Slides and example codes are available:
https://github.com/sunglok/3dv_tutorial

An Invitation
~~Introduction~~ to 3D Vision
: A Tutorial for Everyone

Sunglok Choi, Senior Researcher
sunglok@etri.re.kr | http://sites.google.com/site/sunglok
Electronics and Telecommunication Research Institute (ETRI)

What is Computer Vision?

Computer vision is an interdisciplinary field that deals with how computers can be made to gain high-level understanding from digital images or videos. From the perspective of engineering, it seeks to automate tasks that the human visual system can do.[1][2][3] "Computer vision is concerned with the automatic extraction, analysis and understanding of useful information from a single image or a sequence of images. It involves the development of a theoretical and algorithmic basis to achieve automatic visual understanding."[9] As a scientific discipline, computer vision is concerned with the theory behind artificial systems that extract information from images. The image data can take many forms, such as video sequences, views from multiple cameras, or multi-dimensional data from a medical scanner.[10] As a technological discipline, computer vision seeks to apply its theories and models for the construction of computer vision systems.

Computer Vision by Wikipedia

What is Computer Vision?

Computer vision is an interdisciplinary field that deals with how computers can be made to gain high-level understanding from digital images or videos. From the perspective of engineering, it seeks to **automate** tasks that the human visual system can do.[1][2][3] "Computer vision is concerned with the **automatic** extraction, analysis and understanding of useful information **from a single image or a sequence of images**. It involves the development of a theoretical and algorithmic basis to achieve **automatic visual understanding**."[9] As a scientific discipline, computer vision is concerned with the theory behind artificial systems that extract information from images. The image data can take many forms, such as video sequences, views from multiple cameras, or multi-dimensional data from a medical scanner.[10] As a technological discipline, computer vision seeks to apply its theories and models for the construction of computer vision systems.

Computer Vision by Wikipedia

What is Computer Vision?

Image Understanding

(Synthesized) Image

Computer Graphics

inverse relationship

Tower of Pisa

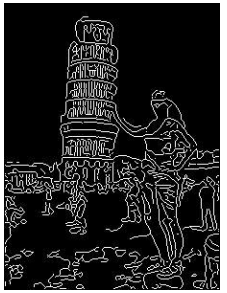
Real World

(Projected) Image

Computer Vision

Information

Image Processing



(transformed) Image/Signal

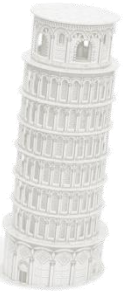
What is Computer Vision?

Computer Vision

What is it?

- Label (e.g. Tower of Pisa)
- Shape (e.g. )

Where am I?

- Place (e.g. Piazza del Duomo, Pisa, Italy)
- Location (e.g. )

(84, 10, 18) [m]

What is 3D Vision?

Visual Geometry (Multiple View Geometry)

Geometric Vision

Computer Vision

What is it?

- Label (e.g. Tower of Pisa)
- Shape (e.g. )

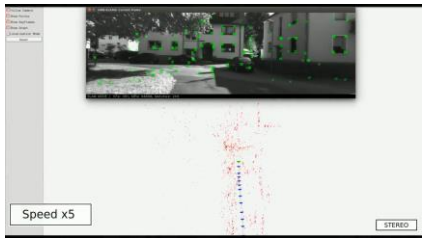
Where am I?

- Place (e.g. Piazza del Duomo, Pisa, Italy)
- Location (e.g. )

(84, 10, 18) [m]

Recognition Problems **v.s.** **Reconstruction Problems**

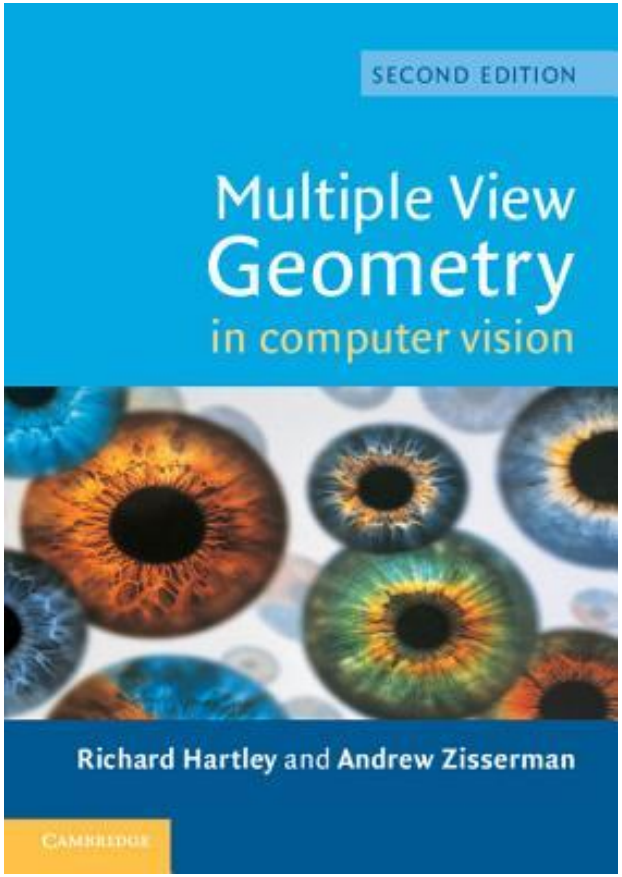
YOLO v2 (2016)

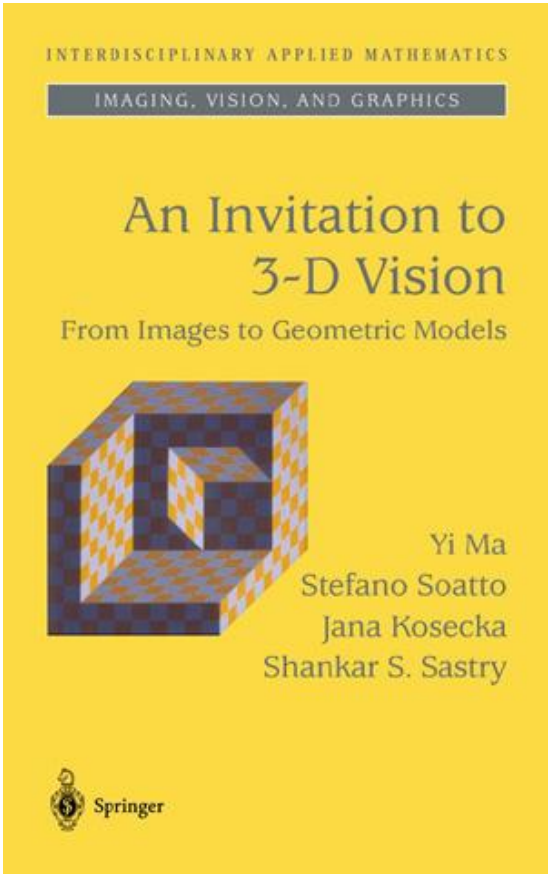


ORB-SLAM2 (2016)

What is 3D Vision?

- Reference Books





What is 3D Vision?

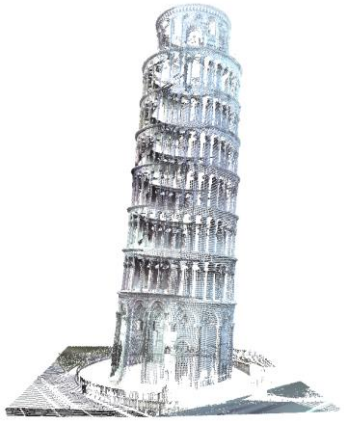
- OpenCV (Open Source Computer Vision)
  - `calib3d` Module: Camera Calibration and 3D Reconstruction

- OpenCV API Reference
  - Introduction
  - core. The Core Functionality
  - imgproc. Image Processing
  - imgcodecs. Image file reading and writing
  - videoio. Media I/O
  - highgui. High-level GUI and Media I/O
  - video. Video Analysis
  - calib3d. Camera Calibration and 3D Reconstruction
  - features2d. 2D Features Framework
  - objdetect. Object Detection
  - ml. Machine Learning
  - flann. Clustering and Search in Multi-Dimensional Spaces
  - photo. Computational Photography
  - stitching. Images stitching
  - cuda. CUDA-accelerated Computer Vision
  - cudaarithm. CUDA-accelerated Operations on Matrices
  - cudabgsegm. CUDA-accelerated Background Segmentation
  - cudacodec. CUDA-accelerated Video Encoding/Decoding
  - cudafeatures2d. CUDA-accelerated Feature Detection and Description
  - cudafilters. CUDA-accelerated Image Filtering
  - cudaimgproc. CUDA-accelerated Image Processing
  - cudaoptflow. CUDA-accelerated Optical Flow
  - cudastereo. CUDA-accelerated Stereo Correspondence
  - cudawarping. CUDA-accelerated Image Warping
  - shape. Shape Distance and Matching
  - superres. Super Resolution
  - videostab. Video Stabilization
  - viz. 3D Visualizer

cf. All examples in this tutorial are mostly **less than 100 lines** and based on **recent OpenCV (> 3.0.0)**.

What is 3D Vision?

image



depth image, range data, point cloud, polygon mesh, …

Perspective Camera

3D Vision **v.s.** **3D Data Processing**

RGB-D Camera
(Stereo, Structured Light, ToF, Light Field)

Omni-directional Camera

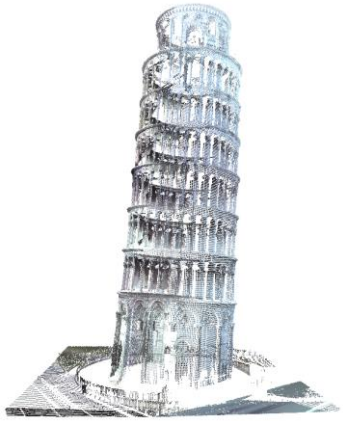
Range Sensor
(LiDAR, RADAR)

What is 3D Vision?

OpenCV

pcl OPEN3D

image



depth image, range data, point cloud, polygon mesh, …

Perspective Camera

3D Vision **v.s.** **3D Data Processing**

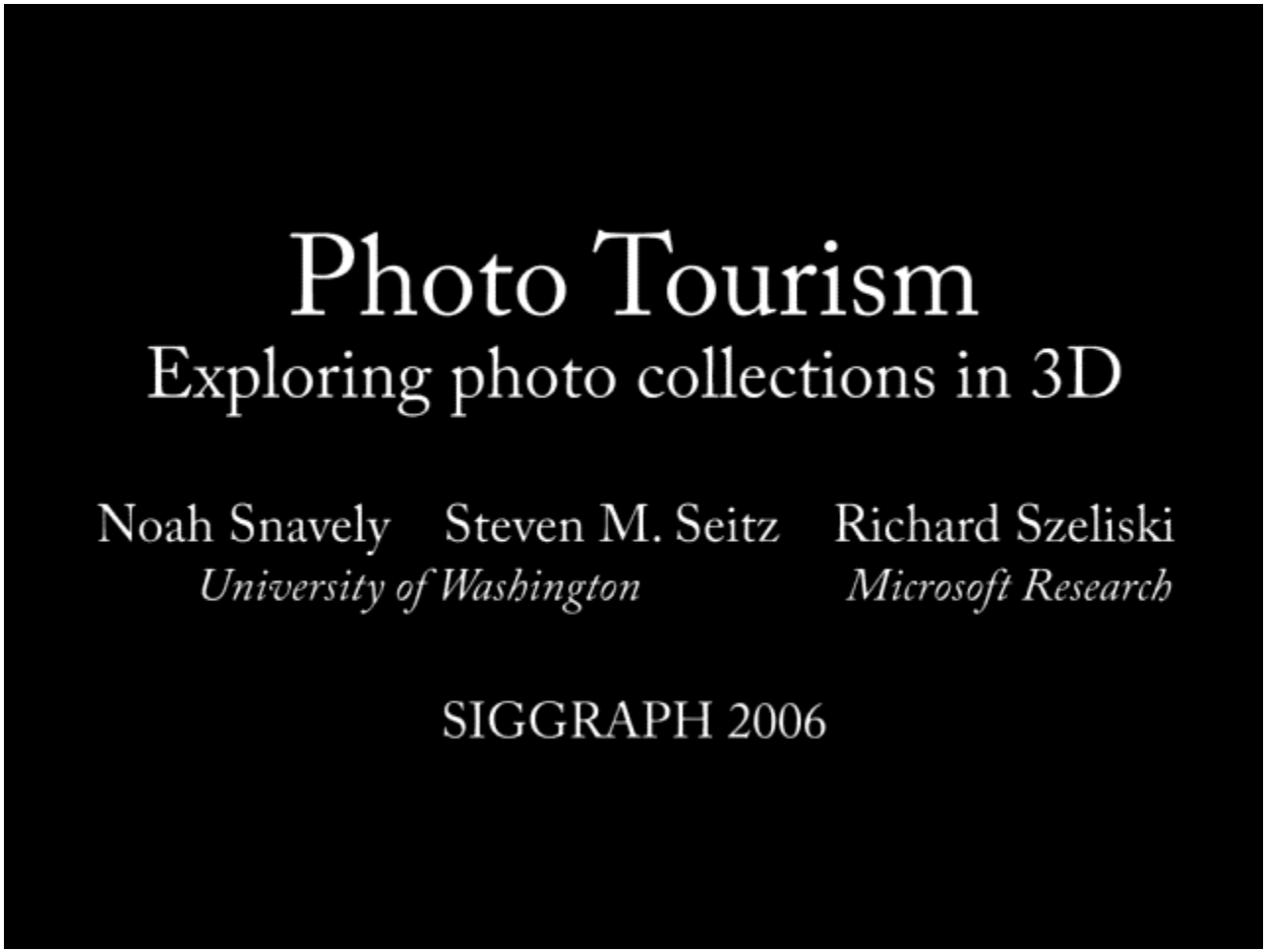
RGB-D Camera
(Stereo, Structured Light, ToF, Light Field)

Omni-directional Camera

Range Sensor
(LiDAR, RADAR)

Applications: Photo Browsing

- Photo Tourism (2006)



[Reference] Snavely et al., **Photo Tourism: Exploring Photo Collections in 3D**, SIGGRAPH, 2006

Applications: 3D Reconstruction

- Building Rome in a Day (2009)


photo by Ceci And Brandon

2106 images, 819242 points
click here to see more views


photo by TORISFERICK

1936 images, 656699 points
click here to see more views


photo by act2win

1815 images, 422593 points
click here to see more views


photo by angle_in_soul

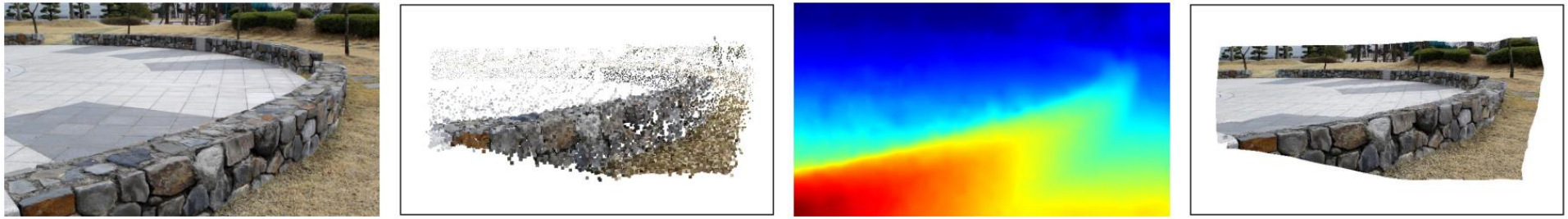
1381 images, 499044 points
click here to see more views

YouTube

[Reference] Agarwal et al., **Building Rome in a Day**, ICCV, 2009

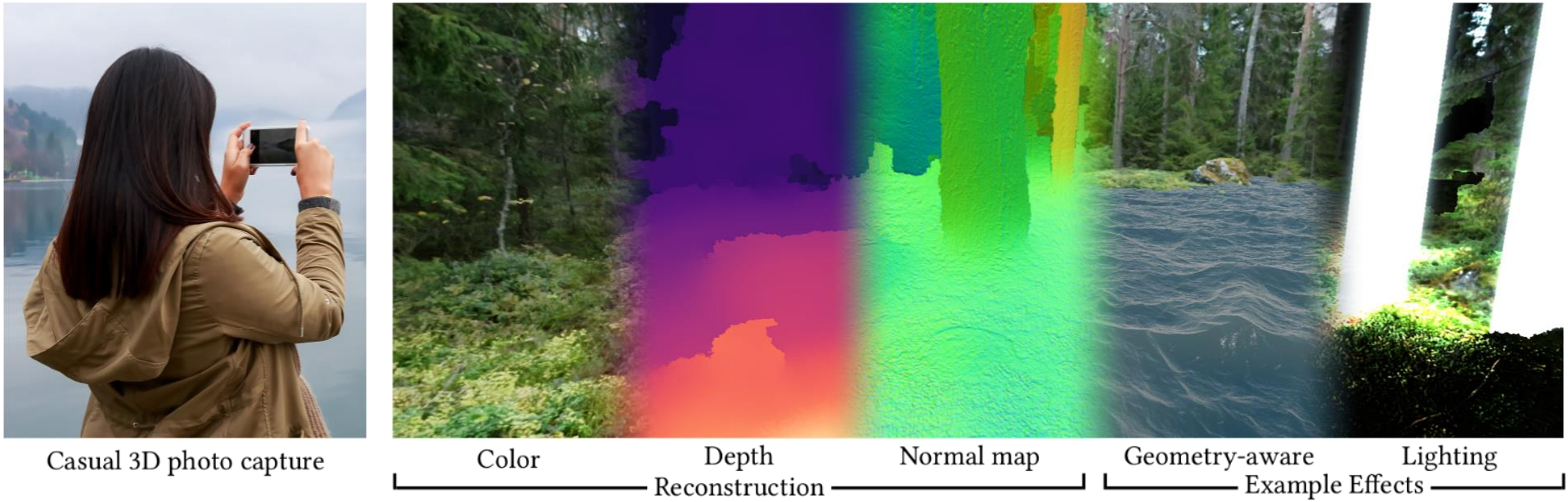
Applications: 3D Reconstruction from Cellular Phones

- Structure from Small Motion (SfSM; 2015)



(a) Reference images (b) SfSM results (c) Depth maps (d) Our 3D meshes

- Casual 3D Photography (2017)



Casual 3D photo capture

[Reference] Im et al., **High Quality Structure from Small Motion for Rolling Shutter Cameras**, ICCV, 2015
[Reference] Hedman et al., **Causual 3D Photography**, SIGGRAPH Asia, 2017

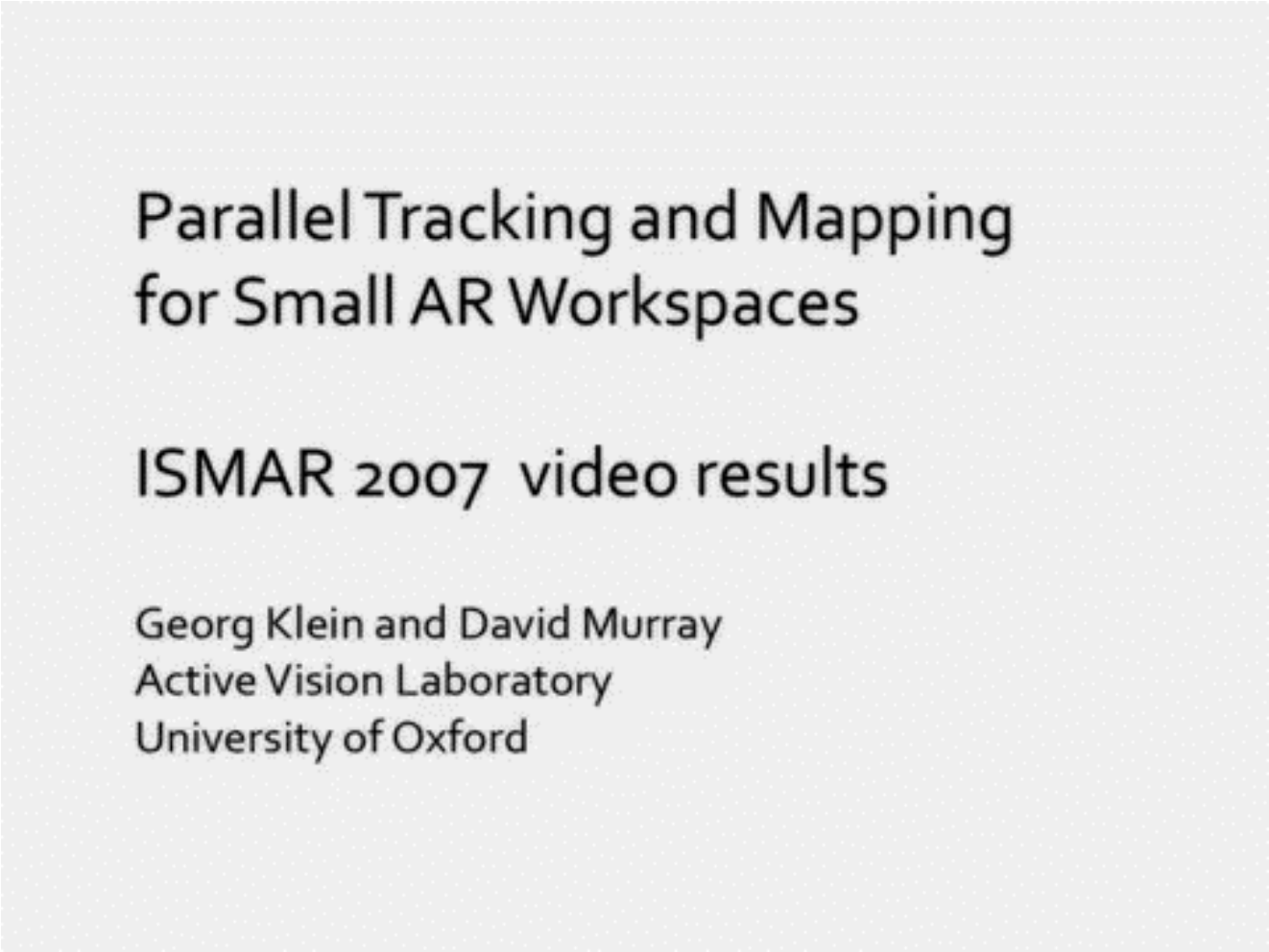
Applications: Real-time Visual SLAM

- ORB-SLAM (2014)

[Reference] Mur-Artal et al., **ORB-SLAM: A Versatile and Accurate Monocular SLAM System**, T-RO, 2015

Applications: Augmented Reality

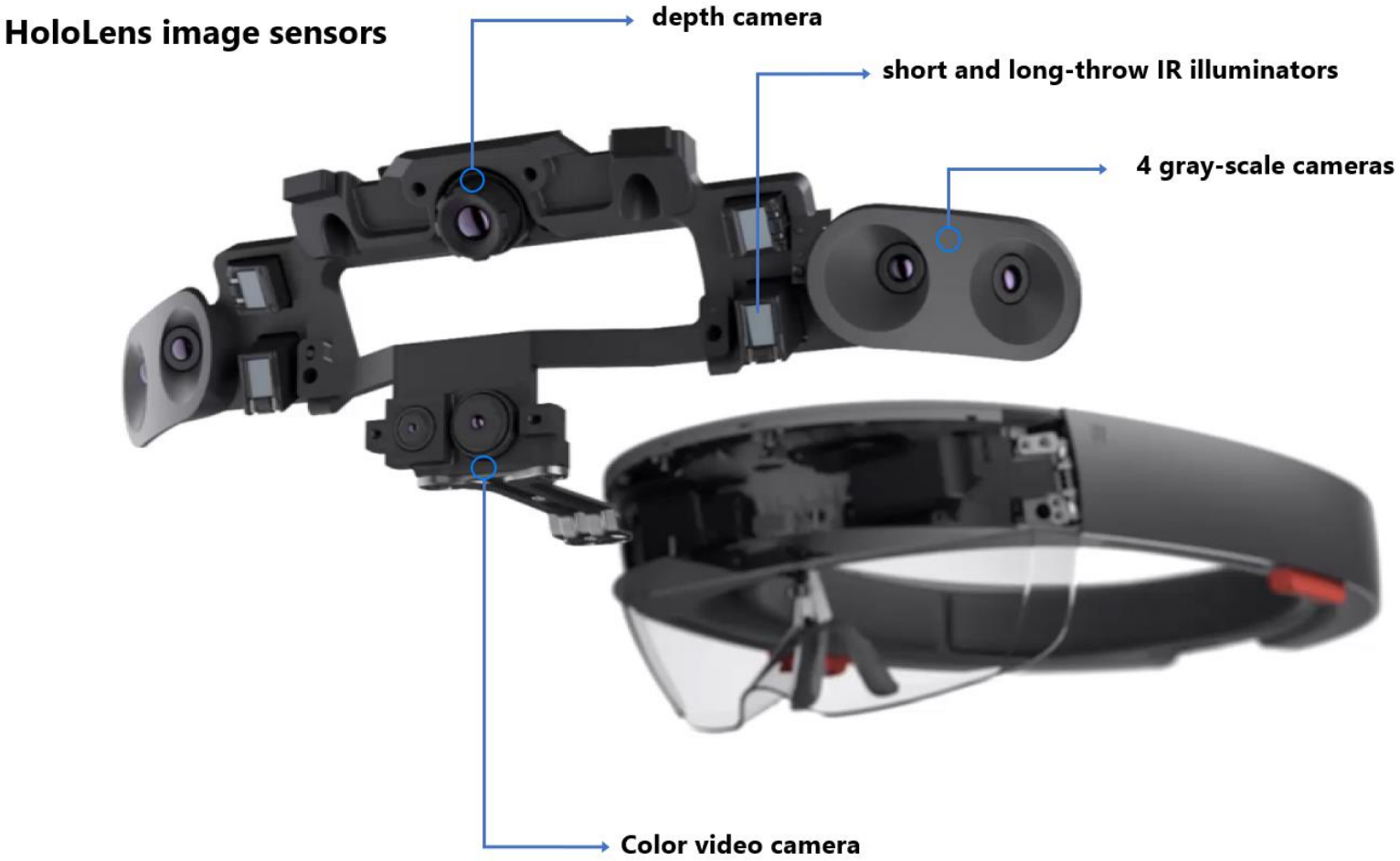
- PTAM: Parallel Tracking and Mapping (2007)



[Reference] Klein and Murray, **Parallel Tracking and Mapping for Small AR Workspaces**, ISMAR, 2007

Applications: Mixed Reality

- Microsoft Hololens 2 (2019)
  - Head tracking: 4 x visible light cameras



An image is from SlashGear (https://www.slashgear.com/microsoft-hololens-image-sensors-windows-10-update-18534584/)

Applications: Virtual Reality

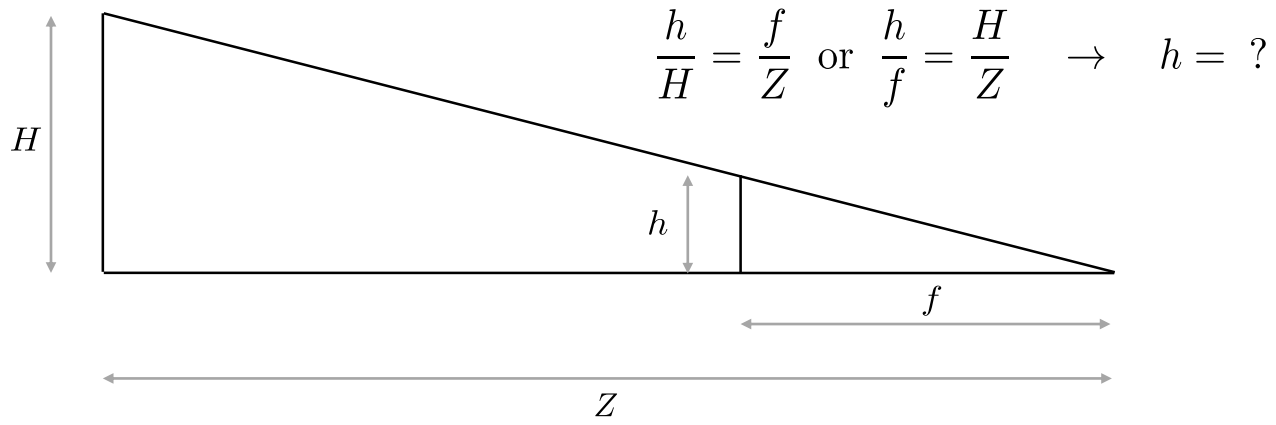
- Oculus Quest (2019)

An image is from TechSpot (https://www.techspot.com/products/audio-video/oculus-quest.202424/)

Table of Contents

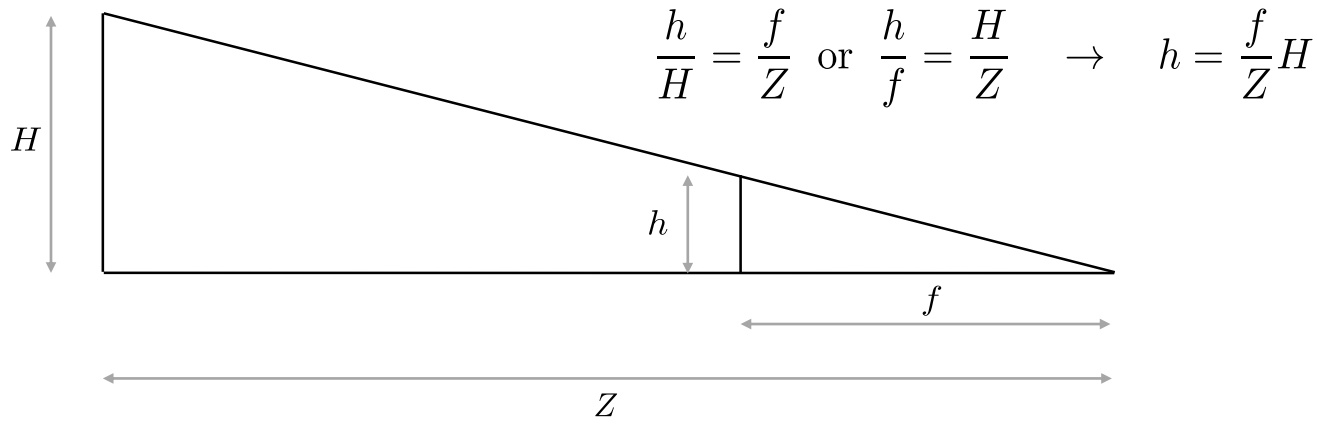
Getting Started with 2D

- Similarity



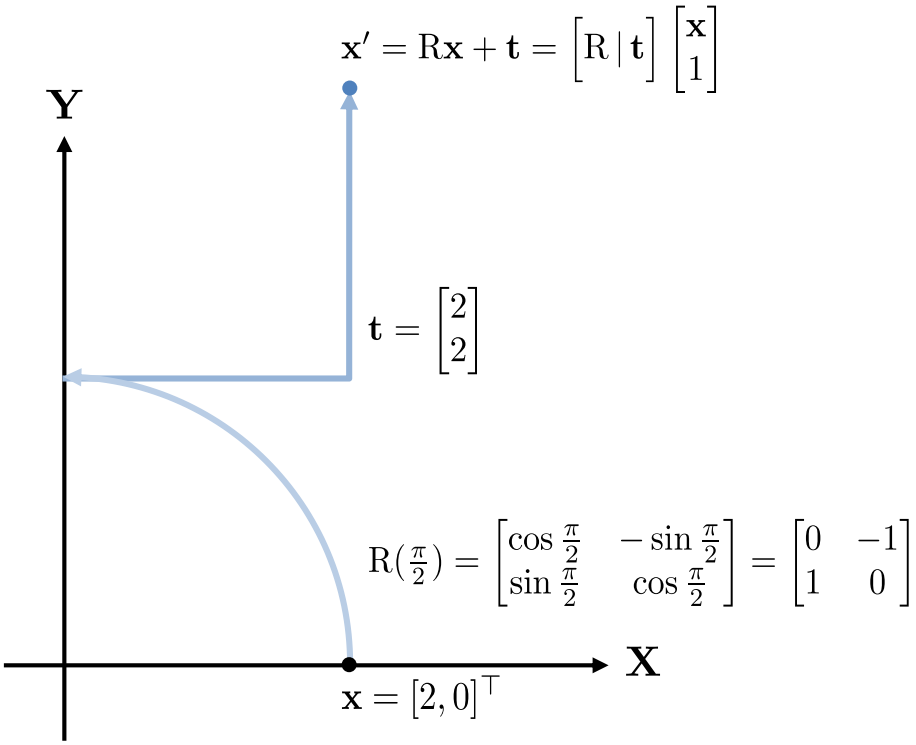
Getting Started with 2D

- Similarity



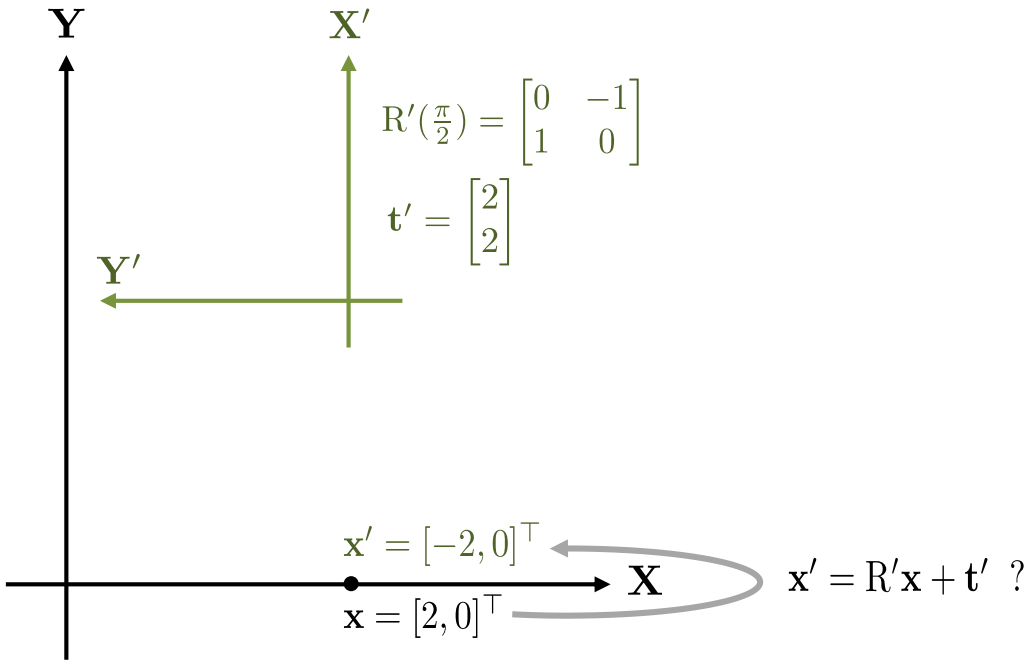
Getting Started with 2D

- Point transformation



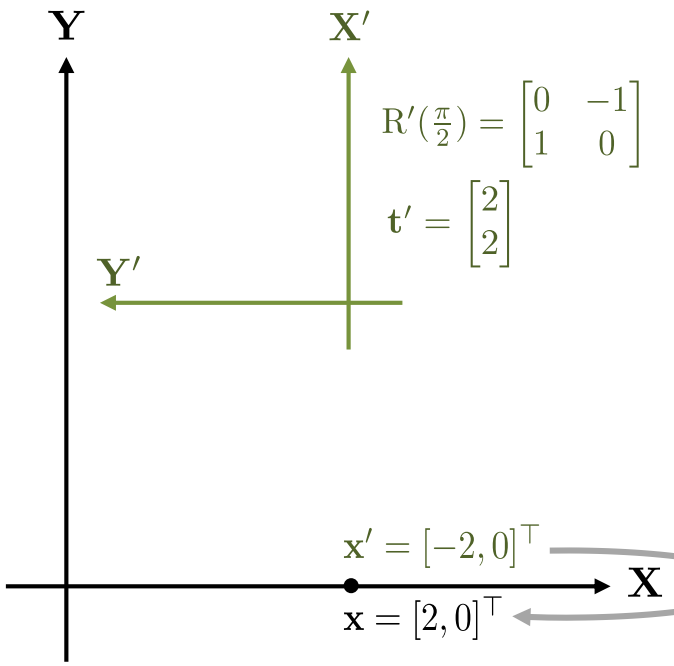
Getting Started with 2D

- Coordinate transformation



Getting Started with 2D

- Coordinate transformation is the **inverse** of point transformation.



$$x = R'x' + t'$$

$$R'^\top(x - t') = x'$$

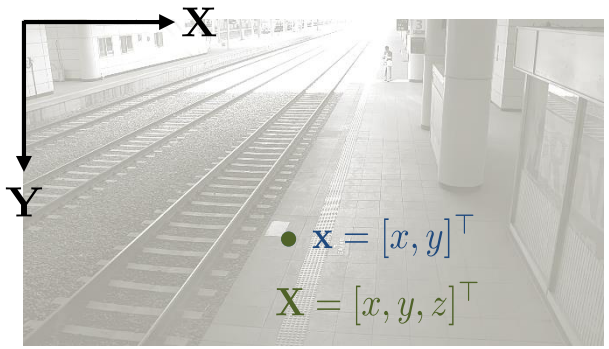
$$x' = Rx + t$$

$$(R = R'^\top, \quad t = -R'^\top t')$$

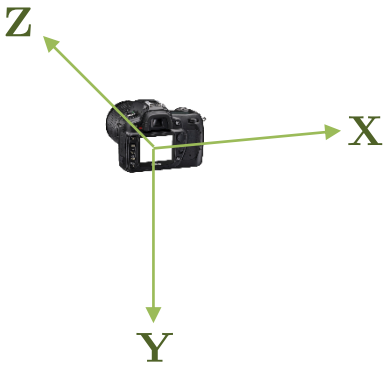
$$R(-\tfrac{\pi}{2}) = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \quad t = \begin{bmatrix} -2 \\ 2 \end{bmatrix}$$

Getting Started with 2D

- Image coordinate (unit: [pixel])

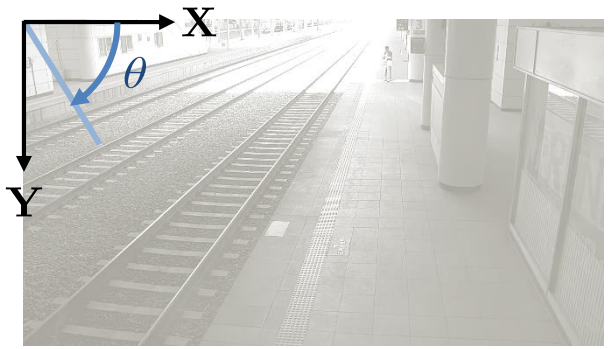


- Camera coordinate (unit: [meter])



Getting Started with 2D

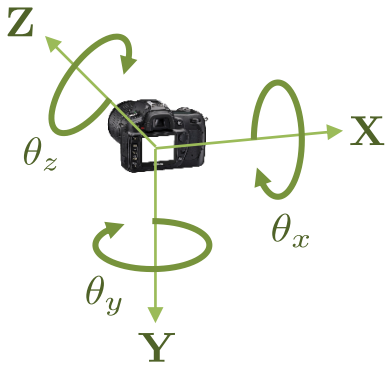
- 2D rotation matrix



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

cf. Properties of a rotation matrix
- $R^{-1} = R^{\intercal}$ (orthogonal matrix)
- $\det(R) = 1$

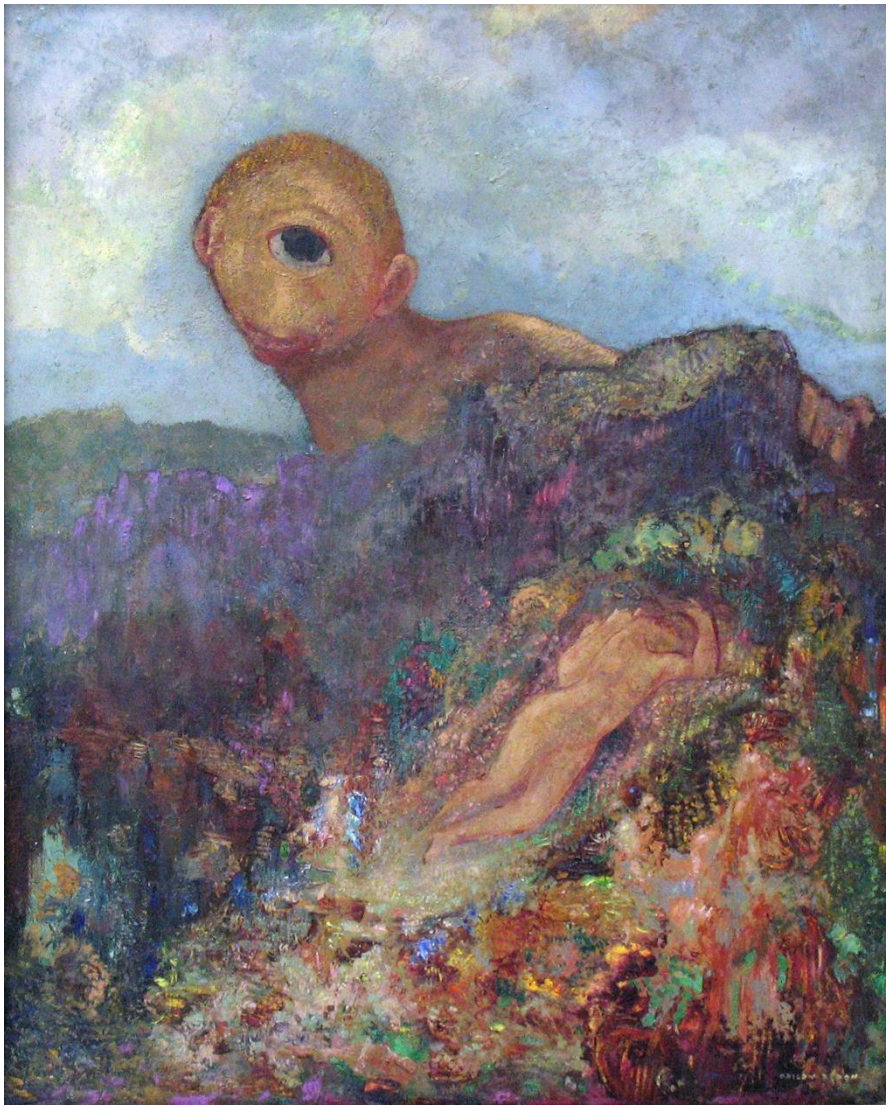
- 3D rotation matrix



	Cameras	Vehicles	Airplanes	Telescopes
θ_x	Tilt	Pitch	Attitude	Elevation
θ_y	Pan	Yaw	Heading	Azimuth
θ_z	Roll	Roll	Bank	Horizon

$$\mathrm{R}(\theta_x, \theta_y, \theta_z) = \mathrm{R}_z(\theta_z)\mathrm{R}_y(\theta_y)\mathrm{R}_x(\theta_x) = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix}$$

Single-view Geometry



Camera Projection Model

- Pinhole Camera Model
- Geometric Distortion

General 2D-3D Geometry

- Camera Calibration
- Absolute Camera Pose Estimation

The Cyclops, gouache and oil by Odilon Redon

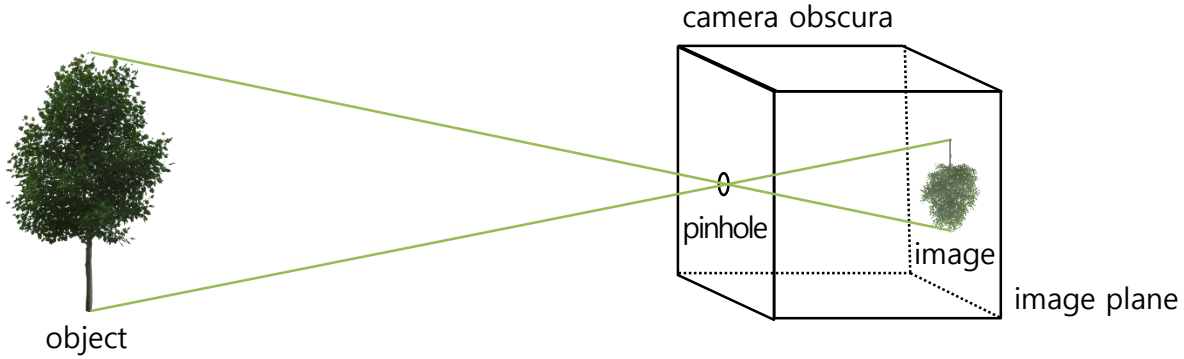
Camera Projection Model

- **Pinhole Camera Model**

A large-scale camera obscura
at San Francisco, California

A modern-day camera obscura

An Image in camera obscura
at Portslade, England



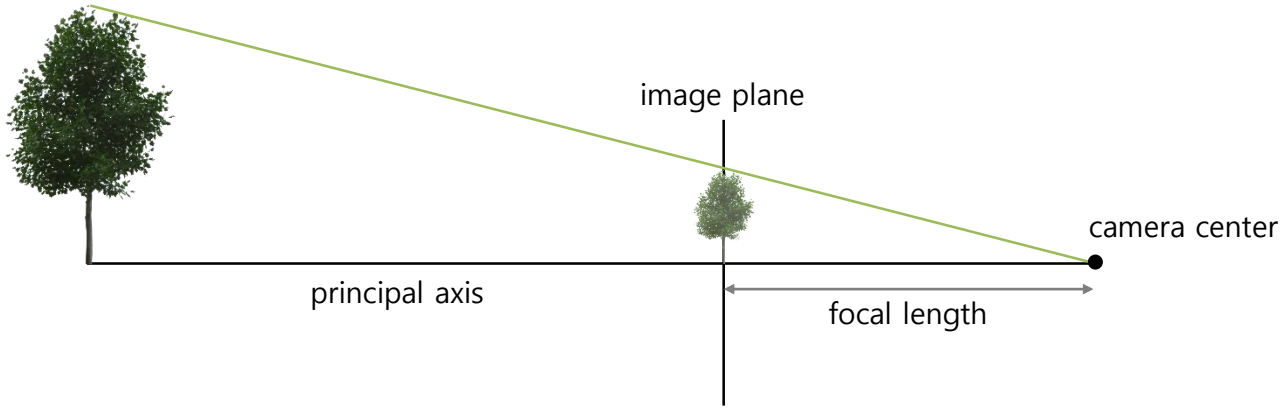
Images are from Wikipedia (https://en.wikipedia.org/wiki/Camera_obscura).

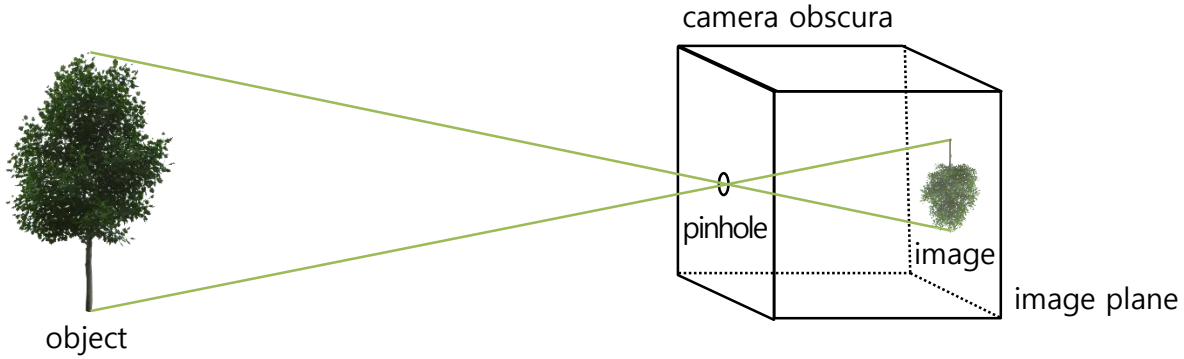
Camera Projection Model

- **Pinhole Camera Model**

cf. conclusion in advance

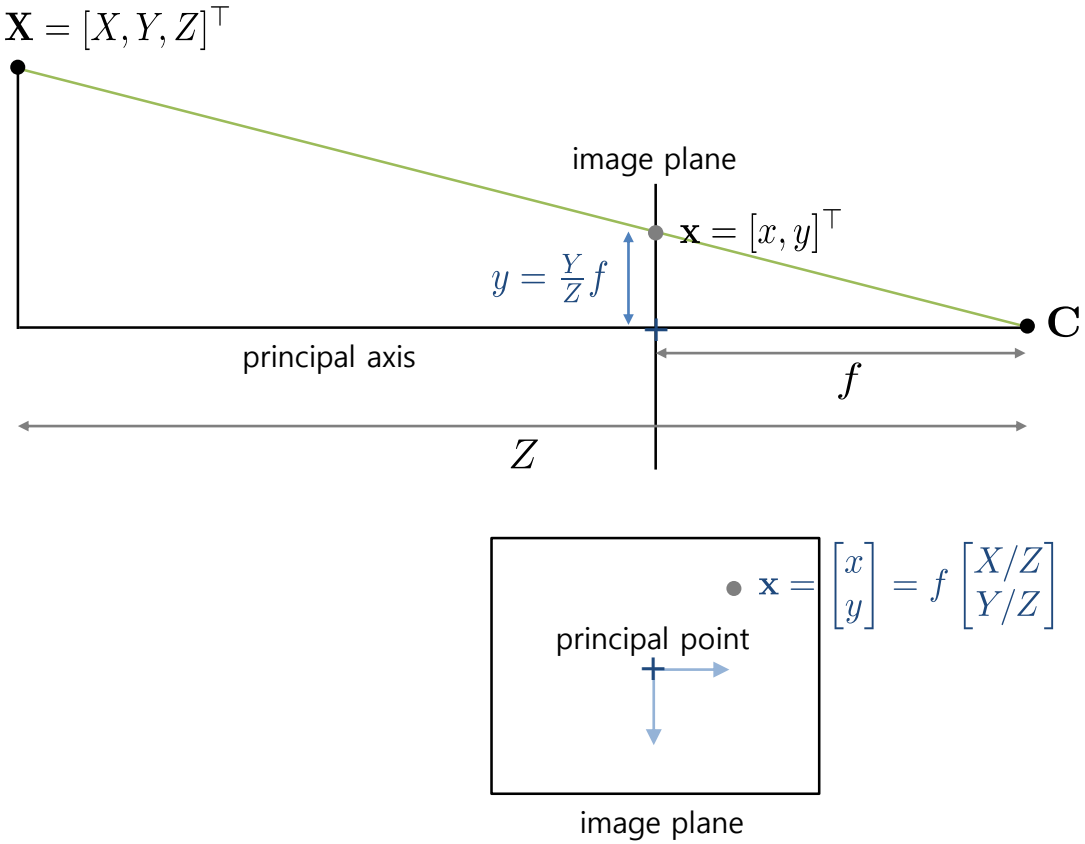
$$\mathbf{x} = \mathrm{P}\mathbf{X} \quad \left(\mathrm{P} = \mathrm{K}[\mathrm{R}\,|\,\mathbf{t}]\right)$$





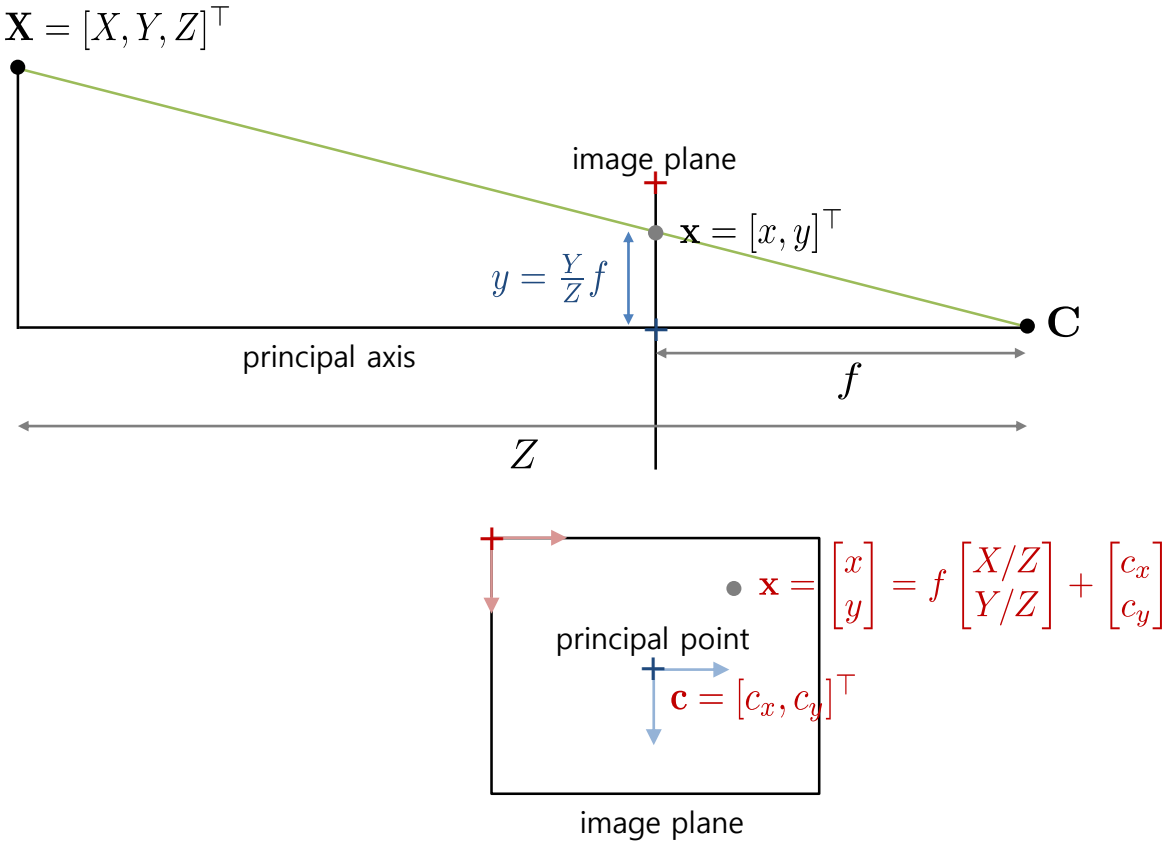
Camera Projection Model

- **Pinhole Camera Model**



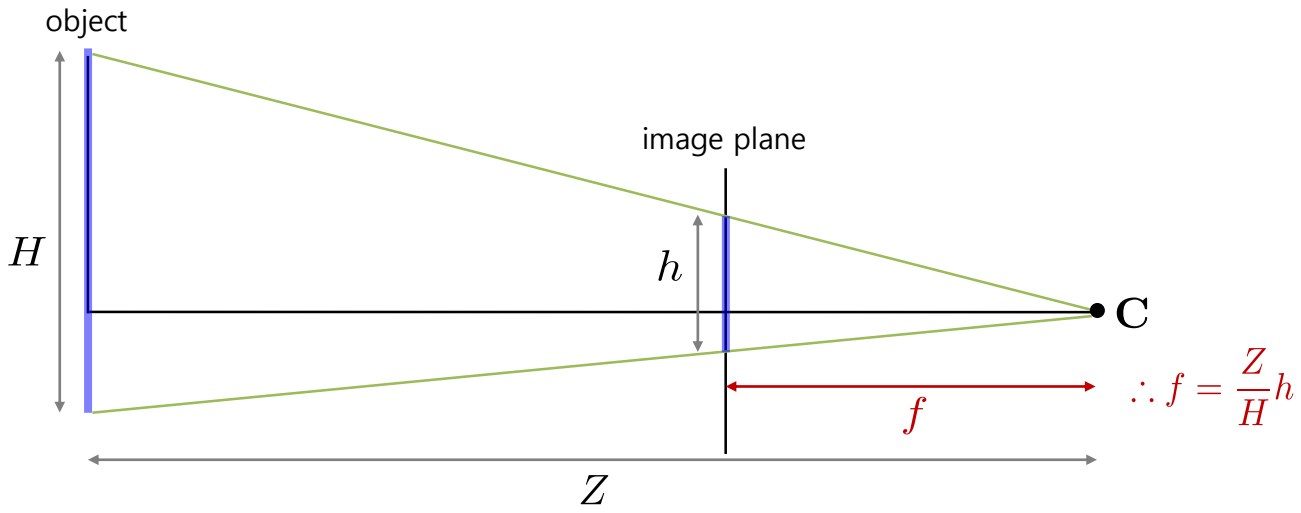
Camera Projection Model

- **Pinhole Camera Model**



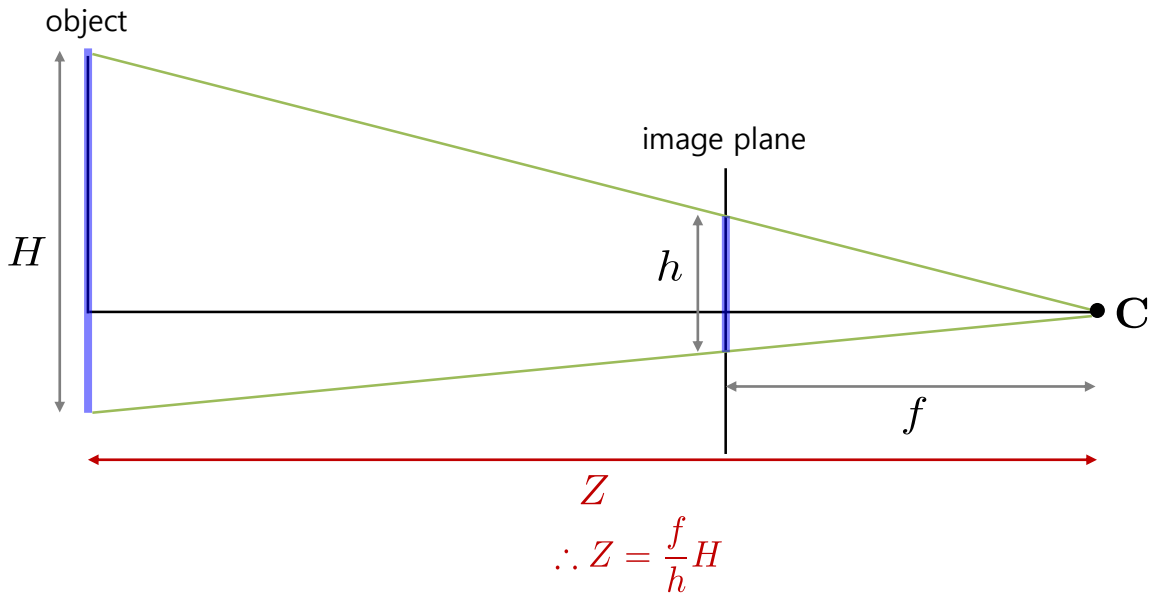
Camera Projection Model

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



Camera Projection Model

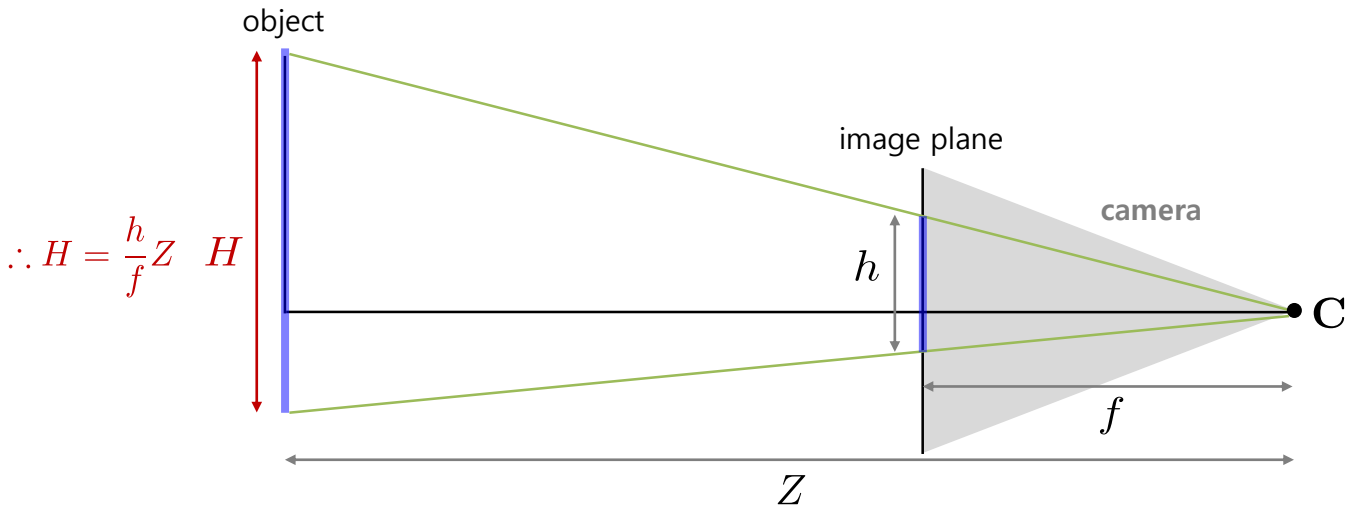
- Example: **Simple Distance Measurement**
  - Unknown: Object distance (Z) (unit: [m])
  - Given: The observed object height (h) on the image plane (unit: [pixel])
  - Assumptions
    - The object height (H) and focal length (f) are known.
    - The object is aligned with the image plane.



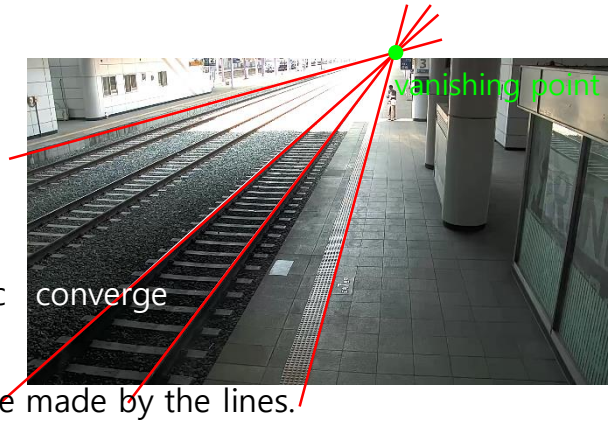
$$\therefore Z = \frac{f}{h} H$$

Camera Projection Model

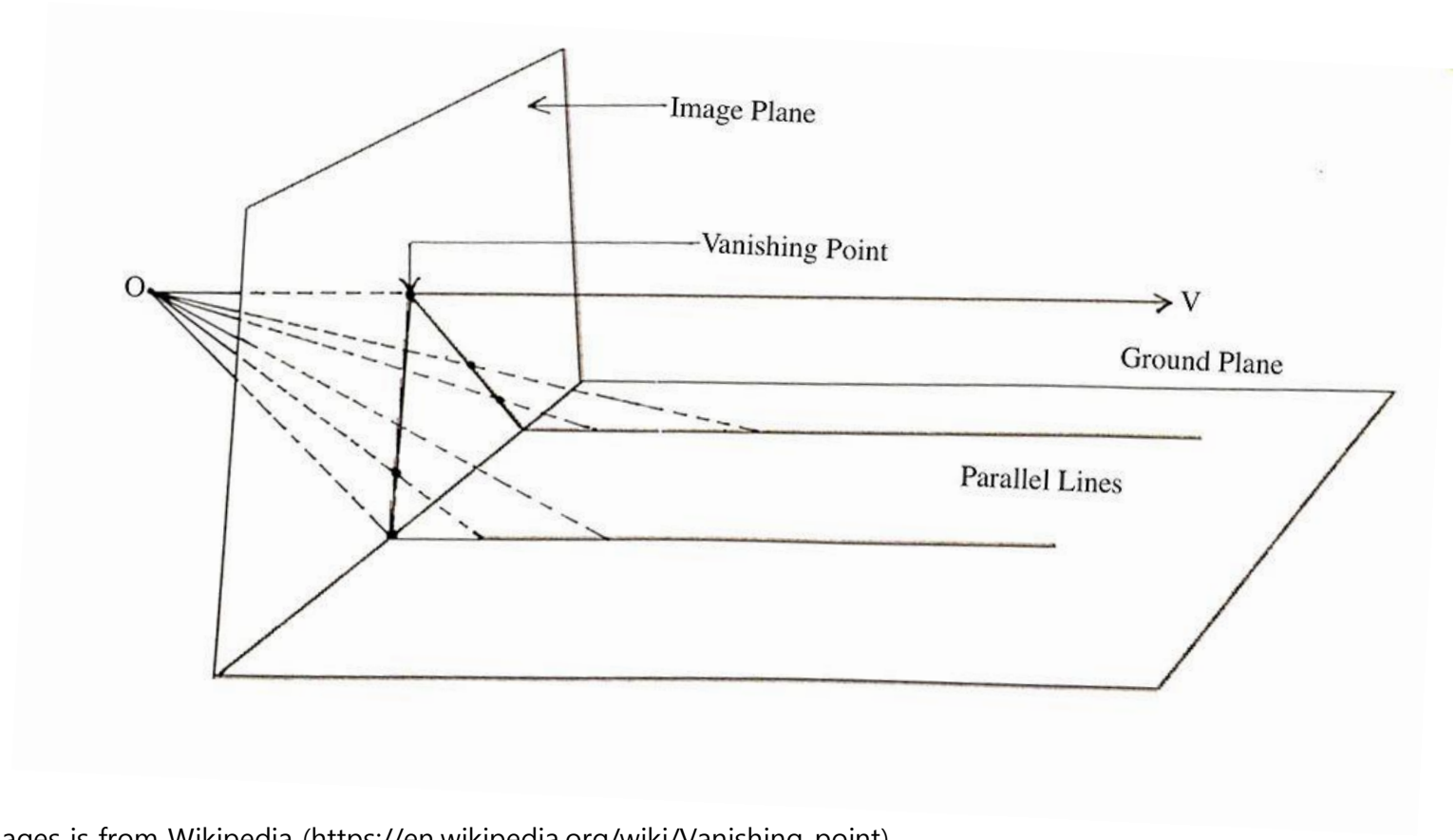
- Example: **Simple Height Measurement**
  - Unknown: Object height (H) (unit: [m])
  - Given: The observed object height (h) on the image plane (unit: [pixel])
  - Assumptions
    - The object distance (Z) from the camera and focal length (f) are known.
    - The object is aligned with the image plane.



Camera Projection Model

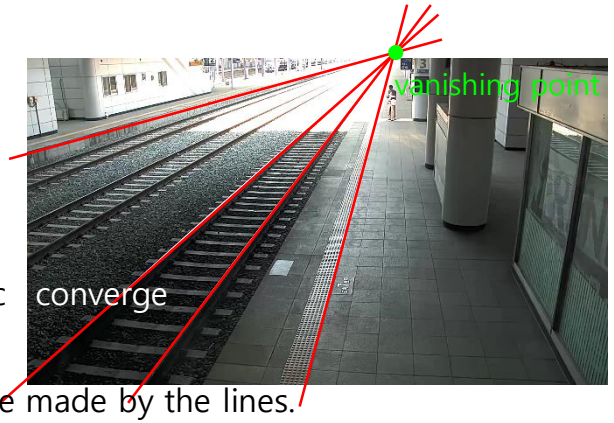


- **Vanishing Points**
  - A point on the image plane where mutually parallel lines in 3D spac
    - A vector to the vanishing point is parallel to the lines.
    - A vector to the vanishing point is parallel to the reference plane made by the lines.

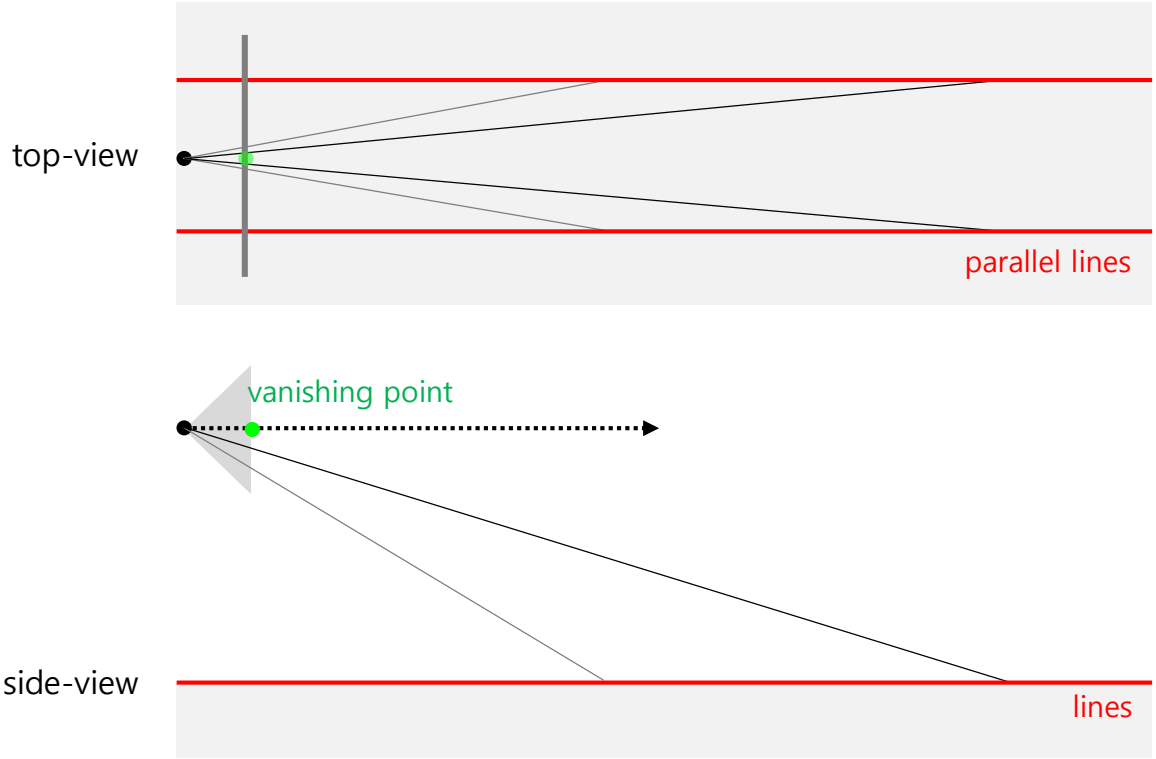


An images is from Wikipedia (https://en.wikipedia.org/wiki/Vanishing_point).

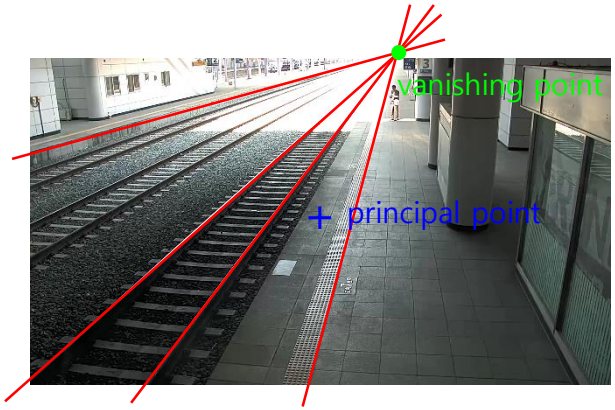
Camera Projection Model



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



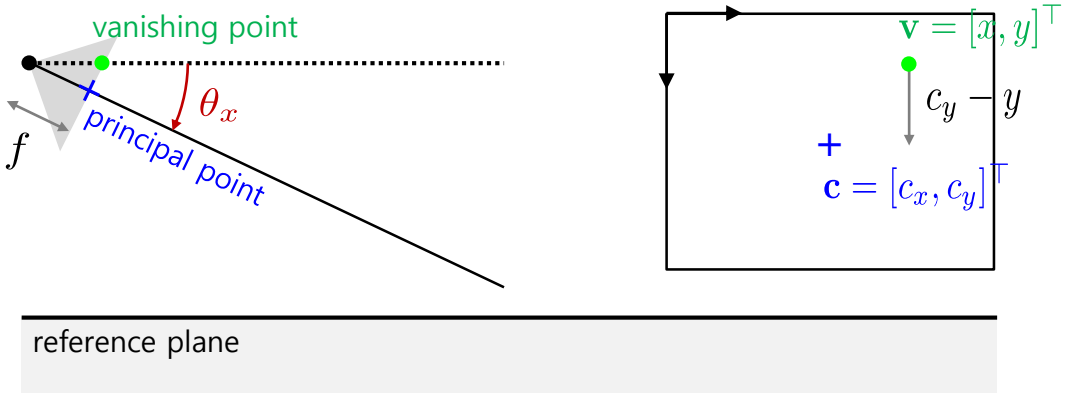
Camera Projection Model



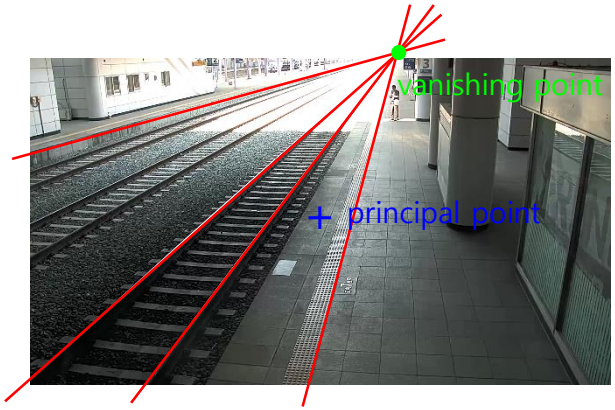
- Example: **Simple Visual Compass**
  - Unknown: Tilt angle (θ_x) (unit: [rad])
  - Given: A vanishing point (x, y) from the reference plane
  - Assumptions
    - The focal length (f) is known.
    - The principal point (c_x, c_y) is known or selected as the center of images.
    - The camera has no roll, $\theta_z = 0$.

cf. The tilt angle in this page is defined as the opposite direction of the common notation.

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

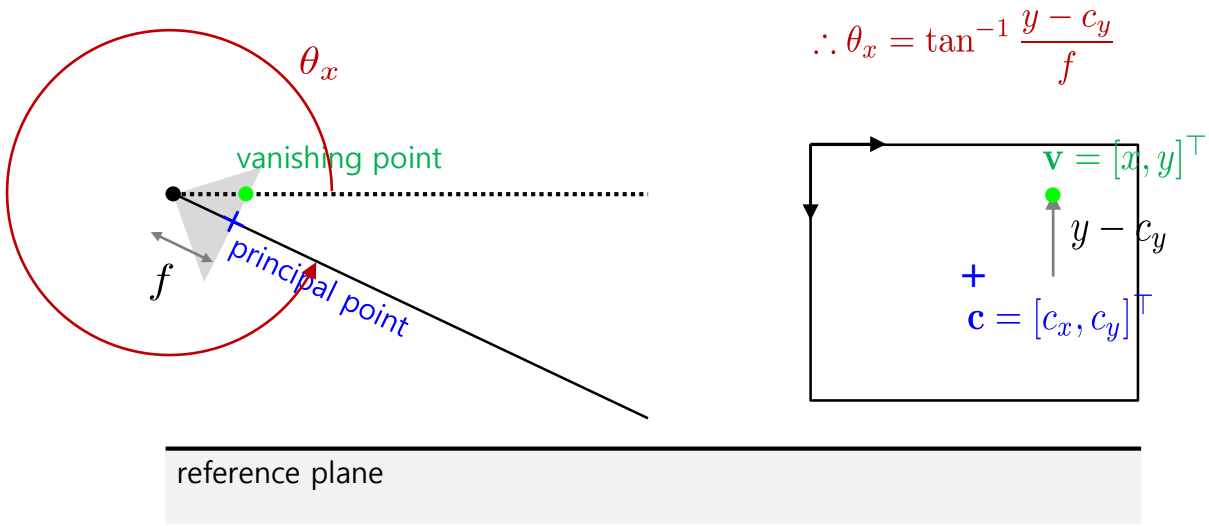


Camera Projection Model



- Example: **Simple Visual Compass**
  - Unknown: Tilt angle (θ_x) (unit: [rad])
  - Given: A vanishing point (x, y) from the reference plane
  - Assumptions
    - The focal length (f) is known.
    - The principal point (c_x, c_y) is known or selected as the center of images.
    - The camera has no roll, $\theta_z = 0$.

cf. Similarly, the pan angle (θ_y) w.r.t. rails also can be calculated using x instead of y.

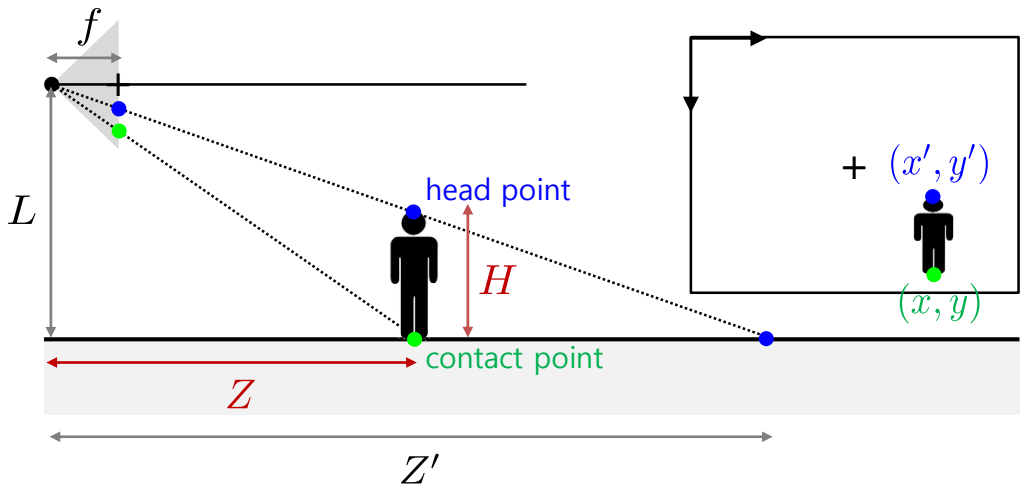


Camera Projection Model

- Example: **Object Localization and Measurement #1**
  - Unknown: Object position and height (unit: [m])
  - Given: The object's contact and head points on the image (unit: [pixel])
  - Assumptions
    - The focal length, principal points, and camera height, are known.
    - The camera is aligned to the reference plane.
    - The object is on the reference plane.

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

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



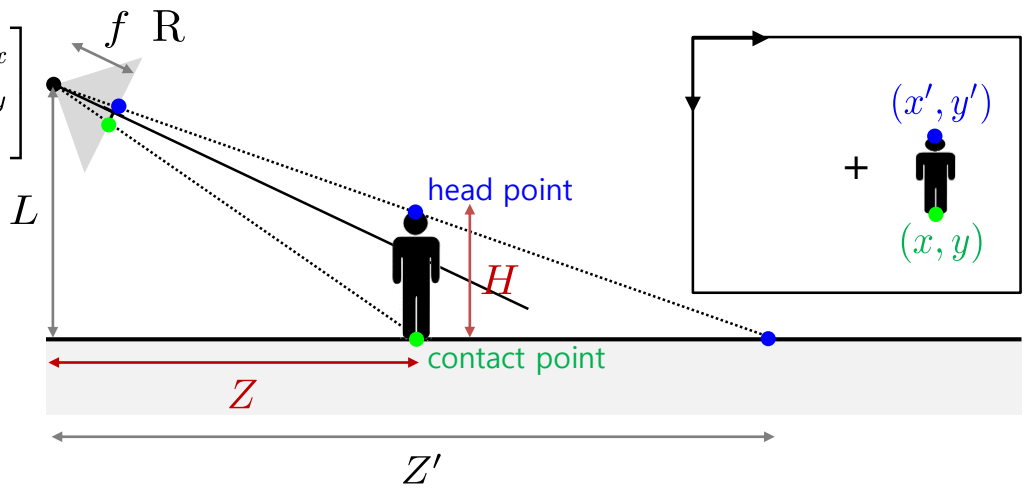
Camera Projection Model

- Example: **Object Localization and Measurement #2**
  - Unknown: Object position and height (unit: [m])
  - Given: The object's contact and head points on the image (unit: [pixel])
  - Assumptions
    - The focal length, principal points, and camera height, are known.
    - ~~The camera is aligned to the reference plane.~~ The camera orientation is known.
    - The object is on the reference plane.

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

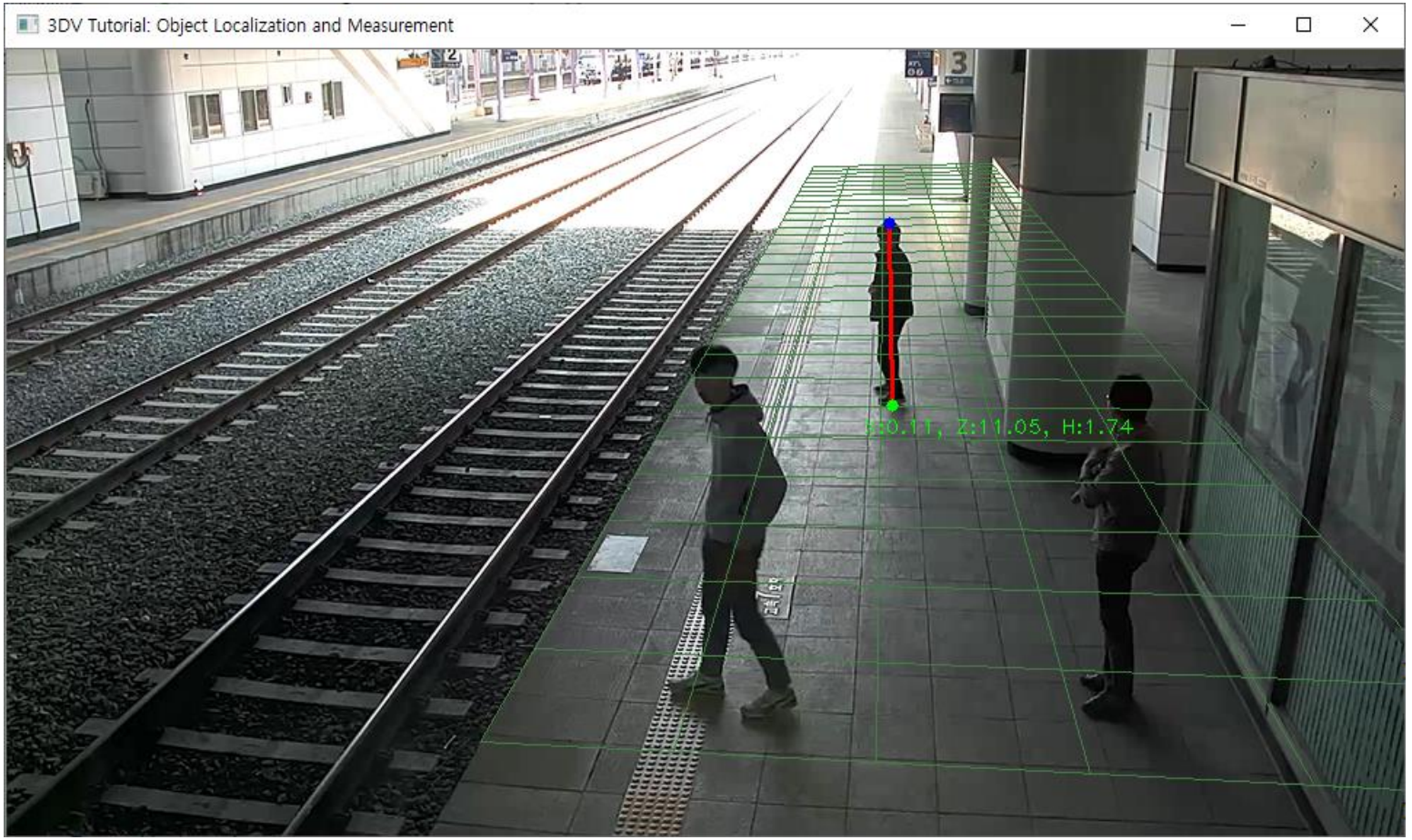
$\because$ same camera center

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



Camera Projection Model

- Example: **Object Localization and Measurement #2** `[object_localization.cpp]`




```cpp
#include "opencv2/opencv.hpp"

...

int main()
{
    const char* input = "data/daejeon_station.png";
    double f = 810.5, cx = 480, cy = 270, L = 3.31;
    cv::Point3d cam_ori(DEG2RAD(-18.7), DEG2RAD(-8.2), DEG2RAD(2.0));
    cv::Range grid_x(-2, 3), grid_z(5, 35);

    ...

    while (true)
    {
        cv::Mat image_copy = image.clone();
        if (drag.end.x > 0 && drag.end.y > 0)
        {
            // Calculate object location and height
            cv::Point3d c = R.t() * cv::Point3d(drag.start.x - cx, drag.start.y - cy, f);
            if (c.y < DBL_EPSILON) continue; // Skip the degenerate case (beyond the horizon)
            cv::Point3d h = R.t() * cv::Point3d(drag.end.x - cx, drag.end.y - cy, f);
            double Z = c.z / c.y * L, X = c.x / c.y * L, H = (c.y / c.z - h.y / h.z) * Z;

            // Draw head/contact points and location/height
            cv::line(image_copy, drag.start, drag.end, cv::Vec3b(0, 0, 255), 2);
            cv::circle(image_copy, drag.end, 4, cv::Vec3b(255, 0, 0), -1);
            cv::circle(image_copy, drag.start, 4, cv::Vec3b(0, 255, 0), -1);
            cv::putText(image_copy, cv::format("X:%.2f, Z:%.2f, H:%.2f", X, Z, H), ...);
        }

        // Show the image
        cv::imshow("3DV Tutorial: Object Localization and Measurement", image_copy);
        int key = cv::waitKey(1);
        if (key == 27) break; // 'ESC' key: Exit
    }
    return 0;
}
```

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

$$Z = \frac{\bar{z}}{\bar{y}} L$$

$$X = \frac{\bar{x}}{\bar{y}} L$$

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

Camera Projection Model

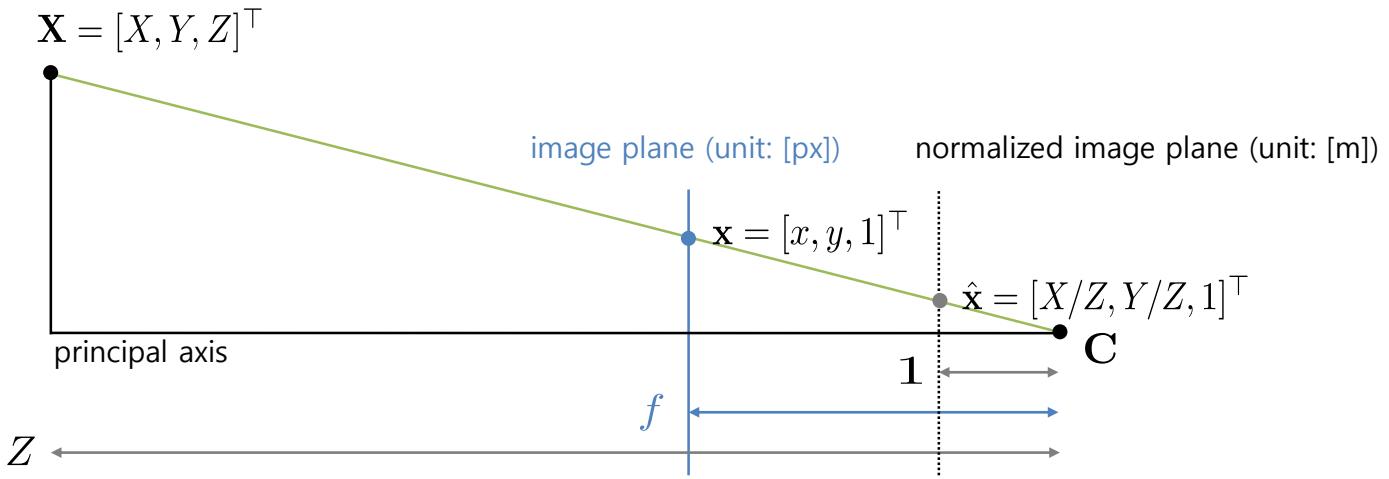
- **Camera Matrix** K

$$\bullet\ \mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} = f \begin{bmatrix} X/Z \\ Y/Z \end{bmatrix} + \begin{bmatrix} c_x \\ c_y \end{bmatrix} \longrightarrow \mathbf{x} = \mathrm{K}\hat{\mathbf{x}}$$

$$\mathrm{K} = \begin{bmatrix} f & 0 & w/2 \\ 0 & f & h/2 \\ 0 & 0 & 1 \end{bmatrix} \quad \text{specialized}$$

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

$$\mathrm{K} = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \quad \text{generalized}$$

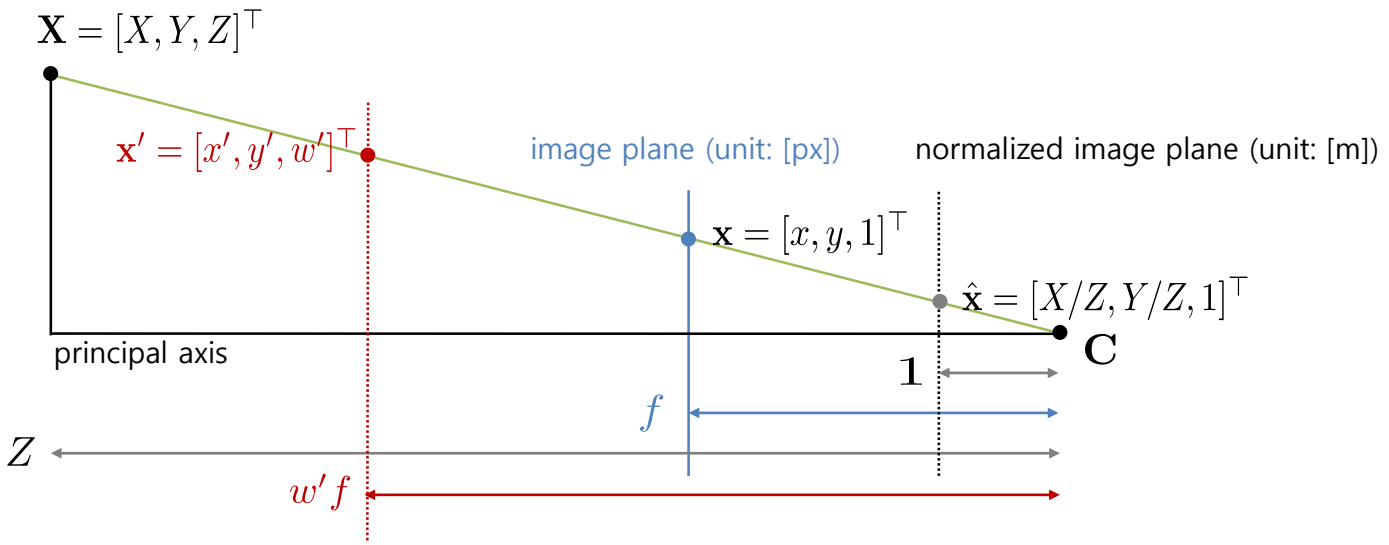


Camera Projection Model

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

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

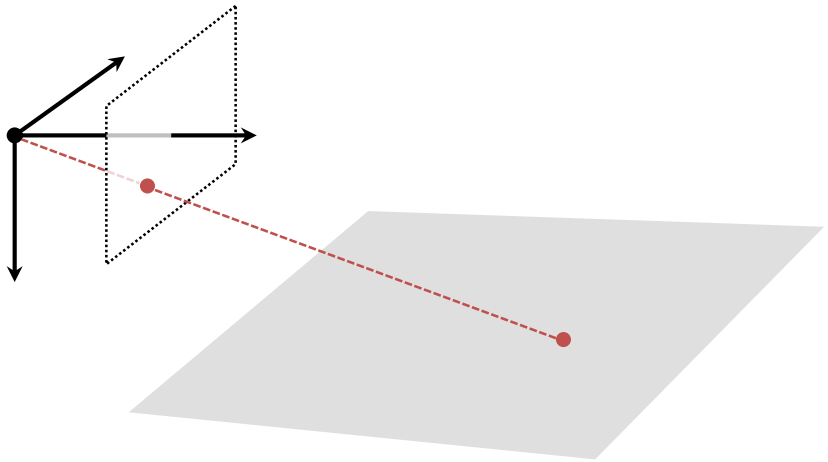
$$\mathbf{x}' = \mathrm{K}\mathbf{X} \quad \text{where} \quad \mathrm{K} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}, \ \mathbf{x}' = \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix}, \ \text{and} \ \mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

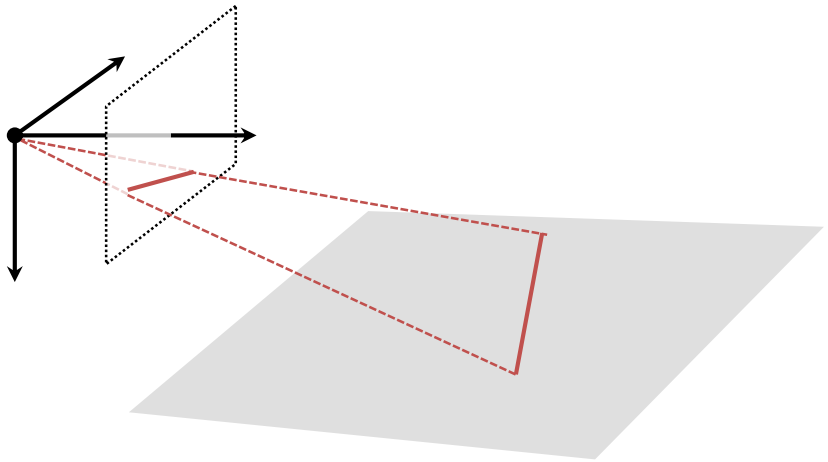


Camera Projection Model

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

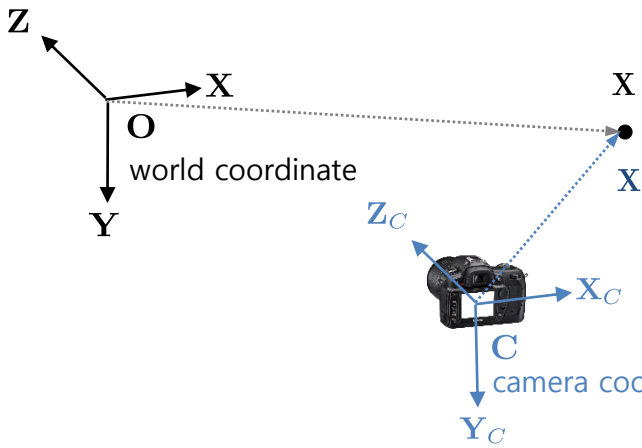
    } Duality of a point and line
  - …





Camera Projection Model

- **Projection Matrix** P
  - If a 3D point is not based on the camera coordinate, the point should be transformed.



3x4 matrix

$$\mathbf{X}' = \begin{bmatrix} \mathrm{R} \mid \mathbf{t} \end{bmatrix} \mathbf{X} \quad \text{where} \quad \mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$\mathrm{x} = \mathrm{K}\mathbf{X}'$$

$$\mathbf{x} = \mathrm{K}\begin{bmatrix} \mathrm{R} \mid \mathbf{t} \end{bmatrix} \mathbf{X}$$

$$\mathbf{x} = \mathrm{P}\mathbf{X} \quad \text{where} \quad \mathrm{P} = \mathrm{K}\begin{bmatrix} \mathrm{R} \mid \mathbf{t} \end{bmatrix}$$

3x4 matrix

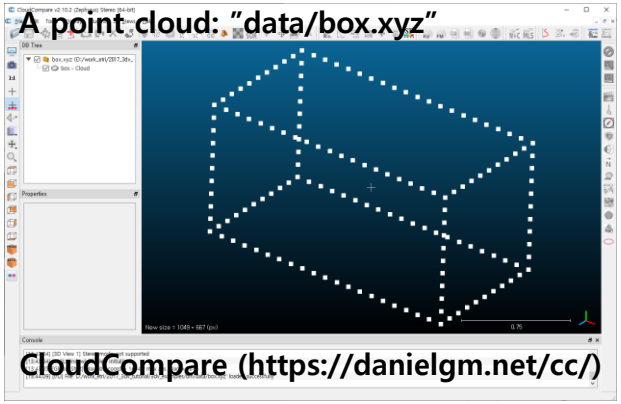
  - The camera pose can be derived from $\mathrm{R}^\intercal$ and $-\mathrm{R}^\intercal\mathbf{t}$.

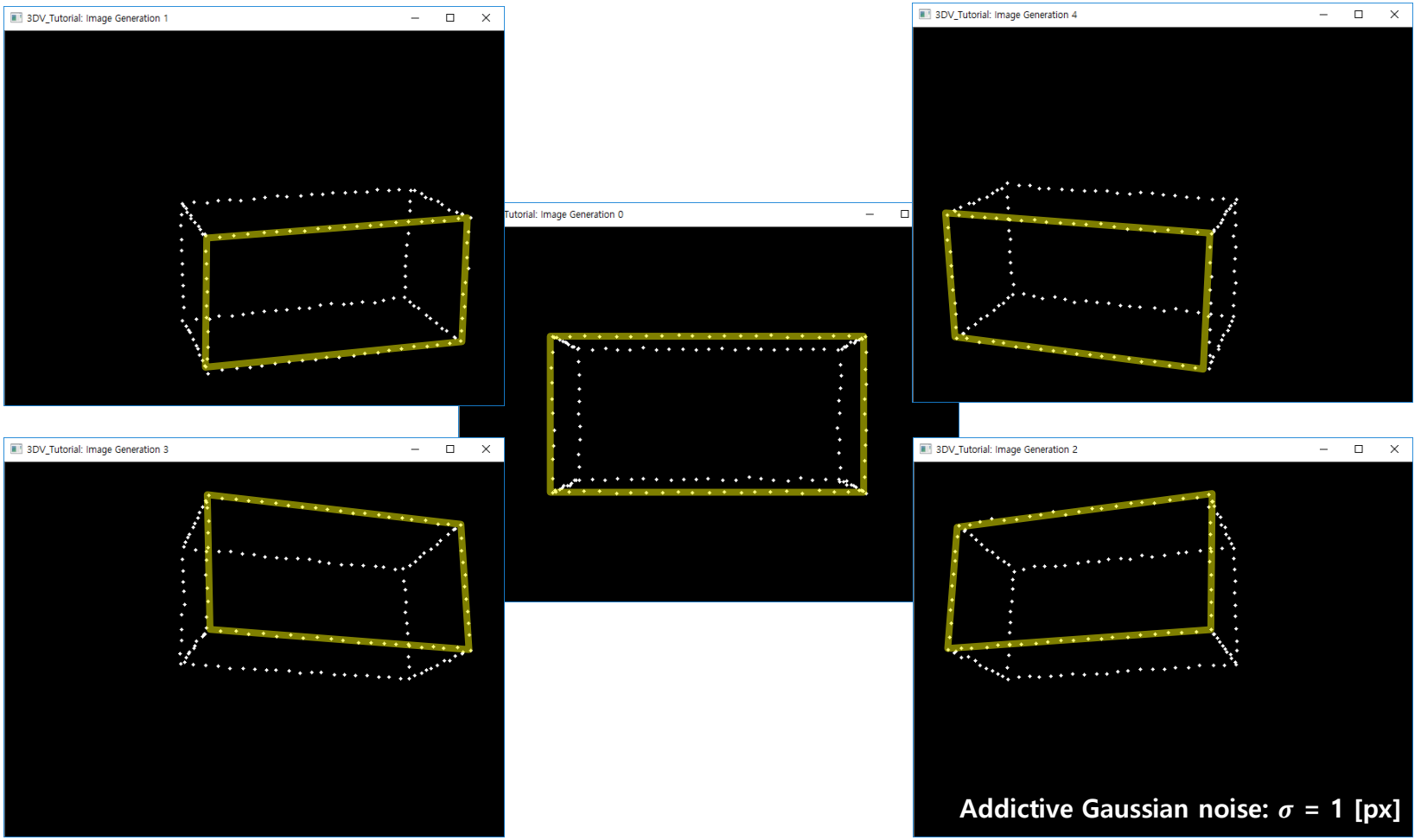
$$\text{c.f.} \quad \begin{bmatrix} \mathrm{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathrm{R}^\top & -\mathrm{R}^\top\mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$$

- **Camera Parameters** ~ projection matrix
  - **Intrinsic parameters** ~ camera matrix
    - e.g. focal length, principle point, skew, distortion coefficient, …
  - **Extrinsic parameters** ~ point transformation
    - e.g. rotation and translation

Camera Projection Model

- Example: **Image Formation** `[image_formation.cpp]`





Camera Projection Model

- Example: **Image Formation** [image_formation.cpp]

```
1.  #include "opencv2/opencv.hpp"

2.  #define Rx(rx)      (cv::Mat_<double>(3, 3) << 1, 0, 0, 0, cos(rx), -sin(rx), 0, sin(rx), cos(rx))
3.  #define Ry(ry)      (cv::Mat_<double>(3, 3) << cos(ry), 0, sin(ry), 0, 1, 0, -sin(ry), 0, cos(ry))
4.  #define Rz(rz)      (cv::Mat_<double>(3, 3) << cos(rz), -sin(rz), 0, sin(rz), cos(rz), 0, 0, 0, 1)

5.  int main()
6.  {
7.      // The given camera configuration: focal length, principal point, image resolution, position, and orientation
8.      double f = 1000, cx = 320, cy = 240, noise_std = 1;
9.      cv::Size img_res(640, 480);
10.     std::vector<cv::Point3d> cam_pos = { cv::Point3d(0, 0, 0), cv::Point3d(-2, -2, 0), ... };
11.     std::vector<cv::Point3d> cam_ori = { cv::Point3d(0, 0, 0), cv::Point3d(-CV_PI / 12, CV_PI / 12, 0), ... };

12.     // Load a point cloud in the homogeneous coordinate
13.     FILE* fin = fopen("data/box.xyz", "rt");
14.     if (fin == NULL) return -1;
15.     cv::Mat X;
16.     while (!feof(fin))
17.     {
18.         double x, y, z;
19.         if (fscanf(fin, "%lf %lf %lf", &x, &y, &z) == 3) X.push_back(cv::Vec4d(x, y, z, 1));
20.     }
21.     fclose(fin);
22.     X = X.reshape(1).t(); // Convert to a 4 x N matrix
```

```
24.     // Generate images for each camera pose
25.     cv::Mat K = (cv::Mat_<double>(3, 3) << f, 0, cx, 0, f, cy, 0, 0, 1);
26.     for (size_t i = 0; i < cam_pos.size(); i++)
27.     {
28.         // Derive a projection matrix
29.         cv::Mat Rc = Rz(cam_ori[i].z) * Ry(cam_ori[i].y) * Rx(cam_ori[i].x);
30.         cv::Mat tc(cam_pos[i]);
31.         cv::Mat Rt;
32.         cv::hconcat(Rc.t(), -Rc.t() * tc, Rt);
33.         cv::Mat P = K * Rt;

34.         // Project the points (cf. OpenCV provides 'cv::projectPoints()' with consideration of distortion.)
35.         cv::Mat x = P * X;
36.         x.row(0) = x.row(0) / x.row(2);
37.         x.row(1) = x.row(1) / x.row(2);
38.         x.row(2) = 1;

39.         // Add Gaussian noise
40.         cv::Mat noise(2, x.cols, x.type());
41.         cv::randn(noise, cv::Scalar(0), cv::Scalar(noise_std));
42.         x.rowRange(0, 2) = x.rowRange(0, 2) + noise;

43.         // Show and store the points
44.         cv::Mat image = cv::Mat::zeros(img_res, CV_8UC1);
45.         for (int c = 0; c < x.cols; c++)
46.         {
47.             cv::Point p(x.col(c).rowRange(0, 2));
48.             if (p.x >= 0 && p.x < img_res.width && p.y >= 0 && p.y < img_res.height)
49.                 cv::circle(image, p, 2, 255, -1);
50.         }
51.         cv::imshow(cv::format("3DV_Tutorial: Image Formation %d", i), image);

52.         FILE* fout = fopen(cv::format("image_formation%d.xyz", i).c_str(), "wt");
53.         if (fout == NULL) return -1;
54.         for (int c = 0; c < x.cols; c++)
55.             fprintf(fout, "%f %f 1\n", x.at<double>(0, c), x.at<double>(1, c));
56.         fclose(fout);
57.     }

58.     cv::waitKey(0);
59.     return 0;
60. }
```

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

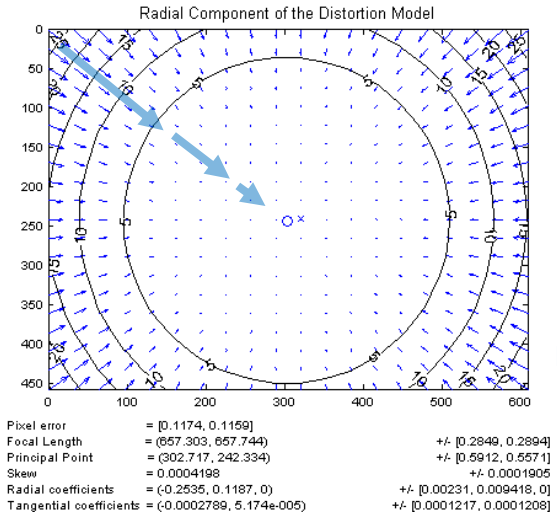
$$\because \begin{bmatrix} \mathrm{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathrm{R}^\top & -\mathrm{R}^\top \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$$

$$\because \begin{bmatrix} x \\ y \\ w \end{bmatrix} \sim \begin{bmatrix} x/w \\ y/w \\ 1 \end{bmatrix}$$

Camera Projection Model

- **Geometric Distortion**
  - Geometric distortion models are usually defined on the normalized image plane.
  - **Polynomial distortion model** (a.k.a. Brown-Conrady model)
    - Radial distortion: k_1, k_2, ...
    - Tangential distortion: p_1, p_2, ... (usually negligible)

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

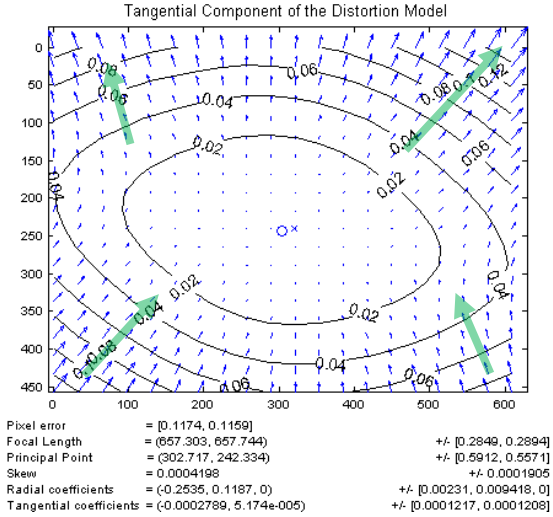


radial distortion

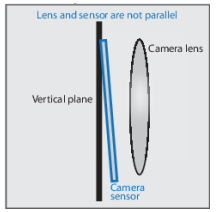
barrel dist.

pincushion dist.

mustache dist.



tangential distortion



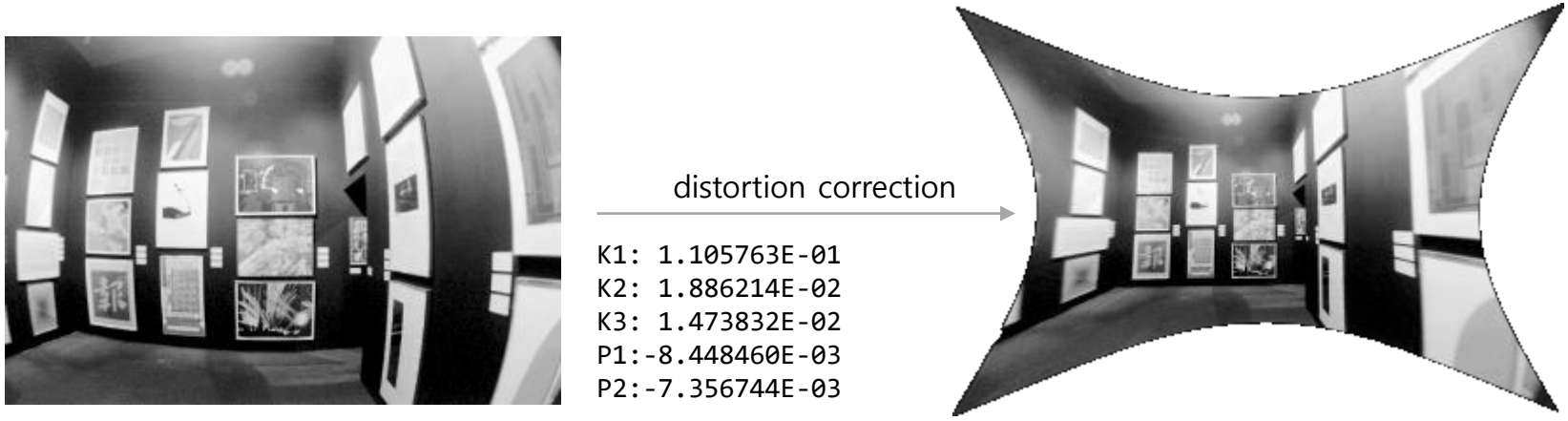
Why? lens and sensor are not parallel.

  - **Field-of-view (FOV) model**
    - Radial distortion is modeled by a tangent function (good for severe distortion; e.g. fisheye lens).

Images are from Camera Calibration Toolbox for MATLAB (http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/example.html), Wikipedia (https://en.wikipedia.org/wiki/Distortion_(optics)), and MathWorks (https://mathworks.com/help/vision/ug/camera-calibration.html)

Camera Projection Model

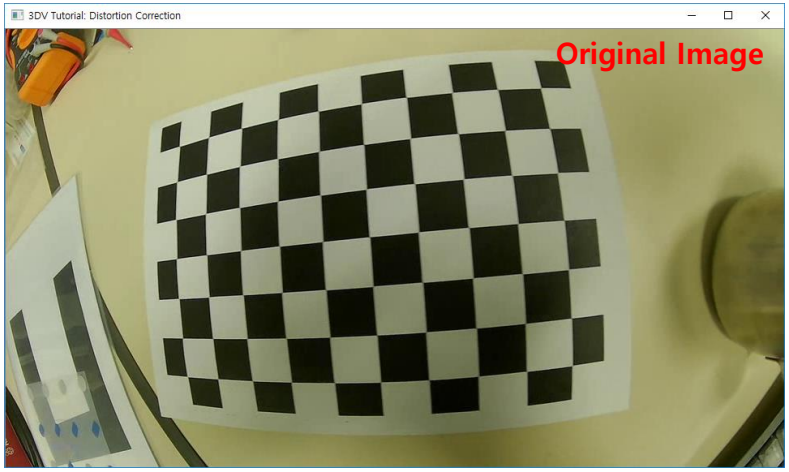
- **Geometric Distortion Correction**
  - Input: The original image
  - Output: Its rectified image (w/o geometric distortion)
  - Given: Its camera matrix and distortion coefficient
  - Solutions for the polynomial distortion model
    - OpenCV `cv::undistort()` and `cv::undistortPoints()` (cf. included in `imgproc` module)
      ↔ `cv::projectPoints()` (cf. included in `calib3d` module)
    - Camera Distortion Correction: Theory and Practice, http://darkpgmr.tistory.com/31

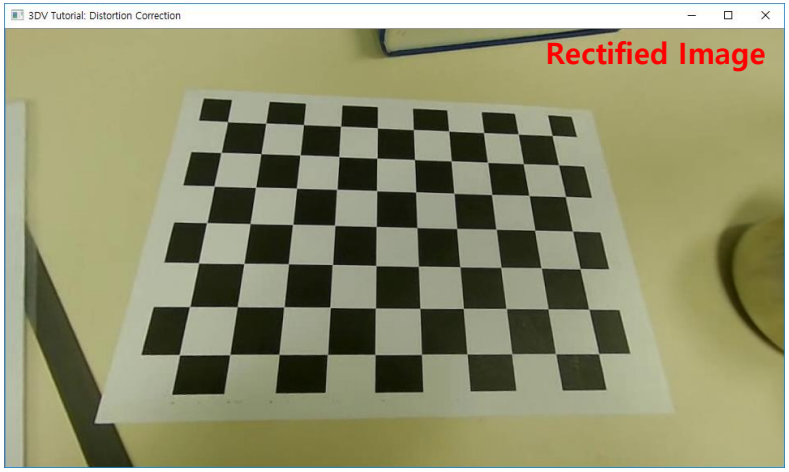


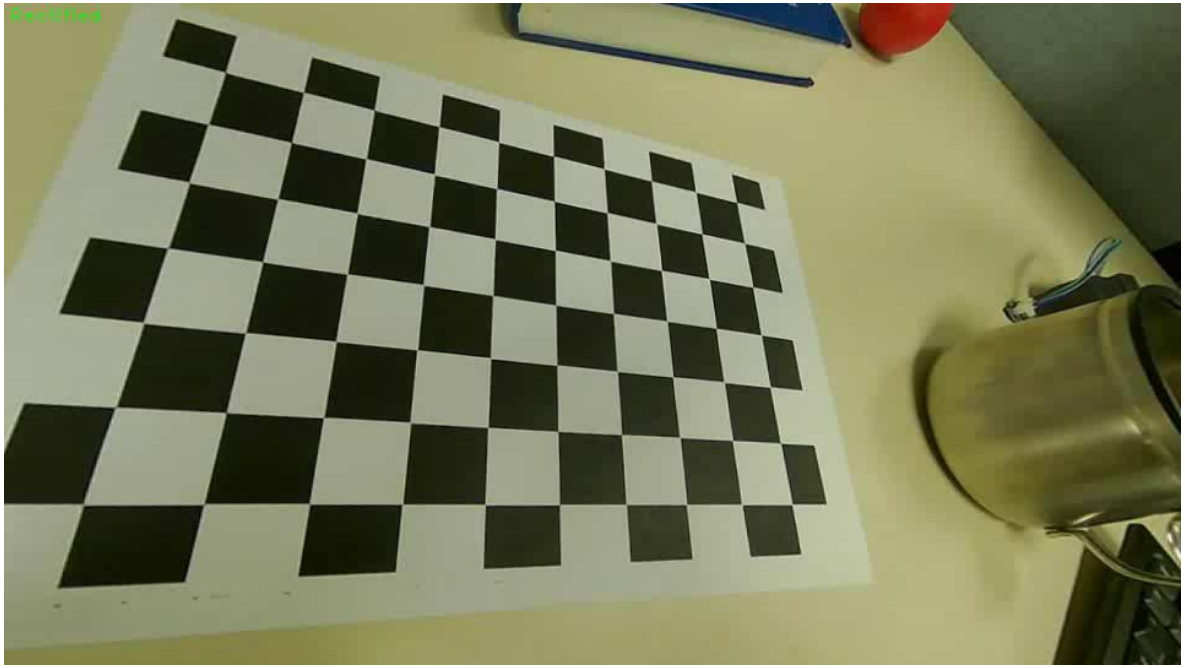
Images are from Shawn Becker (http://alumni.media.mit.edu/~sbeck/)

Camera Projection Model

- Example: **Geometric Distortion Correction** `[distortion_correction.cpp]`



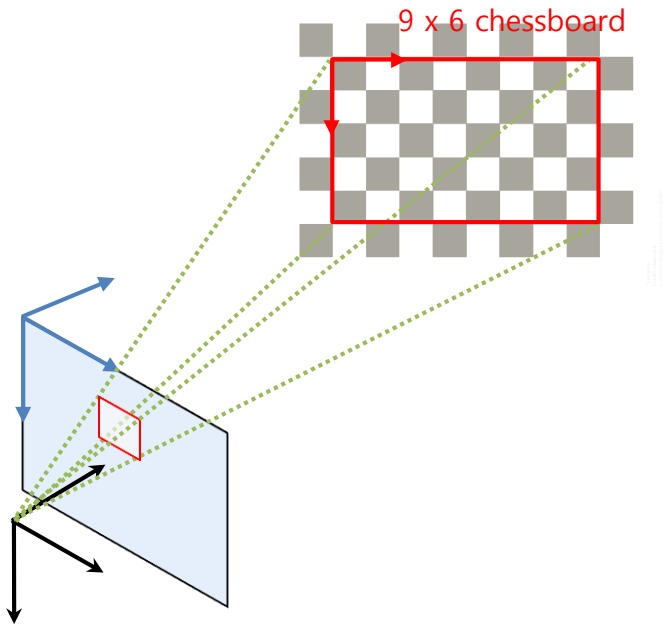




```cpp
#include "opencv2/opencv.hpp"

int main()
{
    const char* input = "data/chessboard.avi";
    cv::Matx33d K(432.7390364738057, 0, 476.0614994349778, 0, 431.2395555913084, 288.7602152621297, 0, 0, 1);
    std::vector<double> dist_coeff = { -0.2852754904152874, 0.1016466459919075, ... };

    // Open a video
    cv::VideoCapture video;
    if (!video.open(input)) return -1;

    // Run distortion correction
    bool show_rectify = true;
    cv::Mat map1, map2;
    while (true)
    {
        // Grab an image from the video
        cv::Mat image;
        video >> image;
        if (image.empty()) break;

        // Rectify geometric distortion (cf. 'cv::undistort()' can be applied for one-time remapping.)
        if (show_rectify)
        {
            if (map1.empty() || map2.empty())
                cv::initUndistortRectifyMap(K, dist_coeff, cv::Mat(), cv::Mat(), image.size(), CV_32FC1, map1, map2);
            cv::remap(image, image, map1, map2, cv::InterpolationFlags::INTER_LINEAR);
        }

        // Show the image
        cv::imshow("3DV Tutorial: Distortion Correction", image);
        int key = cv::waitKey(1);
        if (key == 27) break;                                   // 'ESC' key: Exit
        else if (key == 9) show_rectify = !show_rectify;        // 'Tab' key: Toggle rectification
        ...
    }

    video.release();
    return 0;
}
```

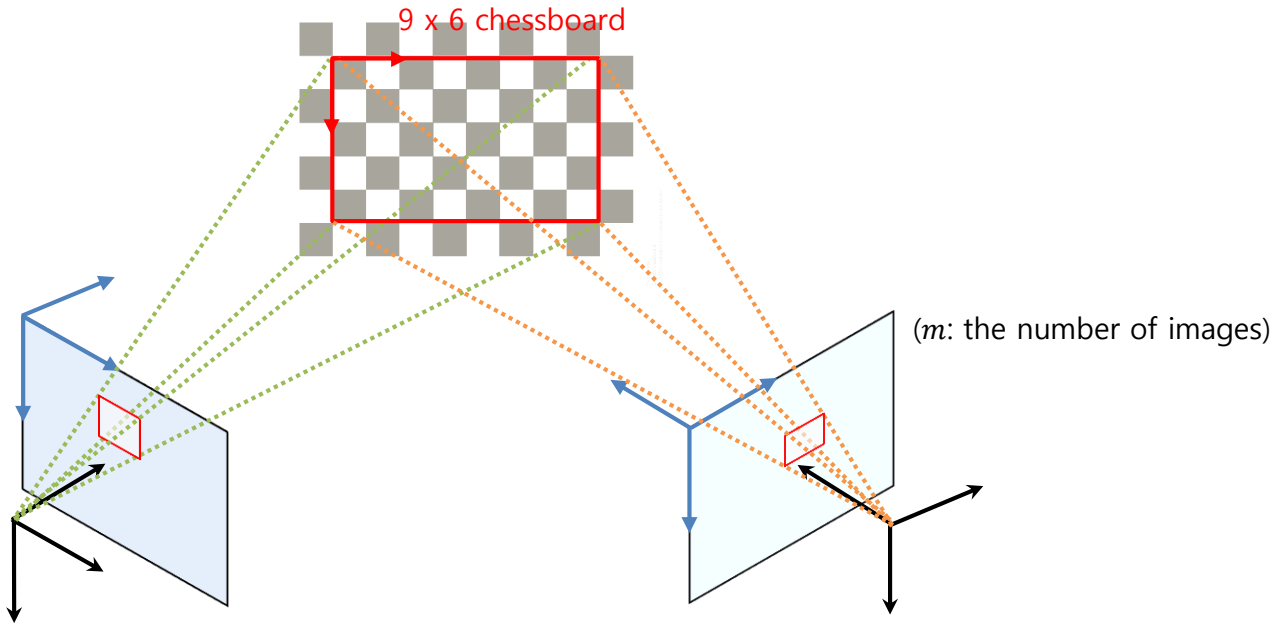
General 2D-3D Geometry

- **Camera Calibration**
  - Unknown: Intrinsic + extrinsic parameters (5* + 6 DoF)
    - The number of intrinsic parameters* can be varied w.r.t. user preference.
  - Given: 3D points $\mathbf{X}_1$, $\mathbf{X}_2$, …, $\mathbf{X}_n$ and their projected points $\mathbf{x}_1$, $\mathbf{x}_2$, …, $\mathbf{x}_n$
  - Constraints: n x projection $\mathbf{x}_i = \mathrm{K}\,[\,\mathrm{R}\,|\,\mathbf{t}\,]\,\mathbf{X}_i$



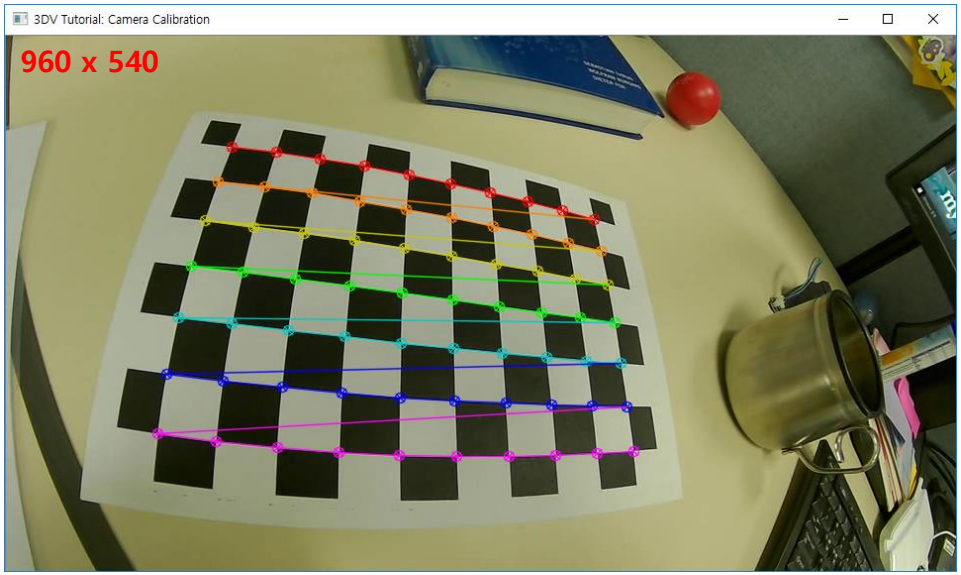
General 2D-3D Geometry

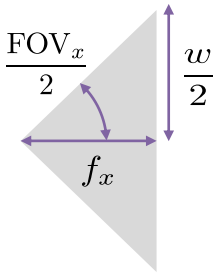
- **Camera Calibration**
  - Unknown: Intrinsic + m x extrinsic parameters (5* + m x 6 DoF)
  - Given: 3D points $\mathbf{X}_1$, $\mathbf{X}_2$, ..., $\mathbf{X}_n$ and their projected points from jth camera $\mathbf{x}_i^j$
  - Constraints: n x m x projection $\mathbf{x}_i^j = \mathrm{K}\left[\,\mathrm{R}_j|\,\mathbf{t}_j\,\right]\mathbf{X}_i$
  - Solutions
    - OpenCV `cv::calibrateCamera()` and `cv::initCameraMatrix2D()`
    - Camera Calibration Toolbox for MATLAB, http://www.vision.caltech.edu/bouguetj/calib_doc/
    - GML C++ Camera Calibration Toolbox, http://graphics.cs.msu.ru/en/node/909
    - DarkCamCalibrator, http://darkpgmr.tistory.com/139



General 2D-3D Geometry

- Example: **Camera Calibration** `[camera_calibration.cpp]`





field-of-view (FOV) = focal length

- Horizontal: $2 \times \tan^{-1} \frac{f_x}{W/2} = 84°$
- Vertical: $2 \times \tan^{-1} \frac{f_y}{h/2} = 116°$

```
## Camera Calibration Results
* The number of applied images = 22
* RMS error = 0.473353
* Camera matrix (K) =
  [432.7390364738057, 0, 476.0614994349778]
  [0, 431.2395555913084, 288.7602152621297]
  [0, 0, 1]
* Distortion coefficient (k1, k2, p1, p2, k3, ...) =
  [-0.2852754904152874, 0.1016466459919075,
   -0.0004420196146339175, 0.0001149909868437517,
   -0.01803978785585194]
```

~ close to the center of the image, (480, 270)

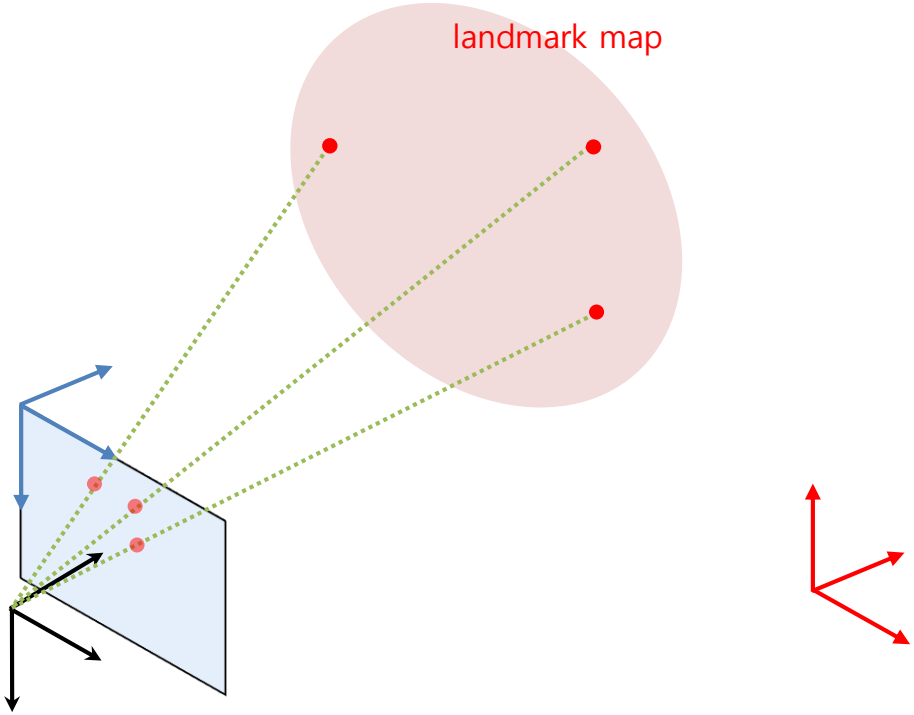
~ close to zero (usually negligible)


```cpp
#include "opencv2/opencv.hpp"

int main()
{
    const char* input = "data/chessboard.avi";
    cv::Size board_pattern(10, 7);
    float board_cellsize = 0.025f;

    // Select images
    std::vector<cv::Mat> images;
    ...

    // Find 2D corner points from the given images
    std::vector<std::vector<cv::Point2f>> img_points;
    for (size_t i = 0; i < images.size(); i++)
    {
        std::vector<cv::Point2f> pts;
        if (cv::findChessboardCorners(images[i], board_pattern, pts))
            img_points.push_back(pts);
    }
    if (img_points.empty()) return -1;

    // Prepare 3D points of the chess board
    std::vector<std::vector<cv::Point3f>> obj_points(1);
    for (int r = 0; r < board_pattern.height; r++)
        for (int c = 0; c < board_pattern.width; c++)
            obj_points[0].push_back(cv::Point3f(board_cellsize * c, board_cellsize * r, 0));
    obj_points.resize(img_points.size(), obj_points[0]); // Copy

    // Calibrate the camera
    cv::Mat K = cv::Mat::eye(3, 3, CV_64F);
    cv::Mat dist_coeff = cv::Mat::zeros(4, 1, CV_64F);
    std::vector<cv::Mat> rvecs, tvecs;
    double rms = cv::calibrateCamera(obj_points, img_points, images[0].size(), K, dist_coeff, rvecs, tvecs);

    // Report calibration results
    ...
    return 0;
}
```

$\mathbf{X}_1, \mathbf{X}_2, \cdots \quad \mathbf{x}_1, \mathbf{x}_2, \cdots$

General 2D-3D Geometry

- **Absolute Camera Pose Estimation** (Perspective-n-Point; PnP)
  - Unknown: Camera pose (6 DoF)
  - Given: 3D points $\mathbf{X}_1$, $\mathbf{X}_2$, ..., $\mathbf{X}_n$, their projected points $\mathbf{x}_1$, $\mathbf{x}_2$, ..., $\mathbf{x}_n$, and camera matrix K
  - Constraints: n x projection $\mathbf{x}_i = \mathrm{K}\,[\,\mathrm{R}\,|\,\mathbf{t}\,]\,\mathbf{X}_i$
  - Solutions ($n \geq 3$) → 3-point algorithm
    - OpenCV `cv::solvePnP()` and `cv::solvePnPRansac()`
    - Efficient PnP (EPnP), http://cvlab.epfl.ch/EPnP/



General 2D-3D Geometry

- Example: **Pose Estimation (Chessboard)** `[pose_estimation_chessboard.cpp]`

- Prepare a 3D box (AR model)
- Prepare 3D points (chessboard)

↓

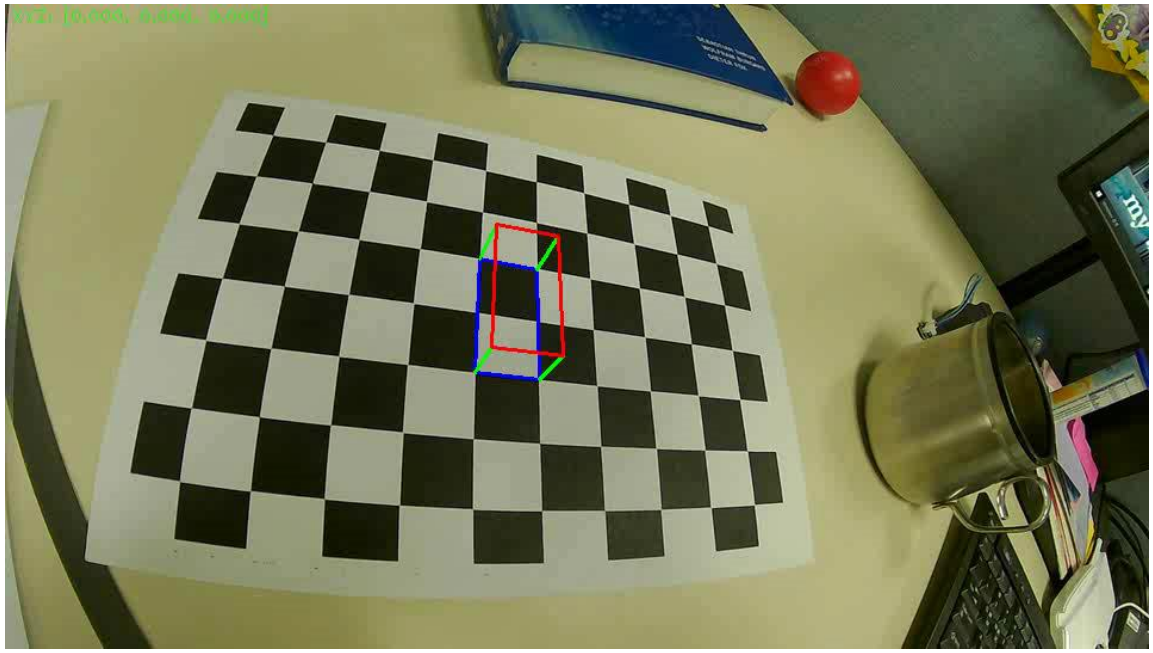
- Grab an image
- Find a chessboard on the image

↓

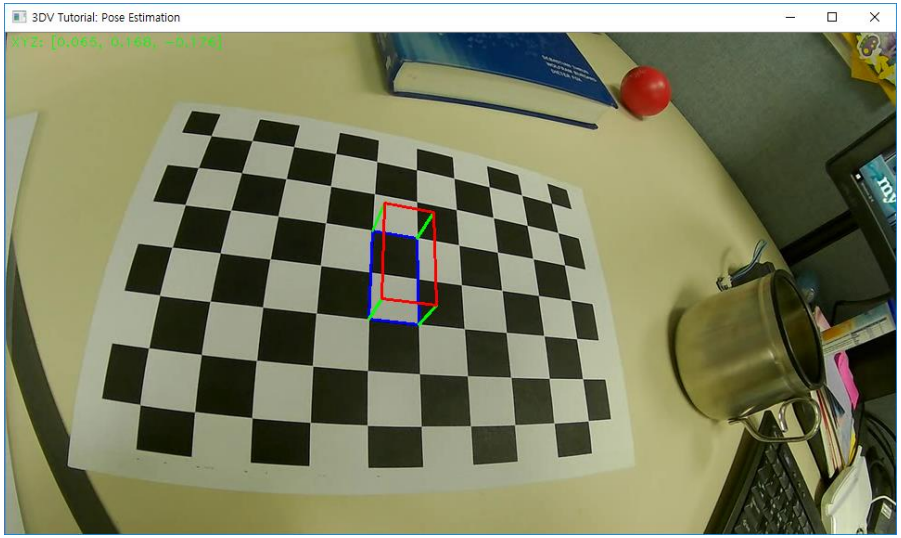
[Camera Pose Estimation]
- Estimate camera pose using `cv::solvePnP()`

↓

[AR Rendering]
- Draw the 3D box on the image using `cv::projectPoints()`



```cpp
#include "opencv2/opencv.hpp"

int main()
{
    const char* input = "data/chessboard.avi";
    cv::Matx33d K(432.7390364738057, 0, 476.0614994349778, 0, 431.2395555913084, 288.7602152621297, 0, 0, 1);
    std::vector<double> dist_coeff = { -0.2852754904152874, 0.1016466459919075, ... };
    cv::Size board_pattern(10, 7);
    double board_cellsize = 0.025;

    // Open a video
    cv::VideoCapture video;
    if (!video.open(input)) return -1;

    // Prepare a 3D box for simple AR
    std::vector<cv::Point3d> box_lower = { cv::Point3d(4 * board_cellsize, 2 * board_cellsize, 0), ... };
    std::vector<cv::Point3d> box_upper = { cv::Point3d(4 * board_cellsize, 2 * board_cellsize, -board_cellsize), ...) };

    // Prepare 3D points on a chessboard
    std::vector<cv::Point3d> obj_points;
    for (int r = 0; r < board_pattern.height; r++)
        for (int c = 0; c < board_pattern.width; c++)
            obj_points.push_back(cv::Point3d(board_cellsize * c, board_cellsize * r, 0));
```




```
// Run pose estimation
while (true)
    {
        // Grab an image from the video
        cv::Mat image;
        video >> image;
        if (image.empty()) break;

        // Estimate camera pose
        std::vector<cv::Point2d> img_points;
        bool success = cv::findChessboardCorners(image, board_pattern, img_points, ...);
        if (success)
        {
            cv::Mat rvec, tvec;
            cv::solvePnP(obj_points, img_points, K, dist_coeff, rvec, tvec);
                         X1, X2, ···   x1, x2, ···
            // Draw the box on the image
            cv::Mat line_lower, line_upper;
            cv::projectPoints(box_lower, rvec, tvec, K, dist_coeff, line_lower);
            cv::projectPoints(box_upper, rvec, tvec, K, dist_coeff, line_upper);
            ...

            // Print camera position
            cv::Mat R;
            cv::Rodrigues(rvec, R);
            cv::Mat p = -R.t() * tvec;
            cv::String info = cv::format("XYZ: [%.3f, %.3f, %.3f]", cv::Point3d(p));
            cv::putText(image, info, cv::Point(5, 15), cv::FONT_HERSHEY_PLAIN, 1, cv::Vec3b(0, 255, 0));
        }

        // Show the image
        cv::imshow("3DV Tutorial: Pose Estimation", image);
        int key = cv::waitKey(1);
        ...
    }

    video.release();
    return 0;
}
```

General 2D-3D Geometry

- Example: **Pose Estimation (Book)** `[pose_estimation_book1.cpp]`

- Load the cover image
- Extract ORB features
- Build **3D object points** $\mathbf{X}_1, \mathbf{X}_2, \ldots$

Pass tentative matches

- Grab an input image
- Extract **ORB features** $\mathbf{x}_1, \mathbf{x}_2, \ldots$
- Match them to the object features

[Outlier Rejection]
- Remove wrong matches using `cv::solvePnPRansac()`

Pass correct matches

[Camera Pose Estimation]
- Estimate camera pose using `cv::solvePnP()`

[Camera Pose Update]
- Intrinsic: K (given and fixed)
- Extrinsic: R, $\mathbf{t}$

Draw the box using `cv::projectPoints()`

General 2D-3D Geometry

- Example: **Pose Estimation (Book) + Camera Calibration** `[pose_estimation_book2.cpp]`
  (∵ unknown, autofocus)

- Load the cover image
- Extract ORB features
- Build **3D object points** $\mathbf{X}_1, \mathbf{X}_2, \ldots$

Pass tentative matches

- Grab an input image
- Extract **ORB features** $\mathbf{x}_1, \mathbf{x}_2, \ldots$
- Match them to the object features

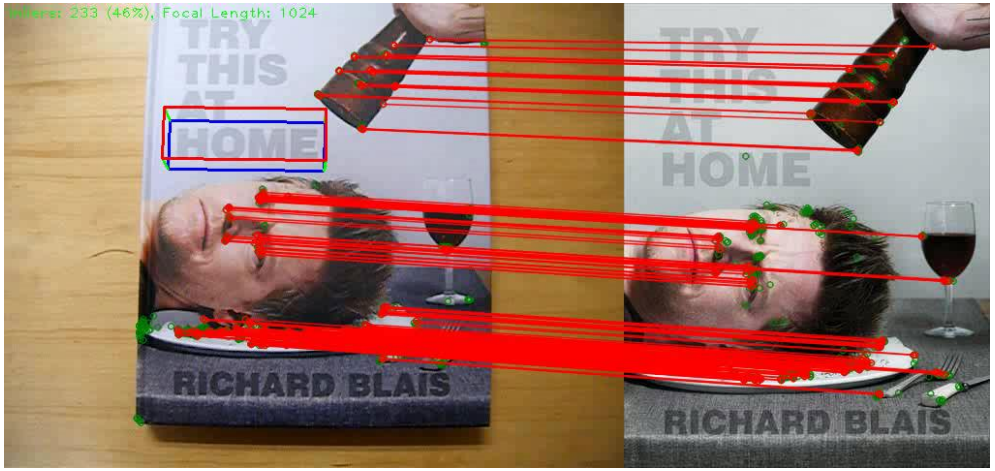
[Outlier Rejection]
- Remove wrong matches using `cv::solvePnPRansac()`

Pass correct matches

[Camera Pose Estimation]
- Estimate camera parameters using `cv::calibrateCamera()`

[Camera Pose Update]
- Intrinsic: K (initially known)
- Extrinsic: R, $\mathbf{t}$

Draw the box using `cv::projectPoints()`



General 2D-3D Geometry

[pose_estimation_book3.cpp]

- Example: **Pose Estimation (Book) + Camera Calibration – Initially Given** K (∵ unknown)

- Load the cover image
- Extract ORB features
- Build **3D object points** $\mathbf{X}_1, \mathbf{X}_2, \ldots$

Pass tentative matches

- Grab an input image
- Extract **ORB features** $\mathbf{x}_1, \mathbf{x}_2, \ldots$
- Match them to the object features

[Outlier Rejection]
- Remove wrong matches using `cv::findHomography()`

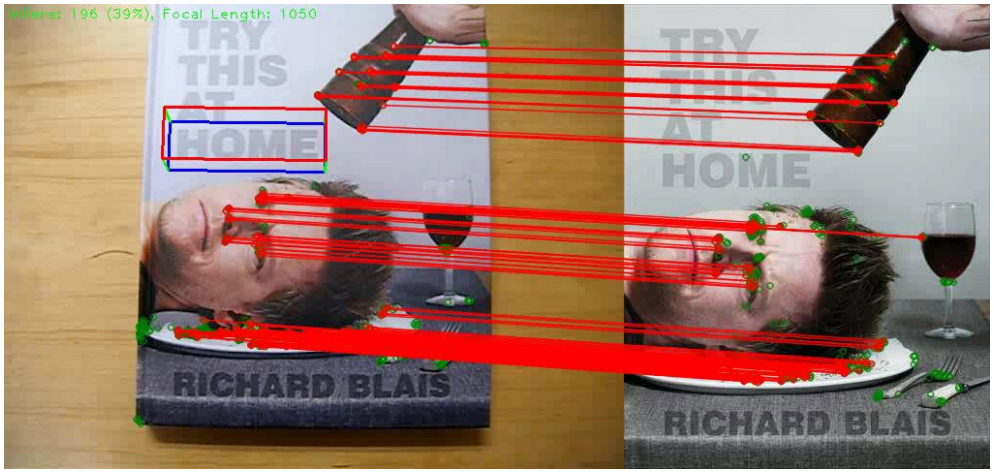
Free from K

Pass correct matches

[Camera Pose Estimation]
- Estimate camera parameters using `cv::calibrateCamera()`

[Camera Pose Update]
- Intrinsic: K (initially known)
- Extrinsic: R, $\mathbf{t}$

Draw the box using `cv::projectPoints()`



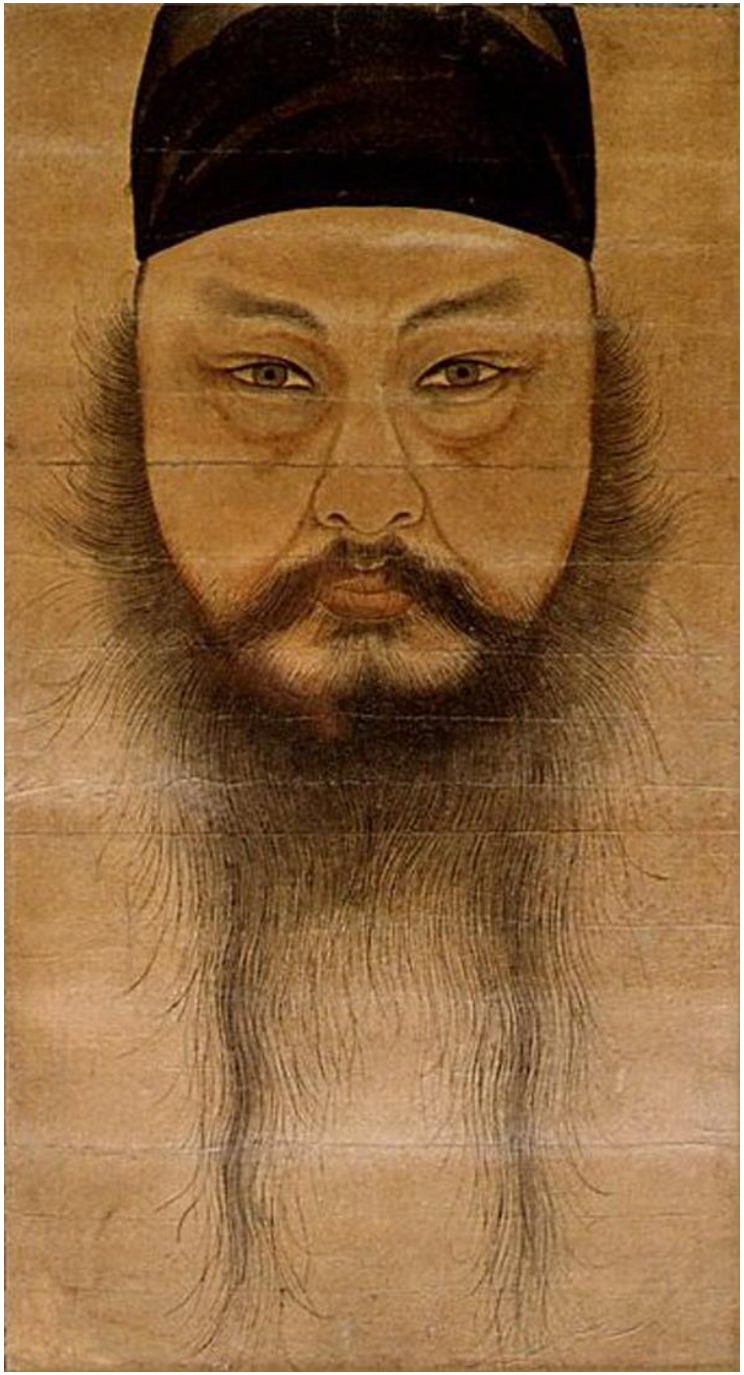
Two-view Geometry

Planar 2D-2D Geometry (Projective Geometry)

- Planar Homography Estimation

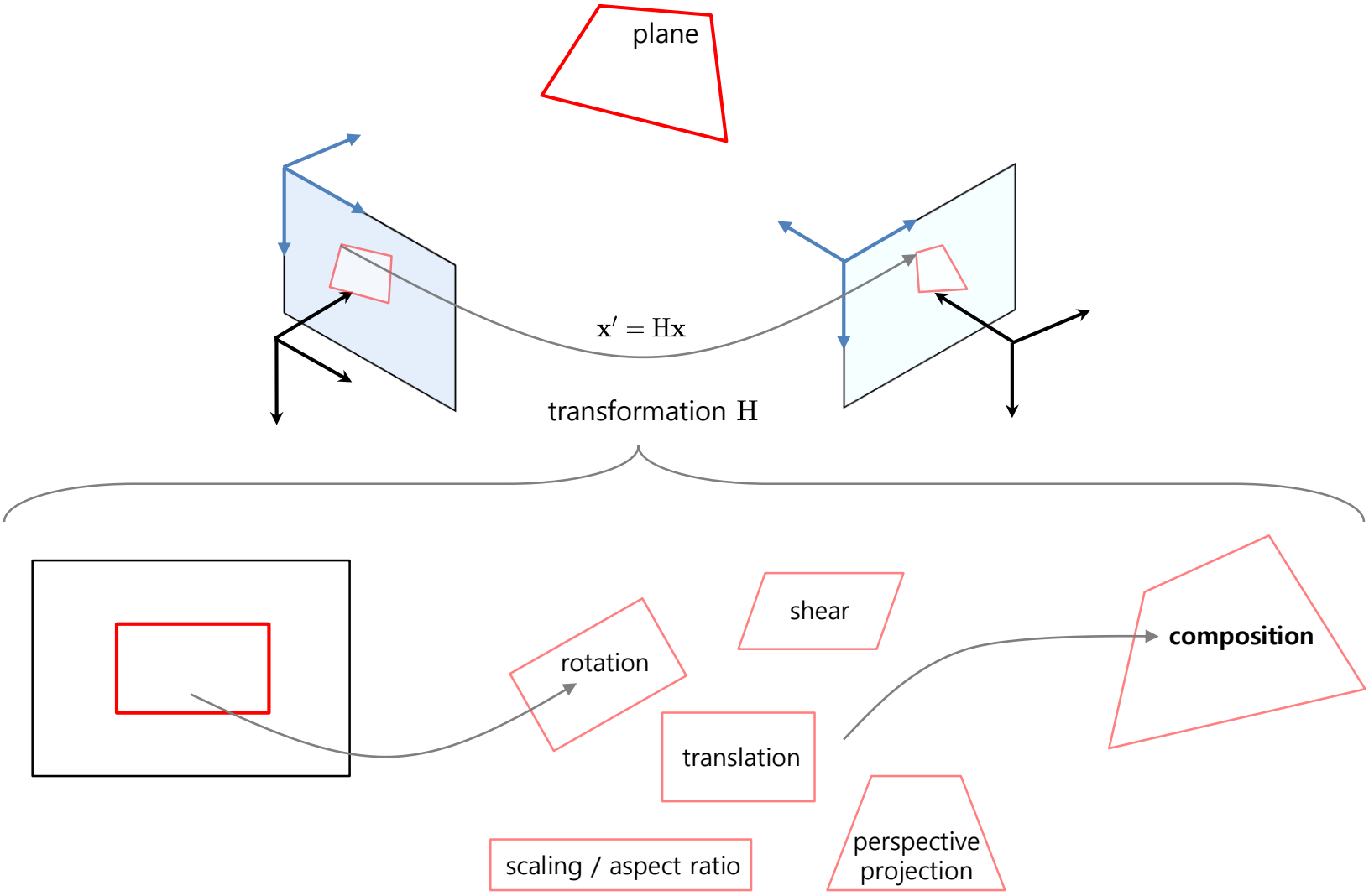
General 2D-2D Geometry (Epipolar Geometry)

- Relative Camera Pose Estimation
- Triangulation

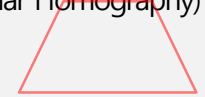


윤두서(1668-1715) 자화상, 국보 제240호
Korean National Treasure No. 240

Planar 2D-2D Geometry (Projective Geometry)



Overview of Projective Geometry

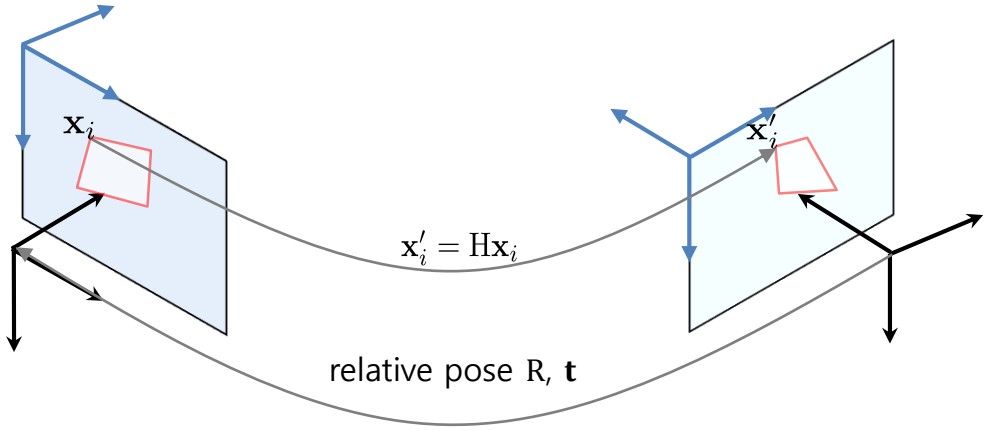
		**Euclidean Transform** (a.k.a. Rigid Transform) 	**Similarity Transform** 	**Affine Transform** 	**Projective Transform** (a.k.a. Planar Homography) 
Matrix Forms H		$\begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} s\cos\theta & -s\sin\theta & t_x \\ s\sin\theta & s\cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ v_1 & v_2 & 1 \end{bmatrix}$
DoF		3	4	6	8
Transformations					
- rotation		O	O	O	O
- translation		O	O	O	O
- scaling		X	O	O	O
- aspect ratio		X	X	O	O
- shear		X	X	O	O
- perspective projection		X	X	X	O
Invariants					
- length		O	X	X	X
- angle		O	O	X	X
- ratio of lengths		O	O	X	X
- parallelism		O	O	O	X
- incidence		O	O	O	O
- cross ratio		O	O	O	O

OpenCV Functions				`cv::getAffineTransform()`
				`cv::estimateRigidTransform()`
				-
				`cv::warpAffine()`
					`cv::getPerspectiveTransform()`
					-
					`cv::findHomography()`
					`cv::warpPerspective()`

cf. Similarly **3D transformations** (3D-3D geometry) are represented as **4x4 matrices**.

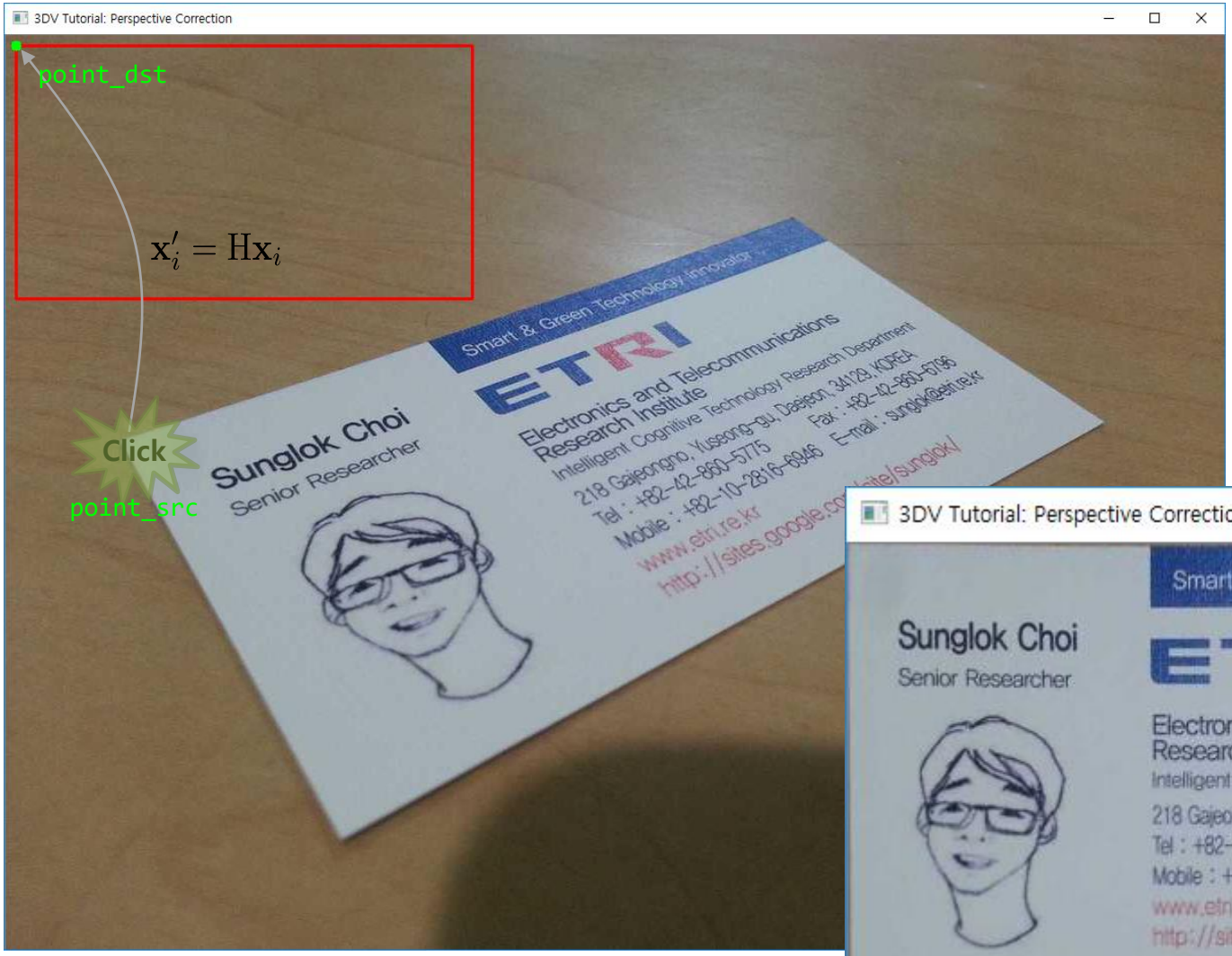
Planar 2D-2D Geometry (Projective Geometry)

- **Planar Homography Estimation**
  - Unknown: Planar homography (8 DoF)
  - Given: Point correspondence $(\mathbf{x}_1, \mathbf{x}'_1)$, …, $(\mathbf{x}_n, \mathbf{x}'_n)$
  - Constraints: n x projective transformation $\mathbf{x}'_i = \mathrm{H}\mathbf{x}_i$
  - Solutions ($n \geq 4$) → 4-point algorithm
    - OpenCV `cv::getPerspectiveTransform()` and `cv::findHomography()`
    - cf. More simplified transformations need less number of minimal correspondence.
      - Affine ($n \geq 3$), similarity ($n \geq 2$), Euclidean ($n \geq 2$)
  - [Note] Planar homography can be decomposed as relative camera pose.
    - OpenCV `cv::decomposeHomographyMat()`
    - cf. However, the decomposition needs to know camera matrices.



Planar 2D-2D Geometry (Projective Geometry)

- Example: **Perspective Distortion Correction** `[perspective_correction.cpp]`



```
1. #include "opencv2/opencv.hpp"

2. void MouseEventHandler(int event, int x, int y, int flags, void* param)
3. {
4.     ...
5.         std::vector<cv::Point> *points_src = (std::vector<cv::Point> *)param;
6.         points_src->push_back(cv::Point(x, y));
7.     ...
8. }

9. int main()
10.{
11.    const char* input = "data/sunglok_desk.jpg";
12.    cv::Size card_size(450, 250);

13.    // Prepare the rectified points
14.    std::vector<cv::Point> points_dst; x'_1, x'_2, ···
15.    points_dst.push_back(cv::Point(0, 0));
16.    points_dst.push_back(cv::Point(card_size.width, 0));
17.    points_dst.push_back(cv::Point(0, card_size.height));
18.    points_dst.push_back(cv::Point(card_size.width, card_size.height));

19.    // Load an image
20.    cv::Mat original = cv::imread(input);
21.    if (original.empty()) return -1;

22.    // Get the matched points from a user's mouse
23.    std::vector<cv::Point> points_src; x_1, x_2, ···
24.    cv::namedWindow("3DV Tutorial: Perspective Correction");
25.    cv::setMouseCallback("3DV Tutorial: Perspective Correction", MouseEventHandler, &points_src);
26.    while (points_src.size() < 4)
27.    {
28.        ...
29.    }
30.    if (points_src.size() < 4) return -1;

31.    // Calculate planar homography and rectify perspective distortion
32.    cv::Mat H = cv::findHomography(points_src, points_dst);
33.    cv::Mat rectify;
34.    cv::warpPerspective(original, rectify, H, card_size);

35.    // Show the rectified image
36.    ...
37.}
```

Planar 2D-2D Geometry (Projective Geometry)

- Example: **Planar Image Stitching** [image_stitching.cpp]

```
1. #include "opencv2/opencv.hpp"

2. int main()
3. {
4.     // Load two images (cf. We assume that two images have the same size and type)
5.     cv::Mat image1 = cv::imread("data/hill01.jpg");
6.     cv::Mat image2 = cv::imread("data/hill02.jpg");
7.     if (image1.empty() || image2.empty()) return -1;

8.     // Retrieve matching points
9.     cv::Ptr<cv::FeatureDetector> fdetector = cv::ORB::create();
10.    std::vector<cv::KeyPoint> keypoint1, keypoint2;
11.    cv::Mat descriptor1, descriptor2;
12.    fdetector->detectAndCompute(image1, cv::Mat(), keypoint1, descriptor1);
13.    fdetector->detectAndCompute(image2, cv::Mat(), keypoint2, descriptor2);
14.    cv::Ptr<cv::DescriptorMatcher> fmatcher = cv::DescriptorMatcher::create("BruteForce-Hamming");
15.    std::vector<cv::DMatch> match;
16.    fmatcher->match(descriptor1, descriptor2, match);

17.    // Calculate planar homography and merge them
18.    std::vector<cv::Point2f> points1, points2;
19.    for (size_t i = 0; i < match.size(); i++)
20.    {
21.        points1.push_back(keypoint1.at(match.at(i).queryIdx).pt);
22.        points2.push_back(keypoint2.at(match.at(i).trainIdx).pt);
23.    }
24.    cv::Mat H = cv::findHomography(points2, points1, cv::RANSAC);
25.    cv::Mat merge;
26.    cv::warpPerspective(image2, merge, H, cv::Size(image1.cols * 2, image1.rows));
27.    merge.colRange(0, image1.cols) = image1 * 1; // Copy

28.    // Show the merged image
29.    ...
30. }
```

+

=

Planar 2D-2D Geometry (Projective Geometry)

- Example: **2D Video Stabilization** [video_stabilization.cpp]

```
1. #include "opencv2/opencv.hpp"

2. int main()
3. {
4.     // Open a video and get the reference image and feature points
5.     cv::VideoCapture video;
6.     if (!video.open("data/traffic.avi")) return -1;

7.     cv::Mat gray_ref;
8.     video >> gray_ref;
9.     if (gray_ref.empty())
10.    {
11.        video.release();
12.        return -1;
13.    }
14.    if (gray_ref.channels() > 1) cv::cvtColor(gray_ref, gray_ref, cv::COLOR_RGB2GRAY);

15.    std::vector<cv::Point2f> point_ref;
16.    cv::goodFeaturesToTrack(gray_ref, point_ref, 2000, 0.01, 10);
17.    if (point_ref.size() < 4)
18.    {
19.        video.release();
20.        return -1;
21.    }
```

A shaking CCTV video: "data/traffic.avi"

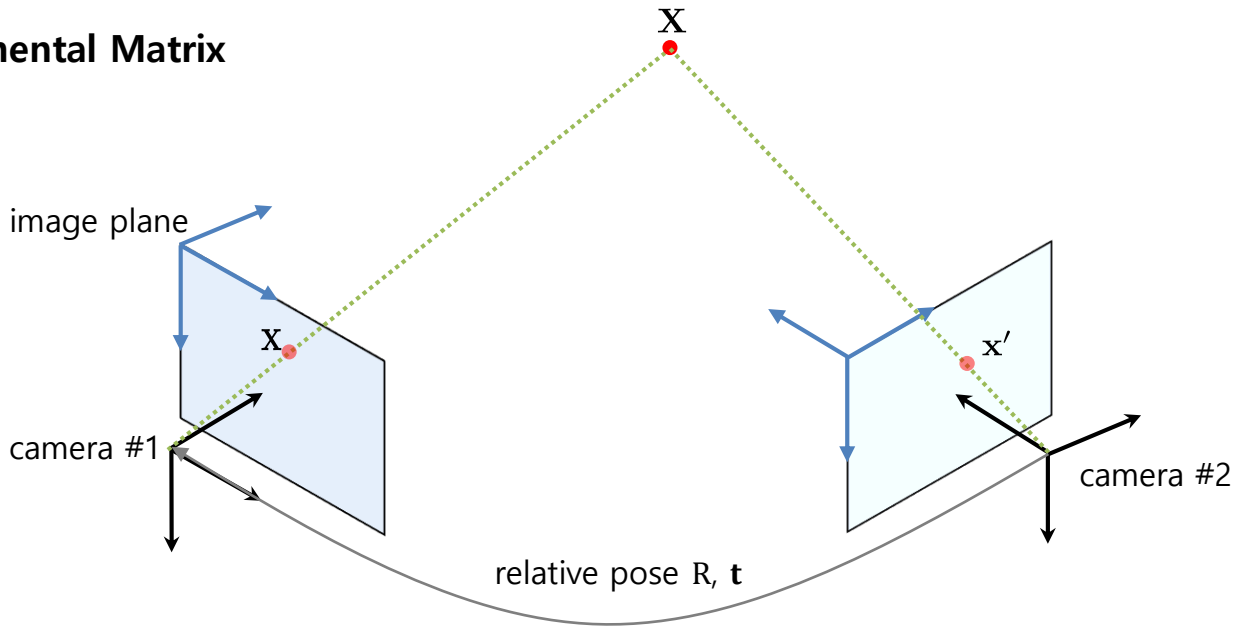
```
22.      // Run and show video stabilization
23.      while (true)
24.      {
25.          // Grab an image from the video
26.          cv::Mat image, gray;
27.          video >> image;
28.          if (image.empty()) break;
29.          if (image.channels() > 1) cv::cvtColor(image, gray, cv::COLOR_RGB2GRAY);
30.          else                      gray = image.clone();

31.          // Extract optical flow and calculate planar homography
32.          std::vector<cv::Point2f> point;
33.          std::vector<uchar> m_status;
34.          cv::Mat err, inlier_mask;
35.          cv::calcOpticalFlowPyrLK(gray_ref, gray, point_ref, point, m_status, err);
36.          cv::Mat H = cv::findHomography(point, point_ref, inlier_mask, cv::RANSAC);

37.          // Synthesize a stabilized image
38.          cv::Mat warp;
39.          cv::warpPerspective(image, warp, H, cv::Size(image.cols, image.rows));

40.          // Show the original and rectified images together
41.          for (int i = 0; i < point_ref.size(); i++)
42.          {
43.              if (inlier_mask.at<uchar>(i) > 0) cv::line(image, point_ref[i], point[i], cv::Vec3b(0, 0, 255));
44.              else cv::line(image, point_ref[i], point[i], cv::Vec3b(0, 127, 0));
45.          }
46.          cv::hconcat(image, warp, image);
47.          cv::imshow("3DVT Tutorial: Video Stabilization", image);
48.          if (cv::waitKey(1) == 27) break; // 'ESC' key
49.      }

50.      video.release();
51.      return 0;
52.  }
```

Assumption: **A plane observed by two views**

- Image stitching: Distance ≫ depth variation
- Video stabilization: Small motion

General 2D-2D Geometry (Epipolar Geometry)

- **Fundamental Matrix**



Epipolar Constraint: $\mathbf{x}'^{\top}\mathrm{F}\mathbf{x} = 0$ on the image plane

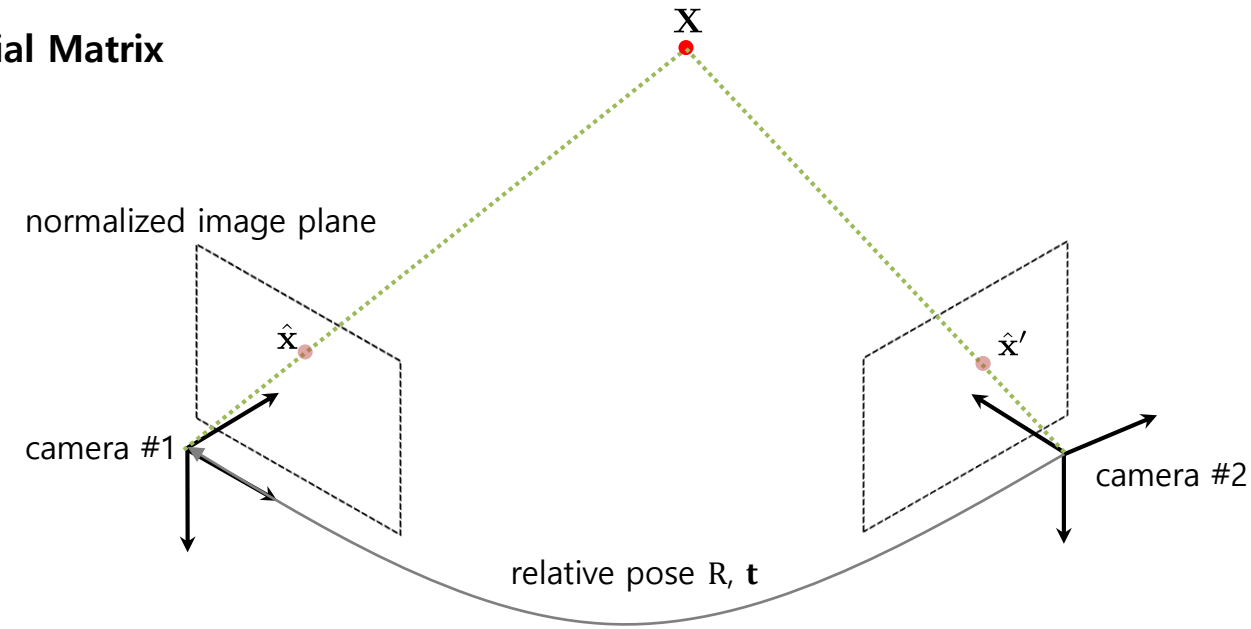
$\mathbf{x} = \mathrm{K}\hat{\mathbf{x}}$

$\hat{\mathbf{x}}'^{\top}\mathrm{K}'^{\top}\mathrm{F}\mathrm{K}\hat{\mathbf{x}} = 0$

$\hat{\mathbf{x}}'^{\top}\mathrm{E}\hat{\mathbf{x}} = 0 \quad (\mathrm{E} = \mathrm{K}'^{\top}\mathrm{F}\mathrm{K})$ on the normalized image plane

General 2D-2D Geometry (Epipolar Geometry)

- **Essential Matrix**



Epipolar Constraint: $\hat{\mathbf{x}}'^{\top}\mathrm{E}\hat{\mathbf{x}} = 0 \quad (\mathrm{E} = [\mathbf{t}]_{\times}\mathrm{R})$

(Derivation)

c.f. $[\mathbf{t}]_{\times} = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}$

$$\hat{\mathbf{x}}' \cdot \mathbf{t} \times \mathrm{R}\hat{\mathbf{x}} = 0$$

$\hat{\mathbf{x}}' \cdot (\mathbf{t} \times \hat{\mathbf{x}}') = 0$ ↑

$$\lambda'\hat{\mathbf{x}}' \cdot (\mathbf{t} \times \hat{\mathbf{x}}') = \lambda\hat{\mathbf{x}}' \cdot \mathbf{t} \times \mathrm{R}\hat{\mathbf{x}}$$

$\hat{\mathbf{x}}'\cdot$ ↑

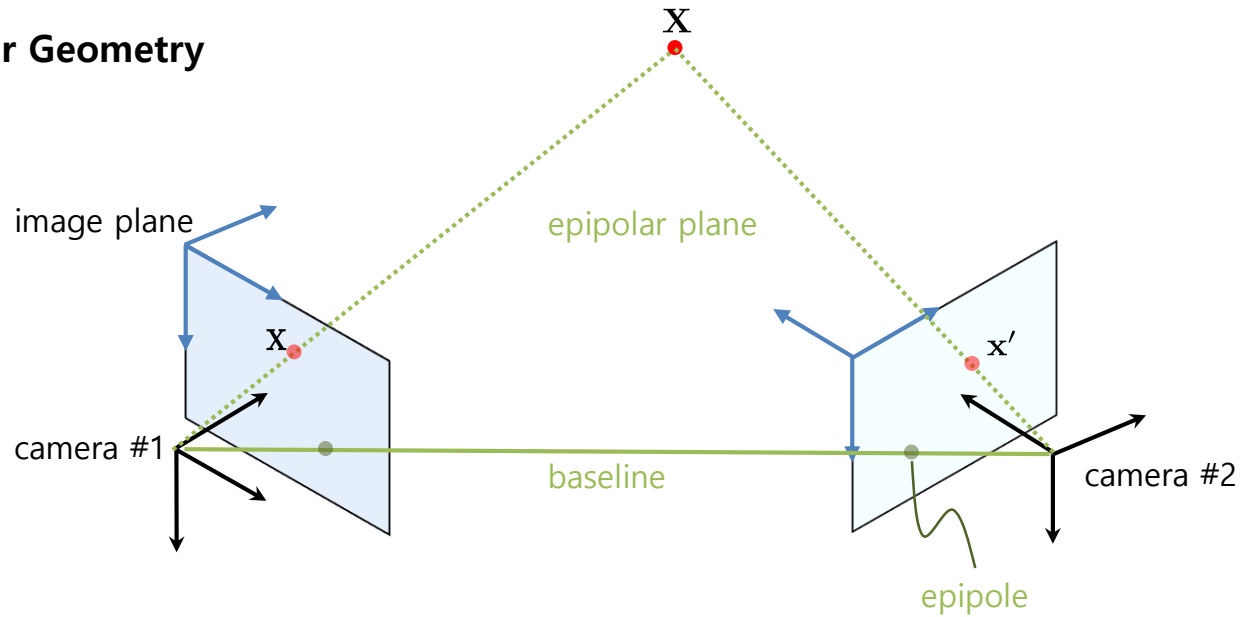
$$\lambda'\mathbf{t} \times \hat{\mathbf{x}}' = \lambda\mathbf{t} \times \mathrm{R}\hat{\mathbf{x}}$$

$\mathbf{t}\times$ ↑

$$\lambda'\hat{\mathbf{x}}' = \lambda\mathrm{R}\hat{\mathbf{x}} + \mathbf{t} \xleftarrow{\mathbf{X} = \lambda\hat{\mathbf{x}}} \mathbf{X}' = \mathrm{R}\mathbf{X} + \mathbf{t}$$

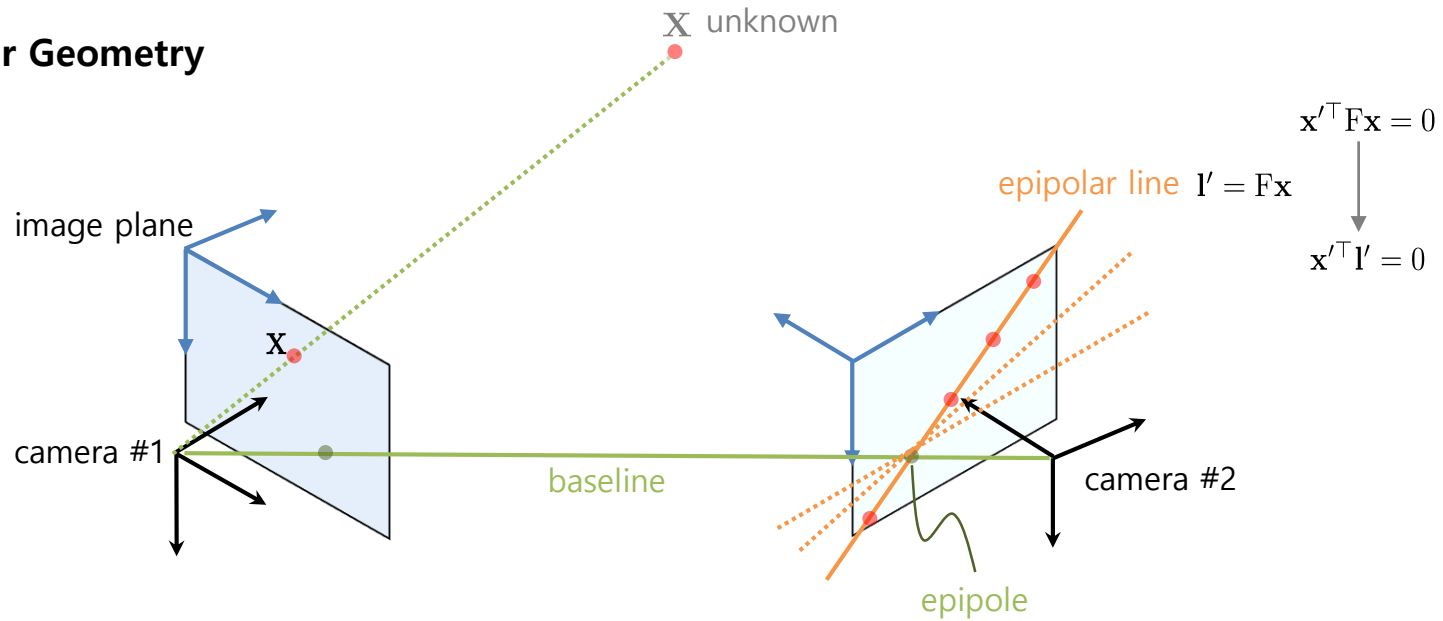
General 2D-2D Geometry (Epipolar Geometry)

- **Epipolar Geometry**

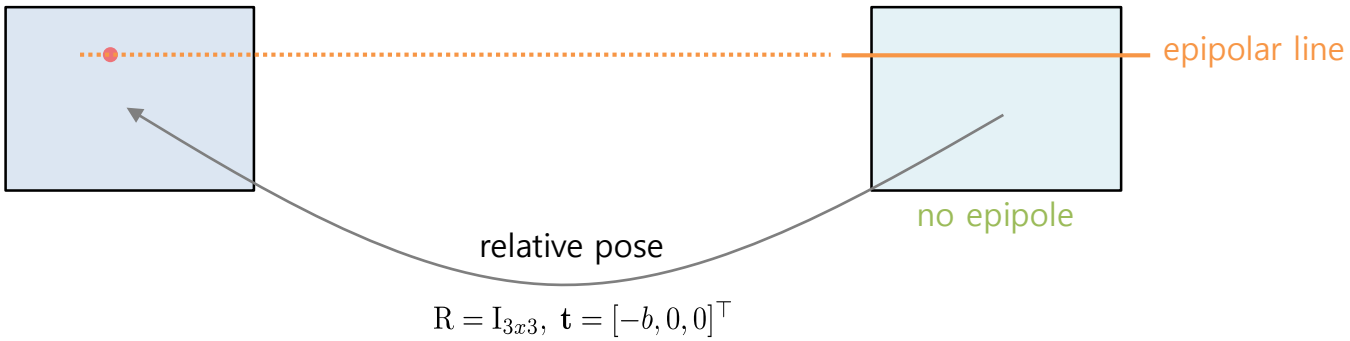


General 2D-2D Geometry (Epipolar Geometry)

- **Epipolar Geometry**



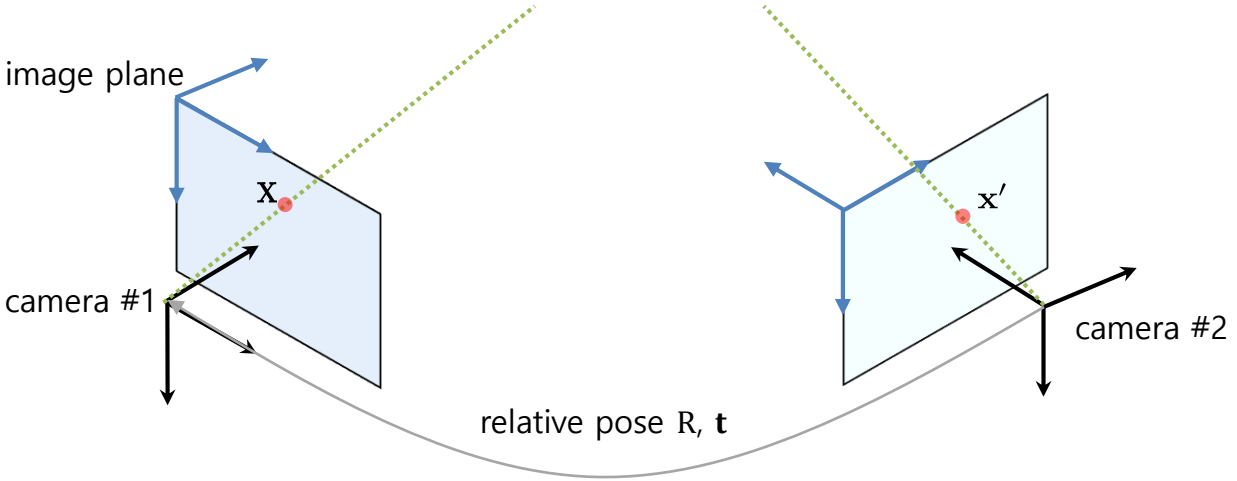
Special case) Stereo Cameras



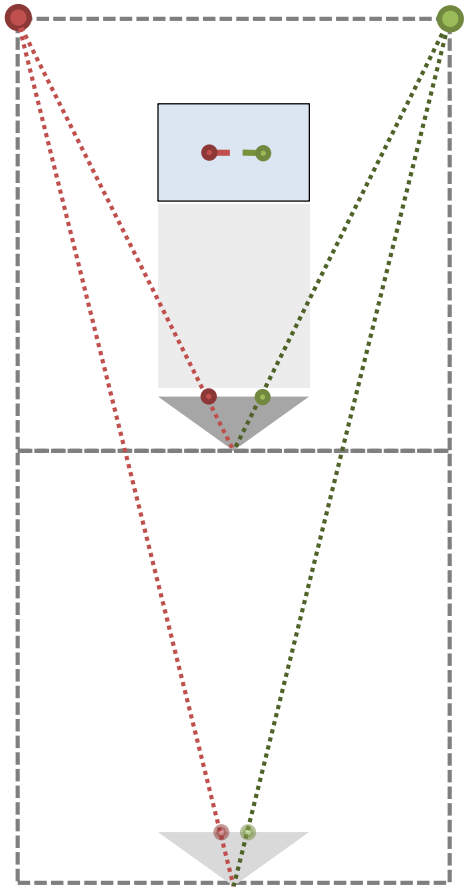
General 2D-2D Geometry (Epipolar Geometry)

- **Relative Camera Pose Estimation (~ Fundamental/Essential Matrix Estimation)**
  - Unknown: Rotation and translation R, $\mathbf{t}$ (**5 DoF**; up-to scale **"scale ambiguity"**)
  - Given: Point correspondence $(\mathbf{x}_1, \mathbf{x}'_1)$, …, $(\mathbf{x}_n, \mathbf{x}'_n)$ and camera matrices K, K'
  - Constraints: n x epipolar constraint ($\mathbf{x}'^{\mathsf{T}}\mathrm{F}\mathbf{x} = 0$ or $\hat{\mathbf{x}}'^{\mathsf{T}}\mathrm{E}\hat{\mathbf{x}} = 0$)
  - Solutions (OpenCV)
    - **Fundamental matrix**: 7/8-point algorithm (7 DoF)
      - **Estimation**: `cv::findFundamentalMat()` → 1 solution
      - **Conversion to E**: $\mathrm{E} = \mathrm{K}'^{\mathsf{T}}\mathrm{F}\mathrm{K}$
    - **Essential matrix**: 5-point algorithm (5 DoF)
      - **Estimation**: `cv::findEssentialMat()` → k solutions
      - **Decomposition**: `cv::decomposeEssentialMat()` → 4 solutions **"relative pose ambiguity"**
      - **Decomposition with positive-depth check**: `cv::recoverPose()` → 1 solution

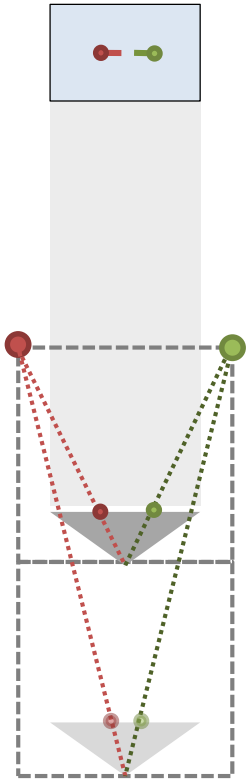
$$\mathrm{F} \;\downarrow\; \mathrm{E} = \mathrm{K}'^{\mathsf{T}}\mathrm{F}\mathrm{K} \;\downarrow\; \mathrm{E} = [\mathbf{t}]_{\times}\mathrm{R} \;\downarrow\; \mathrm{R},\ \mathbf{t}$$



Epipolar Geometry: Scale Ambiguity



(a) 2-meter-wide tunnel



(b) 1-meter-wide tunnel

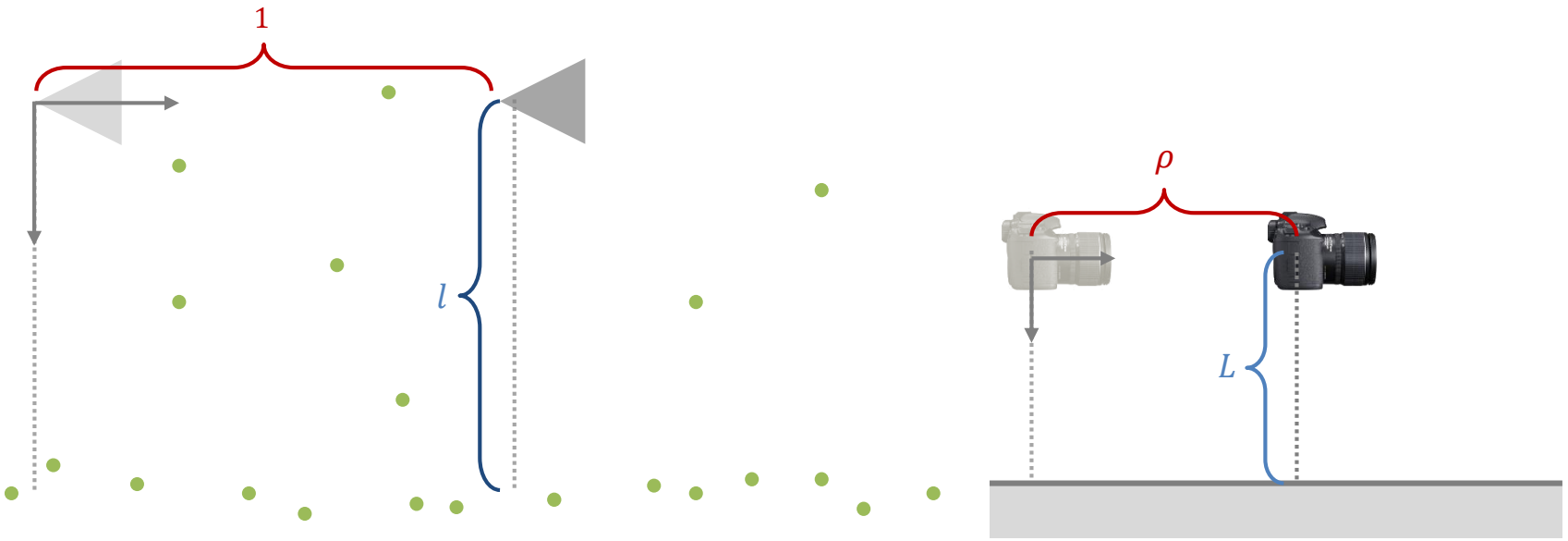
Images are from *Resolving Scale Ambiguity in Monocular Visual Odometry* (by Choi et al., 2013).

Epipolar Geometry: Scale Ambiguity

The Sandcrawler @ Star Wars IV: A New Hope (1977)

Images are from http://ricardodelgadoart.blogspot.com/, https://www.galangpratama.com/, and https://www.pinterest.com/

Epipolar Geometry: Resolving Scale Ambiguity



(a) The reconstructed world

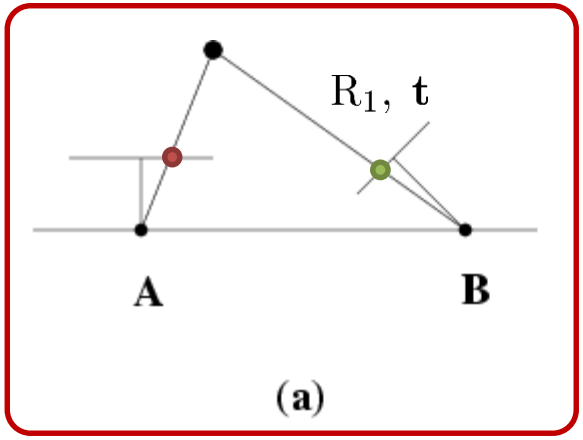
(b) The real world

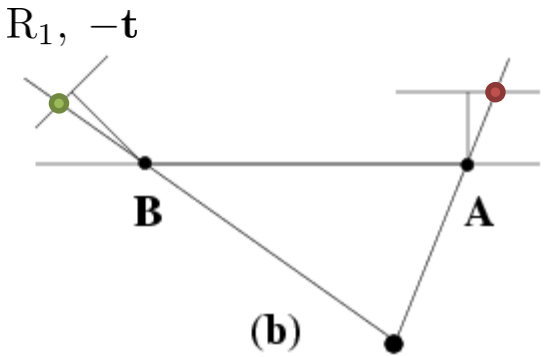
How to resolve scale ambiguity

- Additional **sensors**: speedometers, IMUs, GPSs, depth/distance (stereo, RGB-D, LiDAR, ...)
- More **constraints**: known size of objects, known and fixed height of camera ($\rho = \frac{L}{l}$)

Images are from *Resolving Scale Ambiguity in Monocular Visual Odometry* (by Choi et al., 2013).

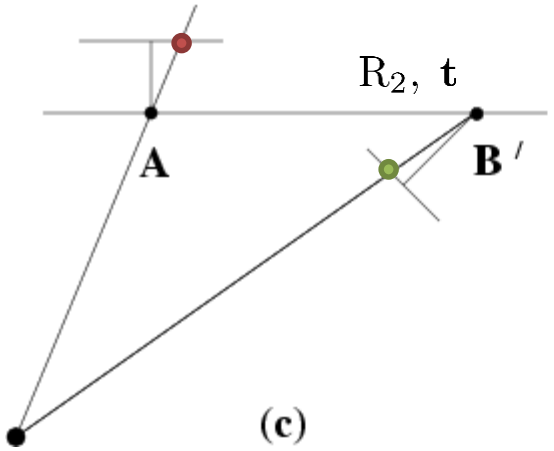
Epipolar Geometry: Relative Pose Ambiguity

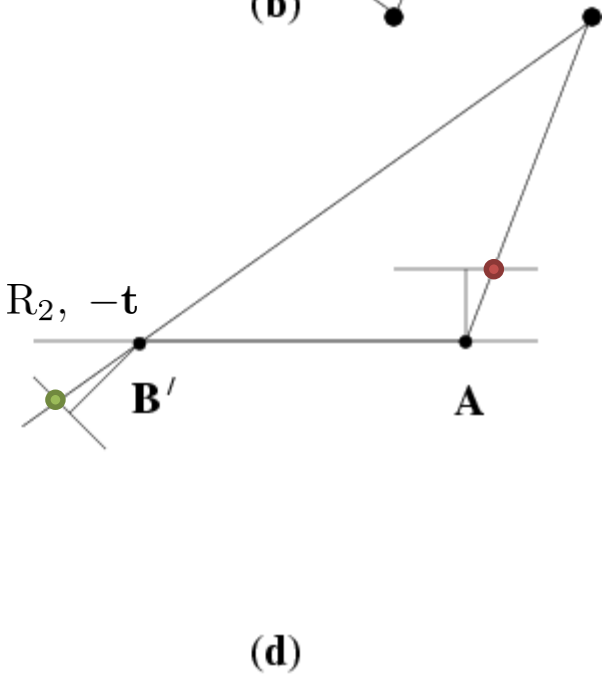




How to resolve pose ambiguity

- Positive depth constraint





Images are from *Multiple View Geometry in Computer Vision* (by Hartley and Zisserman, 2nd Edition, 2004).

General 2D-2D Geometry (Epipolar Geometry)

- **Relative Camera Pose Estimation**
  - Unknown: Rotation and translation R, $\mathbf{t}$ (**5 DoF**; up-to scale)
  - Given: Point correspondence $(\mathbf{x}_1, \mathbf{x}'_1)$, …, $(\mathbf{x}_n, \mathbf{x}'_n)$ and camera matrices K, K'
  - Constraints: n x epipolar constraint ($\mathbf{x}'^{\mathsf{T}}\mathrm{F}\mathbf{x} = 0$ or $\hat{\mathbf{x}}'^{\mathsf{T}}\mathrm{E}\hat{\mathbf{x}} = 0$)
  - Solutions (OpenCV)

intrinsic & extrinsic camera parameters

- **Fundamental matrix**: 7/8-point algorithm (**7 DoF**)

$$\mathrm{F} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \quad \& \quad \begin{matrix} \mathrm{rank}(\mathrm{F}) = 2 \\ \det(\mathrm{F}) = 0 \end{matrix}$$

  - **Estimation**: `cv::findFundamentalMat()` → 1 solution
  - **Conversion to E**: $\mathrm{E} = \mathrm{K}'^{\mathsf{T}}\mathrm{F}\mathrm{K}$
  - **Degenerate cases**: No translation, correspondence from a single plane

extrinsic camera parameters

- **Essential matrix**: 5-point algorithm (**5 DoF**)

$$2\mathrm{E}\mathrm{E}^{\top}\mathrm{E} - \mathrm{tr}(\mathrm{E}\mathrm{E}^{\top})\mathrm{E} = 0$$

  - **Estimation**: `cv::findEssentialMat()` → k solutions
  - **Decomposition**: `cv::decomposeEssentialMat()` → 4 solutions
  - **Decomposition with positive-depth check**: `cv::recoverPose()` → 1 solution
  - **Degenerate cases**: No translation ($\because \mathrm{E} = [\mathbf{t}]_{\times}\mathrm{R}$)

General 2D-2D Geometry (Epipolar Geometry)

- **Relative Camera Pose Estimation**
  - Unknown: Rotation and translation R, $\mathbf{t}$ (**5 DoF**; up-to scale)
  - Given: Point correspondence $(\mathbf{x}_1, \mathbf{x}'_1)$, …, $(\mathbf{x}_n, \mathbf{x}'_n)$ and camera matrices K, K'
  - Constraints: n x epipolar constraint ($\mathbf{x}'^{\mathsf{T}}\mathrm{F}\mathbf{x} = 0$ or $\hat{\mathbf{x}}'^{\mathsf{T}}\mathrm{E}\hat{\mathbf{x}} = 0$)
  - Solutions (OpenCV)

intrinsic & extrinsic camera parameters

- **Fundamental matrix**: 7/8-point algorithm (**7 DoF**)

$$\mathrm{F} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \quad \& \quad \begin{matrix} \mathrm{rank}(\mathrm{F}) = 2 \\ \det(\mathrm{F}) = 0 \end{matrix}$$

  - **Estimation**: `cv::findFundamentalMat()` → 1 solution
  - **Conversion to E**: $\mathrm{E} = \mathrm{K}'^{\mathsf{T}}\mathrm{F}\mathrm{K}$
  - **Degenerate cases**: No translation, correspondence from a single plane

extrinsic camera parameters

- **Essential matrix**: 5-point algorithm (**5 DoF**)

$$2\mathrm{E}\mathrm{E}^{\top}\mathrm{E} - \mathrm{tr}(\mathrm{E}\mathrm{E}^{\top})\mathrm{E} = 0$$

  - **Estimation**: `cv::findEssentialMat()` → k solutions
  - **Decomposition**: `cv::decomposeEssentialMat()` → 4 solutions
  - **Decomposition with positive-depth check**: `cv::recoverPose()` → 1 solution
  - **Degenerate cases**: No translation ($\because \mathrm{E} = [\mathbf{t}]_{\times}\mathrm{R}$)

intrinsic & extrinsic camera parameters + plane normal

- **Planar homography**: 4-point algorithm (**8 DoF**)
  - **Estimation**: `cv::findHomography()` → 1 solutions
  - **Conversion to calibrated H**: $\hat{\mathrm{H}} = \mathrm{K}'^{-1}\mathrm{H}\mathrm{K}$
  - **Decomposition**: `cv::decomposeHomographyMat()` → 4 solutions
  - **Degenerate cases**: Correspondence **not** from a single plane

$$\mathrm{H} = \mathrm{K}'\hat{\mathrm{H}}\mathrm{K}^{-1} \uparrow$$

$$\hat{\mathrm{H}} = \lambda''\left(\mathrm{R} + \frac{1}{d}\mathbf{t}\mathbf{n}^{\top}\right) \leftarrow \hat{\mathbf{x}}' = \lambda''(\mathrm{R} + \frac{1}{d}\mathbf{t}\mathbf{n}^{\top})\hat{\mathbf{x}} \leftarrow \lambda'\hat{\mathbf{x}}' = \lambda\mathrm{R}\hat{\mathbf{x}} + \mathbf{t}$$

$$\frac{1}{d}\mathbf{n}^{\top}\hat{\mathbf{x}} = 1 \quad (\because n_x\hat{x} + n_y\hat{y} + n_z\hat{z} - d = 0) \uparrow$$

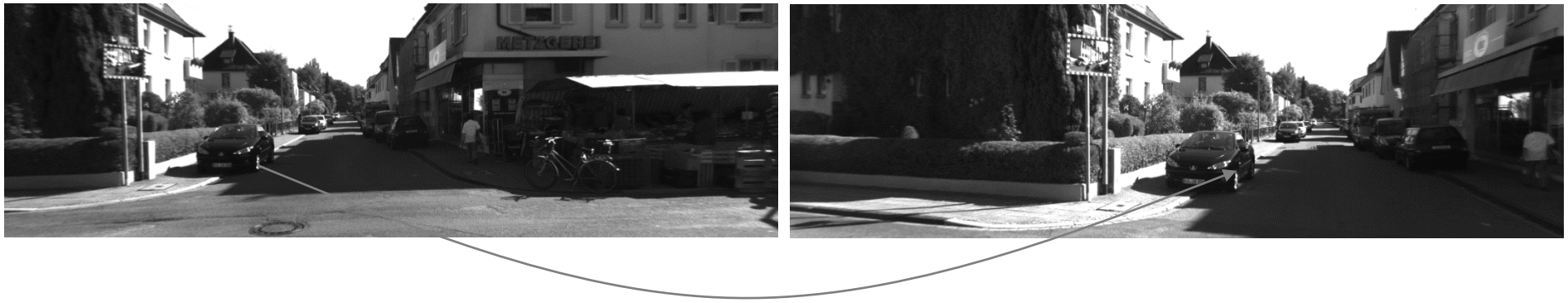
Overview of Epipolar Geometry

	Items	General 2D-2D Geometry	Planar 2D-2D Geometry
On Image Planes	Model	**Fundamental Matrix (7 DoF)**	**Planar Homography (8 DoF)**
	Formulation	$\mathrm{F} = \mathrm{K}'^{-\mathsf{T}}\mathrm{E}\mathrm{K}^{-1}$ $\quad \mathrm{E} = \mathrm{K}'^{\mathsf{T}}\mathrm{F}\mathrm{K}$	$\mathrm{H} = \mathrm{K}'\widehat{\mathrm{H}}\mathrm{K}^{-1}$ $\quad \widehat{\mathrm{H}} = \mathrm{K}'^{-1}\mathrm{H}\mathrm{K}$
	Estimation	- **7-point algorithm** ($n \geq 7$) → k solution - **(normalized) 8-point algorithm** → 1 solution - `cv::findFundamentalMat()`	- **4-point algorithm** ($n \geq 4$) → 1 solution - `cv::findHomography()`
	Input	- $(\mathbf{x}_i, \mathbf{x}'_i)$ [px] on the image plane	- $(\mathbf{x}_i, \mathbf{x}'_i)$ [px] on a plane in the image plane
	Degenerate Cases	- no translational motion - correspondence from a single plane	- correspondence not from a single plane
	Decomposition to R and $\mathbf{t}$	- convert to an essential matrix and decompose it	- `cv::decomposeHomographyMat()`
On Normalized Image Planes	Model	**Essentials Matrix (5 DoF)**	**(Calibrated) Planar Homgraphy (8 DoF)**
	Formulation	$\mathrm{E} = [\mathbf{t}]_\times \mathrm{R}$	$\widehat{\mathrm{H}} = \mathrm{R} + \frac{1}{d}\mathbf{t}\mathbf{n}^{\mathsf{T}}$
	Estimation	- **5-point algorithm** ($n \geq 5$) → k solution - `cv::findEssentialMat()`	- **4-point algorithm** ($n \geq 4$) → 1 solution - `cv::findHomography()`
	Input	- $(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}'_i)$ [m] on the normalized image plane	- $(\hat{\mathbf{x}}_i, \hat{\mathbf{x}}'_i)$ [m] on a plane in the normalized image plane
	Degenerate Cases	- no translational motion	- correspondence not from a single plane
	Decomposition to R and $\mathbf{t}$	- `cv::decomposeEssentialMat()` - `cv::recoverPose()`	- `cv::decomposeHomographyMat()` with $\mathrm{K} = \mathrm{I}_{3\times3}$

General 2D-2D Geometry (Epipolar Geometry)

- Example: **Monocular Visual Odometry (Epipolar Version)** `[vo_epipolar.cpp]`

KITTI Odometry Dataset (#: 00, Left): "data/KITTI_07_L/%06d.png"



relative pose R, $\mathbf{t}$

Given K

`Use_5pt = true`

[Tentative Matching]
- Extract optical flow between adjacent images

[Outlier Rejection]
- Remove wrong matches using `cv::findEssentialMat()`

[Relative Pose Estimation]
- Decompose camera pose using `cv::recoverPose()`

Pass tentative matches

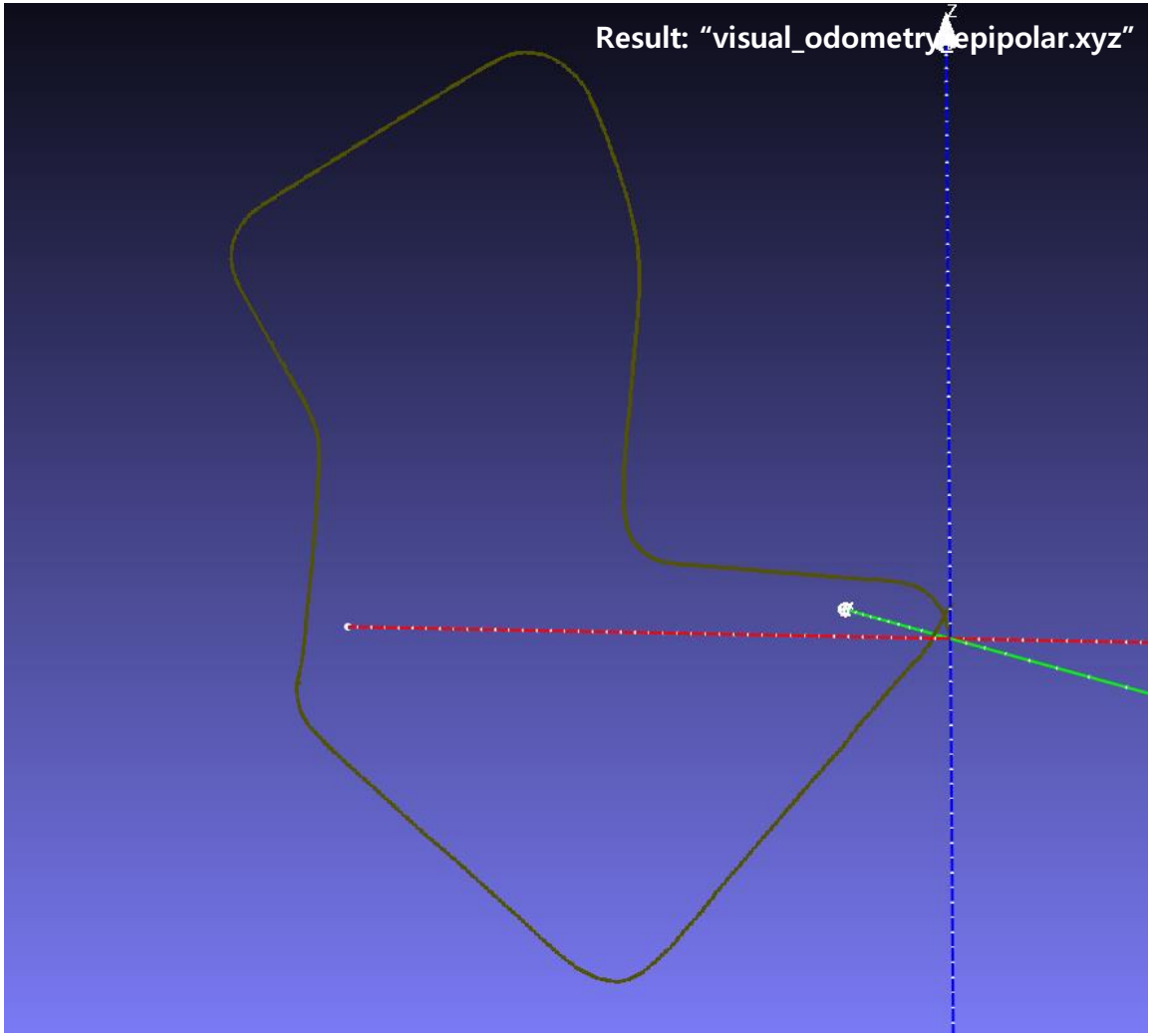
Pass the essential matrix E

[Relative Pose Accumulation]

$$\mathrm{P}_n^0 = \mathrm{P}_{n-1}^0 \begin{bmatrix} \mathrm{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}^{-1}$$

Pass the relative pose R, $\mathbf{t}$ (cf. $\| \mathbf{t} \| = 1$)


Inliers: 2 (0%), XYZ: [0.000, 0.000, 0.000]


Result: “visual_odometry_epipolar.xyz”
Z


```
1.  #include "opencv2/opencv.hpp"

2.  int main()
3.  {
4.      const char* input = "data/KITTI_07_L/%06d.png";
5.      double f = 707.0912;
6.      cv::Point2d c(601.8873, 183.1104);
7.      bool use_5pt = true;
8.      int min_inlier_num = 100;

9.      ...

10.     // Open a video and get the initial image
11.     cv::VideoCapture video;
12.     if (!video.open(input)) return -1;

13.     cv::Mat gray_prev;
14.     video >> gray_prev;
15.     ...

16.     // Run and record monocular visual odometry
17.     cv::Mat camera_pose = cv::Mat::eye(4, 4, CV_64F);
18.     while (true)
19.     {
20.         // Grab an image from the video
21.         cv::Mat image, gray;
22.         video >> image;
23.         if (image.empty()) break;
24.         if (image.channels() > 1) cv::cvtColor(image, gray, cv::COLOR_RGB2GRAY);
25.         else                      gray = image.clone();

26.         // Extract optical flow
27.         std::vector<cv::Point2f> point_prev, point;
28.         cv::goodFeaturesToTrack(gray_prev, point_prev, 2000, 0.01, 10);
29.         std::vector<uchar> m_status;
30.         cv::Mat err;
31.         cv::calcOpticalFlowPyrLK(gray_prev, gray, point_prev, point, m_status, err);
32.         gray_prev = gray;
```

```
33.         // Calculate relative pose
34.         cv::Mat E, inlier_mask;
35.         if (use_5pt)
36.         {
37.             E = cv::findEssentialMat(point_prev, point, f, c, cv::RANSAC, 0.99, 1, inlier_mask);
38.         }
39.         else
40.         {
41.             cv::Mat F = cv::findFundamentalMat(point_prev, point, cv::FM_RANSAC, 1, 0.99, inlier_mask);
42.             cv::Mat K = (cv::Mat_<double>(3, 3) << f, 0, c.x, 0, f, c.y, 0, 0, 1);
43.             E = K.t() * F * K;
44.         }
45.         cv::Mat R, t;
46.         int inlier_num = cv::recoverPose(E, point_prev, point, R, t, f, c, inlier_mask);

47.         // Accumulate relative pose if result is reliable
48.         if (inlier_num > min_inlier_num)
49.         {
50.             cv::Mat T = cv::Mat::eye(4, 4, R.type());
51.             T(cv::Rect(0, 0, 3, 3)) = R * 1.0;
52.             T.col(3).rowRange(0, 3) = t * 1.0;
53.             camera_pose = camera_pose * T.inv();
54.         }

55.         // Show the image and write camera pose
56.         ...
57.     }

58.     video.release();
59.     fclose(camera_traj);
60.     return 0;
61. }
```

$$\because \mathrm{P}_n^0 = \mathrm{P}_{n-1}^0 \begin{bmatrix} \mathrm{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}^{-1}$$

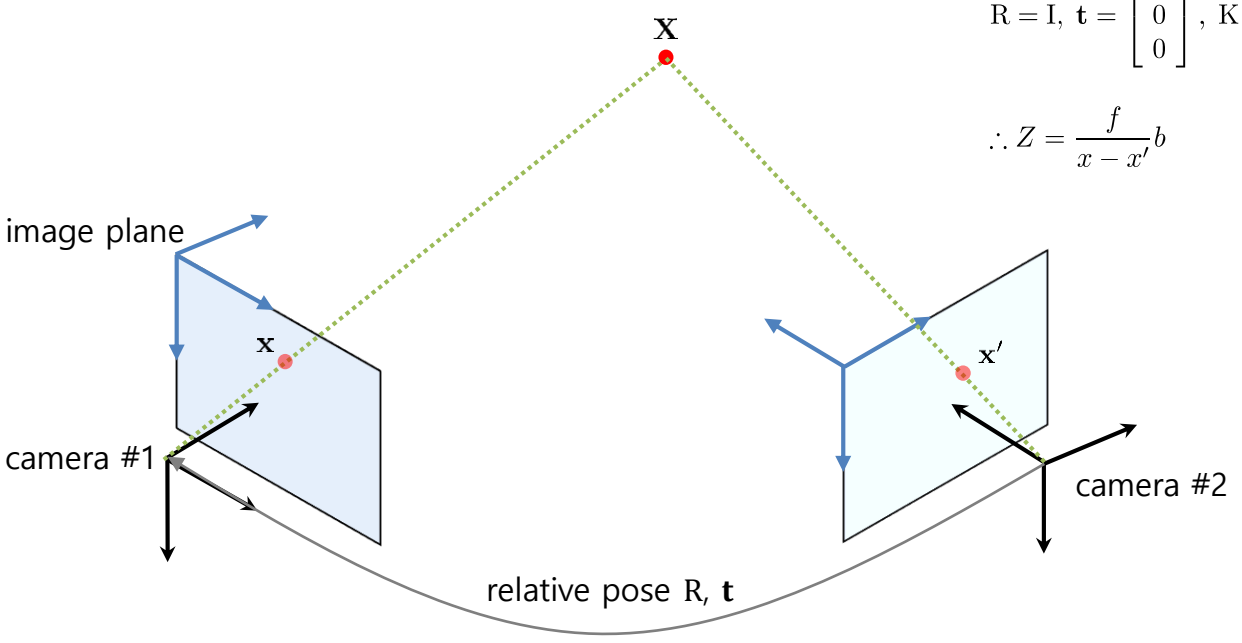
General 2D-2D Geometry (Epipolar Geometry)

- **Triangulation** (Point Localization)
  - Unknown: Position of a 3D point $\mathbf{X}$ (3 DoF)
  - Given: Point correspondence $(\mathbf{x}, \mathbf{x}')$, camera matrices $(\mathrm{K}, \mathrm{K}')$, and relative pose $(\mathrm{R}, \mathbf{t})$
  - Constraints: 2 x projection $\mathbf{x} = \mathrm{K}\,[\,\mathrm{I}\,|\,\mathbf{0}\,]\,\mathbf{X}$, $\mathbf{x}' = \mathrm{K}'\,[\,\mathrm{R}\,|\,\mathbf{t}\,]\,\mathbf{X}$
  - Solution (OpenCV): `cv::triangulatePoints()`

Special case) Stereo cameras

$$\mathrm{R} = \mathrm{I},\ \mathbf{t} = \begin{bmatrix} -b \\ 0 \\ 0 \end{bmatrix},\ \mathrm{K} = \mathrm{K}' = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$\therefore Z = \frac{f}{x - x'} b$$



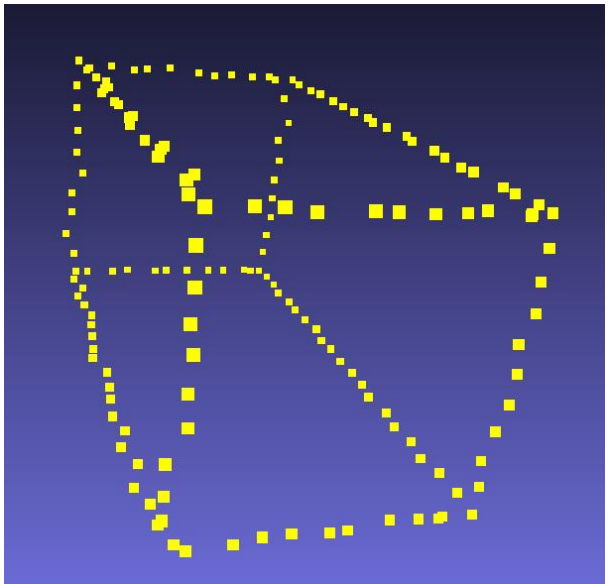
- Example: **Triangulation (Two-view Reconstruction)** [triangulation.cpp]


```
#include "opencv2/opencv.hpp"

int main()
{
    // cf. You need to run 'image_formation.cpp' to generate point observation.
    const char *input0 = "image_formation0.xyz", *input1 = "image_formation1.xyz";
    double f = 1000, cx = 320, cy = 240;

    // Load 2D points observed from two views
    std::vector<cv::Point2d> points0, points1;
    ...

    // Estimate relative pose of two views
    cv::Mat F = cv::findFundamentalMat(points0, points1, cv::FM_8POINT);
    cv::Mat K = (cv::Mat_<double>(3, 3) << f, 0, cx, 0, f, cy, 0, 0, 1);
    cv::Mat E = K.t() * F * K;
    cv::Mat R, t;
    cv::recoverPose(E, points0, points1, K, R, t);

    // Reconstruct 3D points (triangulation)
    cv::Mat P0 = K * cv::Mat::eye(3, 4, CV_64F);
    cv::Mat Rt, X;
    cv::hconcat(R, t, Rt);
    cv::Mat P1 = K * Rt;
    cv::triangulatePoints(P0, P1, points0, points1, X);
    X.row(0) = X.row(0) / X.row(3);
    X.row(1) = X.row(1) / X.row(3);
    X.row(2) = X.row(2) / X.row(3);
    X.row(3) = 1;

    // Store the 3D points
    FILE* fout = fopen("triangulation.xyz", "wt");
    if (fout == NULL) return -1;
    for (int c = 0; c < X.cols; c++)
        fprintf(fout, "%f %f %f\n", X.at<double>(0, c), X.at<double>(1, c), X.at<double>(2, c));
    fclose(fout);
    return 0;
}
```

$\because \mathbf{x} = \mathrm{K}\left[\mathrm{I} \mid \mathbf{0}\right]\mathbf{X}$

$\because \mathbf{x}' = \mathrm{K}'\left[\mathrm{R} \mid \mathbf{t}\right]\mathbf{X}$

Result: "triangulation.xyz"

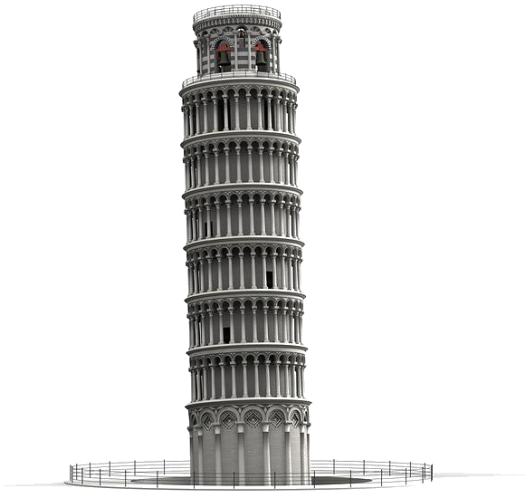


Correspondence Problem

Feature Correspondence

Robust Parameter Estimation

- RANSAC
- M-estimator



Images are from pixabay.

Quiz

600 pixels

What is it?

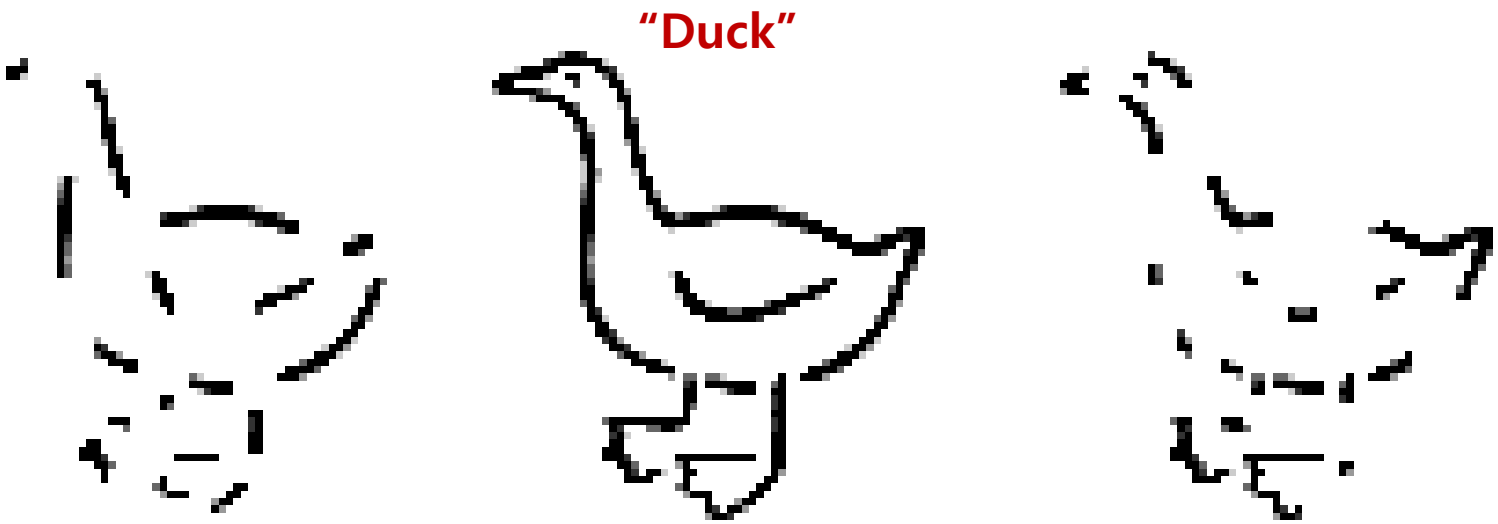
Quiz

What is it?

600 pixels

Quiz

- **Why corners (junction)?**
  - a.k.a. keypoints, interest points, salient points, and feature points
  - cf. ⊂ local invariant features (e.g. corner, edge, region, …)

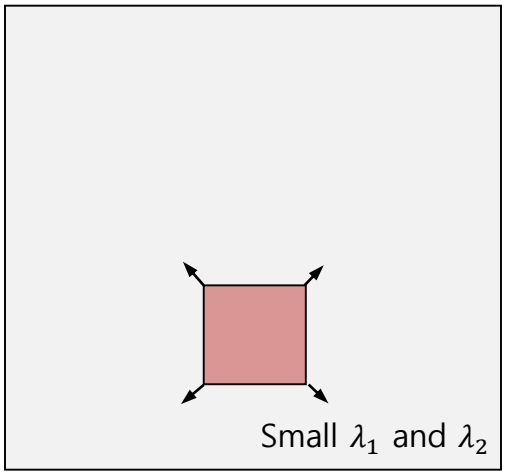


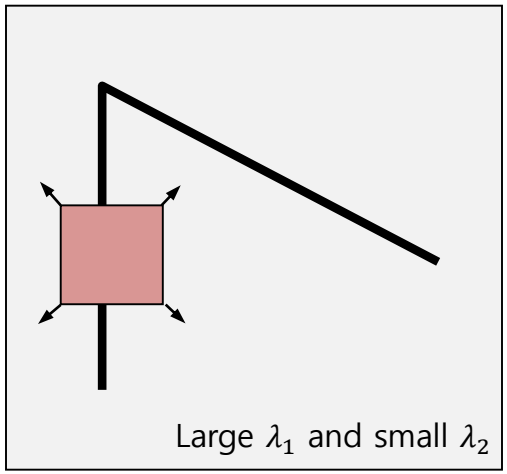
- **Requirements of local invariance features**
  - **repeatability** (invariance, robustness), **distinctiveness**, **locality** (due to occlusion)
  - quantity, accuracy, efficiency

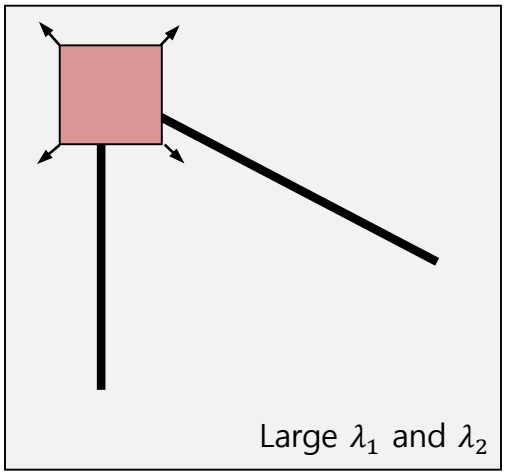
[Reference] Tuytelaars et al., **Local Invariant Feature Detectors: A Survey**, Foundations and Trends in Computer Graphics and Vision, 2008

Harris Corner (1988)

- Key idea: **Sliding window**







"flat" region:
no change
in all directions

"edge":
no change
along the edge direction

"corner":
significant change
in all directions

$$C(\Delta_x, \Delta_y) = \sum_{(x,y)\in W} \Big(I(x+\Delta_x, y+\Delta_y) - I(x,y)\Big)^2$$

$$\approx \begin{bmatrix}\Delta_x & \Delta_y\end{bmatrix} \underbrace{\begin{bmatrix}\sum_W I_x^2 & \sum_W I_x I_y \\ \sum_W I_x I_y & \sum_W I_y^2\end{bmatrix}}_{M} \begin{bmatrix}\Delta_x \\ \Delta_y\end{bmatrix}$$

c.f. $I(x+\Delta_x, y+\Delta_y) \approx I(x,y) + \begin{bmatrix}I_x(x,y) & I_y(x,y)\end{bmatrix} \begin{bmatrix}\Delta_x \\ \Delta_y\end{bmatrix}$ and $I_x = \dfrac{\partial I}{\partial x}$

Harris corner response:

$$R = \det(M) - k\,\mathrm{trace}(M)^2$$

c.f. $\det(M) = \lambda_1\lambda_2$, $\mathrm{trace}(M) = \lambda_1 + \lambda_2$, $k \in [0.04, 0.06]$

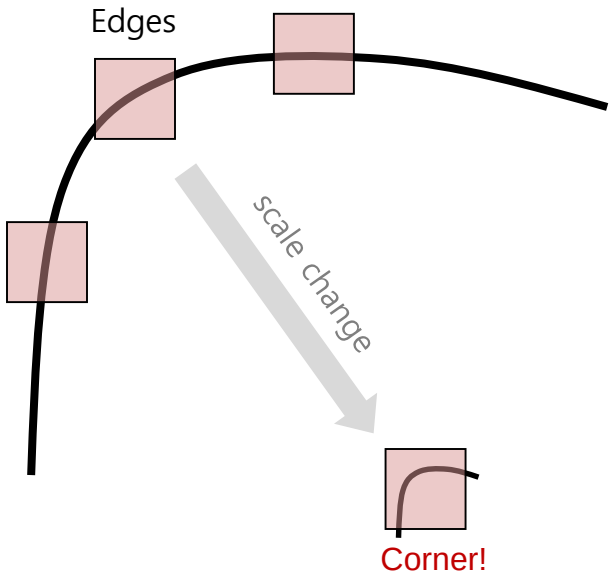
cf. Good-Feature-to-Track (Shi-Tomasi; 1994):

$$R = \min(\lambda_1, \lambda_2)$$

Images are from *Matching with Invariant Features* (by Frolova and Simakov, Lecture Slides, 2004).

Harris Corner (1988)

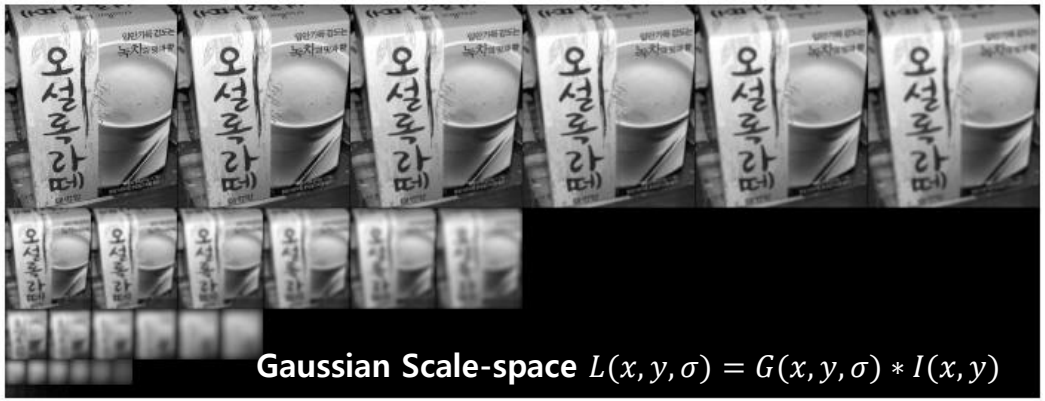
- Properties
  - Invariant to translation, rotation, and intensity shift ($I \rightarrow I + b$) ~~intensity scaling ($I \rightarrow aI$)~~
  - But variant to **image scaling**

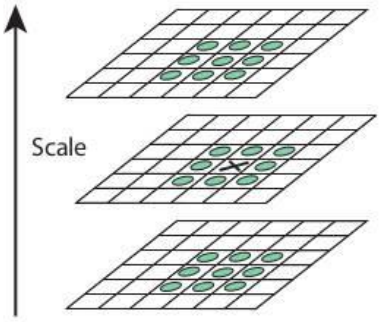


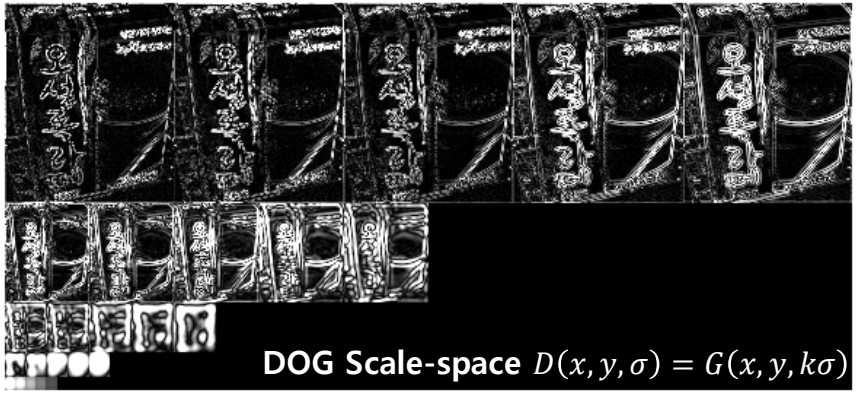
Images are from *Matching with Invariant Features* (by Frolova and Simakov, Lecture Slides, 2004).

SIFT (Scale-Invariant Feature Transform; 1999)

- Key idea: **Scale-space** (~ image pyramid)

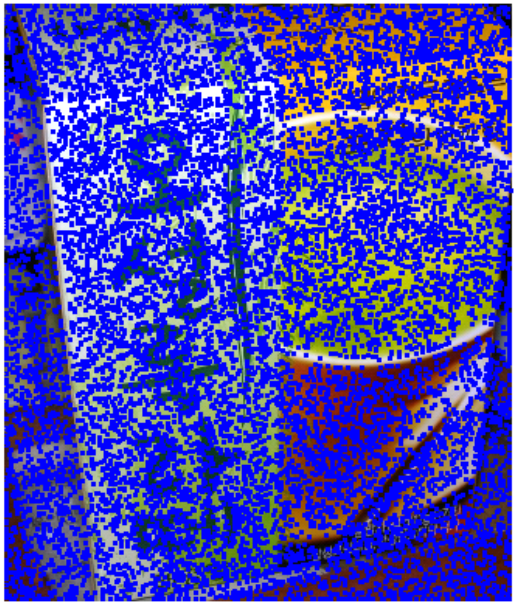






- Part #1) **Feature point detection**
  1. Find **local extrema** (minima and maxima) in DOG scale-space
  2. Localize its position accurately (sub-pixel level) using 3D quadratic function
  3. Eliminate **low contrast candidates**, $|D(\mathbf{x})| < \tau$
  4. Eliminate **candidates on edges**, $\frac{\mathrm{trace}(H)^2}{\det(H)} < \frac{(r+1)^2}{r}$ where $H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}$

SIFT (Scale-Invariant Feature Transform; 1999)



local extrema (N: 11479)

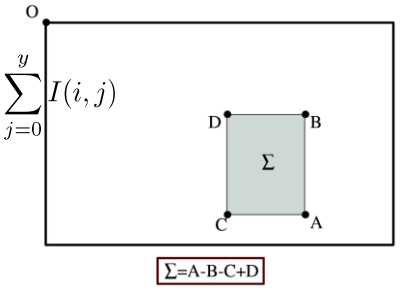
feature points (N: 971)

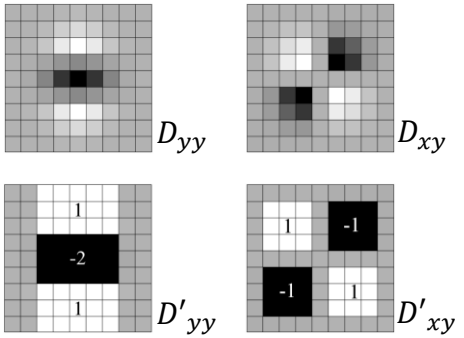
feature scales and orientations

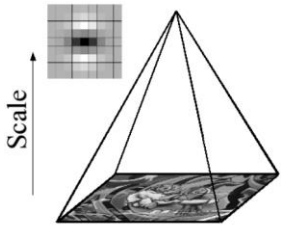
cf. SURF (Speeded Up Robust Features; 2006):

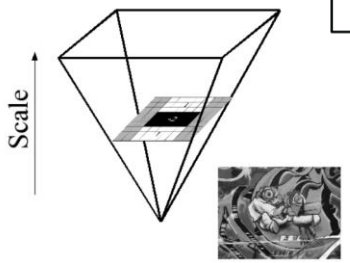
- Key idea: Approximation of SIFT using **integral image** and ...

$$S(x,y) = \sum_{i=0}^{x}\sum_{j=0}^{y} I(i,j)$$





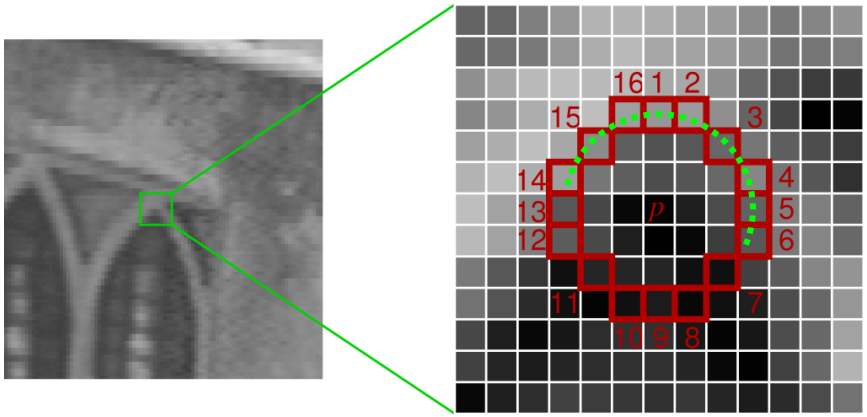




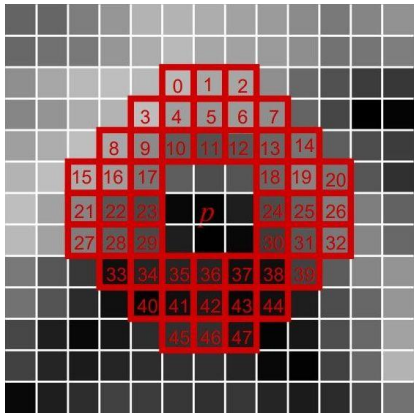
[Reference] Bay et. al., **SURF: Speeded Up Robust Features**, CVIU, 2008

FAST (Features from Accelerated Segment Test; 2006)

- Key idea: **Continuous arc of N or more pixels**
  - Is this patch a corner?
    - Is the segment brighter than $p + t$? Is the segment darker than $p - t$?
    - t: The threshold of similar intensity
  - Too many corners! it needs non-maximum suppression.



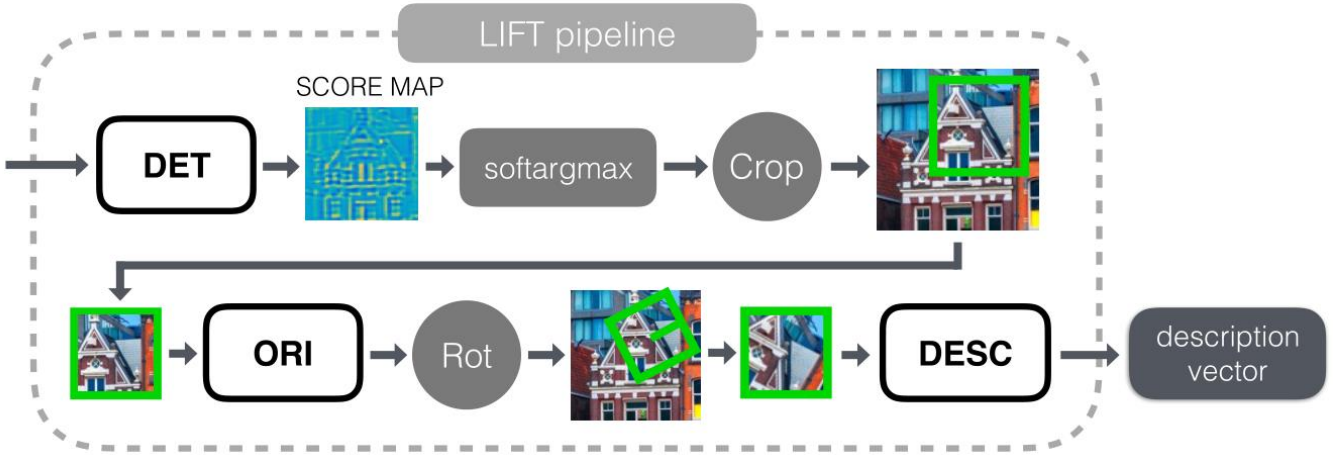
- Versions
  - **FAST-9** (N: 9), FAST-12 (N: 12), ...
  - FAST-**ER**: Training a decision tree to **e**nhance **r**epeatability with more pixels

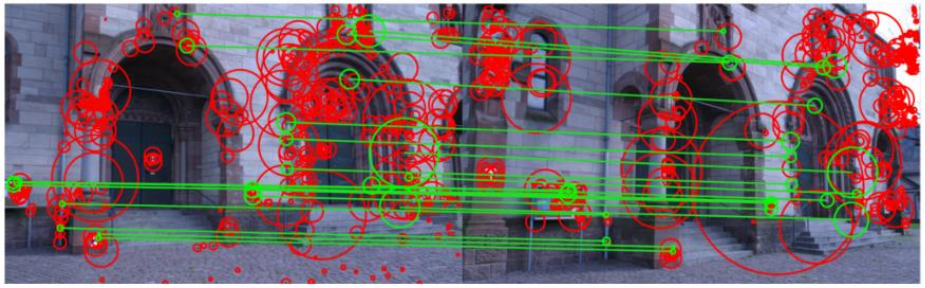


[Reference] Rosten et al., **FASTER and better: A machine learning approach to corner detection**, T-PAMI, 2010

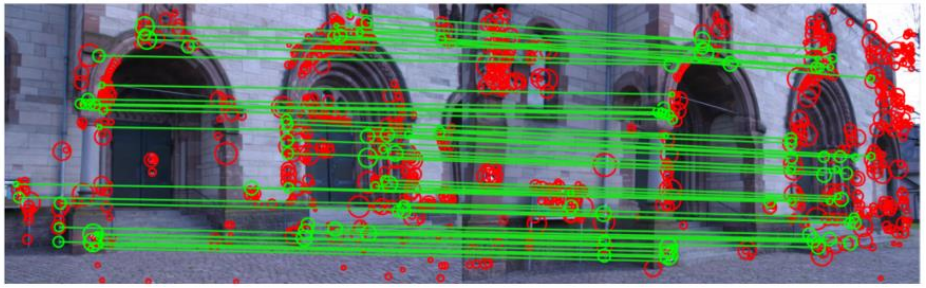
LIFT (Learned Invariant Feature Transform; 2016)

- Key idea: **Deep neural network**
  - **DET** (feature detector) + **ORI** (orientation estimator) + **DESC** (feature descriptor)





SIFT



LIFT

[Reference] Yi et. al., **LIFT: Learned Invariant Feature Transform**, ECCV, 2016

Overview of Feature Correspondence

- **Features**
  - **Corners**: Harris corner, GFTT (Shi-Tomasi corner), SIFT, SURF, FAST, LIFT, …
  - Edges, line segments, regions, …
- **Feature Descriptors and Matching**
  - **Patch**: Raw intensity
    - Measures: SSD (sum of squared difference), ZNCC (zero normalized cross correlation), …
  - **Floating-point descriptors**: SIFT, SURF, (DAISY), LIFT, … → e.g. A 128-dim. vector (a histogram of gradients)
    - Measures: Euclidean distance, cosine distance, (the ratio of first and second bests)
    - Matching: Brute-force matching ($O(N^2)$), ANN (approximated nearest neighborhood) search ($O(\log N)$)
    - Pros (+): **High discrimination power**
    - Cons (–): **Heavy computation**
  - **Binary descriptors**: BRIEF, ORB, (BRISK), (FREAK), … → e.g. A 128-bit string (a series of intensity comparison)
    - Measures: Hamming distance
    - Matching: Brute-force matching ($O(N^2)$)
    - Pros (+): **Less storage and faster extraction/matching**
    - Cons (–): **Less performance**
- **Feature Tracking (a.k.a. Optical Flow)**
  - **Optical flow**: (Horn-Schunck method), Lukas-Kanade method
    - Measures: SSD (sum of squared difference)
    - Tracking: Finding displacement of a similar patch
    - Pros (+): **No descriptor and matching** (faster and compact)
    - Cons (–): **Not working in wide baseline**

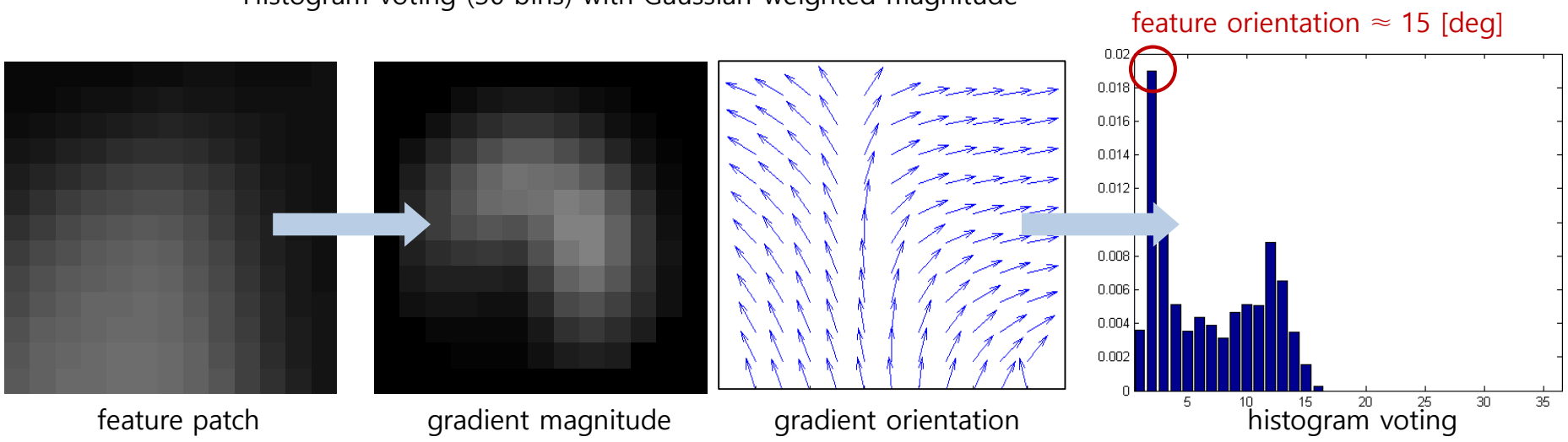
SIFT (Scale-Invariant Feature Transform; 1999)

- Part #2) **Orientation assignment**
  1. Derive magnitude and orientation of gradient of each patch

$$m(x,y) = \sqrt{\big(L(x+1,y) - L(x-1,y)\big)^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \tan^{-1}\frac{L(x,y+1) - L(x,y-1)}{L(x+1,y) - L(x-1,y)}$$

  2. Find the strongest orientation
     - Histogram voting (36 bins) with Gaussian-weighted magnitude

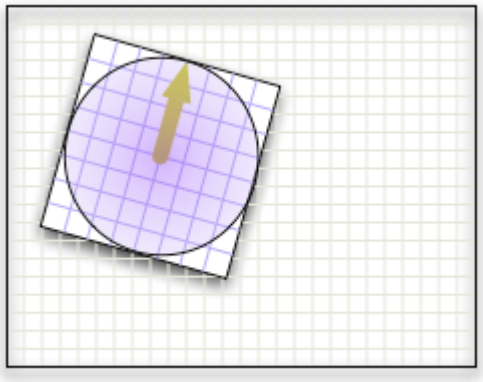


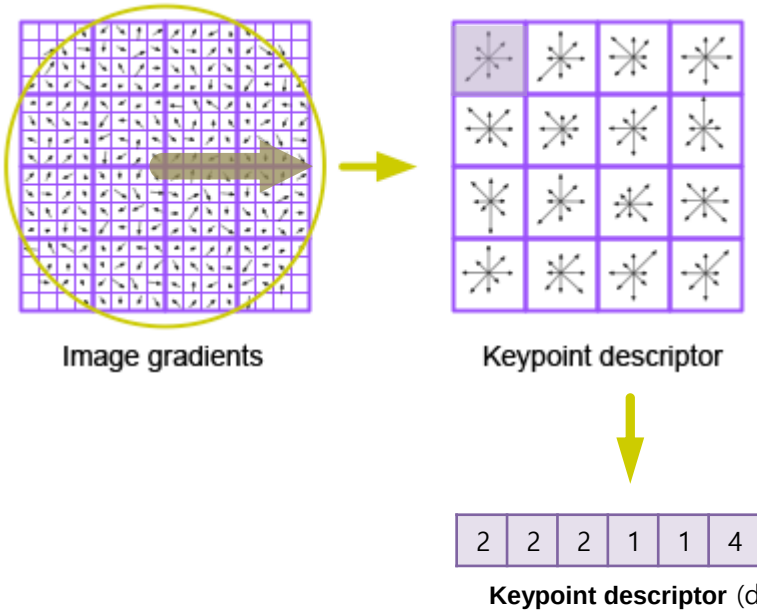
feature patch | gradient magnitude | gradient orientation | histogram voting

SIFT (Scale-Invariant Feature Transform; 1999)

- Part #3) **Feature descriptor extraction**
  - Build a 4x4 gradient histogram (8 bins) from each patch (16x16 pixels)
    - Use Gaussian-weighted magnitude again
    - Use relative angles w.r.t. the assigned feature orientation
  - Encode the histogram into a 128-dimensional vector

feature scales and orientations

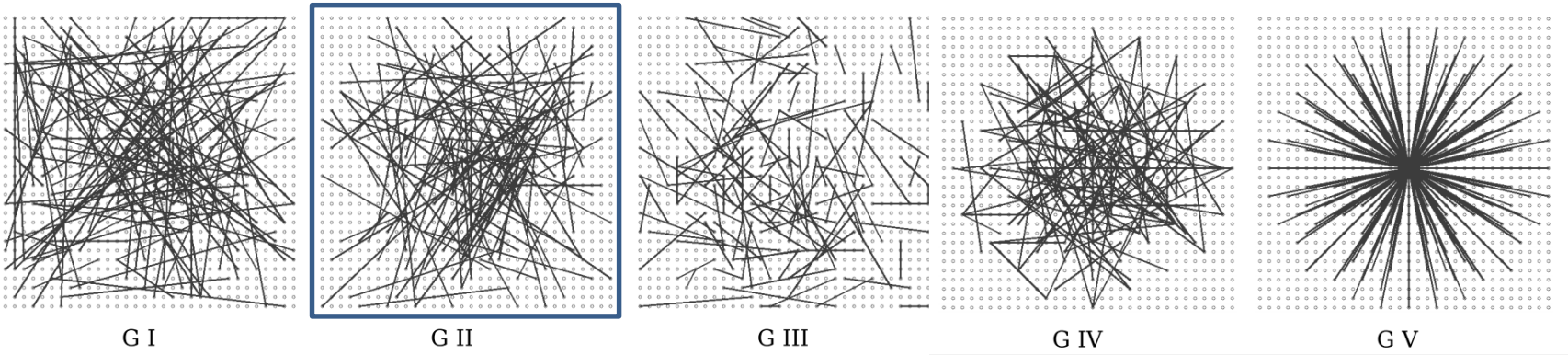




Images are from Jonas Hurrelmann (https://www.eecs.tu-berlin.de/fileadmin/fg144/Courses/06WS/scanning/Jonas/html/index.html).

BRIEF (Binary Robust Independent Elementary Features; 2010)

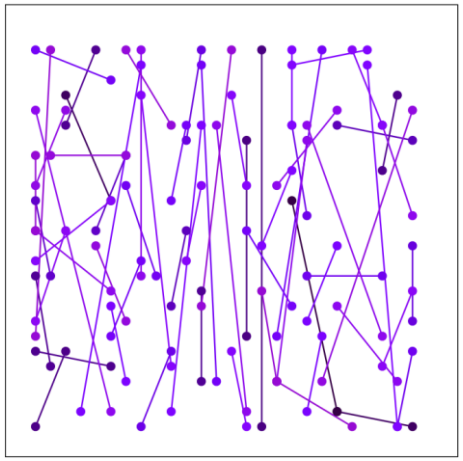
- Key idea: **A sequence of intensity comparison of random pairs**
  - Applying smoothing for stability and repeatability
  - Path size: 31 x 31 pixels



- Versions: The number of tests
  - BRIEF-32, BRIEF-64, BRIEF-128, BRIEF-256 …
- Examples of combinations
  - CenSurE detector (a.k.a. Star detector) + BRIEF descriptor
  - SURF detector + BRIEF descriptor

[Reference] Calonder et. al., **BRIEF: Computing a Local Binary Descriptor Very Fast**, T-PAMI, 2012

ORB (Oriented FAST and rotated BRIEF, 2011)



- Key idea: **Adding rotation invariance to BRIEF**
  - **Oriented FAST**
    - Generate scale pyramid for scale invariance
    - Detect *FAST-9* points (filtering with Harris corner response)
    - Calculate feature orientation by *intensity centroid*

      $\theta = \tan^{-1}\frac{m_{01}}{m_{10}}$ where $m_{pq} = \sum_{x,y} x^p y^q I(x,y)$
  - **Rotation-aware BRIEF**
    - Extract BRIEF descriptors w.r.t. the known orientation
    - Use better comparison pairs trained by greedy search

- Combination: **ORB**
  - FAST-9 detector (with orientation) + BRIEF-256 descriptor (with trained pairs)

- Computing time
  - ORB: **15.3 [msec]** / SURF: 217.3 [msec] / SIFT: 5228.7 [msec] @ 24 images (640x480) in Pascal dataset

[Reference] Rublee et. al., **ORB: An Efficient Alternative to SIFT or SURF**, ICCV, 2011

Lukas-Kanade Optical Flow

- Key idea: **Finding movement of a patch**

  Brightness constancy constraint: $I(x, y, t) = I(x + \Delta_x, y + \Delta_y, t + \Delta_t)$

  (if same patch)

  $$I_x \frac{\Delta_x}{\Delta_t} + I_y \frac{\Delta_y}{\Delta_t} + I_t = 0 \quad \text{because} \quad I(x + \Delta_x, y + \Delta_y, t + \Delta_t) \approx I(x, y, t) + I_x \Delta_x + I_y \Delta_y + I_t \Delta_t$$

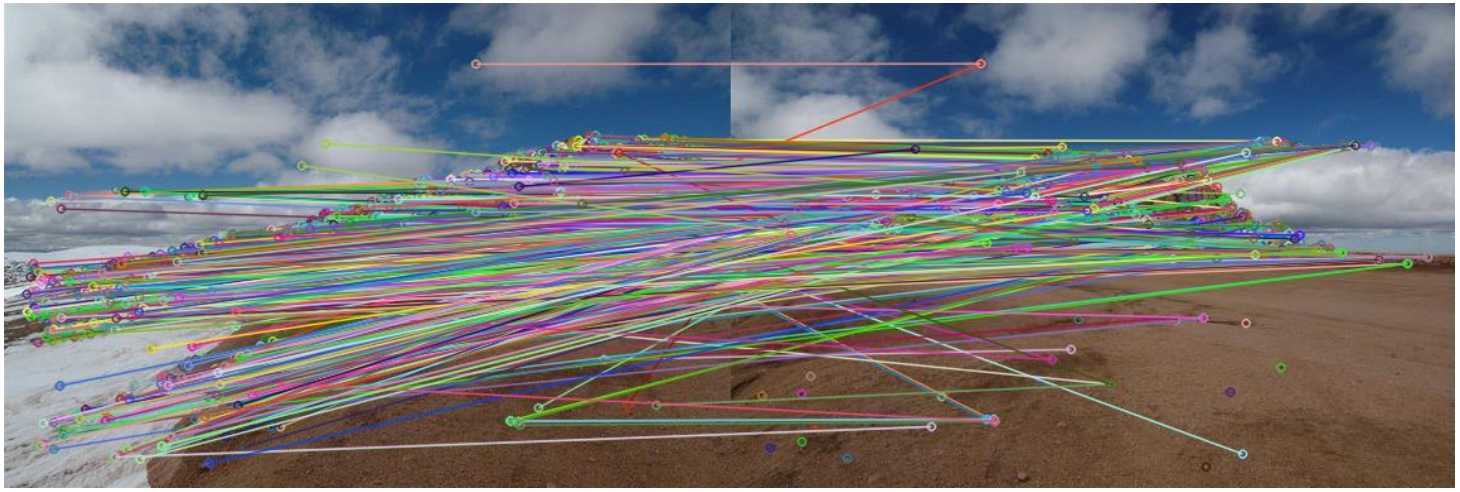
  $$\mathrm{A}\boldsymbol{v} = \boldsymbol{b} \quad \text{where} \quad \mathrm{A} = \begin{bmatrix} I_x(p_1) & I_y(p_1) \\ \vdots & \vdots \\ I_x(p_n) & I_y(p_n) \end{bmatrix}, \boldsymbol{v} = \begin{bmatrix} V_x \\ V_y \end{bmatrix}, \boldsymbol{b} = \begin{bmatrix} -I_t(p_1) \\ \vdots \\ -I_t(p_n) \end{bmatrix}, \text{ and } p_i \in W$$

  $$\therefore \boldsymbol{v} = \mathrm{A}^{\dagger} \boldsymbol{b} = (\mathrm{A}^{\mathrm{T}}\mathrm{A})^{-1}\mathrm{A}^{\mathrm{T}} \boldsymbol{b}$$

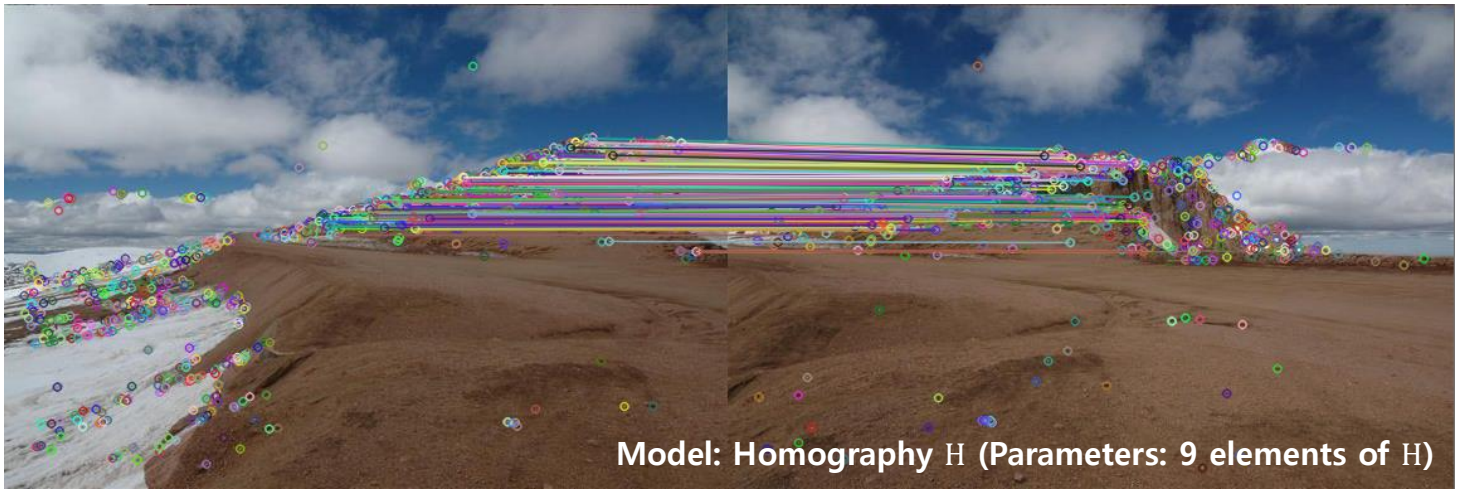
- Combination: **KLT tracker**
  - Shi-Tomasi detector (a.k.a. GFTT) + Lukas-Kanade optical flow

Why Outliers?

Putative matches (inliers + outliers)

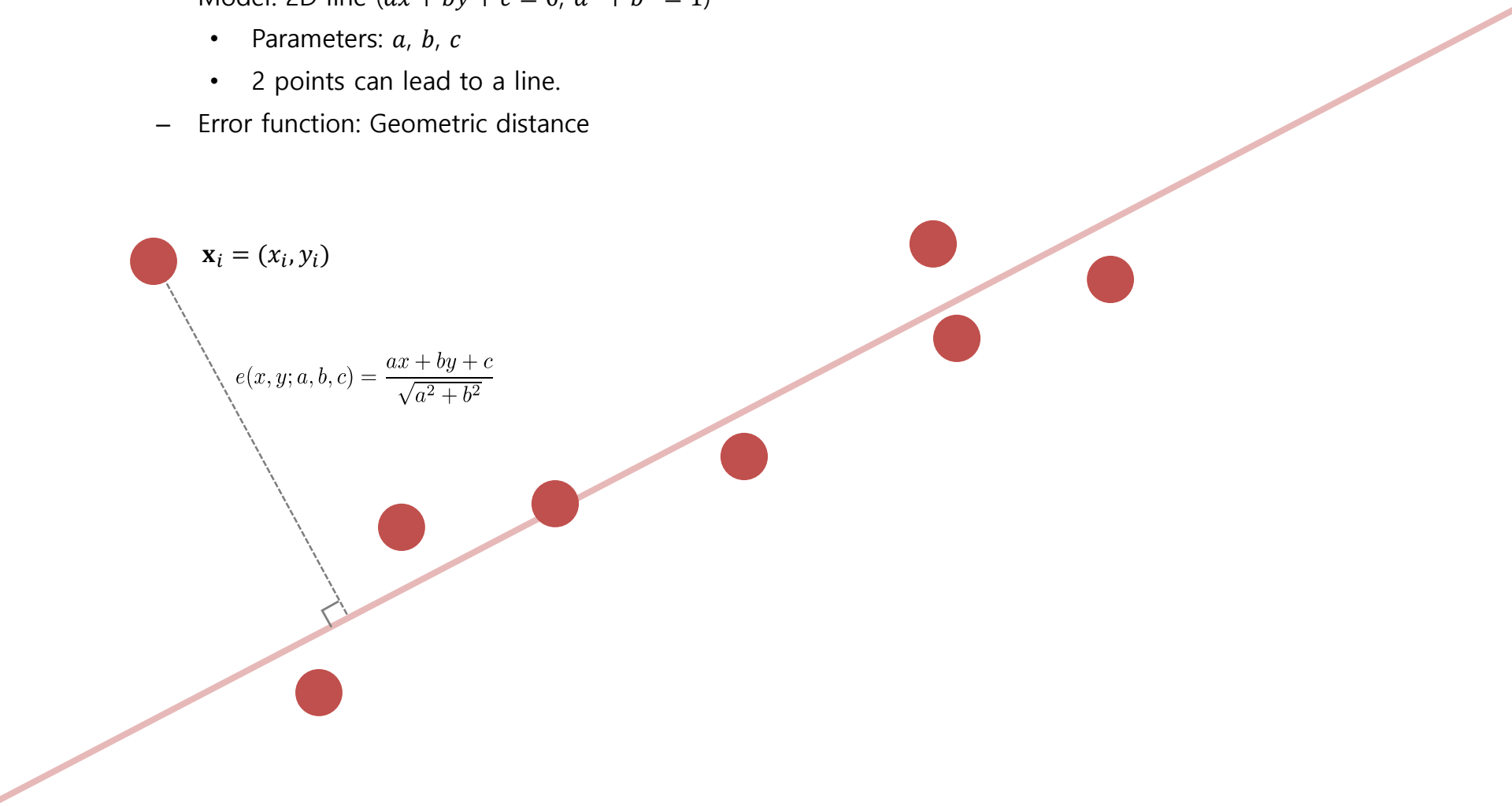


After applying RANSAC (inliers)

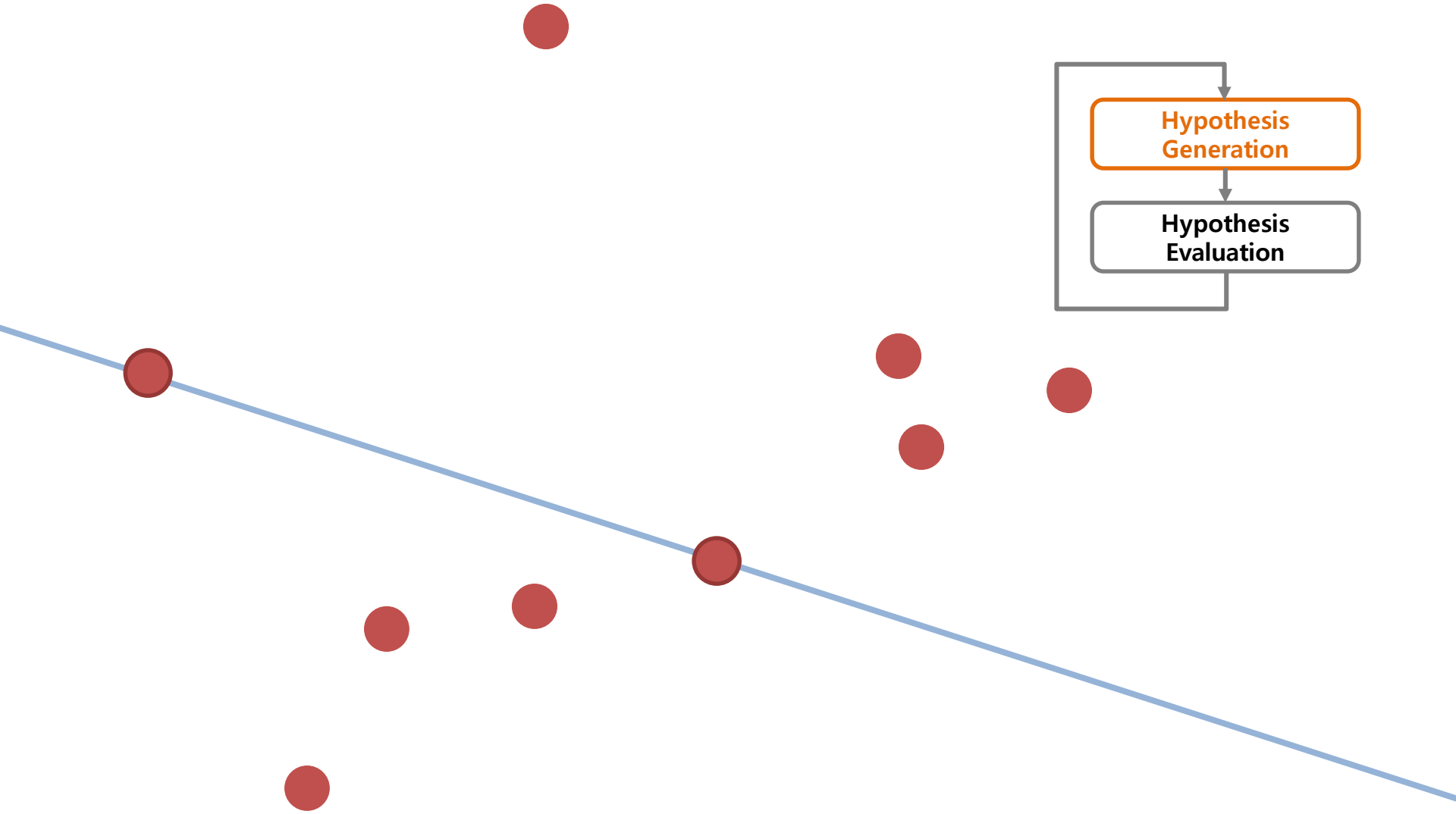


RANSAC: Random Sample Consensus

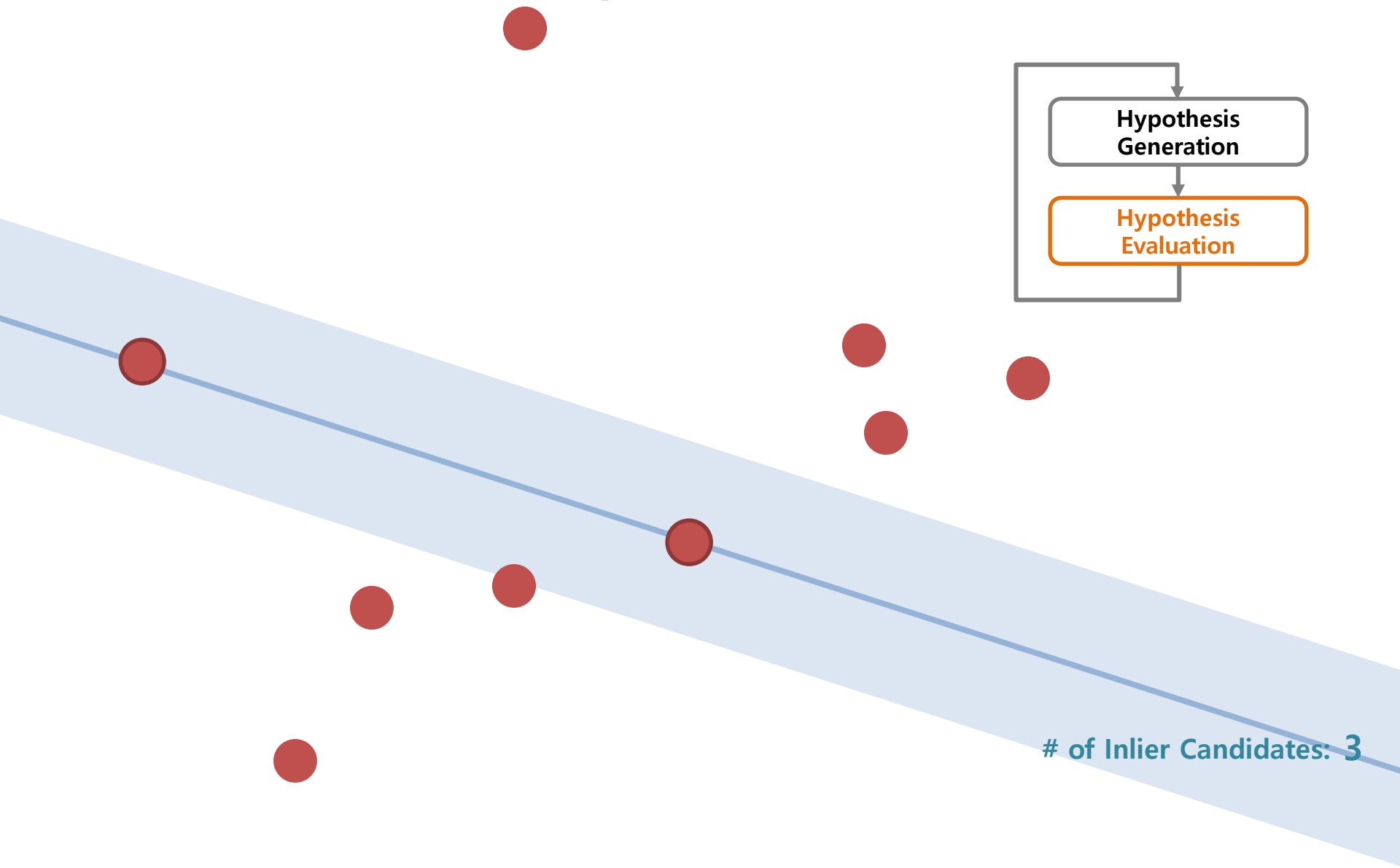
- Example: **Line Fitting with RANSAC**
  - Model: 2D line ($ax + by + c = 0$; $a^2 + b^2 = 1$)
    - Parameters: a, b, c
    - 2 points can lead to a line.
  - Error function: Geometric distance



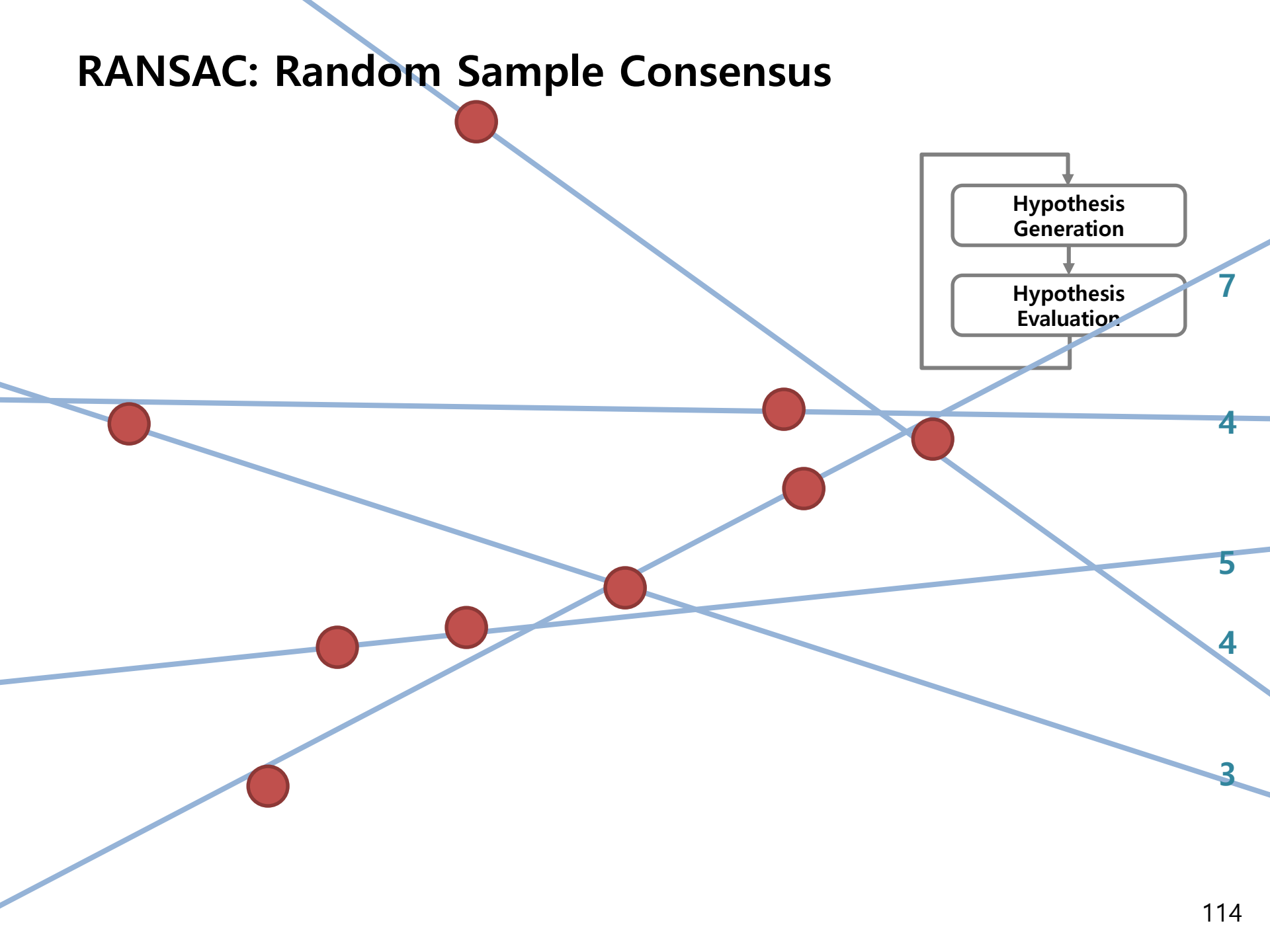
RANSAC: Random Sample Consensus



RANSAC: Random Sample Consensus



RANSAC: Random Sample Consensus

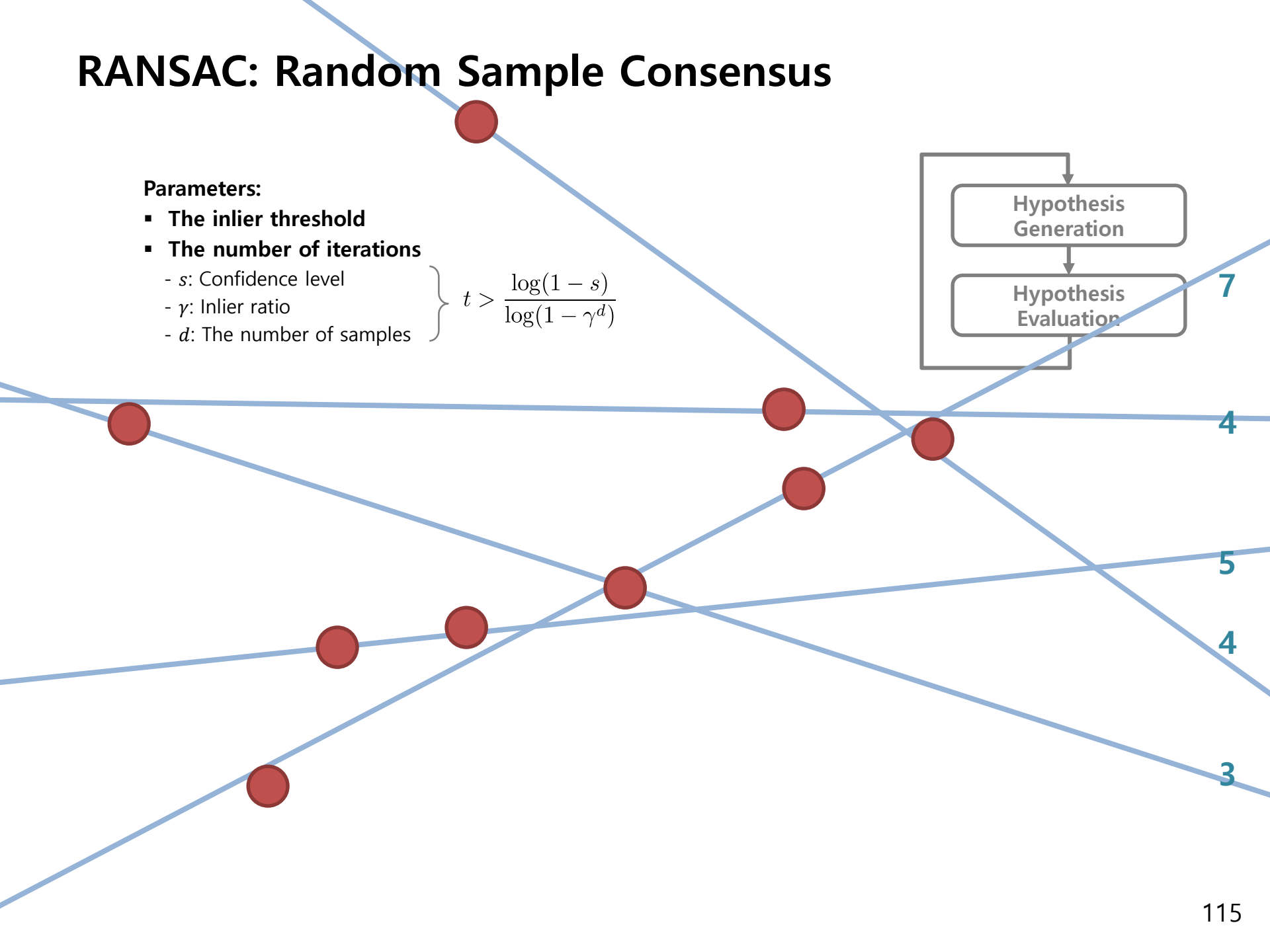


RANSAC: Random Sample Consensus

Parameters:

- **The inlier threshold**
- **The number of iterations**
  - s: Confidence level
  - γ: Inlier ratio
  - d: The number of samples

$$t > \frac{\log(1-s)}{\log(1-\gamma^d)}$$

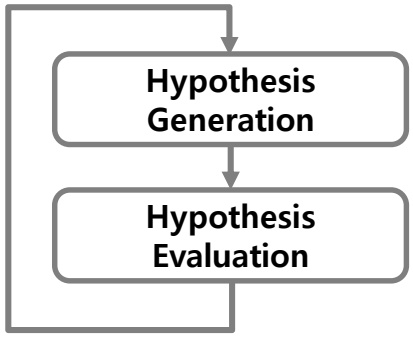


- Example: **Line Fitting with RANSAC** [line_fitting_ransac.cpp]


```
1. #include "opencv2/opencv.hpp"

2. // Convert a line format, [n_x, n_y, x_0, y_0] to [a, b, c]
3. // cf. A line model in OpenCV: n_x * (x - x_0) = n_y * (y - y_0)
4. #define CONVERT_LINE(line) (cv::Vec3d(line[0], -line[1], -line[0] * line[2] + line[1] * line[3]))

5. int main()
6. {
7.     cv::Vec3d truth(1.0 / sqrt(2.0), 1.0 / sqrt(2.0), -240.0); // The line model: a*x + b*y + c = 0 (a^2 + b^2 = 1)
8.     int ransac_trial = 50, ransac_n_sample = 2;
9.     double ransac_thresh = 3.0; // 3 x 'data_inlier_noise'
10.    int data_num = 1000;
11.    double data_inlier_ratio = 0.5, data_inlier_noise = 1.0;

12.    // Generate data
13.    std::vector<cv::Point2d> data;
14.    cv::RNG rng;
15.    for (int i = 0; i < data_num; i++)
16.    {
17.        if (rng.uniform(0.0, 1.0) < data_inlier_ratio)
18.        {
19.            double x = rng.uniform(0.0, 480.0);
20.            double y = (truth(0) * x + truth(2)) / -truth(1);
21.            x += rng.gaussian(data_inlier_noise);
22.            y += rng.gaussian(data_inlier_noise);
23.            data.push_back(cv::Point2d(x, y)); // Inlier
24.        }
25.        else data.push_back(cv::Point2d(rng.uniform(0.0, 640.0), rng.uniform(0.0, 480.0))); // Outlier
26.    }

27.    // Estimate a line using RANSAC ...
55.    // Estimate a line using least-squares method (for reference) ...

59.    // Display estimates
60.    printf("* The Truth: %.3f, %.3f, %.3f\n", truth[0], truth[1], truth[2]);
61.    printf("* Estimate (RANSAC): %.3f, %.3f, %.3f (Score: %d)\n", best_line[0], best_line[1], ..., best_score);
62.    printf("* Estimate (LSM): %.3f, %.3f, %.3f\n", lsm_line[0], lsm_line[1], lsm_line[2]);
63.    return 0;
64. }
```

c.f. $t > \frac{\log(1-s)}{\log(1-\gamma^d)} = \frac{\log(1-0.999)}{\log(1-0.5^2)} = 24$


```
27.     // Estimate a line using RANSAC
28.     int best_score = -1;
29.     cv::Vec3d best_line;
30.     for (int i = 0; i < ransac_trial; i++)
31.     {
32.         // Step 1: Hypothesis generation
33.         std::vector<cv::Point2d> sample;
34.         for (int j = 1; j < ransac_n_sample; j++)
35.         {
36.             int index = rng.uniform(0, int(data.size()));
37.             sample.push_back(data[index]);
38.         }
39.         cv::Vec4d nnxy;
40.         cv::fitLine(sample, nnxy, CV_DIST_L2, 0, 0.01, 0.01);
41.         cv::Vec3d line = CONVERT_LINE(nnxy);

42.         // Step 2: Hypothesis evaluation
43.         int score = 0;
44.         for (size_t j = 0; j < data.size(); j++)
45.         {
46.             double error = fabs(line(0) * data[j].x + line(1) * data[j].y + line(2));
47.             if (error < ransac_thresh) score++;
48.         }

49.         if (score > best_score)
50.         {
51.             best_score = score;
52.             best_line = line;
53.         }
54.     }

55.     // Estimate a line using least squares method (for reference)
56.     cv::Vec4d nnxy;
57.     cv::fitLine(data, nnxy, CV_DIST_L2, 0, 0.01, 0.01);
58.     cv::Vec3d lsm_line = CONVERT_LINE(nnxy);
```



Line Fitting Result

```
* The Truth: 0.707, 0.707, -240.000
* Estimate (RANSAC): 0.712, 0.702, -242.170 (Score: 434)
* Estimate (LSM): 0.748, 0.664, -314.997
```

Least Squares Method, RANSAC, and M-estimator

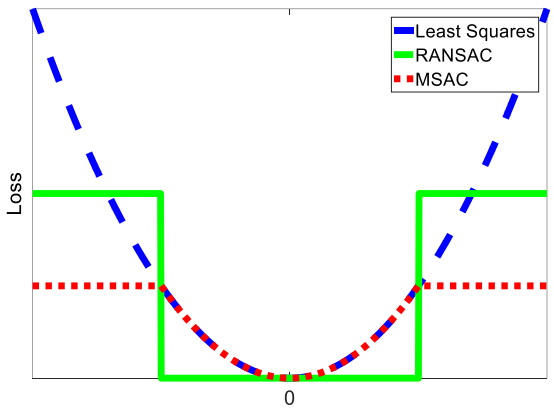
- **Least Squares Method**
  - Find a model while minimizing sum of squared errors, $\arg\min_{a,b,c} \sum_i e(\mathbf{x}_i; a, b, c)^2$

- **RANSAC**
  - Find a model while maximizing the number of supports (~ inlier candidates)
    ~ minimizing the number of outlier candidates

- **Why RANSAC was robust to outliers?**

  Problem: $\arg\min_{a,b,c} \sum_i \rho\big(e(\mathbf{x}_i; a, b, c)\big)$

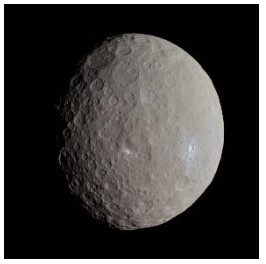
  In the view of **loss functions** ρ,
  - Least squares method: $\rho(x) = x^2$
  - RANSAC: $\rho(x) = \begin{cases} 0 & \text{if } |x| < \tau \\ 1 & \text{o.w.} \end{cases}$



- **M-estimator** (~ weighted least squares) | **MSAC** (in OpenCV)
  - Find a model while minimizing sum of (squared) errors **with a truncated loss function**

[Reference] Choi et al., **Performance Evaluation of RANSAC Family**, BMVC, 2009

Ceres (an asteroid)



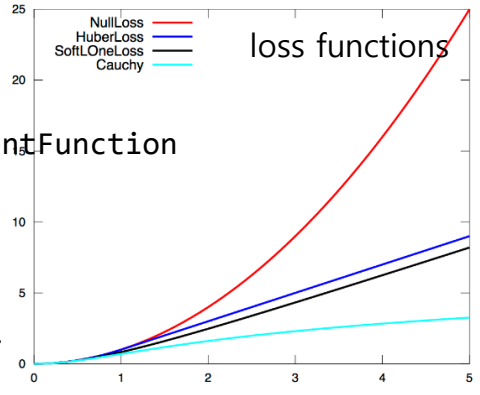
One-page Tutorial for Ceres Solver

- **Ceres Solver?**
  - An **open source C++** library for modelling and solving large and complicated **optimization** problems.
    - Since 2010 by Google (BSD license)
  - Problem types: 1) **Non-linear least squares** (with bounds), 2) General unconstrained minimization
  - Homepage: http://ceres-solver.org/

- **Solving Non-linear Least Squares** $\arg\min_{\mathbf{m}} \sum_i \rho_i\left(\|r_i(\mathbf{m})\|^2\right)$
  1. Define residual functions (or cost function or error function)
  2. Instantiate `ceres::Problem` and add residuals using its member function, `AddResidualBlock()`
     - Instantiate each residual r_i in the form of `ceres::CostFunction` and add it
       - Select how to calculate its derivative (Jacobian)

         (`ceres::AutoDiffCostFunction` or `ceres::NumericDiffCostFunction` or `ceres::SizedCostFunction`)
       - cf. Automatic derivation (using the chain rule) is recommended for convenience and performance.
     - Instantiate its `ceres::LossFunction` ρ_i and add it (if the problem needs robustness against outliers)
  3. Instantiate `ceres::Solver::Options` (and also `ceres::Solver::Summary`) and configure the option
  4. Run `ceres::Solve()`

- **Solving General Minimization** $\arg\min_{\mathbf{x}} f(\mathbf{x})$
  - `ceres::CostFunction` → `ceres::FirstOrderFunction`, `ceres::GradientFunction`
  - `ceres::Problem` → `ceres::GradientProblem`
  - `ceres::Solver` → `ceres::GradientProblemSolver`



Images are from Wikipedia (https://en.wikipedia.org/wiki/Ceres_(dwarf_planet)) and Ceres Solver

- Example: **Line Fitting with M-estimator** [line_fitting_m_est.cpp]

```
1. #include "opencv2/opencv.hpp"
2. #include "ceres/ceres.h"
3. ...
4. struct GeometricError
5. {
6.     GeometricError(const cv::Point2d& pt) : datum(pt) { }
7.     template<typename T>
8.     bool operator()(const T* const line, T* residual) const
9.     {
10.        residual[0] = (line[0] * T(datum.x) + line[1] * T(datum.y) + line[2]) / sqrt(line[0] * line[0] + line[1] * line[1]);
11.        return true;
12.    }
13. private:
14.    const cv::Point2d datum;
15. };

16. int main()
17. {
18.    ...
19.    // Estimate a line using M-estimator
20.    cv::Vec3d opt_line(1, 0, 0);
21.    ceres::Problem problem;
22.    for (size_t i = 0; i < data.size(); i++)
23.    {
24.        ceres::CostFunction* cost_func = new ceres::AutoDiffCostFunction<GeometricError, 1, 3>(new GeometricError(data[i]));
25.        ceres::LossFunction* loss_func = NULL;
26.        if (loss_width > 0) loss_func = new ceres::CauchyLoss(loss_width);
27.        problem.AddResidualBlock(cost_func, loss_func, opt_line.val);
28.    }
29.    ceres::Solver::Options options;
30.    options.linear_solver_type = ceres::ITERATIVE_SCHUR;
31.    options.num_threads = 8;
32.    options.minimizer_progress_to_stdout = true;
33.    ceres::Solver::Summary summary;
34.    ceres::Solve(options, &problem, &summary);
35.    std::cout << summary.FullReport() << std::endl;
36.    opt_line /= sqrt(opt_line[0] * opt_line[0] + opt_line[1] * opt_line[1]); // Normalize
37.    ...
38.    return 0;
39. }
```

1) Define a residual as C++ generic function (`T` ~ `double`)
cf. The generic is necessary for automatic differentiation.

c.f. $e(x, y; a, b, c) = \frac{ax + by + c}{\sqrt{a^2 + b^2}}$

2) Instantiate a problem and add a residual for each datum

The dimension of a residual
The dimension of the first model parameter

3) Instantiate options and configure it

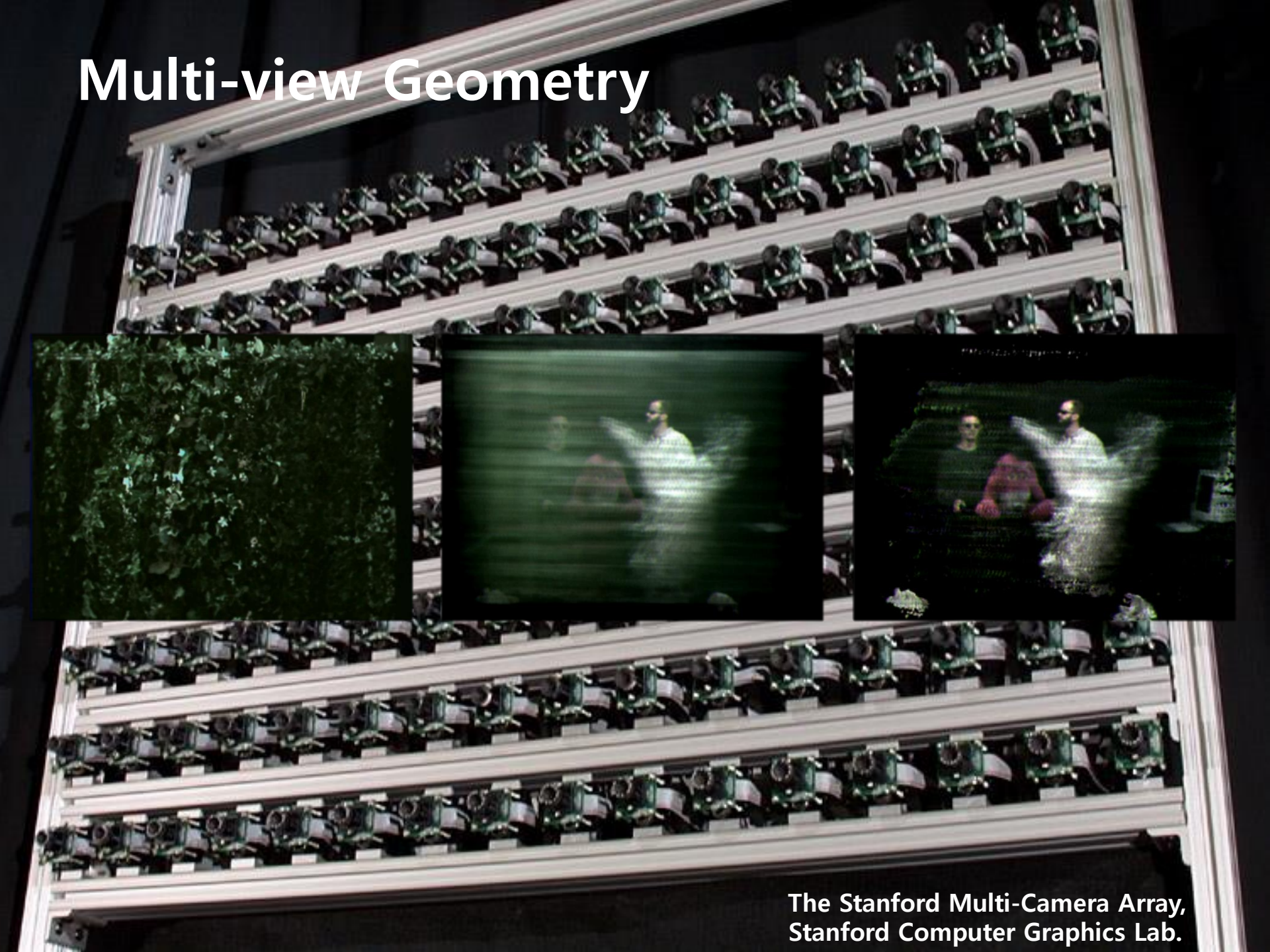
4) Solve the minimization problem

Overview of Robust Parameter Estimation

- Bottom-up Approaches (~ Voting) e.g. line fitting, relative pose estimation
  - **Hough transform**
    - A datum votes multiple parameter candidates.
      - cf. The parameter space is maintained as a multi-dimensional histogram (discretization).
    - Score: The number of hits by data
    - Selection: Finding a peak on the histogram after voting
  - **RANSAC family**
    - A sample of data votes a single parameter candidate.
    - Score: The number of inlier candidates (whose error is within threshold)
    - Selection: Keeping the best model during RANSAC's iterations
      - cf. RANSAC involves many iterations of parameter estimation and error calculation.

- Top-down Approaches e.g. graph SLAM, multi-view reconstruction
  - **M-estimator**
    - All data aims to find the best parameter (from its initial guess).
    - Score: A cost function
      - The cost function includes a truncated loss function.
    - Selection: Minimizing the cost function (following its gradient)
      - cf. Nonlinear optimization is computationally heavy and leads to a local minima.

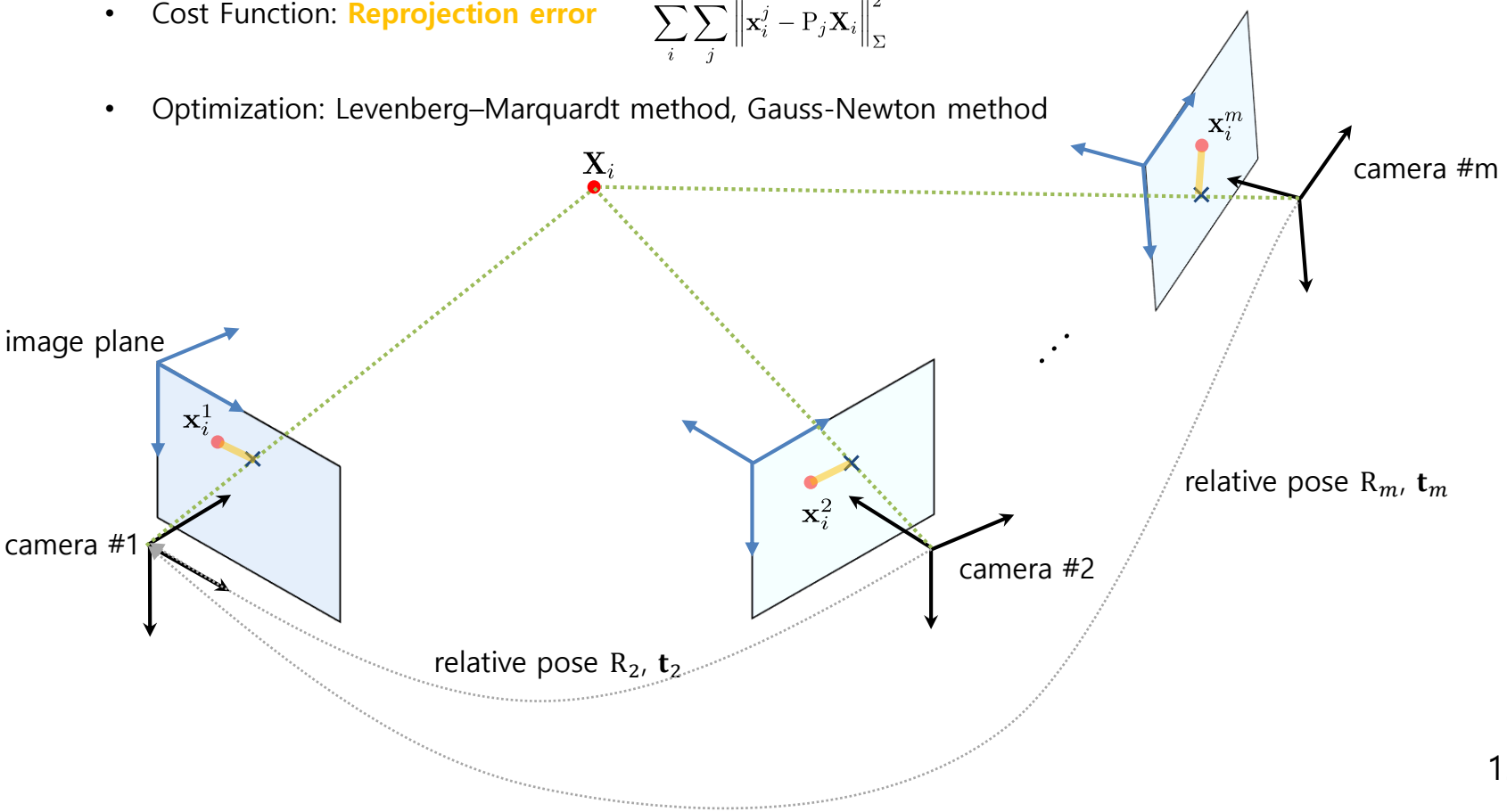
Multi-view Geometry



The Stanford Multi-Camera Array,
Stanford Computer Graphics Lab.

Bundle Adjustment

- **Bundle Adjustment**
  - Unknown: Position of 3D points and each camera's relative pose ($6n + 3m$ DoF)
  - Given: Point correspondence, camera matrices, position of 3D points, and each camera's relative pose
    cf. initial values
  - Constraints: n x m x projection $\mathbf{x}_i^j = \mathrm{P}_j \mathbf{X}_i = \mathrm{K}_j \left[\mathrm{R}_j \mid \mathbf{t}_j \right] \mathbf{X}_i$
  - Solution: **Non-linear least-square optimization**
    - Cost Function: **Reprojection error** $\sum_i^n \sum_j^m \left\| \mathbf{x}_i^j - \mathrm{P}_j \mathbf{X}_i \right\|_\Sigma^2$
    - Optimization: Levenberg–Marquardt method, Gauss-Newton method



Bundle Adjustment

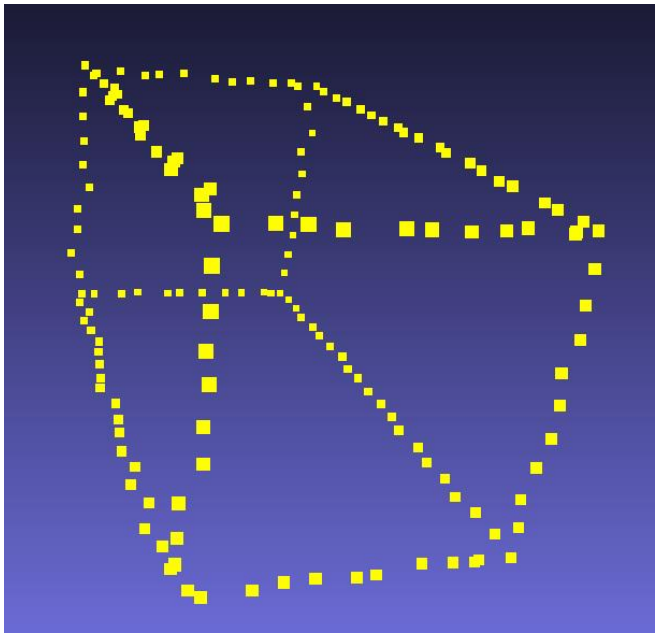
- **Bundle Adjustment**
  - Unknown: Position of 3D points and each camera's relative pose ($6n + 3m$ DoF)
  - Given: Point correspondence, camera matrices, position of 3D points, and each camera's relative pose
    cf. initial values
  - Constraints: n x m x projection $\mathbf{x}_i^j = \mathrm{P}_j \mathbf{X}_i = \mathrm{K}_j \left[\mathrm{R}_j \mid \mathbf{t}_j \right] \mathbf{X}_i$
  - Solution: **Non-linear least-square optimization**
    - Cost Function: **Reprojection error** $\sum_i^n \sum_j^m v_i^j \left\| \mathbf{x}_i^j - \mathrm{P}_j \mathbf{X}_i \right\|_\Sigma^2$ where $v_i^j = \begin{cases} 1 & \text{if } \mathbf{x}_i^j \text{ is visible} \\ 0 & \text{otherwise} \end{cases}$
    - Optimization: Levenberg–Marquardt method, Gauss-Newton method

- **Bundle Adjustment and Optimization Tools**
  - OpenCV: No function (but cvsba is available as a sba wrapper.)
  - Bundle adjustment: sba (Sparse Bundle Adjustment), SSBA (Simple Sparse Bundle Adjustment), pba (Multicore Bundle Adjustment)
  - (Graph) optimization: g2o (General Graph Optimization), iSAM (Incremental Smoothing and Mapping), GTSAM, Ceres Solver (Google)

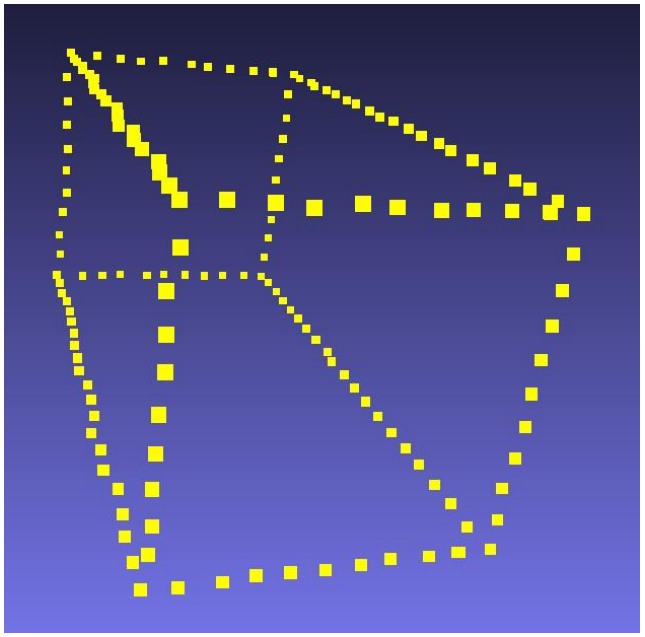
Bundle Adjustment

- Example: **Bundle Adjustment (Global)** `[bundle_adjustment_global.cpp, bundle_adjustment.hpp]`

Result: "triangulation.xyz"

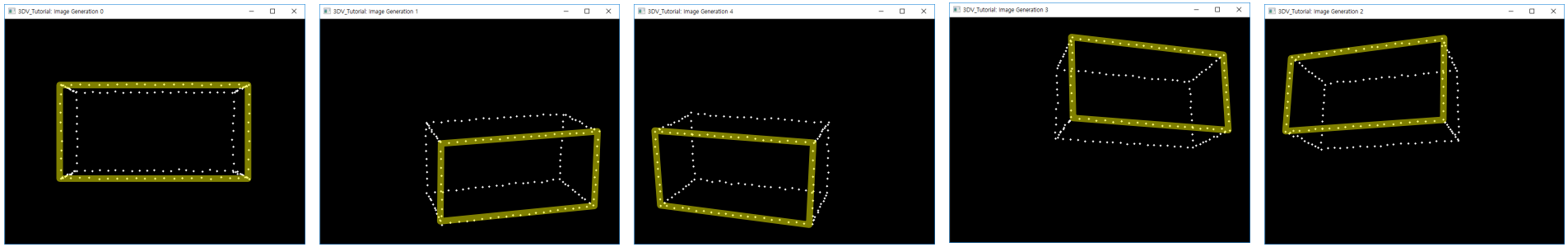


Result: "bundle_adjustment_global.xyz"



[Given]

- **Intrinsic parameters**: camera matrices K_j (all cameras has same and fixed camera matrix.)
- **2D point correspondence** $\mathbf{x}_i^j$ (all points are visible on all camera views.)



[Unknown]

- **Extrinsic parameters**: camera poses

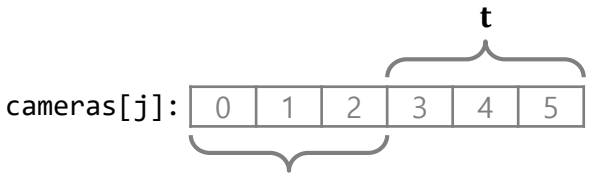
$$\mathrm{R}_0 = \mathrm{I}_{3x3} \quad \mathbf{t}_0 = [0,0,0]^\top \qquad \mathrm{R}_1 = \mathrm{I}_{3x3} \quad \mathbf{t}_1 = [0,0,0]^\top \qquad \mathrm{R}_2 = \mathrm{I}_{3x3} \quad \mathbf{t}_2 = [0,0,0]^\top \qquad \mathrm{R}_3 = \mathrm{I}_{3x3} \quad \mathbf{t}_3 = [0,0,0]^\top \qquad \mathrm{R}_4 = \mathrm{I}_{3x3} \quad \mathbf{t}_4 = [0,0,0]^\top$$

- **3D points** $\mathbf{X}_i = [0,0,5.5]^\intercal$ initial values

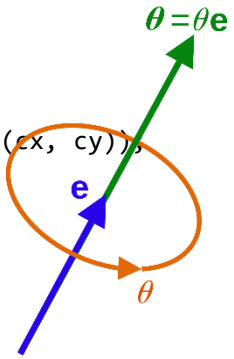
Ceres Solver (non-linear least-square optimization)

$$\sum_i^n \sum_j^m v_i^j \left\| \mathbf{x}_i^j - \mathrm{P}_j \mathbf{X}_i \right\|_\Sigma^2 \quad \text{where} \quad v_i^j = 1 \ (\because \text{all points are visible})$$

```cpp
#include "bundle_adjustment.hpp"

int main()
{
    // cf. You need to run 'image_formation.cpp' to generate point observation.
    const char* input = "image_formation%d.xyz";
    int input_num = 5;
    double f = 1000, cx = 320, cy= 240;

    // Load 2D points observed from multiple views
    std::vector<std::vector<cv::Point2d>> xs;
    ...

    // Initialize cameras and 3D points
    std::vector<cv::Vec6d> cameras(xs.size());
    std::vector<cv::Point3d> Xs(xs.front().size(), cv::Point3d(0, 0, 5.5));

    // Run bundle adjustment
    ceres::Problem ba;
    for (size_t j = 0; j < xs.size(); j++)
    {
        for (size_t i = 0; i < xs[j].size(); i++)
        {
            ceres::CostFunction* cost_func = ReprojectionError::create(xs[j][i], f, cv::Point2d(cx, cy));
            double* view = (double*)(&(cameras[j]));
            double* X = (double*)(&(Xs[i]));
            ba.AddResidualBlock(cost_func, NULL, view, X);
        }
    }
    ceres::Solver::Options options;
    options.linear_solver_type = ceres::ITERATIVE_SCHUR;
    options.num_threads = 8;
    options.minimizer_progress_to_stdout = true;
    ceres::Solver::Summary summary;
    ceres::Solve(options, &ba, &summary);

    // Store the 3D points and camera pose to an XYZ file
    ...
        cv::Vec3d rvec(cameras[j][0], cameras[j][1], cameras[j][2]), t(cameras[j][3], cameras[j][4], cameras[j][5]);
        cv::Matx33d R;
        cv::Rodrigues(rvec, R);
    ...
    return 0;
}
```



Axis-angle representation of R
(a.k.a. Rodrigues' rotation formula)



Images are from Wikipedia (https://en.wikipedia.org/wiki/Axis–angle_representation).

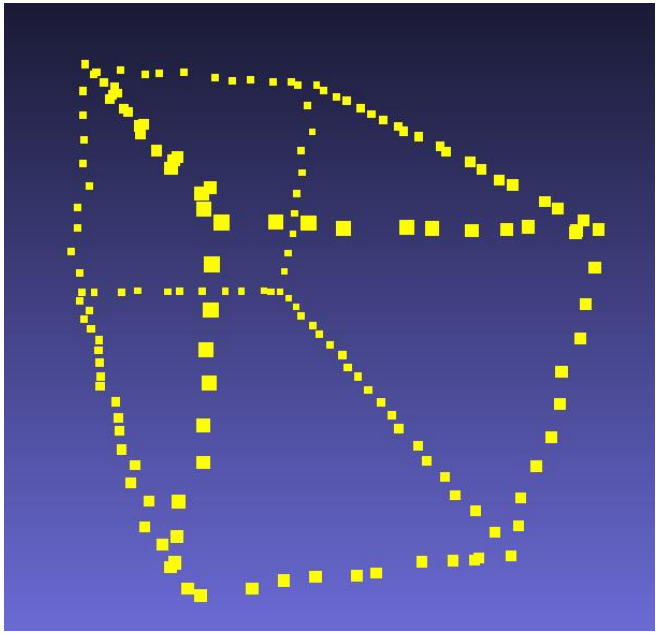

```cpp
#include "opencv2/opencv.hpp"
#include "ceres/ceres.h"
#include "ceres/rotation.h"

// Reprojection error for bundle adjustment
struct ReprojectionError
{
    ReprojectionError(const cv::Point2d& _x, double _f, cv::Point2d& _c) : x(_x), f(_f), c(_c) { }

    template <typename T>
    bool operator()(const T* const camera, const T* const point, T* residuals) const
    {
        // X' = R*X + t
        T X[3];
        ceres::AngleAxisRotatePoint(camera, point, X);
        X[0] += camera[3];
        X[1] += camera[4];
        X[2] += camera[5];

        // x' = K*X'
        T x_p = f * X[0] / X[2] + cx;
        T y_p = f * X[1] / X[2] + cy;

        // residual = x - x'
        residuals[0] = T(x.x) - x_p;
        residuals[1] = T(x.y) - y_p;
        return true;
    }

    static ceres::CostFunction* create(const cv::Point2d& _x, double _f, cv::Point2d& _c)
    {
        return (new ceres::AutoDiffCostFunction<ReprojectionError, 2, 6, 3>(new ReprojectionError(_x, _f, _c)));
    }

private:
    const cv::Point2d x;
    const double f;
    const cv::Point2d c;
};
```

$$x - PX$$

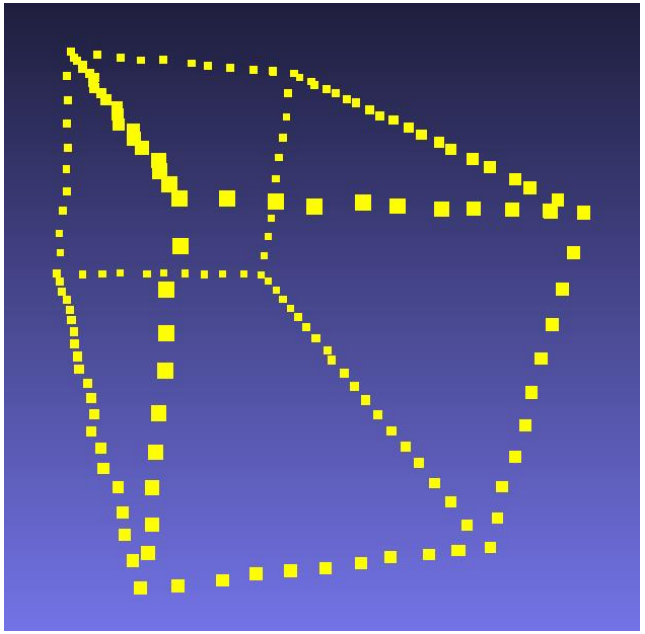
Bundle Adjustment

- Example: **Bundle Adjustment (Incremental)** `[bundle_adjustment_inc.cpp, bundle_adjustment.hpp]`

Result: "triangulation.xyz"



Result: "bundle_adjustment_inc.xyz"

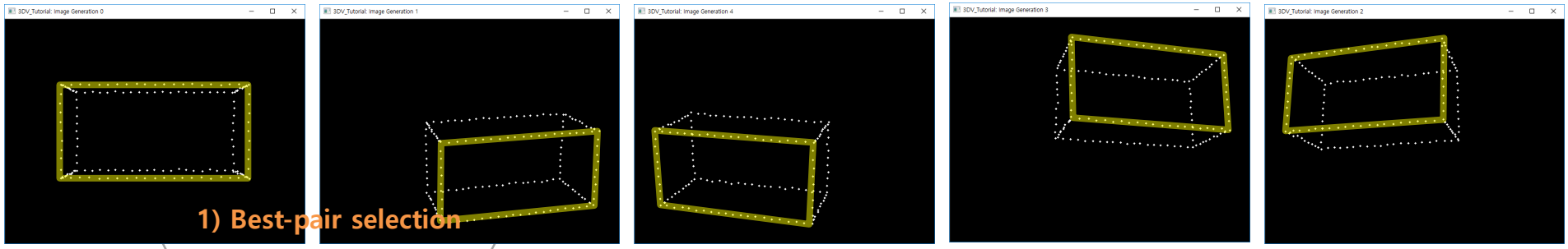


[bundle_adjusetment_global.xyz]
- # of iterations: **20**
- Reprojection error: 0.113 ← **2637.626**

[bundle_adjusetment_inc.xyz]
- # of iterations: **17 + 4 + 6**
- Reprojection error: 0.113 ← **722.350**

[Given]

- **Intrinsic parameters**: camera matrices K_j (all cameras has same and fixed camera matrix.)
- **2D point correspondence** $\mathbf{x}_i^j$ (all points are visible on all camera views.)



1) Best-pair selection

[Unknown]

- **Extrinsic parameters**: camera poses

$\mathrm{R}_0 = \mathrm{I}_{3x3}$
$\mathbf{t}_0 = [0,0,0]^\top$

$\mathrm{R}_1,\ \mathbf{t}_1$

$\mathrm{R}_2,\ \mathbf{t}_2$

$\mathrm{R}_3,\ \mathbf{t}_3$

$\mathrm{R}_4,\ \mathbf{t}_4$

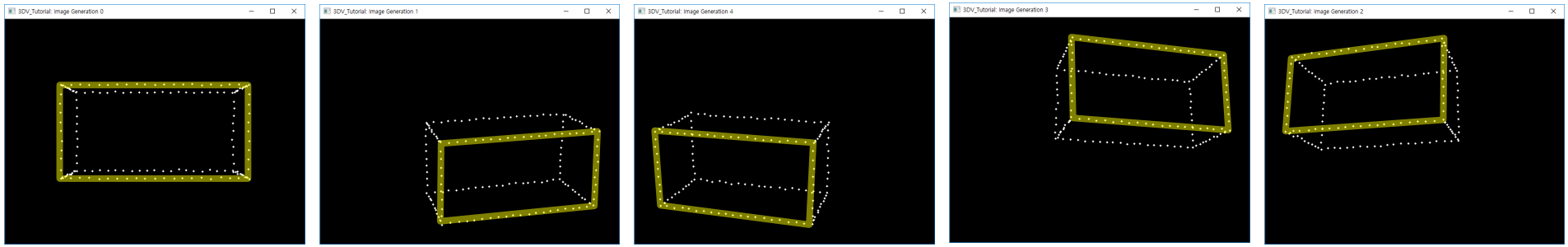
2) Epipolar geometry

3) Triangulation

- **3D points** $\mathbf{X}_i$

[Given]

- **Intrinsic parameters**: camera matrices K_j (all cameras has same and fixed camera matrix.)
- **2D point correspondence** $\mathbf{x}_i^j$ (all points are visible on all camera views.)



4) Next-view selection

[Unknown]

- **Extrinsic parameters**: camera poses

$\mathrm{R}_0 = \mathrm{I}_{3x3}$
$\mathbf{t}_0 = [0,0,0]^\top$

$\mathrm{R}_1,\ \mathbf{t}_1$

$\mathrm{R}_2,\ \mathbf{t}_2$

$\mathrm{R}_3,\ \mathbf{t}_3$

$\mathrm{R}_4,\ \mathbf{t}_4$

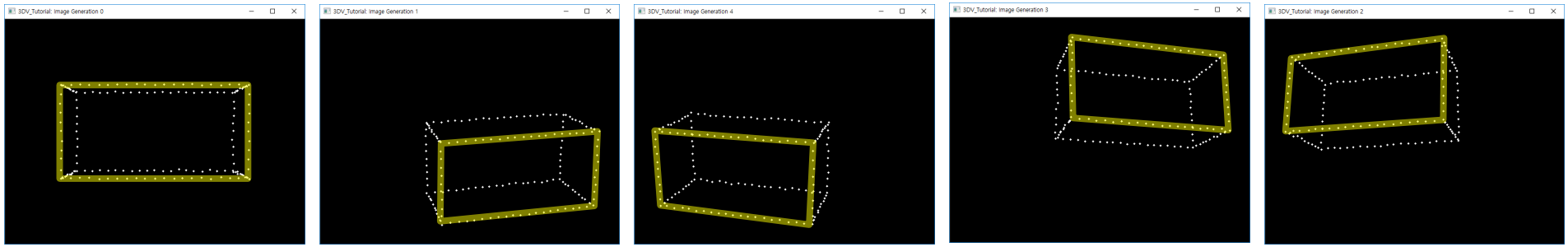
5) Perspective-n-point (PnP)

6) Triangulation

- **3D points** $\mathbf{X}_i$

[Given]

- **Intrinsic parameters**: camera matrices K_j (all cameras has same and fixed camera matrix.)
- **2D point correspondence** $\mathbf{x}_i^j$ (all points are visible on all camera views.)



[Unknown]

- **Extrinsic parameters**: camera poses

$\mathrm{R}_0 = \mathrm{I}_{3x3}$
$\mathbf{t}_0 = [0,0,0]^\top$

$\mathrm{R}_1,\ \mathbf{t}_1$

$\mathrm{R}_2,\ \mathbf{t}_2$

$\mathrm{R}_3,\ \mathbf{t}_3$

$\mathrm{R}_4,\ \mathbf{t}_4$

- **3D points** $\mathbf{X}_i$

4) Next-view selection

Repeat 4 – 7 until the last image

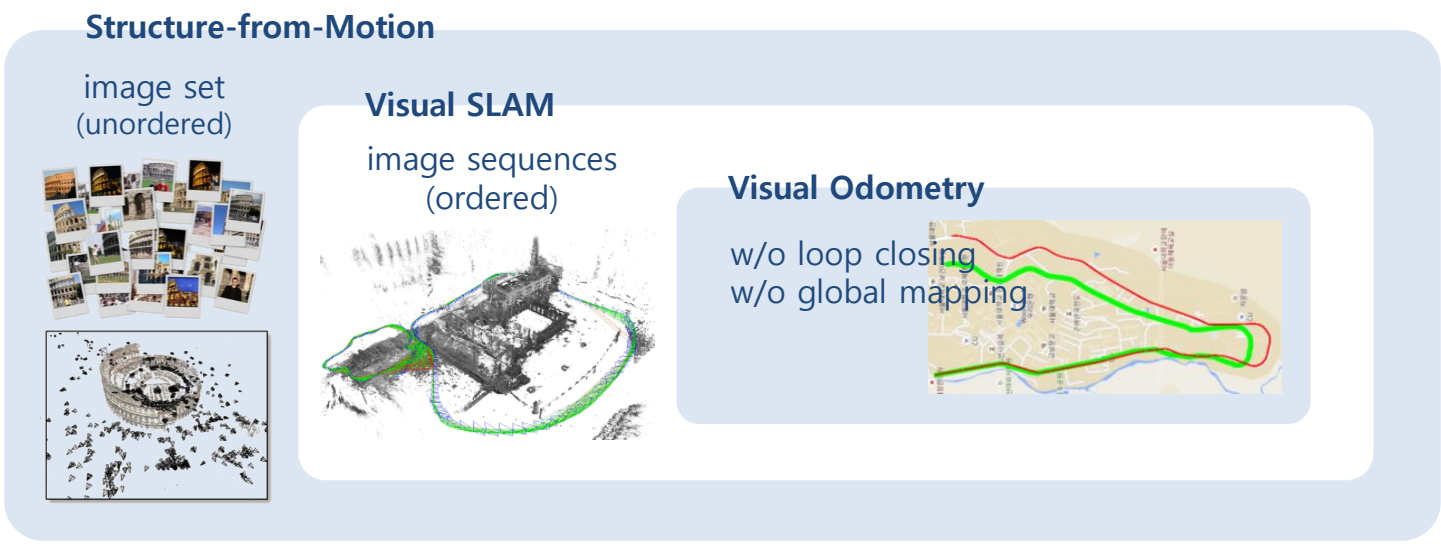
5) Perspective-n-point (PnP)

6) Triangulation

7) Bundle Adjustment

Applications

- **Structure-from-Motion (SfM)** → 3D Reconstruction, Photo Browsing
  - Bundler, COLMAP, MVE, Theia, openMVG, OpenSfM, Toy SfM / VisualSFM (GUI, binary only)
- **Visual SLAM** → Augmented Reality, Navigation (Mapping and Localization)
  - PTAM (Parallel Tracking and Mapping), DTAM (Dense Tracking and Mapping), ORB-SLAM2, LSD-SLAM
  - cf. Visual loop closure (a.k.a. visual place recognition): DBoW2, FBoW, PoseNet, NetVLAD
- **Visual Odometry** → Navigation (Localization)
  - LIBVISO2 (C++ Library for Visual Odometry 2), SVO (Semi-direct Monocular Visual Odometry), DVO (Direct Sparse Odometry), DeepVO, UnDeepVO



Structure-from-Motion

- Example: **Structure-from-Motion (Global)** `[sfm_global.cpp]`

0) Load images and extract features

0) Match features and find good matches (which has enough inliers)

Structure-from-Motion

- Example: **Structure-from-Motion (Global)** `[sfm_global.cpp]`

0) Load images and extract features

0) Match features and find good matches (which has enough inliers)

1) Initialize cameras (R, **t**, f, ...)

2) Initialize 3D points and build a visibility graph

3) Optimize camera pose and 3D points together (BA)

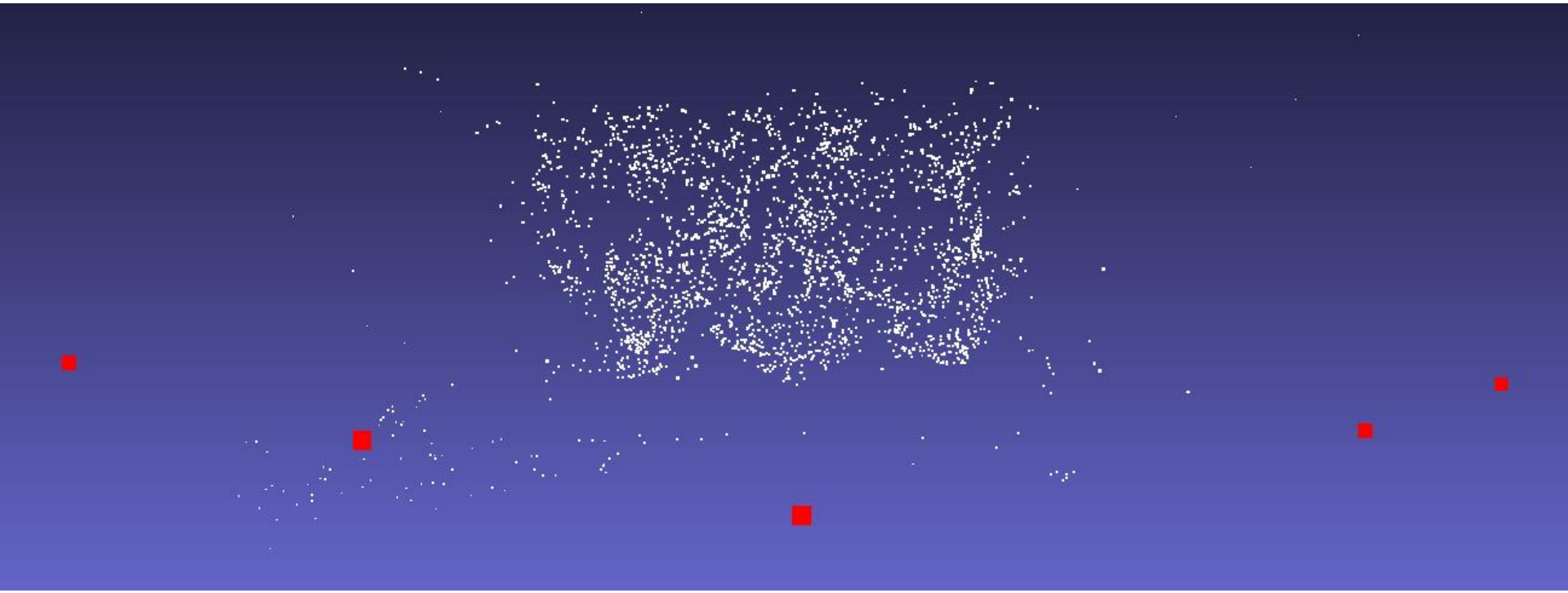
$$\mathrm{K}_i = \begin{bmatrix} f & 0 & w/2 \\ 0 & f & h/2 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathrm{R}_i = \mathrm{I}_{3x3}, \quad \text{and} \quad \mathbf{t}_i = [0,0,0]^\top$$

$$\mathbf{X}_i = [0,0,2]^\top$$

Ceres Solver (non-linear least-square optimization)

Structure-from-Motion

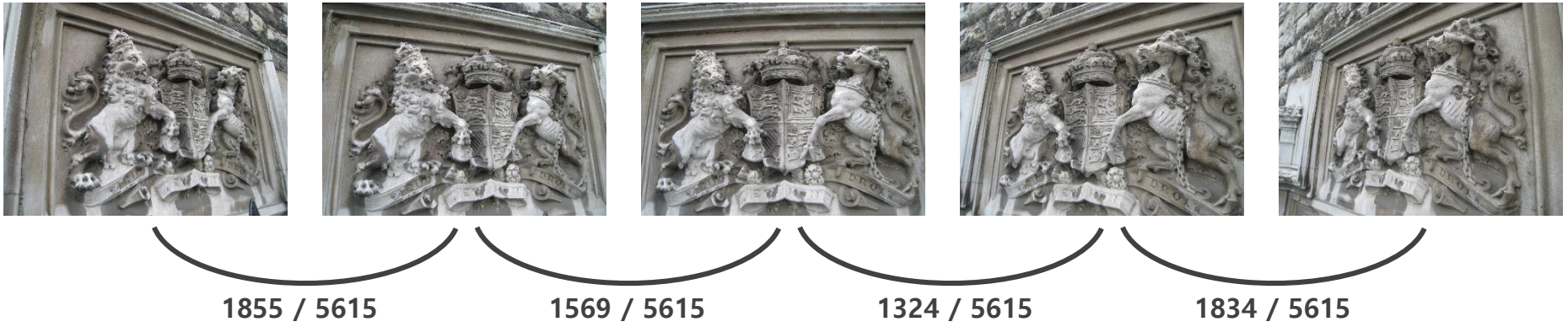
- Example: **Structure-from-Motion (Global)** `[sfm_global.cpp]`



Structure-from-Motion

- Example: **Structure-from-Motion (Incremental)** `[sfm_inc.cpp]`

0) Build a viewing graph (well-matched pairs) while finding inliers



1) Select the best pair

2) Estimate relative pose from the best two views (epipolar geometry)

3) Reconstruct 3D points of the best two views (triangulation)

4) Select the next image to add
Separate points into known and unknown for PnP (known) and triangulation (unknown)

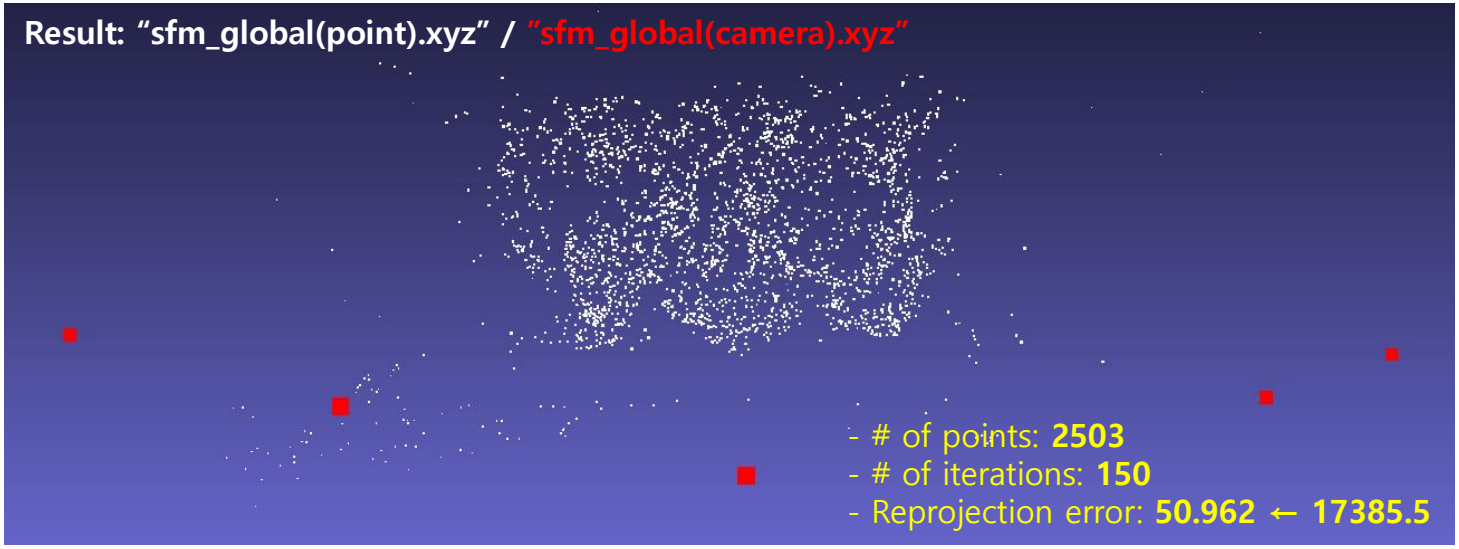
5) Estimate relative pose of the next view (PnP)

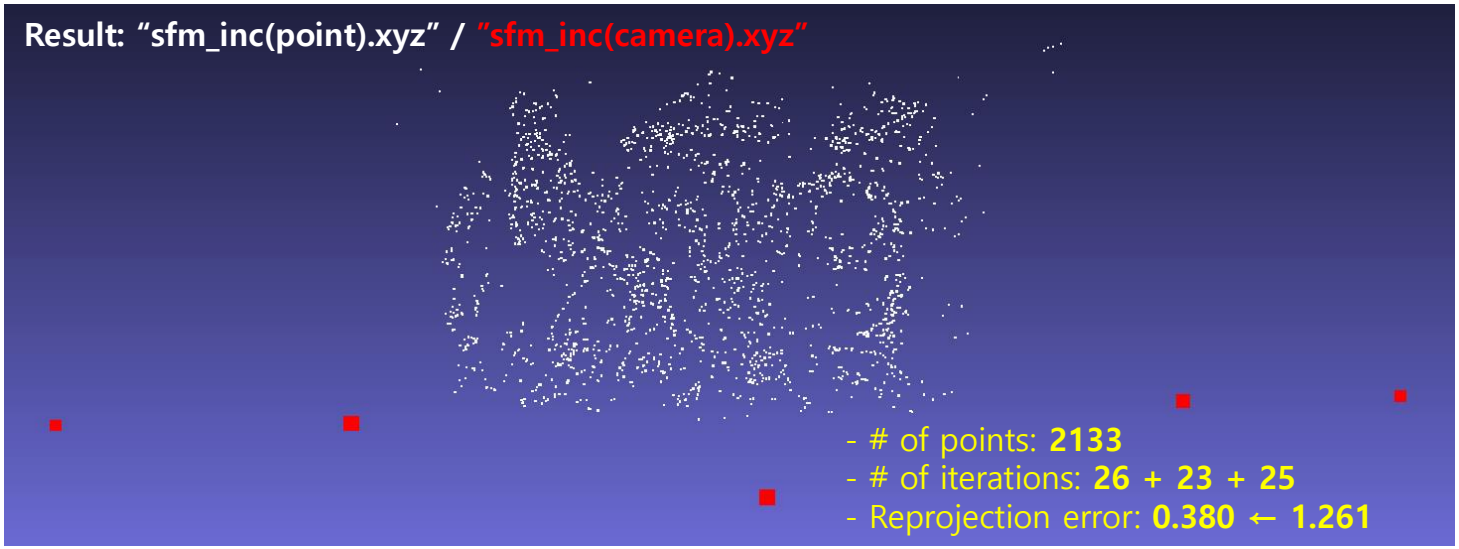
6) Reconstruct newly observed 3D points (triangulation)

7) Optimize camera pose and 3D points together (BA)

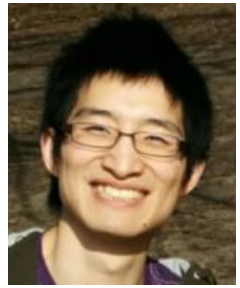
Incrementally add more views

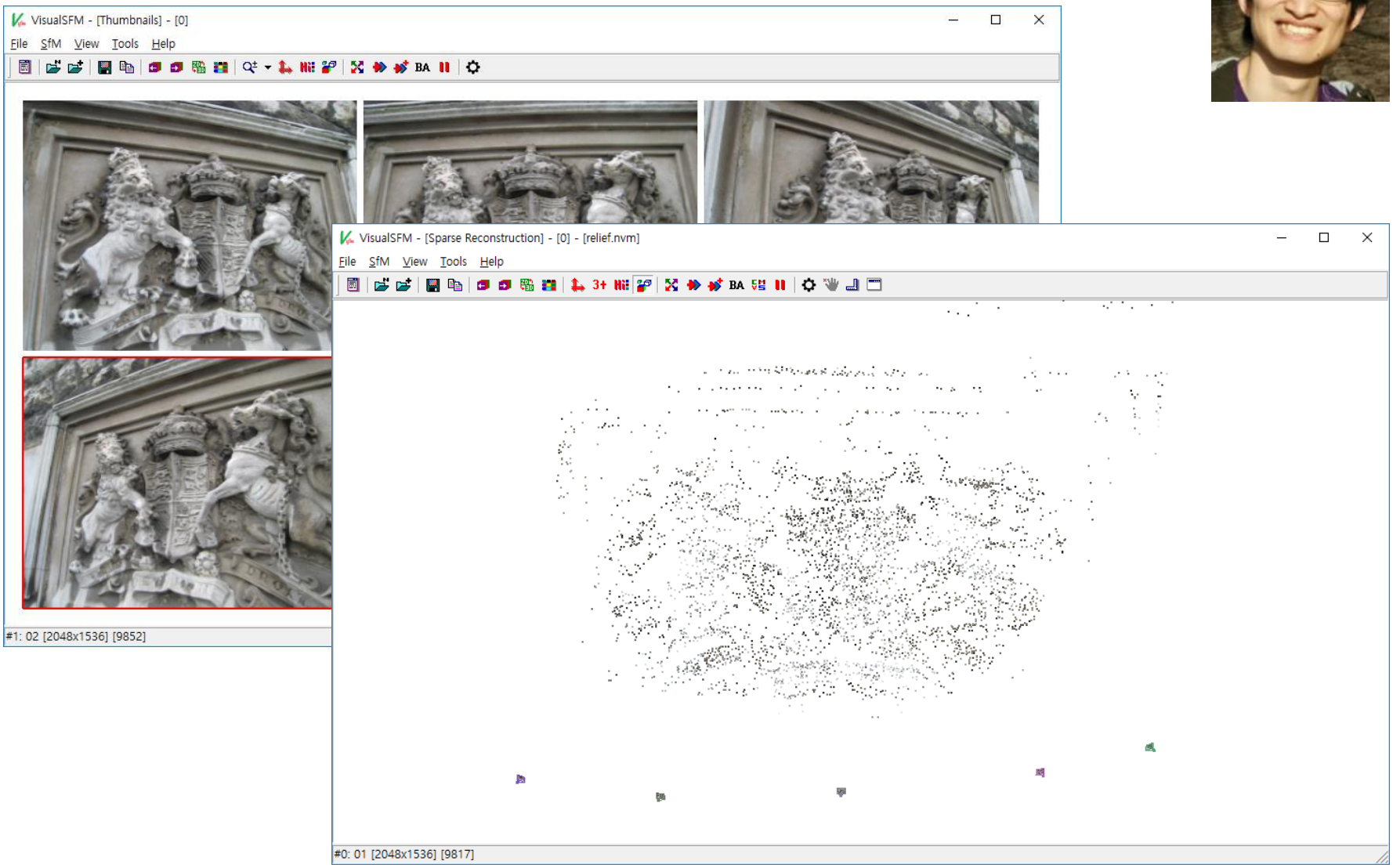
Structure-from-Motion





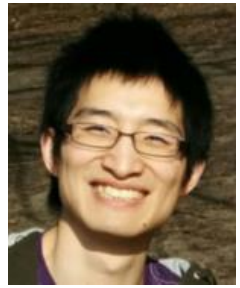
Structure-from-Motion using VisualSFM

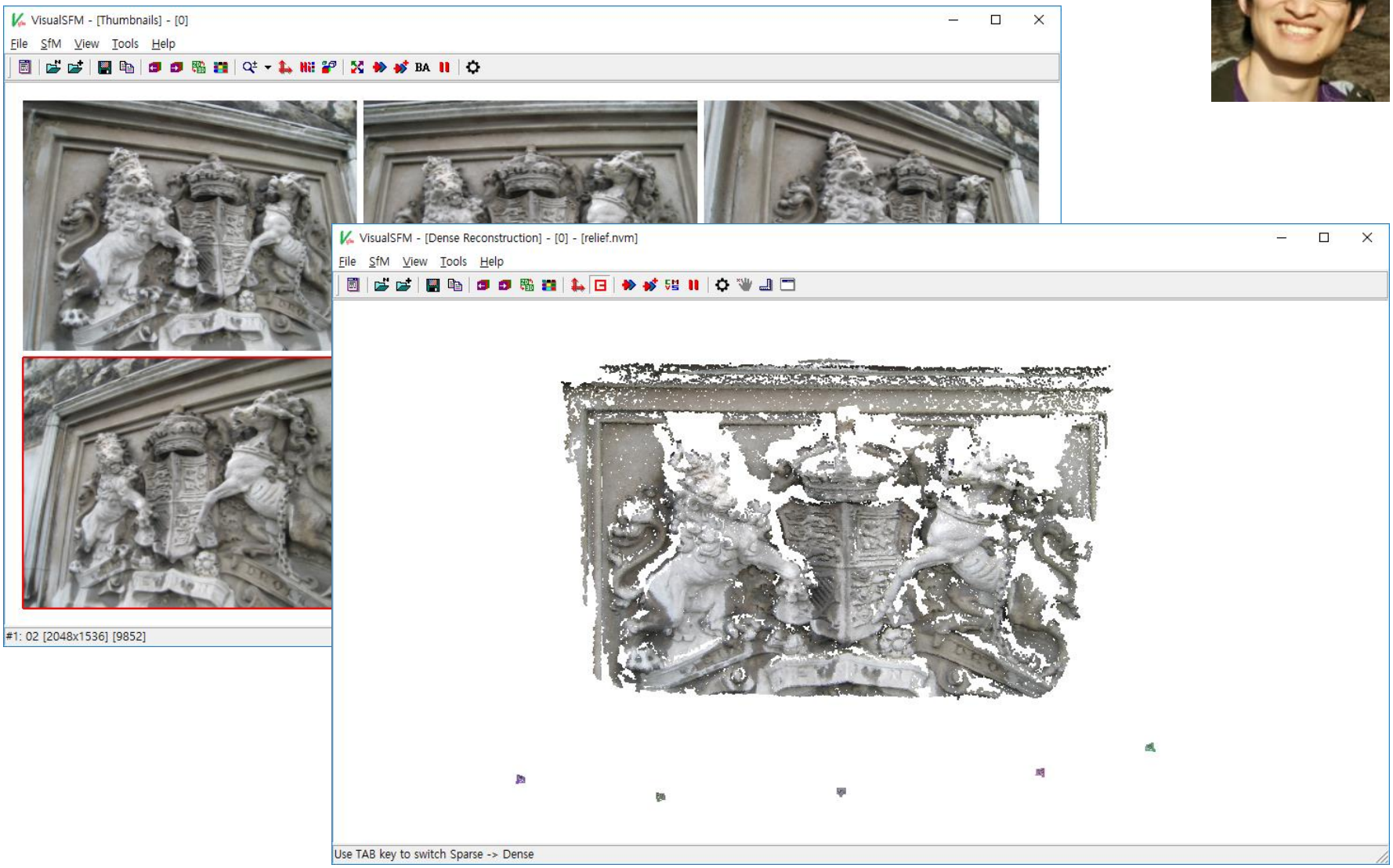




Sparse 3D Reconsturction by VisualSFM (SiftGPU and pba)

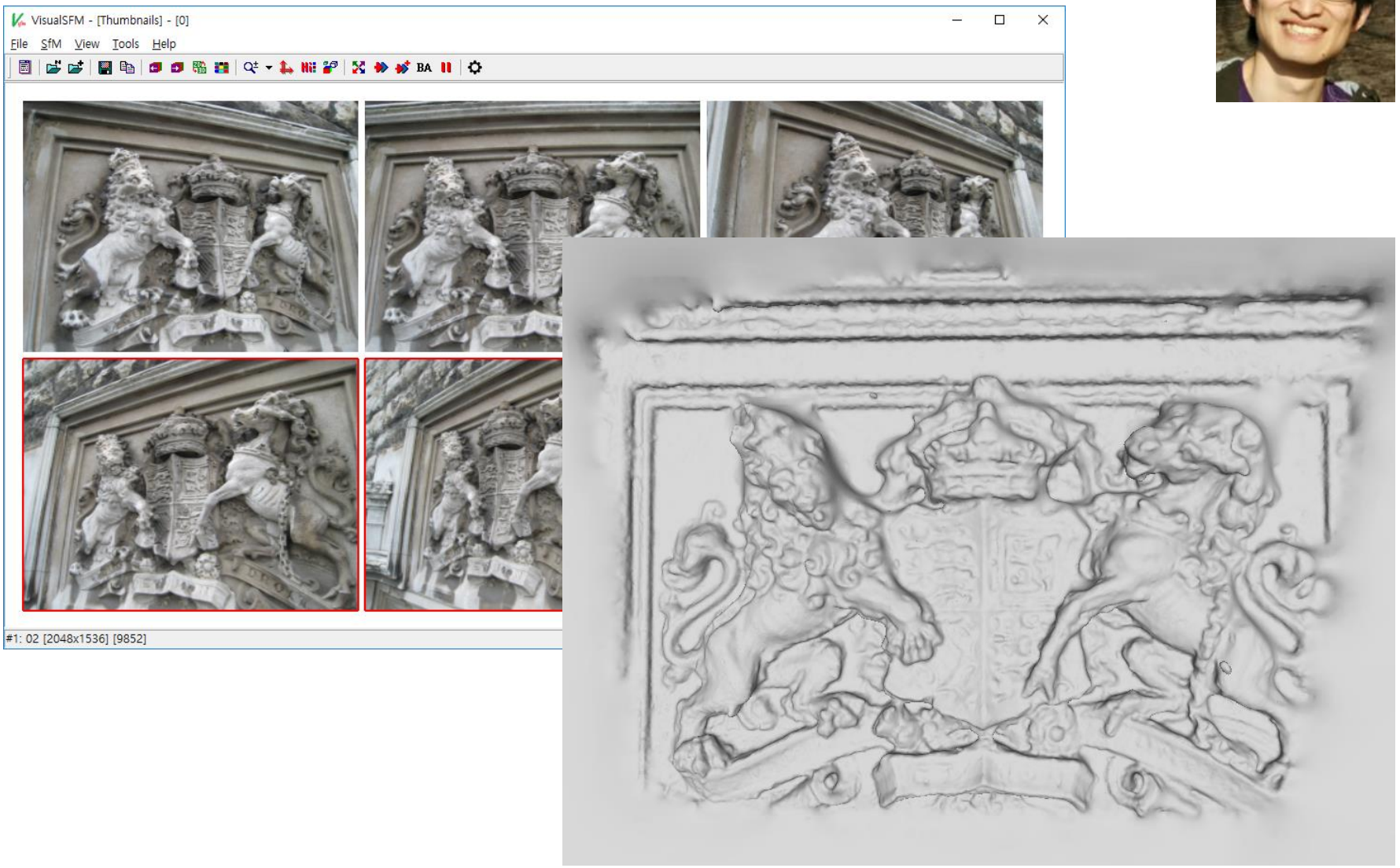
Structure-from-Motion using VisualSFM





Dense 3D Reconsturction by VisualSFM (CMVS and PMVS2)

Structure-from-Motion using VisualSFM

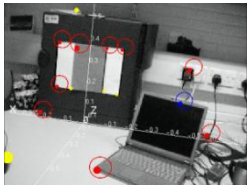


Surface Reconstruction by VisualSFM (Poisson Surface Reconstruction) [URL]

Visual Odometry and SLAM: History

Monocular Camera

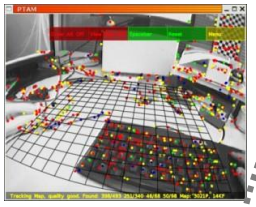
SLAM in Robotics



MonoSLAM (2003): features, Bayesian filtering

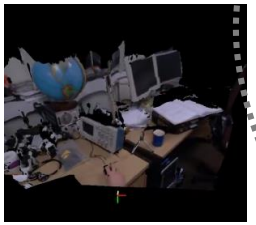
v.s.

SfM in Computer Vision



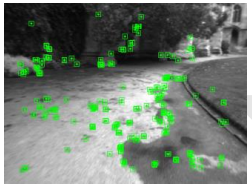
PTAM (2007): more features, bundle adjustment (keyframes)

RGB-D Camera



DTAM (2011): dense volumetric

KineticFusion (2011): dense volumetric



ORB-SLAM (2014)
better feature management

v.s.

LSD-SLAM (2014)
direct, semi-dense

ElasticFusion (2015)
dense surfels

DynamicFusion (2015)
dynamic dense

Paradigm #1: Bayesian Filtering v.s. Bundle Adjustment

Robotics

Localization + Mapping

SLAM

(Simultaneous Localization and Mapping)

Bayesian Filtering

e.g. **MonoSLAM**

[Davison et al., T-PAMI, 2007]

VS

Computer Vision

Motion Estimation + Reconstruction

SfM

(Structure from Motion)

Bundle Adjustment

e.g. **PTAM**

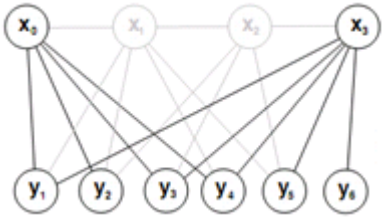
[Klein and Murray, ISMAR, 2007]

"Real-time Monocular SLAM: Why Filter?"

[Strasdat et al., ICRA, 2010]

≤

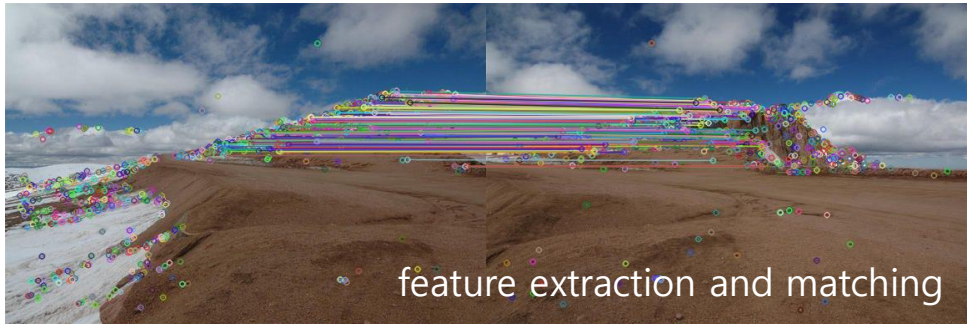
1. Global optimization
2. # of features (100 vs 4000)



Paradigm #2: Feature-based Method v.s. Direct Method

- e.g. Image stitching

Feature-based Method VS **Direct Method**



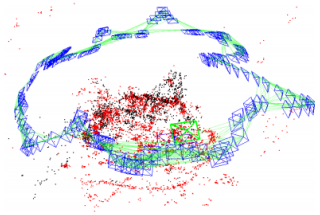
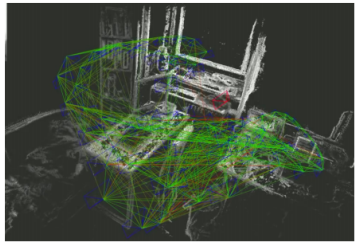
$$\arg\min_{\mathrm{H}} \sum_i \left\| \mathrm{H}\mathbf{x}_i - \mathbf{x}'_i \right\|^2$$

$$\arg\min_{\mathrm{H}} \sum_{u,v} \left\| I(\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}) - I'(\mathrm{H}\begin{bmatrix} u \\ v \\ 1 \end{bmatrix}) \right\|^2$$

Paradigm #2: Feature-based Method v.s. Direct Method

Feature-based/Direct SLAM

Universidad Zaragoza

Feature-Based SLAM	Direct SLAM
	
Minimize **Feature Reprojection Error**	Minimize **Photometric Error**
Sparse Reconstruction	**Semi Dense / Dense** Reconstruction
PTAM, ORB-SLAM	**DTAM, LSD-SLAM, DPPTAM**

ROBOTICS UNIVERSIDAD DE ZARAGOZA

Why should we still use features?

Universidad Zaragoza

Robustness

- **Reliable two-view monocular initialization**
- **Good invariance to viewpoint and illumination**
- **Less affected by auto-gain and auto-exposure**
- **Less affected by dynamic elements**

Accuracy

- **Bundle adjustment (joint map-trajectory optimization)**

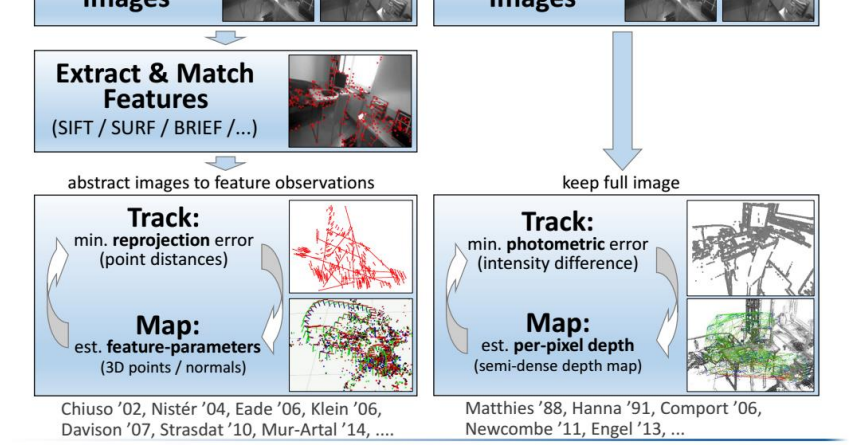
Place Recogniton (loop detection, relocalization)

- **Bags of Words**

But sparse reconstructions ...

ROBOTICS UNIVERSIDAD DE ZARAGOZA

Comparison



Comparison

Feature-Based	Direct
can only use & reconstruct corners	can use & reconstruct whole image
faster	slower (but good for parallelism)
flexible: outliers can be removed retroactively.	inflexible: difficult to remove outliers retroactively.
robust to inconsistencies in the model/system (rolling shutter).	not robust to inconsistencies in the model/system (rolling shutter).
decisions (KP detection) based on less complete information.	decision (linearization point) based on more complete information.
no need for good initialization.	needs good initialization.
~20+ years of intensive research	~4 years of research (+5years 25 years ago)

Why Visual Odometry?

Visual odometry

VS

Wheel odometry

+ direct motion measure
+ six degree-of-freedoms
+ easy to install
– heavy computation
– visual disturbance (e.g. moving objects)

– indirect motion measure (e.g. slippage)
– two degree-of-freedoms
– necessary to be on rotor/shaft
+ simple calculation

Why Visual Odometry?

Visual odometry

vs

Wheel odometry

- indirect motion measure (e.g. slippage)
- two degree-of-freedoms
- necessary to be on rotor/shaft

\+ simple calculation

no wheels / rough terrains, in-the-sky, under-the-water

Why Visual Odometry?

Visual odometry **VS**

Wheel odometry

– indirect motion measure (e.g. slippage)
– two degree-of-freedoms
– necessary to be on rotor/shaft
+ simple calculation

no wheels / rough terrains, in-the-sky, under-the-water

Visual Odometry **VS**

Visual SLAM

no assumption on trajectories
→ navigation / large space (outdoor)

closed-loop is preferred for convergence
→ mapping / small space (indoor, campus)

Why Visual Odometry?

Visual odometry **VS**

Wheel odometry

– indirect motion measure (e.g. slippage)
– two degree-of-freedoms
– necessary to be on rotor/shaft
+ simple calculation

no wheels / rough terrains, in-the-sky, under-the-water

Visual Odometry **VS**

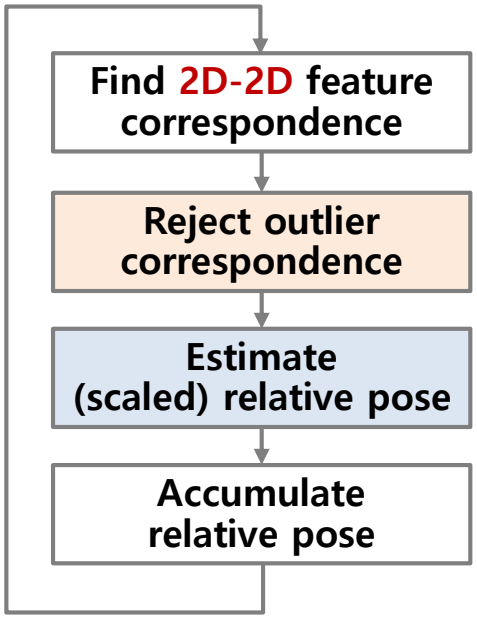
Visual SLAM

closed-loop is preferred for convergence
→ mapping / small space (indoor, campus)

real navigation situations

Feature-based Monocular Visual Odometry

Two-view Motion Estimation

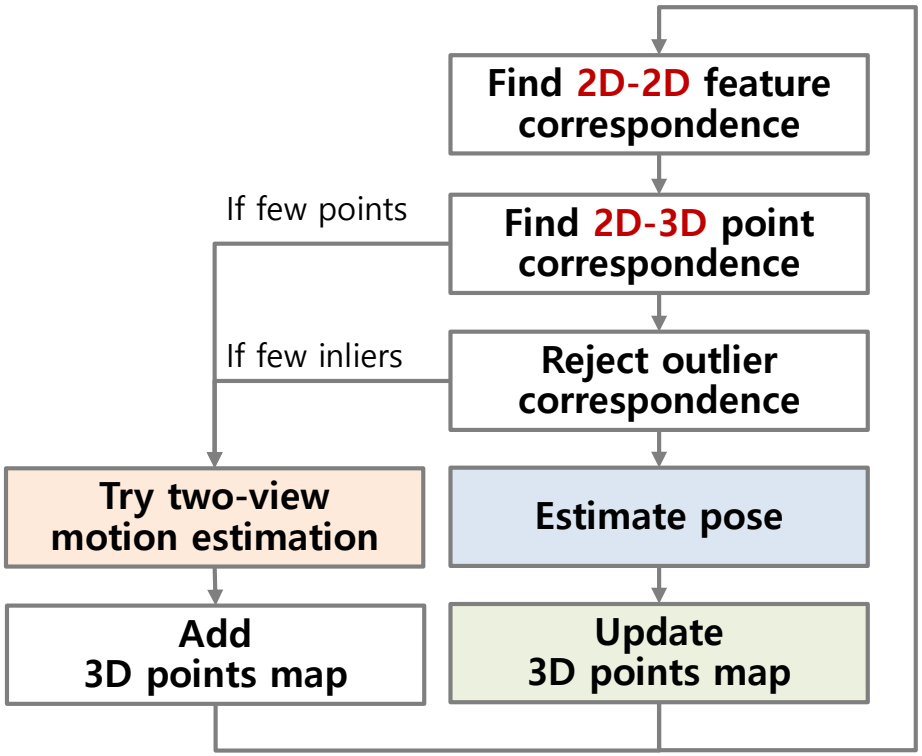


- Feature: **Good-Feature-to-Track** [Shi94_CVPR]
  with bucketing to distribute features
- Correspondence: **Lucas-Kanade optical flow** [Lucas81_IJCAI]

- Adaptive MSAC [Choi09_IROS]
- **Iterative 5-point algorithm** [Choi15_IJCAS]
- Error measure: Sampson distance

- Normalized 8-point algorithm
- Scale-from-ground with **asymmetric kernels** [Choi13_URAI]

Feature-based Monocular Visual Odometry



- Feature: **Good-Feature-to-Track** [Shi94_CVPR]
  with bucketing to distribute features
- Correspondence: **Lucas-Kanade optical flow**
  [Lucas81_IJCAI]

- Adaptive MSAC
- **Iterative PnP algorithm** (3-point algorithm)
- Error measure: Projection error

- Iterative PnP algorithm
- Scale-from-ground with **asymmetric kernels**
  [Choi13_URAI]

- **Bundle adjustment** over last K keyframes
  Reprojection error with Cauchy loss function

Feature-based Monocular Visual Odometry

Summary

Slides and example codes are available:
https://github.com/sunglok/3dv_tutorial

- **What is 3D Vision?**
- **Single-view Geometry**
  - **Camera Projection Model**
    - Pinhole Camera Model
    - Geometric Distortion Models
  - **General 2D-3D Geometry**
    - Camera Calibration
    - Absolute Camera Pose Estimation (PnP Problem)
- **Two-view Geometry**
  - **Planar 2D-2D Geometry (Projective Geometry)**
    - Planar Homography
  - **General 2D-2D Geometry (Epipolar Geometry)**
    - Fundamental/Essential Matrix
    - Relative Camera Pose Estimation
    - Triangulation (Point Localization)
- **Multi-view Geometry**
  - Bundle Adjustment (Non-linear Optimization)
  - Applications: Structure-from-motion, Visual SLAM, and Visual Odometry
- **Correspondence Problem**
  - Feature Correspondence: Feature Matching and Tracking
  - Robust Parameter Estimation: (Hough Transform), RANSAC, M-estimator

$$\therefore \mathbf{x} = \mathrm{P}\mathbf{X} \quad \left(\mathrm{P} = \mathrm{K}\left[\mathrm{R} \,|\, \mathbf{t}\right]\right)$$
$$\mathbf{x} = \mathrm{K}\hat{\mathbf{x}}$$

$$\therefore \mathbf{x}' = \mathrm{H}\mathbf{x}$$
$$\hat{\mathbf{x}}' = \hat{\mathrm{H}}\hat{\mathbf{x}} \quad \left(\hat{\mathrm{H}} = \mathrm{R} + \frac{1}{d}\mathbf{t}\mathbf{n}^\top\right)$$

$$\therefore \mathbf{x}'^\top \mathrm{F}\mathbf{x} = 0$$
$$\hat{\mathbf{x}}'^\top \mathrm{E}\hat{\mathbf{x}} = 0 \quad \left(\mathrm{E} = [\mathbf{t}]_\times \mathrm{R}\right)$$

$$\therefore \operatorname{argmin} \sum_i^n \sum_j^m \left\| \mathbf{x}_i^j - \mathrm{P}_j \mathbf{X}_i \right\|_\Sigma^2$$

Applications in Deep Learning Era

- **There are still many researches and applications.**
  - 3D reconstruction
  - Real-time visual odometry/SLAM
  - Augmented reality (mixed reality), virtual reality

model-based problem solving
(e.g. calculating camera pose; minimizing a cost function)

unknown models and procedures
(e.g. recognizing objects, finding correspondence)

- 3D understanding of **results** from deep learning
  - e.g. Bounding boxes from object recognition → metric position and size

- Designing **cost functions** for deep learning
  - e.g. Left-right consistency in MonoDepth [CVPR 2017]
  - e.g. Eigendecomposition-free training of deep learning [ECCV 2018]

- Building **datasets** for deep learning
  - e.g. LIFT [ECCV 2016]

Appendix: How to Solve Problems

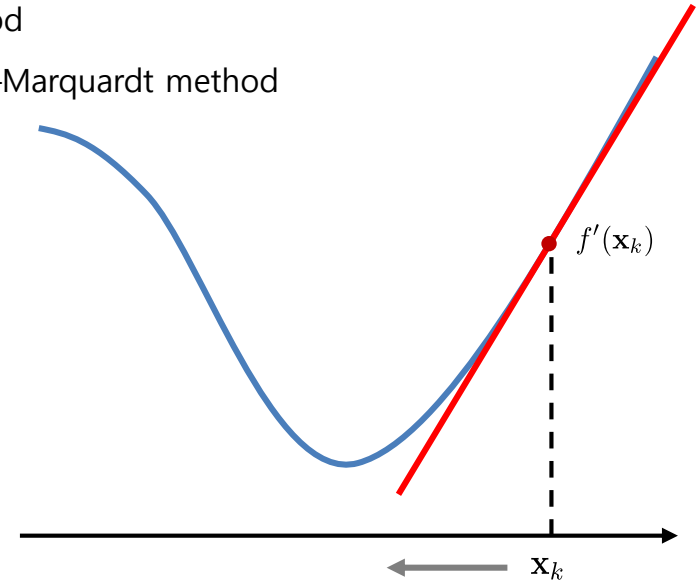
- **Linear Equations**
  - Inhomogeneous cases: Multiplying a pseudo-inverse

$$\mathrm{A}\mathbf{x} = \mathbf{b} \longrightarrow \mathbf{x} = \mathrm{A}^{\dagger}\mathbf{b}$$

  - Homogenous cases: Finding a null vector (or a vector which has the smallest singular value)

$$\mathrm{A}\mathbf{x} = 0 \longrightarrow \mathbf{x} = \mathrm{V}\,[0, 0, \cdots, 1]^{\top} \quad \text{where} \quad \mathrm{A} = \mathrm{U}\mathrm{D}\mathrm{V}^{\top}$$

- **Non-linear Equations**
  - Non-linear optimization $\hat{\mathbf{x}} = \underset{\mathbf{x}}{\operatorname{argmin}} f(\mathbf{x})$
  - General cases: Gradient-descent method, Newton method
  - Least squares cases: Gauss-Newton method, Levenberg–Marquardt method



Appendix: Further Information

- **Beyond Point Features**
  - Other features: OPVO, Kimera
  - Direct methods (w/o features): Already mentioned (including deep learning)
- ~~**Real-time / Large-scale SfM**~~
- **(Spatially / Temporally) Non-static SfM**
  - Deformable (or moving) objects: Non-rigid SfM
- **Depth, Object, and Semantic Recognition**

- **Sensor Fusion and New Sensors**
  - + Depth (ORB-SLAM2, LSD-SLAM, DVO have their variants with depth.)
    - RGB-D: ElasticFusion, RGB-D SLAM, BaMVO, RTAB-Map
    - Stereo cameras: S-PTAM, ProSLAM
    - LiDAR: LIMO / cf. Cartographer
  - + IMU: OKVIS, ROVIO, VINS-Mono
    - Visual-inertia Calibration: Kalib
  - + GPS
  - Omni-directional cameras: Multi-FoV datasets
  - Light-field cameras
  - Event camera: ETAM*
- **Minimal Solvers / Self-calibration**
  - OpenGV, Minimal Problems in Computer Vision
- **Public Datasets and Evaluations**
  - Datasets: Awesome Robotics Datasets
  - Evaluations: KITTI Odometry/SLAM Evaluation 2012, GSLAM, evo