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

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)

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)

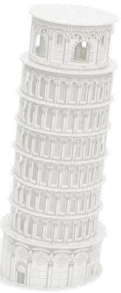
What is 3D Vision?

Computer Vision

What is it?

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

Where is it?

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

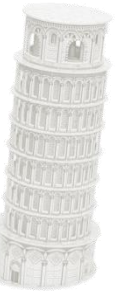
What is 3D Vision?

Computer Vision

What is it?

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

Where is it?

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

(84, 10, 18) [m]

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

What is 3D Vision?

Visual Geometry (Multiple View Geometry)

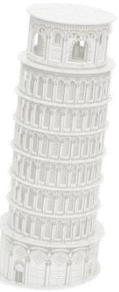
Geometric Vision

Computer Vision

What is it?

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

Where is it?

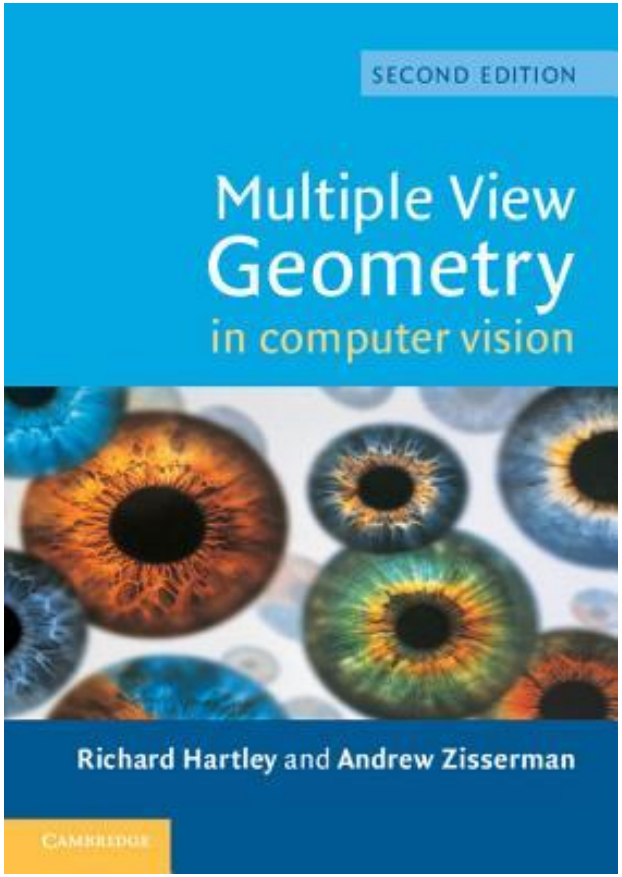
- Place (e.g. Piazza del Duomo, Pisa, Italy)
- Location (e.g. )

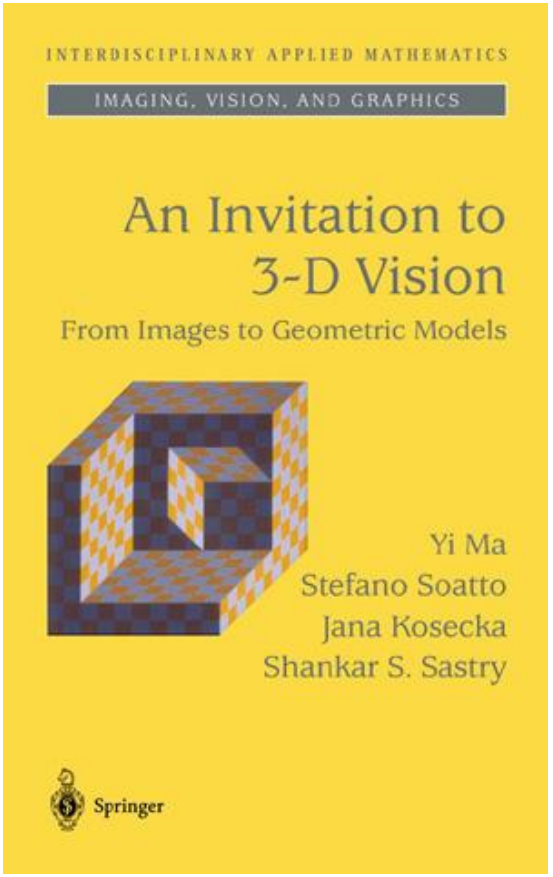
(84, 10, 18) [m]

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

What is 3D Vision?

- Reference Books





What is 3D Vision?

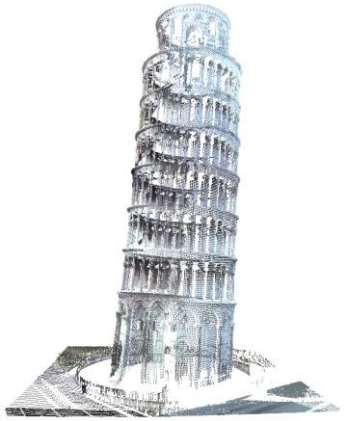
- OpenCV (Open Source Computer Vision)
  - `calib3d` Module: Camera Calibration and 3D Reconstruction

- 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

c.f. All examples are **less than 100 lines** and based on **recent OpenCV (> 3.0.0)**.

What is 3D Vision?

image



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

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

Perspective Camera

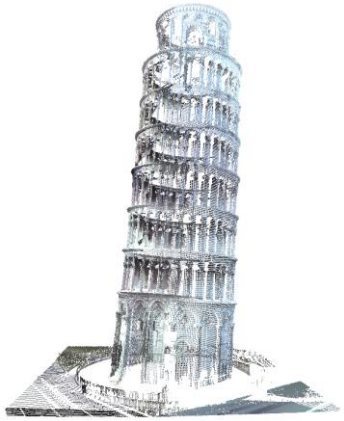
Omni-directional Camera

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

Range Sensor
(LiDAR, RADAR)

What is 3D Vision?

image



point cloud, depth/range image, polygon mesh, ...

Perspective Camera

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

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

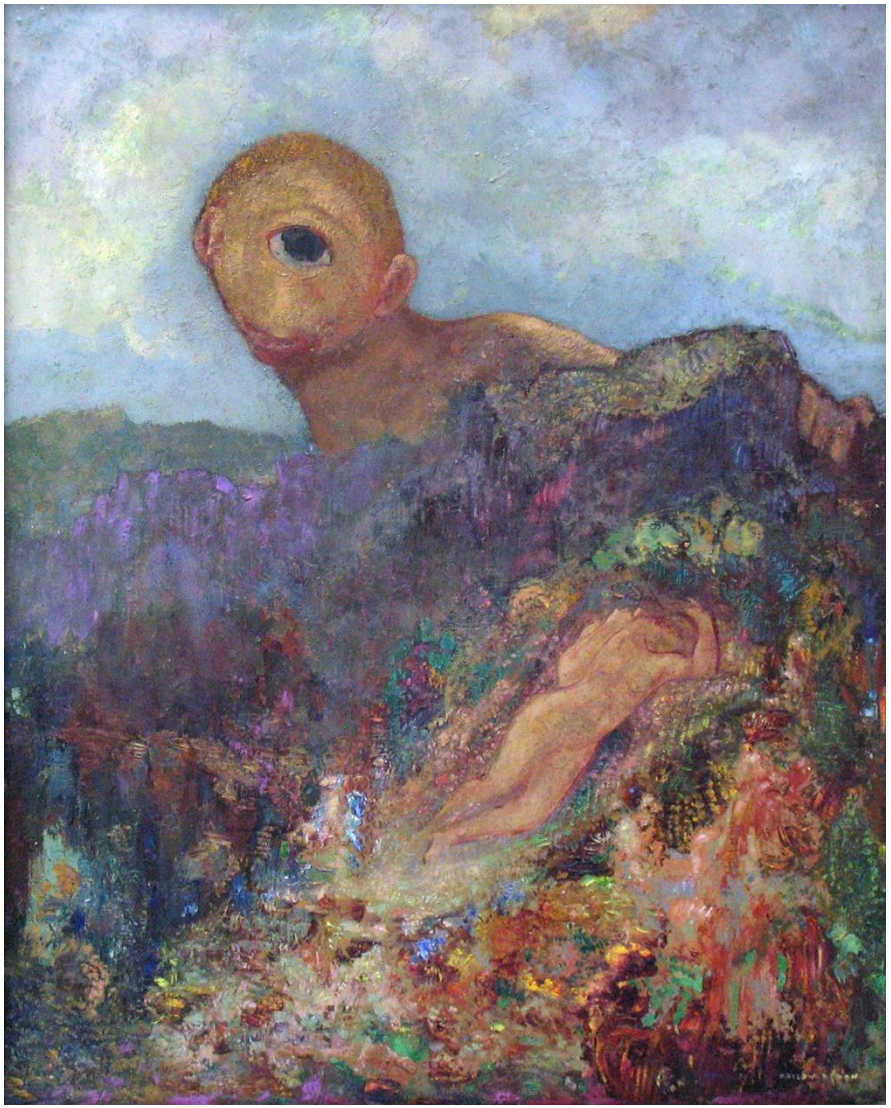
Omni-directional Camera

Range Sensor
(LiDAR, RADAR)

Table of Contents

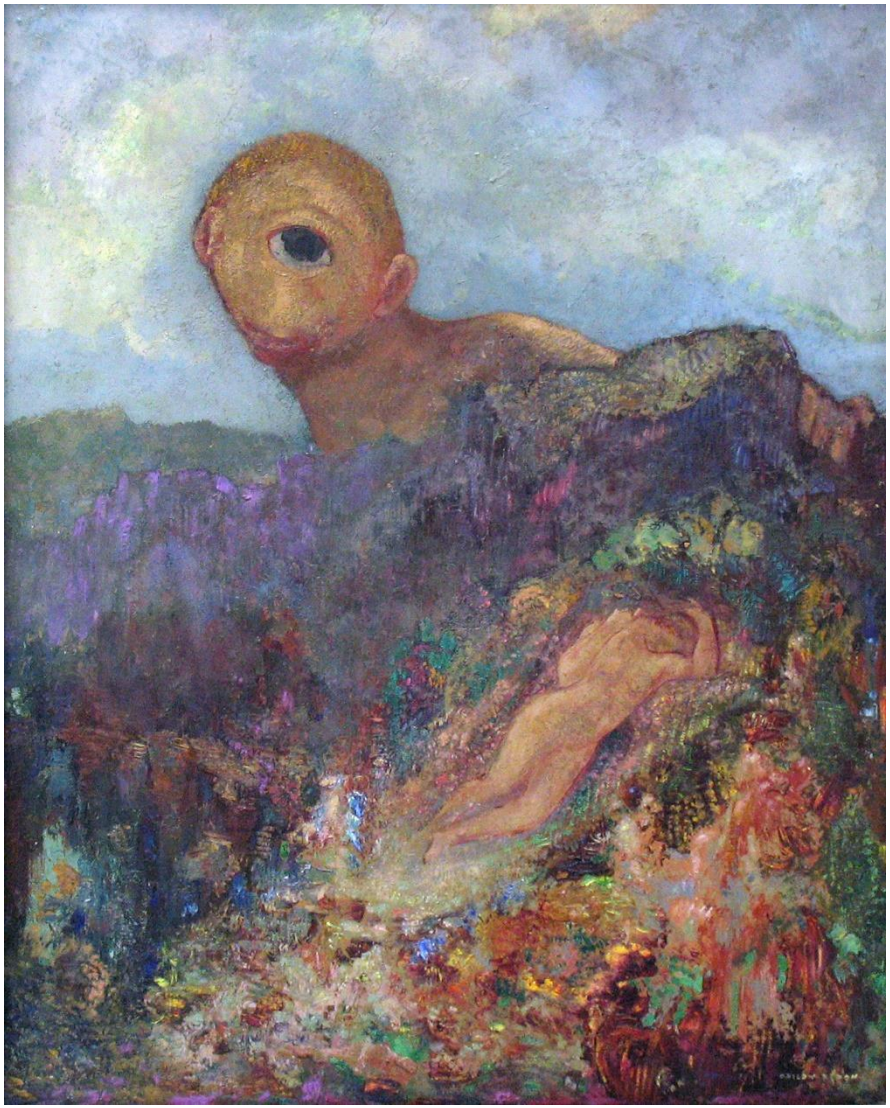
- **What is 3D Vision?**
- **Single-view Geometry**
  - Camera Projection Model
  - General 2D-3D Geometry
- **Two-view Geometry**
  - Planar 2D-2D Geometry (**Projective Geometry**)
  - General 2D-2D Geometry (**Epipolar Geometry**)
- **Multi-view Geometry**
- **Correspondence Problem**
- **Summary**

Single-view Geometry



The Cyclops, gouache and oil by Odilon Redon

Single-view Geometry



Camera Projection Model

General 2D-3D Geometry

The Cyclops, gouache and oil by Odilon Redon

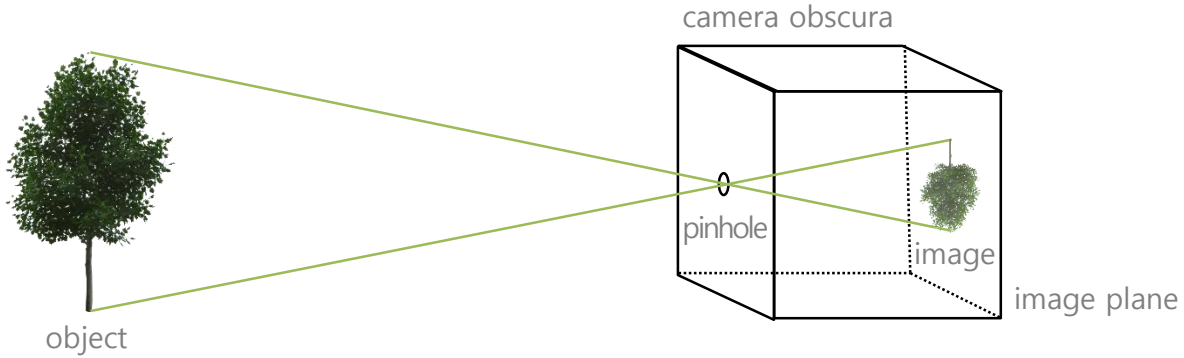
Camera Projection Model

- **The Pinhole Camera Model**

A large-scale camera obscura at San Francisco, California

A modern-day camera obscura

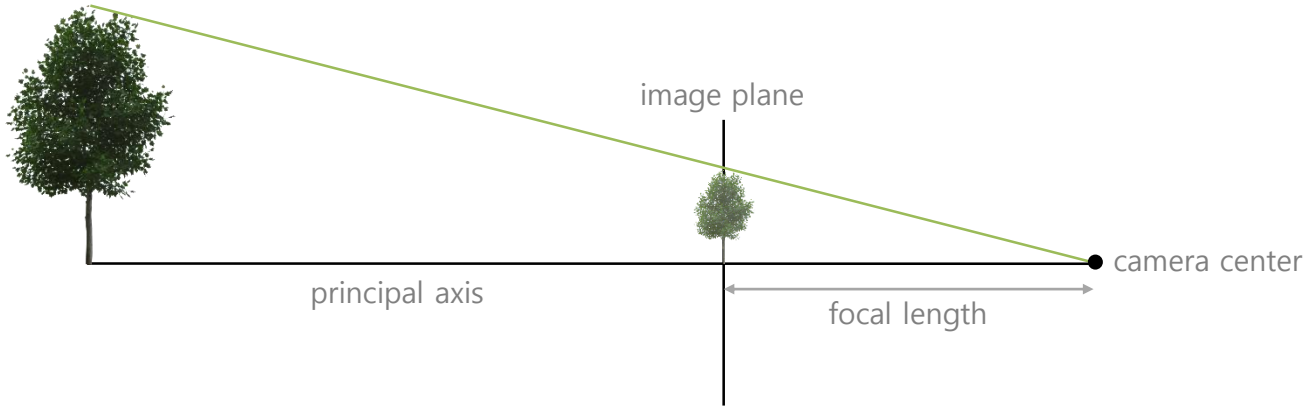
An Image in camera obscura at Portslade, England

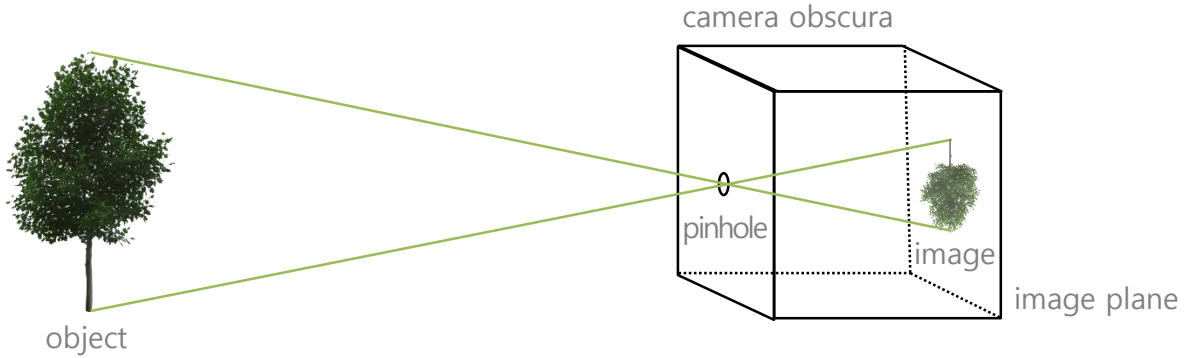


Images are from Wikipedia (https://en.wikipedia.org/wiki/Camera_obscura).

Camera Projection Model

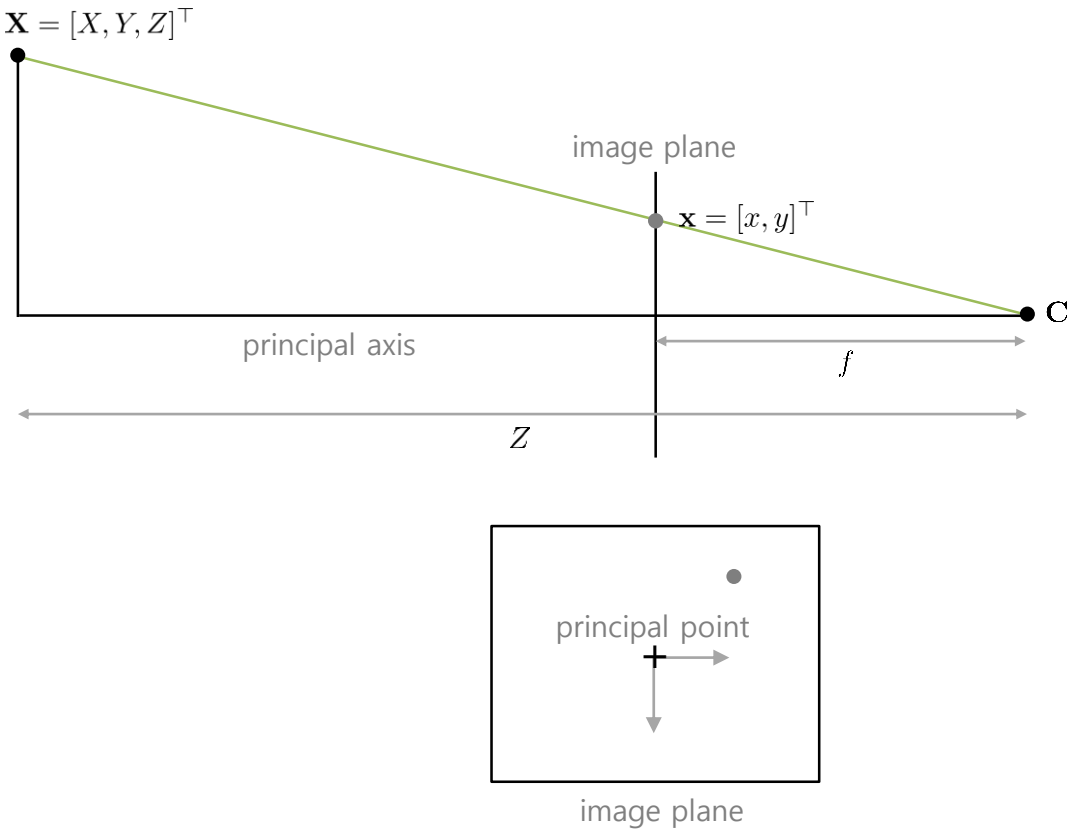
- **The Pinhole Camera Model**





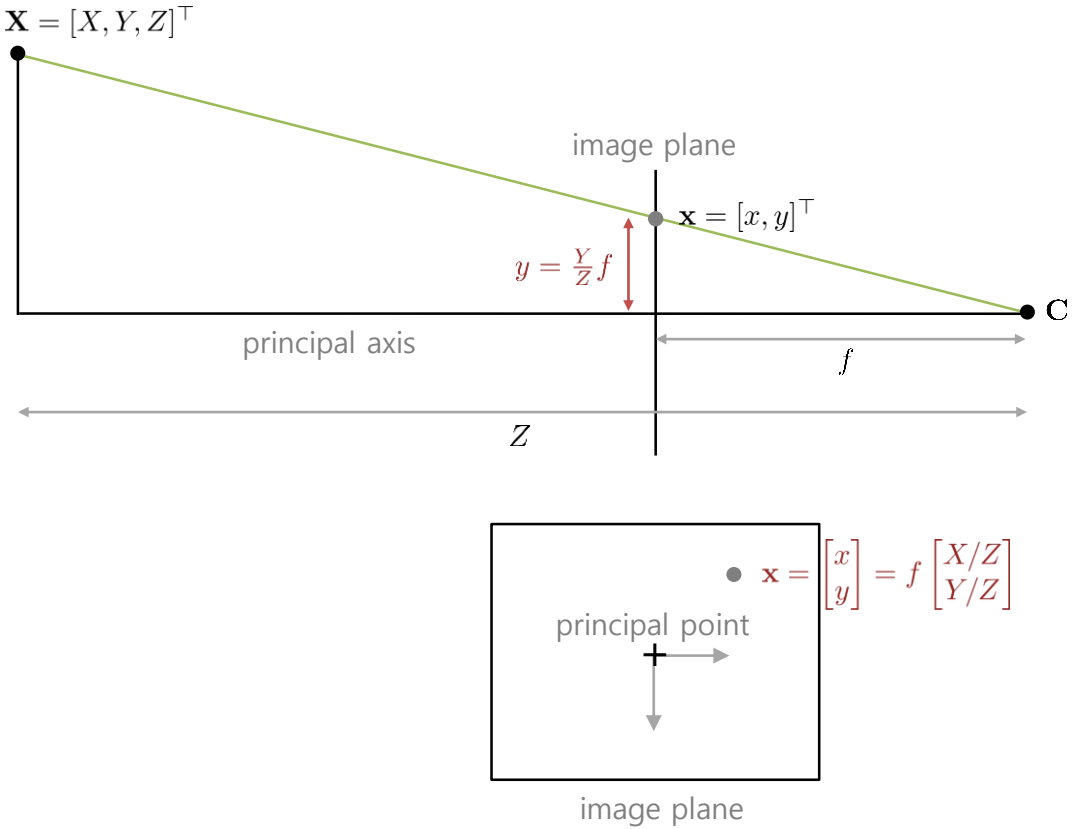
Camera Projection Model

- **The Pinhole Camera Model**



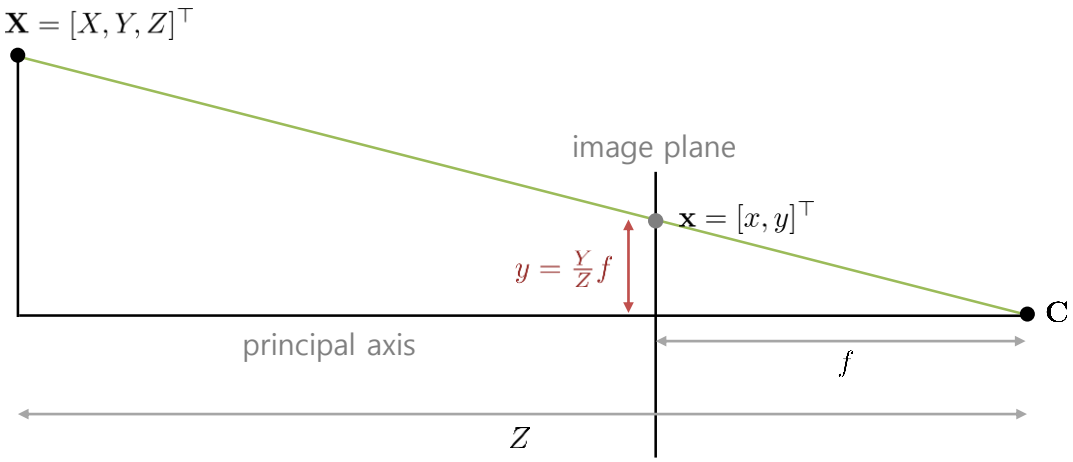
Camera Projection Model

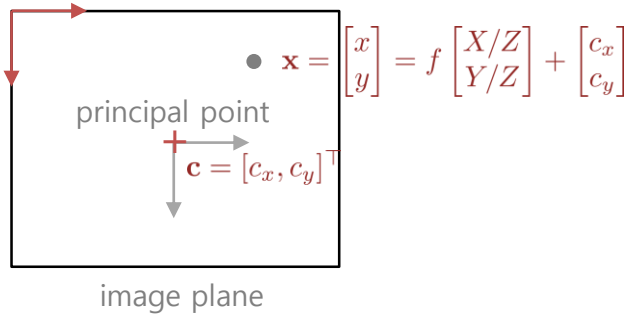
- **The Pinhole Camera Model**



Camera Projection Model

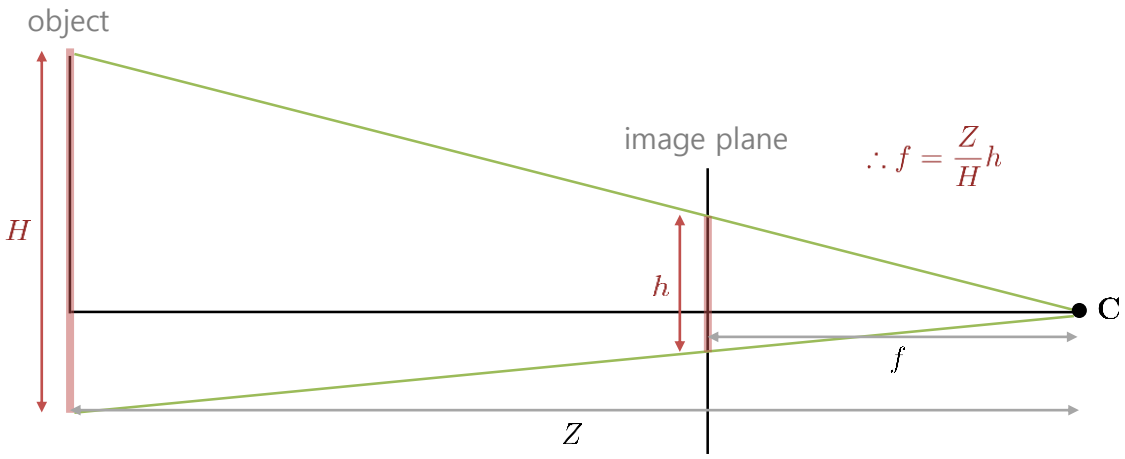
- **The Pinhole Camera Model**





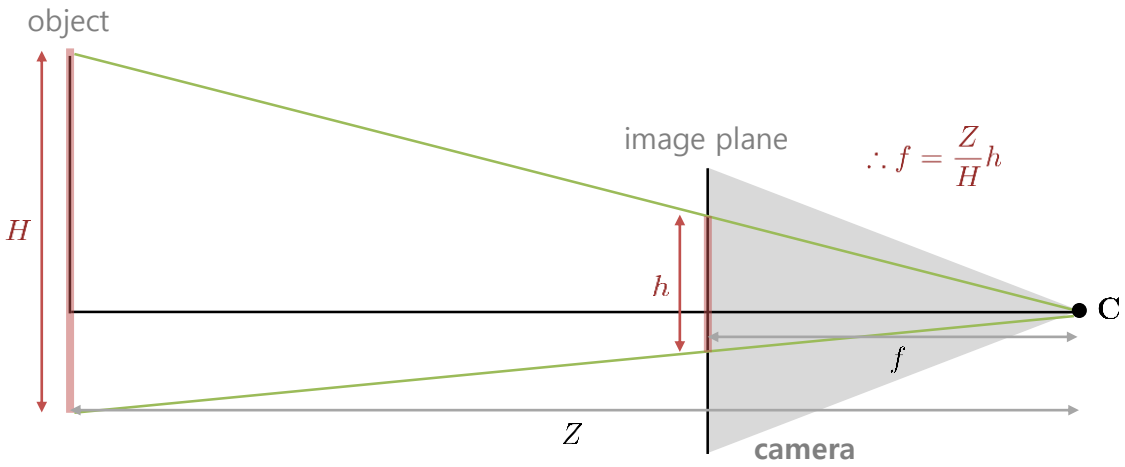
Camera Projection Model

- **The Pinhole Camera Model**
  - Example: **Simple Camera Calibration #1**
    - Unknown: **Focal length** (unit: [pixel])
    - Assumptions
      - The object length and distance from the camera are known.
      - The object is aligned with the image plane.

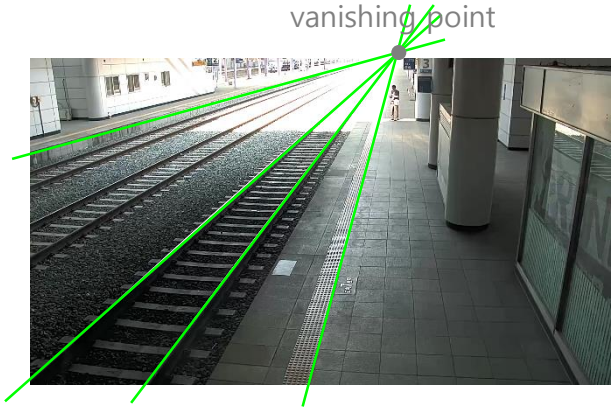


Camera Projection Model

- **The Pinhole Camera Model**
  - Example: **Simple Camera Calibration #1**
    - Unknown: **Focal length** (unit: [pixel])
    - Assumptions — Simple Object Measurement
      - The object length and distance from the camera are known.
      - The object is aligned with the image plane.

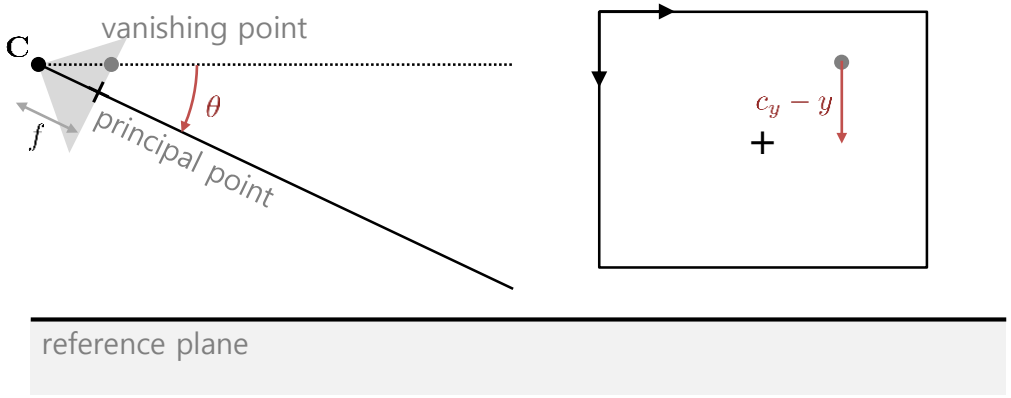


Camera Projection Model



- **The Pinhole Camera Model**
  - Example: **Simple Camera Calibration #2**
    - Unknown: **Tilt angle** (unit: [rad])
    - Assumptions
      - The focal length is known.
      - The principal point is known or selected as the center of images.
      - A vanishing point from the reference plane is known.

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

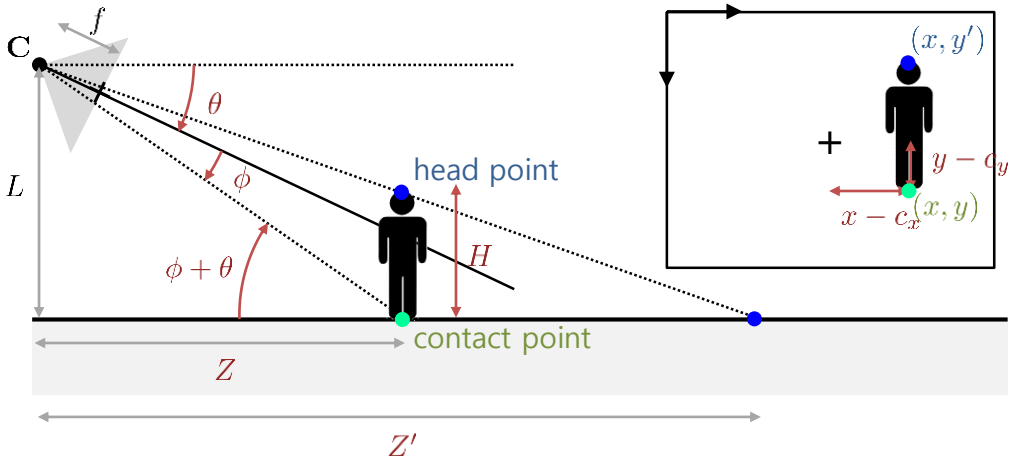


Camera Projection Model

- **The Pinhole Camera Model**
  - Example: **Simple Object Localization and Measurement**
    - Unknown: **Position** and also **height** (unit: [m])
    - Assumptions
      - The focal length, principal points, camera height, and tilt angle are known.
      - The object is on the reference plane.

$$\therefore Z = \frac{L}{\tan(\phi + \theta)} \quad \left(\phi = \tan^{-1}\frac{y - c_y}{f}\right)$$

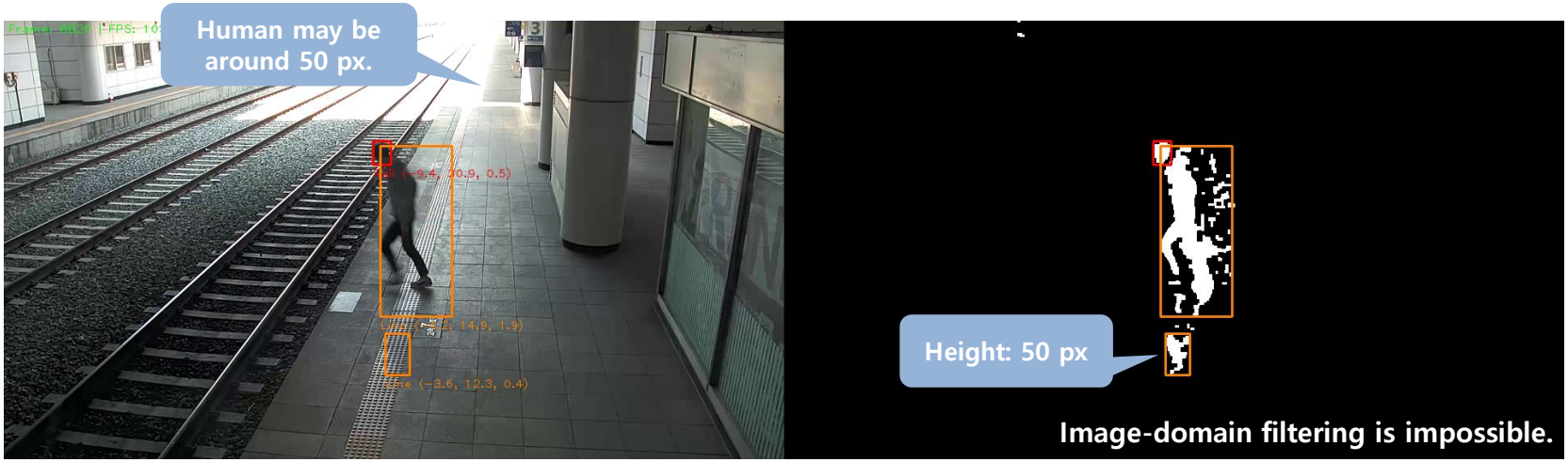
$$X = \frac{x - c_x}{f} Z \qquad H = \frac{Z' - Z}{Z'} L$$

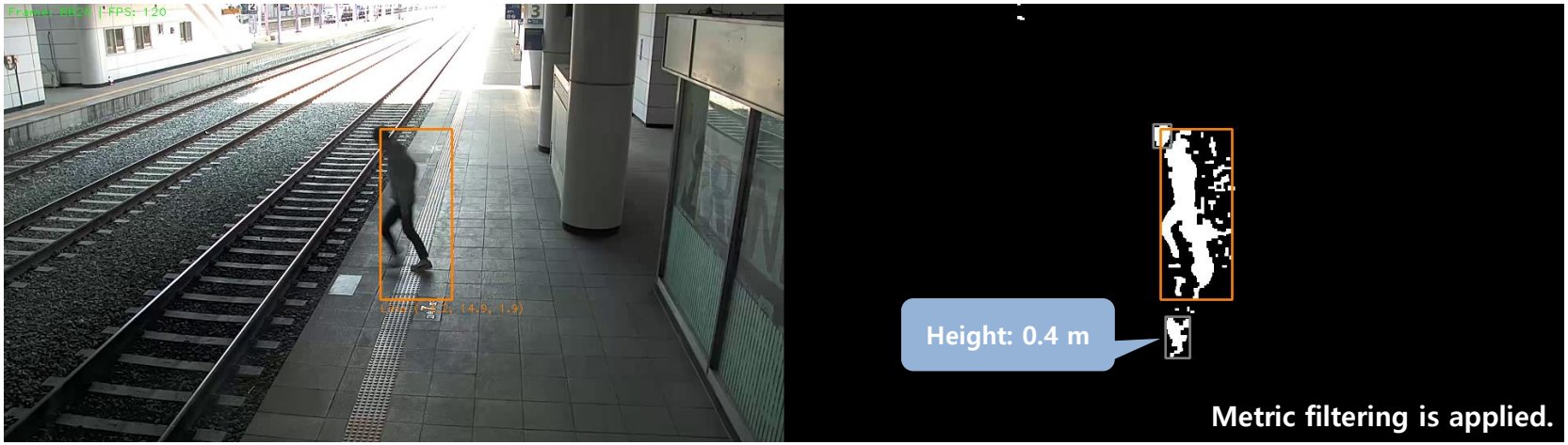


Ref. Hoiem et al., Putting Objects in Perspective, CVPR, 2006

Camera Projection Model

- **The Pinhole Camera Model**
  - Example: **Simple Object Filtering**





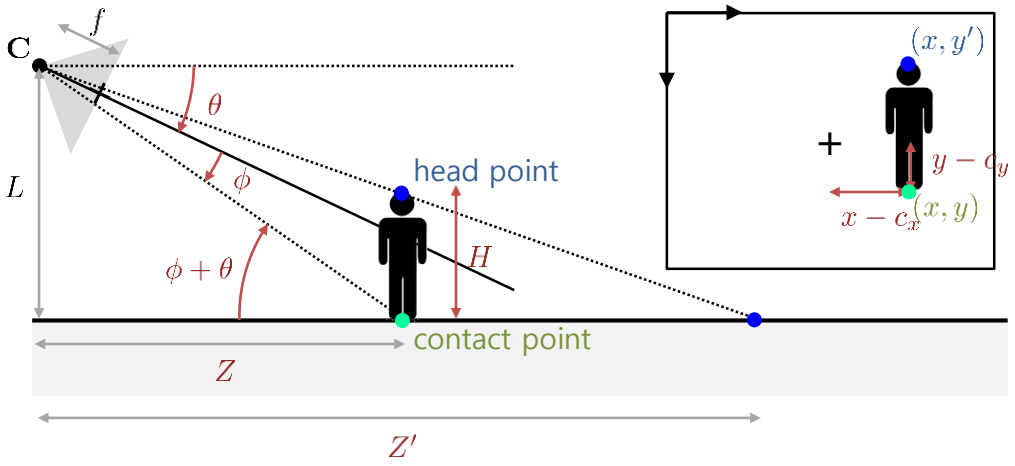
Ref. Hoiem et al., Putting Objects in Perspective, CVPR, 2006

Camera Projection Model

- **The Pinhole Camera Model**
  - Example: **Simple Object Proposal**
    - Unknown: **Height** on the given image (unit: [pixel])
    - Assumptions
      - The focal length, principal points, camera height, and tilt angle are known.
      - The object is on the reference plane and its metric height is known.

$$\therefore Z = \frac{L}{\tan(\phi + \theta)} \quad \left(\phi = \tan^{-1}\frac{y - c_y}{f}\right)$$

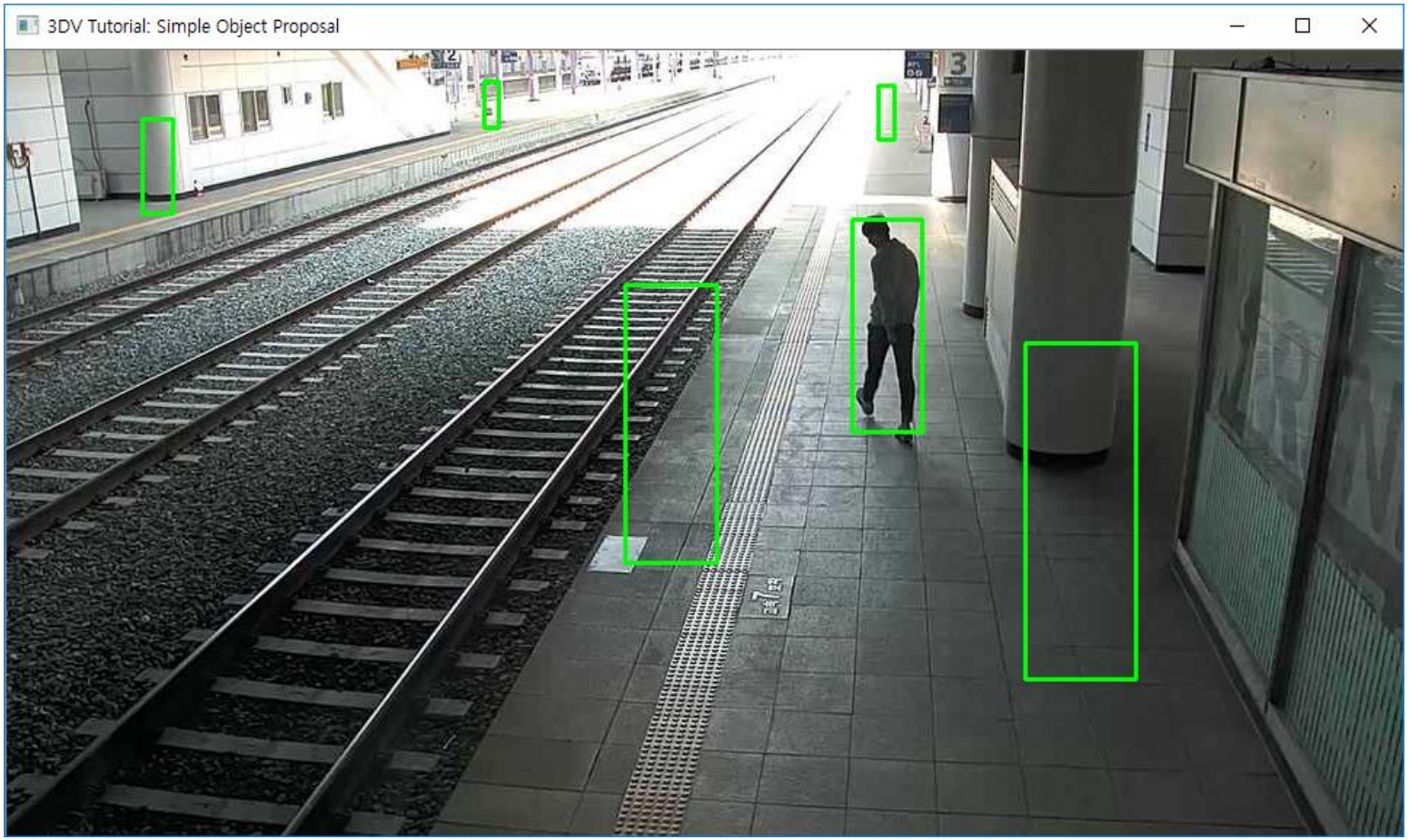
$$y' = c_y + f \cdot \tan\left(\tan^{-1}\frac{L - H}{Z} - \theta\right)$$



Ref. Hoiem et al., Putting Objects in Perspective, CVPR, 2006

Camera Projection Model

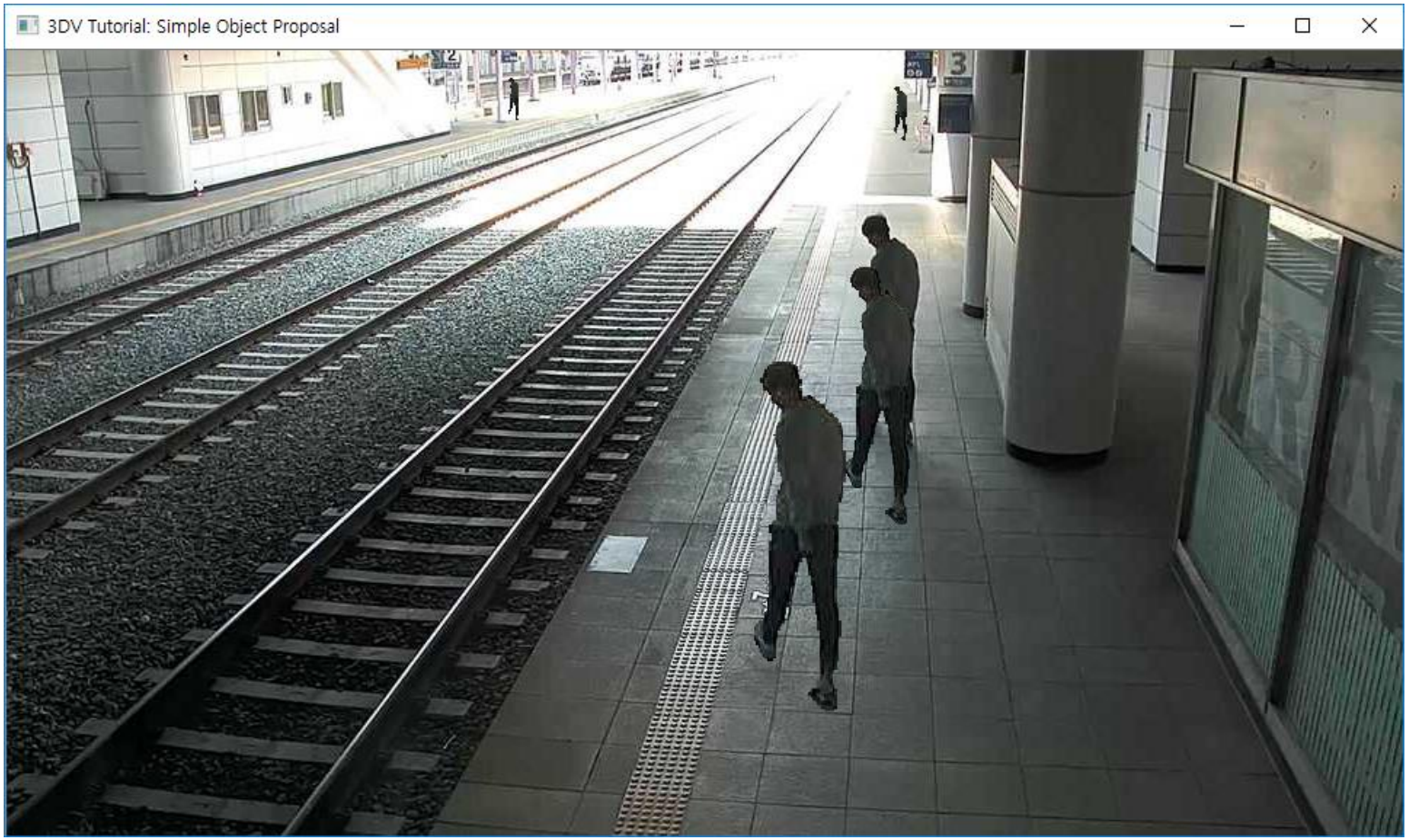
- **The Pinhole Camera Model**
  - Example: **Simple Object Proposal**



Ref. Hoiem et al., Putting Objects in Perspective, CVPR, 2006

Camera Projection Model

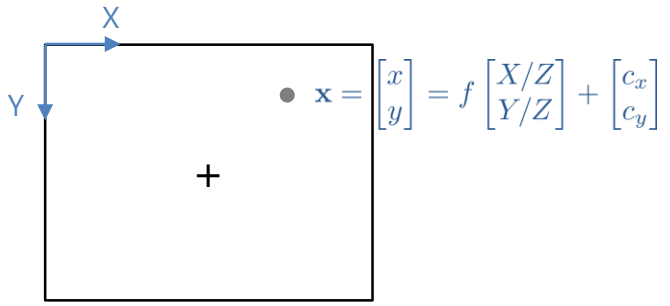
- **The Pinhole Camera Model**
  - Example: **Simple Object Proposal**



Ref. Hoiem et al., Putting Objects in Perspective, CVPR, 2006

Camera Projection Model

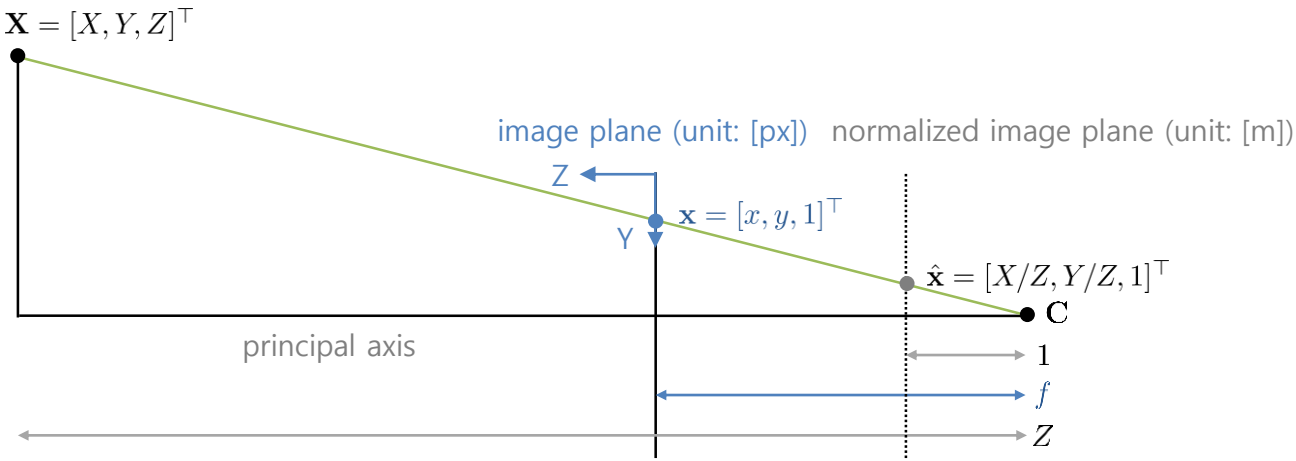
- **Camera Matrix** (Intrinsic Parameters)



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

generalized

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



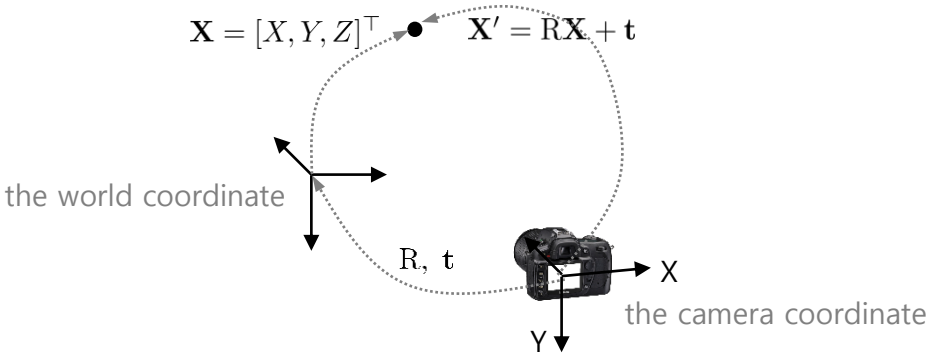
Camera Projection Model

- **Projection Matrix** (Intrinsic/Extrinsic Parameters)
  - If a 3D point is not based on the camera coordinate, the point should be transformed.

$$\mathbf{X}' = \mathrm{R}\mathbf{X} + \mathbf{t} \longrightarrow \mathbf{X}' = \underbrace{\left[\mathrm{R}\,|\,\mathbf{t}\right]}_{\text{3x4 matrix}}\mathbf{X} \quad \text{where} \quad \mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$\mathbf{x} = \mathrm{P}\mathbf{X} \quad \text{where} \quad \mathrm{P} = \mathrm{K}\underbrace{\left[\mathrm{R}\,|\,\mathbf{t}\right]}_{\text{3x4 matrix}} \quad \text{and} \quad \mathbf{x} = \begin{bmatrix} x \\ y \\ w \end{bmatrix} \quad \because \mathbf{X}' \text{ is not normalized.}$$

  - No problem! To get its projected point on the image plane, we can normalize it later, $\begin{bmatrix} x/w \\ y/w \\ 1 \end{bmatrix}$.

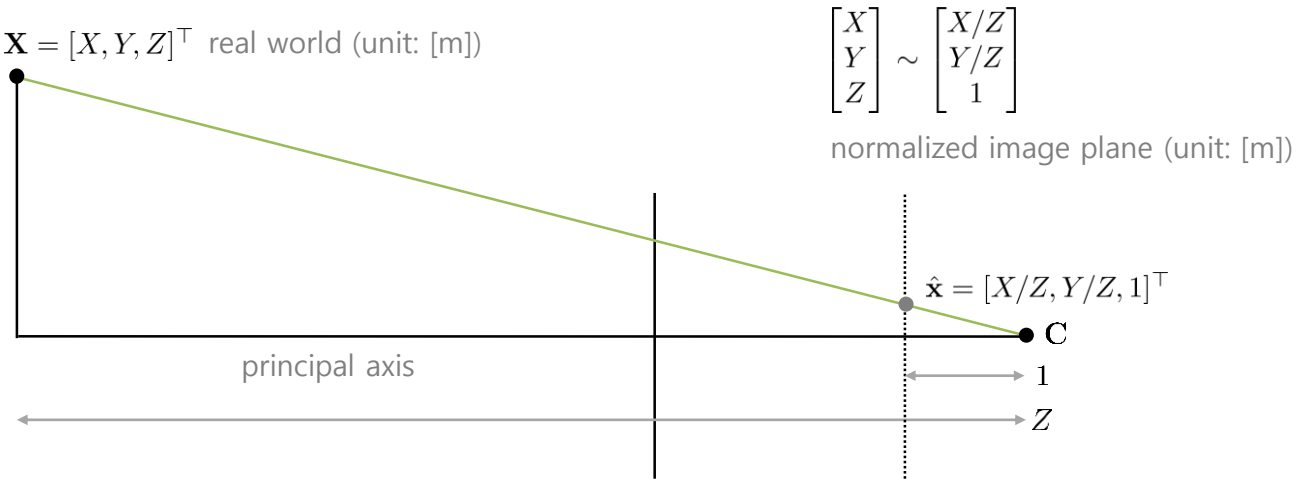


camera pose $\mathrm{R}^\top, -\mathrm{R}^\top\mathbf{t}$ $\quad \because \begin{bmatrix} \mathrm{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathrm{R}^\top & -\mathrm{R}^\top\mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$

Camera Projection Model

$$\therefore \mathbf{x} = \mathrm{P}\mathbf{X} \quad \text{where} \quad \mathrm{P} = \mathrm{K}\left[\mathrm{R} \,|\, \mathbf{t}\right], \; \mathbf{x} = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, \text{ and } \mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

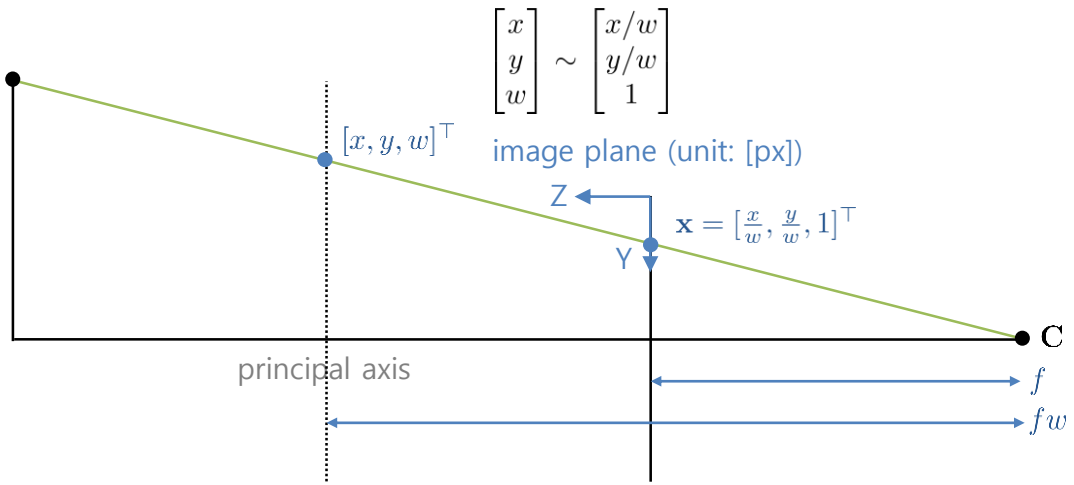
- **Why Homogeneous Coordinate?**
  - A linear transformation (rotation and translation) is applied by a single matrix multiplication.
  - A light ray can be represented as a point on the image plane.
  - An infinite point (a.k.a. ideal point) is numerically represented by $w = 0$.
  - A point and line are represented beautifully as like $\mathbf{l}^\top \mathbf{x} = 0$ or $\mathbf{x}^\top \mathbf{l} = 0$ ($\mathbf{l} = [a, b, c]^\top$).
  - …



Camera Projection Model

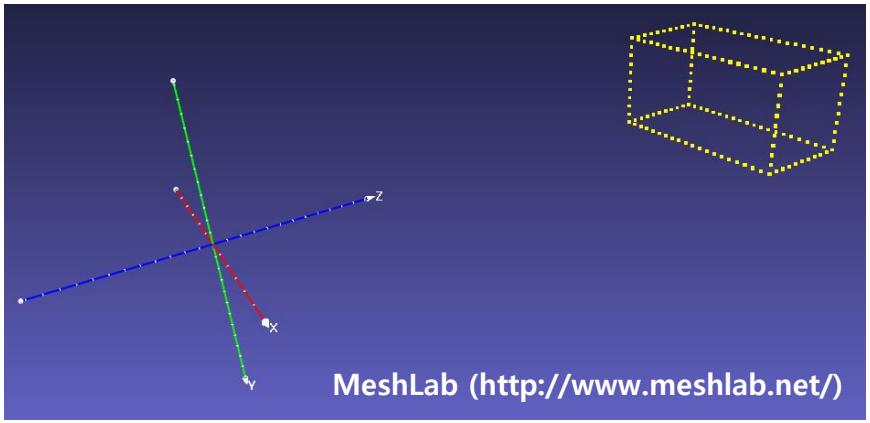
$$\therefore \mathbf{x} = \mathrm{P}\mathbf{X} \quad \text{where} \quad \mathrm{P} = \mathrm{K}\left[\mathrm{R} \,|\, \mathbf{t}\right], \; \mathbf{x} = \begin{bmatrix} x \\ y \\ w \end{bmatrix}, \text{ and } \mathbf{X} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

- **Why Homogeneous Coordinate?**
  - A linear transformation (rotation and translation) is applied by a single matrix multiplication.
  - A light ray can be represented as a point on the image plane.
  - An infinite point (a.k.a. ideal point) is numerically represented by $w = 0$.
  - A point and line are represented beautifully as like $\mathbf{l}^\top \mathbf{x} = 0$ or $\mathbf{x}^\top \mathbf{l} = 0$.
  - …



Example: Image Formation


```
1.  #include "opencv_all.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(void)
6.  {
7.      // The given camera configuration: focal length, principal point, image resolution, position, and orientation
8.      double camera_focal = 1000;
9.      cv::Point2d camera_center(320, 240);
10.     cv::Size camera_res(640, 480);
11.     cv::Point3d camera_pos[] = { cv::Point3d(0, 0, 0), cv::Point3d(-2, -2, 0), ... };
12.     cv::Point3d camera_ori[] = { cv::Point3d(0, 0, 0), cv::Point3d(-CV_PI / 12, CV_PI / 12, 0), ... };
13.     double camera_noise = 1;

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

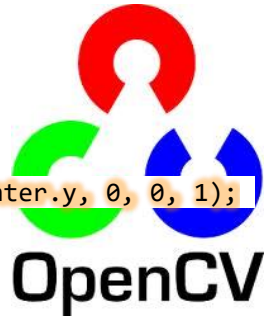


A point cloud: data/box.xyz

ETRI

Example: Image Formation

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




```
31.     // Generate images for each camera pose
32.     cv::Mat K = (cv::Mat_<double>(3, 3) << camera_focal, 0, camera_center.x, 0, camera_focal, camera_center.y, 0, 0, 1);
33.     for (int i = 0; i < sizeof(camera_pos) / sizeof(cv::Point3d); i++)
34.     {
35.         // Derive a projection matrix
36.         cv::Mat Rc = Rz(camera_ori[i].z) * Ry(camera_ori[i].y) * Rx(camera_ori[i].x);
37.         cv::Mat tc = (cv::Mat_<double>(3, 1) << camera_pos[i].x, camera_pos[i].y, camera_pos[i].z);
38.         cv::Mat Rt;
39.         cv::hconcat(Rc.t(), -Rc.t() * tc, Rt);
40.         cv::Mat P = K * Rt;
```

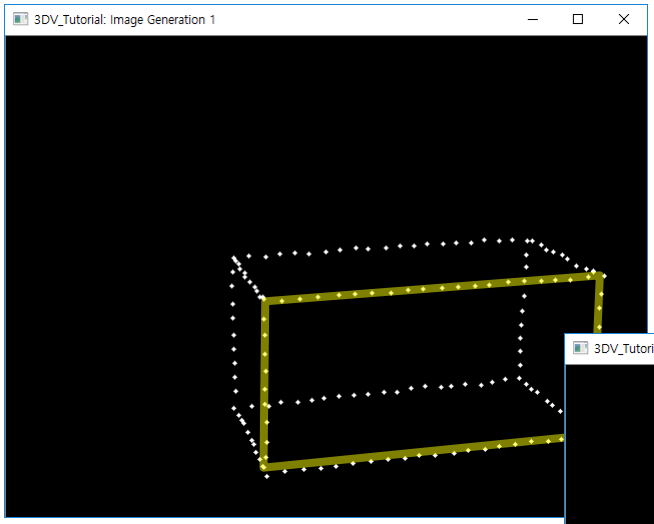
$$\because \begin{bmatrix} \mathrm{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathrm{R}^\top & -\mathrm{R}^\top \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix}$$

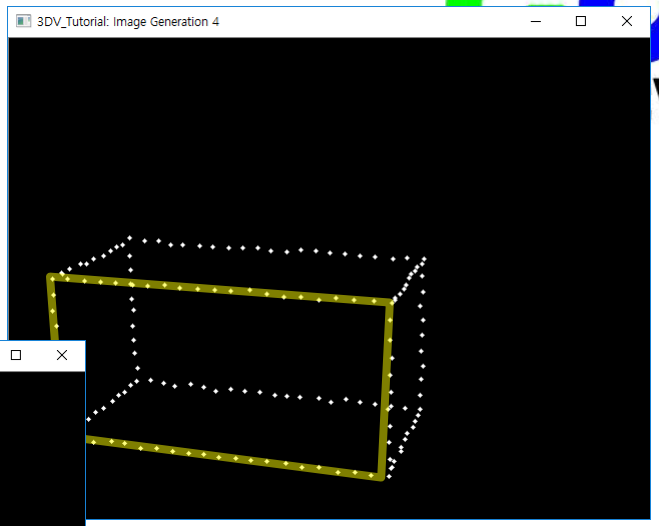
```
41.         // Project the points (c.f. OpenCV provide 'cv::projectPoints()' with consideration of distortion.)
42.         cv::Mat x = P * X;
43.         x.row(0) = x.row(0) / x.row(2);
44.         x.row(1) = x.row(1) / x.row(2);
45.         x.row(2) = 1;
```

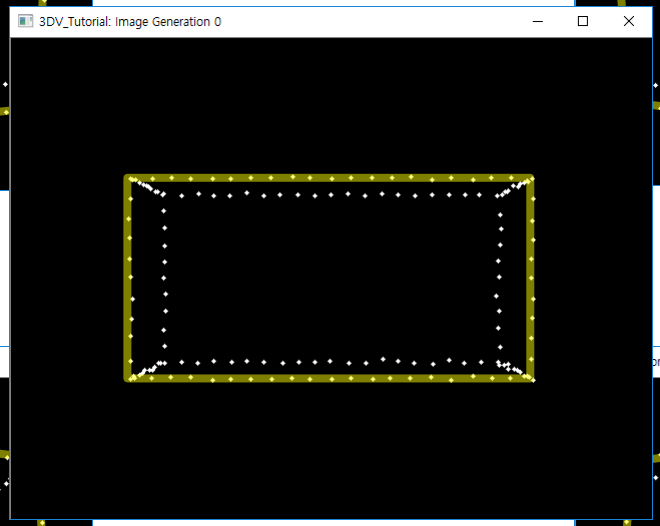
$$\because \begin{bmatrix} x \\ y \\ w \end{bmatrix} \sim \begin{bmatrix} x/w \\ y/w \\ 1 \end{bmatrix}$$

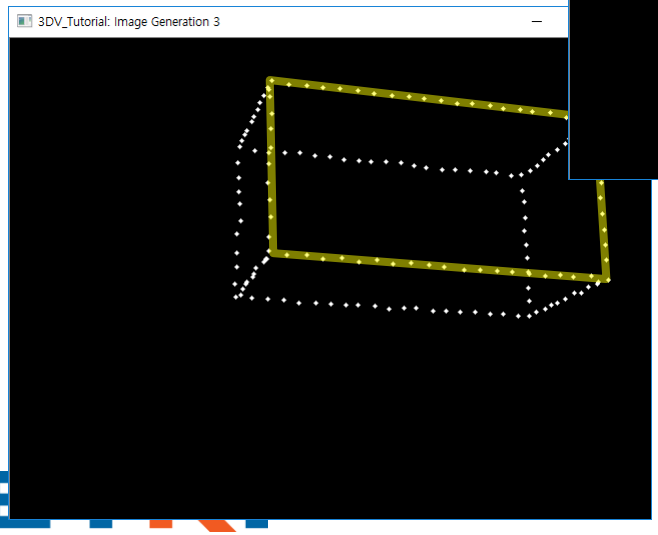
```
46.         cv::Mat noise(2, x.cols, x.type());
47.         cv::randn(noise, cv::Scalar(0), cv::Scalar(camera_noise));
48.         x.rowRange(0, 2) = x.rowRange(0, 2) + noise; // Add noise

49.         // Show and store the points
50.         cv::Mat image = cv::Mat::zeros(camera_res, CV_8UC1);
51.         for (int c = 0; c < x.cols; c++)
52.         {
53.             cv::Point p(x.at<double>(0, c), x.at<double>(1, c));
54.             if (p.x >= 0 && p.x < camera_res.width && p.y >= 0 && p.y < camera_res.height)
55.                 cv::circle(image, p, 2, 255, -1);
56.         }
57.         cv::imshow(cv::format("3DV_Tutorial: Image Formation %d", i), image);

58.         FILE* fout = fopen(cv::format("image_formation%d.xyz", i).c_str(), "wt");
59.         if (fout == NULL) return -1;
60.         for (int c = 0; c < x.cols; c++)
61.             fprintf(fout, "%f %f 1\n", x.at<double>(0, c), x.at<double>(1, c));
62.         fclose(fout);
63.     }
64.     ...
65. }
```

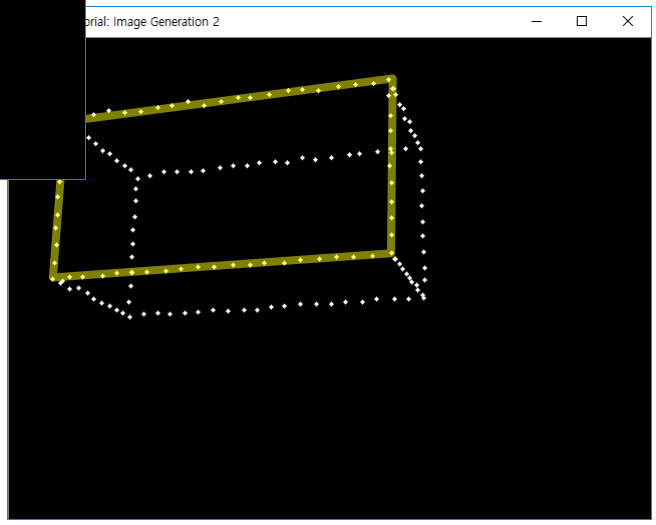
Example: Image Formation



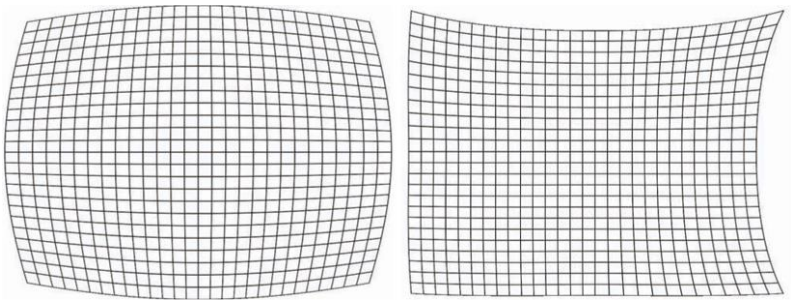








Camera Projection Model



Radial Distortion Tangential Distortion

- **Geometric Distortion**
  - Radial/Tangential Distortion Model

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

where $r^2 = \hat{x}^2 + \hat{y}^2$

defined on the normalized image plane
- Radial distortion: k_1, k_2, ...
- Tangential distortion: p_1, p_2

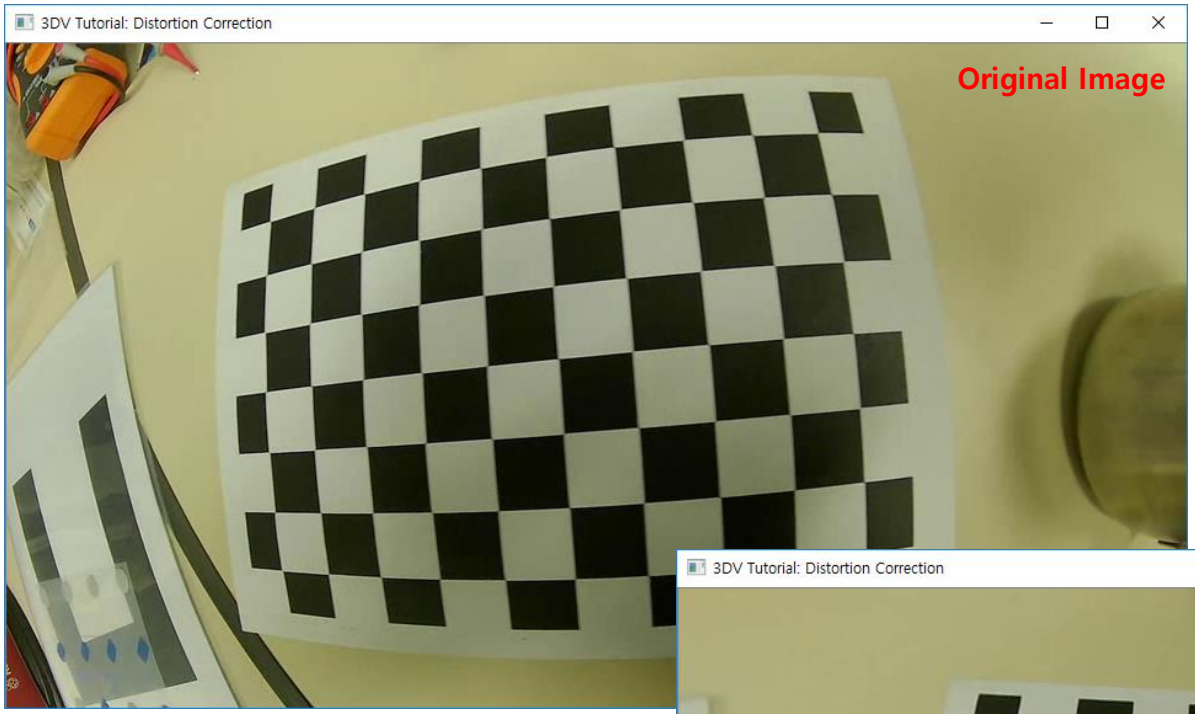
  - Image Rectification
    - OpenCV `cv::undistort()` and `cv::undistortPoints()` (c.f. included in `imgproc` module)
      ↔ `cv::projectPoints()` (c.f. included in `calib3d` module)
    - Camera Distortion Correction: Theory and Practice, http://darkpgmr.tistory.com/31

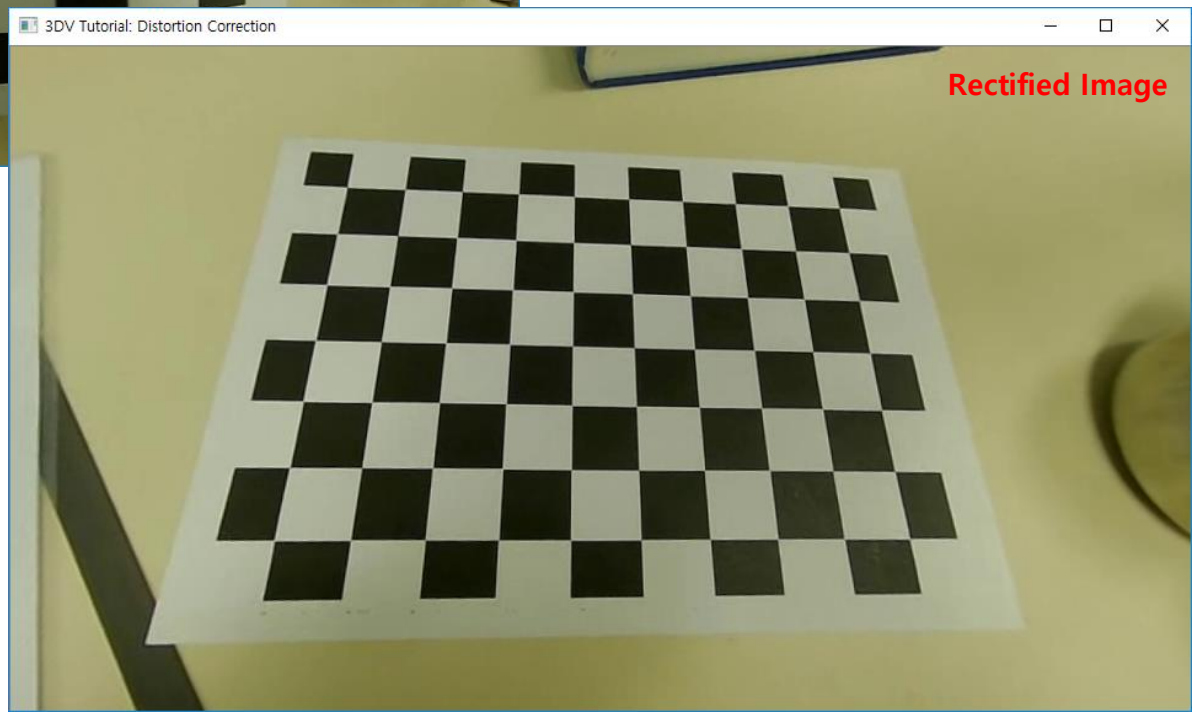
correction

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

Images are from IEEE and http://alumni.media.mit.edu/~sbeck/.

Example: Geometric Distortion Correction



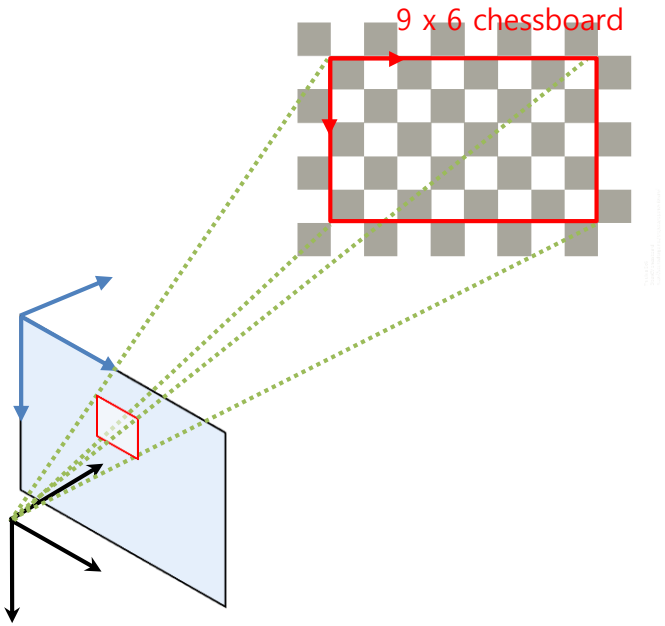


Example: Geometric Distortion Correction


```
1.  #include "opencv_all.hpp"

2.  int main(void)
3.  {
4.      cv::Mat K = (cv::Mat_<double>(3, 3) << 432.7390364738057, 0, 476.0614994349778, ..., 0, 0, 1);
5.      cv::Mat dist_coeff = (cv::Mat_<double>(5, 1) << -0.2852754904152874, 0.1016466459919075, ...);

6.      // Open a video
7.      cv::VideoCapture video;
8.      if (!video.open("data/chessboard.avi")) return -1;

9.      // Run distortion correction
10.     bool show_rectify = true;
11.     cv::Mat map1, map2;
12.     while (true)
13.     {
14.         // Grab an image from the video
15.         cv::Mat image;
16.         video >> image;
17.         if (image.empty()) break;

18.         // Rectify geometric distortion (c.f. 'cv::undistort()' can be applied for one-time remapping.)
19.         if (show_rectify)
20.         {
21.             if (map1.empty() || map2.empty())
22.                 cv::initUndistortRectifyMap(K, dist_coeff, cv::Mat(), cv::Mat(), image.size(), CV_32FC1, map1, map2);
23.             cv::remap(image, image, map1, map2, cv::InterpolationFlags::INTER_LINEAR);
24.         }

25.         // Show the image
26.         cv::imshow("3DV Tutorial: Distortion Correction", image);
27.         int key = cv::waitKey(1);
28.         if (key == 27) break;                                   // 'ESC' key: Exit
29.         else if (key == 9) show_rectify = !show_rectify;        // 'Tab' key: Toggle rectification
30.         ...
31.     }

32.     video.release();
33.     return 0;
34. }
```

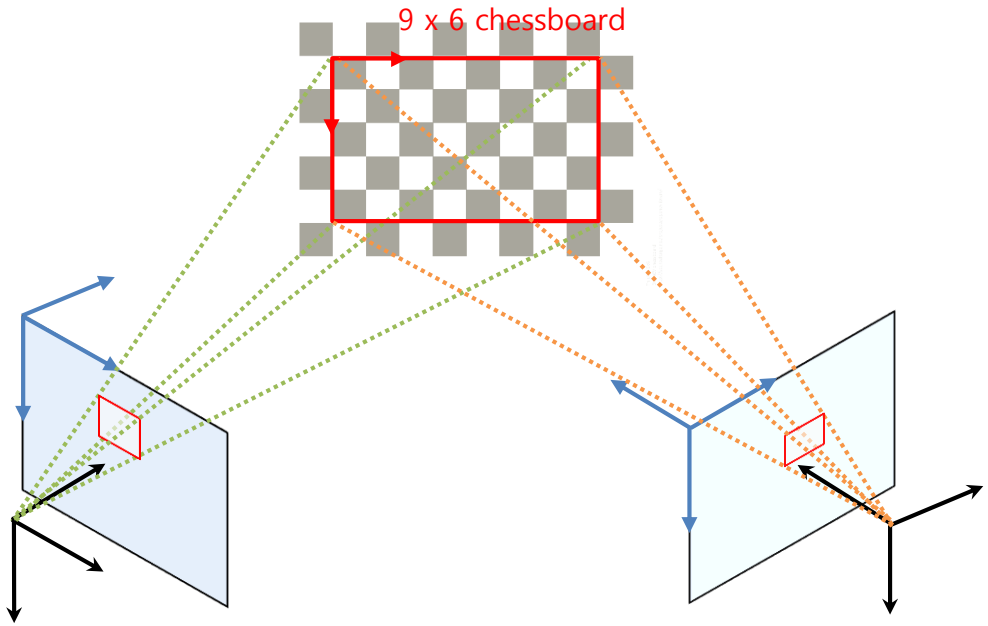
General 2D-3D Geometry

- **Camera Calibration**
  - Known: 3D points $\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_n$ and their projected points $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n$
  - Unknown: 5 (intrinsic w/o distortion) + 6 (extrinsic) DoF
  - Constraints: *n* x projection $\mathbf{x}_i = \mathrm{P}\mathbf{X}_i$



General 2D-3D Geometry

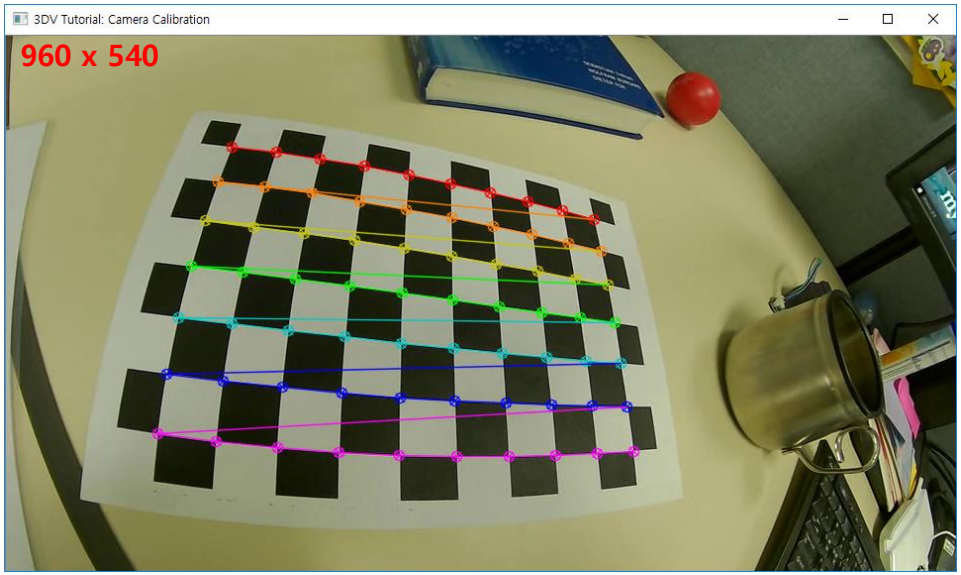
- **Camera Calibration**
  - Known: 3D points $\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_n$ and their projected points from each camera $\mathbf{x}_i^j$
  - Unknown: 5 (intrinsic w/o distortion) + *m* x 6 (extrinsic) DoF
  - Constraints: *n* x *m* x projection $\mathbf{x}_i^j = \mathrm{K}\left[\mathrm{R}_j \,|\, \mathbf{t}_j\right]\mathbf{X}_i$
  - 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



Example: Camera Calibration


```
1. #include "opencv_all.hpp"

2. int main(void)
3. {
4.     bool select_images = true;
5.     cv::Size board_pattern(10, 7);
6.     float board_cellsize = 0.025f;

7.     // Select images
8.     std::vector<cv::Mat> images;
9.     ...

10.    // Find 2D corner points from the given images
11.    std::vector<std::vector<cv::Point2f> > img_points;
12.    for (size_t i = 0; i < images.size(); i++)
13.    {
14.        std::vector<cv::Point2f> pts;
15.        if (cv::findChessboardCorners(images[i], board_pattern, pts))
16.            img_points.push_back(pts);
17.    }
18.    if (img_points.empty()) return -1;

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

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

30.    // Report calibration results
31.    ...
32.    return 0;
33. }
```

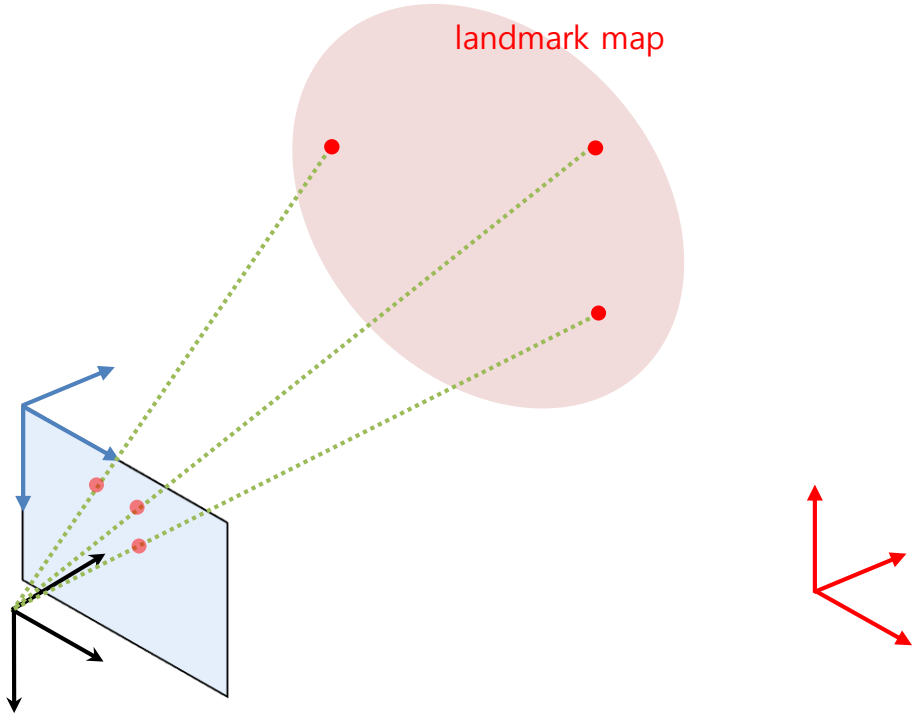
Example: Camera Calibration




```
## Camera Calbration 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]
```

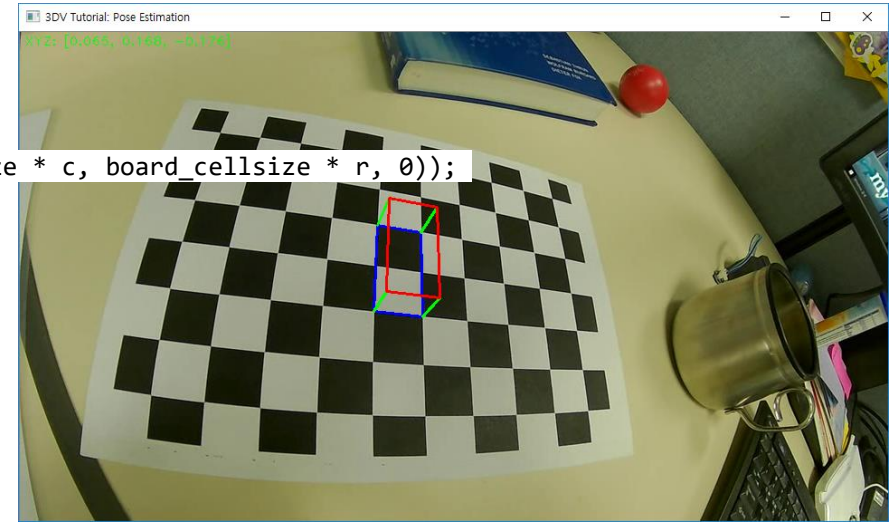
General 2D-3D Geometry

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



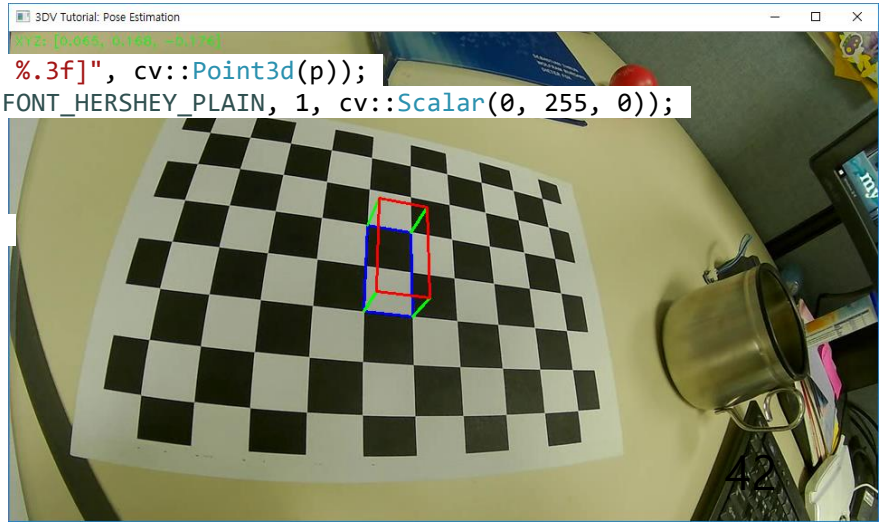
Example: Camera Pose Estimation (Chessboard)

```
1. #include "opencv_all.hpp"

2. int main(void)
3. {
4.     cv::Mat K = (cv::Mat_<double>(3, 3) << 432.7390364738057, 0, 476.0614994349778, ...);
5.     cv::Mat dist_coeff = (cv::Mat_<double>(5, 1) << -0.2852754904152874, 0.1016466459919075, ...);
6.     cv::Size board_pattern(10, 7);
7.     double board_cellsize = 0.025;

8.     // Open a video
9.     cv::VideoCapture video;
10.    if (!video.open("data/chessboard.avi")) return -1;

11.    // Prepare a box for simple AR
12.    std::vector<cv::Point3d> box_lower, box_upper;
13.    box_lower.push_back(cv::Point3d(4 * board_cellsize, 2 * board_cellsize, 0));
14.    box_lower.push_back(cv::Point3d(5 * board_cellsize, 2 * board_cellsize, 0));
15.    box_lower.push_back(cv::Point3d(5 * board_cellsize, 4 * board_cellsize, 0));
16.    box_lower.push_back(cv::Point3d(4 * board_cellsize, 4 * board_cellsize, 0));
17.    box_upper.push_back(cv::Point3d(4 * board_cellsize, 2 * board_cellsize, -board_cellsize));
18.    box_upper.push_back(cv::Point3d(5 * board_cellsize, 2 * board_cellsize, -board_cellsize));
19.    box_upper.push_back(cv::Point3d(5 * board_cellsize, 4 * board_cellsize, -board_cellsize));
20.    box_upper.push_back(cv::Point3d(4 * board_cellsize, 4 * board_cellsize, -board_cellsize));

21.    // Run pose estimation
22.    std::vector<cv::Point3d> obj_points;
23.    for (int r = 0; r < board_pattern.height; r++)
24.        for (int c = 0; c < board_pattern.width; c++)
25.            obj_points.push_back(cv::Point3d(board_cellsize * c, board_cellsize * r, 0));
```

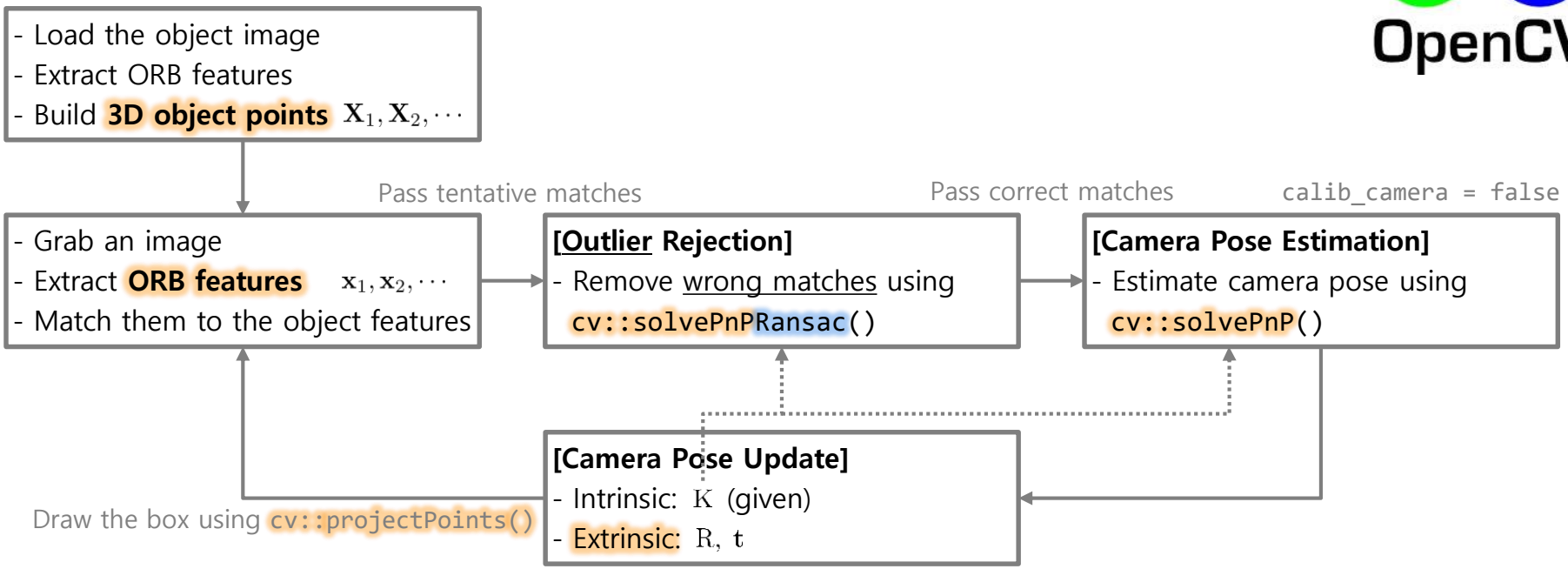


Example: Camera Pose Estimation (Chessboard)

```
1. while (true)
2.     {
3.         // Grab an image from the video
4.         cv::Mat image;
5.         video >> image;
6.         if (image.empty()) break;

7.         // Estimate camera pose
8.         std::vector<cv::Point2d> img_points;
9.         bool success = cv::findChessboardCorners(image, board_pattern, img_points, ...);
10.        if (success)
11.        {
12.            cv::Mat rvec, tvec;
13.            cv::solvePnP(obj_points, img_points, K, dist_coeff, rvec, tvec);

14.            // Draw the box on the image
15.            cv::Mat line_lower, line_upper;
16.            cv::projectPoints(box_lower, rvec, tvec, K, dist_coeff, line_lower);
17.            cv::projectPoints(box_upper, rvec, tvec, K, dist_coeff, line_upper);
18.            ...

19.            // Print camera position
20.            cv::Mat R;
21.            cv::Rodrigues(rvec, R);
22.            cv::Mat p = -R.t() * tvec;
23.            cv::String info = cv::format("XYZ: [%.3f, %.3f, %.3f]", cv::Point3d(p));
24.            cv::putText(image, info, cv::Point(5, 15), cv::FONT_HERSHEY_PLAIN, 1, cv::Scalar(0, 255, 0));
25.        }

26.        // Show the image
27.        cv::imshow("3DV Tutorial: Pose Estimation", image);
28.        int key = cv::waitKey(1);
29.        ...
30.    }

31.    video.release();
32.    return 0;
33. }
```

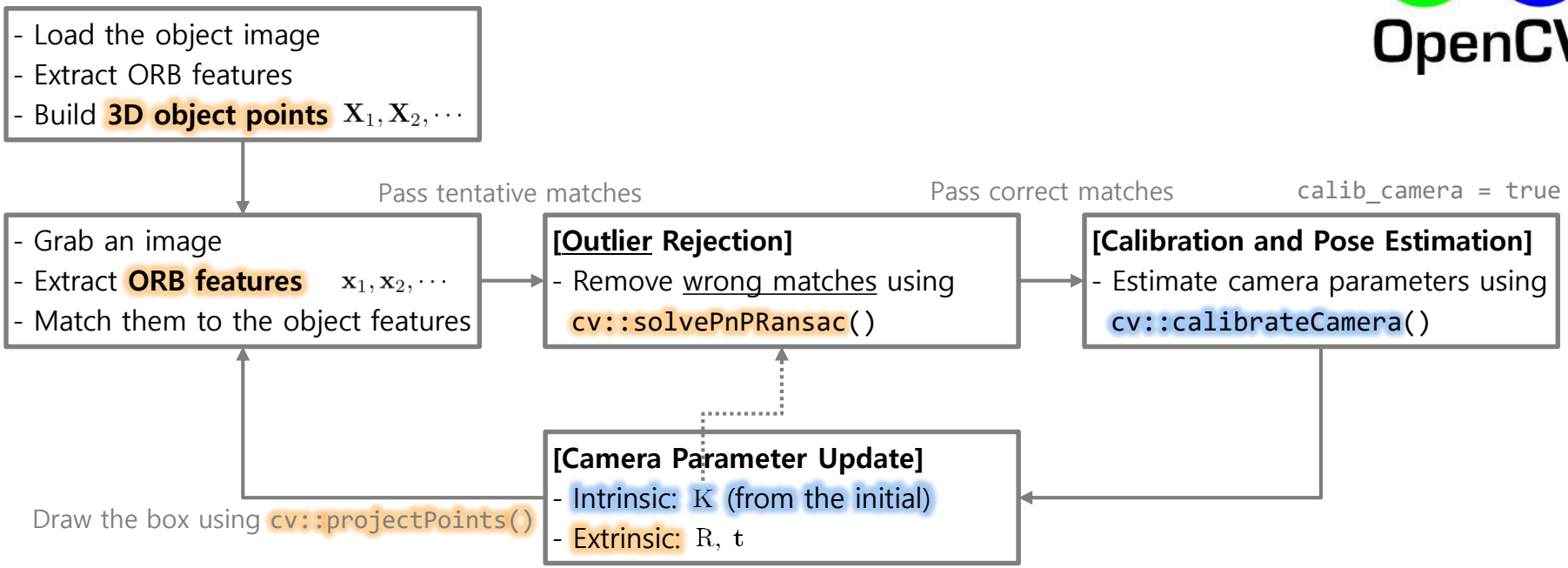


Example: Camera Pose Estimation (Book)



Example: Camera Calibration and Pose Estimation (Book)

(∵ initially known camera parameters + autofocus)



Example: Camera Calibration and Pose Estimation (Book)

(∵ unknown camera parameters + autofocus)

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

Pass tentative matches

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

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

`assume_plane = true`

Pass correct matches

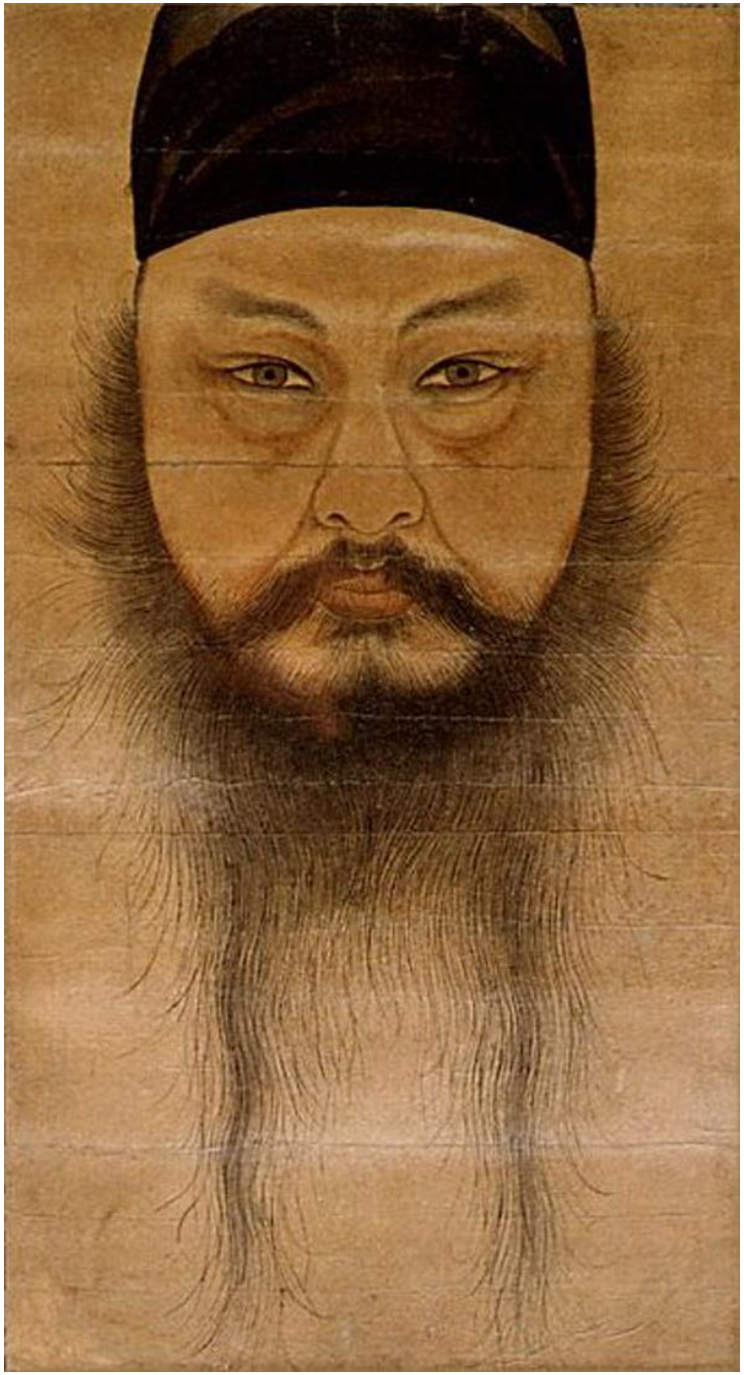
`calib_camera = true`

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

[Camera Parameter Update]
- Intrinsic: K (from the initial)
- Extrinsic: R, $\mathbf{t}$

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

Two-view Geometry

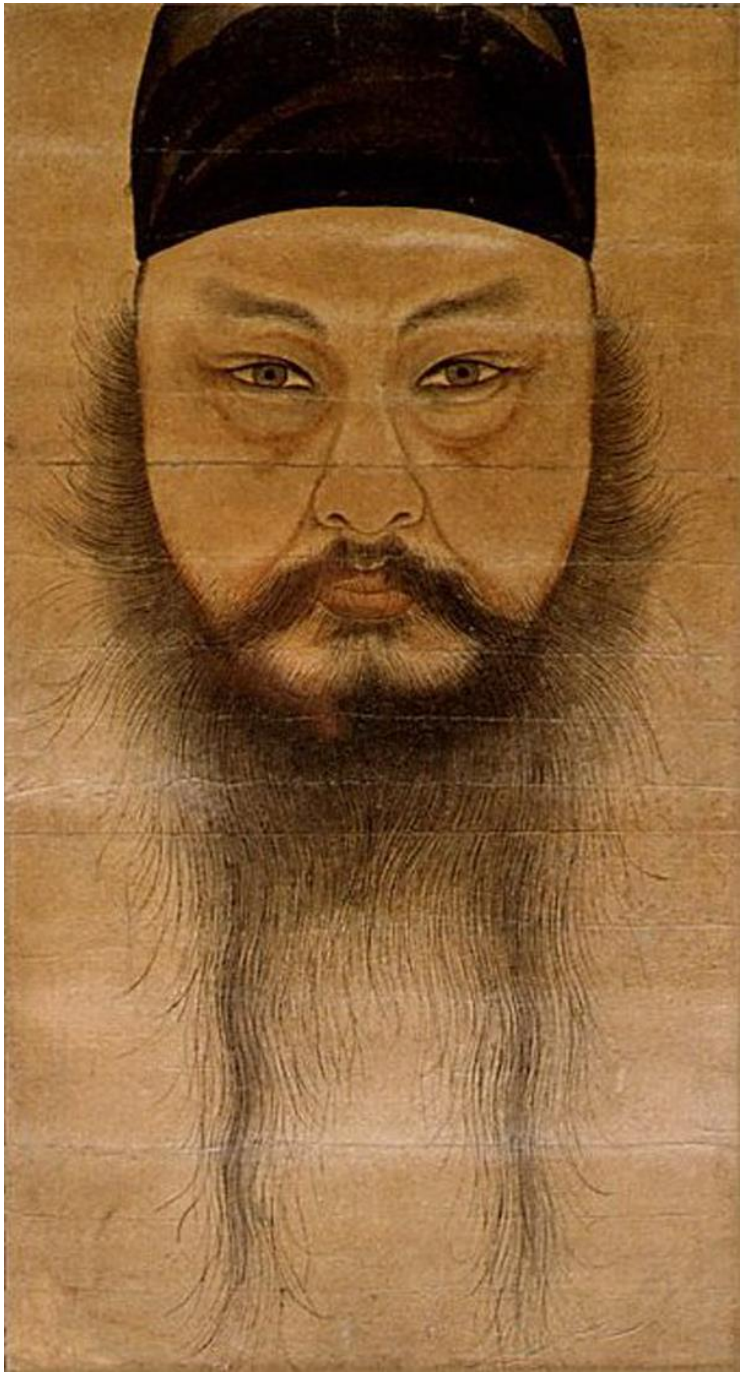


윤두서(1668-1715) 자화상, 국보 제240호
Korean National Treasure No. 240

Two-view Geometry

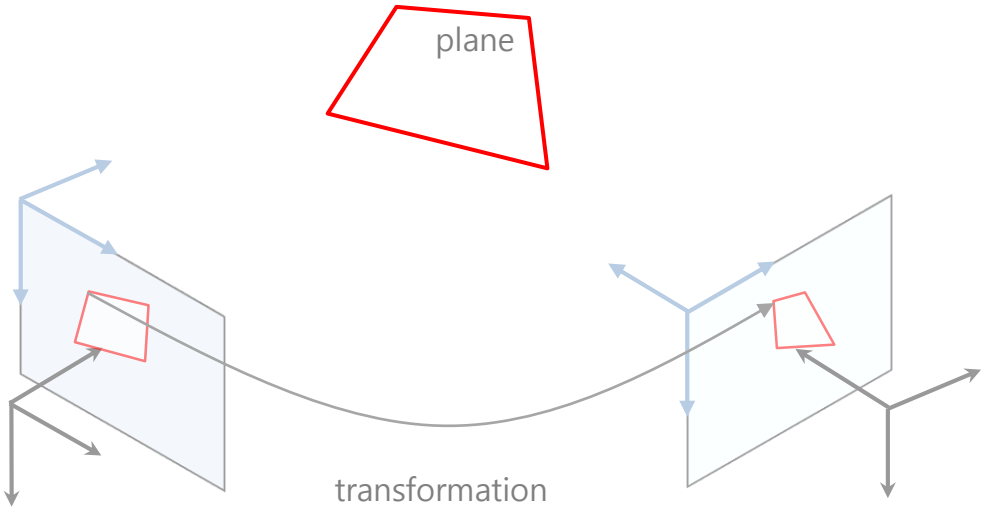
Planar 2D-2D Geometry (Projective Geometry)

General 2D-2D Geometry (Epipolar Geometry)

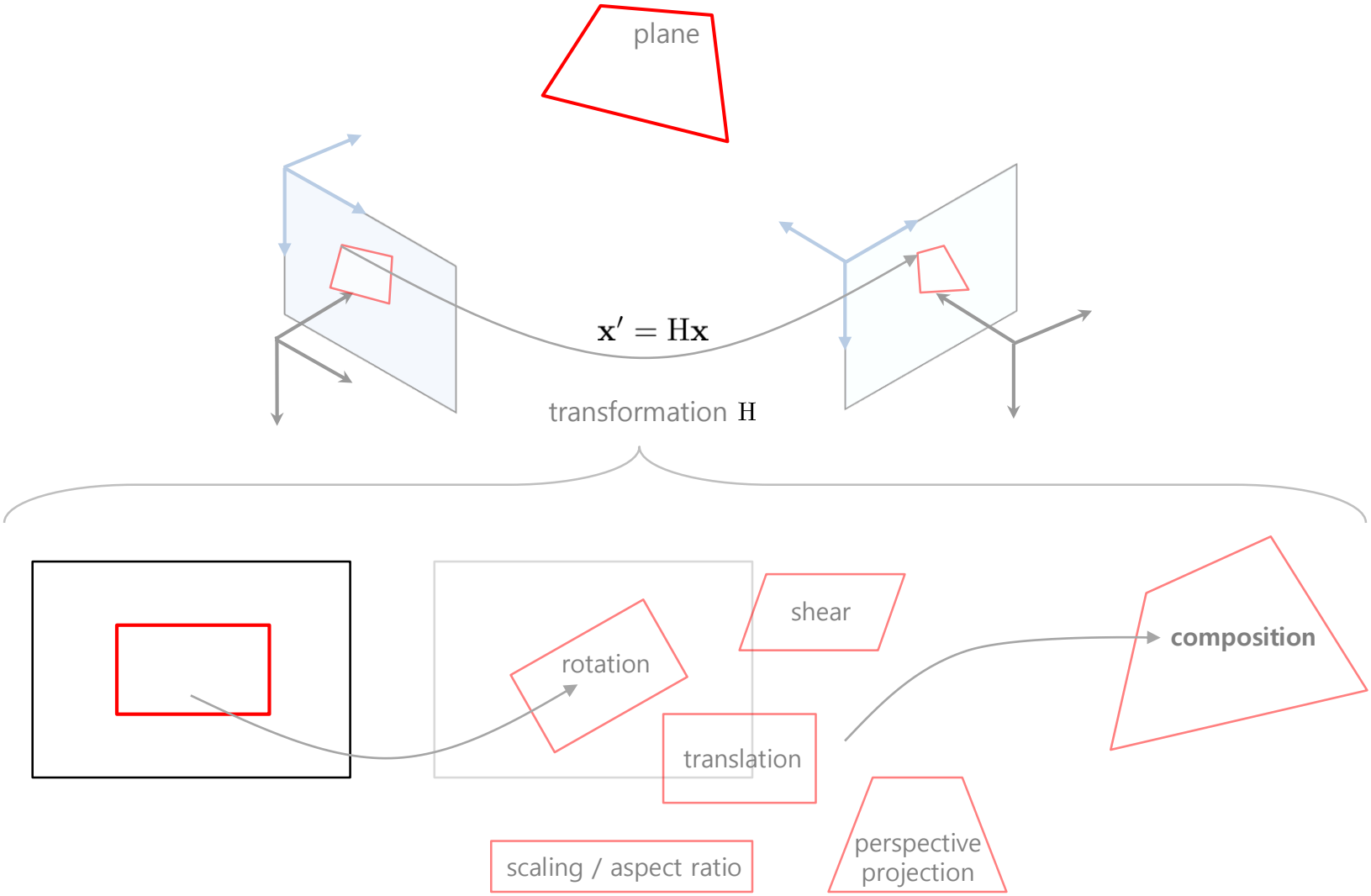


윤두서(1668-1715) 자화상, 국보 제240호
Korean National Treasure No. 240

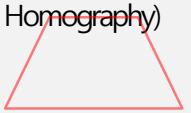
Planar 2D-2D Geometry (Projective Geometry)



Planar 2D-2D Geometry (Projective Geometry)



Overview of Projective Geometry

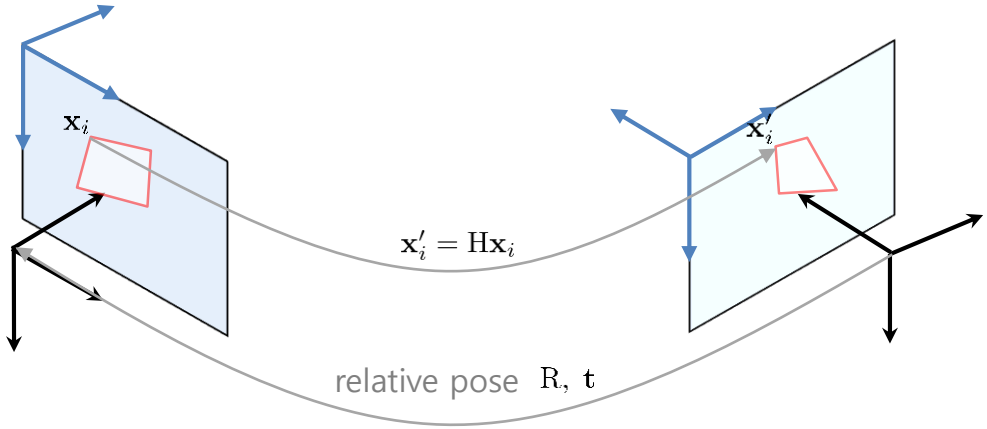
	**Euclidean Transform** (a.k.a. Rigid Transform) 	**Similarity Transform** 	**Affine Transform** 	**Projective Transform** (a.k.a. Planar Homography) 
Matrix Forms H	$\begin{bmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} s\cos\theta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

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

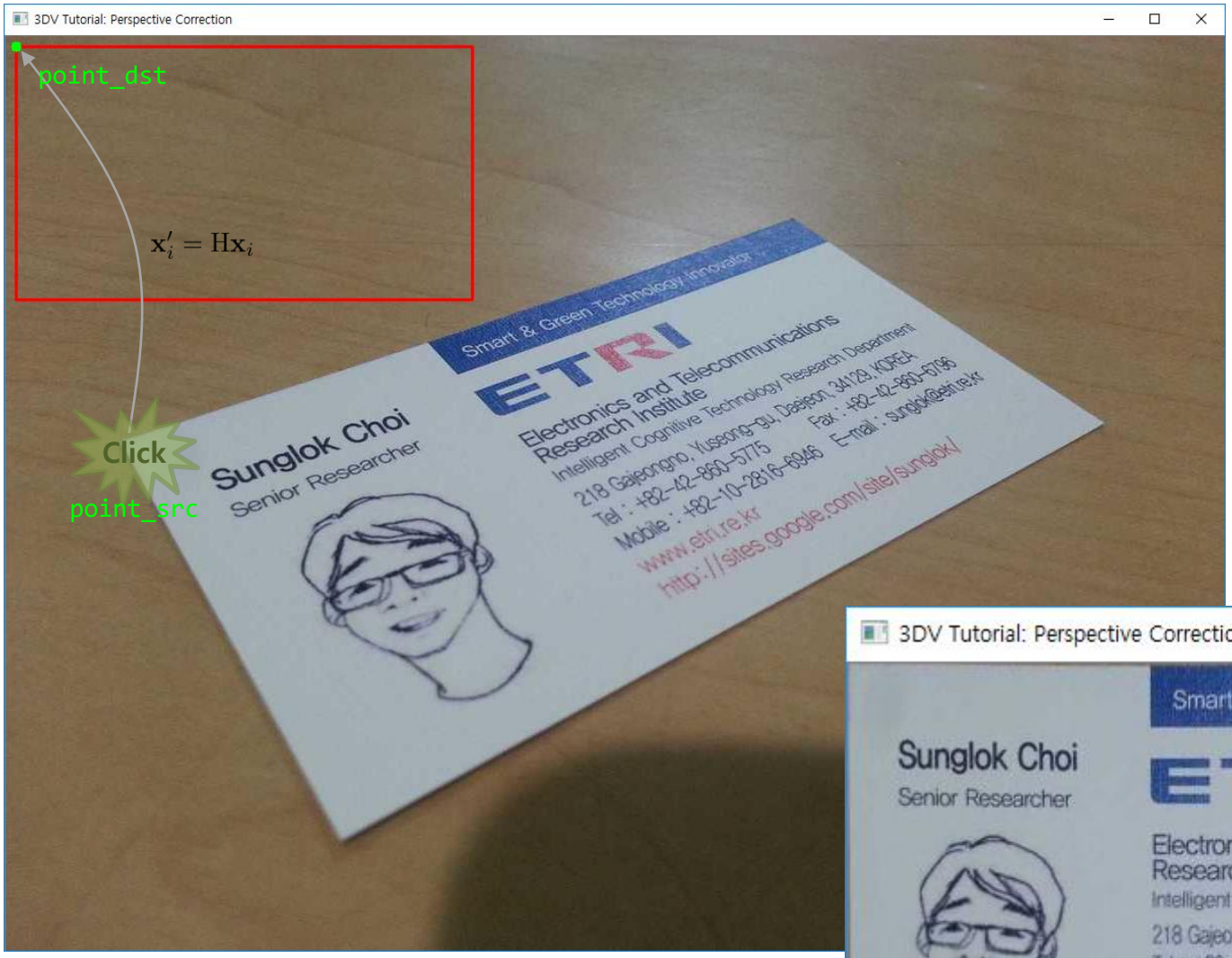
c.f. Similarly **3D transformations** (3D-3D geometry) are represented as **4x4 matrices**.

Planar 2D-2D Geometry (Projective Geometry)

- **Planar Homography Estimation**
  - Known: Point correspondence $(\mathbf{x}_1, \mathbf{x}'_1), \cdots, (\mathbf{x}_n, \mathbf{x}'_n)$
  - Unknown: Planar Homography (8 DoF)
  - Constraints: n x projective transformation $\mathbf{x}'_i = \mathrm{H}\mathbf{x}_i$
  - Solutions ($n \geq 4$)
    - OpenCV `cv::getPerspectiveTransform()` and `cv::findHomography()`
    - c.f. 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 R, $\mathbf{t}$.
    - OpenCV `cv::decomposeHomographyMat()`
    - c.f. However, the decomposition needs to know camera matrices.



Example: Perspective Distortion Correction



Example: Perspective Distortion Correction


```
1. #include "opencv_all.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(void)
10.{
11.    cv::Size card_size(450, 250);

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

18.    // Load an image
19.    cv::Mat original = cv::imread("data/sunglok.jpg");
20.    if (original.empty()) return -1;

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

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

Example: Planar Image Stitching

```
1. #include "opencv_all.hpp"

2. int main(void)
3. {
4.     // Load two images (c.f. 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.}
```

+

=

Example: 2D Video Stabilization


```
1. #include "opencv_all.hpp"

2. int main(void)
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.    }
```

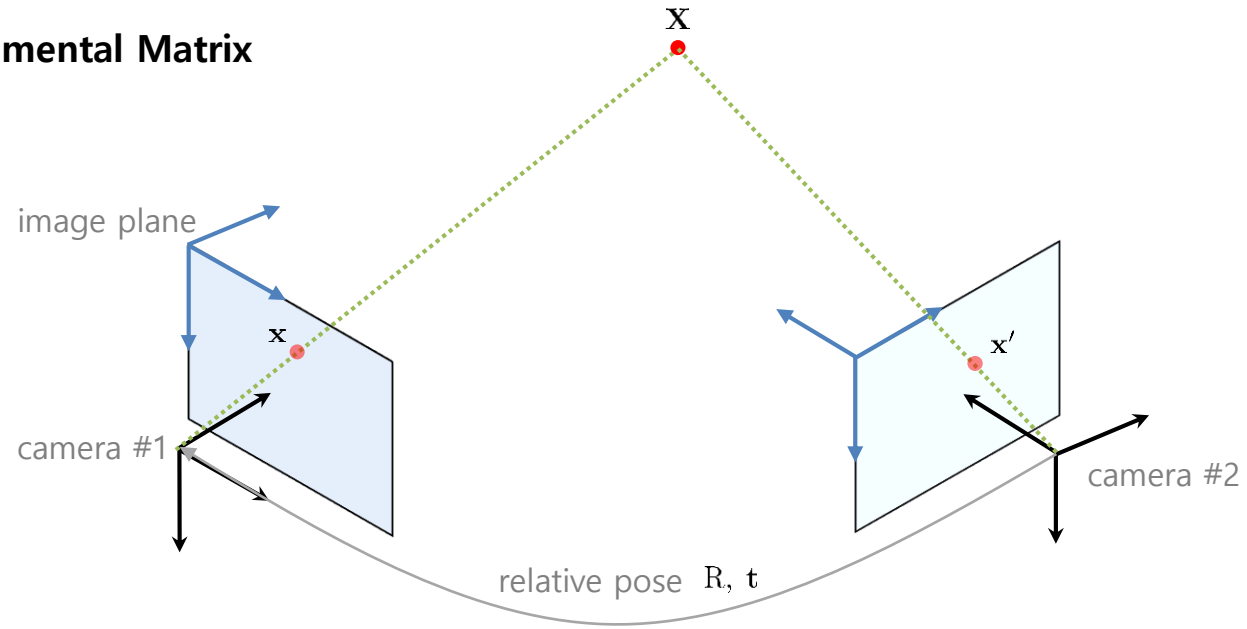
A shaking CCTV video: data/traffic.avi

Example: 2D Video Stabilization


```
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 (size_t 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::Scalar(0, 0, 255));
44.              else cv::line(image, point_ref[i], point[i], cv::Scalar(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.  }
```

General 2D-2D Geometry (Epipolar Geometry)

- **Fundamental Matrix**



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

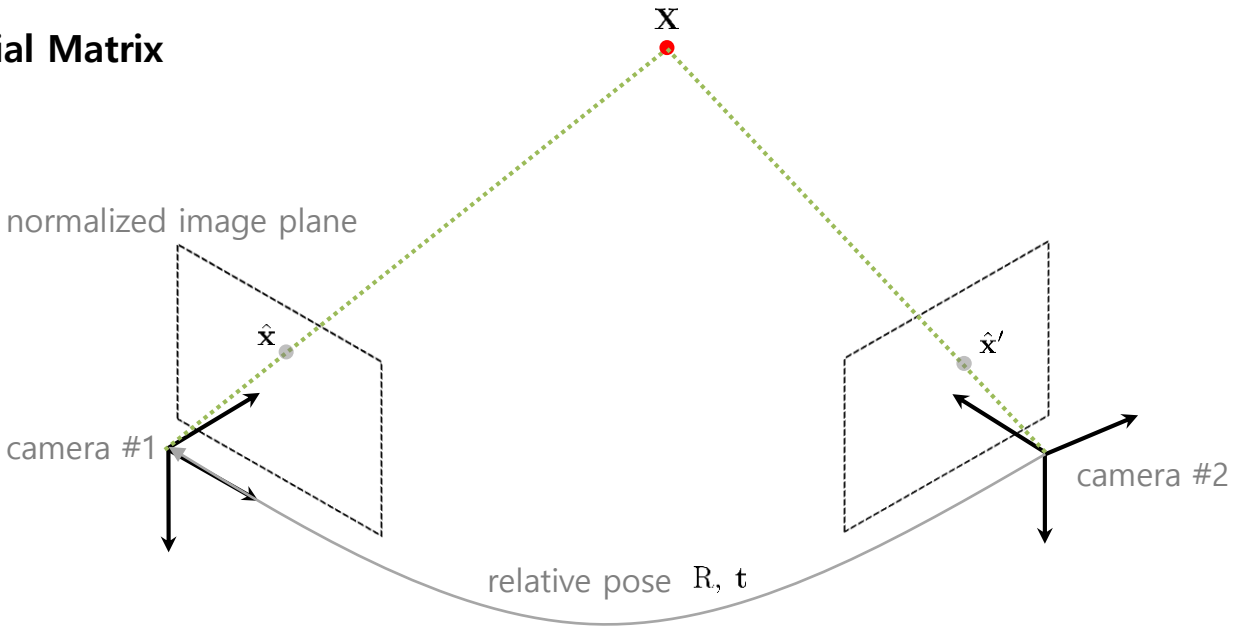
$\mathbf{x} = \mathrm{K}\hat{\mathbf{x}}$

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

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

General 2D-2D Geometry (Epipolar Geometry)

- **Essential Matrix**



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

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

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

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

$\hat{\mathbf{x}}'\cdot$

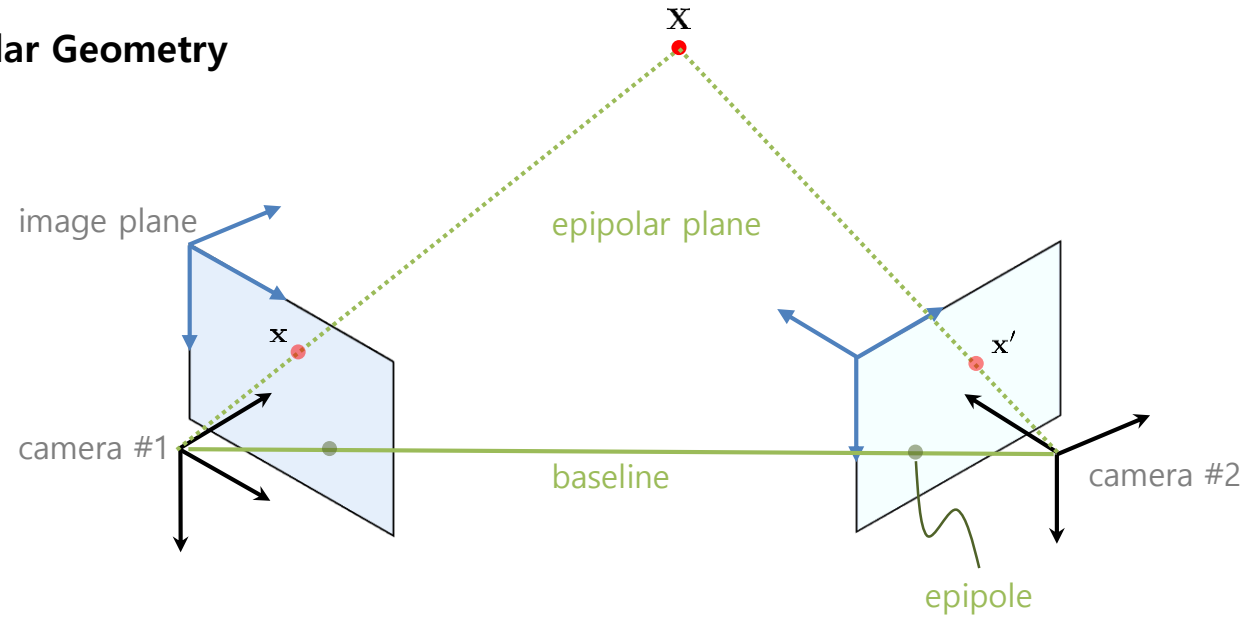
$\mathbf{t} \times \hat{\mathbf{x}}' = \mathbf{t} \times \mathrm{R}\hat{\mathbf{x}}$

$\mathbf{t}\times$

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

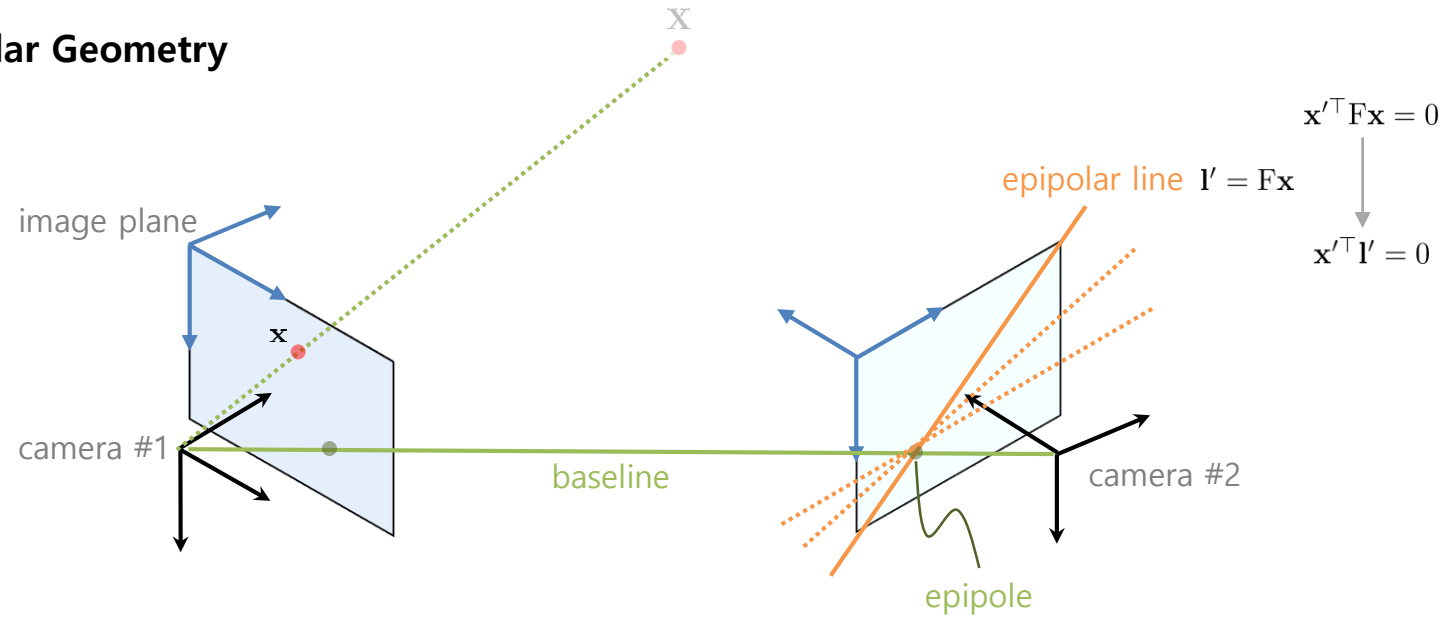
General 2D-2D Geometry (Epipolar Geometry)

- **Epipolar Geometry**



General 2D-2D Geometry (Epipolar Geometry)

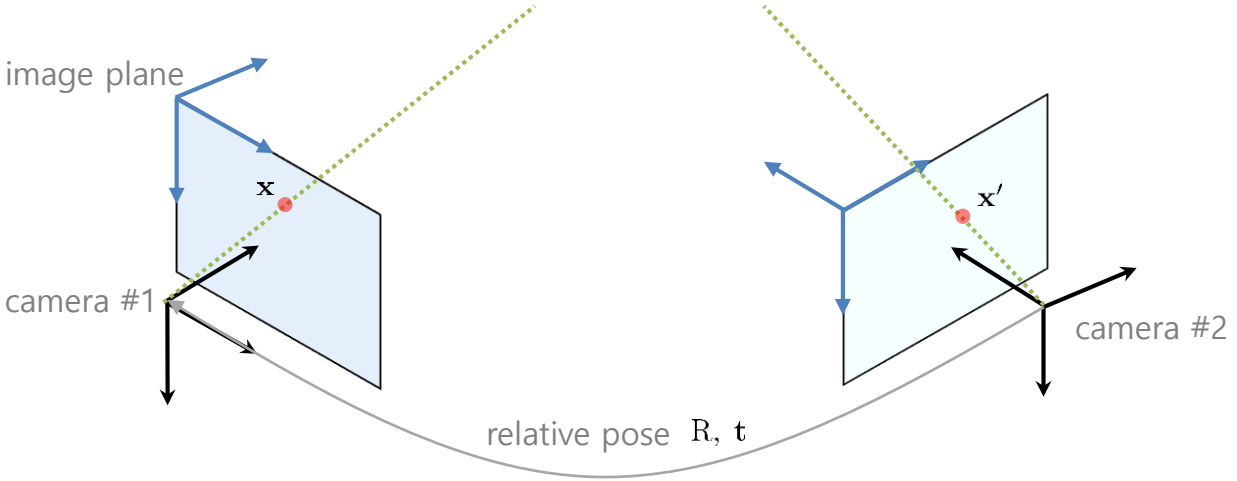
- **Epipolar Geometry**



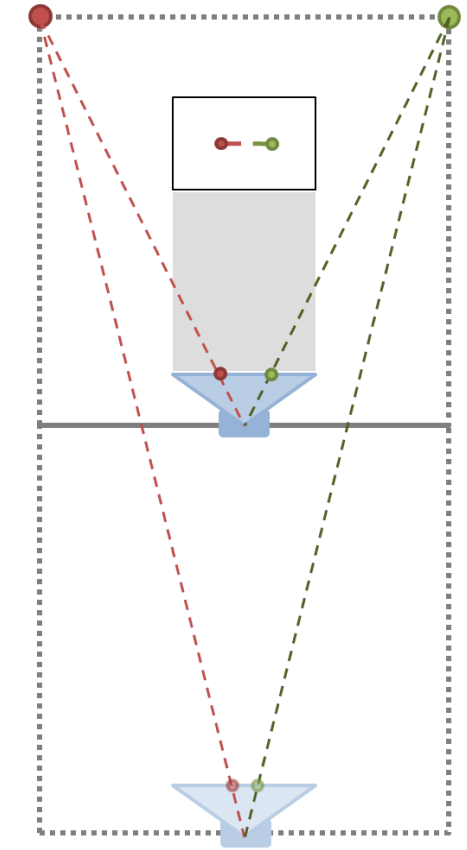
General 2D-2D Geometry (Epipolar Geometry)

- **Relative Camera Pose Estimation**
  - Known: Point correspondence $(\mathbf{x}_1, \mathbf{x}'_1), \cdots, (\mathbf{x}_n, \mathbf{x}'_n)$ and camera matrices K, K'
  - Unknown: Rotation and translation (5 DoF)
  - Constraints: n x epipolar constraint $\mathbf{x}'^{\top}\mathrm{F}\mathbf{x} = 0$
  - Solutions (OpenCV) — up-to scale → scale ambiguity
    - **Fundamental matrix**: 7/8-point algorithm
      - **Estimation**: `cv::findFundamentalMat()` → 1 solution
      - **Conversion to E**: $\mathrm{E} = \mathrm{K}'^{\top}\mathrm{F}\mathrm{K}$
    - **Essential matrix**: 5-point algorithm
      - **Estimation**: `cv::findEssentialMat()` → k solutions
      - **Decomposition**: `cv::decomposeEssentialMat()` → 4 solutions
      - **Decomposition with positive-depth check**: `cv::recoverPose()` → 1 solution

F
↓ $\mathrm{E} = \mathrm{K}'^{\top}\mathrm{F}\mathrm{K}$
E
↓ $\mathrm{E} = [\mathbf{t}]_{\times}\mathrm{R}$
$\mathrm{R},\ \mathbf{t}$

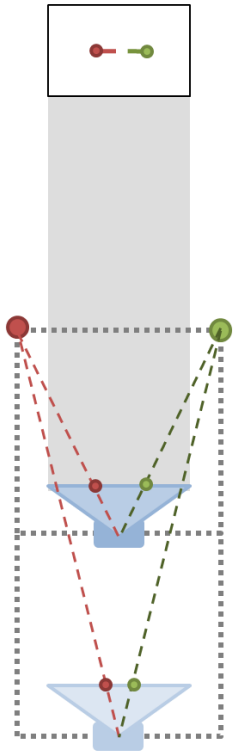


Epipolar Geometry: Scale Ambiguity



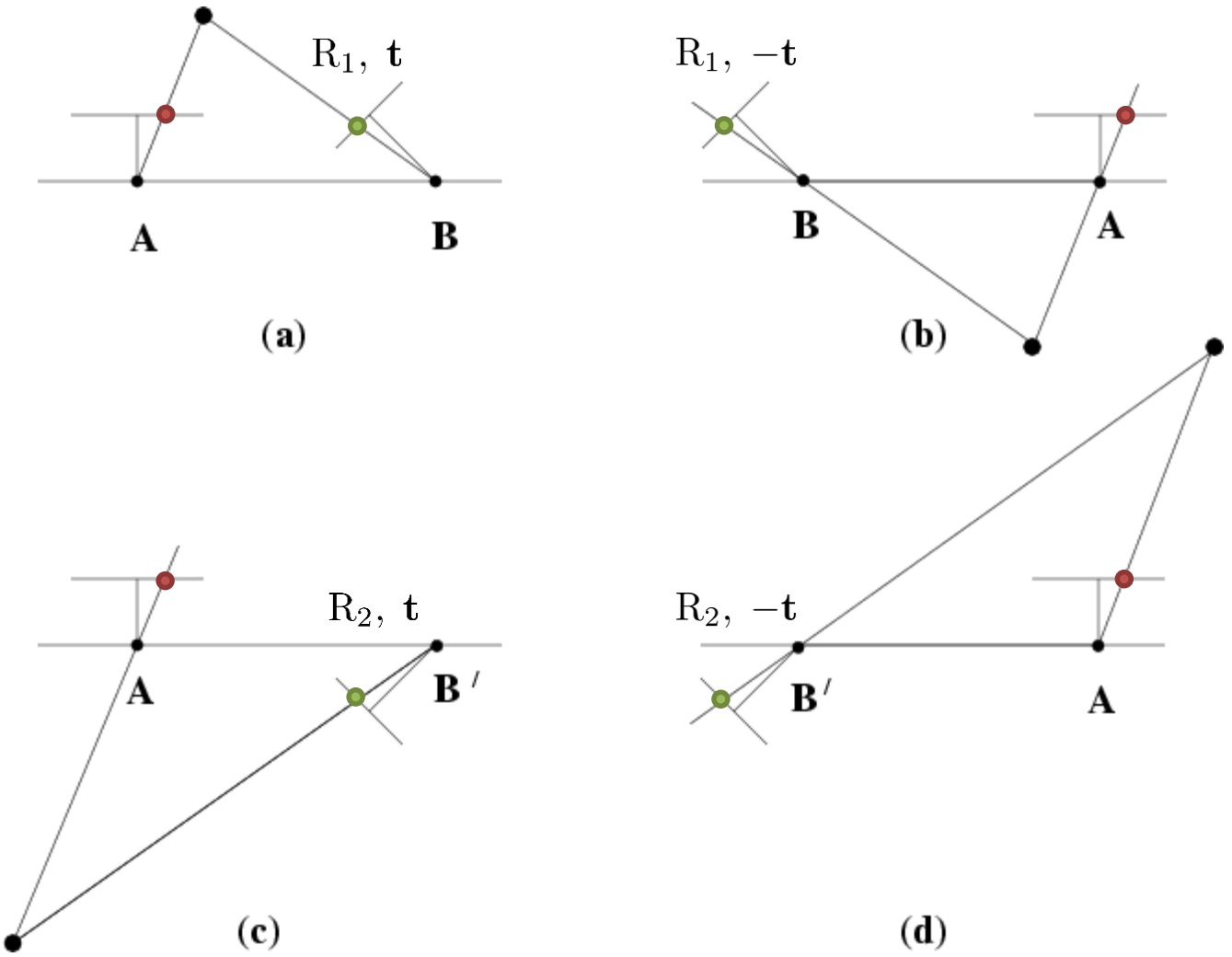
(a) 2-meter-wide tunnel

optical flow on an image



(b) 1-meter-wide tunnel

Epipolar Geometry: Relative Pose Ambiguity



Images are from *Multiple View Geometry in Computer Vision* (by Hartley and Zisserman).

General 2D-2D Geometry (Epipolar Geometry)

- **Relative Camera Pose Estimation**
  - Known: Point correspondence $(\mathbf{x}_1, \mathbf{x}'_1), \cdots, (\mathbf{x}_n, \mathbf{x}'_n)$ and camera matrices K, K'
  - Unknown: Rotation and translation (5 DoF)
  - Constraints: n x epipolar constraint $\mathbf{x}'^{\top}\mathrm{F}\mathbf{x} = 0$
  - 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**: $\mathrm{E} = \mathrm{K}'^{\top}\mathrm{F}\mathrm{K}$
  - **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**
  - Known: Point correspondence $(\mathbf{x}_1, \mathbf{x}'_1), \cdots, (\mathbf{x}_n, \mathbf{x}'_n)$ and camera matrices K, K'
  - Unknown: Rotation and translation (5 DoF)
  - Constraints: n x epipolar constraint $\mathbf{x}'^{\top}\mathrm{F}\mathbf{x} = 0$
  - 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**: $\mathrm{E} = \mathrm{K}'^{\top}\mathrm{F}\mathrm{K}$
  - **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

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

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

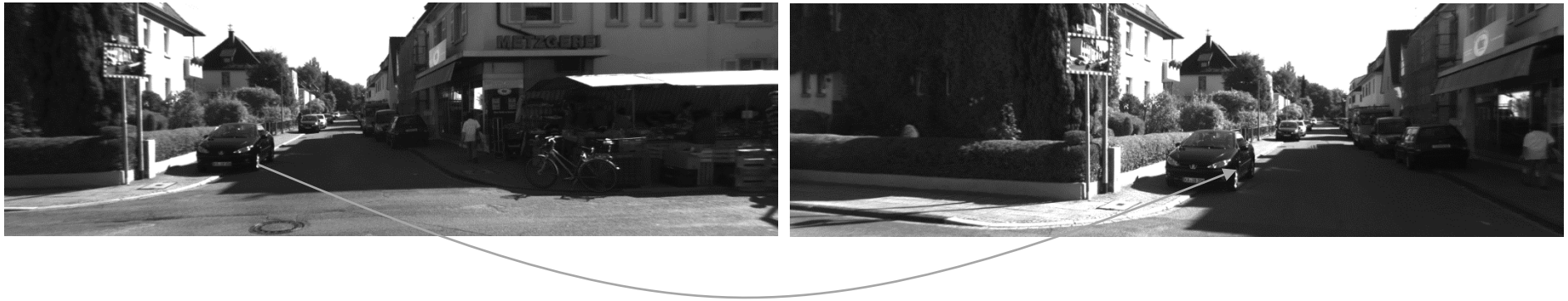
$$\frac{1}{d}\mathbf{n}^{\top}\hat{\mathbf{x}} = 1 \quad (\because n_x\hat{x} + n_y\hat{y} + n_z\hat{z} - d = 0)$$

Overview of Relative Pose Estimation

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

Example: Monocular Visual Odometry (Epipolar)

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()`

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

Pass tentative matches

Pass the essential matrix E

[Camera 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, $\mathbf{t}$

Example: Monocular Visual Odometry (Epipolar)


```
1.  #include "opencv_all.hpp"

2.  int main(void)
3.  {
4.      bool use_5pt = false;
5.      double camera_focal = 718.8560;
6.      cv::Point2d camera_center(607.1928, 185.2157);

7.      // Open a file to write camera trajectory
8.      FILE* camera_traj = fopen("visual_odometry_epipolar.xyz", "wt");
9.      if (camera_traj == NULL) return -1;

10.     // Open a video and get the initial image
11.     cv::VideoCapture video;
12.     if (!video.open("data/KITTI_00_L/%06d.png")) return -1;

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

16.     // Run and store 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.         . . .

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

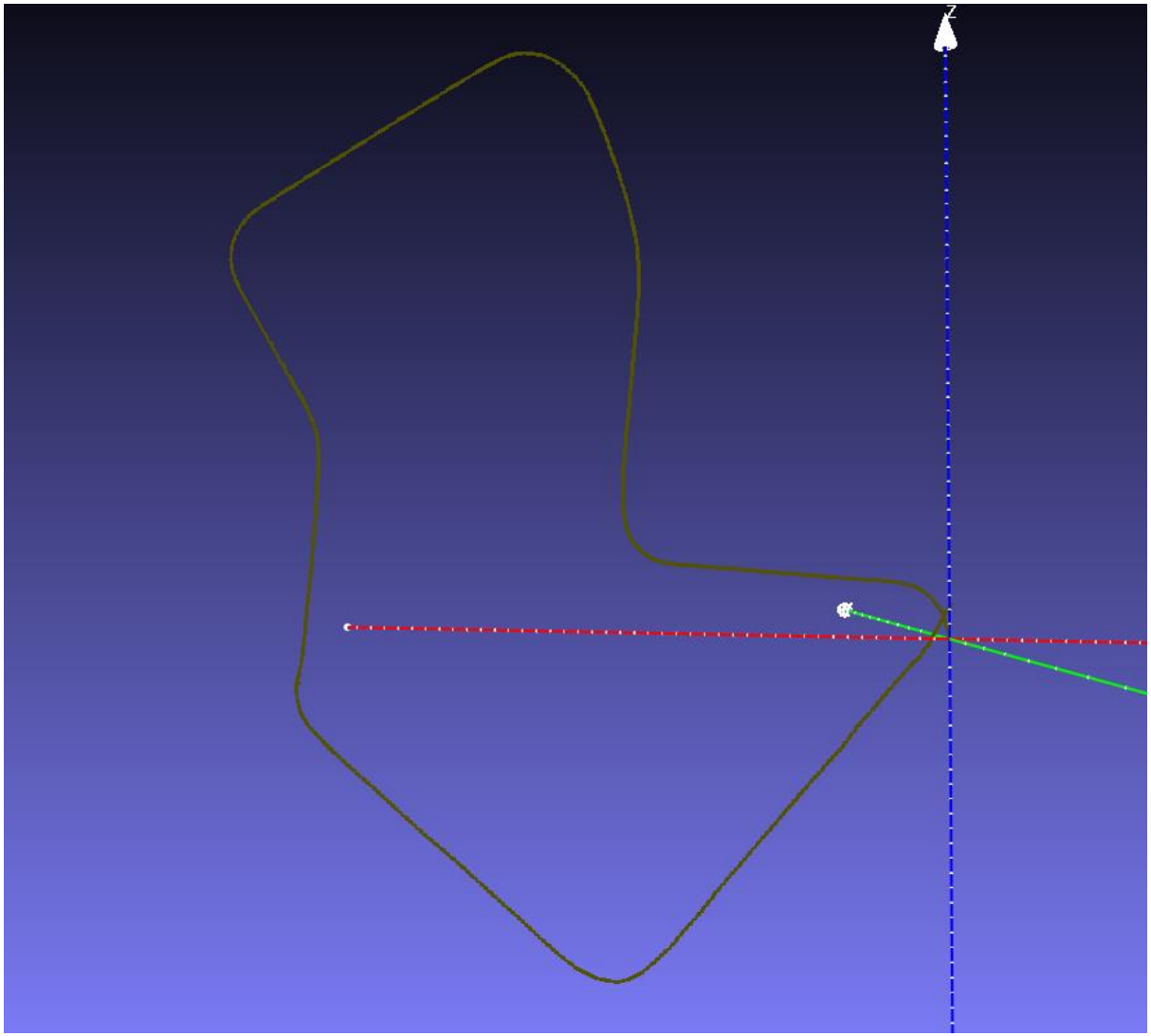
ETRI

Example: Monocular Visual Odometry (Epipolar)

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

45.         // Accumulate pose
46.         cv::Mat T = cv::Mat::eye(4, 4, R.type());
47.         T(cv::Rect(0, 0, 3, 3)) = R * 1.0;
48.         T.col(3).rowRange(0, 3) = t * 1.0;
49.         camera_pose = camera_pose * T.inv();

50.         // Show the image and write camera pose
51.         if (image.channels() < 3) cv::cvtColor(image, image, CV_GRAY2RGB);
52.         for (size_t i = 0; i < point_prev.size(); i++)
53.         {
54.             if (inlier_mask.at<uchar>(i) > 0) cv::line(image, point_prev[i], point[i], cv::Scalar(0, 0, 255));
55.             else cv::line(image, point_prev[i], point[i], cv::Scalar(0, 127, 0));
56.         }
57.         cv::imshow("3DVT Tutorial: Monocular Visual Odometry", image);
58.         fprintf(camera_traj, "%.6f %.6f %.6f\n", camera_pose.at<double>(0, 3), camera_pose.at<double>(1, 3),
                                                     camera_pose.at<double>(2, 3));
59.         if (cv::waitKey(1) == 27) break; // 'ESC' key
60.     }

61.     video.release();
62.     fclose(camera_traj);
63.     return 0;
64. }
```

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

Example: Monocular Visual Odometry (Epipolar)



Result: visual_odometry_epipolar.xyz

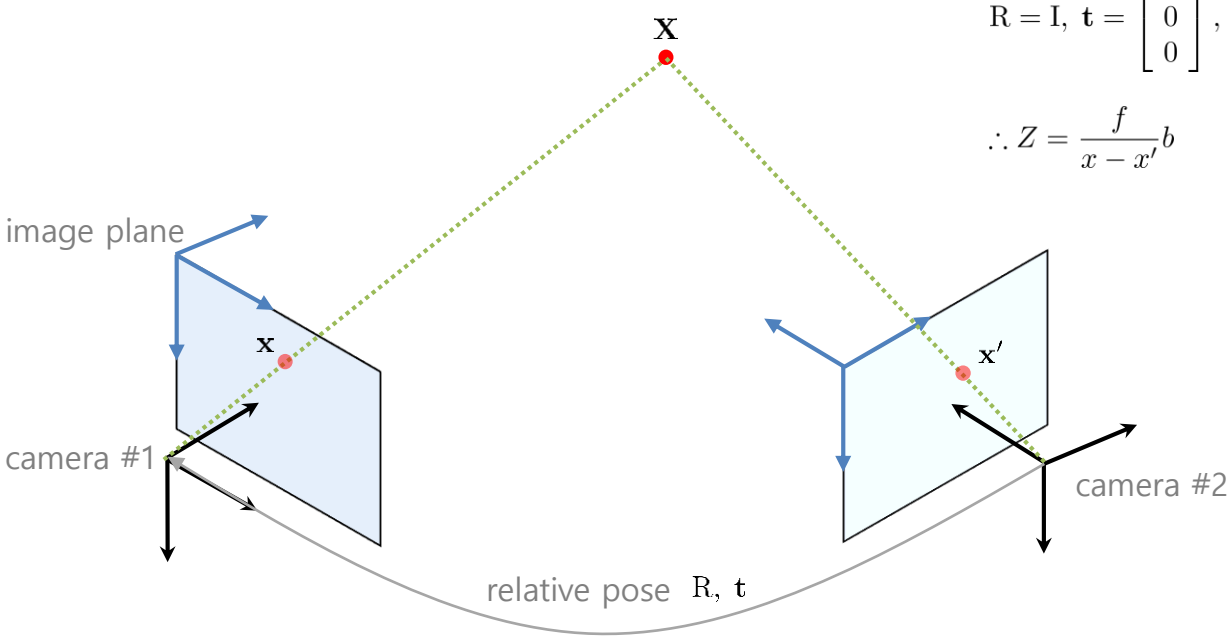
General 2D-2D Geometry (Epipolar Geometry)

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

Special case) Stereo cameras

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

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



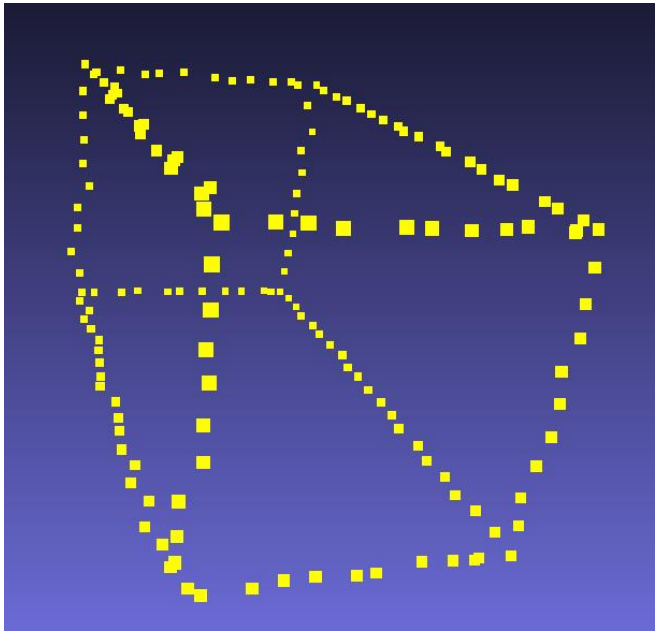
Example: Triangulation (Two-view Reconstruction)

```
1.  #include "opencv_all.hpp"

2.  int main(void)
3.  {
4.      double camera_focal = 1000;
5.      cv::Point2d camera_center(320, 240);

6.      // Load two views of 'box.xyz'
7.      std::vector<cv::Point2d> points0, points1;
8.      FILE* fin0 = fopen("image_formation0.xyz", "rt");
9.      FILE* fin1 = fopen("image_formation1.xyz", "rt");
10.     ...

11.     // Esitmate relative pose of two views
12.     cv::Mat F = cv::findFundamentalMat(points0, points1, cv::FM_8POINT);
13.     cv::Mat K = (cv::Mat_<double>(3, 3) << camera_focal, ...);
14.     cv::Mat E = K.t() * F * K;
15.     cv::Mat R, t;
16.     cv::recoverPose(E, points0, points1, K, R, t);

17.     // Reconstruct 3D points of 'box.xyz' (triangulation)
18.     cv::Mat P0 = K * cv::Mat::eye(3, 4, CV_64F);
19.     cv::Mat Rt, X;
20.     cv::hconcat(R, t, Rt);
21.     cv::Mat P1 = K * Rt;
22.     cv::triangulatePoints(P0, P1, points0, points1, X);
23.     X.row(0) = X.row(0) / X.row(3);
24.     X.row(1) = X.row(1) / X.row(3);
25.     X.row(2) = X.row(2) / X.row(3);
26.     X.row(3) = 1;

27.     // Store the 3D points
28.     FILE* fout = fopen("triangulation.xyz", "wt");
29.     if (fout == NULL) return -1;
30.     for (int c = 0; c < X.cols; c++)
31.         fprintf(fout, "%f %f %f\n", X.at<double>(0, c), X.at<double>(1, c), X.at<double>(2, c));
32.     fclose(fout);
33.     return 0;
34. }
```

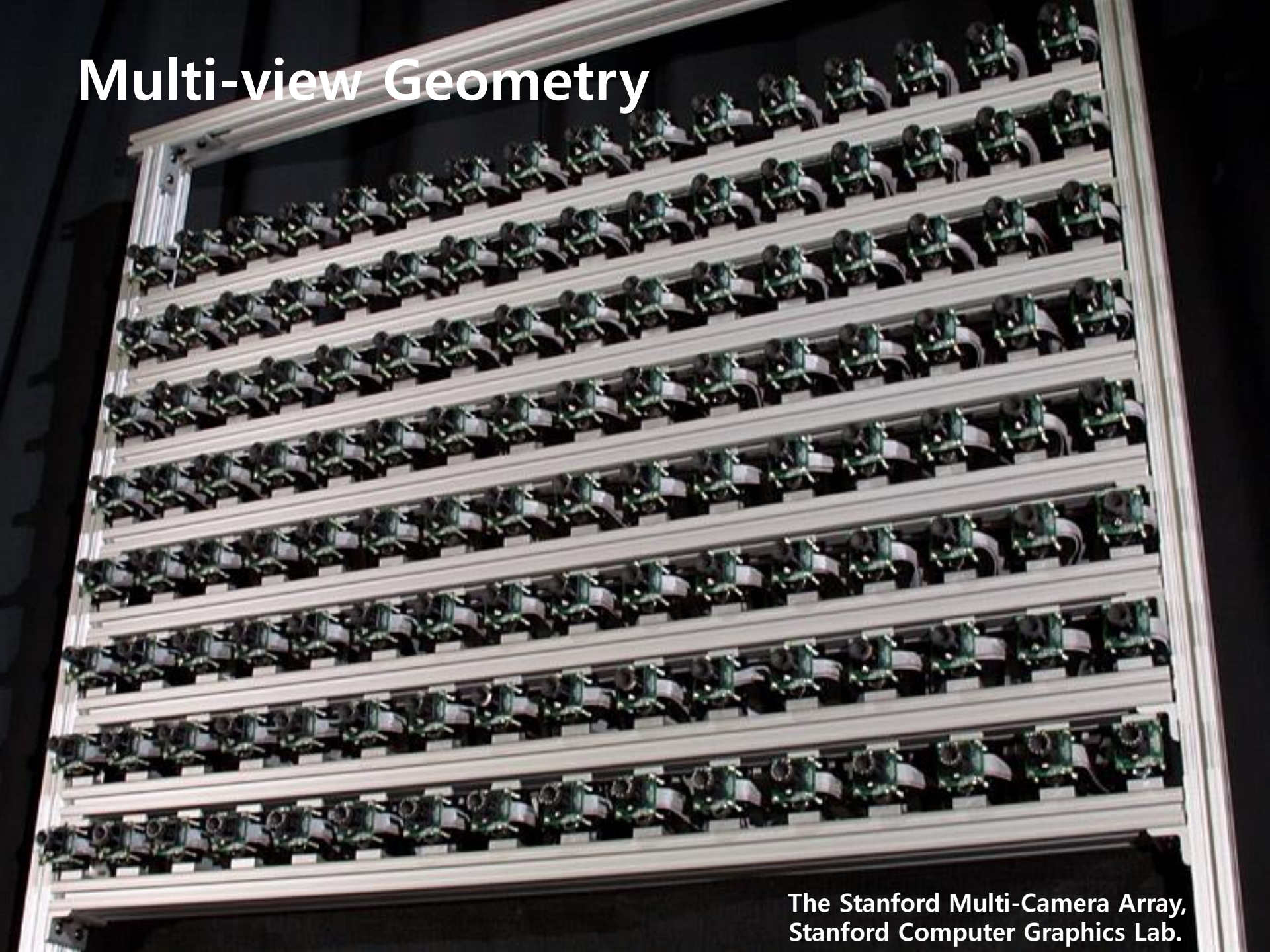
$\because \mathbf{x} = \mathrm{K}\left[\mathrm{I} \mid \mathbf{0}\right]\mathbf{X}$

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

Result: triangulation.xyz

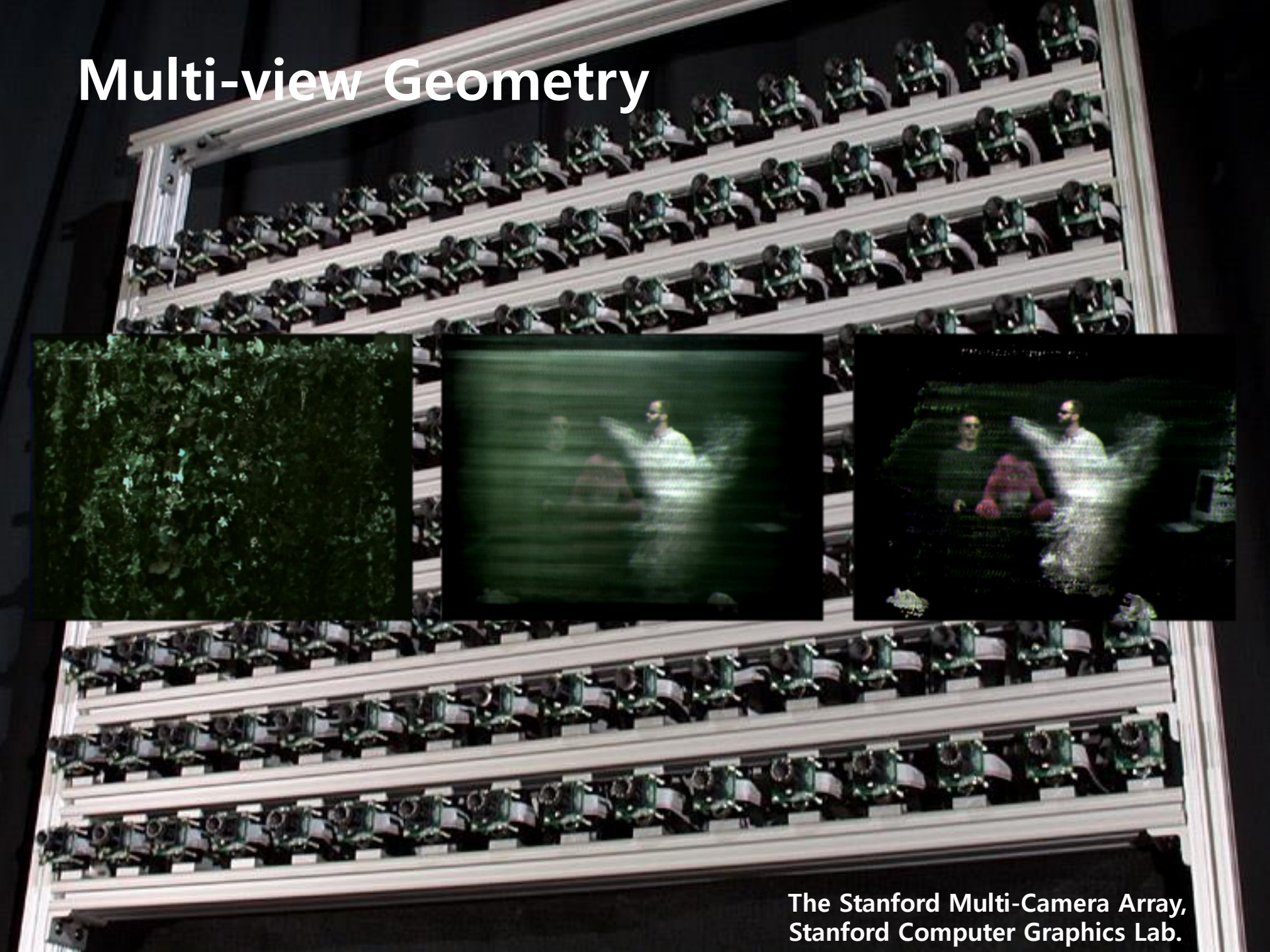


Multi-view Geometry



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

Multi-view Geometry

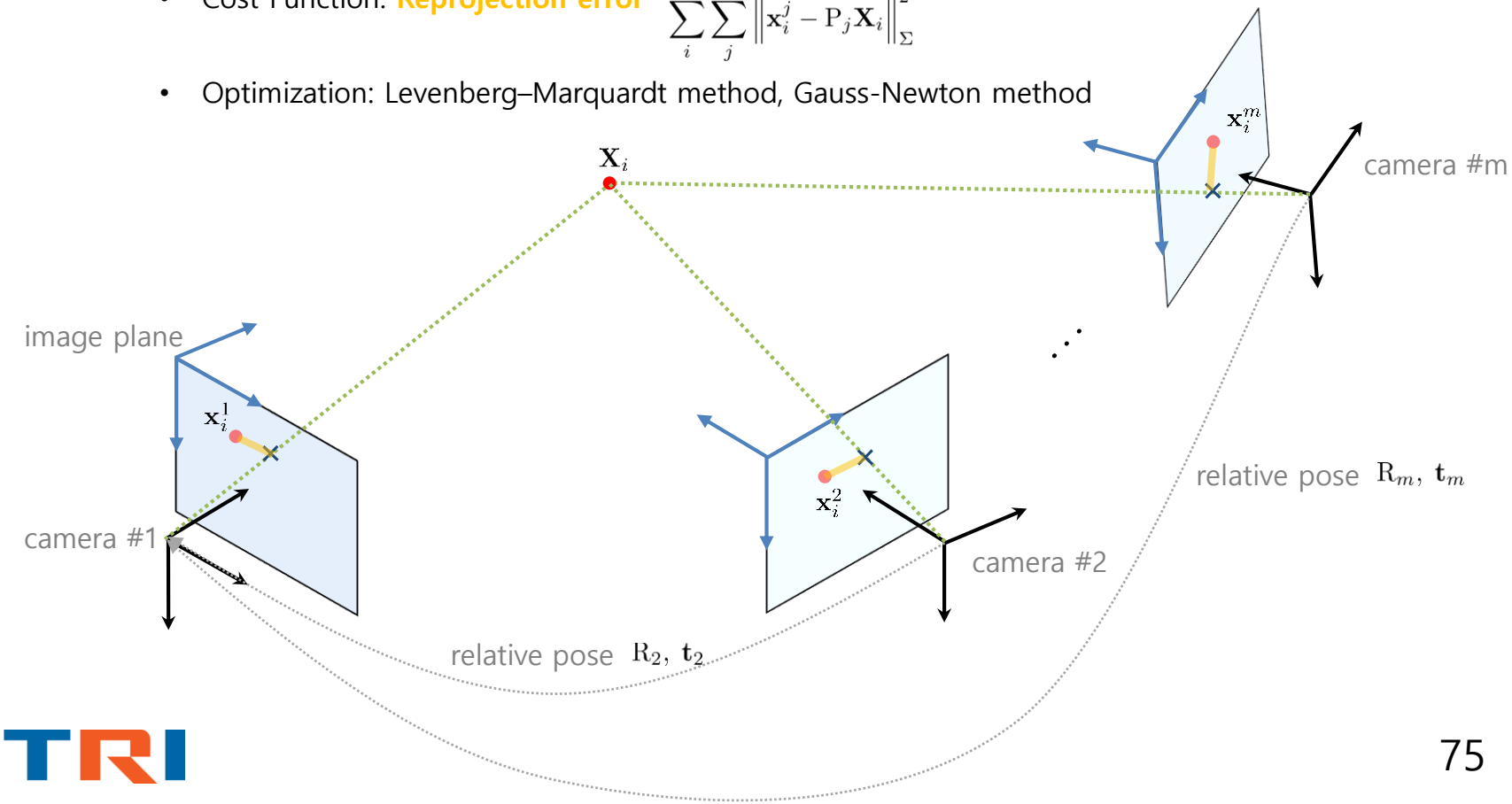


The Stanford Multi-Camera Array,
Stanford Computer Graphics Lab.

Bundle Adjustment

- **Bundle Adjustment**
  - Known: Point correspondence, camera matrices, position of 3D points, and each camera's relative pose
  - Unknown: Position of 3D points and each camera's relative pose ($6n + 3m$ DoF)
  - Constraints: n x m x projection $\mathbf{x}_i^j = \mathrm{P}_j \mathbf{X}_i = \mathrm{K}_j \left[\mathrm{R}_j \,|\, \mathbf{t}_j \right] \mathbf{X}_i$
  - 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

c.f. initial values



Bundle Adjustment

- **Bundle Adjustment**

  c.f. initial values

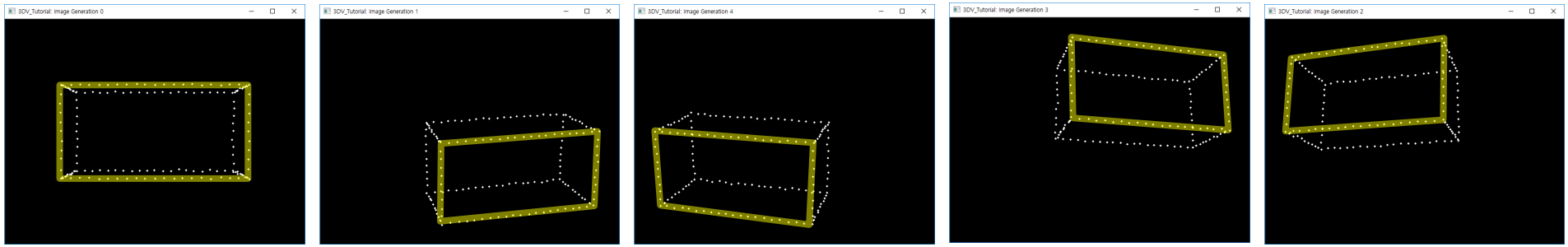
  - Known: Point correspondence, camera matrices, position of 3D points, and each camera's relative pose
  - Unknown: Position of 3D points and each camera's relative pose ($6n + 3m$ DoF)
  - Constraints: n x m x projection $\mathbf{x}_i^j = \mathrm{P}_j \mathbf{X}_i = \mathrm{K}_j \left[\mathrm{R}_j \,|\, \mathbf{t}_j \right] \mathbf{X}_i$
  - Solution: **Non-linear least-square optimization**
    - Cost Function: **Reprojection error** $\sum_{i}^{n} \sum_{j}^{m} v_i^j \left\| \mathbf{x}_i^j - \mathrm{P}_j \mathbf{X}_i \right\|_{\Sigma}^2 \quad \text{where} \quad v_i^j = \begin{cases} 1 & \text{if } \mathbf{x}_i^j \text{ is visible} \\ 0 & \text{otherwise} \end{cases}$
    - 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)
  - Optimization: g2o (General Graph Optimization), iSAM (Incremental Smoothing and Mapping), GTSAM, Ceres Solver (Google)

Example: Bundle Adjustment (Global)

[Known]

- **Intrinsic parameters**: camera matrices K_j (and also distortion coefficients)
- **2D point correspondence** $\mathbf{x}_i^j$



[Unknown]

- **Extrinsic parameters**: camera poses

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

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

cvsba (non-linear least-square optimization)

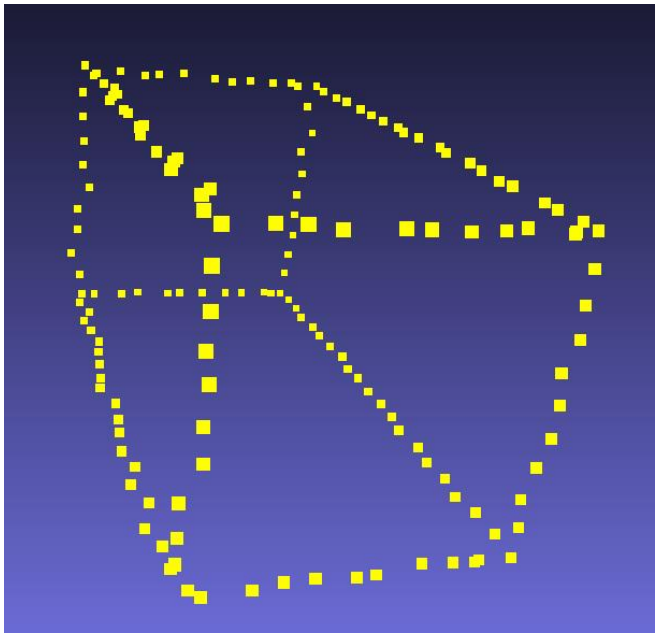
$$\sum_i^n \sum_j^m v_i^j \left\| \mathbf{x}_i^j - \mathrm{P}_j \mathbf{X}_i \right\|_\Sigma^2 \quad \text{where} \quad v_i^j = 1 \ (\because \text{all points are visible})$$

Example: Bundle Adjustment (Global)

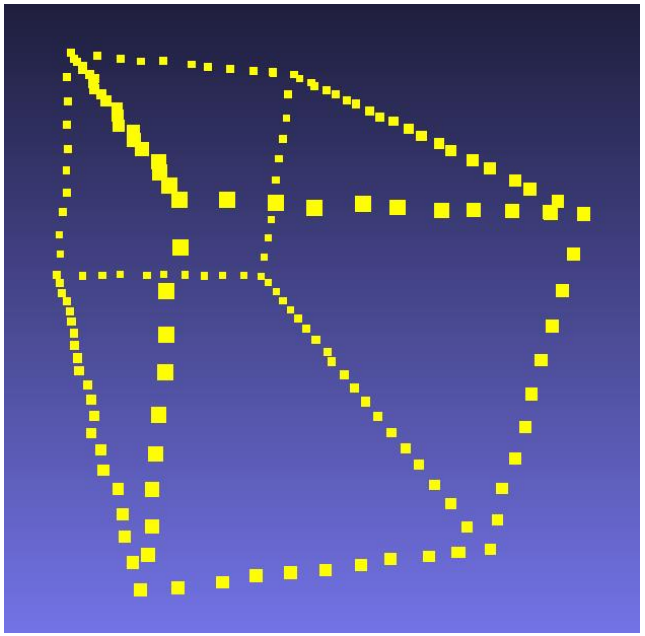

```
1.  #include "opencv_all.hpp"
2.  #include "cvsba.h"

3.  int main(void)
4.  {
5.      // Load multiple views of 'box.xyz'
6.      std::vector<std::vector<cv::Point2d> > xs;
7.      ...

8.      // Assume that all cameras have the same and known camera matrix
9.      // Assume that all feature points are visible on all views
10.     cv::Mat K = (cv::Mat_<double>(3, 3) << camera_focal, 0, camera_center.x, 0, camera_focal, camera_center.y, 0, 0, 1);
11.     cv::Mat dist_coeff = cv::Mat::zeros(5, 1, CV_64F);
12.     std::vector<int> visible_all(xs.front().size(), 1);

13.     // Initialize each camera projection matrix
14.     std::vector<cv::Mat> Ks, dist_coeffs, Rs, ts;
15.     std::vector<std::vector<int> > visibility;
16.     for (int i = 0; i < n_views; i++)
17.     {
18.         visibility.push_back(visible_all);
19.         Ks.push_back(K.clone());                                  // K for all cameras
20.         dist_coeffs.push_back(cv::Mat::zeros(5, 1, CV_64F));     // dist_coeff for all cameras
21.         Rs.push_back(cv::Mat::eye(3, 3, CV_64F));                // R for all cameras
22.         ts.push_back(cv::Mat::zeros(3, 1, CV_64F));              // t for all cameras
23.     }

24.     // Initialize 3D points
25.     std::vector<cv::Point3d> Xs;
26.     Xs.resize(xs.front().size(), cv::Point3d(0, 0, 5.5));

27.     // Optimize camera pose and 3D points
28.     try
29.     {
30.         cvsba::Sba sba;
31.         cvsba::Sba::Params param;
32.         param.type = cvsba::Sba::MOTIONSTRUCTURE;
33.         ...
34.         double error = sba.run(Xs, xs, visibility, Ks, Rs, ts, dist_coeffs);
35.     }
```

Example: Bundle Adjustment (Global)

Result: triangulation.xyz



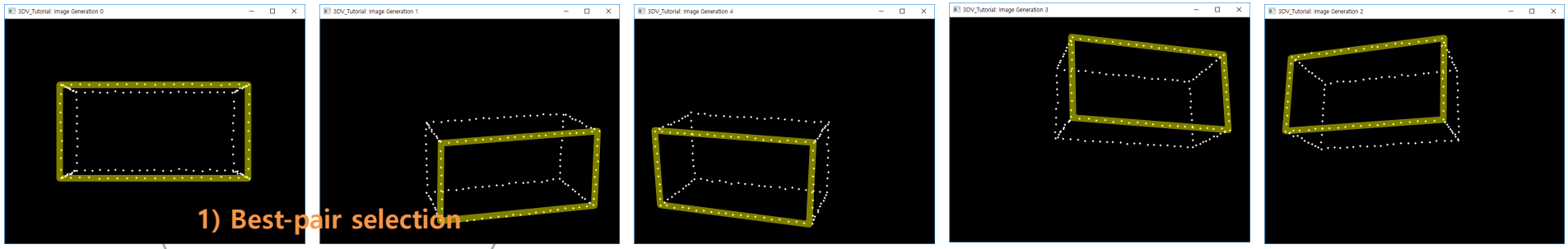
Result: bundle_adjustment_global.xyz



Example: Bundle Adjustment (Incremental)

[Known]

- **Intrinsic parameters**: camera matrices K_j (and also distortion coefficients)
- **2D point correspondence** $\mathbf{x}_i^j$



[Unknown]

- **Extrinsic parameters**: camera poses

2) Origin

$\mathrm{R}_0 = \mathrm{I}_{3x3}$

$\mathbf{t}_0 = [0, 0, 0]^\top$

$\mathrm{R}_1,\ \mathbf{t}_1$ $\mathrm{R}_2,\ \mathbf{t}_2$ $\mathrm{R}_3,\ \mathbf{t}_3$ $\mathrm{R}_4,\ \mathbf{t}_4$

3) Epipolar geometry

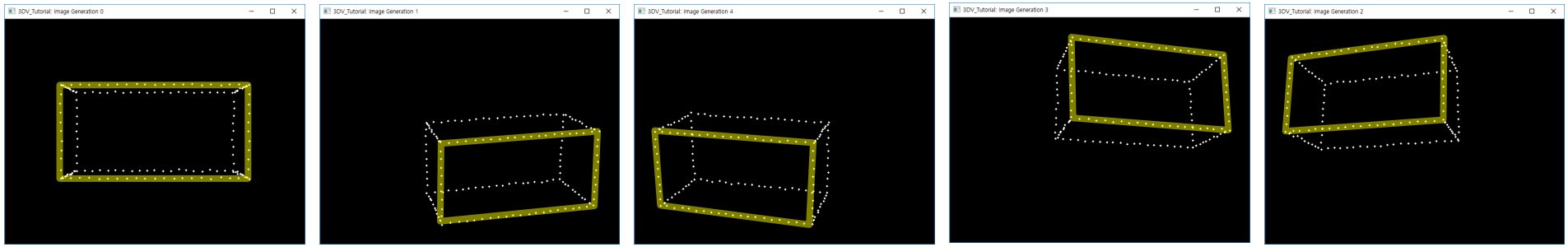
4) Triangulation

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

Example: Bundle Adjustment (Incremental)

[Known]

- **Intrinsic parameters**: camera matrices K_j (and also distortion coefficients)
- **2D point correspondence** $\mathbf{x}_i^j$



[Unknown]

- **Extrinsic parameters**: camera poses

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

5) Next-view selection

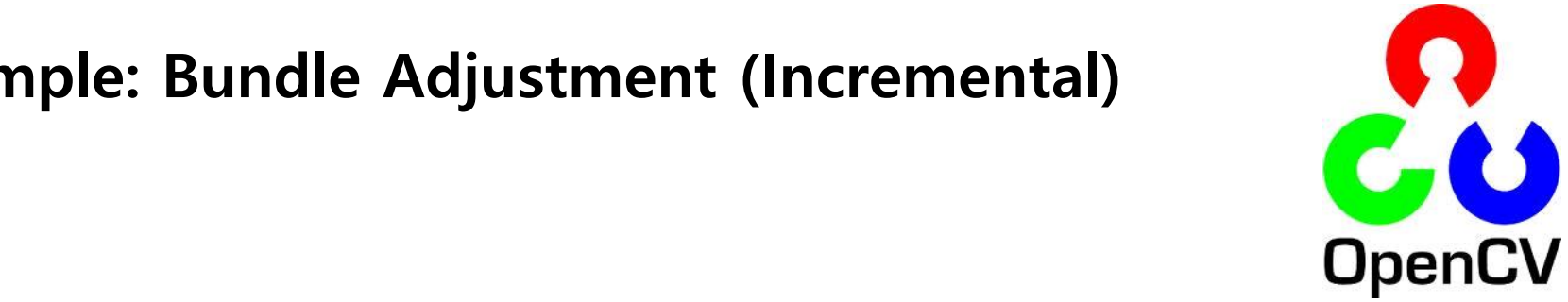
Repeat 5 – 8 until the last image

6) Perspective-n-point (PnP)

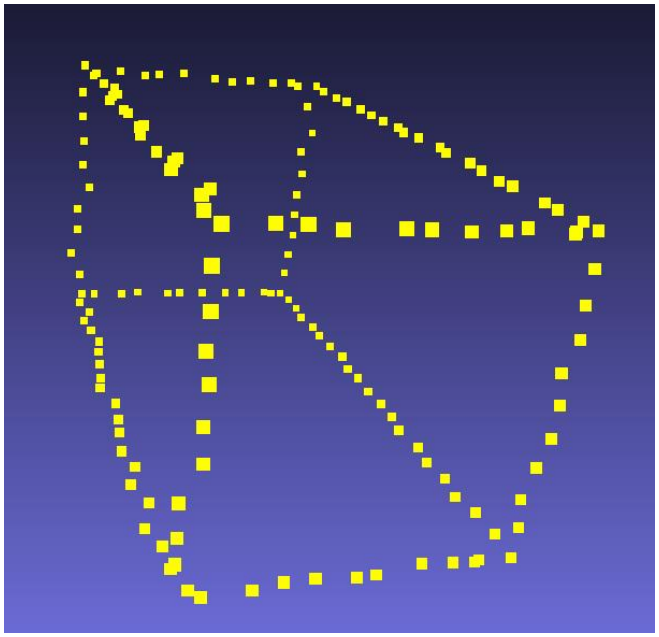
- **3D points** $\mathbf{X}_i$

7) Triangulation + 8) Bundle Adjustment

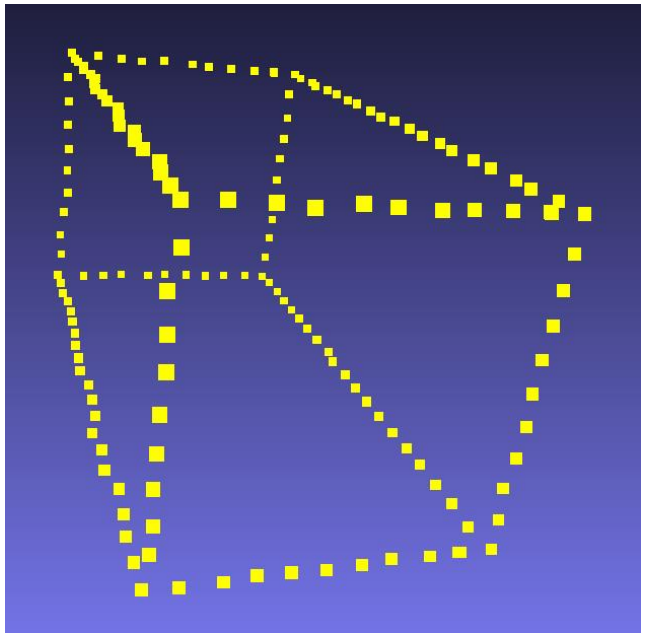
Example: Bundle Adjustment (Incremental)



Result: triangulation.xyz



Result: bundle_adjustment_global/_inc.xyz

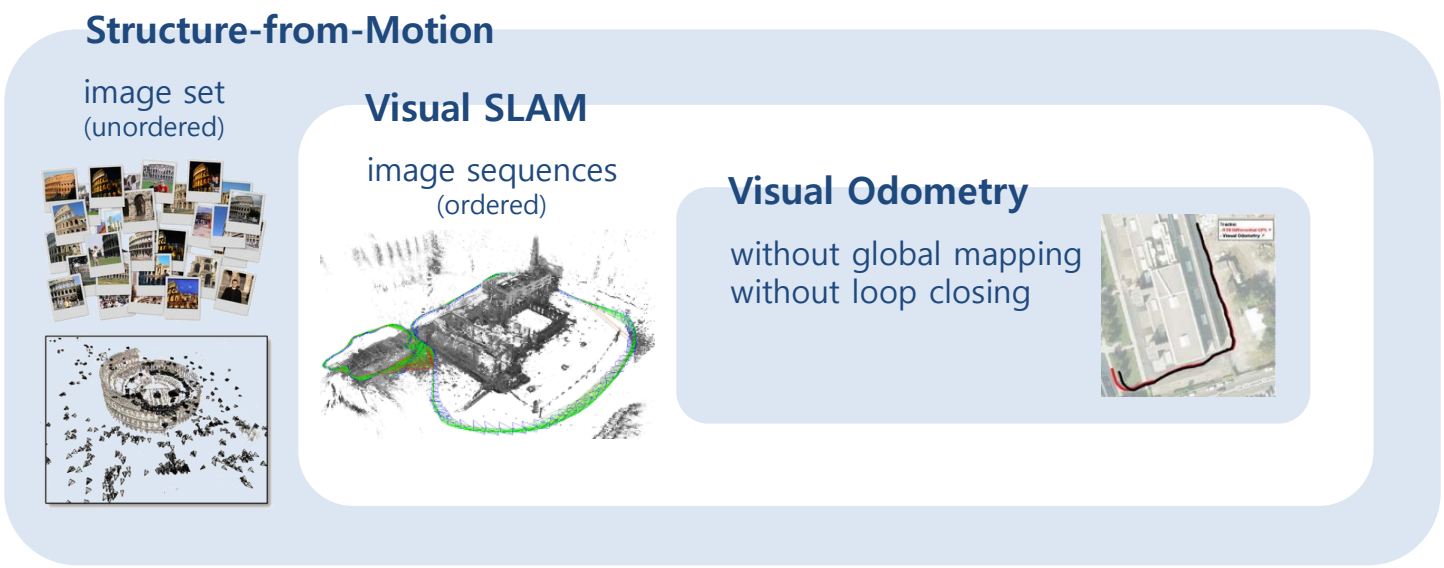


[bundle_adjusetment_global.xyz]
- # of iterations: **12**
- Reprojection error: 1.409 ← **32970.3**

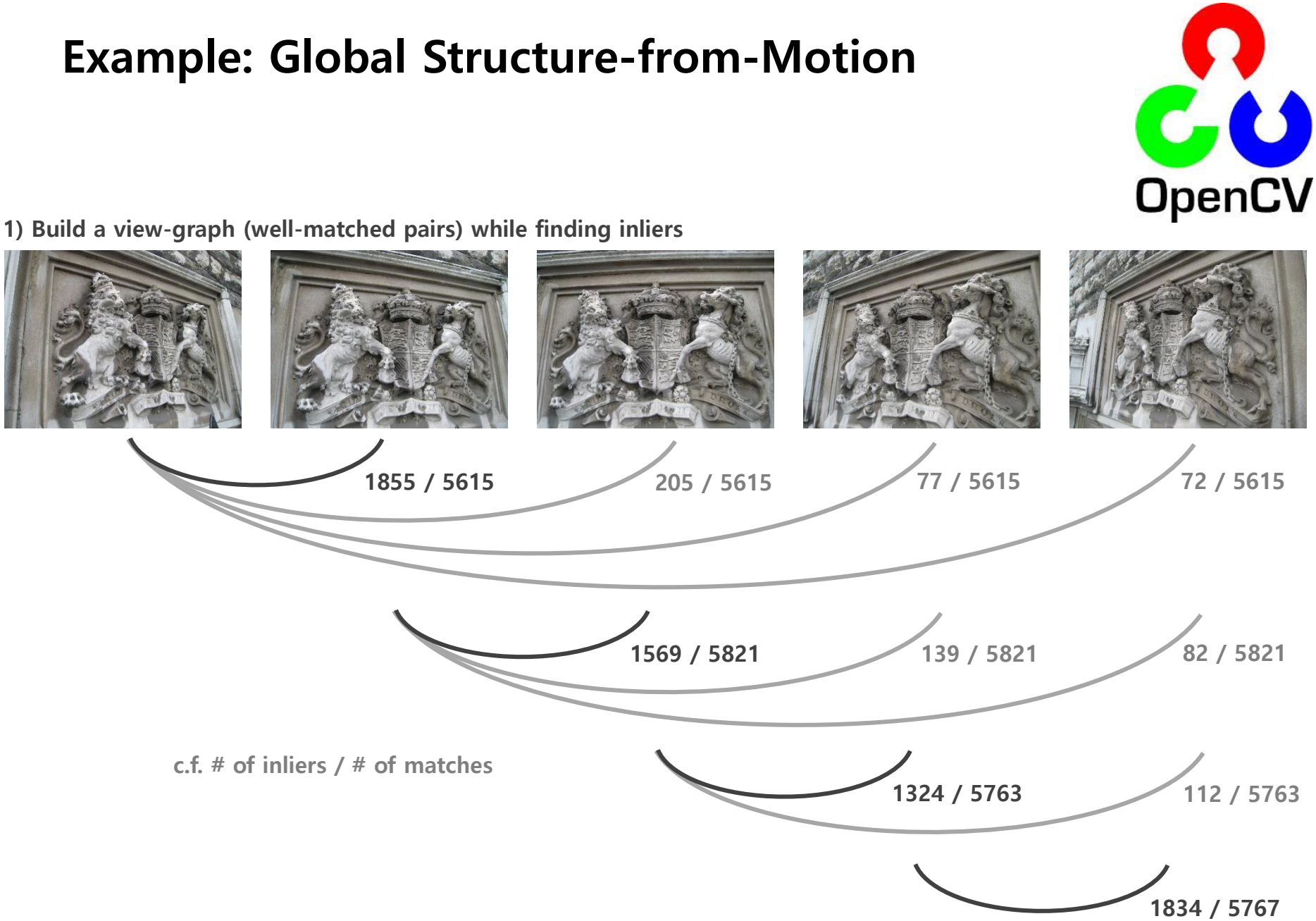
[bundle_adjusetment_inc.xyz]
- # of iterations: **10 + 8 + 8**
- Reprojection error: 1.409 ← **3.604**

Applications

- **Structure-from-Motion (SfM)** → 3D Reconstruction, Photo Browsing
  - Bundler, COLMAP, CMPMVS, OpenSfM, Toy SfM / VisualSFM (GUI, binary only)
    - Photo Toursim, PhotoSynth, PhotoCity, Building Rome in a Day, Building Rome on a Cloudless Day
- **Visual SLAM** → Augmented Reality, Navigation (Mapping and Localization)
  - PTAM (Parallel Tracking and Mapping), ORB-SLAM / DTAM (Dense Tracking and Mapping), LSD-SLAM
    - ORB-SLAM for Windows
- **Visual Odometry** → Navigation (Localization)
  - LIBVISO2 (C++ Library for Visual Odometry 2) / SVO (Semi-direct Monocular Visual Odometry)



Example: Global Structure-from-Motion



Example: Global Structure-from-Motion

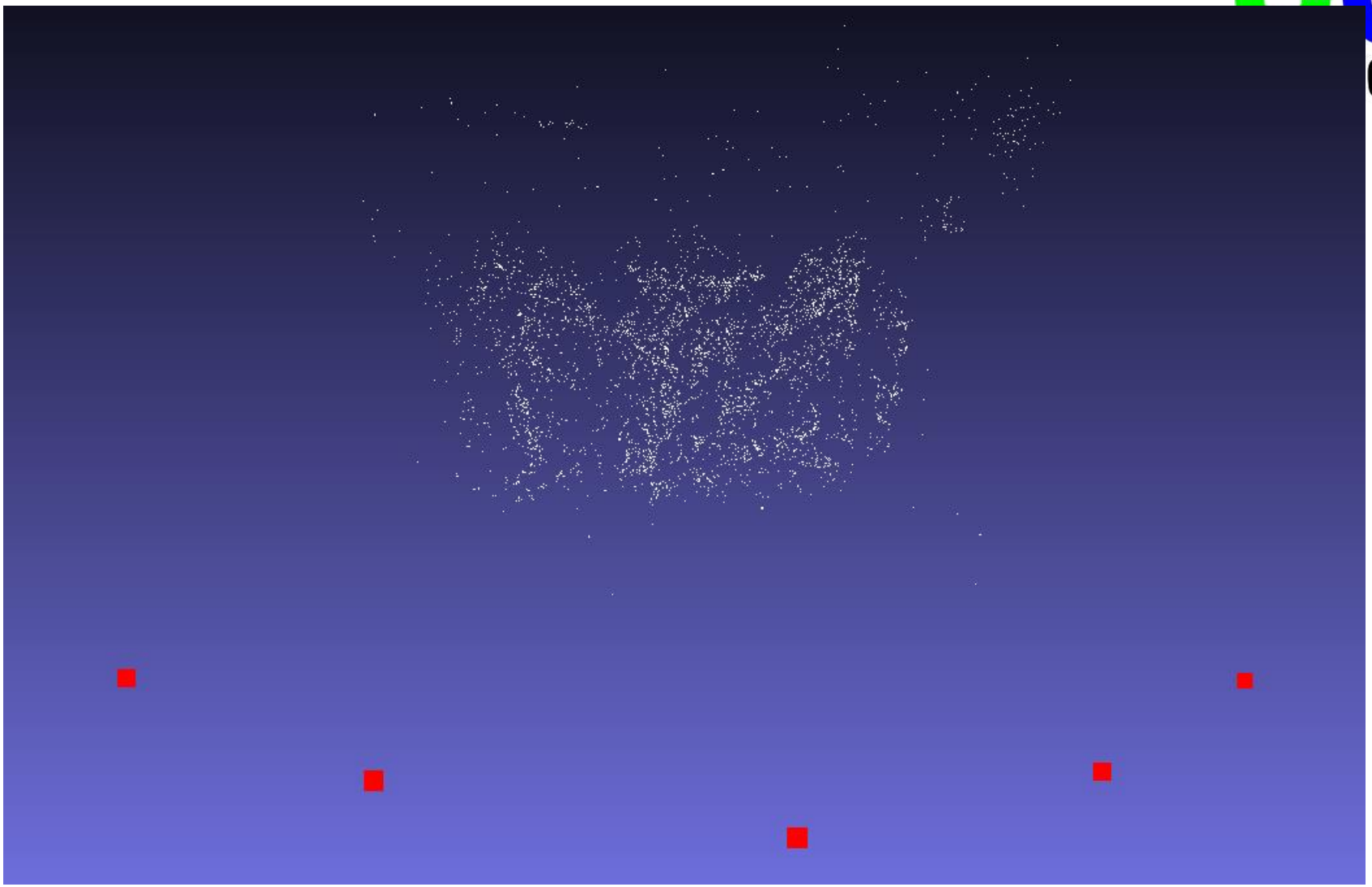
1) Build a view-graph (well-matched pairs) while finding inliers

2) Arrange 3D points, their observation and visibility $\mathbf{X}_i = [0, 0, 5]^\top$

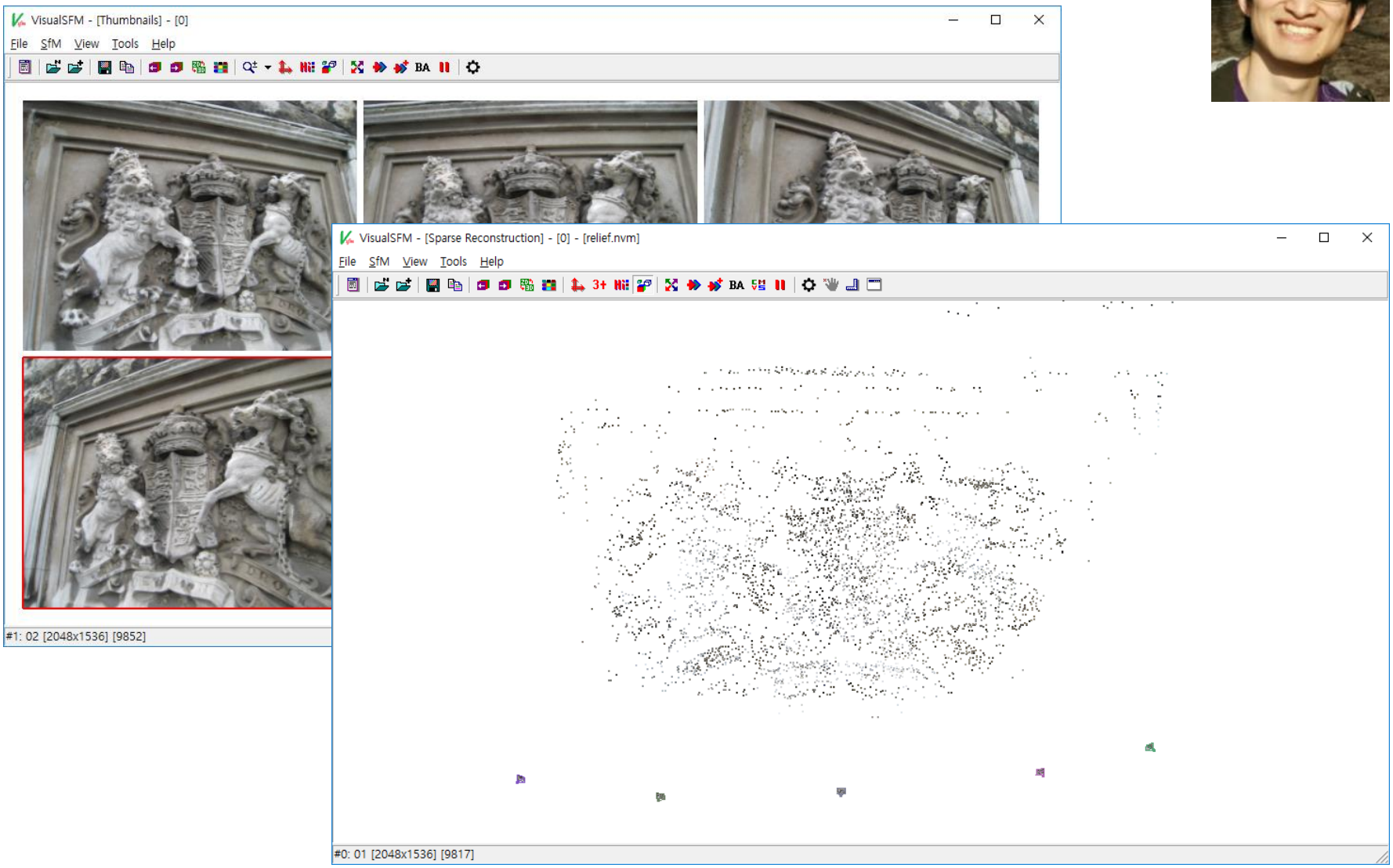
3) Initialize each camera projection matrix $\mathrm{R}_i = \mathrm{I}_{3x3} \quad \text{and} \quad \mathbf{t}_i = [0, 0, 0]^\top$

cvsba (non-linear least-square optimization)

Example: Global Structure-from-Motion



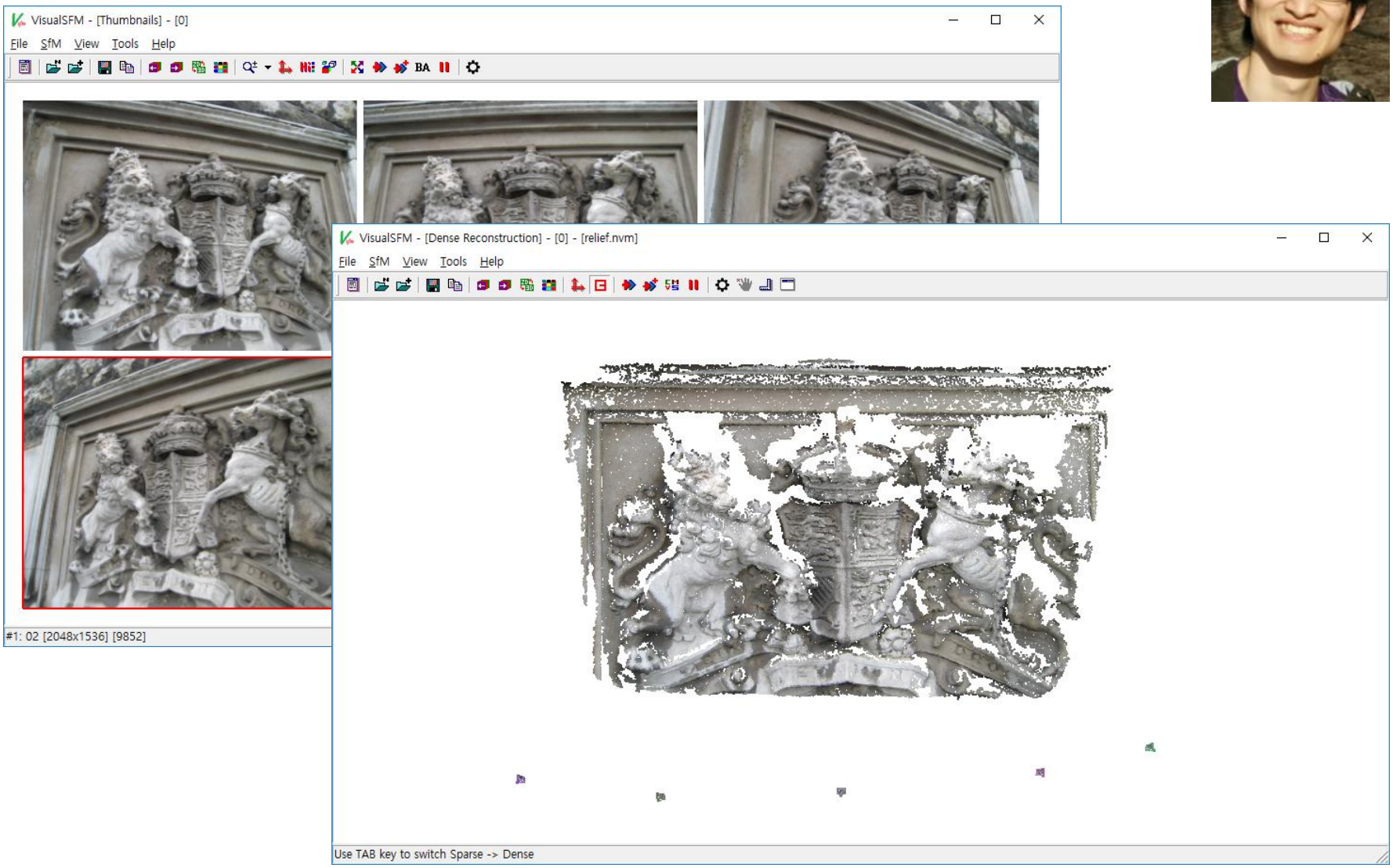
Incremental Structure-from-Motion using VisualSF



Sparse 3D Reconsturction by VisualSFM (SiftGPU and pba)

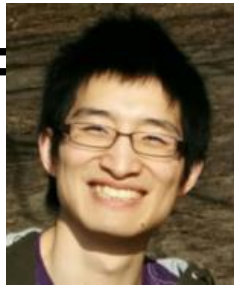
ETRI

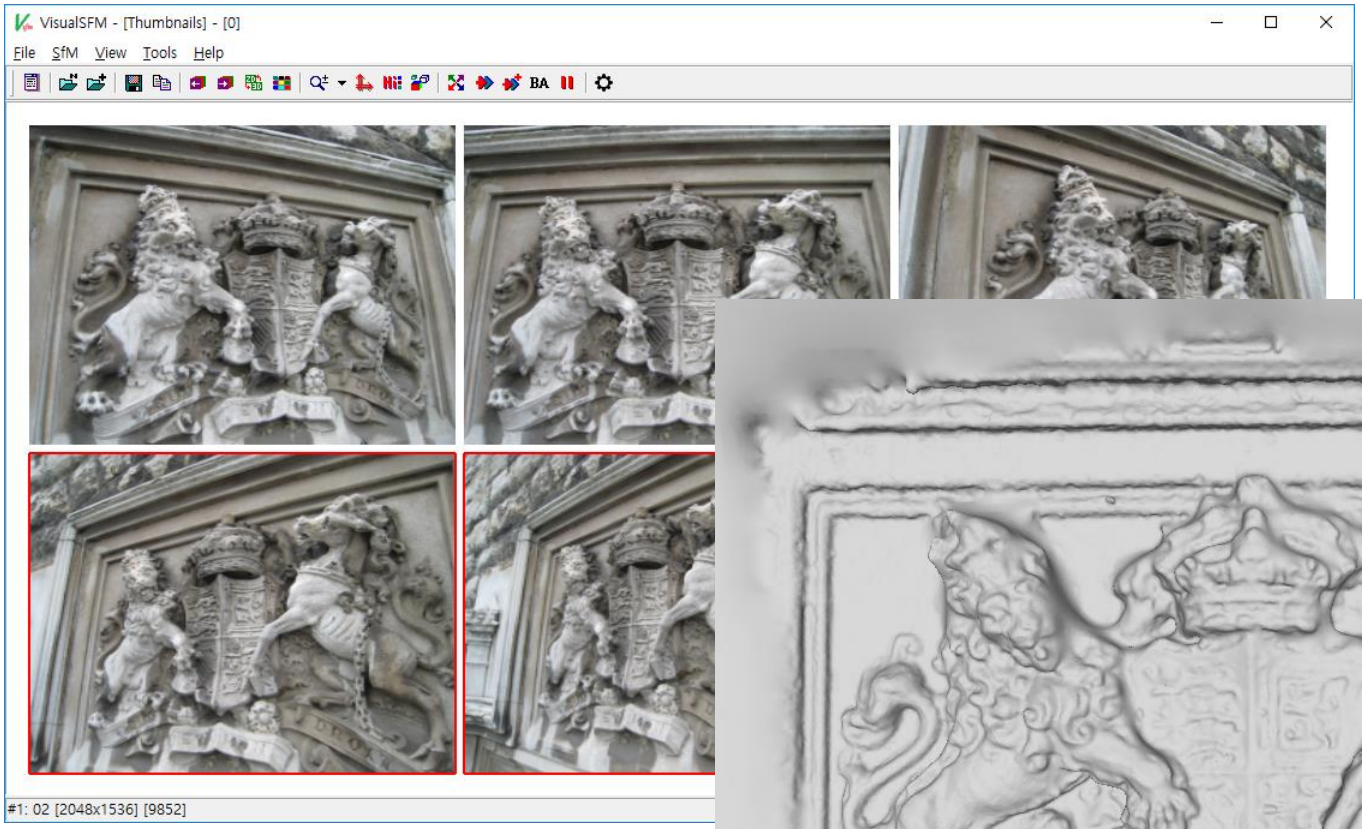
Incremental Structure-from-Motion using VisualSF



Dense 3D Reconsturction by VisualSFM (CMVS and PMVS2)

Incremental Structure-from-Motion using VisualSF





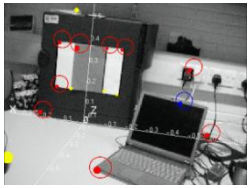
Screened Poisson Surface Reconstruction by VisualSFM [URL]

Visual SLAM: History

- **15 Years of Visual SLAM**

SLAM in Robotics

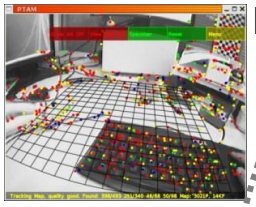
Monocular Camera



MonoSLAM (2003): features, filtering

v.s.

SfM in Computer Vision

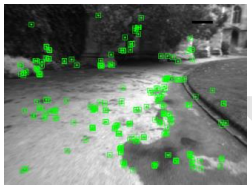


PTAM (2007): more features, optimization (keyframes)

RGB-D Camera

DTAM (2011): dense volumetric

KinectFusion (2011): dense volumetric



ORB-SLAM (2014) v.s. **LSD-SLAM (2014)**

more features

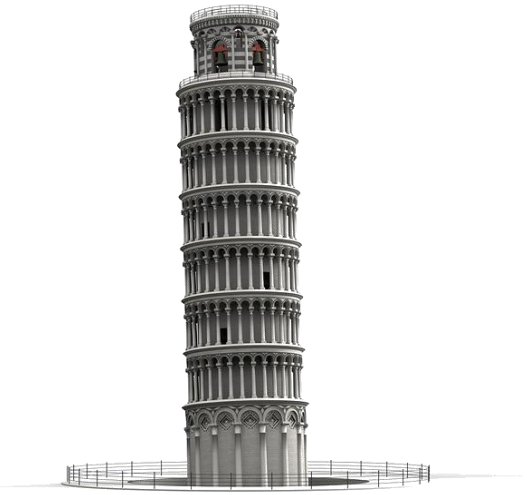
direct, semi-dense

ElasticFusion (2015) / **DynamicFusion (2015)**

dense surfels

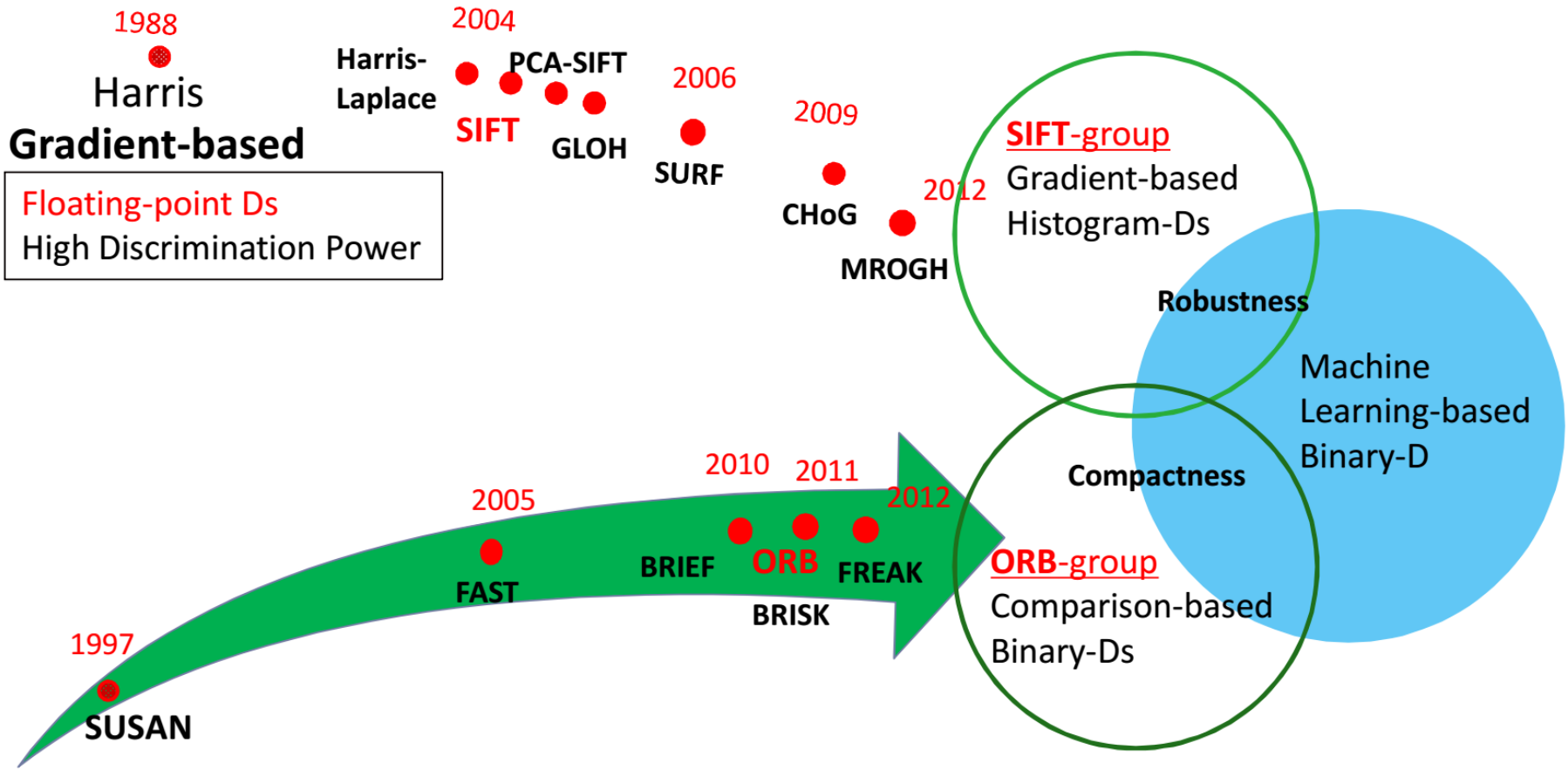
dynamic dense

Correspondence Problem



Images are from pixabay.

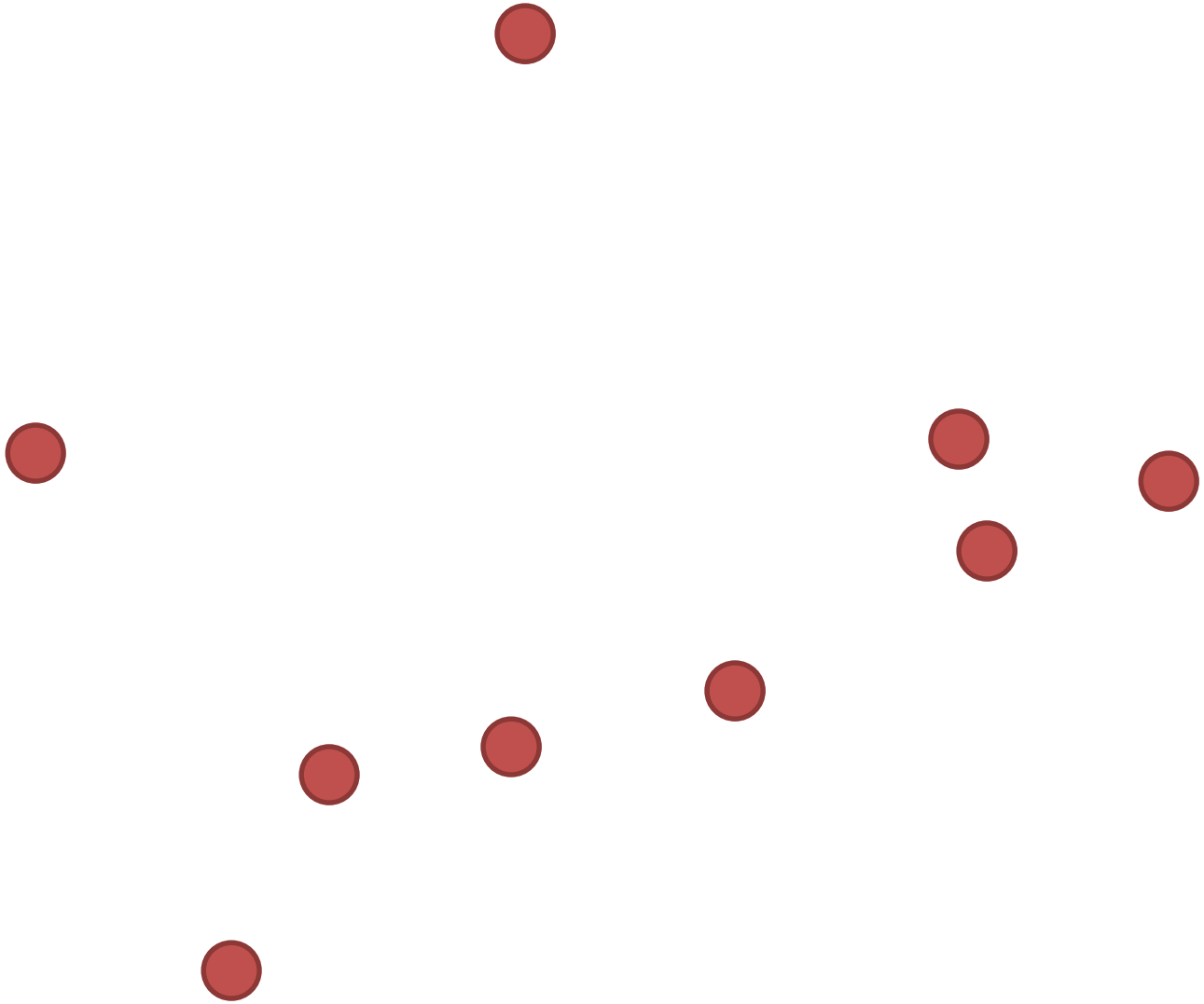
Feature Points and Descriptors



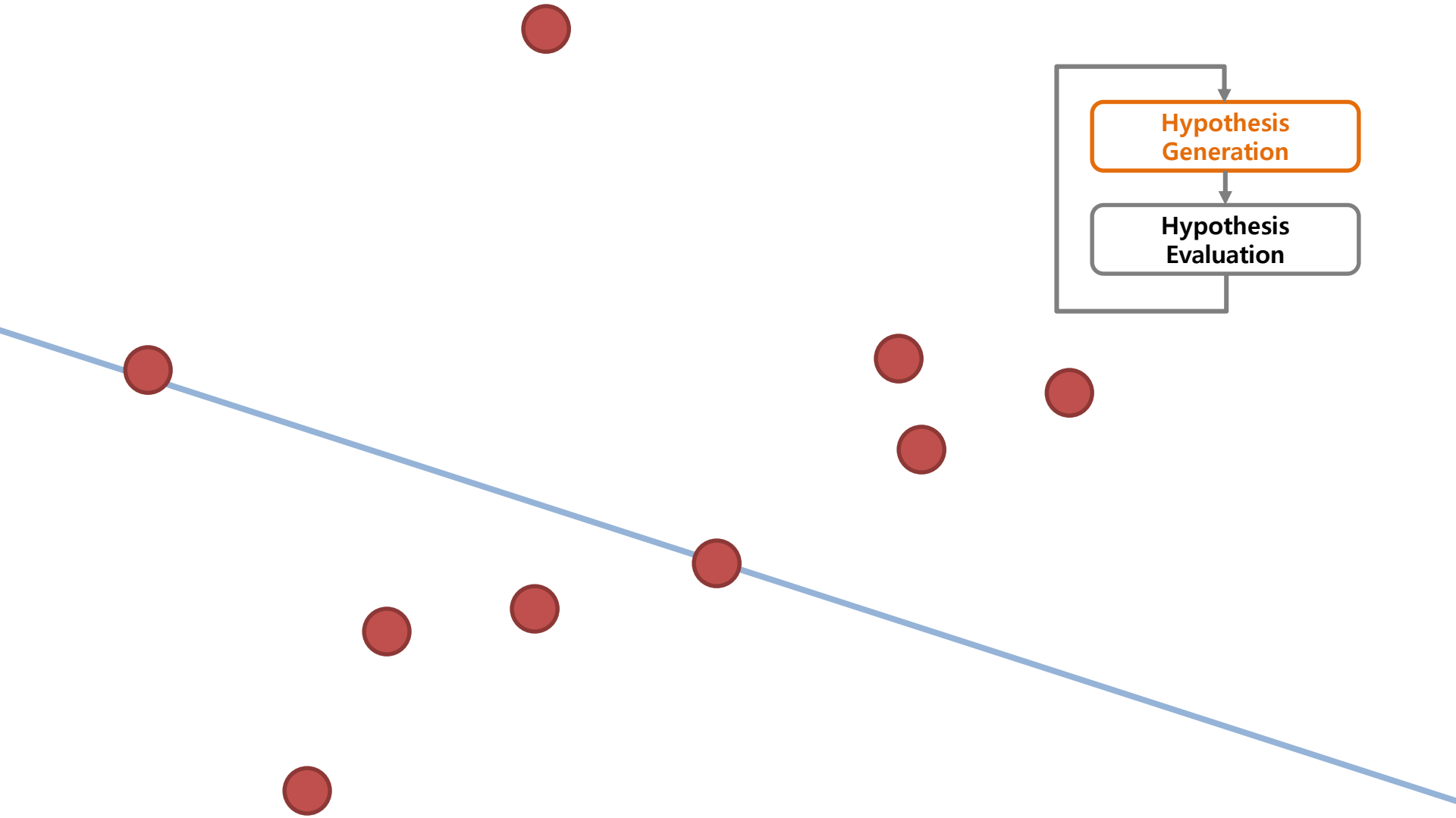
"ORB: an efficient alternative to SIFT or SURF"

Courtesy from Prof. Chee Sun Won
Courtesy from Prof. Dongbo Min

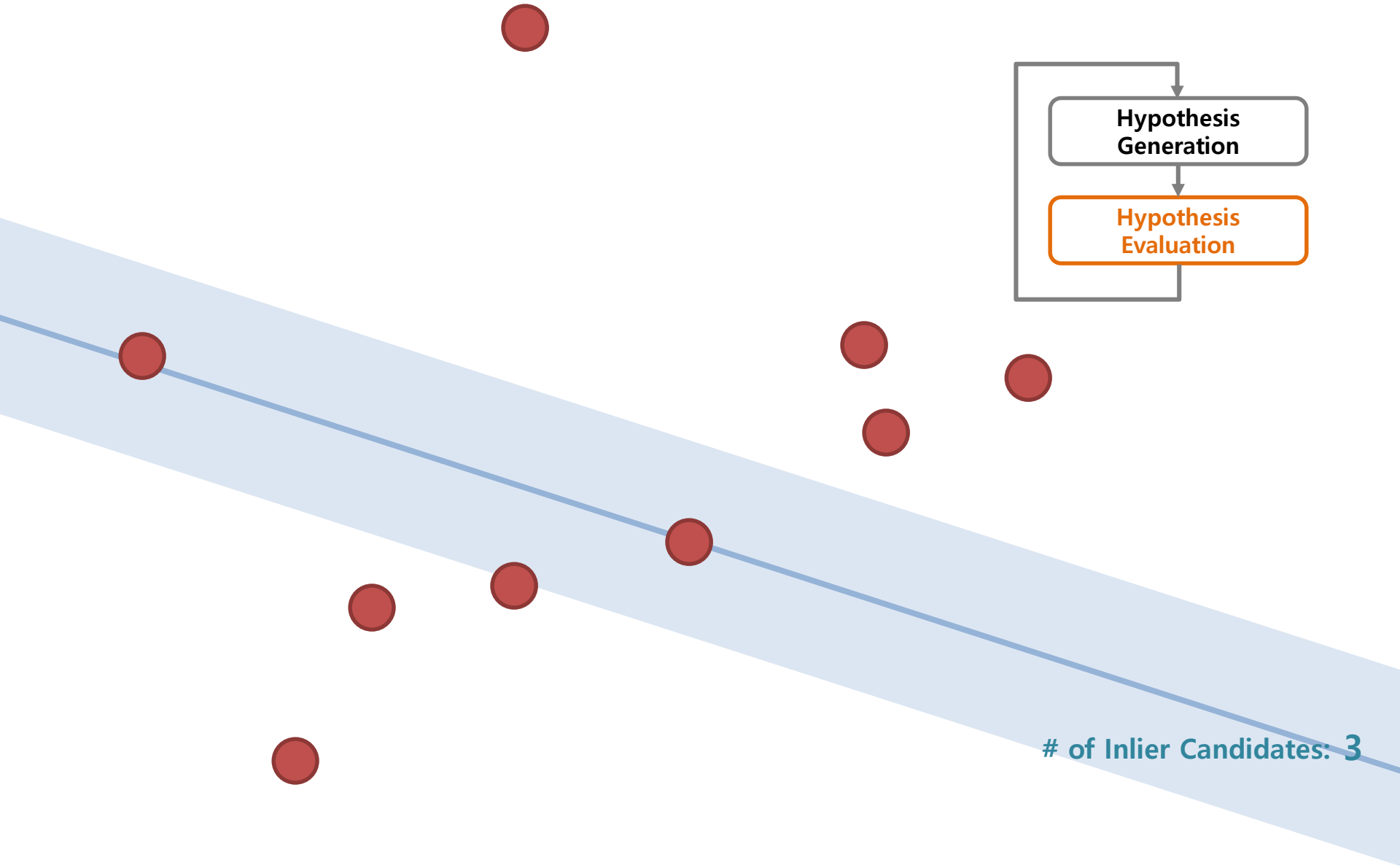
Random Sample Consensus (RANSAC)



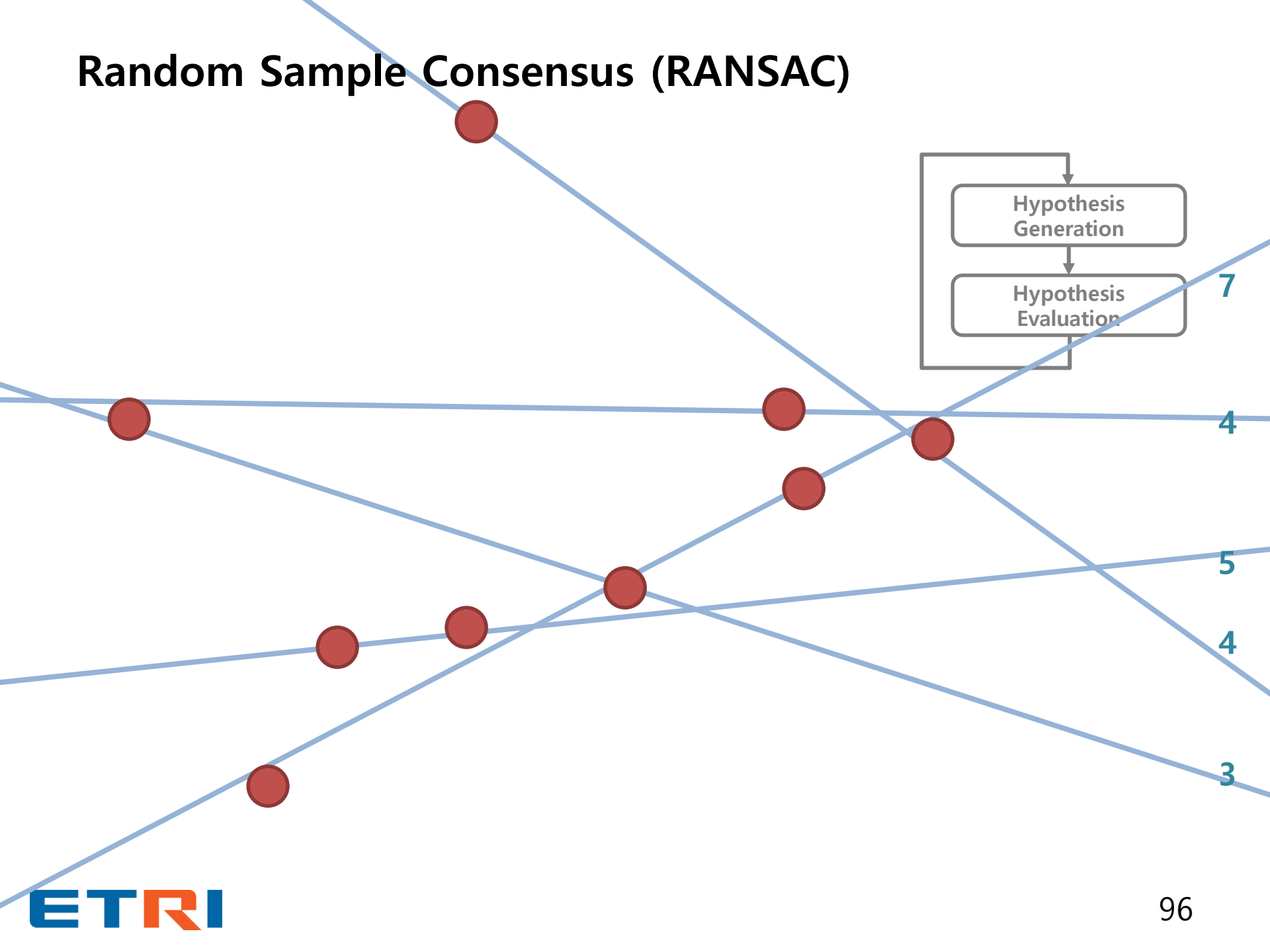
Random Sample Consensus (RANSAC)



Random Sample Consensus (RANSAC)



Random Sample Consensus (RANSAC)



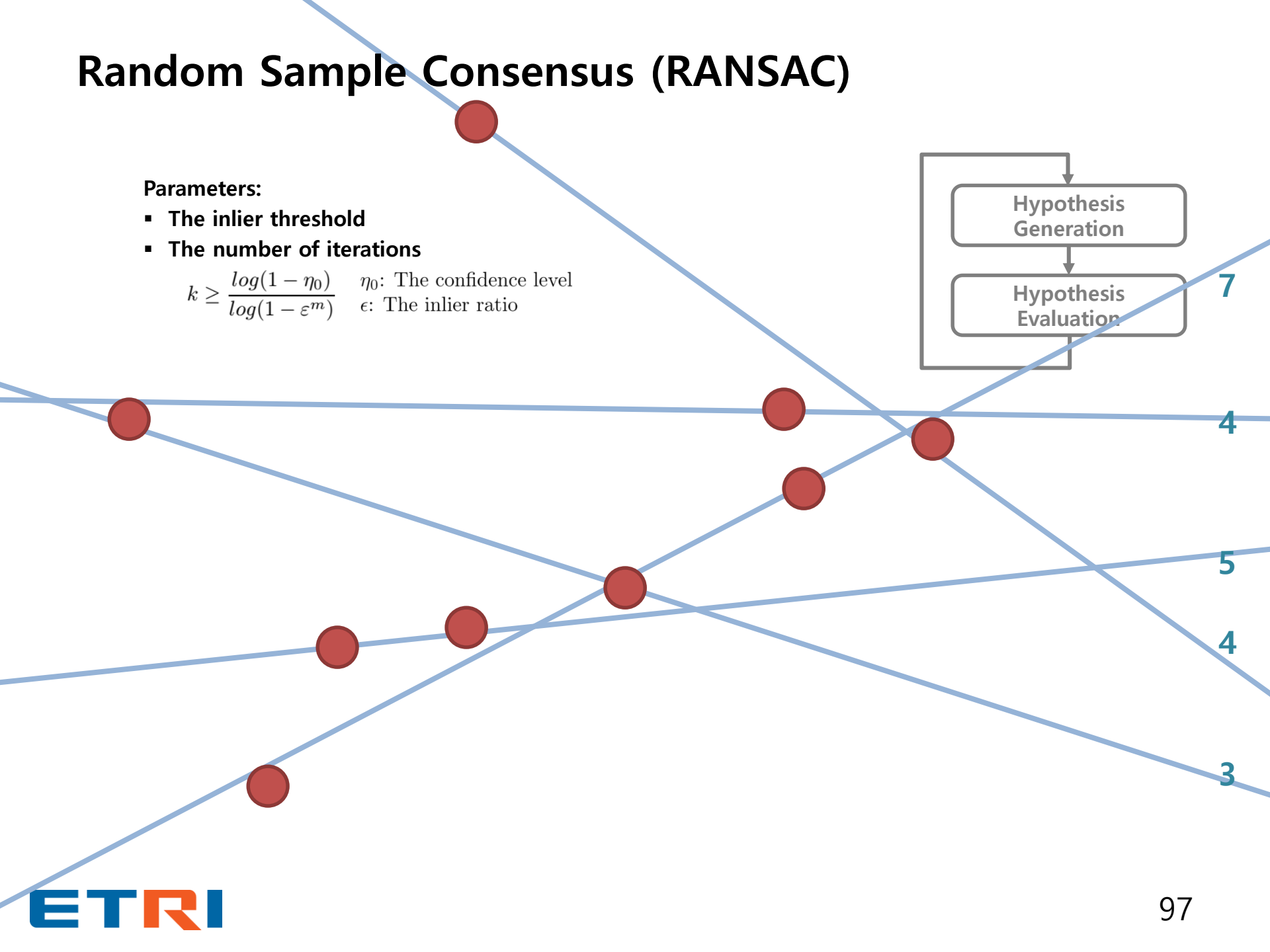
Random Sample Consensus (RANSAC)

Parameters:

- **The inlier threshold**
- **The number of iterations**

$$k \geq \frac{log(1-\eta_0)}{log(1-\varepsilon^m)}$$

η_0: The confidence level
ϵ: The inlier ratio



Example: Line Fitting with RANSAC


```
1.  #include "opencv_all.hpp"

2.  // Convert a line format, [n_x, n_y, x, y] to [a, b, c]
3.  #define CONVERT_LINE(line) (cv::Vec3d(line(0), -line(1), -line(0) * line(2) + line(1) * line(3)))

4.  int main(void)
5.  {
6.      int ransac_trial = 50;
7.      double ransac_thresh = 3.0;
8.      int ransac_n_sample = 2;
9.      int sim_n_data = 1000;
10.     double sim_inlier_ratio = 0.5, sim_inlier_noise = 1.0;
11.     cv::Vec3d truth(1.0 / sqrt(2.0), 1.0 / sqrt(2.0), -240.0); // The line model: a*x + b*y + c = 0

12.     // Generate data
13.     std::vector<cv::Point2d> data;
14.     cv::RNG rng;
15.     for (int i = 0; i < sim_n_data; i++)
16.     {
17.         if (rng.uniform(0.0, 1.0) < sim_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(sim_inlier_noise);
22.             y += rng.gaussian(sim_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. }
```

Example: Line Fitting with RANSAC


```
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, data.size());
37.             sample.push_back(data[index]);
38.         }
39.         cv::Vec4d vvxy;
40.         cv::fitLine(sample, vvxy, CV_DIST_L2, 0, 0.01, 0.01);
41.         cv::Vec3d line = CONVERT_LINE(vvxy);

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 vvxy;
57.     cv::fitLine(data, vvxy, CV_DIST_L2, 0, 0.01, 0.01);
58.     cv::Vec3d lsm_line = CONVERT_LINE(vvxy);
```

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
```

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

Summary: Overview

- **What is 3D Vision?**
- **Single-view Geometry**
  - **Camera Projection Model**
    - A Pinhole Camera Model (**Simple 2D-3D Geometry**) $\therefore \mathbf{x} = \mathrm{P}\mathbf{X} \quad \left(\mathrm{P} = \mathrm{K}\left[\mathrm{R} \,|\, \mathbf{t}\right]\right)$
    - Homogenous Coordinates $\mathbf{x} = \mathrm{K}\hat{\mathbf{x}}$
    - Geometric Distortion and Rectification
  - **General 2D-3D Geometry**
    - Camera Calibration
    - Absolute Camera Pose Estimation (PnP Problem)
- **Two-view Geometry**
  - **Planar 2D-2D Geometry (Projective Geometry)** $\therefore \mathbf{x}' = \mathrm{H}\mathbf{x}$
    - Planar Homography $\hat{\mathbf{x}}' = \hat{\mathrm{H}}\hat{\mathbf{x}} \quad \left(\hat{\mathrm{H}} = \mathrm{R} + \frac{1}{d}\mathbf{t}\mathbf{n}^\top\right)$
  - **General 2D-2D Geometry (Epipolar Geometry)**
    - Fundamental/Essential Matrix $\therefore \mathbf{x}'^\top \mathrm{F}\mathbf{x} = 0$
    - Relative Camera Pose Estimation $\hat{\mathbf{x}}'^\top \mathrm{E}\hat{\mathbf{x}} = 0 \quad \left(\mathrm{E} = [\mathbf{t}]_\times \mathrm{R}\right)$
    - Relative Point Localization (Triangulation)
- **Multi-view Geometry**
  - Bundle Adjustment (Non-linear Optimization) $\sum_i^n \sum_j^m \left\| \mathbf{x}_i^j - \mathrm{P}_j \mathbf{X}_i \right\|_\Sigma^2$
  - Applications: Structure-from-motion, Visual SLAM, and Visual Odometry
- **Correspondence Problem**
  - Random Sample Consensus (RANSAC)

Further Information

- **Beyond Point Features**
  - Other features: Line-based SfM (3DPVT 2006), Pointless SfM (ICCV 2015)
  - Direct methods (w/o feature extraction)
    - Deep learning: PoseNet, CNN-SLAM (CVPR 2017)
    - Photometric error minimization: DTAM, LSD-SLAM, DSO, SVO
- ~~**Real-time / Large-scale SfM**~~
- **(Spatially / Temporally) Non-static SfM**
  - Non-rigid (or moving) objects: Non-rigid SfM
- **Sensor Fusion and New Sensors**
  - Depth (RGB-D, stereo, LiDAR): ElasticFusion, DVO-SLAM, RGB-D SLAM
  - Omni-directional camera: Multi-FoV datasets
  - Light-field camera
  - Event camera: ETAM (ECCV 2016)
  - IMU: OKVIS
  - GPS
- **Minimal Solvers / Self-calibration**
  - OpenGV