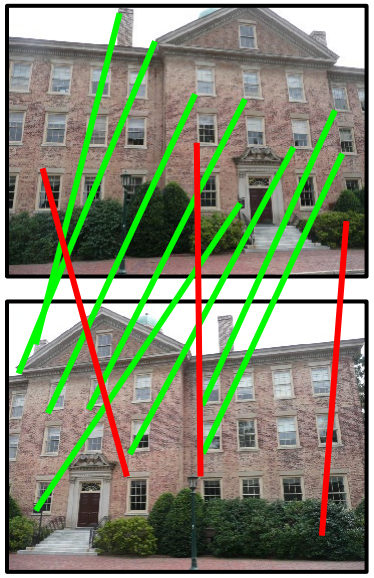
Robotic Mapping & Localization

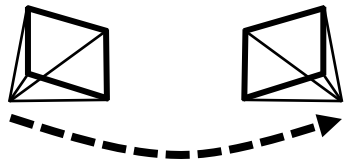
Kaveh Fathian
Assistant Professor
Computer Science Department
Colorado School of Mines

Lec06: Camera Model

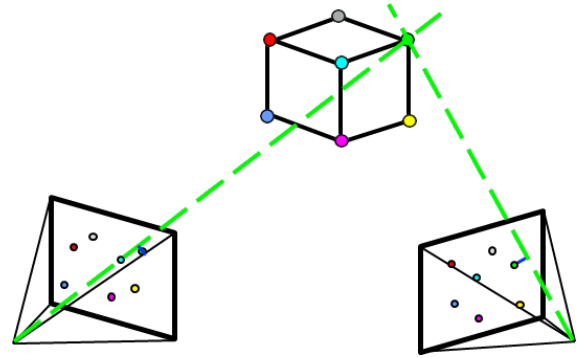
Simple SfM Pipeline

1. Feature Extraction → 2. Feature Matching → 3. Geometric Verification → 4. Triangulation





Robust Two-View Geometry Estimation



Our Simple SLAM System

robotic-mapping / code / visual-odometry / vo_epipolar.cpp

KavehFathian first commit

Code Blame 97 lines (88 loc) · 3.52 KB

```cpp
#include <iostream>
#include <vector>
#include <string>
#include "opencv2/opencv.hpp"

int main()
{
    const char* video_file = "../data/KITTI07/image_0/%06d.png";
    double f = 707.0912;
    cv::Point2d c(601.8873, 183.1104);
    bool use_5pt = true;
    int min_inlier_num = 100;
    double min_inlier_ratio = 0.2;
    const char* traj_file = "vo_epipolar.xyz";
```

Our Simple SLAM System


```cpp
#include <iostream>
#include <vector>
#include <string>
#include "opencv2/opencv.hpp"

int main()
{
    const char* video_file = "../data/KITTI07/image_0/%06d.png";
    double f = 707.0912;
    cv::Point2d c(601.8873, 183.1104);
    bool use_5pt = true;
    int min_inlier_num = 100;
    double min_inlier_ratio = 0.2;
    const char* traj_file = "vo_epipolar.xyz";

    // Open a video and get an initial image
    cv::VideoCapture video;
    if (!video.open(video_file)) return -1;

    cv::Mat gray_prev;
    video >> gray_prev;
    if (gray_prev.empty())
    {
        video.release();
        return -1;
    }
    if (gray_prev.channels() > 1) cv::cvtColor(gray_prev, gray_prev,
    cv::COLOR_RGB2GRAY);

    // Run the monocular visual odometry
    cv::Mat K = (cv::Mat_<double>(3, 3) << f, 0, c.x, 0, f, c.y, 0, 0, 1);
    cv::Mat camera_pose = cv::Mat::eye(4, 4, CV_64F);
    FILE* camera_traj = fopen(traj_file, "wt");
    if (camera_traj == NULL) return -1;
```

OpenCV library

Camera parameters

SLAM system parameters

Convert RGB images to gray

Camera calibration matrix

Pose (position & orientation) estimates

Our Simple SLAM System

```cpp
while (true)
{
    // Grab an image from the video
    cv::Mat img, gray;
    video >> img;
    if (img.empty()) break;
    if (img.channels() > 1) cv::cvtColor(img, gray, cv::COLOR_RGB2GRAY);
    else                    gray = img.clone();

    // Extract optical flow
    std::vector<cv::Point2f> pts_prev, pts;
    cv::goodFeaturesToTrack(gray_prev, pts_prev, 2000, 0.01, 10);
    std::vector<uchar> status;
    cv::Mat err;
    cv::calcOpticalFlowPyrLK(gray_prev, gray, pts_prev, pts, status, err);
    gray_prev = gray;

    // Calculate relative pose
    cv::Mat E, inlier_mask;
    if (use_5pt)
    {
        E = cv::findEssentialMat(pts_prev, pts, f, c, cv::RANSAC, 0.999, 1,
        inlier_mask);
    }
    else
    {
        cv::Mat F = cv::findFundamentalMat(pts_prev, pts, cv::FM_RANSAC, 1, 0.
        99, inlier_mask);
        E = K.t() * F * K;
    }
    cv::Mat R, t;
    int inlier_num = cv::recoverPose(E, pts_prev, pts, R, t, f, c, inlier_mask);
    double inlier_ratio = static_cast<double>(inlier_num) / static_cast<double>
    (pts.size());

    // Accumulate relative pose if result is reliable
    cv::Vec3b info_color(0, 255, 0);
    if ((inlier_num > min_inlier_num) && (inlier_ratio > min_inlier_ratio))
    {
        cv::Mat T = cv::Mat::eye(4, 4, R.type());
        T(cv::Rect(0, 0, 3, 3)) = R * 1.0;
        T.col(3).rowRange(0, 3) = t * 1.0;
        camera_pose = camera_pose * T.inv();
        info_color = cv::Vec3b(0, 0, 255);
    }
```

Iterate over images

Feature detection & tracking using optical flow

Estimate camera pose (in the form of essential matrix) from consecutive images

RANSAC outlier rejection

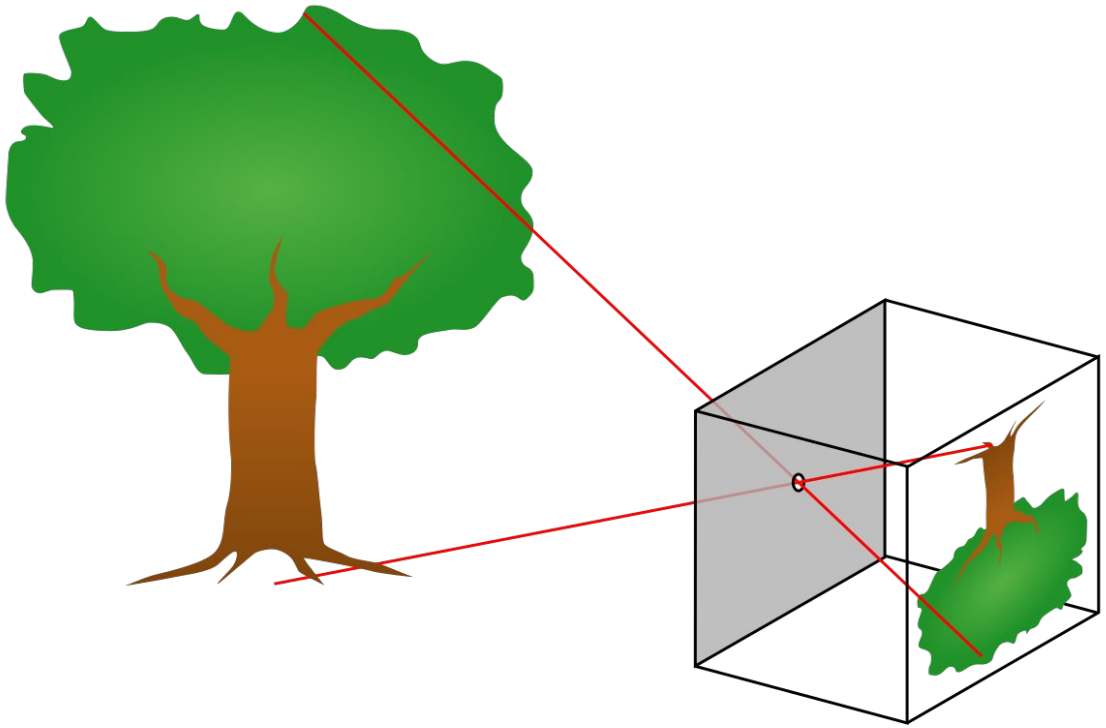
Extract camera pose from essential matrix

Camera pose (and trajectory) over time

Lecture Outline

Camera

- Coordinate transformations
- Pinhole camera model
- Calibration matrix

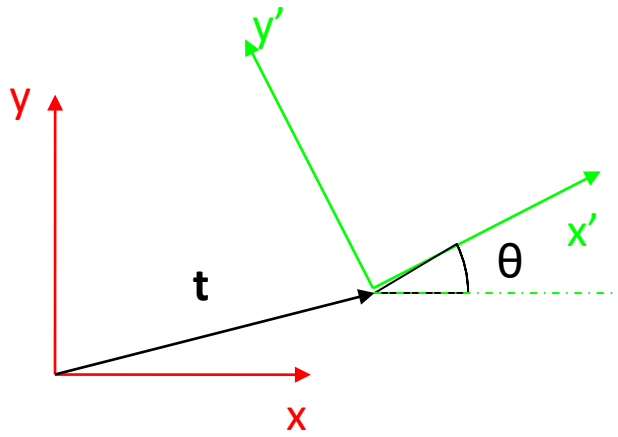


Coordinate Transforms

Slides are courtesy of Tom Williams @ Mines

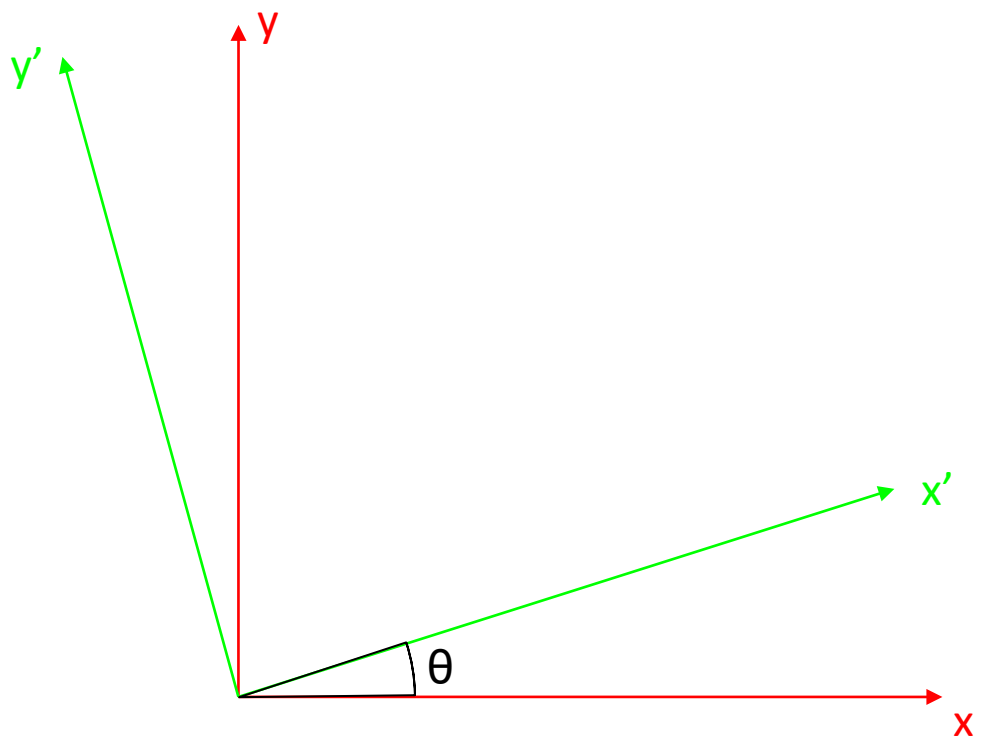
2D Rigid Frame Transformations

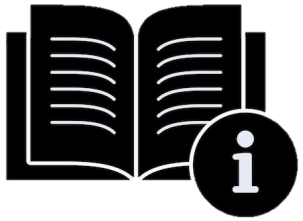
- The pose of one 2D frame with respect to another is described by
  - Translation vector $\mathbf{t}=(\Delta x,\Delta y)^T$
  - Rotation angle θ
    - Rotation can also be represented as a 2x2 matrix **R**
- Object shape and size is preserved



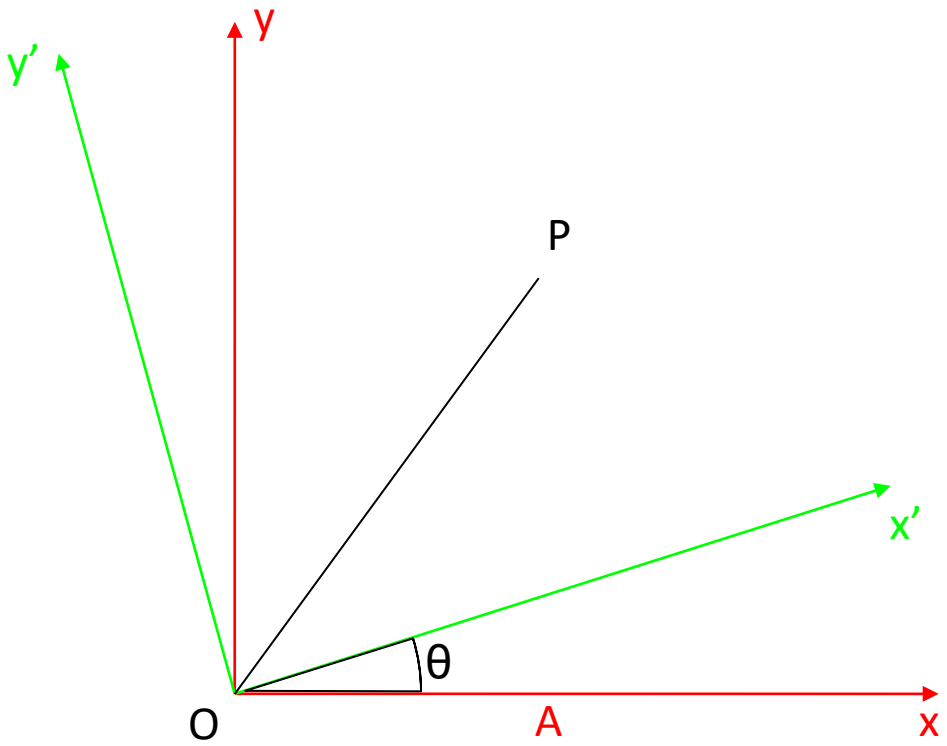
Rotations in 2D

- Let's derive the formula for a 2D rotation

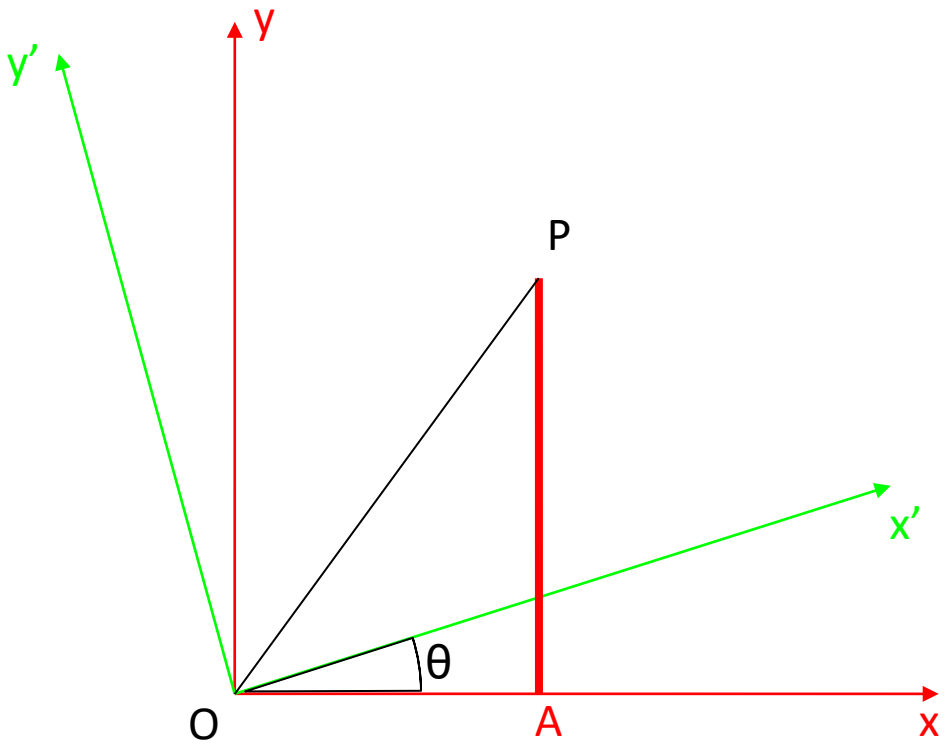




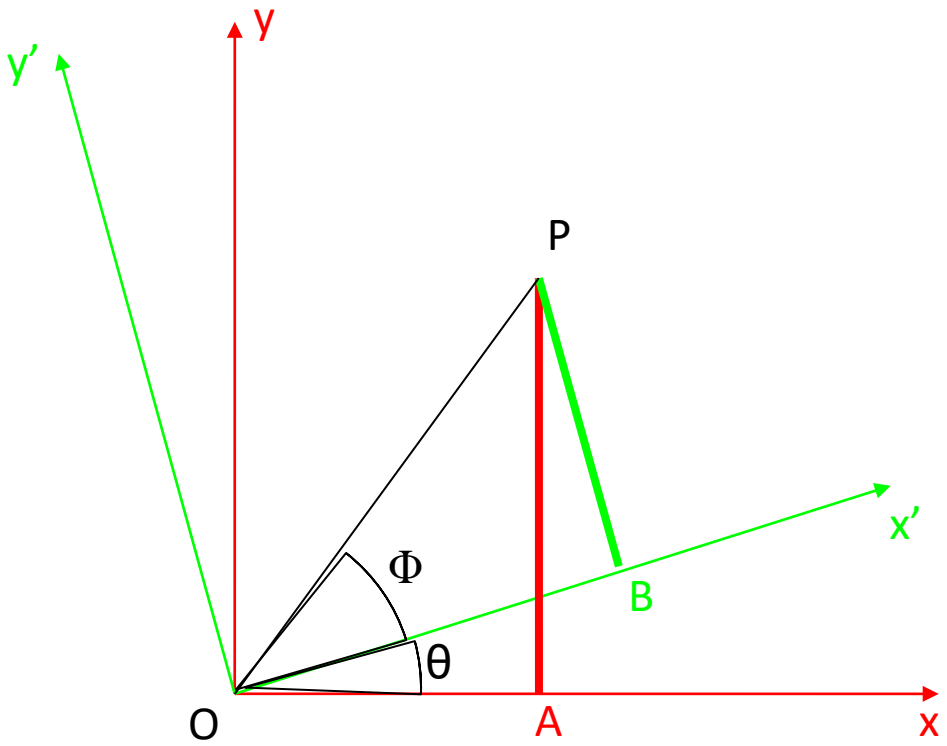
Rotations in 2D



Rotations in 2D

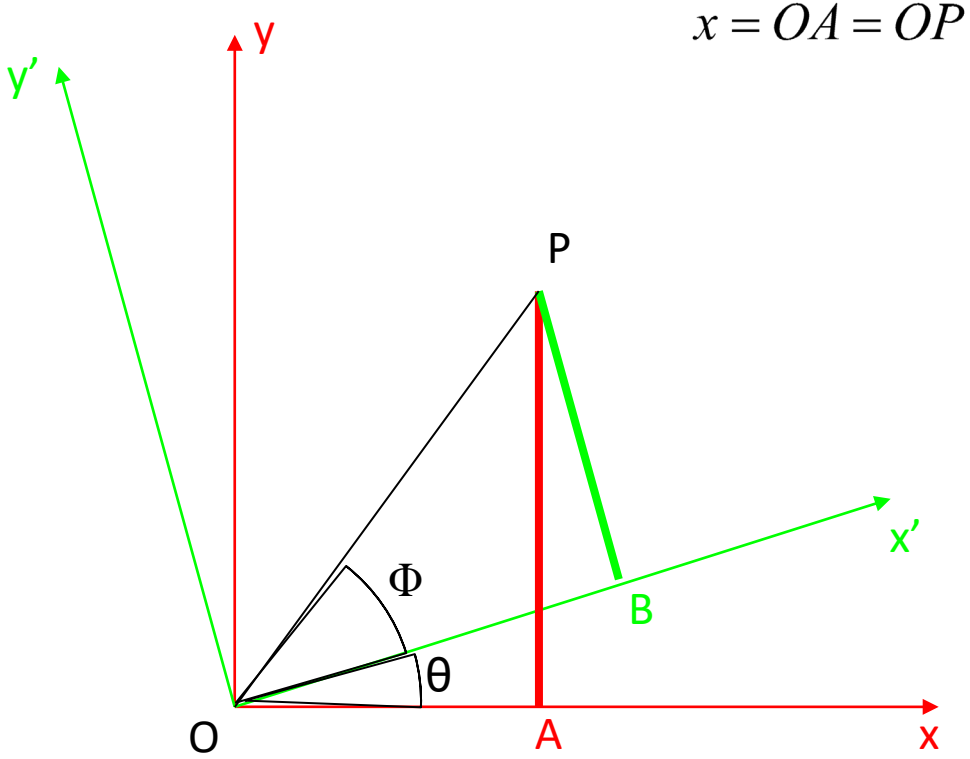


Rotations in 2D



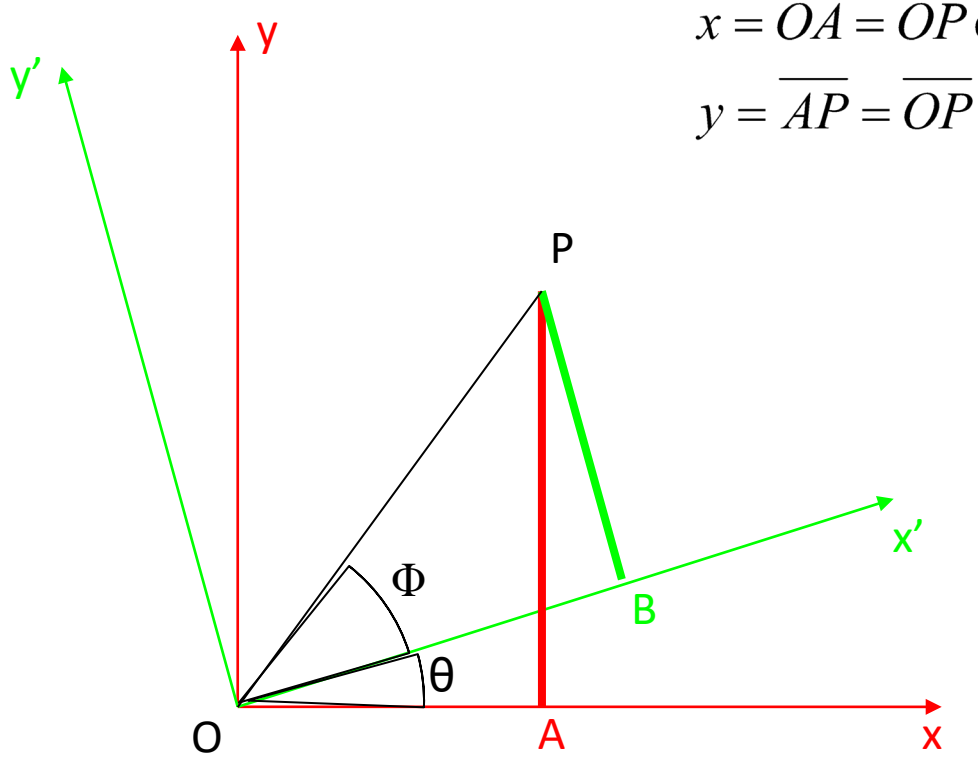
Rotations in 2D

$$x = \overline{OA} = \overline{OP}\cos(\theta + \phi)$$

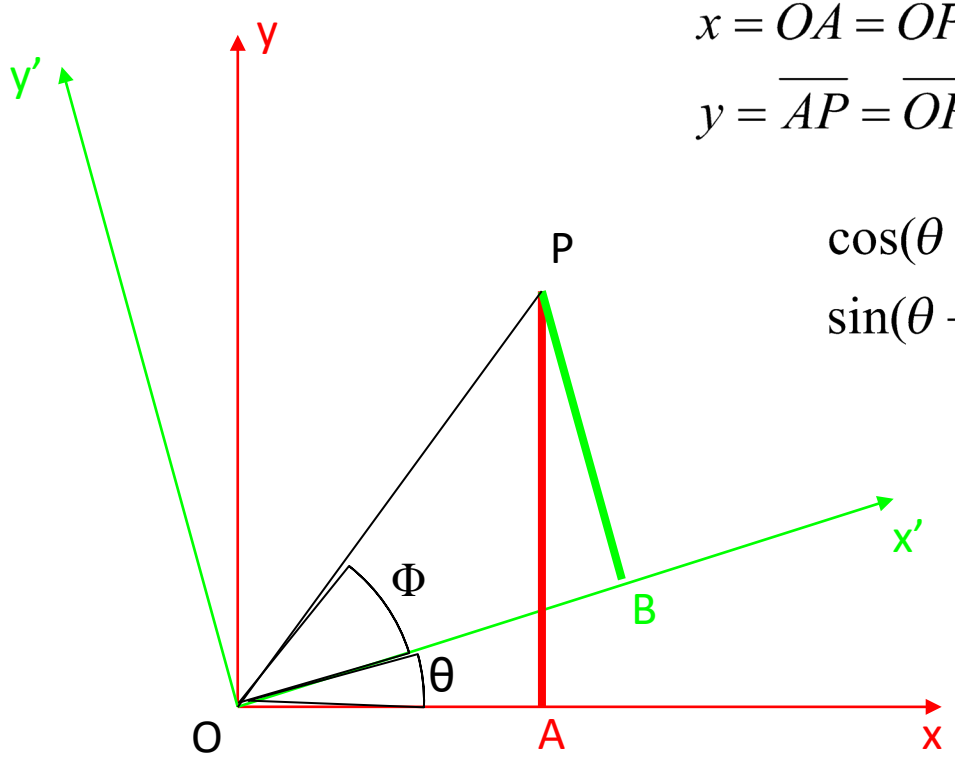


Rotations in 2D

$$x = \overline{OA} = \overline{OP}\cos(\theta + \phi)$$
$$y = \overline{AP} = \overline{OP}\sin(\theta + \phi)$$



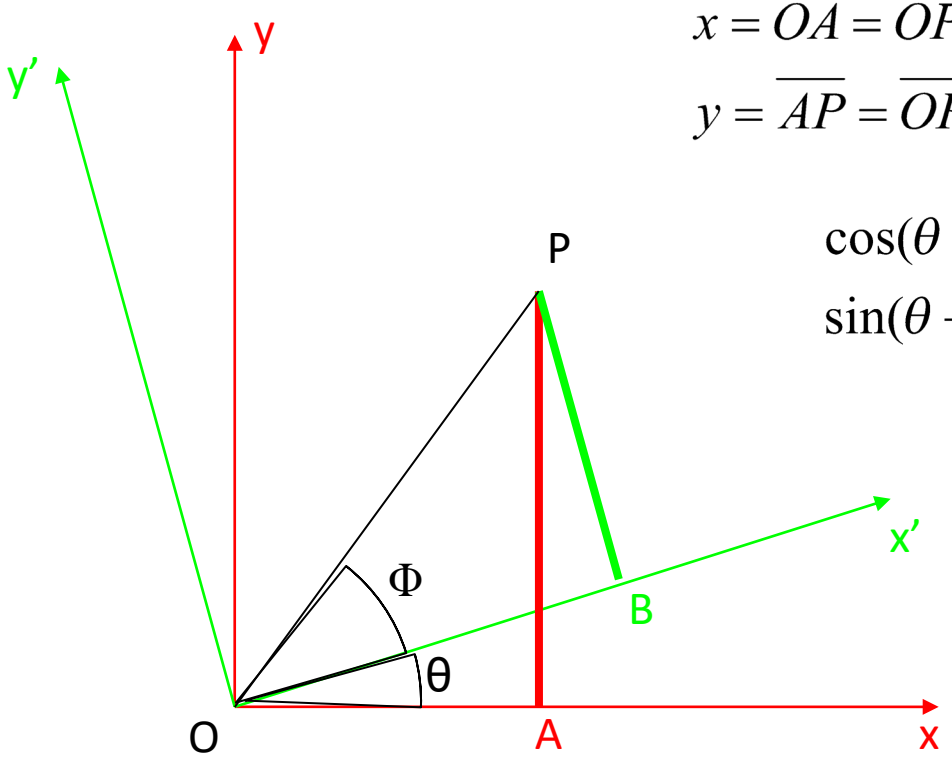
Rotations in 2D



$$x = \overline{OA} = \overline{OP}\cos(\theta + \phi)$$
$$y = \overline{AP} = \overline{OP}\sin(\theta + \phi)$$

$$\cos(\theta + \phi) = \cos\theta\cos\phi - \sin\theta\sin\phi$$
$$\sin(\theta + \phi) = \cos\theta\sin\phi + \sin\theta\cos\phi$$

Rotations in 2D

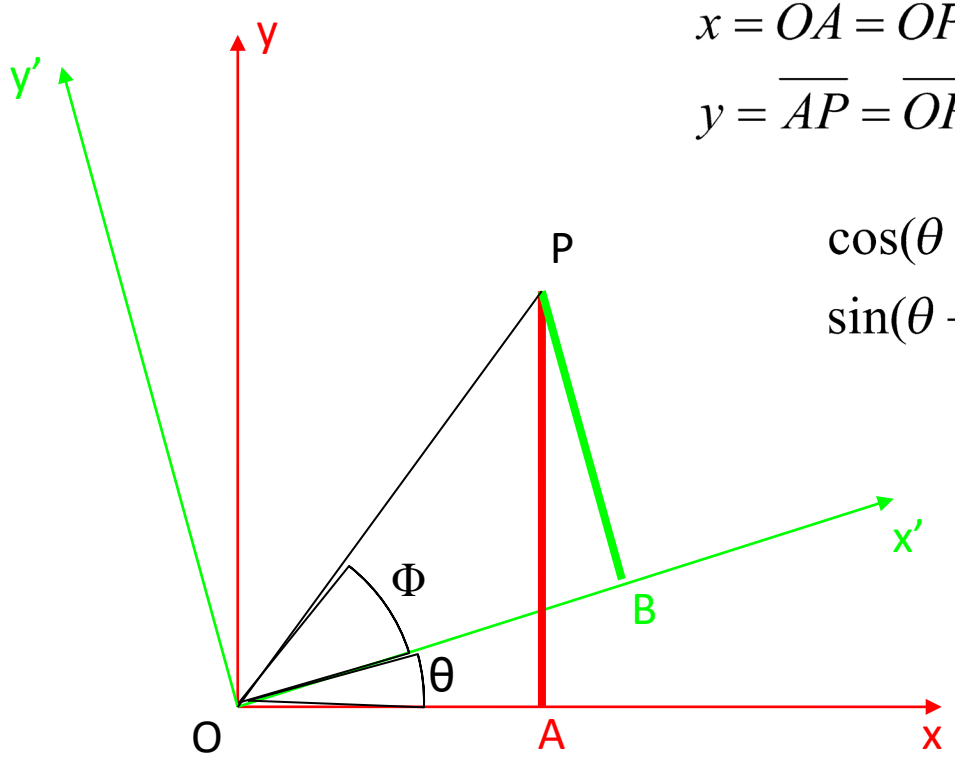


$$x = \overline{OA} = \overline{OP}\cos(\theta + \phi)$$
$$y = \overline{AP} = \overline{OP}\sin(\theta + \phi)$$

$$\cos(\theta + \phi) = \cos\theta\cos\phi - \sin\theta\sin\phi$$
$$\sin(\theta + \phi) = \cos\theta\sin\phi + \sin\theta\cos\phi$$

$$x = \overline{OP}\cos\phi\cos\theta - \overline{OP}\sin\phi\sin\theta$$

Rotations in 2D

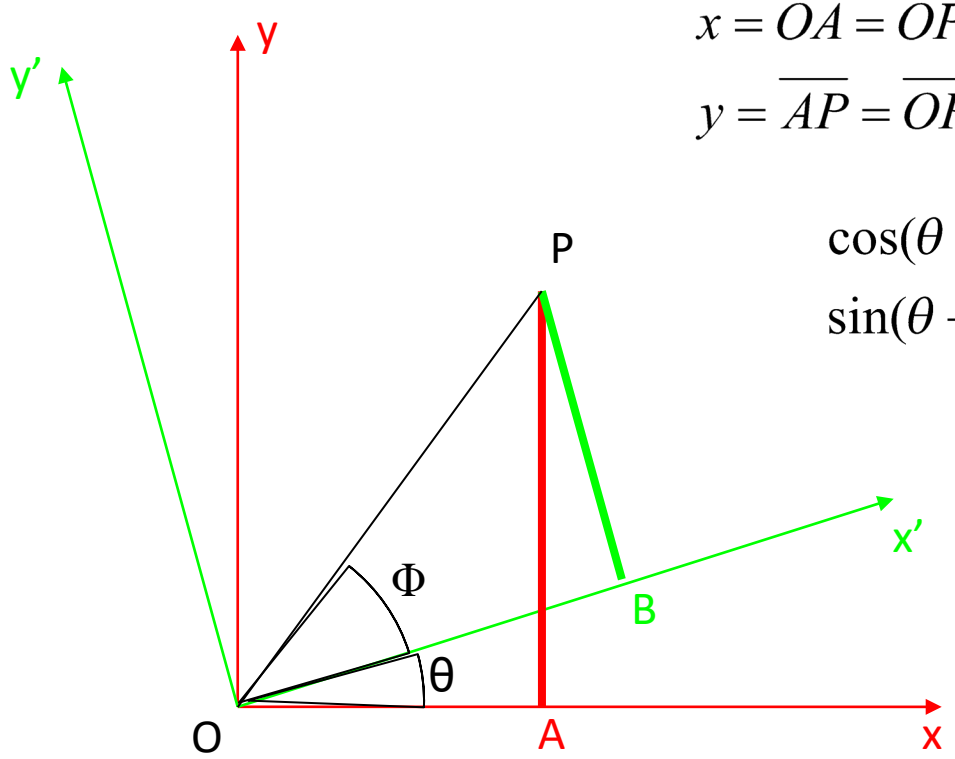


$$x = \overline{OA} = \overline{OP}\cos(\theta + \phi)$$
$$y = \overline{AP} = \overline{OP}\sin(\theta + \phi)$$

$$\cos(\theta + \phi) = \cos\theta\cos\phi - \sin\theta\sin\phi$$
$$\sin(\theta + \phi) = \cos\theta\sin\phi + \sin\theta\cos\phi$$

$$x = \underbrace{\overline{OP}\cos\phi}_{x'}\cos\theta - \underbrace{\overline{OP}\sin\phi}_{y'}\sin\theta$$

Rotations in 2D

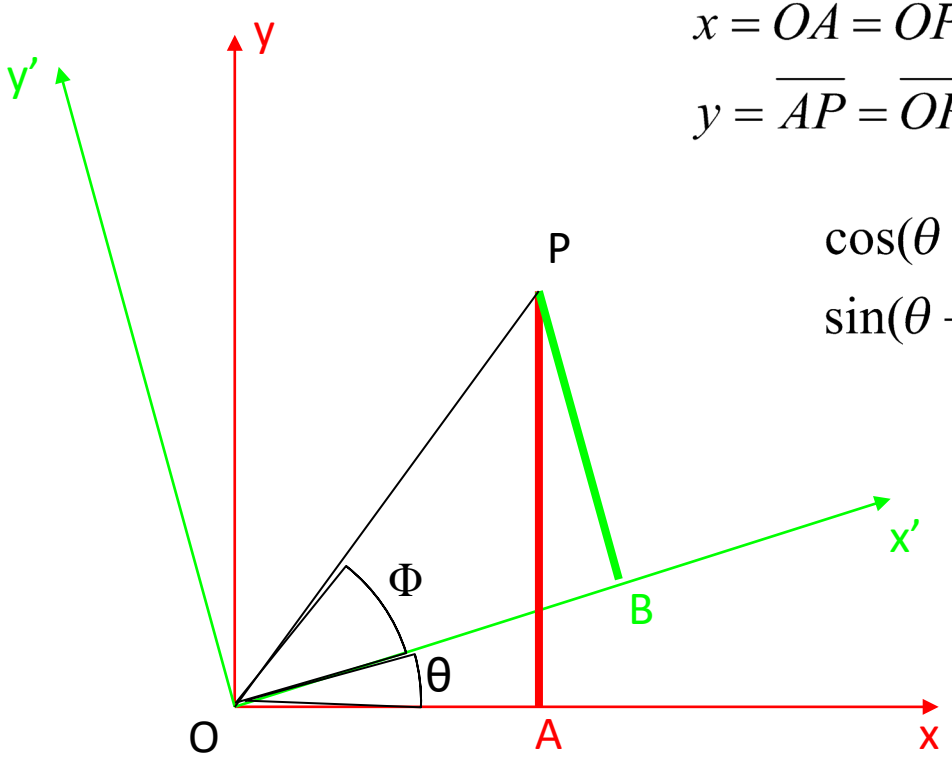


$$x = \overline{OA} = \overline{OP}\cos(\theta + \phi)$$
$$y = \overline{AP} = \overline{OP}\sin(\theta + \phi)$$

$$\cos(\theta + \phi) = \cos\theta\cos\phi - \sin\theta\sin\phi$$
$$\sin(\theta + \phi) = \cos\theta\sin\phi + \sin\theta\cos\phi$$

$$x = \underbrace{\overline{OP}\cos\phi}_{x'}\cos\theta - \underbrace{\overline{OP}\sin\phi}_{y'}\sin\theta$$
$$= x'\cos\theta - y'\sin\theta$$

Rotations in 2D



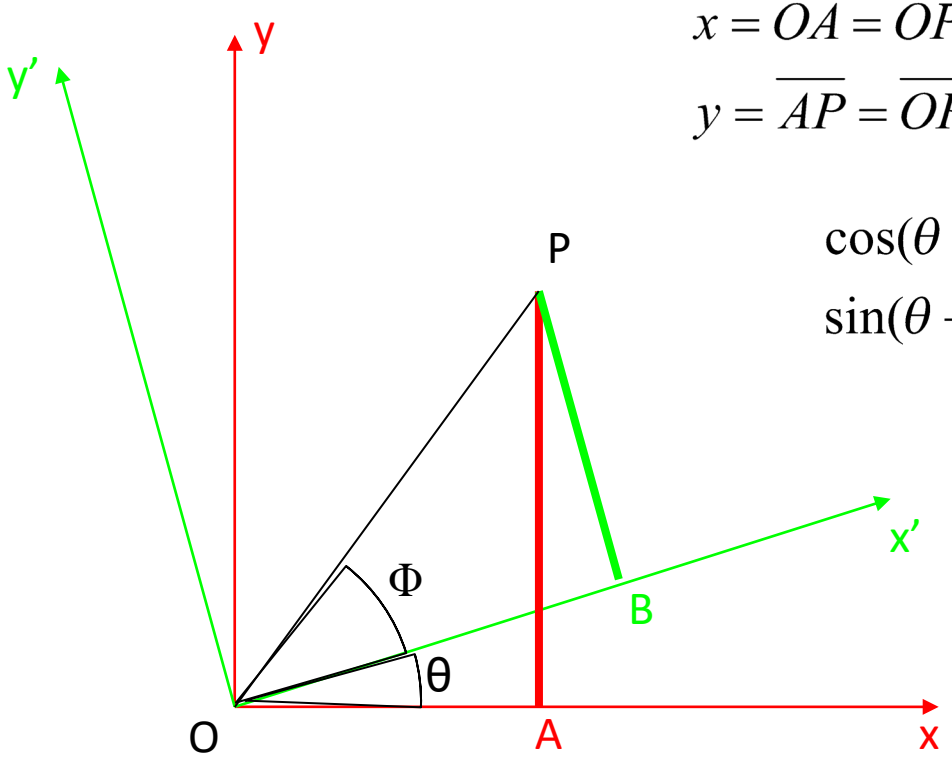
$$x = \overline{OA} = \overline{OP}\cos(\theta + \phi)$$
$$y = \overline{AP} = \overline{OP}\sin(\theta + \phi)$$

$$\cos(\theta + \phi) = \cos\theta\cos\phi - \sin\theta\sin\phi$$
$$\sin(\theta + \phi) = \cos\theta\sin\phi + \sin\theta\cos\phi$$

$$x = \underbrace{\overline{OP}\cos\phi}_{x'}\cos\theta - \underbrace{\overline{OP}\sin\phi}_{y'}\sin\theta$$
$$= x'\cos\theta - y'\sin\theta$$

Similarly, $y = x'\sin\theta + y'\cos\theta$

Rotations in 2D



$$x = \overline{OA} = \overline{OP}\cos(\theta + \phi)$$
$$y = \overline{AP} = \overline{OP}\sin(\theta + \phi)$$

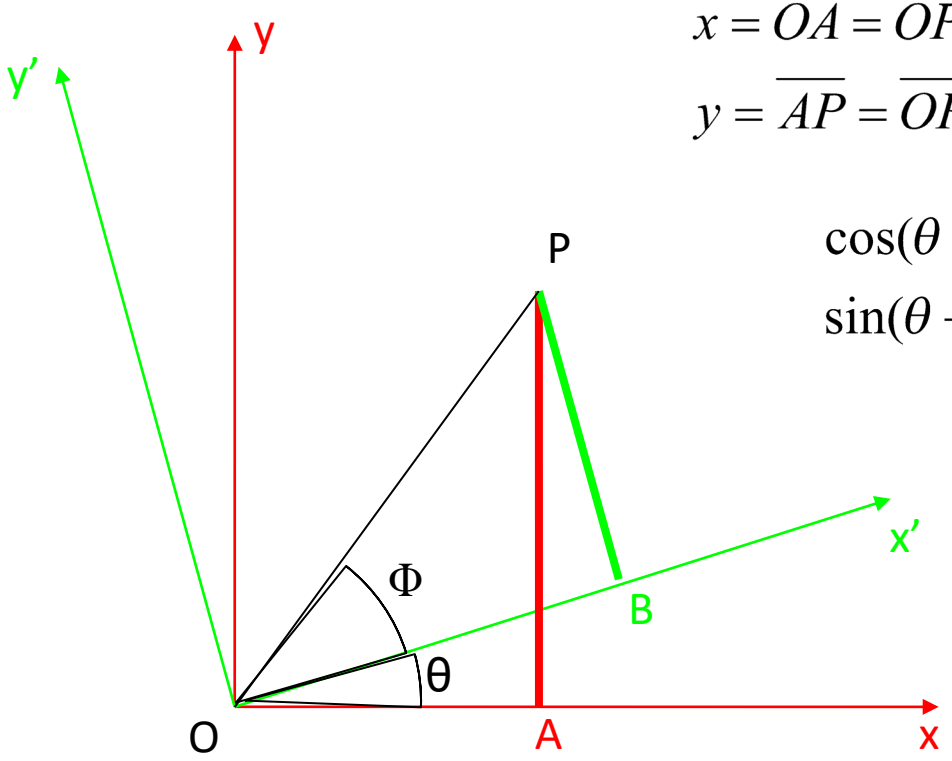
$$\cos(\theta + \phi) = \cos\theta\cos\phi - \sin\theta\sin\phi$$
$$\sin(\theta + \phi) = \cos\theta\sin\phi + \sin\theta\cos\phi$$

$$x = \underbrace{\overline{OP}\cos\phi}_{x'}\cos\theta - \underbrace{\overline{OP}\sin\phi}_{y'}\sin\theta$$
$$= x'\cos\theta - y'\sin\theta$$

Similarly, $y = x'\sin\theta + y'\cos\theta$

So $$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$$

Rotations in 2D



$$x = \overline{OA} = \overline{OP}\cos(\theta + \phi)$$
$$y = \overline{AP} = \overline{OP}\sin(\theta + \phi)$$

$$\cos(\theta + \phi) = \cos\theta\cos\phi - \sin\theta\sin\phi$$
$$\sin(\theta + \phi) = \cos\theta\sin\phi + \sin\theta\cos\phi$$

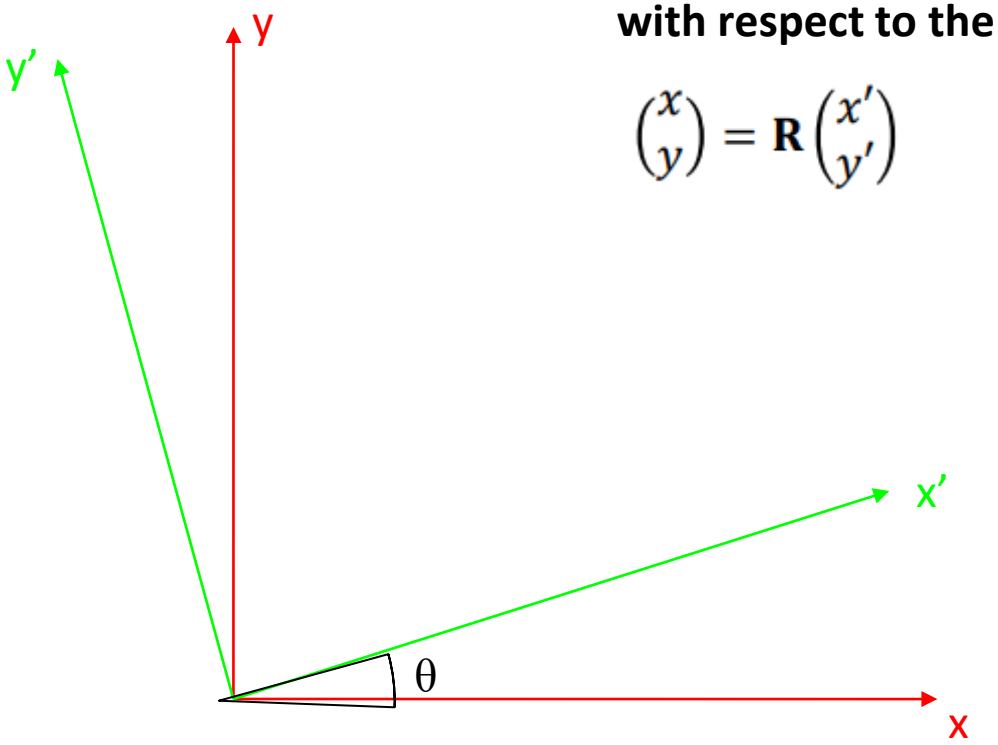
$$x = \underbrace{\overline{OP}\cos\phi}_{x'}\cos\theta - \underbrace{\overline{OP}\sin\phi}_{y'}\sin\theta$$
$$= x'\cos\theta - y'\sin\theta$$

Similarly, $y = x'\sin\theta + y'\cos\theta$

So $\begin{pmatrix} x \\ y \end{pmatrix} = \underbrace{\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}}_{\mathbf{R}} \begin{pmatrix} x' \\ y' \end{pmatrix}$

$$\mathbf{R} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

Rotations in 2D

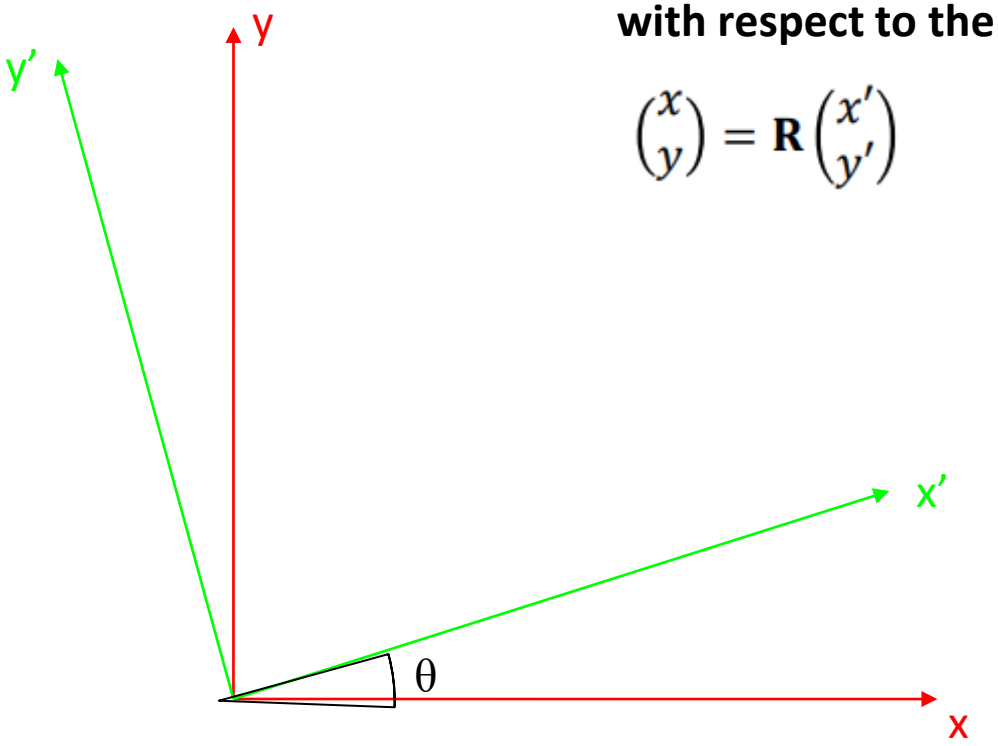


R describes the orientation of the primed frame with respect to the unprimed frame.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \mathbf{R} \begin{pmatrix} x' \\ y' \end{pmatrix}$$

- **R** is orthonormal
  - Rows, columns are orthogonal ($\mathbf{r}_1 \cdot \mathbf{r}_2 = 0$, $\mathbf{c}_1 \cdot \mathbf{c}_2 = 0$)
  - (in both directions you get $\cos\theta\sin\theta - \cos\theta\sin\theta$)
  - Transpose is the inverse; $\mathbf{R}\mathbf{R}^T = \mathbf{I}$
  - Determinant is $|\mathbf{R}| = 1$

Rotations in 2D



R describes the orientation of the primed frame with respect to the unprimed frame.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \mathbf{R} \begin{pmatrix} x' \\ y' \end{pmatrix}$$

- **R** is orthonormal
  - Rows, columns are orthogonal ($\mathbf{r}_1 \cdot \mathbf{r}_2 = 0$, $\mathbf{c}_1 \cdot \mathbf{c}_2 = 0$)
  - (in both directions you get $\cos\theta\sin\theta - \cos\theta\sin\theta$)
  - Transpose is the inverse; $\mathbf{R}\mathbf{R}^T = \mathbf{I}$
  - Determinant is $|\mathbf{R}| = 1$

WE CARE BECAUSE THIS MEANS THAT $\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{R}^T \begin{pmatrix} x \\ y \end{pmatrix}$

Homogeneous Coordinates

- Points can be represented using homogeneous coordinates
  - This simply means to append a 1 as an extra element
  - If the 3rd element becomes ≠ 1, we divide through by it

$$\tilde{\mathbf{x}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} sx \\ sy \\ s \end{pmatrix}$$

Homogeneous Coordinates

- Points can be represented using homogeneous coordinates
  - This simply means to append a 1 as an extra element
  - If the 3rd element becomes ≠ 1, we divide through by it

$$\tilde{\mathbf{x}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} sx \\ sy \\ s \end{pmatrix}$$

- Effectively, vectors that differ only by scale are considered to be equivalent

Homogeneous Coordinates

- Points can be represented using homogeneous coordinates
  - This simply means to append a 1 as an extra element
  - If the 3rd element becomes ≠ 1, we divide through by it

$$\tilde{\mathbf{x}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} sx \\ sy \\ s \end{pmatrix}$$

- Effectively, vectors that differ only by scale are considered to be equivalent
- This simplifies transform equations; instead of

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{R}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} \qquad \mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t}$$

Homogeneous Coordinates

- Points can be represented using homogeneous coordinates
  - This simply means to append a 1 as an extra element
  - If the 3rd element becomes ≠ 1, we divide through by it

$$\tilde{\mathbf{x}} = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} sx \\ sy \\ s \end{pmatrix}$$

- Effectively, vectors that differ only by scale are considered to be equivalent
- This simplifies transform equations; instead of

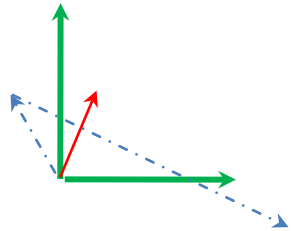
$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \mathbf{R}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} \qquad \mathbf{x}' = \mathbf{R}\mathbf{x} + \mathbf{t}$$

- we have

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R} & \mathbf{t} \\ 0\ 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad \tilde{\mathbf{x}}' = \mathbf{H}\tilde{\mathbf{x}}$$

Example

- Transform the 2D point $x = (10, 20)^T$ using a rotation of 45 degrees and translation of (+40, -30).

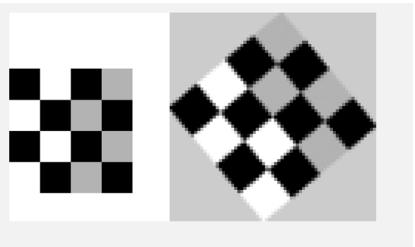


1) Point in Homogeneous Coordinates?
2) Rotation Matrix R?
3) Translation Matrix T?
4) Full Transformation Matrix?
5) How do we apply this transformation to the point?

Other 2D-2D Transforms

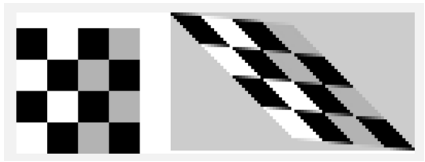
- Scaled (similarity) transform
  - preserves angles but not distances

$$\begin{pmatrix} x_B \\ y_B \\ 1 \end{pmatrix} = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_A \\ y_A \\ 1 \end{pmatrix}$$



- Affine transform
  - Models rotation, translation, scaling, shearing, and reflection

$$\begin{pmatrix} x_B \\ y_B \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_A \\ y_A \\ 1 \end{pmatrix}$$

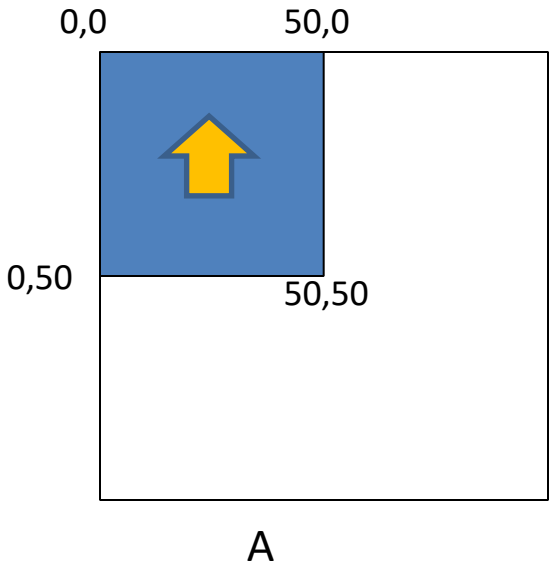


  - How many degrees of freedom?
    How many pairs of corresponding points needed to calculate transformation?

Example

- Image “A” is modified by the affine transform below. Sketch image “B”

$$\begin{pmatrix} x_B \\ y_B \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0.25 & 1.5 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_A \\ y_A \\ 1 \end{pmatrix}$$



Projective Transform (Homography)

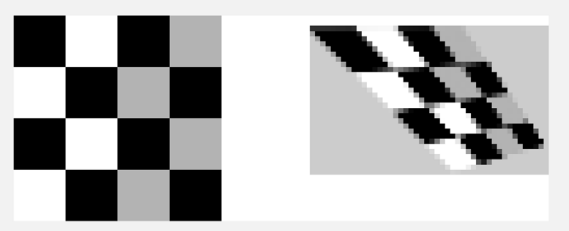
- Most general type of linear 2D-2D transform
- H is an arbitrary 3x3 matrix

$$\tilde{\mathbf{x}}' = \mathbf{H}\,\tilde{\mathbf{x}}$$

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- We still need to divide by the 3rd element

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 / x_3 \\ x_2 / x_3 \\ 1 \end{pmatrix}$$



As we will see later, a homography maps points from the projection of one plane to the projection of another plane

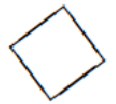
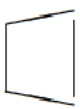
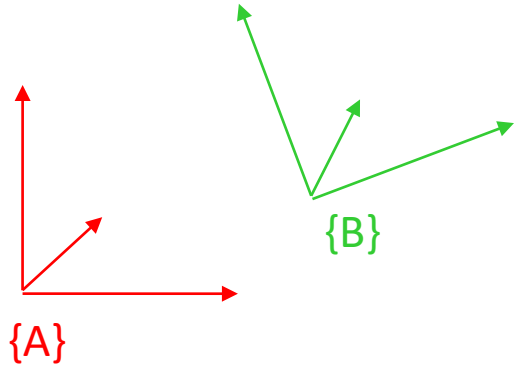
Transformation	Matrix	# DoF	Preserves	Icon
translation	$\begin{bmatrix} \boldsymbol{I} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	2	orientation	
rigid (Euclidean)	$\begin{bmatrix} \boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	3	lengths	
similarity	$\begin{bmatrix} s\boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	4	angles	
affine	$\begin{bmatrix} \boldsymbol{A} \end{bmatrix}_{2\times3}$	6	parallelism	
projective	$\begin{bmatrix} \tilde{\boldsymbol{H}} \end{bmatrix}_{3\times3}$	8	straight lines	

Table 2.1 Hierarchy of 2D coordinate transformations. Each transformation also preserves the properties listed in the rows below it, i.e., similarity preserves not only angles but also parallelism and straight lines. The 2×3 matrices are extended with a third $[\mathbf{0}^T\ 1]$ row to form a full 3×3 matrix for homogeneous coordinate transformations.

From Szeliski, Computer Vision: Algorithms and Applications

3D Coordinate Systems

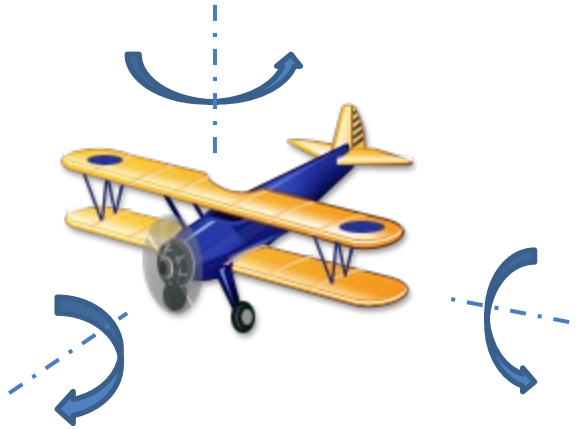
- Coordinate frames
  - Denote as {A}, {B}, etc
  - Examples: camera, world
- The pose of {B} with respect to {A} is described by
  - Translation vector **t**
  - Rotation matrix **R**
- Spoilers: Rotation is a 3x3 matrix
  - It represents 3 angles… with 9 numbers – why so many???



$$\mathbf{R} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix}$$

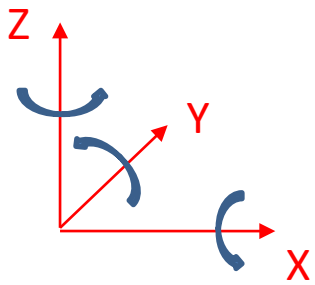
Rotations in 3D

- A 3D rotation has only 3 degrees of freedom
  - I.e., it takes 3 numbers to describe the orientation of an object in the world
  - Think of “roll”, “pitch”, “yaw” for an airplane



XYZ angles to represent rotations

- One way to represent a 3D rotation is by doing successive rotations about the X,Y, and Z axes
- BUT...



XYZ angles to represent rotations

The result depends on the order in which the transforms are applied; i.e., XYZ or ZYX

Easy demo: Hold your phone in front of you facing you.

Rotate around Z 90* then around X 90*
Reset
Rotate around X 90* then around Z 90*

XYZ angles to represent rotations

Some orientations can be represented by multiple XYZ angles

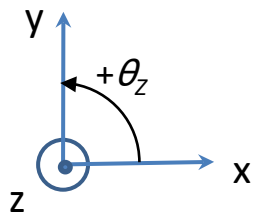
(X=0, Y=90, Z=0) vs (X=90, Y=90, Z=90)

Alternative

Instead of representing orientation as three 2D rotation **angles**, we'll describe an orientation as a matrix composed from three **3x3** 2D rotation **matrices**

XYZ Angles

- Rotation about the Z axis

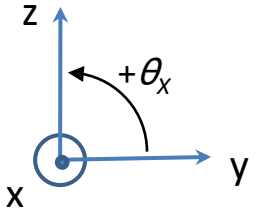


⊙ Points toward me

⊗ Points away from me

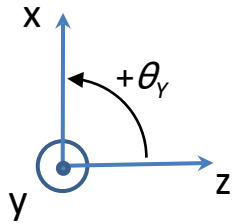
$$\begin{pmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \end{pmatrix} = \begin{pmatrix} \cos\theta_Z & -\sin\theta_Z & 0 \\ \sin\theta_Z & \cos\theta_Z & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \end{pmatrix}$$

- Rotation about the X axis



$$\begin{pmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_X & -\sin\theta_X \\ 0 & \sin\theta_X & \cos\theta_X \end{pmatrix} \begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \end{pmatrix}$$

- Rotation about the Y axis



$$\begin{pmatrix} {}^{B}x \\ {}^{B}y \\ {}^{B}z \end{pmatrix} = \begin{pmatrix} \cos\theta_Y & 0 & \sin\theta_Y \\ 0 & 1 & 0 \\ -\sin\theta_Y & 0 & \cos\theta_Y \end{pmatrix} \begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \end{pmatrix}$$

Note signs are different than in the other cases

3D Rotation Matrix

- We can concatenate the 3 rotations to yield a single 3x3 rotation matrix; e.g.,

$$
{}^{A}_{B}R_{XYZ}(\theta_X,\theta_Y,\theta_Z) = R_Z(\theta_Z)\, R_Y(\theta_Y)\, R_X(\theta_X)
$$

$$
= \begin{pmatrix} cz & -sz & 0 \\ sz & cz & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} cy & 0 & sy \\ 0 & 1 & 0 \\ -sy & 0 & cy \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & cx & -sx \\ 0 & sx & cx \end{pmatrix}
$$

where

$$
cx = \cos(\theta_X),\ sy = \sin(\theta_Y),\ \text{etc}
$$

Result: A unique matrix for each orientation!

- Note: we use the convention that to rotate a vector, we pre-multiply it; i.e., **v'** = **R v**
  - This means that if **R** = $\mathbf{R}_Z\, \mathbf{R}_Y\, \mathbf{R}_X$, we actually apply the X rotation first, then the Y rotation, then the Z rotation

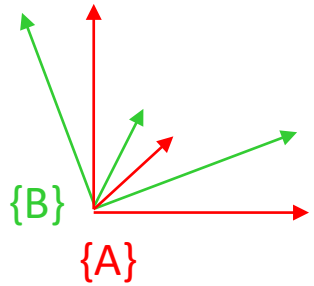
3D Rotation Matrix

- **R** can represent a rotational transformation of one frame to another

$$ {}^{B}_{A}\mathbf{R} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} $$

- We can rotate a vector represented in frame *A* to obtain its representation in frame *B*

$$ {}^{B}\mathbf{v} = {}^{B}_{A}\mathbf{R} \; {}^{A}\mathbf{v} $$



- Note: as in 2D, rotation matrices are orthonormal so the inverse of a rotation matrix is just its transpose

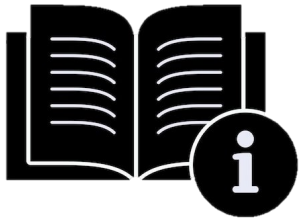
$$ \left({}^{B}_{A}\mathbf{R}\right)^{-1} = \left({}^{B}_{A}\mathbf{R}\right)^{T} = {}^{A}_{B}\mathbf{R} $$

Notation

- For vectors, such as this, the leading superscript represents the coordinate frame that the vector is expressed in
- For transforms, such as this, this matrix represents a rotational transformation of frame *A* to frame *B*
  - The leading subscript indicates “from”
  - The leading superscript indicates “to”

$$ {}^{A}\mathbf{v} = \begin{pmatrix} {}^{A}x \\ {}^{A}y \\ {}^{A}z \end{pmatrix} $$

$$ {}^{B}_{A}\mathbf{R} = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} $$



3D Rotation Matrix

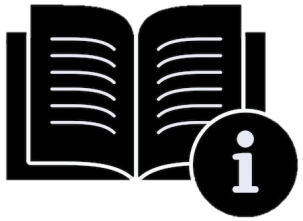
- The elements of **R** are direction cosines (the projections of unit vectors from one frame onto the unit vectors of the other frame)
- The columns of **R** are the unit vectors of A, expressed in the B frame
- The rows of **R** are the unit vectors of {B} expressed in {A}

$$ {}^{B}_{A}\mathbf{R} = \begin{pmatrix} \hat{\mathbf{x}}_A \cdot \hat{\mathbf{x}}_B & \hat{\mathbf{y}}_A \cdot \hat{\mathbf{x}}_B & \hat{\mathbf{z}}_A \cdot \hat{\mathbf{x}}_B \\ \hat{\mathbf{x}}_A \cdot \hat{\mathbf{y}}_B & \hat{\mathbf{y}}_A \cdot \hat{\mathbf{y}}_B & \hat{\mathbf{z}}_A \cdot \hat{\mathbf{y}}_B \\ \hat{\mathbf{x}}_A \cdot \hat{\mathbf{z}}_B & \hat{\mathbf{y}}_A \cdot \hat{\mathbf{z}}_B & \hat{\mathbf{z}}_A \cdot \hat{\mathbf{z}}_B \end{pmatrix} $$

$$ {}^{B}_{A}\mathbf{R} = \begin{pmatrix} \begin{pmatrix} {}^{B}\hat{\mathbf{x}}_A \end{pmatrix} & \begin{pmatrix} {}^{B}\hat{\mathbf{y}}_A \end{pmatrix} & \begin{pmatrix} {}^{B}\hat{\mathbf{z}}_A \end{pmatrix} \end{pmatrix} $$

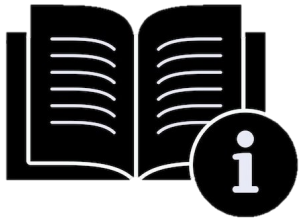
$$ {}^{B}_{A}\mathbf{R} = \begin{pmatrix} \begin{pmatrix} {}^{A}\hat{\mathbf{x}}_B{}^{T} \\ {}^{A}\hat{\mathbf{y}}_B{}^{T} \\ {}^{A}\hat{\mathbf{z}}_B{}^{T} \end{pmatrix} \end{pmatrix} $$

$$ {}^{B}_{A}R\,{}^{A}\hat{x}_A = \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} r_{11} \\ r_{21} \\ r_{31} \end{pmatrix} = {}^{B}\hat{x}_A $$



Python: Creating a Rotation Matrix

```python
import numpy as np

ax, ay, az = 0.1, -0.2, 0.3 # radians

sx, sy, sz = np.sin(ax), np.sin(ay), np.sin(az)
cx, cy, cz = np.cos(ax), np.cos(ax), np.cos(az)

Rx = np.array(((1, 0, 0), (0, cx, -sx), (0, sx, cx)))
Ry = np.array(((cy, 0, sy), (0, 1, 0), (-sy, 0, cy)))
Rz = np.array(((cz, -sz, 0), (sz, cz, 0), (0, 0, 1)))

# Apply X rotation first, then Y, then Z
R = Rz @ Ry @ Rx    # Use @ for matrix mult
print(R)



# Apply Z rotation first, then Y, then X
R = Rx @ Ry @ Rz
print(R)
```

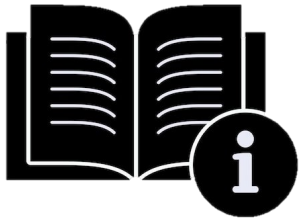


Python: Creating a Rotation Matrix

```python
import numpy as np

ax, ay, az = 0.1, -0.2, 0.3 # radians

sx, sy, sz = np.sin(ax), np.sin(ay), np.sin(az)
cx, cy, cz = np.cos(ax), np.cos(ax), np.cos(az)

Rx = np.array(((1, 0, 0), (0, cx, -sx), (0, sx, cx)))
Ry = np.array(((cy, 0, sy), (0, 1, 0), (-sy, 0, cy)))
Rz = np.array(((cz, -sz, 0), (sz, cz, 0), (0, 0, 1)))

# Apply X rotation first, then Y, then Z
R = Rz @ Ry @ Rx    # Use @ for matrix mult
print(R)
```

```
[[ 0.95056379 -0.31299183 -0.15934508]
 [ 0.29404384  0.94470249 -0.153792  ]
 [ 0.19866933  0.09933467  0.99003329]]
```

```python
# Apply Z rotation first, then Y, then X
R = Rx @ Ry @ Rz
print(R)
```



Python: Creating a Rotation Matrix

```python
import numpy as np

ax, ay, az = 0.1, -0.2, 0.3 # radians

sx, sy, sz = np.sin(ax), np.sin(ay), np.sin(az)
cx, cy, cz = np.cos(ax), np.cos(ax), np.cos(az)

Rx = np.array(((1, 0, 0), (0, cx, -sx), (0, sx, cx)))
Ry = np.array(((cy, 0, sy), (0, 1, 0), (-sy, 0, cy)))
Rz = np.array(((cz, -sz, 0), (sz, cz, 0), (0, 0, 1)))

# Apply X rotation first, then Y, then Z
R = Rz @ Ry @ Rx    # Use @ for matrix mult
print(R)
```

```
[[ 0.95056379 -0.31299183 -0.15934508]
 [ 0.29404384  0.94470249 -0.153792  ]
 [ 0.19866933  0.09933467  0.99003329]]
```

```python
# Apply Z rotation first, then Y, then X
R = Rx @ Ry @ Rz
print(R)
```

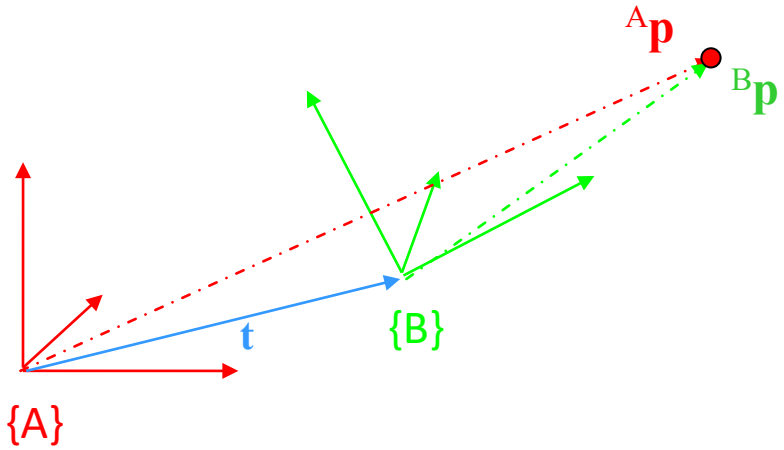
```
[[ 0.95056379 -0.29404384 -0.19866933]
 [ 0.27509585  0.95642509 -0.09933467]
 [ 0.21835066  0.03695701  0.99003329]]
```

Transforming a Point

- We can use **R,t** to transform a point from coordinate frame {B} to frame {A}

$$ {}^{A}\mathbf{p} = {}^{A}_{B}\mathbf{R}\ {}^{B}\mathbf{p} + \mathbf{t} $$

- Where
  - ${}^{A}\mathbf{p}$ is the representation of **p** in frame {A}
  - ${}^{B}\mathbf{p}$ is the representation of **p** in frame {B}



- Note

$\mathbf{t}$ is the translation of B's origin in the A frame, ${}^{A}\mathbf{t}_{B}$

Homogeneous Coordinates

- We can represent the transformation with a single matrix multiplication if we write **p** in homogeneous coordinates
  - This simply means to append a 1 as a 4th element
  - If the 4th element ever becomes ≠ 1, we divide through by it
- Then

The leading superscript indicates what coordinate frame the point is represented in

$$ {}^{A}\mathbf{p} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} sx \\ sy \\ sz \\ s \end{pmatrix} $$

$$ {}^{A}\mathbf{p} = {}^{A}_{B}\mathbf{H}\ {}^{B}\mathbf{p} \quad \text{where} \quad {}^{A}_{B}\mathbf{H} = \begin{bmatrix} {}^{A}_{B}\mathbf{R} & {}^{A}\mathbf{t}_{B} \\ 0\ 0\ 0 & 1 \end{bmatrix} $$

Notation Note: Cancel leading subscript with trailing superscript

Example

- In coordinate frame A, point **p** is (-1,0,1)
- Frame *A* is located at (1,2,4) with respect to B, and is rotated 90 degrees about the x axis with respect to frame *B*
- What is point **p** in frame *B*?

Example

- In coordinate frame A, point **p** is (-1,0,1)
- Frame *A* is located at (1,2,4) with respect to B, and is rotated 90 degrees about the x axis with respect to frame *B*
- What is point **p** in frame *B*?

We want to do

$$ {}^{B}\mathbf{p} = {}^{B}_{A}\mathbf{H} \; {}^{A}\mathbf{p} $$

Example

- In coordinate frame A, point **p** is (-1,0,1)
- Frame *A* is located at (1,2,4) with respect to B, and is rotated 90 degrees about the x axis with respect to frame *B*
- What is point **p** in frame *B*?

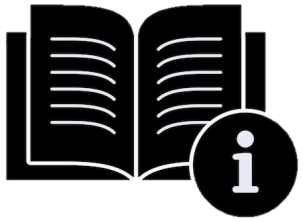
We want to do ${}^{B}\mathbf{p} = {}^{B}_{A}\mathbf{H}\ {}^{A}\mathbf{p}$ where ${}^{B}_{A}\mathbf{H} = \begin{bmatrix} {}^{B}_{A}\mathbf{R} & {}^{B}\mathbf{t}_{A} \\ 0\ 0\ 0 & 1 \end{bmatrix}$

Example

- In coordinate frame A, point **p** is (-1,0,1)
- Frame *A* is located at (1,2,4) with respect to B, and is rotated 90 degrees about the x axis with respect to frame *B*
- What is point **p** in frame *B*?

We want to do ${}^{B}\mathbf{p} = {}^{B}_{A}\mathbf{H}\ {}^{A}\mathbf{p}$ where ${}^{B}_{A}\mathbf{H} = \begin{bmatrix} {}^{B}_{A}\mathbf{R} & {}^{B}\mathbf{t}_{A} \\ 0\ 0\ 0 & 1 \end{bmatrix}$

$${}^{B}_{A}\mathbf{R} = \mathbf{R}_x(90) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(90) & -\sin(90) \\ 0 & \sin(90) & \cos(90) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$



Python and Numpy: Transforming a point

```
# Construct 4x4 transformation matrix to transform A to B

# Rotation matrix of A with respect to B.
R_A_B = np.array(((1,0,0),(0,0,-1),(0,1,0)))    # Get 3x3 matrix

# The translation is the origin of A in B.
tAorg_B = np.array([[1,2,4]]).T   # Get as a 3x1 matrix

# H_A_B means transform A to B.
H_A_B = np.block([[R_A_B, tAorg_B], [0,0,0,1]])  # Get 4x4 matrix

# Define a point in the A frame, as [x,y,z,1].
P_A = np.array([[-1,0,1,1]]).T    # Get as a 4x1 matrix

# Convert point to B frame.
P_B = H_A_B @ P_A
```

Inverse Transformations

- The **matrix inverse** is the inverse transformation

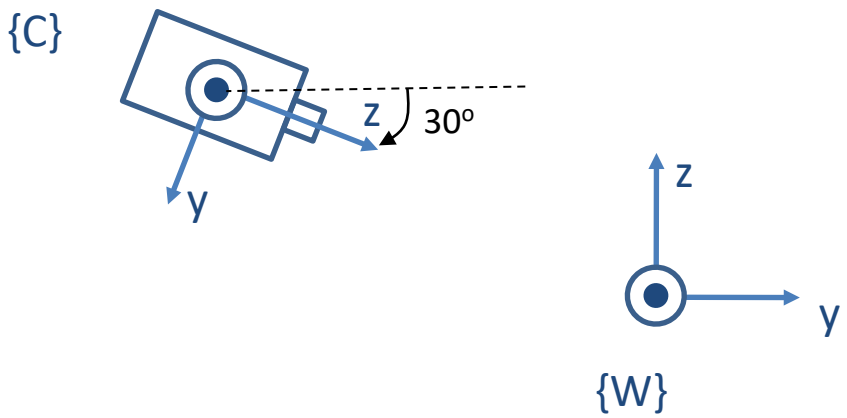
$$ {}^{A}_{B}\mathbf{H} = \left({}^{B}_{A}\mathbf{H} \right)^{-1} $$

- Note – unlike rotation matrices, the inverse of a full 4x4 homogeneous transformation matrix is **not the transpose**

$$ {}^{A}_{B}\mathbf{H} \neq \left({}^{B}_{A}\mathbf{H} \right)^{T} $$

Example

- A camera is located at point (0,-5,3) with respect to the world. The camera is tilted down by 30 degrees from the horizontal. Find the transformation from {W} to {C}. (Note that in "the world" Z is up (X-Y ground plane) but in "the camera", Z is out (X-Y image plane)!)



Example (continued)

- The origin of the camera in the world is ${}^{W}\mathbf{t}_{C} = (0, -5, 3)^{T}$
- The rotation matrix is ${}^{W}_{C}\mathbf{R} = \mathbf{R}_{x}(-120\ deg)$

$$ {}^{W}_{C}\mathbf{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(-120) & -\sin(-120) \\ 0 & \sin(-120) & \cos(-120) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.5 & +0.86 \\ 0 & -0.86 & -0.5 \end{bmatrix} $$

- To check to see if this makes sense … see if the unit axes of the camera are pointing in the right direction in the world

$$ {}^{W}_{C}\mathbf{R} = \left(\begin{pmatrix} \\ {}^{W}\hat{\mathbf{x}}_{C} \\ \\ \end{pmatrix} \begin{pmatrix} \\ {}^{W}\hat{\mathbf{y}}_{C} \\ \\ \end{pmatrix} \begin{pmatrix} \\ {}^{W}\hat{\mathbf{z}}_{C} \\ \\ \end{pmatrix} \right) \Rightarrow {}^{W}\hat{\mathbf{x}}_{C} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \; {}^{W}\hat{\mathbf{y}}_{C} = \begin{pmatrix} 0 \\ -0.5 \\ -0.86 \end{pmatrix}, \; {}^{W}\hat{\mathbf{z}}_{C} = \begin{pmatrix} 0 \\ 0.86 \\ -0.5 \end{pmatrix} $$

- The full 4x4 homogeneous transformation matrix from camera to world is

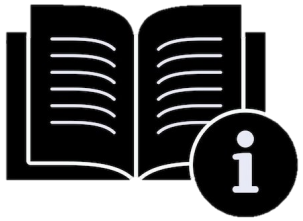
$$ {}^{W}_{C}\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -0.5 & 0.86 & -5 \\ 0 & -0.86 & -0.5 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

- But, we actually wanted the transformation from world to camera, so take the inverse

$$ {}^{C}_{W}\mathbf{H} = {}^{W}_{C}\mathbf{H}^{-1} $$

```
    1.0000         0         0         0
         0   -0.5000   -0.8660    0.0981
         0    0.8660   -0.5000    5.8301
         0         0         0    1.0000
```

- We can visualize the relationships between this coordinate frame and our world origin by plotting them in 3D…



```
# Draw scene in 3D.
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

draw_coordinate_axes(ax, np.eye(4), 'W')
draw_coordinate_axes(ax, H_v_w, 'V')
draw_coordinate_axes(ax, H_m_w, 'M')
draw_coordinate_axes(ax, H_c_w, 'C')

plt.show()    # This shows the plot, and pauses until you close the figure
```

```
# Draw three 3D line segments, representing xyz unit axes, onto the axis figure ax.
# H is the 4x4 transformation matrix representing the pose of the coordinate frame.
def draw_coordinate_axes(ax, H, label):
            p = H[0:3, 3]              # Origin of the coordinate frame
            ux = H @ np.array([1,0,0,1])            # Tip of the x axis
            uy = H @ np.array([0,1,0,1])            # Tip of the y axis
            uz = H @ np.array([0,0,1,1])            # Tip of the z axis
            ax.plot(xs=[p[0], ux[0]], ys=[p[1], ux[1]], zs=[p[2], ux[2]], c='r')            # x
axis
            ax.plot(xs=[p[0], uy[0]], ys=[p[1], uy[1]], zs=[p[2], uy[2]], c='g')            # y
axis
            ax.plot(xs=[p[0], uz[0]], ys=[p[1], uz[1]], zs=[p[2], uz[2]], c='b')            # z
axis
            ax.text(p[0], p[1], p[2], label)        # Also draw the label of the coordinate frame
```

Example (continued)

- We can take a point in the world and calculate its position with respect to the camera

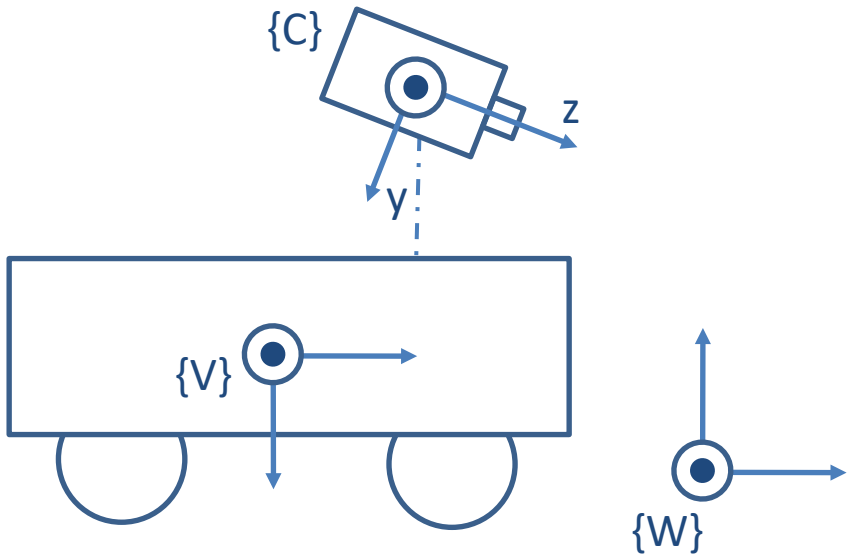
$$ {}^{C}\mathbf{p} = {}^{C}_{W}\mathbf{H} \; {}^{W}\mathbf{p} $$

- For example, transform the point ${}^{W}\mathbf{p} = (0,0,1)^{T}$

```
P_W = np.array([[0,0,1,1]]).T    # Get as a 4x1 matrix
P_C = H_W_C @ P_W        # Convert point to C frame

         0
   -0.7679
    5.3301
    1.0000
```

Combining Transforms

- Take the example of a camera mounted on a moving vehicle
- We have the transformation from the camera frame to the vehicle ${}^{V}_{C}\mathbf{H}$, and the vehicle to the world ${}^{W}_{V}\mathbf{H}$



- We can calculate the transformation from the camera frame to the world :

$$ {}^{W}_{C}\mathbf{H} = {}^{W}_{V}\mathbf{H}\ {}^{V}_{C}\mathbf{H} $$

Equivalent Angle-Axis

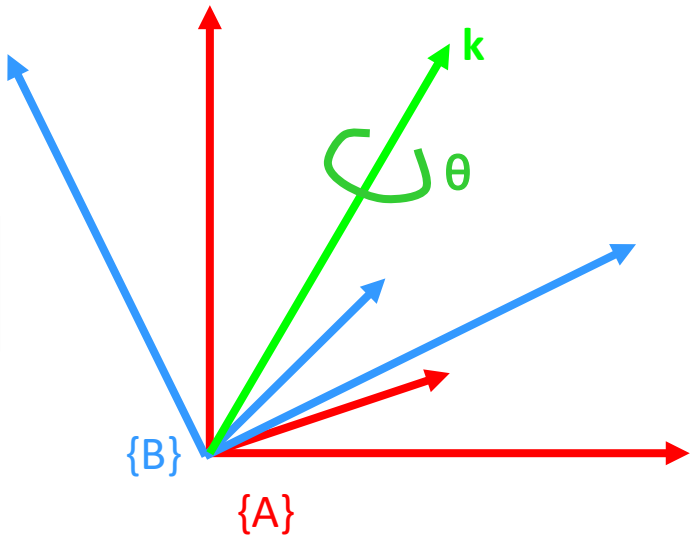
- A general rotation can be expressed as a rotation θ about an axis **k**

$$R_k(\theta) = \begin{pmatrix} k_x k_x v\theta + c\theta & k_x k_y v\theta - k_z s\theta & k_x k_z v\theta + k_y s\theta \\ k_x k_y v\theta + k_z s\theta & k_y k_y v\theta + c\theta & k_y k_z v\theta - k_x s\theta \\ k_x k_z v\theta - k_y s\theta & k_y k_z v\theta + k_x s\theta & k_z k_z v\theta + c\theta \end{pmatrix}$$

where

$$c\theta = \cos\theta,\ s\theta = \sin\theta,\ v\theta = 1 - \cos\theta$$

$$\hat{\mathbf{k}} = (k_x, k_y, k_z)^T$$



- The inverse solution (i.e., given a rotation matrix, find **k** and θ):

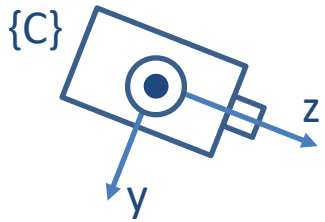
$$\theta = \operatorname{acos}\left(\frac{r_{11} + r_{22} + r_{33} - 1}{2}\right)$$

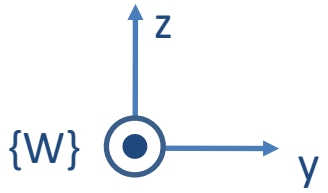
$$\hat{\mathbf{k}} = \frac{1}{2\sin\theta}\begin{pmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{pmatrix}$$

- The product of the unit vector **k** and angle θ , **ω** = θ **k** = (ω_x, ω_y, ω_z) is a minimal representation for a 3D rotation

*Note that (-**k**,-θ) is also a solution*

Repeating the previous example

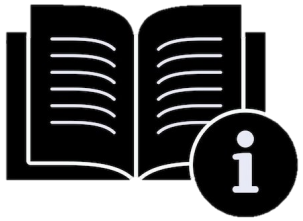





```
k = np.array([1,0,0])
az = np.deg2rad(-120)
R = axis_angle_to_rot(k, az)
```

R:

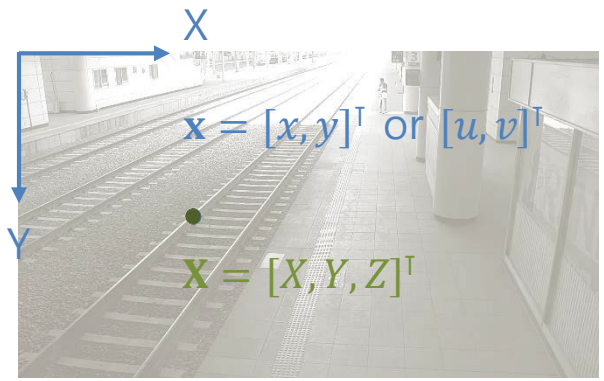
```
[[ 1.      0.      0.     ]
 [ 0.     -0.5     0.8660254]
 [ 0.     -0.8660254 -0.5    ]]
```



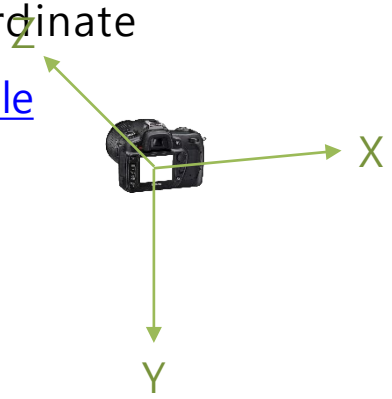
```
def axis_angle_to_rot(k, ang):
        # Convert orientation from axis, angle to rotation matrix.
        # Angle is represented in radians.  Returns 3x3 rotation matrix.
        kx = k[0]
        ky = k[1]
        kz = k[2]
        ca = math.cos(ang)
        sa = math.sin(ang)
        va = 1 - ca
        return np.array([
        [kx*kx*va + ca,    kx*ky*va - kz*sa,  kx*kz*va + ky*sa],
        [kx*ky*va + kz*sa, ky*ky*va + ca,     ky*kz*va - kx*sa],
        [kx*kz*va - ky*sa, ky*kz*va + kx*sa,  kz*kz*va + ca ]])
```

Summary of Rotation Matrices

- **Image coordinate** (unit: [pixel])



- **Camera coordinate** (unit: [meter])
  - Position: Focal point
  - X/Y direction: Image coordinate
  - Z direction: Right-hand rule



Summary of Rotation Matrices

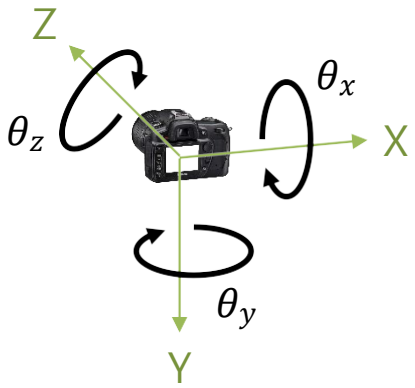
- **2D rotation matrix**
  - Rotational direction: Right-hand rule

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

Properties of a rotation matrix

- $\mathrm{R}^{-1} = \mathrm{R}^{\mathrm{T}}$ (orthogonal matrix)
- $\det(\mathrm{R}) = 1$

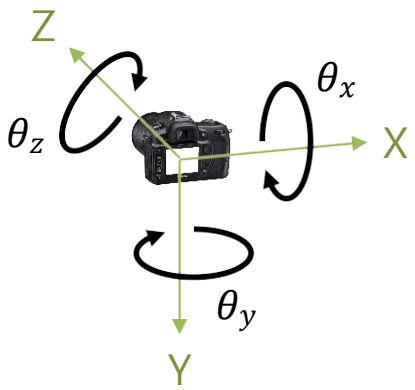
- **3D rotation matrix**
  - Rotational direction: Right-hand rule

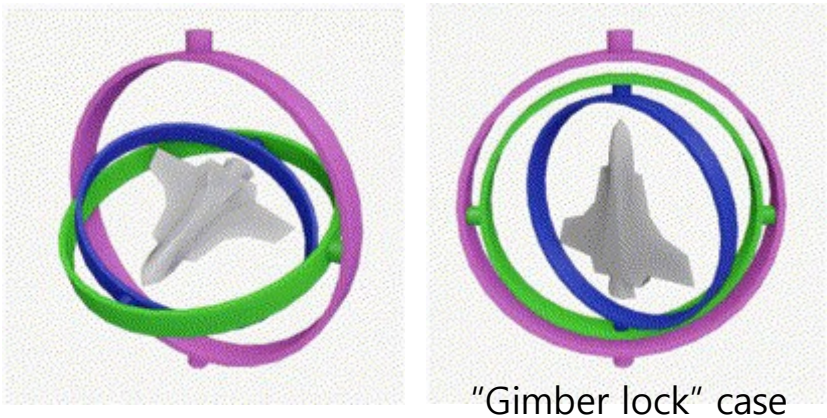


	Cameras	Vehicles	Airplanes	Telescopes
θ_x	Tilt	Pitch	Attitude	Elevation
θ_y	Pan	Yaw	Heading	Azimuth
θ_z	Roll	Roll	Bank	Horizon

Summary of Rotation Matrices

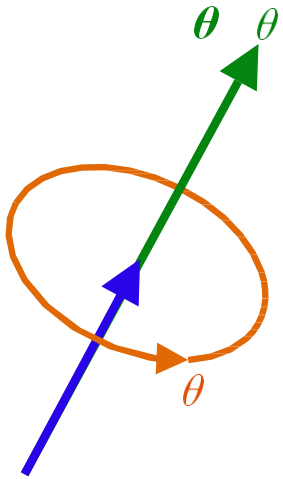
- **3D rotation representation** (3 DOF)
  - **3D rotation matrix** (9 parameters)
    - Notation: 3x3 matrix
      - e.g. $\mathrm{R} = \mathrm{R}_z(\theta_z)\,\mathrm{R}_y(\theta_y)\,\mathrm{R}_x(\theta_x) = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix}$
    - Properties: $\mathrm{R}^{-1} = \mathrm{R}^{\mathrm{T}}$ (orthogonal matrix), $\det(\mathrm{R}) = 1$
  - **Euler angle** (3 parameters)
    - Notation: $[\theta_x, \theta_y, \theta_z]$
    - Issues: Not unique, not continuous, Gimbar lock (loss of DOF)





"Gimber lock" case

Summary of Rotation Matrices

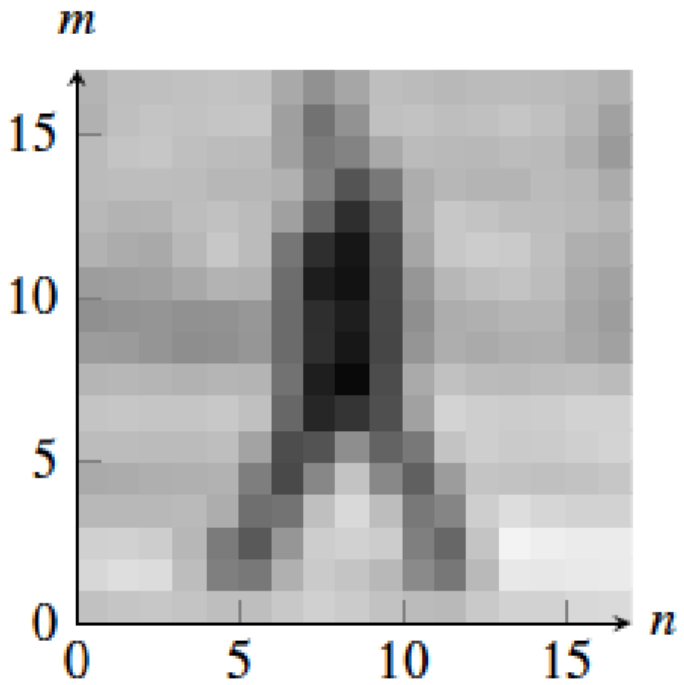


- **3D rotation representation** (3 DOF)
  - **Axis-angle representation** (3 parameters; a.k.a. *rotation vector*, *Rodrigues notation*)
    - Notation: $\mathbf{\theta} = \theta\mathbf{e}$
      - e.g. Axis (unit vector): $\mathbf{e} = [0, 0, 1]$, angle: $\theta = {}^{\pi}\mathrm{T}_2 \rightarrow \mathbf{\theta} = [0, 0, {}^{\pi}\mathrm{T}_2]$
    - Properties: Log map of SO(3), dual ($-\mathbf{e}$ with $-\theta \rightarrow \mathbf{\theta}$), reverse angle ($-\mathbf{\theta}$)
    - Note: The standard notation in OpenCV with `cv.Rodrigues()` (R ↔ rvec)
  - (Unit) **Quaternion** (4 parameters)
    - Notation: $\mathbf{q} = [q_w, q_x, q_y, q_z]$ or $[q_x, q_y, q_z, q_w]$
      - Meaning: $\mathbf{q} = \cos\frac{\theta}{2} + (e_x\mathbf{i} + e_y\mathbf{j} + e_z\mathbf{k})\sin\frac{\theta}{2}$
    - Property: ${q_x}^2 + {q_y}^2 + {q_z}^2 + {q_w}^2 = 1$, dual ($-\mathbf{q}$), reverse angle ($\mathbf{q}^*$; conjugate)

- Note: 3D rotation conversion
  - Python: `scipy.spatial.transform.Rotation`
  - Web apps: NinjaCalc, Glowbuzzer, Andre Gaschler

Pinhole Camera Model

Image as a 2D discrete signal



A tiny person of 18 x 18 pixels

Image as a 2D discrete signal

$$\mathbf{I} = \begin{bmatrix}
160 & 175 & 171 & 168 & 168 & 172 & 164 & 158 & 167 & 173 & 167 & 163 & 162 & 164 & 160 & 159 & 163 & 162 \\
149 & 164 & 172 & 175 & 178 & 179 & 176 & 118 & 97 & 168 & 175 & 171 & 169 & 175 & 176 & 177 & 165 & 152 \\
161 & 166 & 182 & 171 & 170 & 177 & 175 & 116 & 109 & 169 & 177 & 173 & 168 & 175 & 175 & 159 & 153 & 123 \\
171 & 174 & 177 & 175 & 167 & 161 & 157 & 138 & 103 & 112 & 157 & 164 & 159 & 160 & 165 & 169 & 148 & 144 \\
163 & 163 & 162 & 165 & 167 & 164 & 178 & 167 & 77 & 55 & 134 & 170 & 167 & 162 & 164 & 175 & 168 & 160 \\
173 & 164 & 158 & 165 & 180 & 180 & 150 & 89 & 61 & 34 & 137 & 186 & 186 & 182 & 175 & 165 & 160 & 164 \\
152 & 155 & 146 & 147 & 169 & 180 & 163 & 51 & 24 & 32 & 119 & 163 & 175 & 182 & 181 & 162 & 148 & 153 \\
134 & 135 & 147 & 149 & 150 & 147 & 148 & 62 & 36 & 46 & 114 & 157 & 163 & 167 & 169 & 163 & 146 & 147 \\
135 & 132 & 131 & 125 & 115 & 129 & 132 & 74 & 54 & 41 & 104 & 156 & 152 & 156 & 164 & 156 & 141 & 144 \\
151 & 155 & 151 & 145 & 144 & 149 & 143 & 71 & 31 & 29 & 129 & 164 & 157 & 155 & 159 & 158 & 156 & 148 \\
172 & 174 & 178 & 177 & 177 & 181 & 174 & 54 & 21 & 29 & 136 & 190 & 180 & 179 & 176 & 184 & 187 & 182 \\
177 & 178 & 176 & 173 & 174 & 180 & 150 & 27 & 101 & 94 & 74 & 189 & 188 & 186 & 183 & 186 & 188 & 187 \\
160 & 160 & 163 & 163 & 161 & 167 & 100 & 45 & 169 & 166 & 59 & 136 & 184 & 176 & 175 & 177 & 185 & 186 \\
147 & 150 & 153 & 155 & 160 & 155 & 56 & 111 & 182 & 180 & 104 & 84 & 168 & 172 & 171 & 164 & 168 & 167 \\
184 & 182 & 178 & 175 & 179 & 133 & 86 & 191 & 201 & 204 & 191 & 79 & 172 & 220 & 217 & 205 & 209 & 200 \\
184 & 187 & 192 & 182 & 124 & 32 & 109 & 168 & 171 & 167 & 163 & 51 & 105 & 203 & 209 & 203 & 210 & 205 \\
191 & 198 & 203 & 197 & 175 & 149 & 169 & 189 & 190 & 173 & 160 & 145 & 156 & 202 & 199 & 201 & 205 & 202 \\
153 & 149 & 153 & 155 & 173 & 182 & 179 & 177 & 182 & 177 & 182 & 185 & 179 & 177 & 167 & 176 & 182 & 180
\end{bmatrix}$$

Let's say we have a sensor…

How is the image created?

digital sensor
(CCD or CMOS)

… and an object we like to photograph

real-world object

digital sensor (CCD or CMOS)

What would an image taken like this look like?

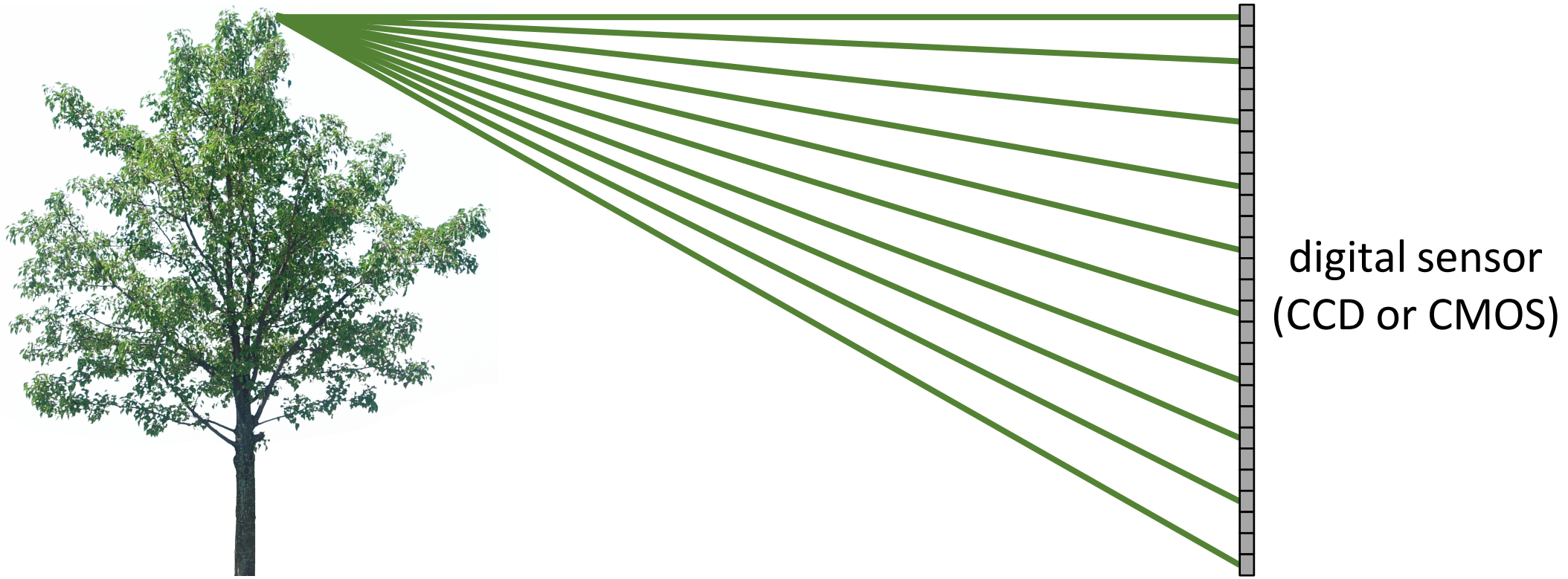
Bare-sensor imaging

real-world object

digital sensor (CCD or CMOS)

Bare-sensor imaging

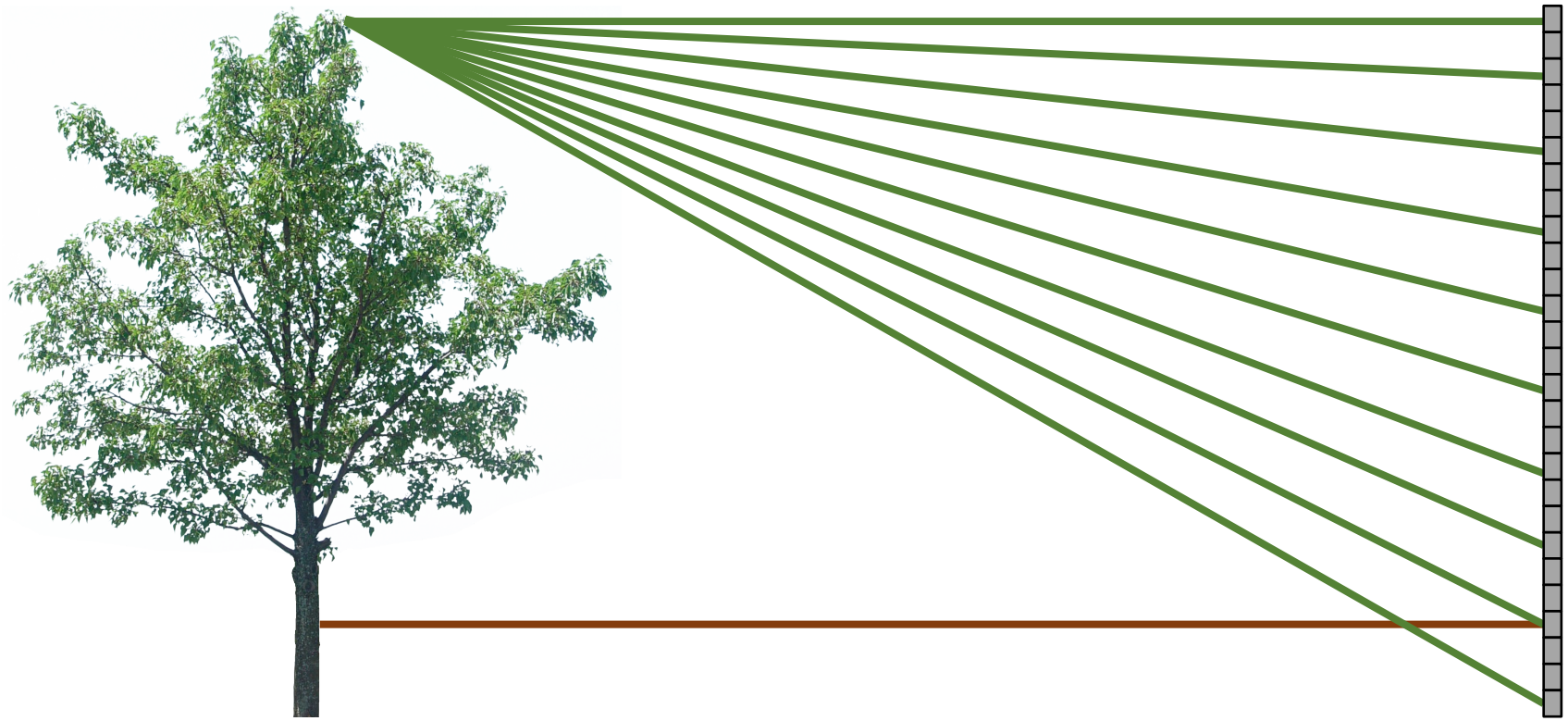
real-world object



digital sensor (CCD or CMOS)

Bare-sensor imaging

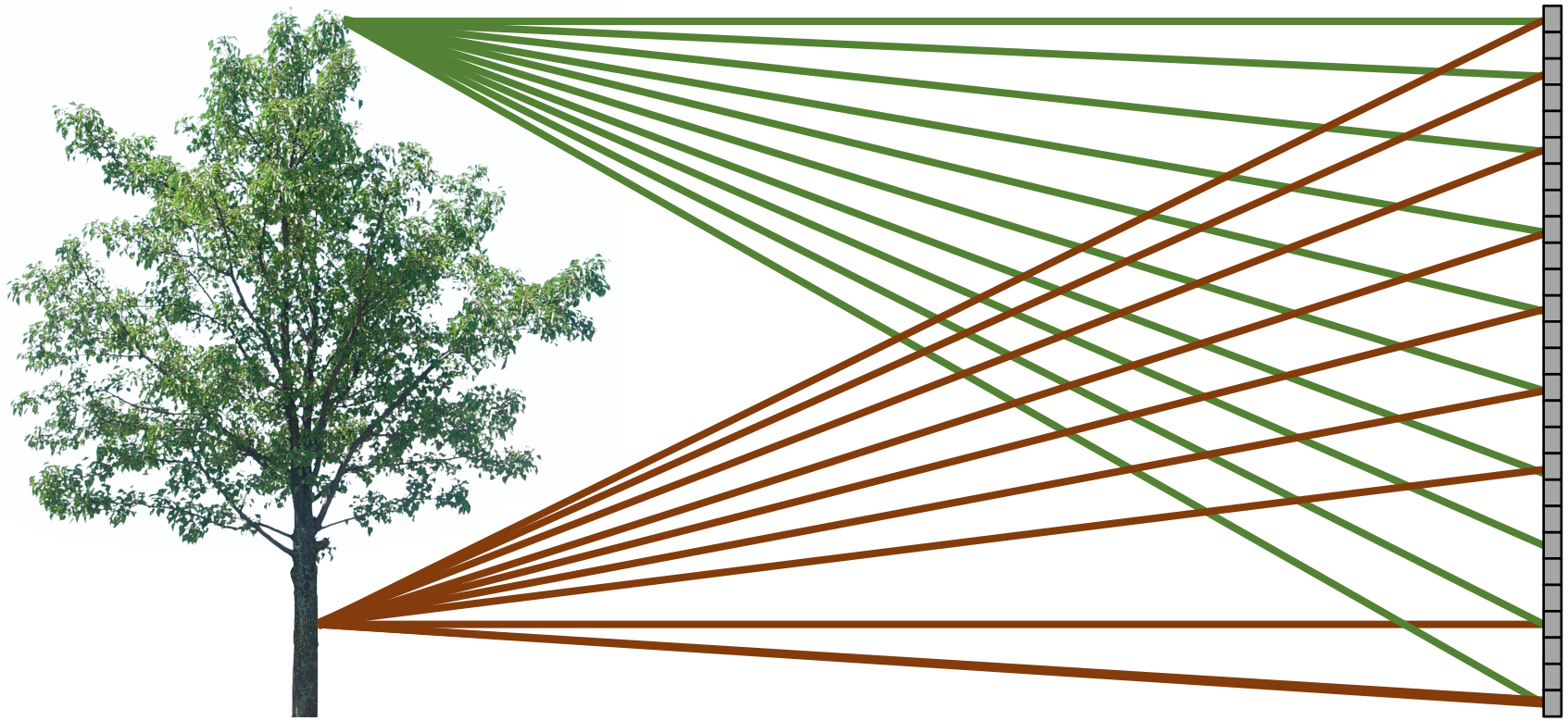
real-world object



digital sensor (CCD or CMOS)

Bare-sensor imaging

real-world object



digital sensor (CCD or CMOS)

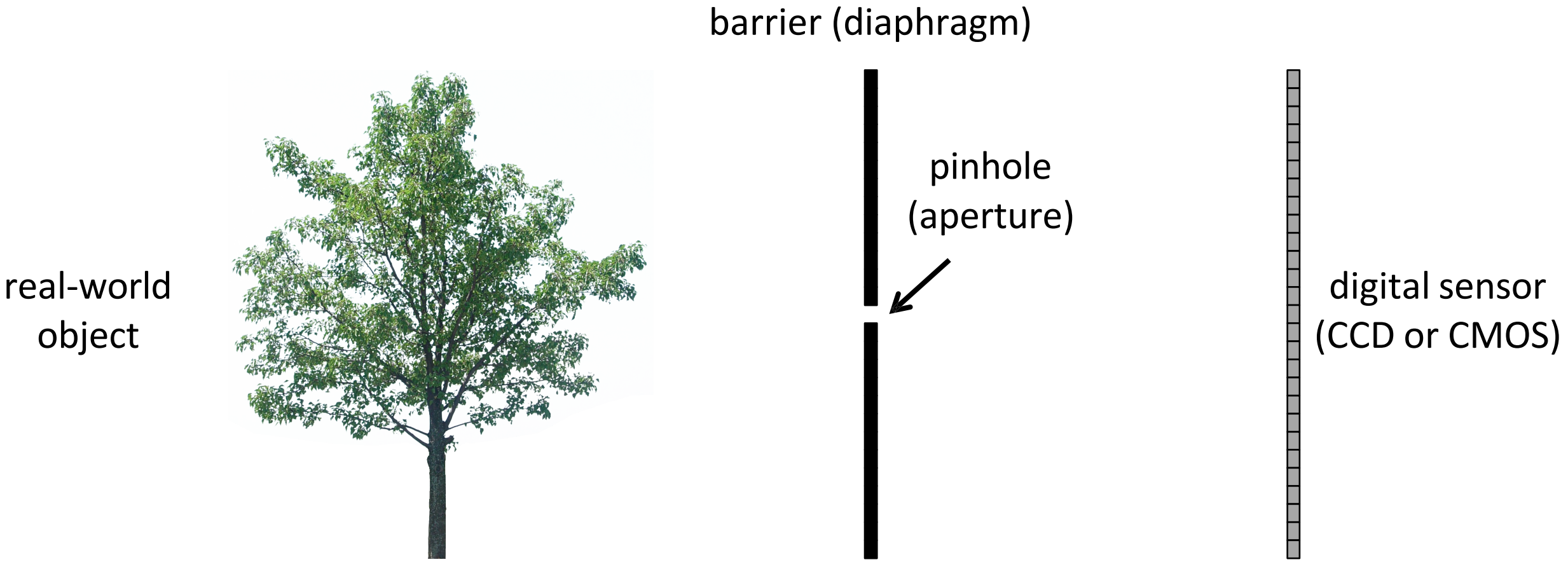
All scene points contribute to all sensor pixels

What does the image on the sensor look like?

Bare-sensor imaging

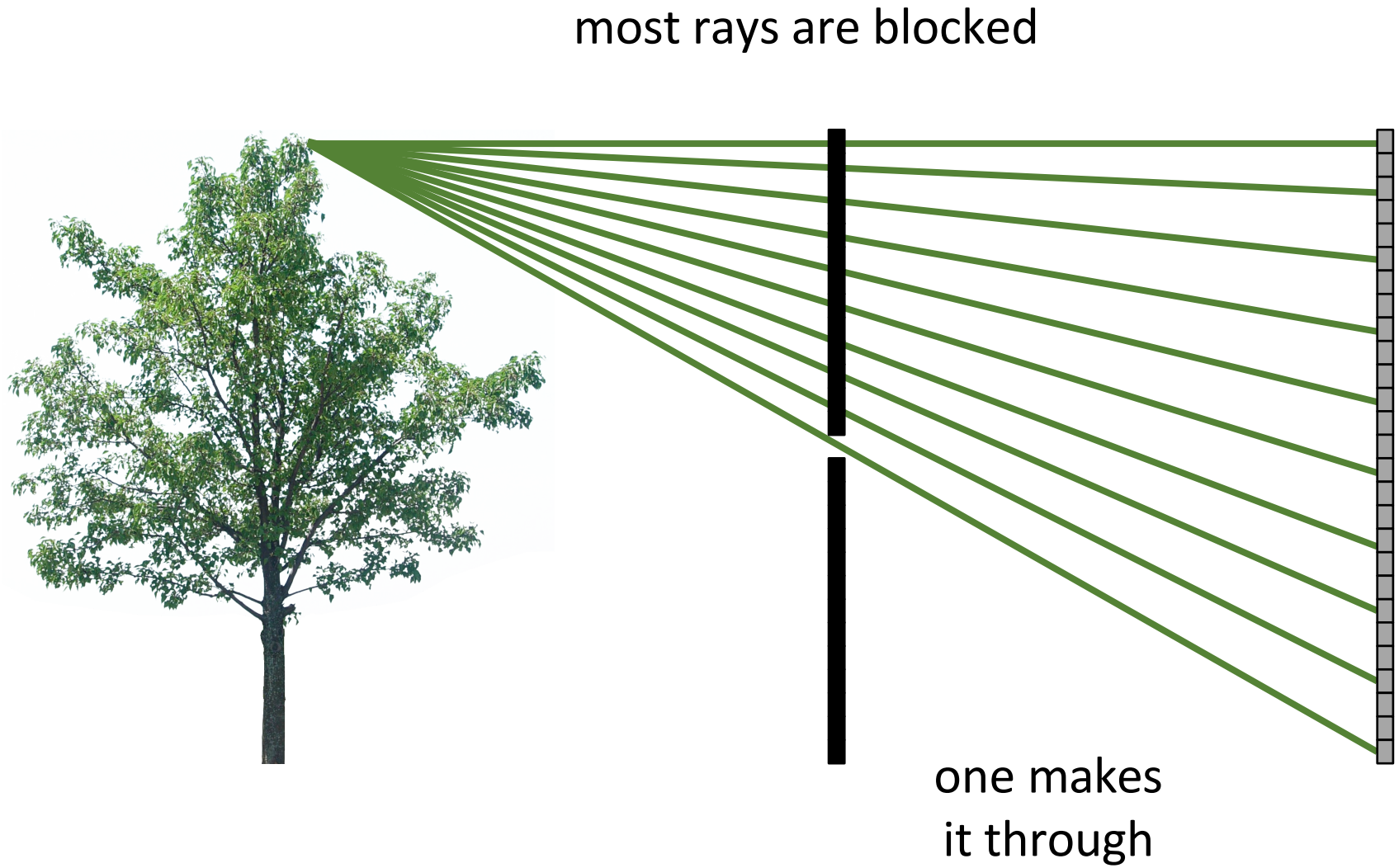
All scene points contribute to all sensor pixels

Let's add something to this scene



What would an image taken like this look like?

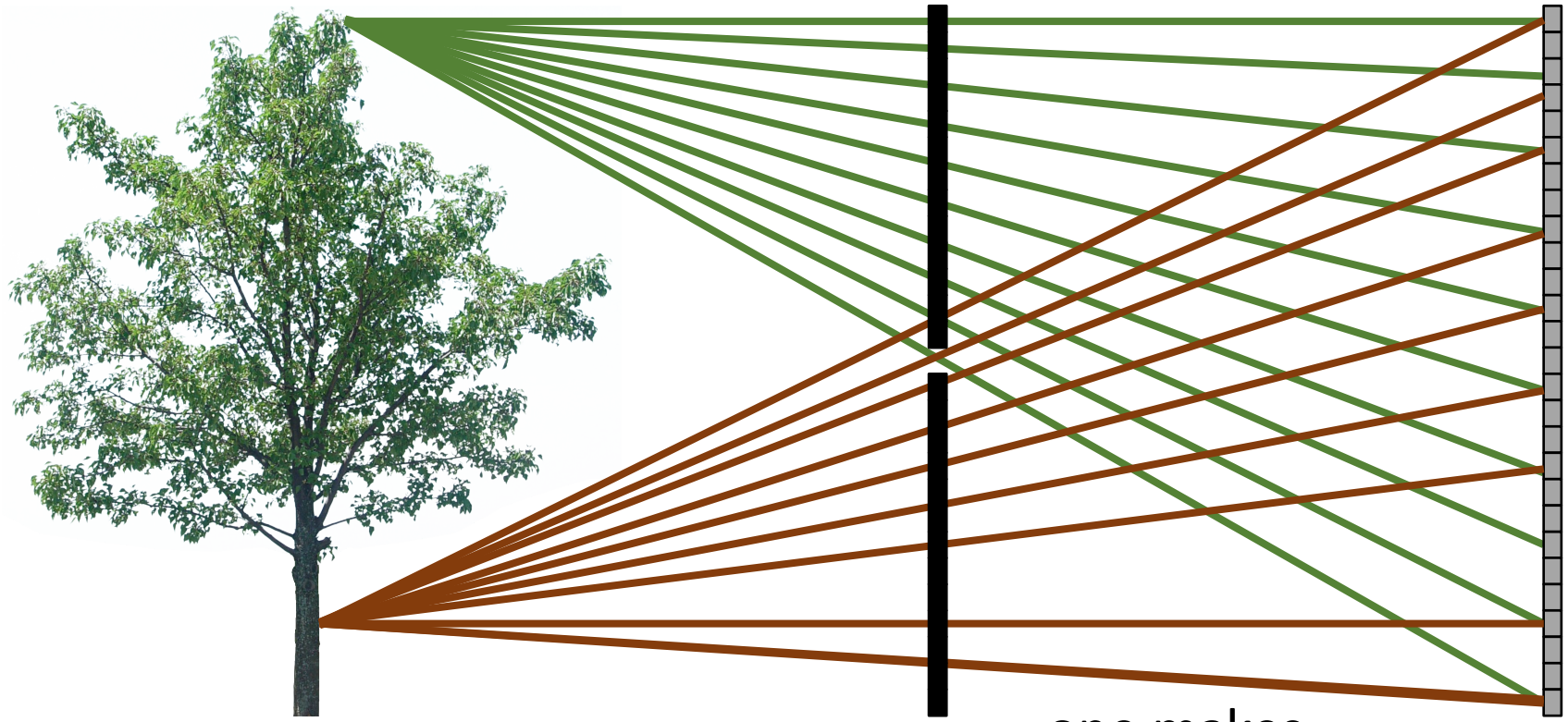
Pinhole imaging



Pinhole imaging

most rays are blocked

real-world object

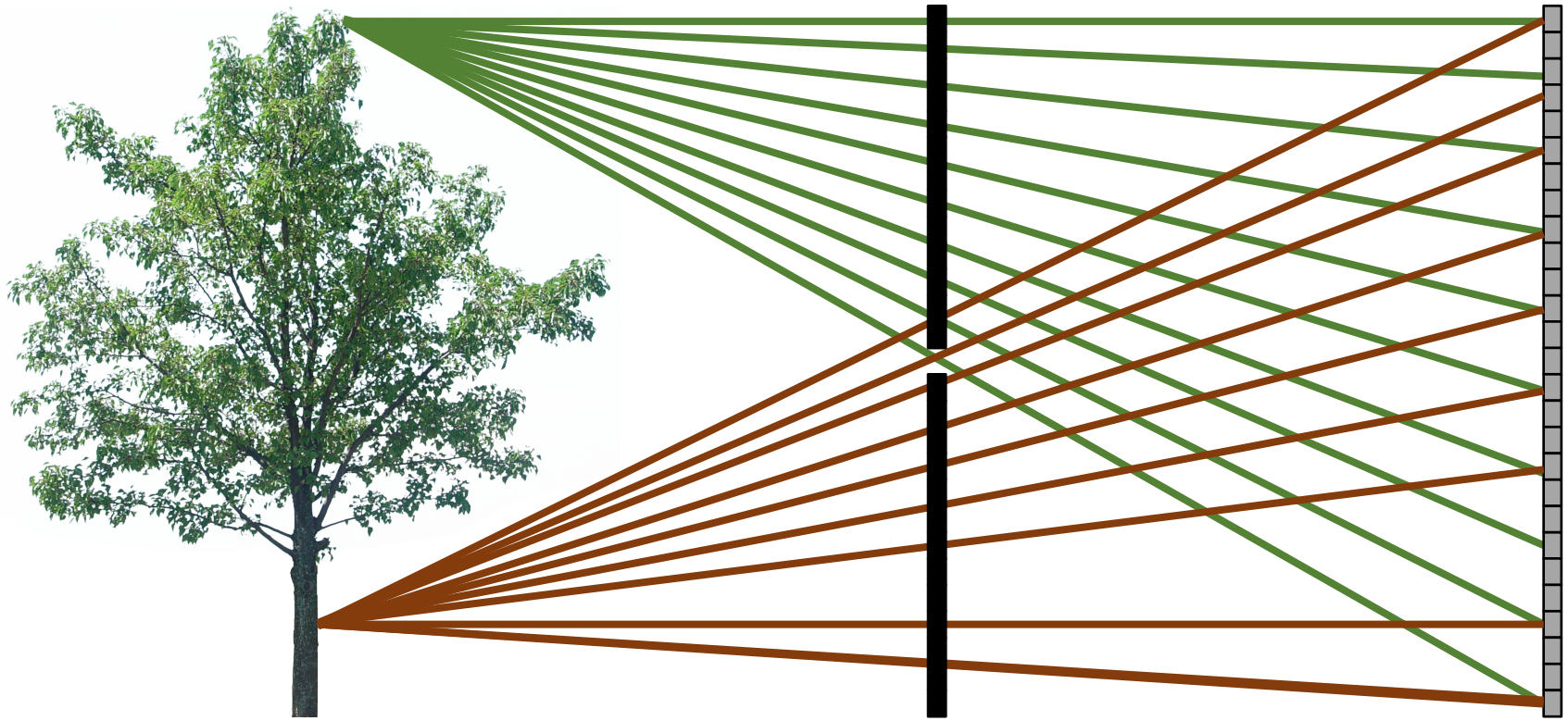


digital sensor (CCD or CMOS)

one makes it through

Pinhole imaging

real-world object



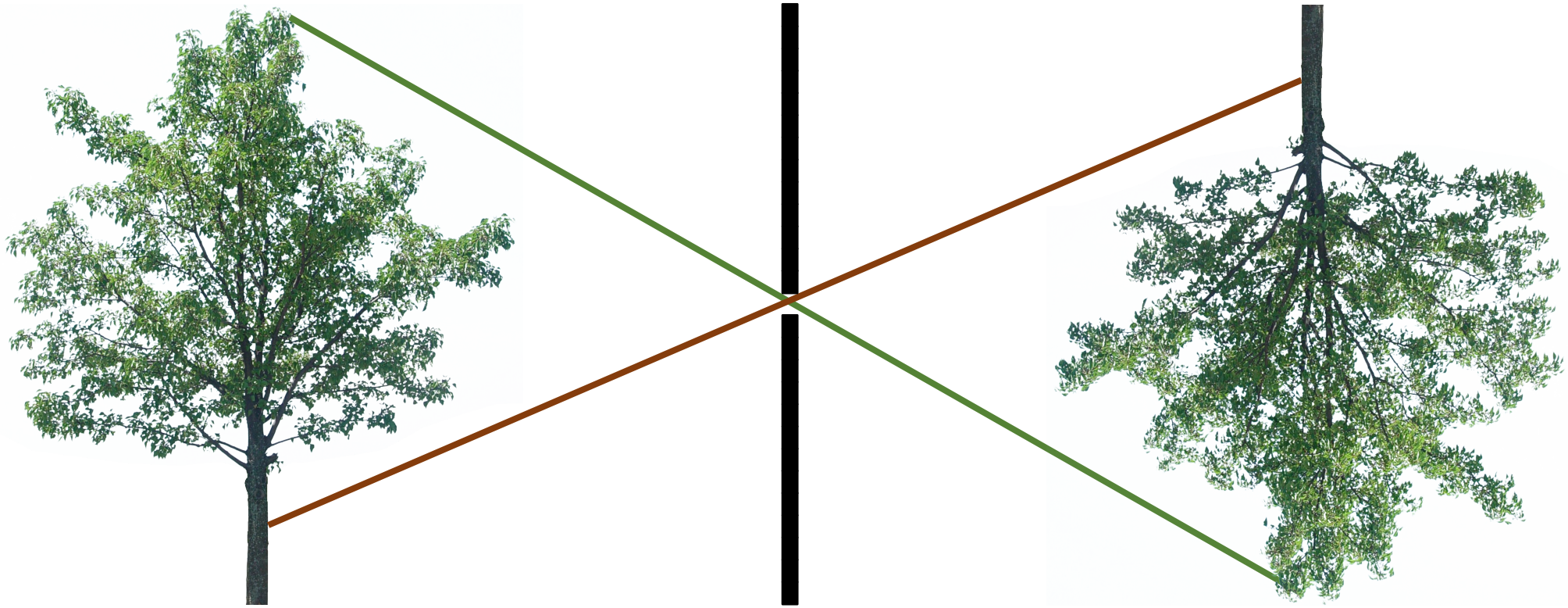
digital sensor (CCD or CMOS)

Each scene point contributes to only one sensor pixel

What does the image on the sensor look like?

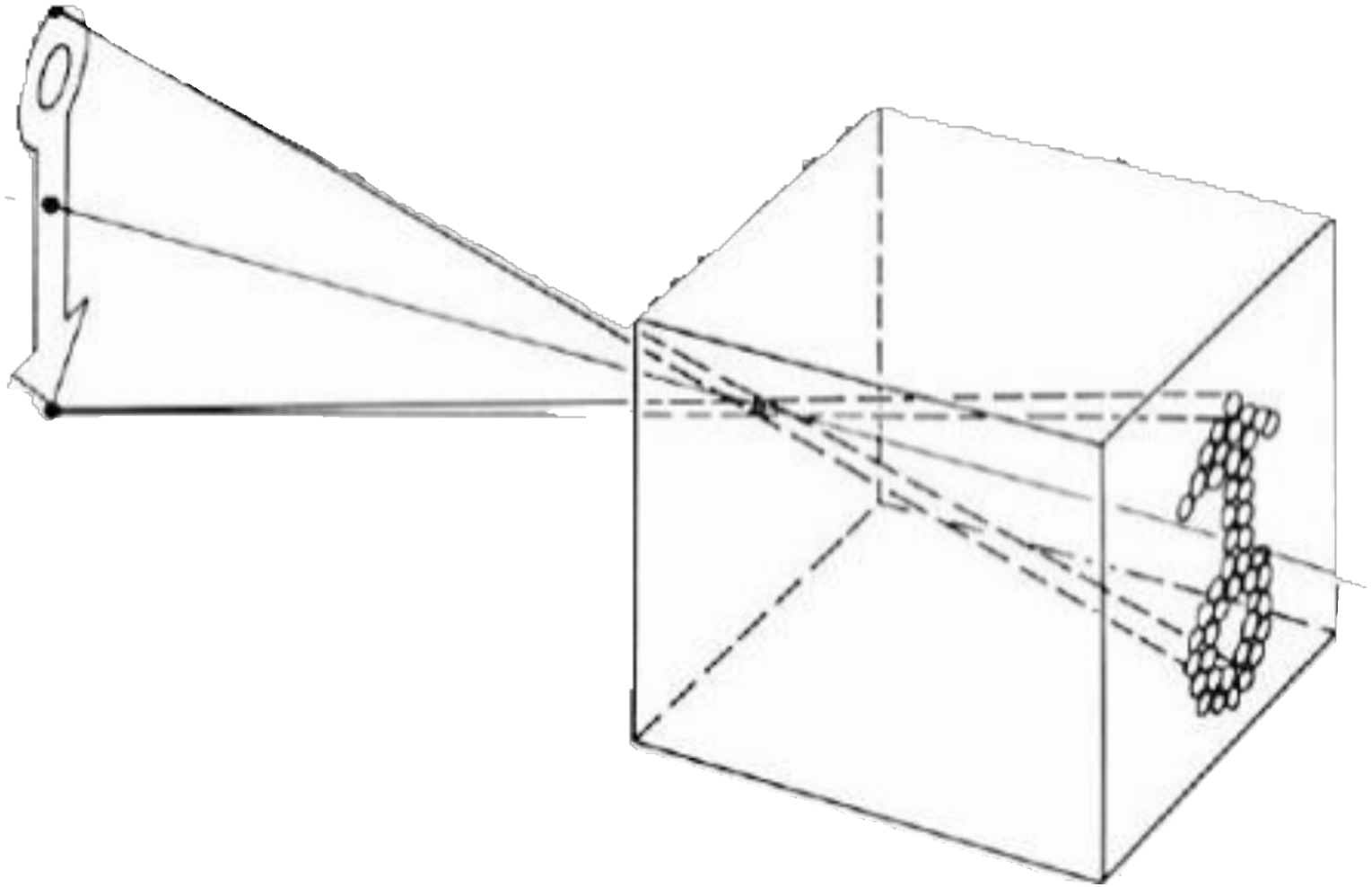
Pinhole imaging

real-world object

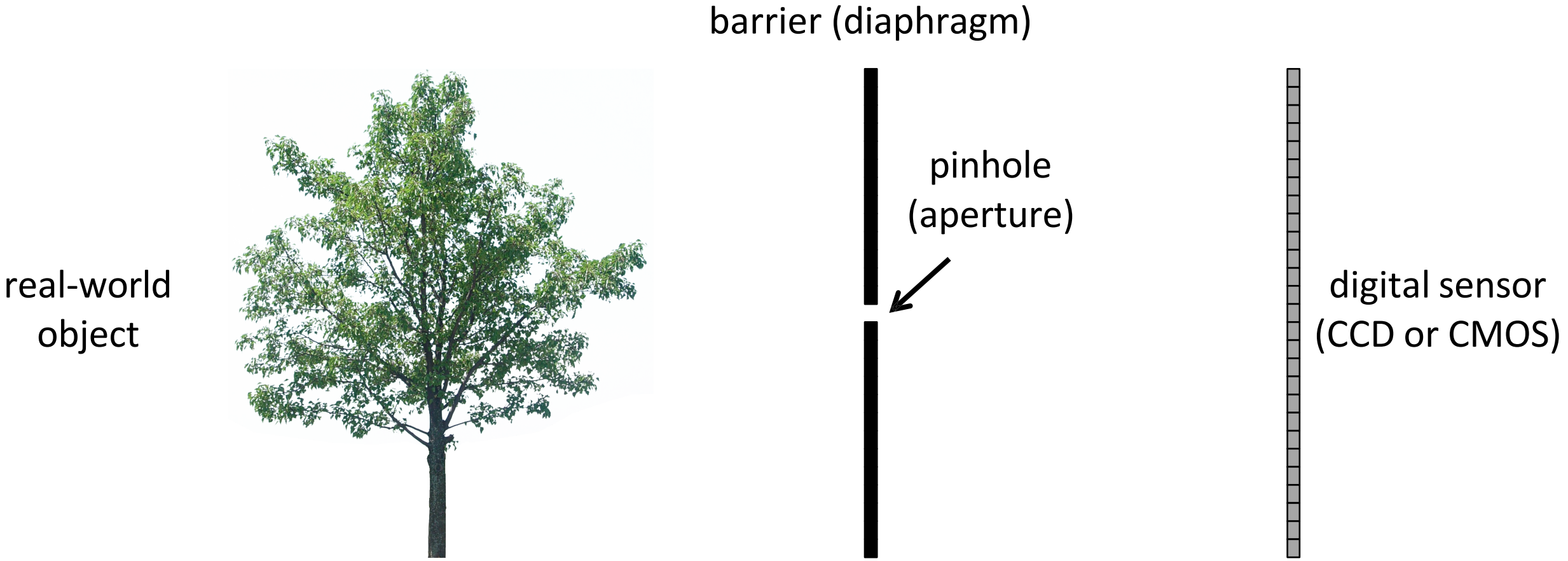


copy of real-world object (inverted and scaled)

Pinhole camera a.k.a. camera obscura

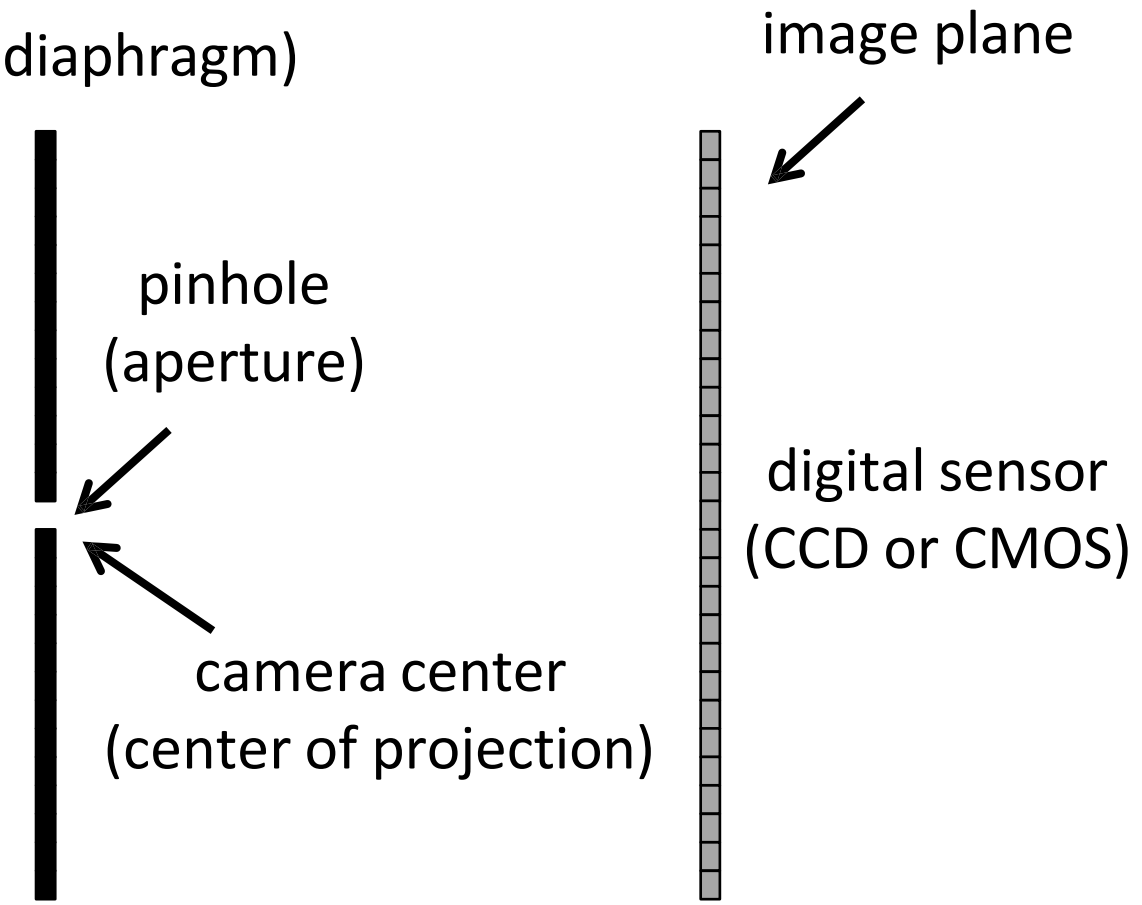


Pinhole camera terms



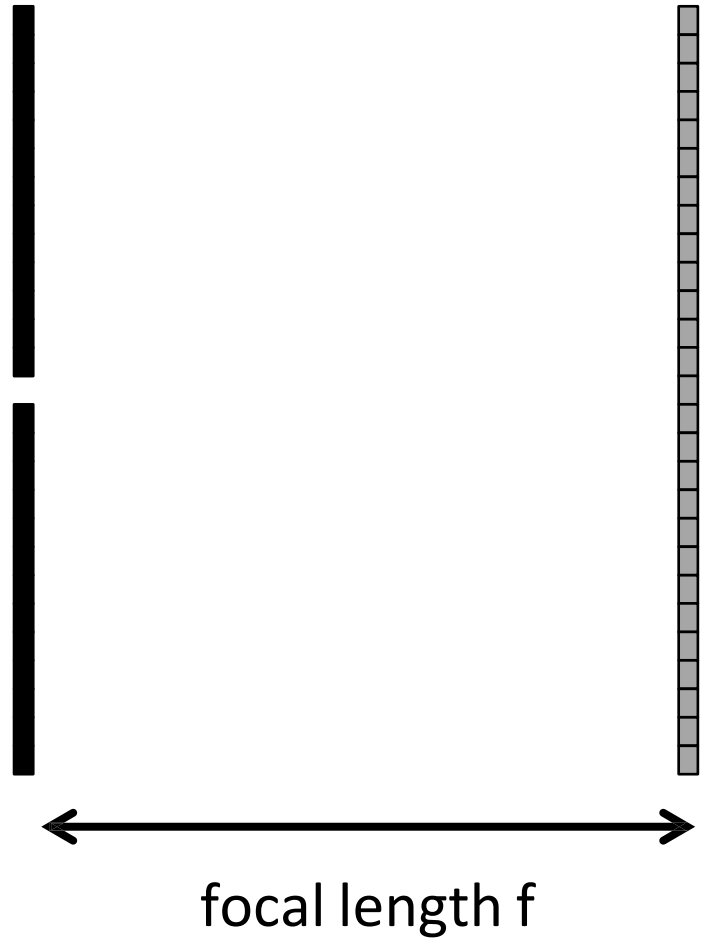
Pinhole camera terms

real-world object



Focal length

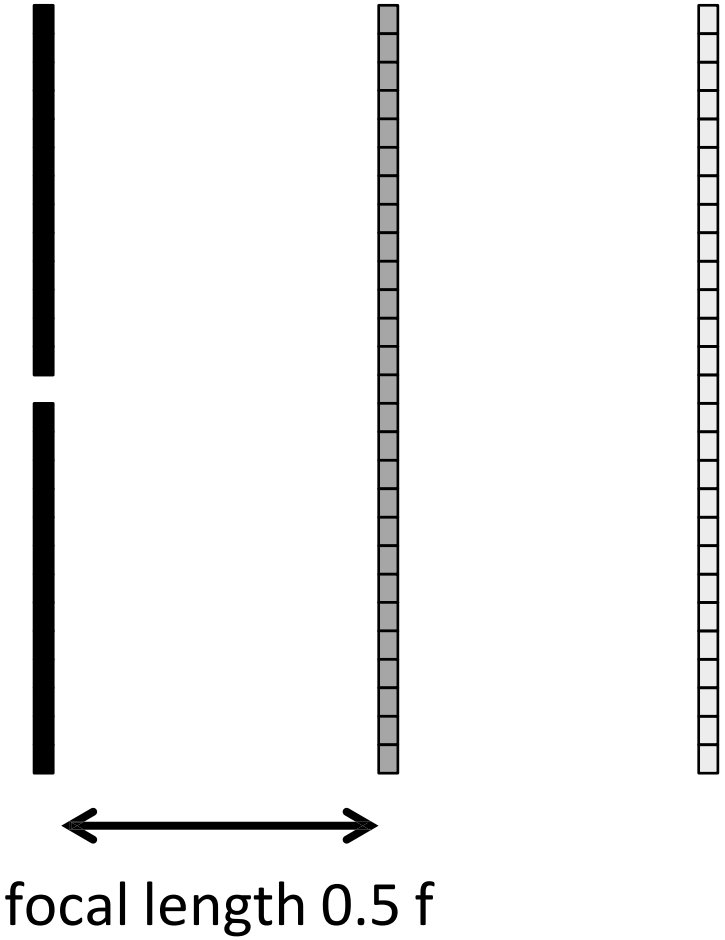
real-world object



Focal length

What happens as we change the focal length?

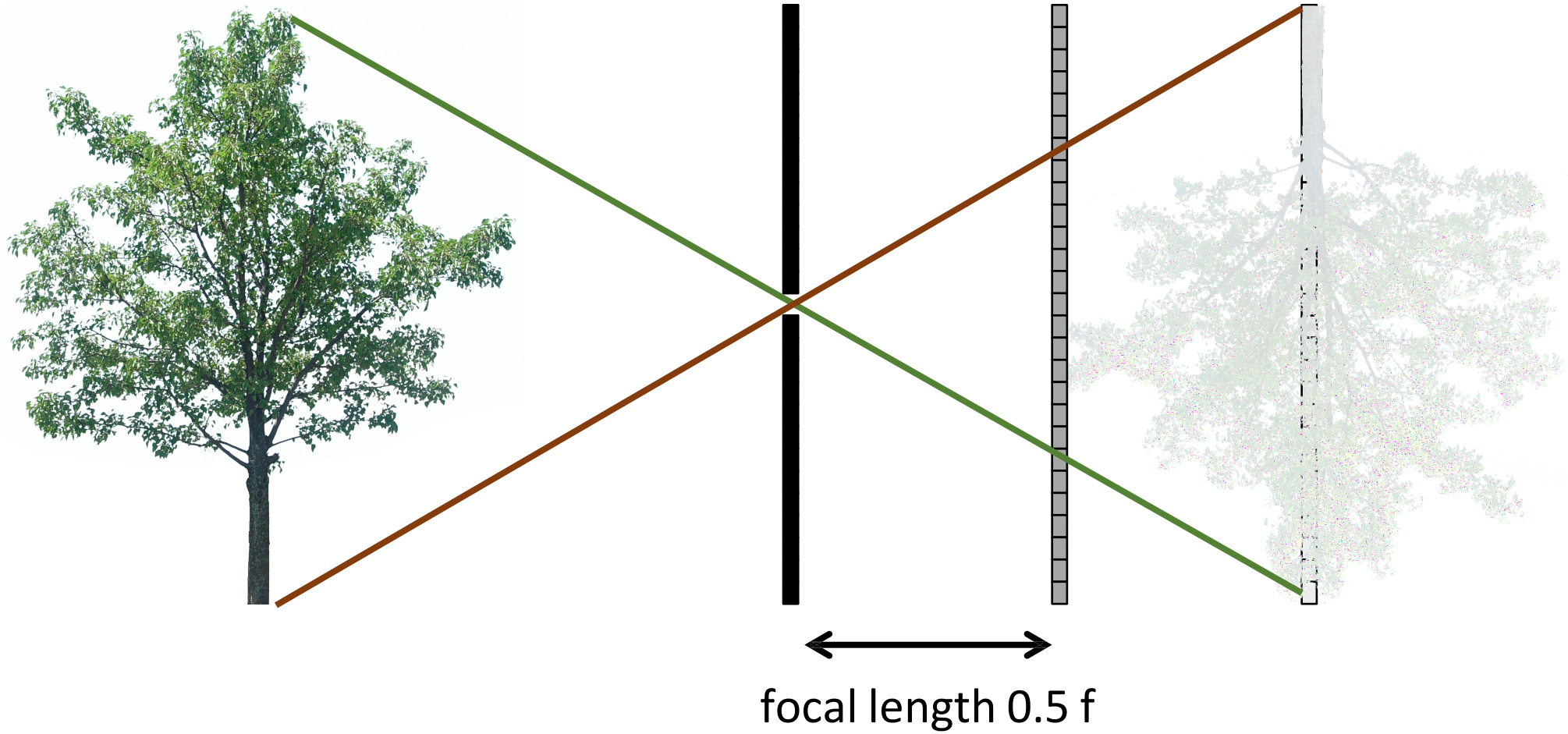
real-world object



Focal length

What happens as we change the focal length?

real-world object

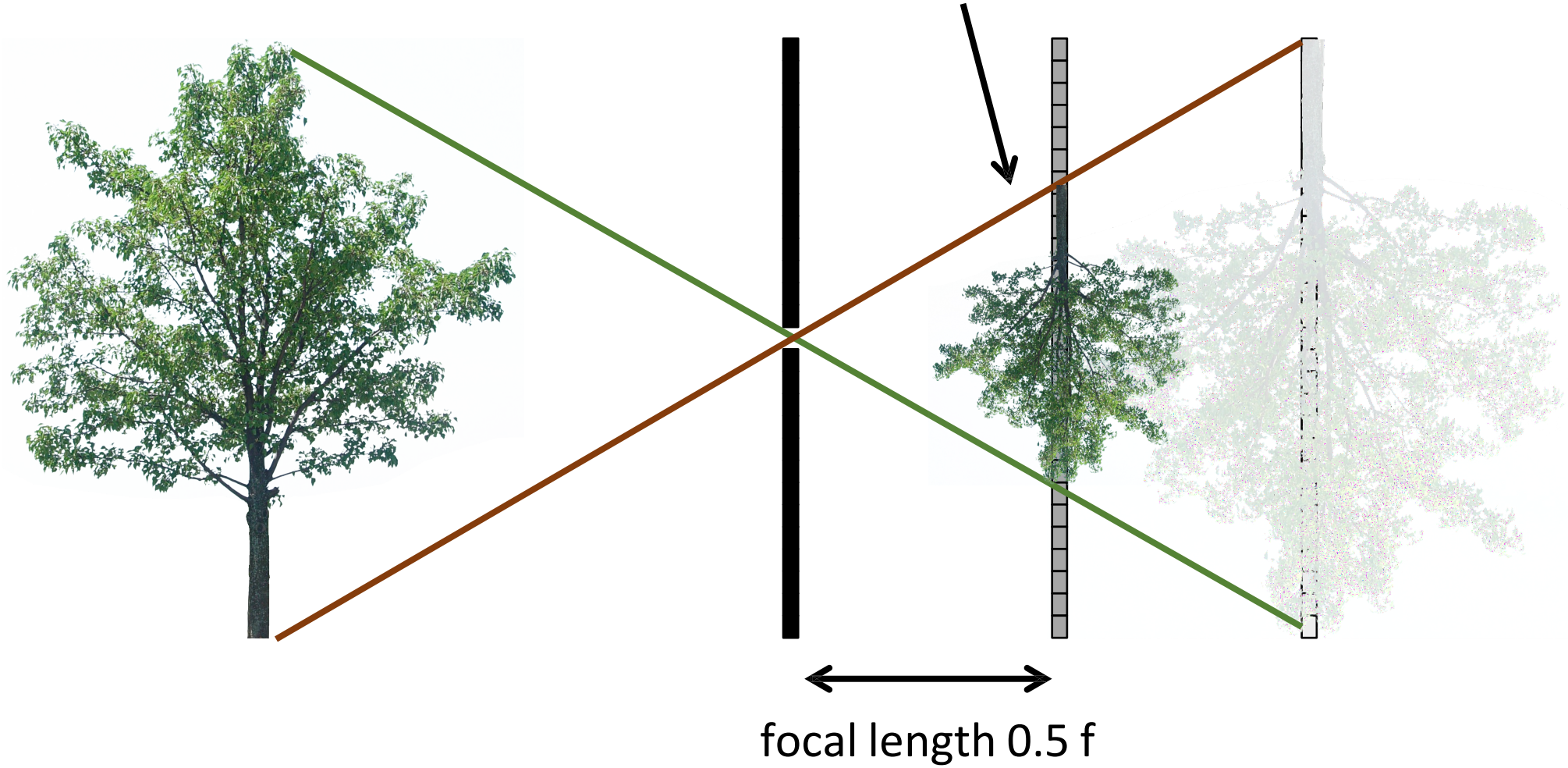


Focal length

What happens as we change the focal length?

object projection is half the size

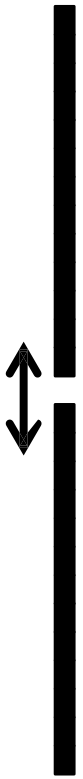
real-world object



Pinhole size

real-world object

pinhole diameter



Ideal pinhole has infinitesimally small size

- In practice that is impossible.

Pinhole size

What happens as we change the pinhole diameter?

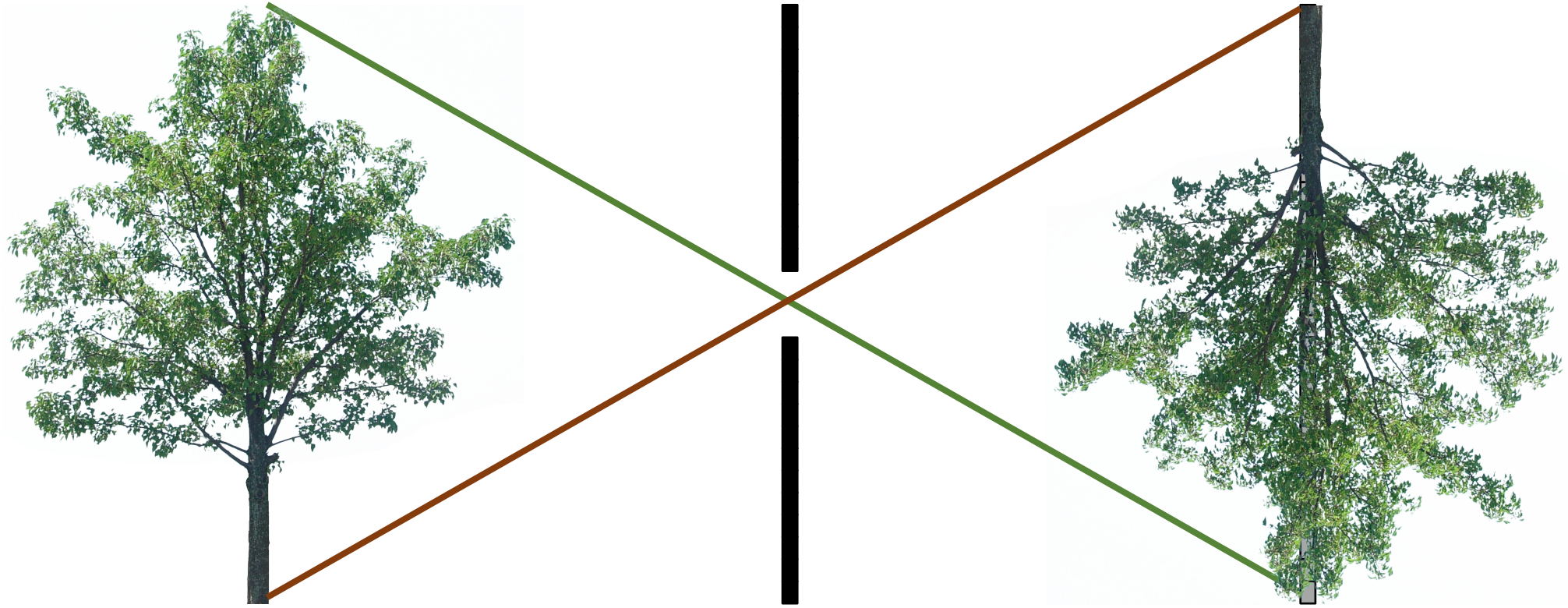
real-world object

pinhole diameter

Pinhole size

What happens as we change the pinhole diameter?

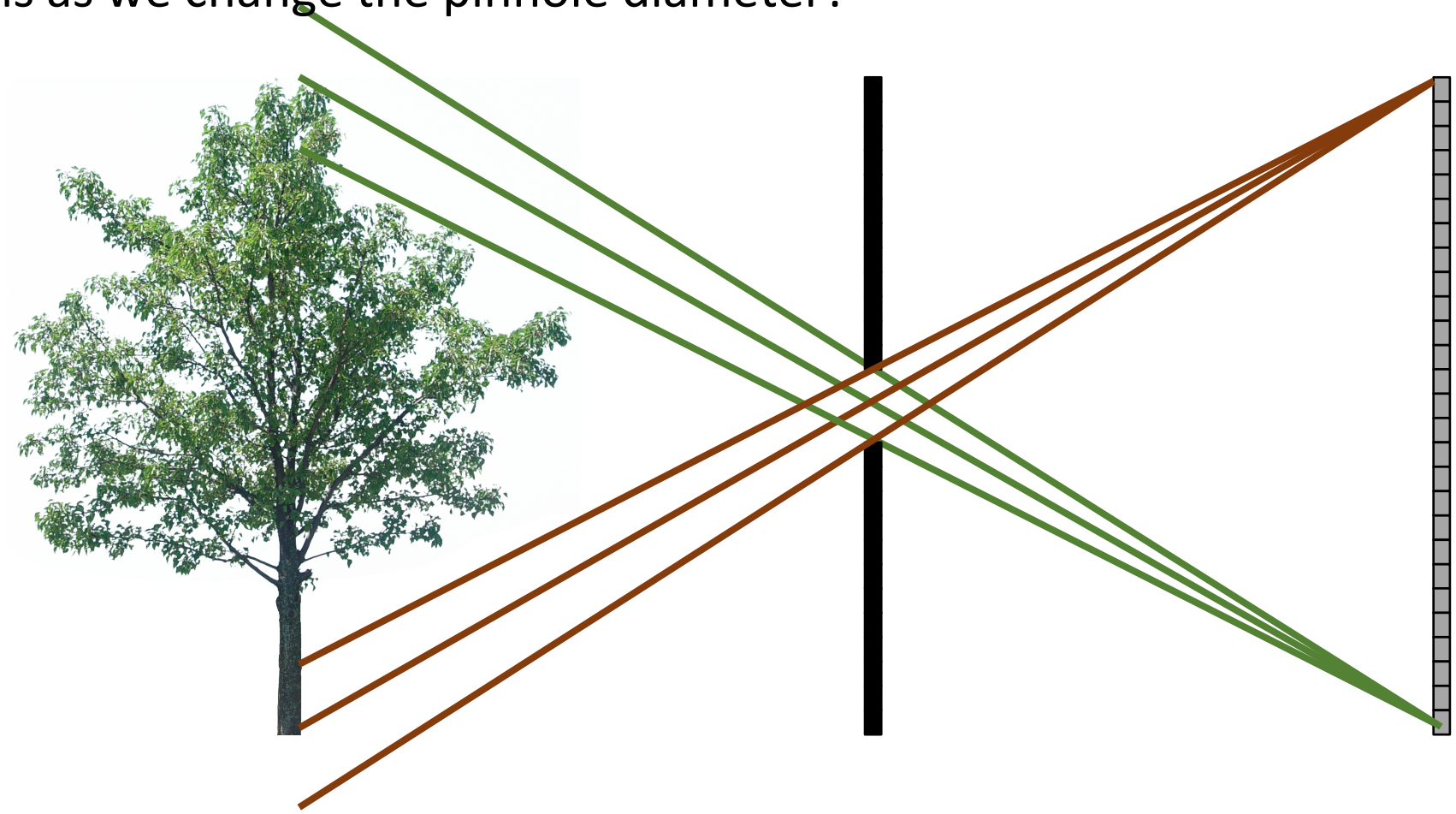
real-world object



Pinhole size

What happens as we change the pinhole diameter?

real-world object

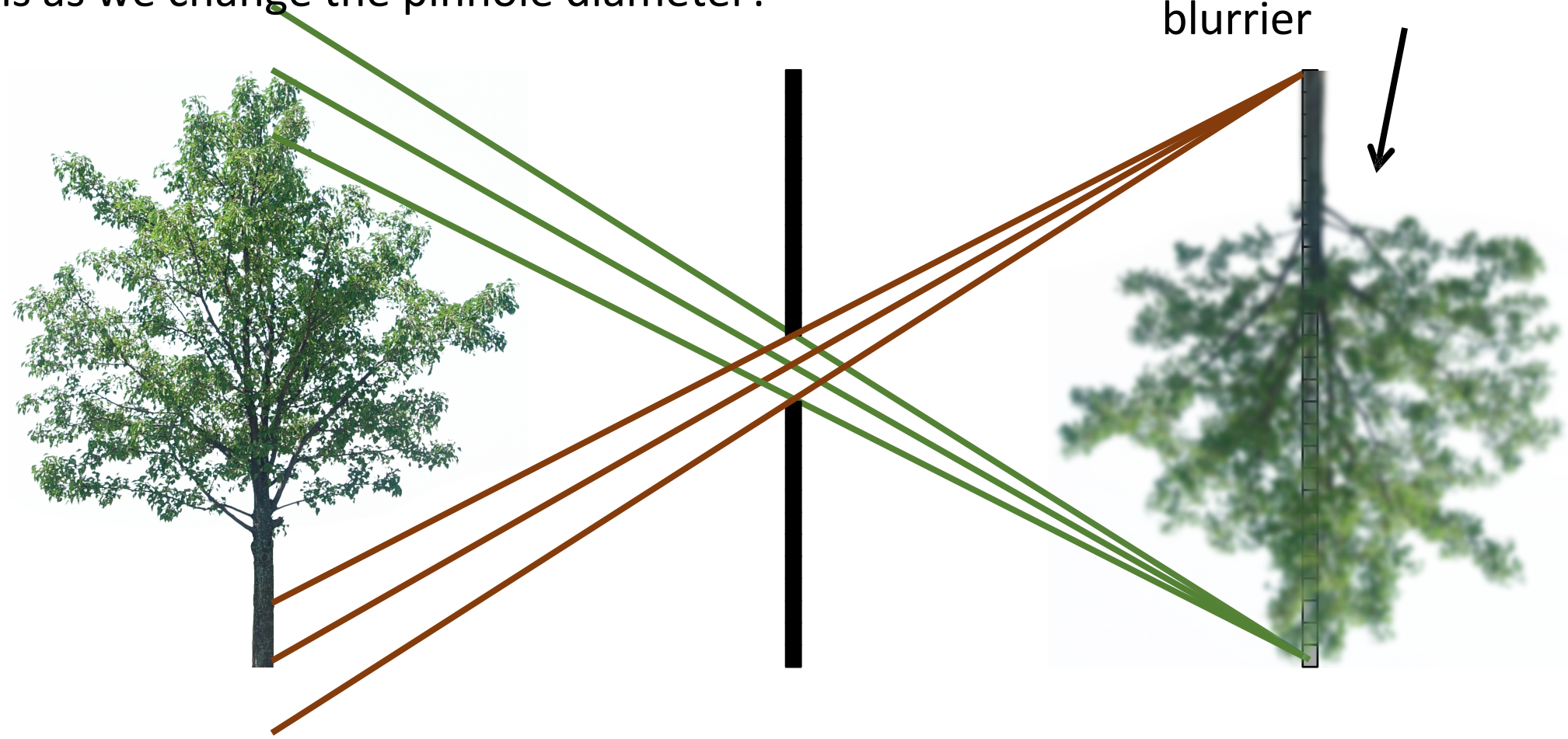


Pinhole size

What happens as we change the pinhole diameter?

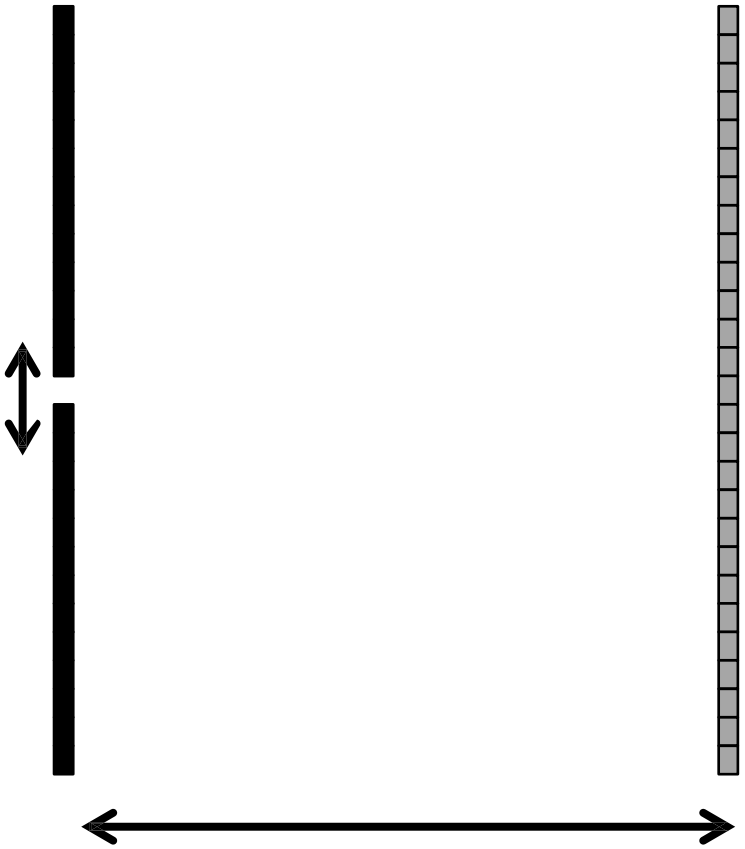
object projection becomes blurrier

real-world object



What about light efficiency?

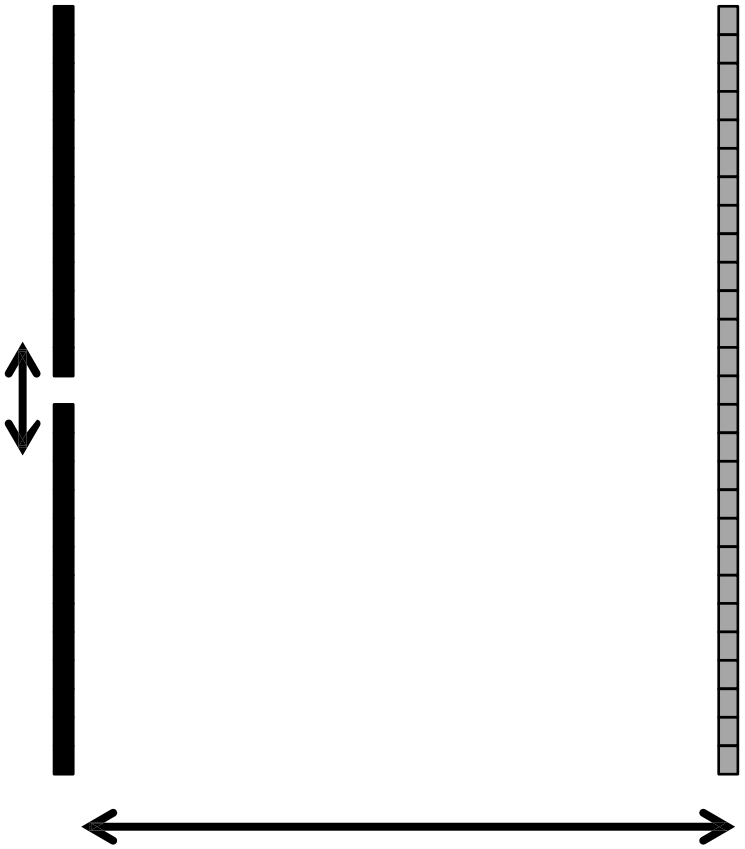
real-world object



- What is the effect of doubling the pinhole diameter?
- What is the effect of doubling the focal length?

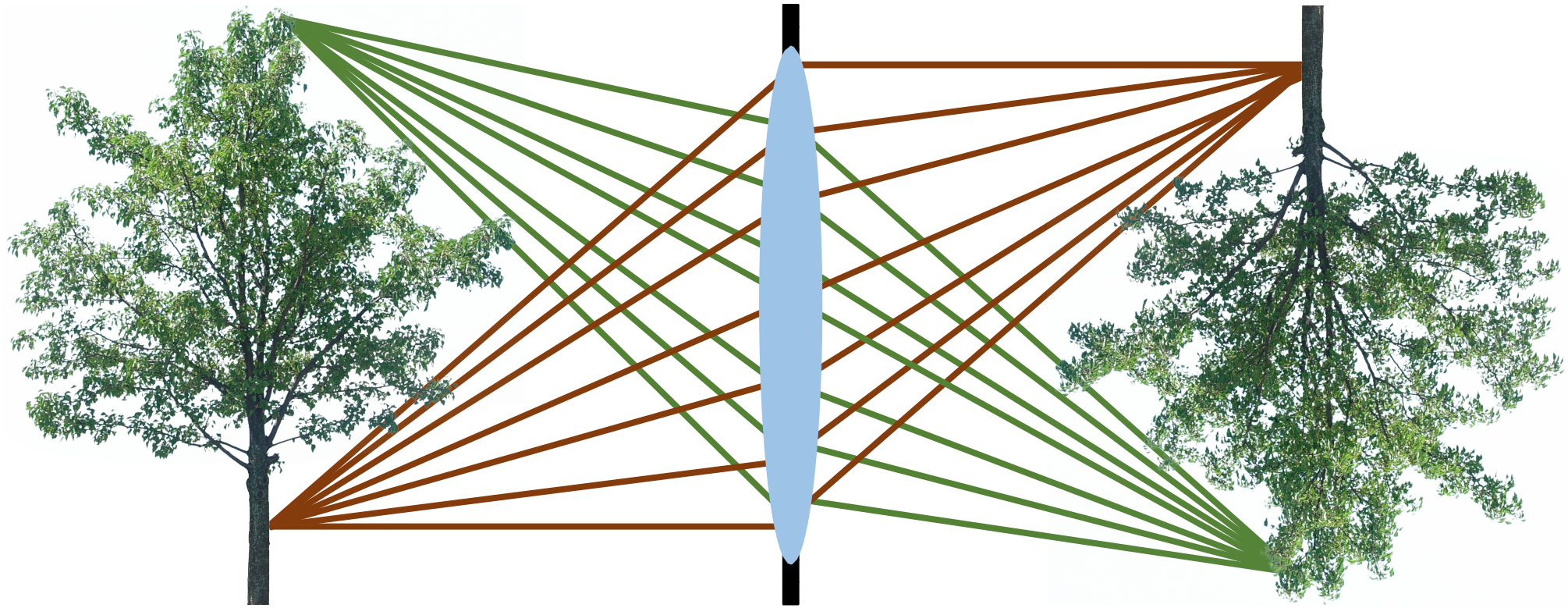
What about light efficiency?

real-world object



- 2x pinhole diameter → 4x light
- 2x focal length → ¼x light

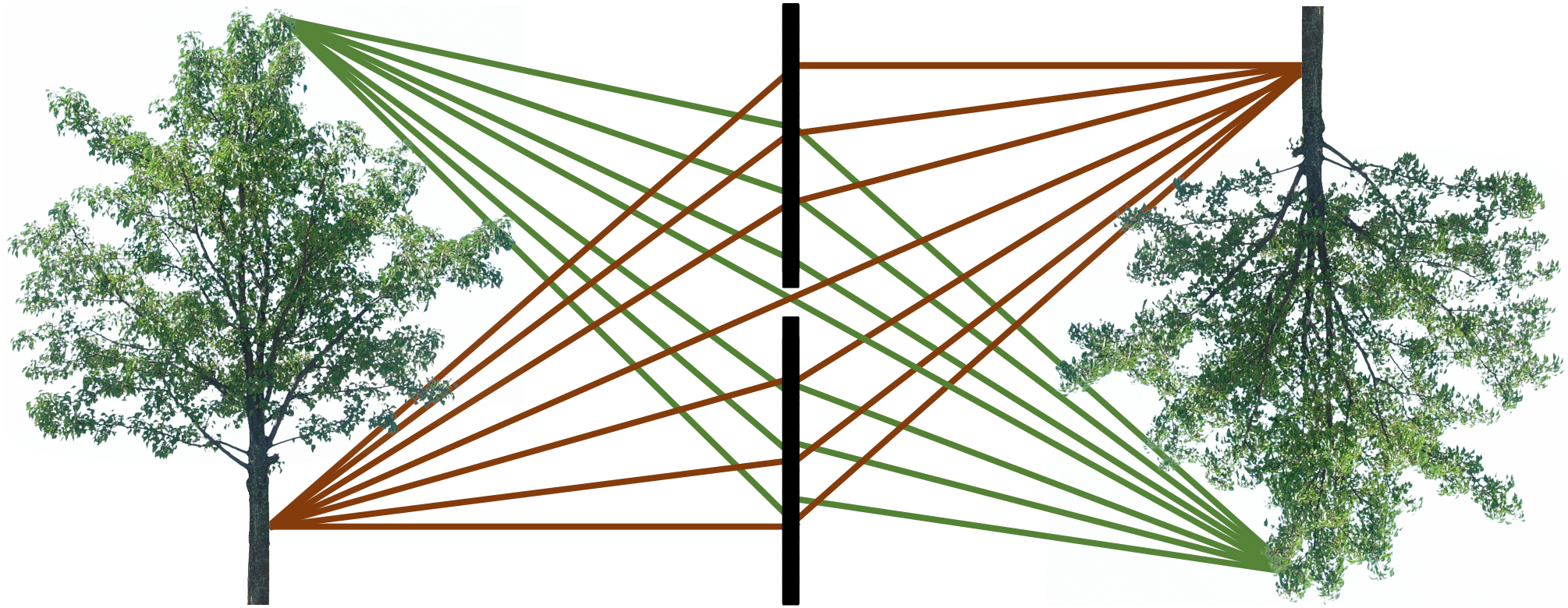
The lens camera



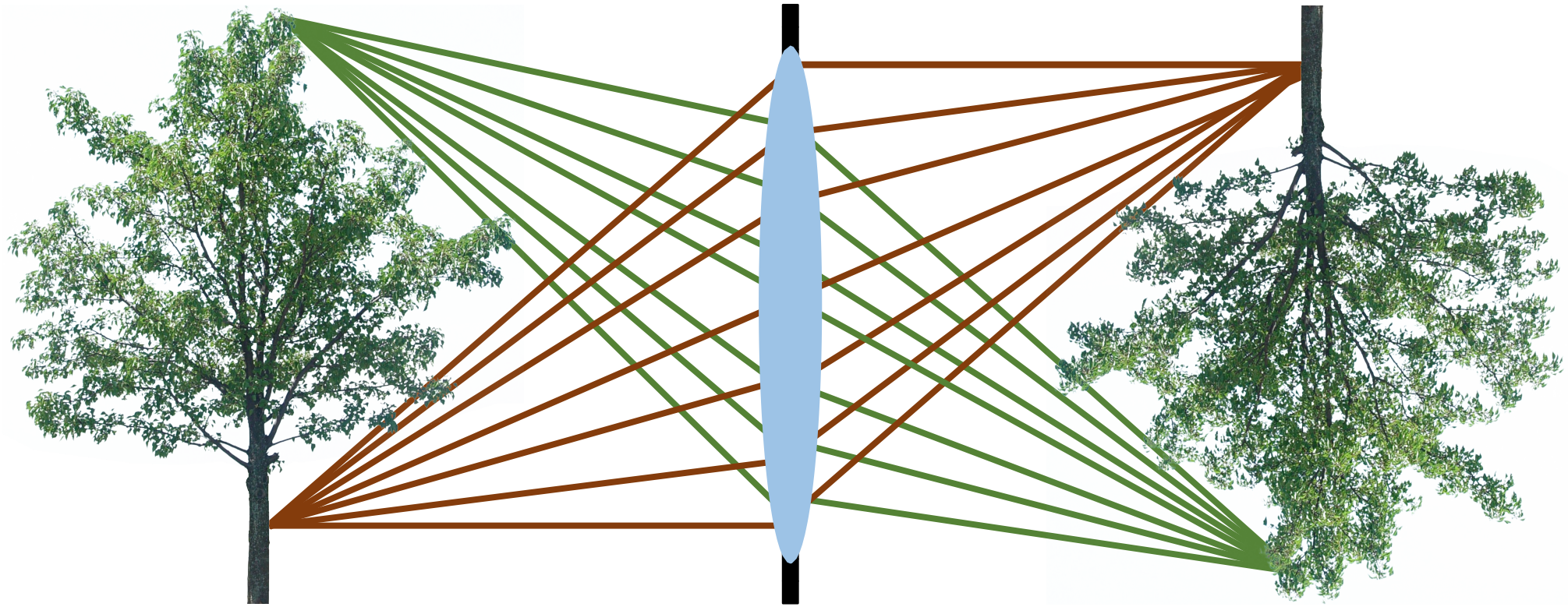
Lenses map “bundles” of rays from points on the scene to the sensor.

How does this mapping work exactly?

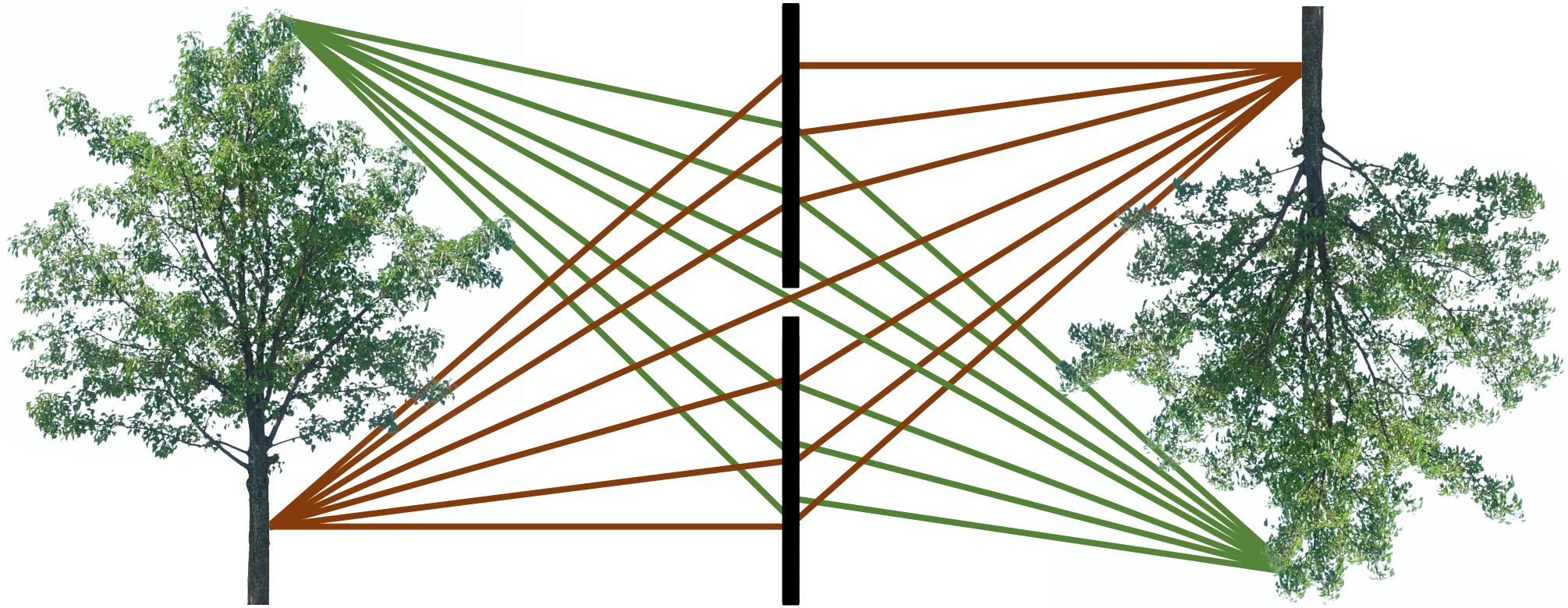
The pinhole camera



The lens camera

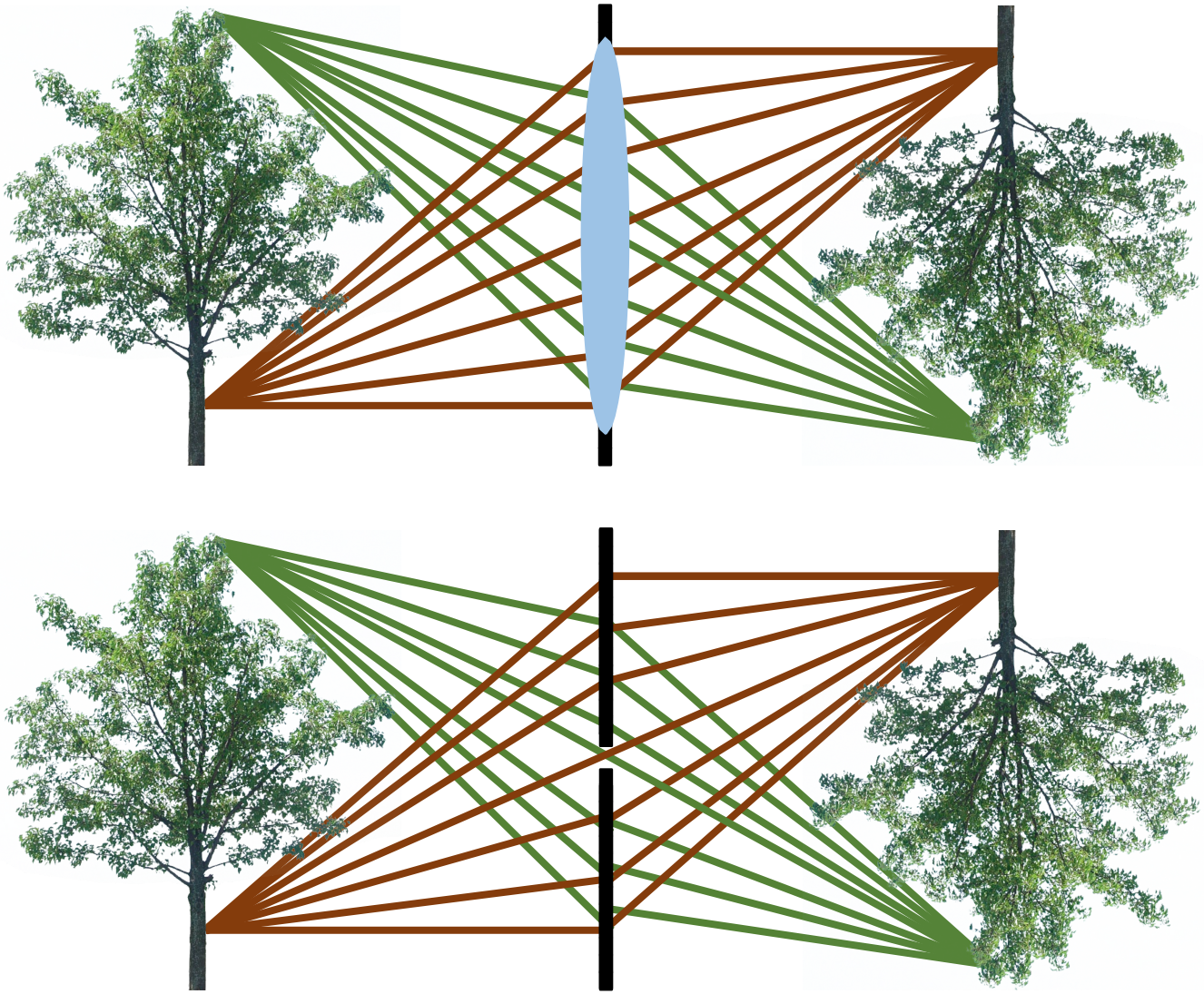


The pinhole camera



Central rays propagate in the same way for both models!

Describing both lens and pinhole cameras

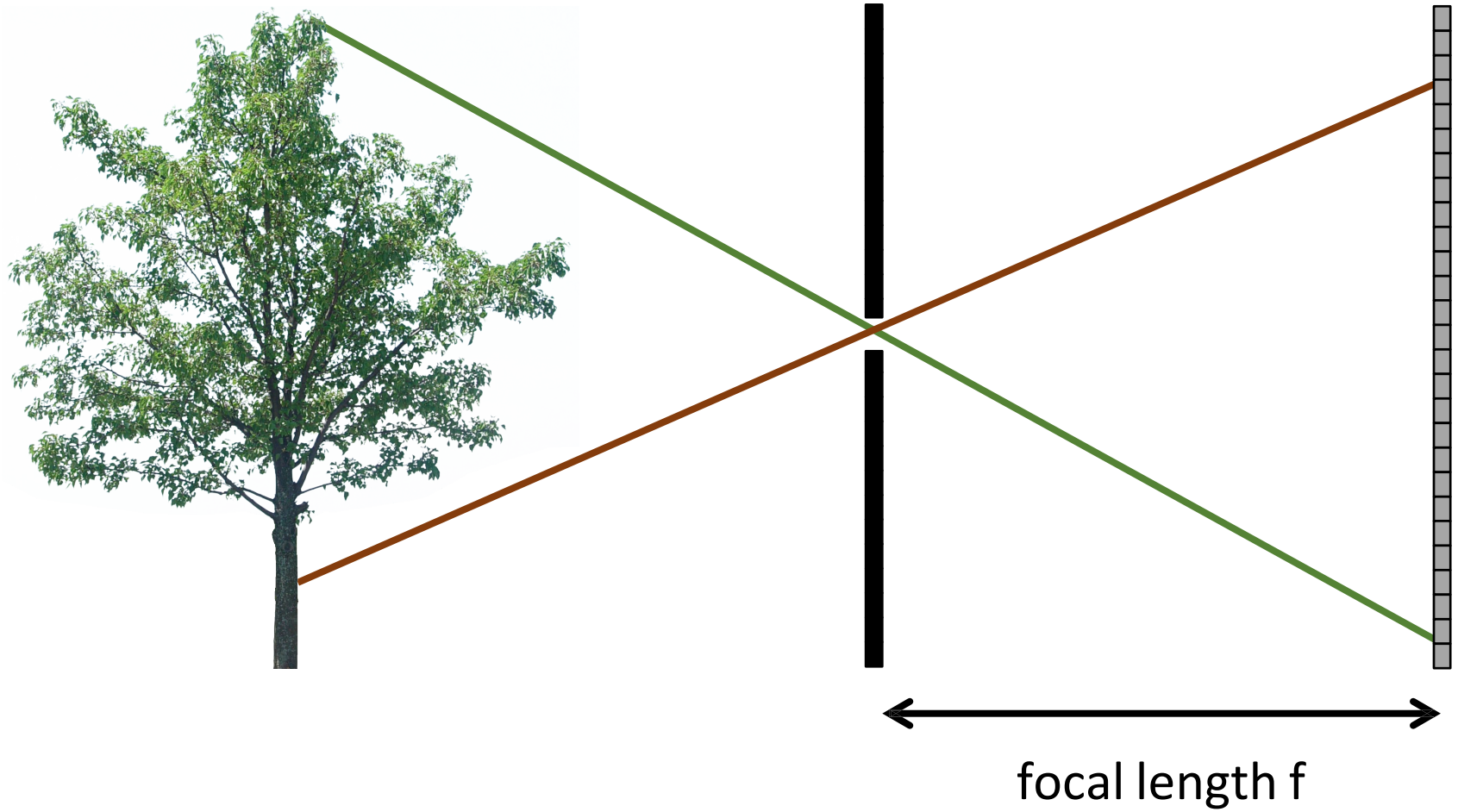


We can derive properties and descriptions that hold for both camera models if:

- We use only central rays.
- We assume the lens camera is in focus.

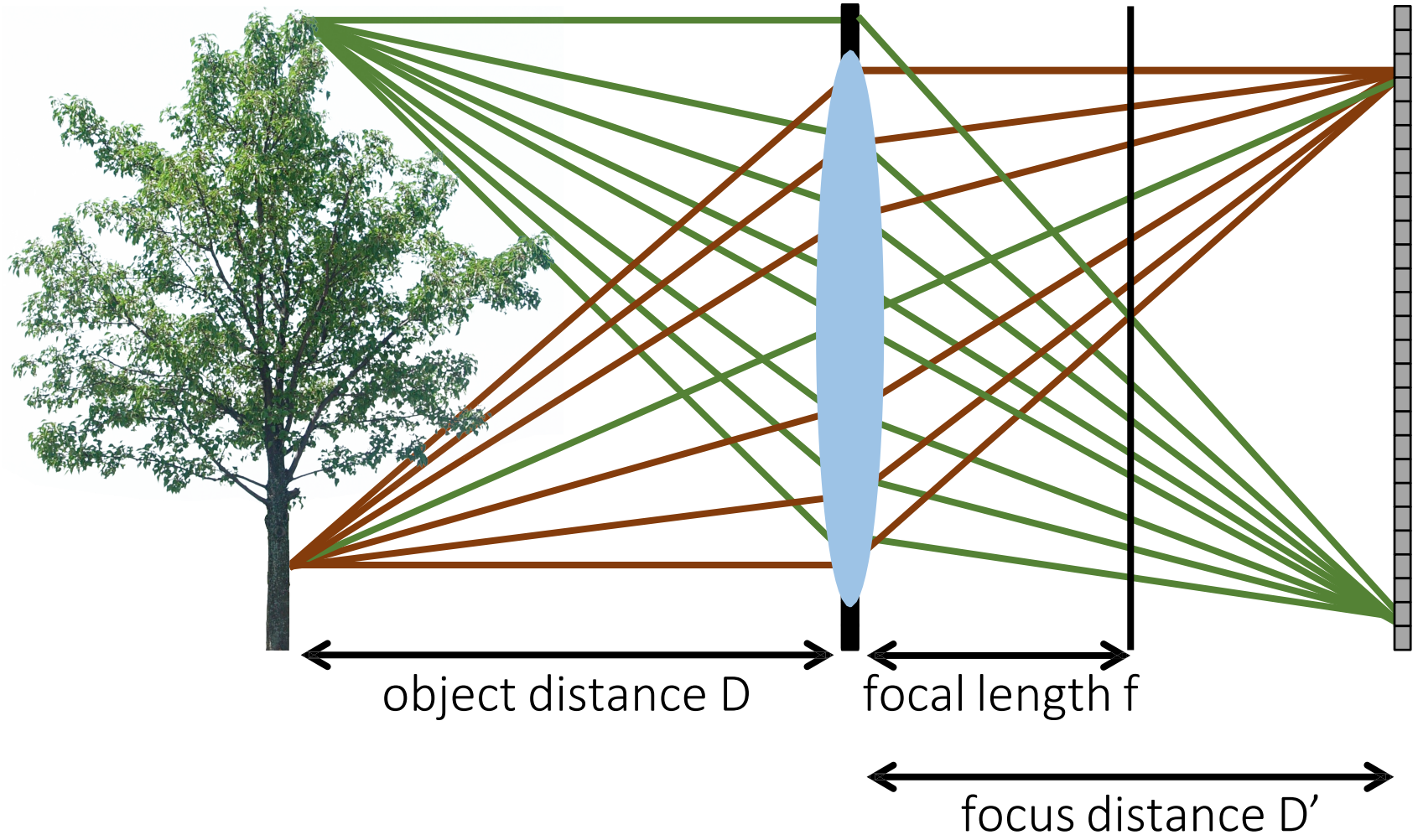
Important difference: focal length

In a pinhole camera, focal length is distance between aperture and sensor

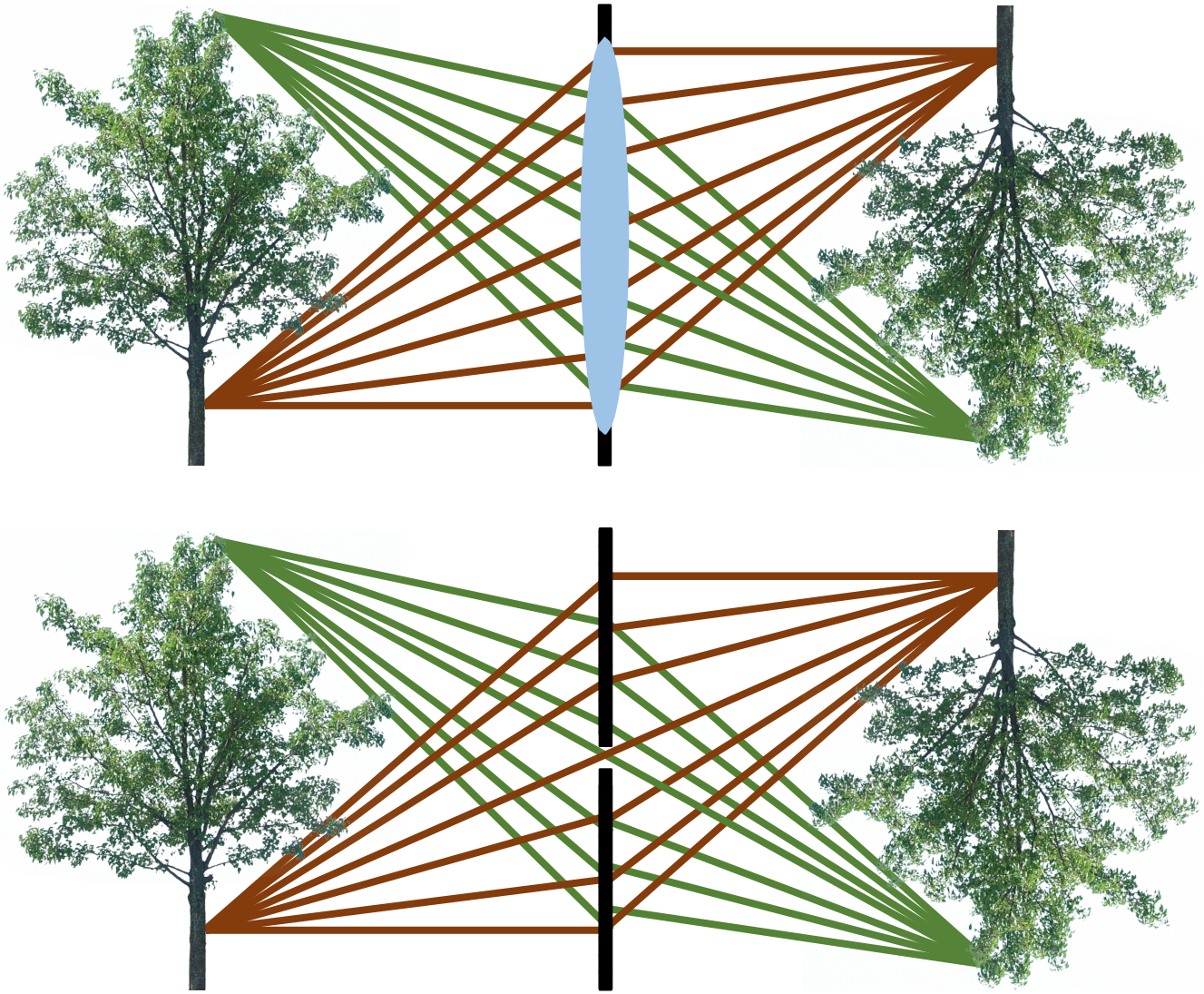


Important difference: focal length

In a lens camera, focal length is distance where parallel rays intersect



Describing both lens and pinhole cameras



We can derive properties and descriptions that hold for both camera models if:

- We use only central rays.
- We assume the lens camera is in focus.
- We assume that the focus distance of the lens camera is equal to the focal length of the pinhole camera.

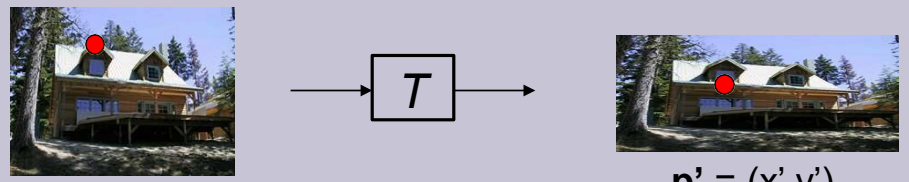
Remember: *focal length* f refers to different things for lens and pinhole cameras.

- In this course, we use it to refer to the aperture-sensor distance, as in the pinhole camera case.

Transformations

Definition

A transformation T is a coordinate-changing operation p’ = T(p)



What does it mean that T is global?

- T is the same for any point p
- T can be described by just a few numbers (parameters)

For linear transformations, we can represent T as a matrix

$$p' = \mathbf{T}p$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{T} \begin{bmatrix} x \\ y \end{bmatrix}$$

Common Transformations

Original

Transformed

Translation

Rotation

Scaling

Affine

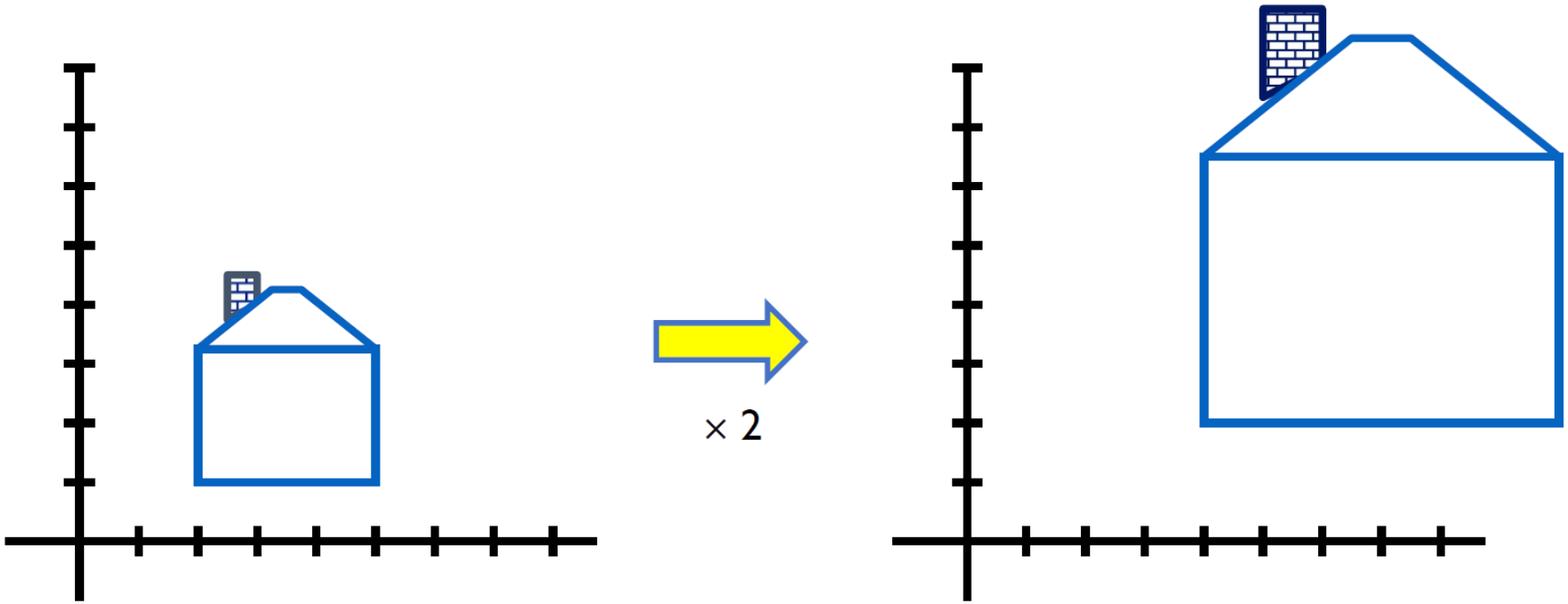
Perspective

next few slides):A. Efros and/or S. Seitz

Scaling

Definition

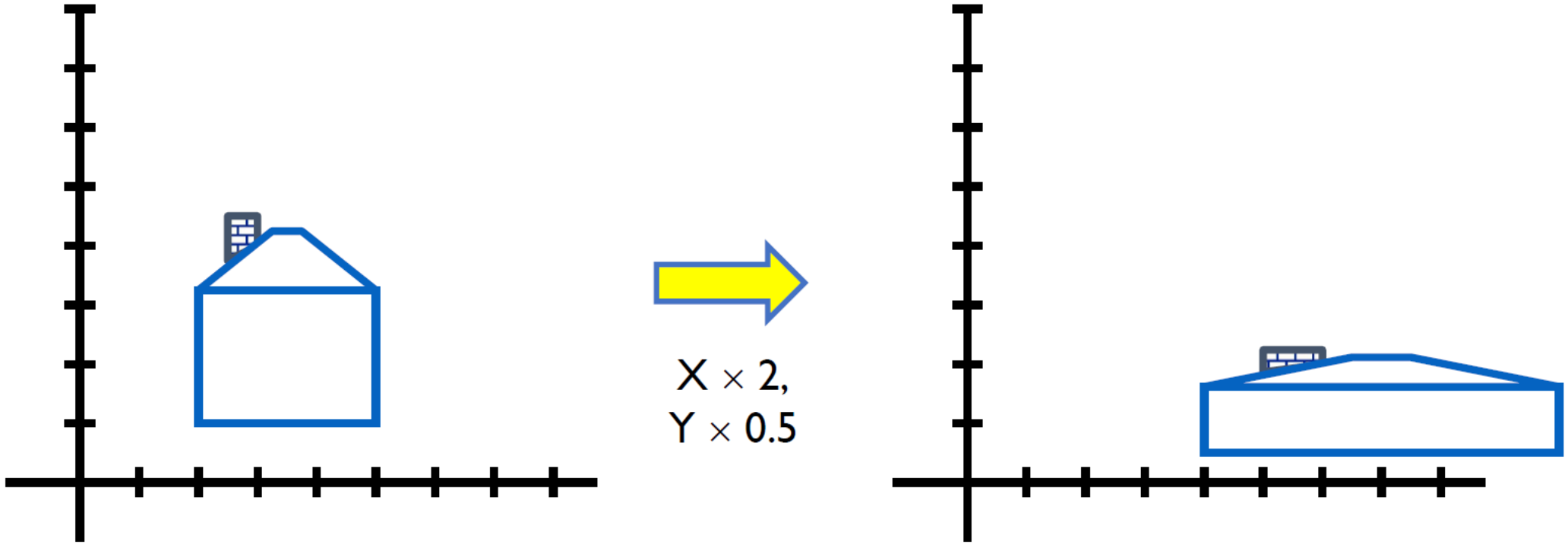
Scaling a coordinate means multiplying each of its components by a scalar.
Uniform scaling means this scalar is the same for all components.



Scaling

Definition

Non-uniform scaling: different scalars per component



Scaling

- Scaling operation:

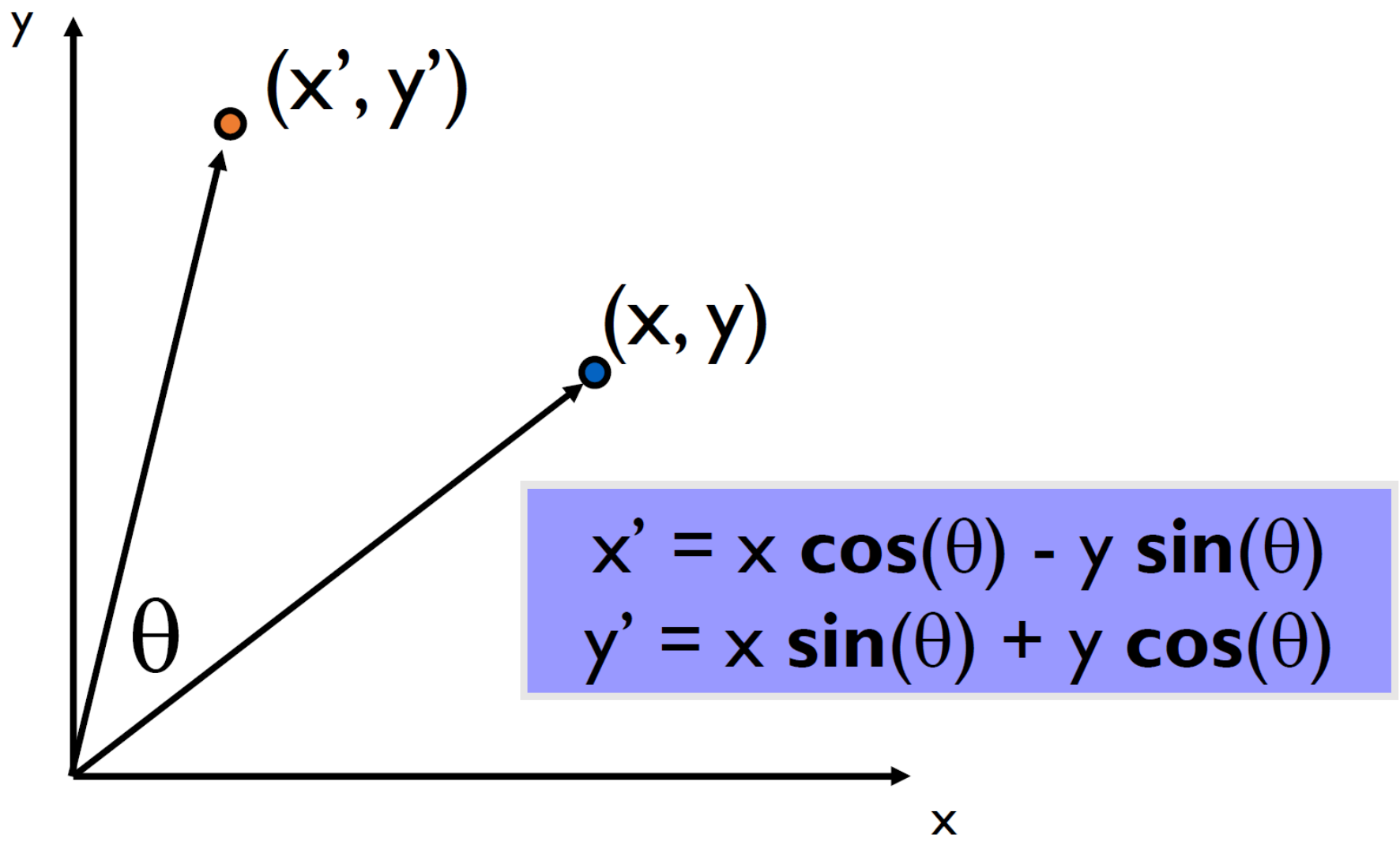
$$x' = ax$$

$$y' = by$$

- Or, in matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \underbrace{\begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix}}_{\text{scaling matrix } S} \begin{bmatrix} x \\ y \end{bmatrix}$$

2D Rotation



$$x' = x\cos(\theta) - y\sin(\theta)$$
$$y' = x\sin(\theta) + y\cos(\theta)$$

Basic 2D Transformations

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Scale

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & \alpha_x \\ \alpha_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Shear

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\Theta & -\sin\Theta \\ \sin\Theta & \cos\Theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Rotate

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Translate

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Affine

Affine is any combination of translation, scale, rotation, shear

Affine Transformations

- Affine transformations are combinations of
  - Linear transformations, and
  - Translations

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

or

- Properties of affine transformations:
  - Lines map to lines
  - Parallel lines remain parallel
  - Ratios are preserved
  - Closed under composition

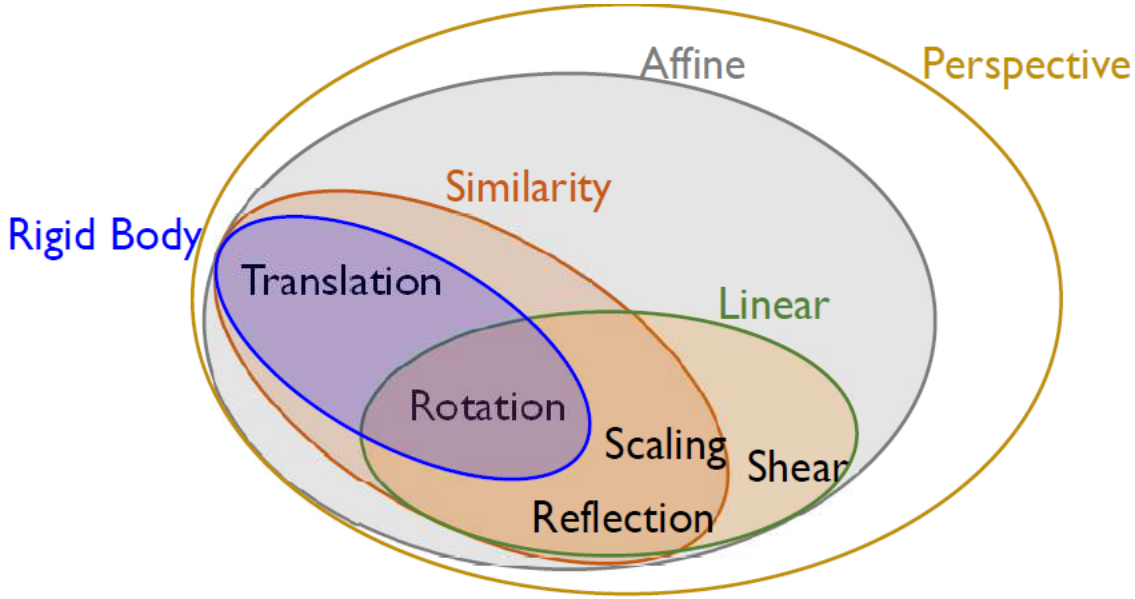
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

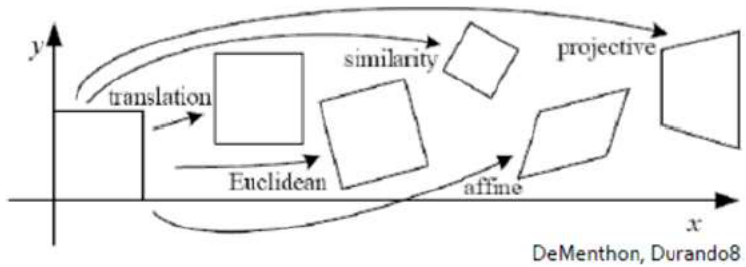
ɔugih Jamin

2D Transformations: Properties Preserved

Transformations can be classified based on the properties they preserve:

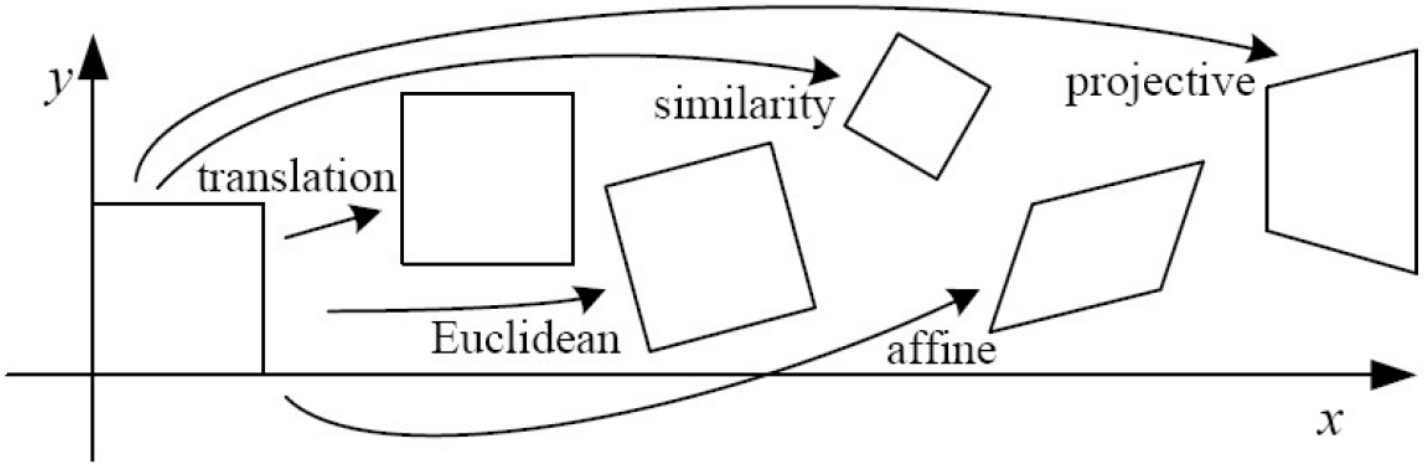
- Rigid Body
  - angles, lengths, areas
- Similarity
  - angles, length ratios
- Linear
  - linear combination
- Affine
  - parallel lines, length ratios, area ratios
- Perspective
  - collinearity, cross-ratio
- …





Sugih Jamin

2D Transformations



Szeliski 2.1

Name	Matrix	# D.O.F.	Preserves:	Icon
translation	$\begin{bmatrix} \boldsymbol{I} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	2	orientation $+\cdots$	
rigid (Euclidean)	$\begin{bmatrix} \boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	3	lengths $+\cdots$	
similarity	$\begin{bmatrix} s\boldsymbol{R} \mid \boldsymbol{t} \end{bmatrix}_{2\times3}$	4	angles $+\cdots$	
affine	$\begin{bmatrix} \boldsymbol{A} \end{bmatrix}_{2\times3}$	6	parallelism $+\cdots$	
projective	$\begin{bmatrix} \tilde{\boldsymbol{H}} \end{bmatrix}_{3\times3}$	8	straight lines	

‘Homography’

Cameras are Projective Transformation Machines

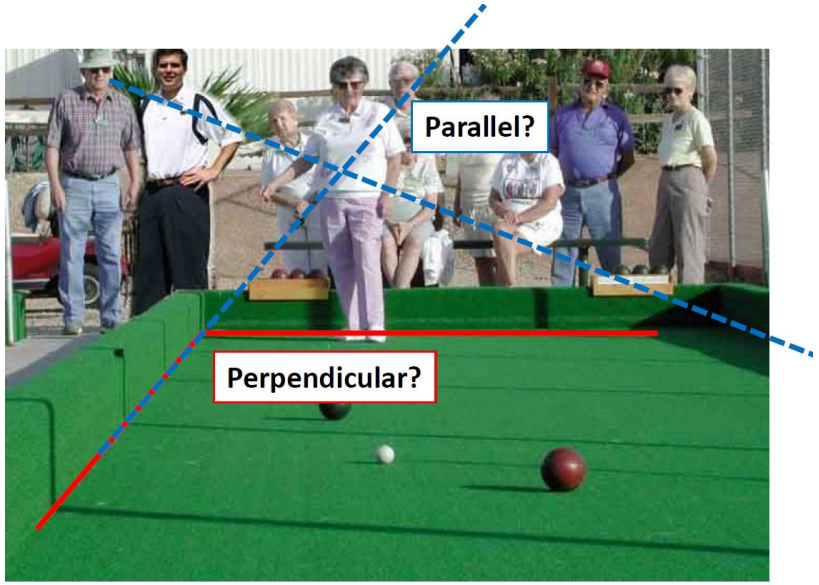
Projective Transformation

3D World

2D Image

Projective Transformations

- Projective transformations are combos of
  - Affine transformations, and
  - Projective warps



- Properties of projective transformations:
  - Lines map to lines
  - Parallel lines do not necessarily remain parallel
  - Ratios are not preserved
  - Closed under composition
  - Models change of basis
  - Projective matrix is defined up to a scale (8 degrees of freedom)

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$