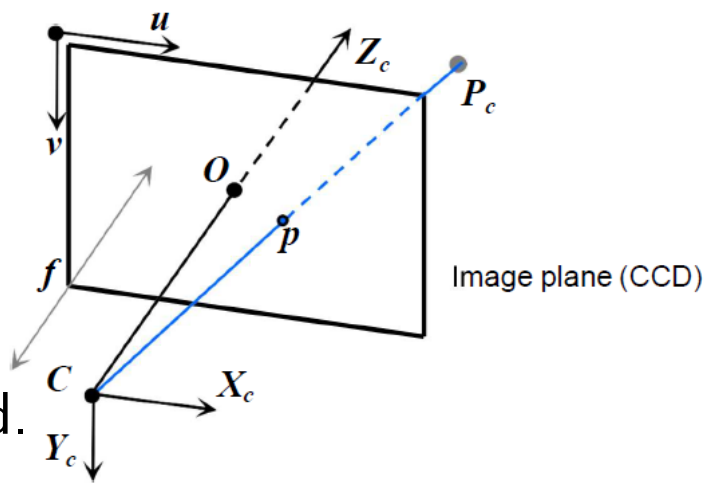
$$p = \begin{bmatrix} u \\ v \end{bmatrix} \Rightarrow \tilde{p} = \begin{bmatrix} \lambda u \\ \lambda v \\ \lambda \end{bmatrix}$$



Note that usually $\lambda = 1$

Perspective Projection

Now we have: $\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u_0 + \dfrac{k_u f X_c}{Z_c} \\ v_0 + \dfrac{k_v f Y_c}{Z_c} \end{bmatrix}$



Rewrite in **matrix form** with homogeneous coord.

$$\begin{bmatrix} \lambda u \\ \lambda v \\ \lambda \end{bmatrix} = \begin{bmatrix} k_u f & 0 & u_0 \\ 0 & k_v f & v_0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix}$$

Or, in a shorter way: $\begin{bmatrix} \lambda u \\ \lambda v \\ \lambda \end{bmatrix} = \underbrace{\begin{bmatrix} \alpha_u & 0 & u_0 \\ 0 & \alpha_v & v_0 \\ 0 & 0 & 0 \end{bmatrix}}_{\text{'Calibration matrix' or 'Matrix of Intrinsic Parameters'}} \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = K \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix}$

α_u: Focal length in u-direction

α_v: Focal length in u-direction

From Camera frame to ~~Image plane~~ ~~Pixel coordinates~~ World frame

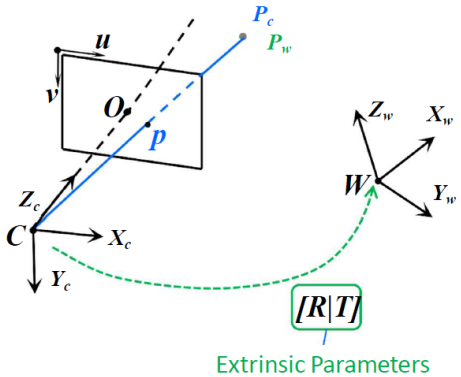
Define Camera frame by World frame (Through spatial trans/rot)

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \end{bmatrix} + \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

$$\xrightarrow[\text{coord.}]{\text{in homog.}} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} R & T \\ 0_{1\times 3} & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Define Pixel Frame by Camera Frame: $\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix}$

Define Pixel Frame by World Frame: $\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \left(\begin{bmatrix} R & T \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \right)$



Here, $K[R|T]$ is the **Projection Matrix**
$[R|T]$ is the **Matrix of Extrinsic Parameters**

Perspective Projection

Define Pixel Frame (2D) by World Frame (3D):

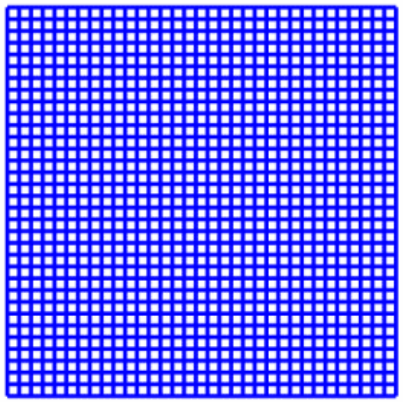
$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \left(\begin{bmatrix} R & T \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \right)$$

$$\Downarrow$$

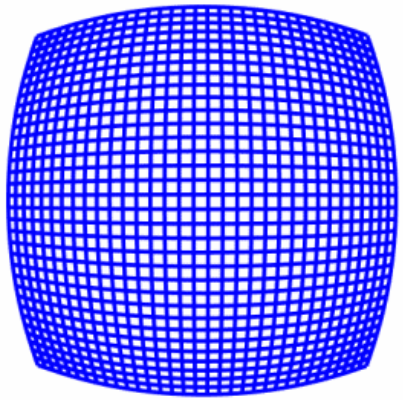
$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Intrinsic Param. Matrix}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \underbrace{\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\text{Extrinsic Param. Matrix}} \underbrace{\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}}_{\{W\}}$$

Radial lens distortion

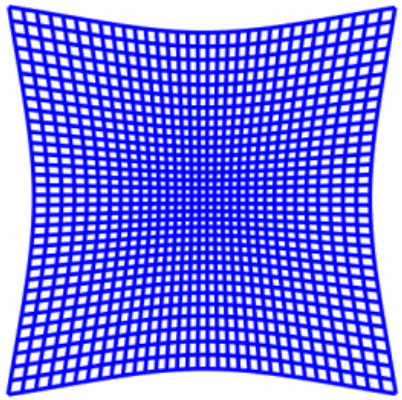
- Radial lens distortion causes straight lines to be mapped as curved lines.
- There are two types of radial lens distortion: **Barrel Distortion** and **Pincushion Distortion**.



INPUT GRID



BARREL DISTORTION



PINCUSHION DISTORTION

Camera Calibration

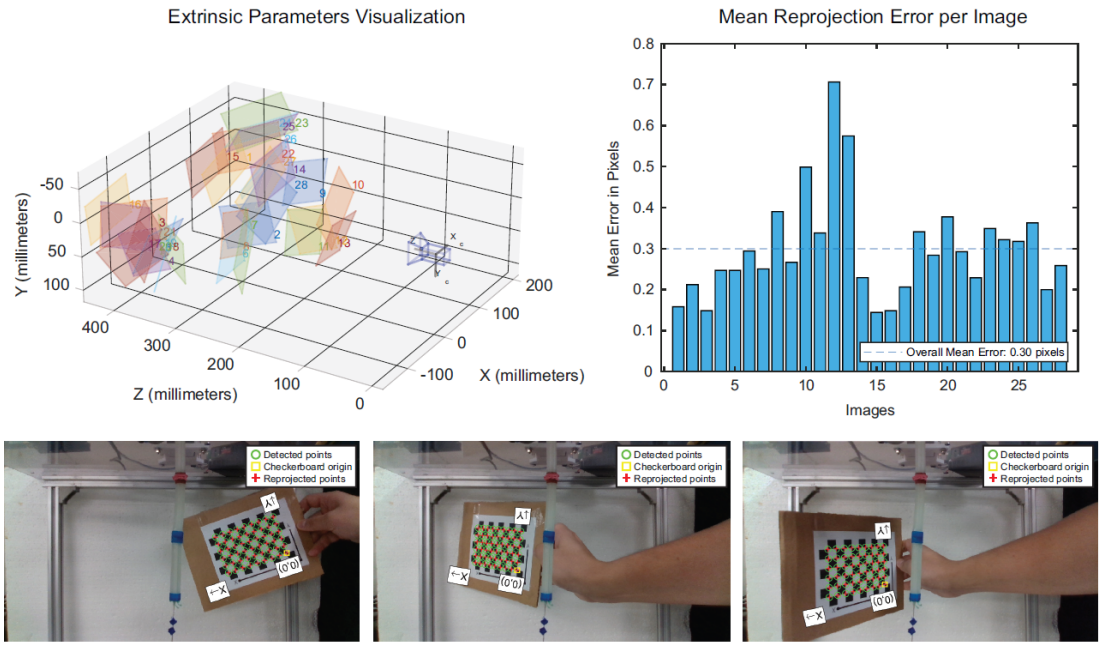
Normally, we will use a **checkerboard** to find the **intrinsic parameters**.

- Camera Calibrator in Matlab:
  https://www.mathworks.com/help/vision/ref/cameracalibrator-app.html
- Camera Calibrator in OpenCV:
  https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html

Camera Calibration

Recall: $\lambda \begin{bmatrix} u & v & 1 \end{bmatrix}^\top = \underbrace{K \begin{bmatrix} R & T \end{bmatrix}}_{\text{Projection Matrix}} \underbrace{\begin{bmatrix} X_w & Y_w & Z_w & 1 \end{bmatrix}^\top}_{\text{World Frame}}$

- Use Cam model to interpret the projection from **world** to **image plane**
- With the known correspondences of p and P_w, we can compute the Projection Matrix by applying the perspective projection equation
- ... so we can associate known physical distance in real-world with pixel-distance in image to figure out the Projection Matrix



Some sample images for calibration

Camera Calibration

Recall: $\lambda \begin{bmatrix} u & v & 1 \end{bmatrix}^\top = \underbrace{K \begin{bmatrix} R & T \end{bmatrix}}_{\text{Projection Matrix}} \underbrace{\begin{bmatrix} X_w & Y_w & Z_w & 1 \end{bmatrix}^\top}_{\text{World Frame}}$

- We know that: $\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$
- So there are 11 values to estimate: (the overall scale doesn't matter, so m_{34} can be set to 1 for now)
- Each observed point gives us a pair of equations:

$$u_i = \frac{\lambda u_i}{\lambda} = \frac{m_{11}X_i + m_{12}Y_i + m_{13}Z_i + m_{14}}{m_{31} + m_{32} + m_{33} + m_{34}}$$

$$v_i = \frac{\lambda v_i}{\lambda} = \frac{m_{21}X_i + m_{22}Y_i + m_{23}Z_i + m_{24}}{m_{31} + m_{32} + m_{33} + m_{34}}$$

- To estimate 11 unknowns, we need at least **6 points** to calibrate the camera using linear least squares

Camera Calibration

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix}}_{\text{Projection Matrix}} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \underbrace{K \begin{bmatrix} R & T \end{bmatrix}}_{\text{Projection Matrix}} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

- **What we obtained**: a 3-by-4 projection matrix
- **What we need**: its decomposition into the **intrinsic parameter matrix** K, and the **rotation** R and **transformation** T of the camera
- For K and R: Use **QR factorization** to decompose the 3-by-3 submatrix ($m_{11:33}$) into the product of an **upper triangular matrix** K and a **rotation matrix** R (which is always an orthogonal matrix)
- The translation T can be subsequently obtained by: $T = K^{-1} \begin{bmatrix} m_{14} \\ m_{24} \\ m_{34} \end{bmatrix}$

Thanks for listening!

Part 3. Prep for Lab 9 - Task 1

Prep for Lab 9 - Task 1

- Copy the bags from TAs
- Download the sample codes from Blackboard:
  `aruco_detector_sample.py`
  `image_listener_sample.py`
  `simple_controller_sample.py`
- Follow the guide on the lab sheet to create a package for lab 9

Part 4. Camera Calibration - Task 2

Camera Calibration - Task 2

Camera calibration steps:

- Type the following command in the terminal

Terminal

```
$ rosrun camera_calibration cameracalibrator.py --size 8x6 -
-square 0.015 image:=/camera/color/image_raw
```

- Play calibration.bag. You will see the checkerboard moving around in the scope of the camera

Terminal

```
$ cd lab9/bags
$ rosbag play calibration.bag
```

- When all the progressing bars in the GUI turn green, click the Calibrate button; the software will calculate the intrinsic param and print the result in the terminal
- Record the results for Task 3

Part 5. Aruco Marker Detection - Task 3

Aruco Marker Detection - Task 3

Detect Aruco marker with RealSense (follow the guide on the lab sheet)

1. Complete TODOs in `aruco_detector_sample.py`
2. Play rosbag `aruco.bag`

Terminal

```
$ rosbag play aruco.bag
```

3. Run your `aruco_detector`

Terminal

```
$ rosrun lab9 aruco_detector_sample.py
```

Part 6. Turtlebot Following Aruco Marker - Task 4

Turtlebot Following Aruco Marker - Task 4

Install turtlebot3 packages in ROS

1. Go to your ws src folder

Terminal

```
$ cd ~/catkin_ws/src
```

2. Download packages from turtlebot3 repository

Terminal

```
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.git
```

3. Download turtlebot3_msgs.

Terminal

```
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git
```

Turtlebot Following Aruco Marker - Task 4

4. Download turtlebot3_simulation packages

Terminal

```
$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
```

5. Go back to your workspace

Terminal

```
$ cd ..
```

6. Compile all the packages newly added

Terminal

```
$ catkin_make
```

Turtlebot Following Aruco Marker - Task 4

Turtlebot follows Aruco marker

1. Complete the TODOs in `simple_controller_sample.py` from Blackboard (we use a P-Controller same as lab 4)
2. Bring up turtlebot

Terminal

```
$ export TURTLEBOT3_MODEL=burger
roslaunch turtlebot3_fake turtlebot3_fake.launch
```

3. Run your `aruco_detector`

Terminal

```
$ rosrun lab9 aruco_detector_simple.py
```

Turtlebot Following Aruco Marker - Task 4

4. cd to bags, and play `aruco.bag`

Terminal

```
$ cd catkin_ws/src/lab9/bags
rosbag play aruco.bag
```

5. Check the result in RViz (see if the frame of aruco marker is moving.)
6. Run your controller.

Terminal

```
$ rosrun lab9 simple_controller_sample.py
```

7. You should see the turtlebot moving in RViz