Lecture 14 — Stereo Vision

Lecture 14

Kaveh Fathian
Computer Science Department, Colorado School of Mines

Learning Objectives

By the end of this lecture, you should be able to:

- Explain how a calibrated stereo pair gives depth from disparity, compute depth with $Z = f\,b\,/\,d$, and explain why depth precision drops with distance.
- Describe why and how images are rectified, and rectify a stereo pair with OpenCV.
- Find dense correspondences by block matching along rows: compare matching costs (SAD, SSD, NCC), and explain the cost volume and the window-size trade-off.
- Explain how semi-global matching adds smoothness, and compute disparity maps with `StereoBM` and `StereoSGBM`.
- Convert a disparity map into a colored 3D point cloud with `reprojectImageTo3D`, and recognize where stereo fails.

From Sparse to Dense

Two-view: depth at 870 matched points

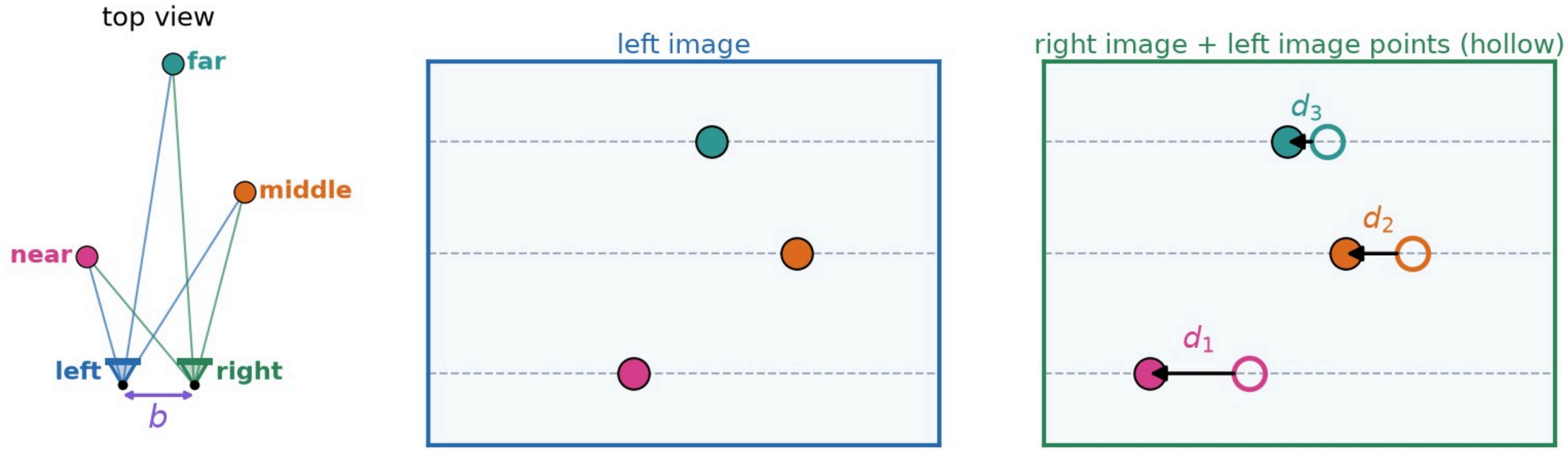
Stereo: depth at 648,115 pixels (63%)



Gray: no reliable match (plain ground, window reflections, areas only one camera sees)

- **Two-view:** match features, then triangulate $\rightarrow$ depth only where features were matched.
- **Stereo:** search for a match for **every pixel** $\rightarrow$ a dense depth map that agrees with the sparse points.

The Stereo Setup

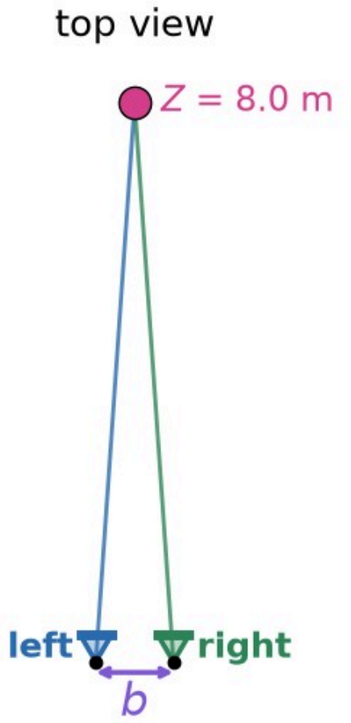


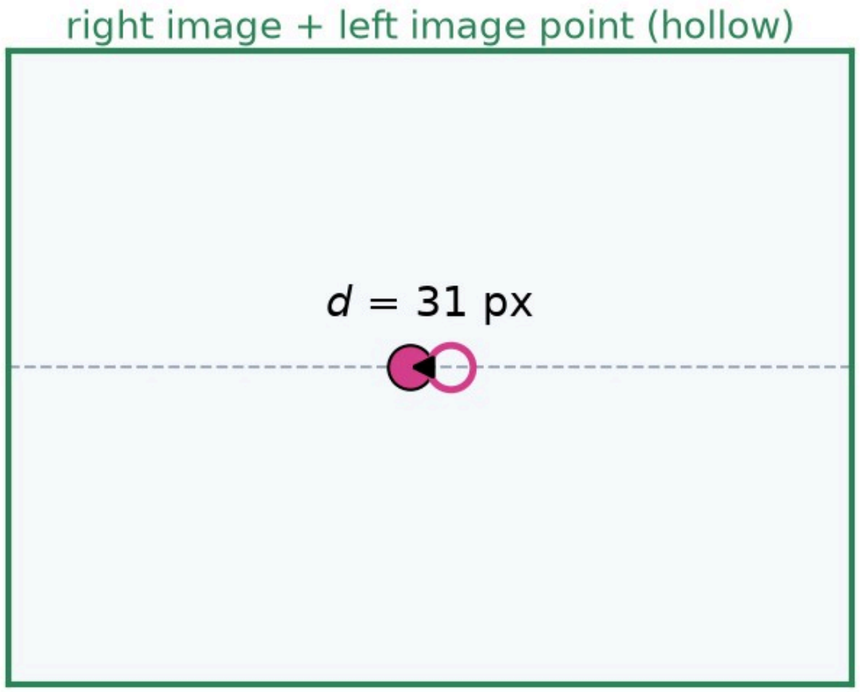
- Two calibrated cameras **side by side**, same orientation, a **baseline** b apart.
- Each point stays on the **same row** and shifts left by the **disparity** d. **Near points shift more.**

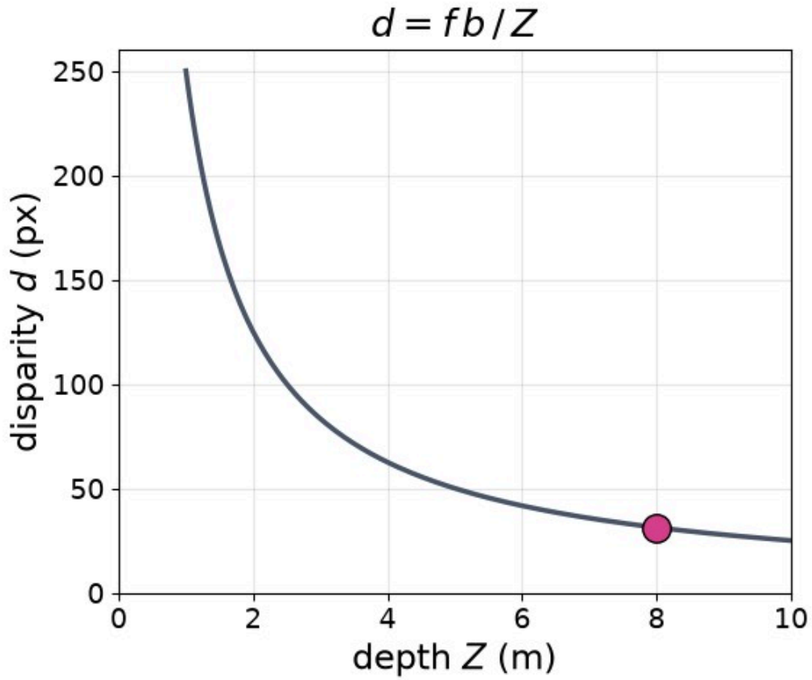
Stereo Reconstruction

Rectify the images to side by side $\rightarrow$ **match** each pixel along its row $\rightarrow$ **disparity** $\rightarrow$ **depth** $\rightarrow$ **3D points**

Disparity Gives Depth







$$Z = \frac{f\,b}{d}$$

depth from disparity

- f: focal length (px), b: baseline (m), d: disparity (px).
- Twice as far $\rightarrow$ half the disparity. Very far: $d \rightarrow 0$.

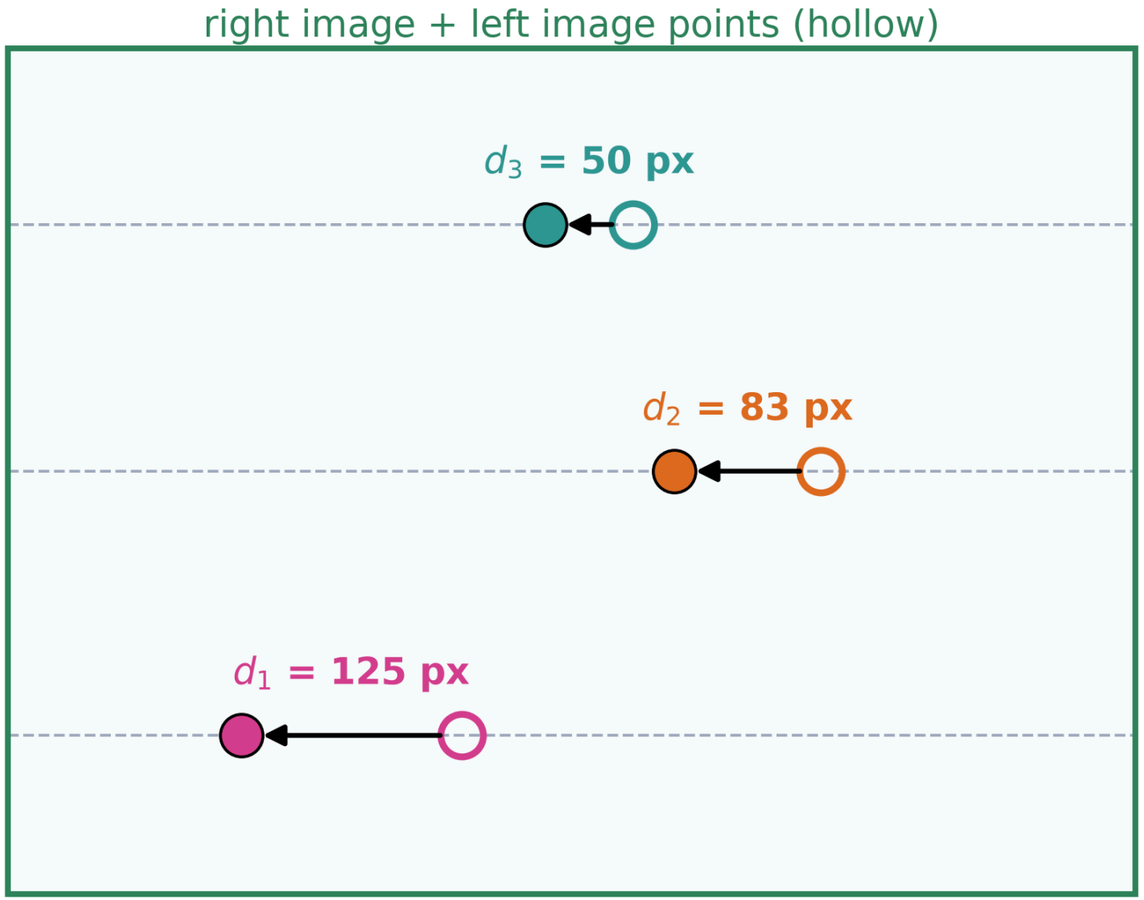
Depth From Disparity: A Numerical Example

The three points from the stereo setup:

1. **Measure** the columns x_L and x_R.
2. **Disparity:** $d = x_L - x_R$.
3. **Depth:** $Z = f\,b\,/\,d$.

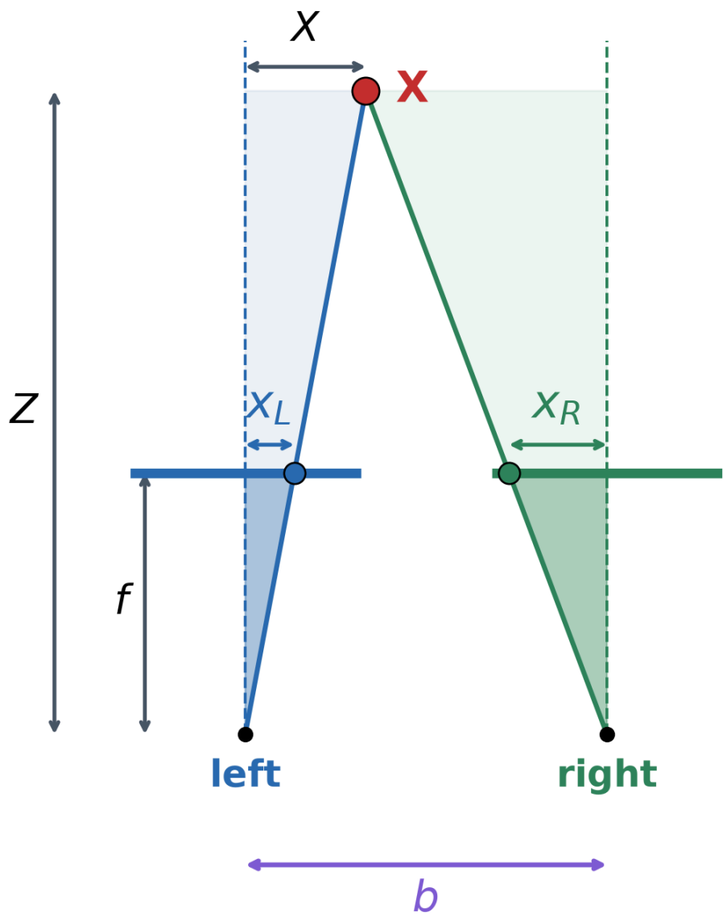
Rig: f = 500 px, b = 0.5 m, so $f\,b$ = 250 px·m.

Point	x_L (px)	x_R (px)	d (px)	$f\,b\,/\,d$ (m)	True Z (m)
near	257.5	132.5	125.0	**2.00**	2.00
middle	461.7	378.3	83.3	**3.00**	3.00
far	355.0	305.0	50.0	**5.00**	5.00



- f in pixels and b in meters $\rightarrow$ Z in meters. The depths match the true ones.

(Supplementary) Why $Z = f\,b\,/\,d$: Similar Triangles



- Image coordinates are measured from each camera's optical axis.
- **Left camera:** the small and large blue triangles are similar:

$$\frac{x_L}{f} = \frac{X}{Z}$$

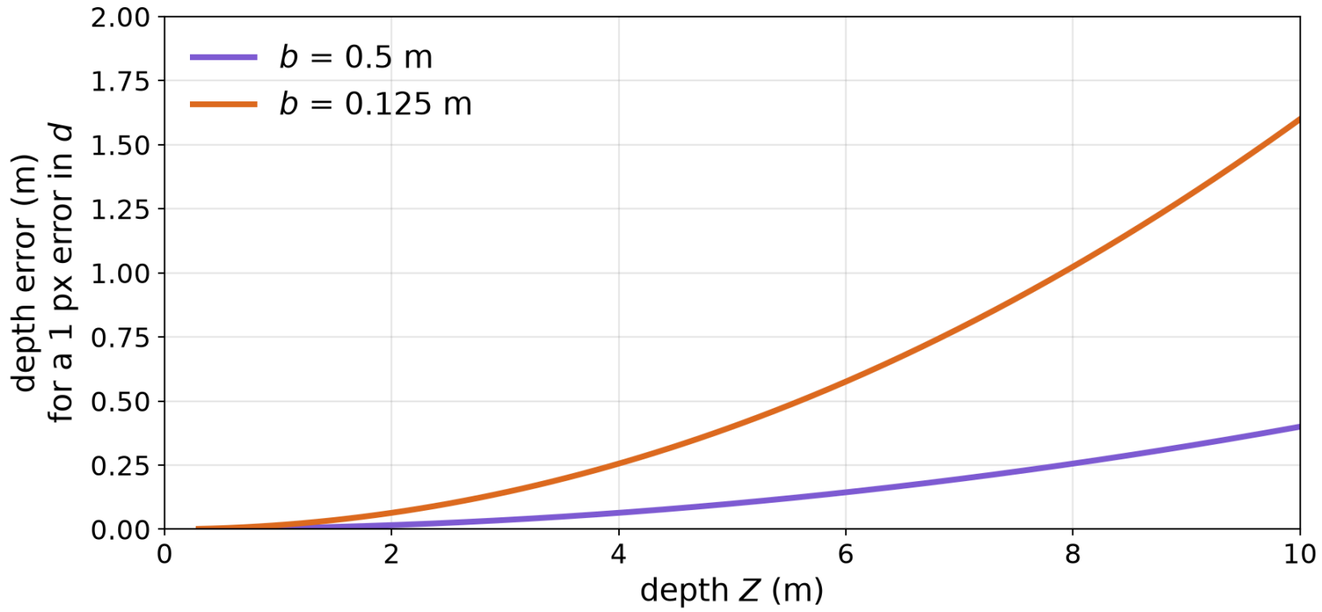
- **Right camera:** the same, with the point at $X - b$ from its axis:

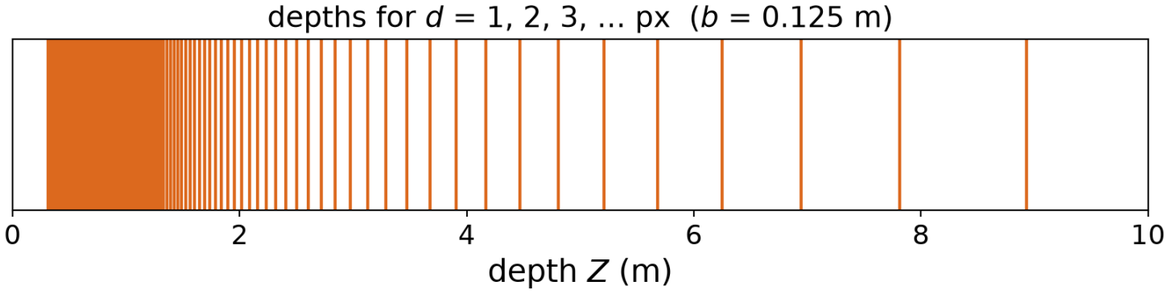
$$\frac{x_R}{f} = \frac{X - b}{Z}$$

- Subtract:

$$d = x_L - x_R = \frac{f\,b}{Z} \quad \Rightarrow \quad Z = \frac{f\,b}{d}$$

Depth Precision Falls With Distance



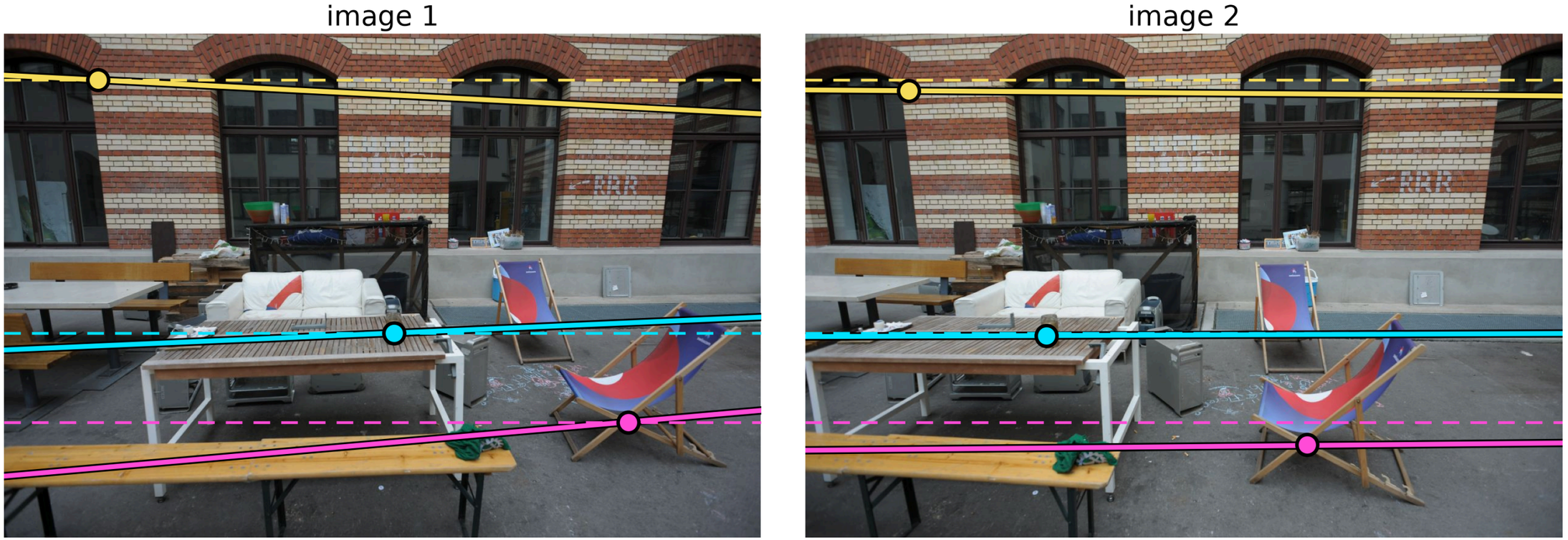


$$\Delta Z \approx \frac{Z^2}{f\,b}\,\Delta d$$

- Error grows with Z^2: twice as far, **four times** the error.
- A longer baseline b or focal length f helps, but makes matching harder (bigger search, more occlusion).
- Same idea as parallax in two-view triangulation.

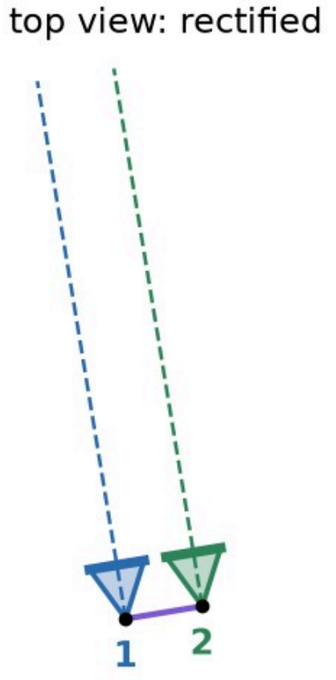
Z	b = **0.5 m**	b = **0.125 m**
2 m	± 2 cm	± 6 cm
5 m	± 10 cm	± 40 cm
10 m	± 40 cm	± 160 cm

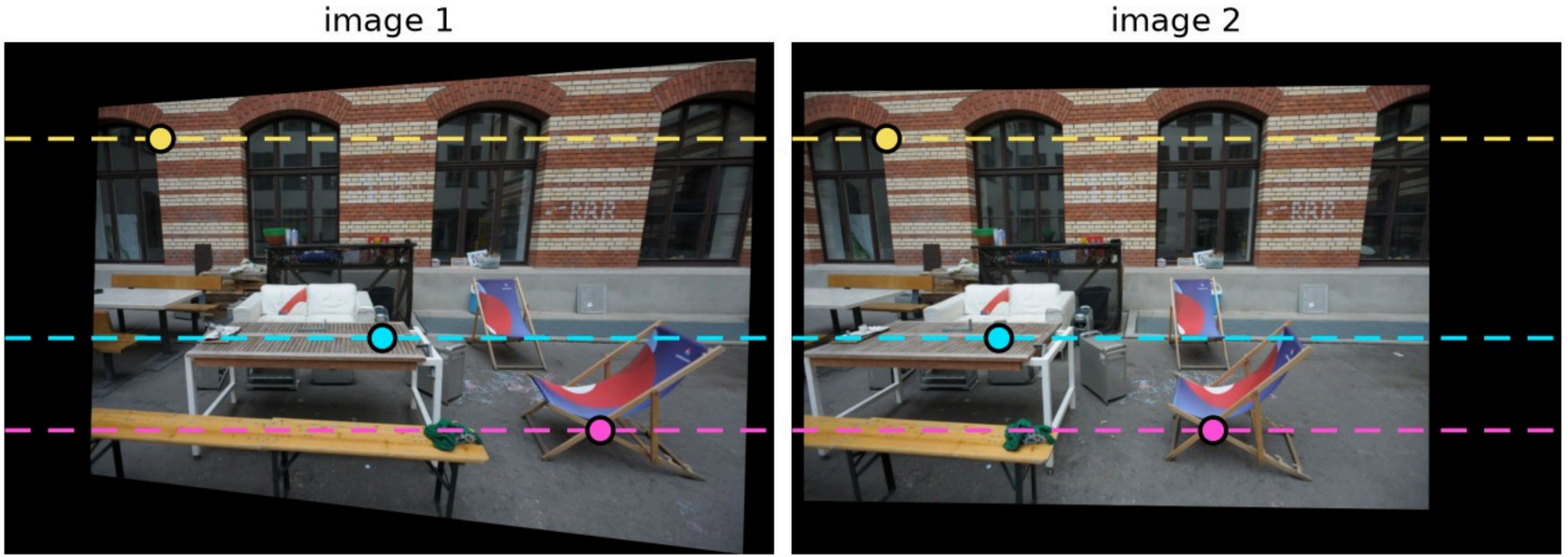
Image Rectification



- If camera 1 and camera 2 are not perfectly aligned **side by side**, the epipolar lines will be slanted in the images.
- A match is **not on the same row** (dashed), but a different slanted epipolar line.
- **Rectification** warps both images so that every epipolar line becomes the **same row** in both.

Rectification: Rotate the Cameras Virtually

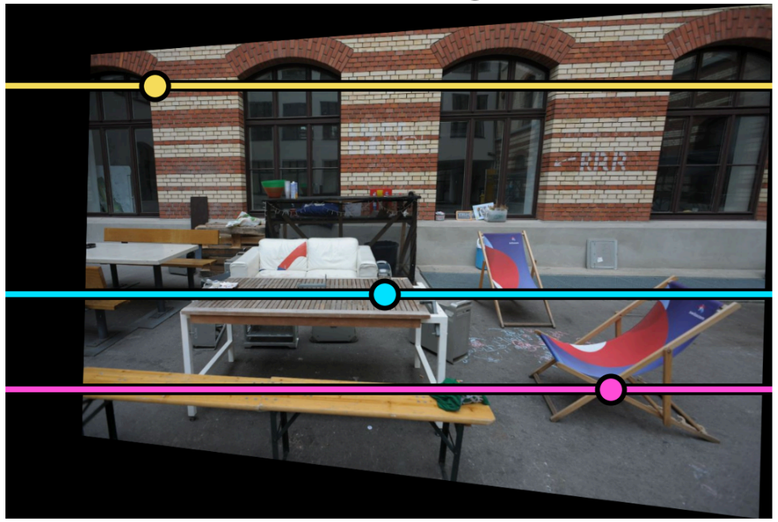




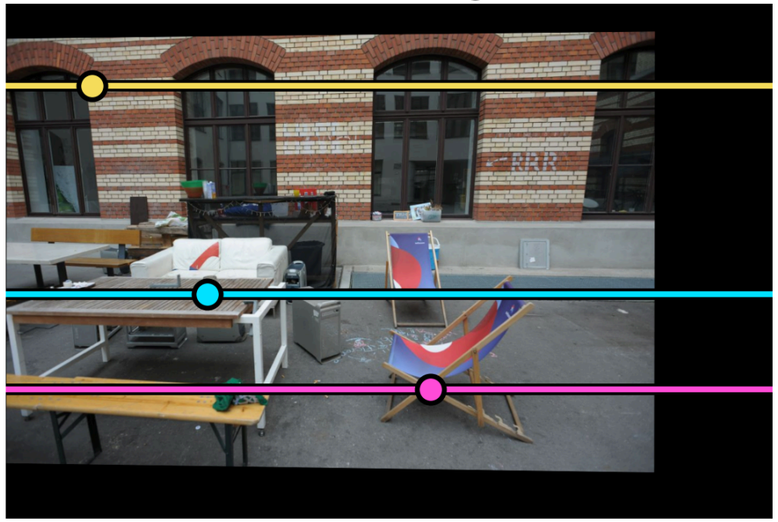
- Turn each camera **about its center** until both face the same way, perpendicular to the baseline (purple).
- Pure rotation $\rightarrow$ each image is warped by a **homography** $\mathbf{H}_i$ (as in two-view).
- The epipoles move to infinity: epipolar lines become **horizontal**, and matches land on the **same row**.

Rectification With OpenCV

rectified image 1



rectified image 2



```python
# R, t: pose of camera 2 relative to camera 1
R1, R2, P1, P2, Q, _, _ = cv2.stereoRectify(
    K, dist, K, dist, (w, h), R, t, alpha=1)
map1 = cv2.initUndistortRectifyMap(
    K, dist, R1, P1, (w, h), cv2.CV_32FC1)
map2 = cv2.initUndistortRectifyMap(
    K, dist, R2, P2, (w, h), cv2.CV_32FC1)
rect1 = cv2.remap(img1, *map1, cv2.INTER_LINEAR)
rect2 = cv2.remap(img2, *map2, cv2.INTER_LINEAR)
```

- `K`: calibration matrix; `dist`: lens distortion; `(w, h)`: image size; `R`, `t`: pose of camera 2 relative to camera 1; `alpha=1`: keep all pixels (`0`: crop to valid pixels).
- `R1`, `R2`: rotations; `P1`, `P2`: new cameras; `Q`: for 3D later.
- `map1`, `map2`: for each rectified pixel, where to sample in the original image (computed once, reused for every frame).
- Uncalibrated? `cv2.stereoRectifyUncalibrated` uses $\mathbf{F}$ and matches instead.

Rectification With OpenCV: Full Code

```python
import cv2
import numpy as np
from scipy.spatial.transform import Rotation

img1 = cv2.imread("assets/14_stereo/frame1.JPG")
img2 = cv2.imread("assets/14_stereo/frame2.JPG")
h, w = img1.shape[:2]

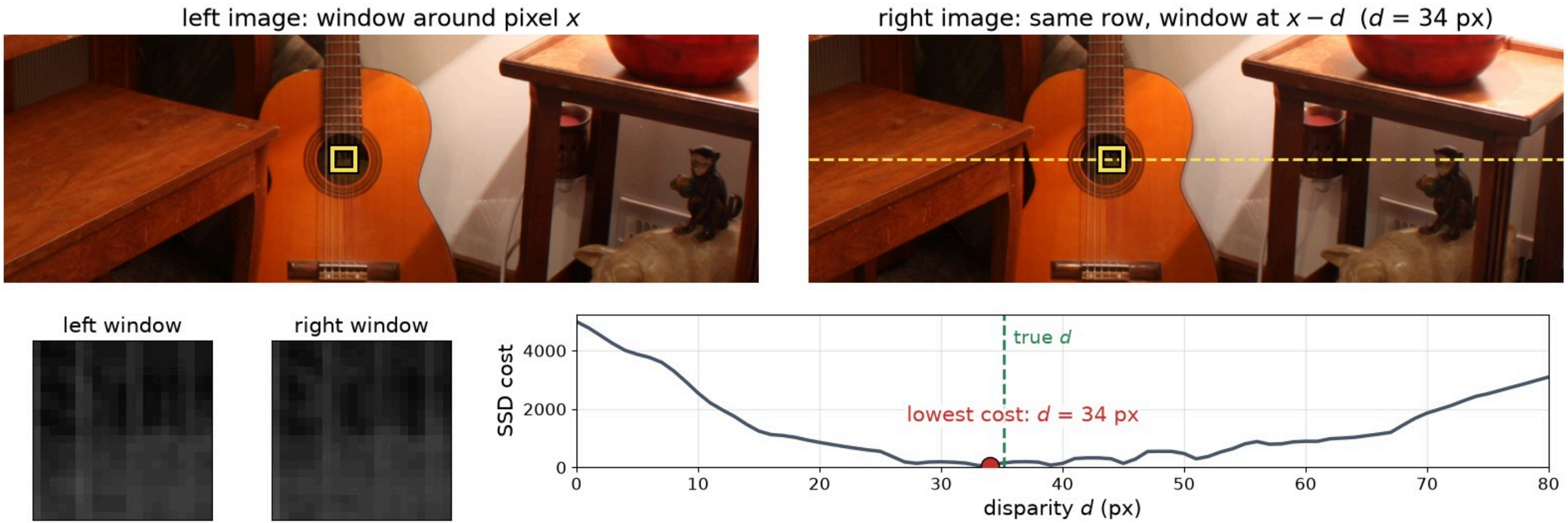
# Calibration: intrinsics (one K, no lens distortion).
K = np.array([[681.916,   0.0,   623.032],
              [  0.0,   681.888, 412.946],
              [  0.0,     0.0,     1.0  ]])
dist = np.zeros(5)

# World-to-camera poses: quaternion (qw, qx, qy, qz) and translation.
R1w = Rotation.from_quat([0.442627, 0.525687, 0.555147, -0.468558], scalar_first=True).as_matrix()
R2w = Rotation.from_quat([0.407402, 0.484639, 0.591031, -0.499831], scalar_first=True).as_matrix()
t1w = np.array([3.22768, 1.01517, 3.73669])
t2w = np.array([2.63246, 1.09683, 3.22009])

# Pose of camera 2 relative to camera 1: x2 = R x1 + t.
R = R2w @ R1w.T
t = (t2w - R @ t1w).reshape(3, 1)

# 1. Rectifying rotations (R1, R2) and new camera matrices (P1, P2).
R1, R2, P1, P2, Q, _, _ = cv2.stereoRectify(K, dist, K, dist, (w, h), R, t, alpha=1)
```

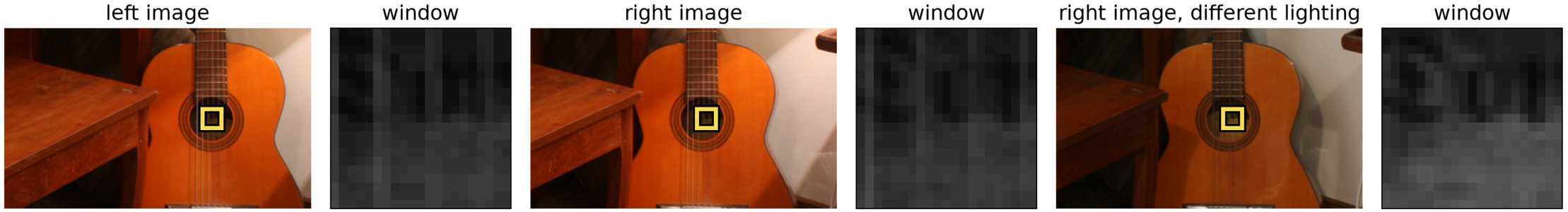
Block Matching: Search for Matches Along the Row



Block Matching

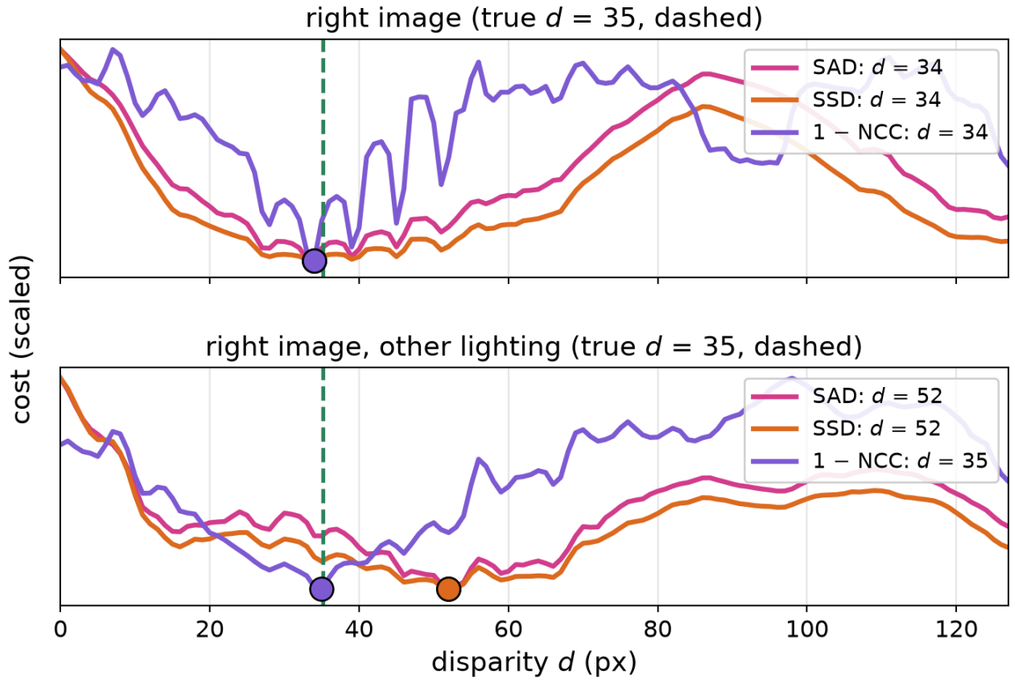
For each pixel (x, y) of the left image: compare its $W \times W$ window with the windows at $(x - d,\ y)$ in the right image for $d = 0, \ldots, D$, and keep the d with the **lowest cost**.

Matching Costs: SAD, SSD, NCC

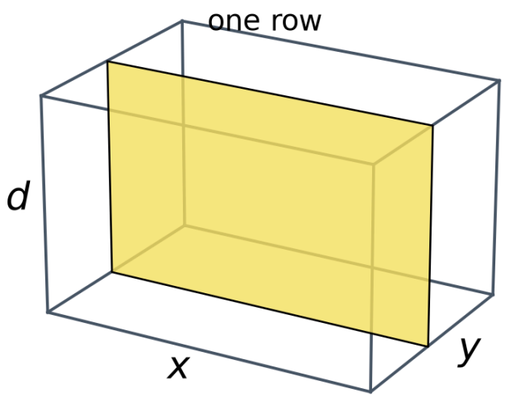


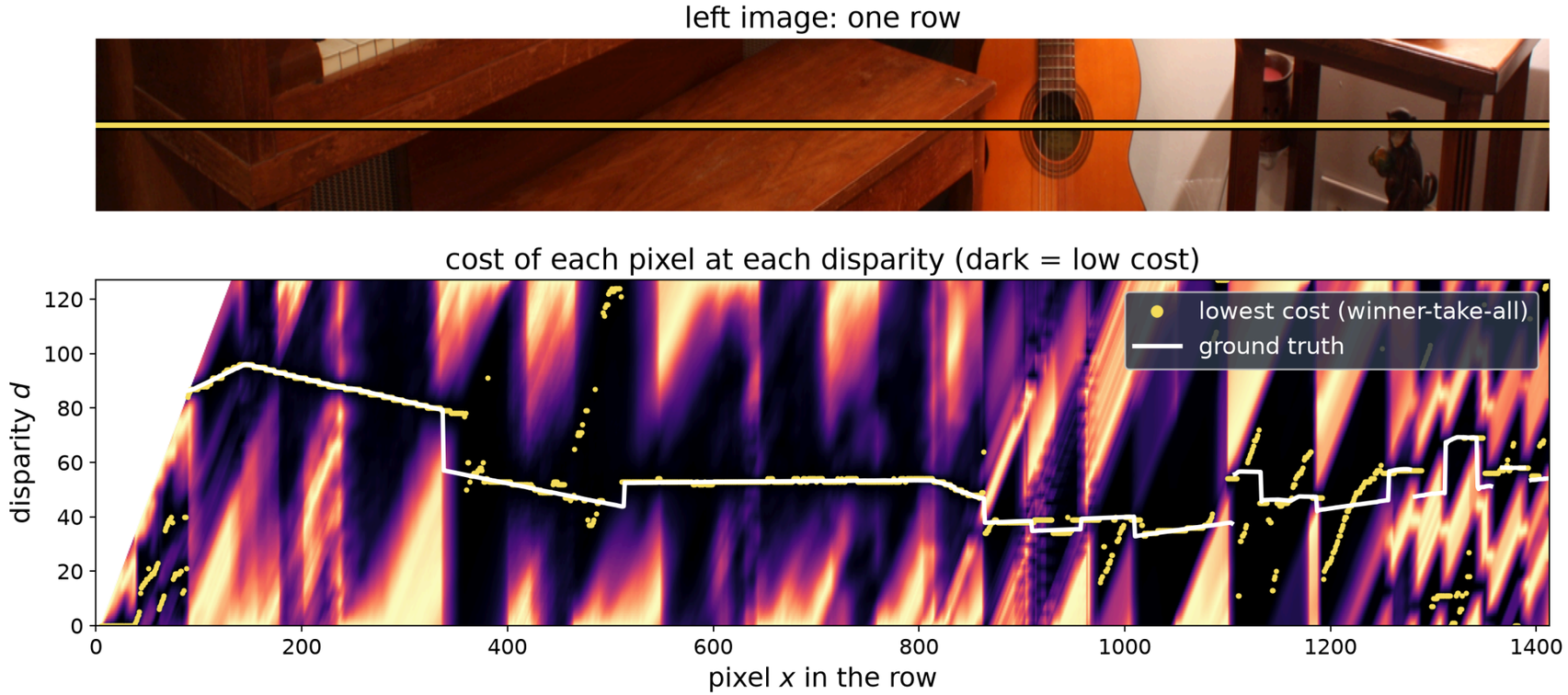
Sums over the window; I_R is taken at $(x - d,\ y)$.

Cost	Formula	Idea
SAD	$\sum \lvert I_L - I_R \rvert$	minimize absolute differences
SSD	$\sum (I_L - I_R)^2$	minimize big differences
NCC	$\dfrac{\sum (I_L - \bar{I}_L)(I_R - \bar{I}_R)}{\sqrt{\sum (I_L - \bar{I}_L)^2 \sum (I_R - \bar{I}_R)^2}}$	maximize correlation; brightness/contrast invariant



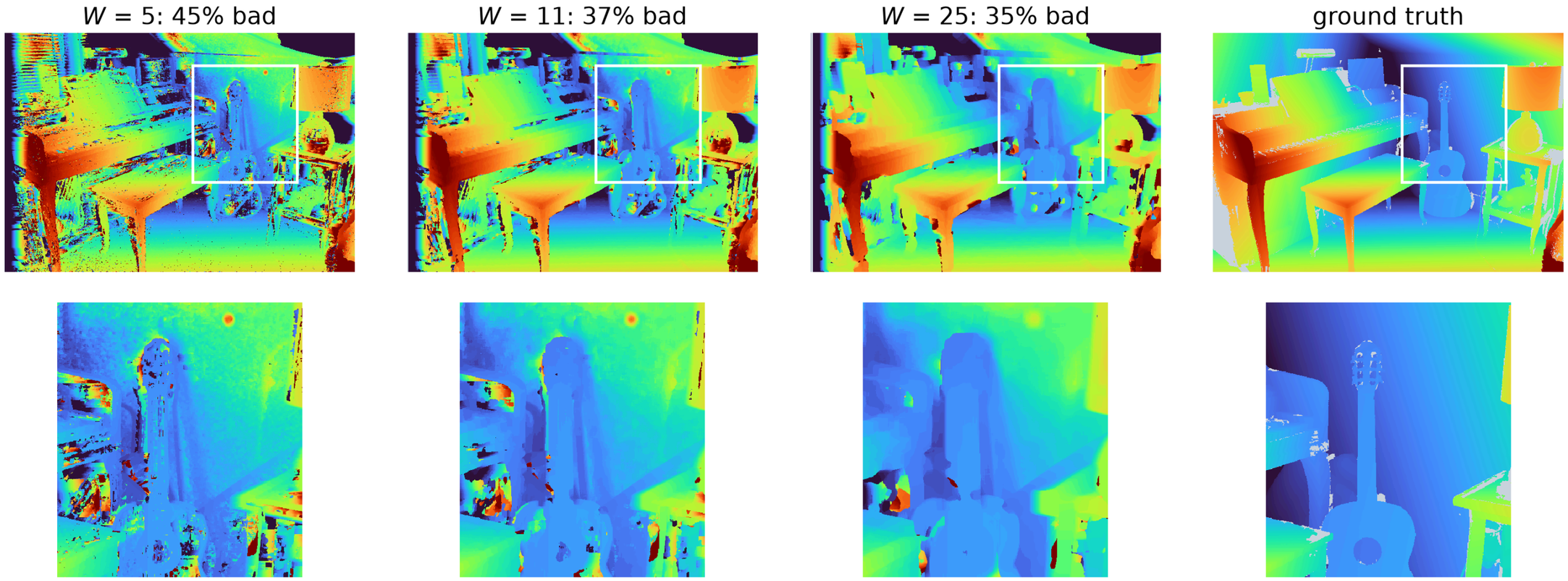
The Cost Volume





- Costs for **every pixel and every disparity** form a rows × columns × disparities **cost volume**.
- **Winner-take-all:** keep the lowest-cost d of each pixel $\rightarrow$ the disparity map.
- Each disparity d is one depth $Z = f\,b\,/\,d$.

Window Size Trade-Off



- **Small window:** detailed, but noisy.
- **Large window:** smooth, but blurs object boundaries (the window mixes two depths).

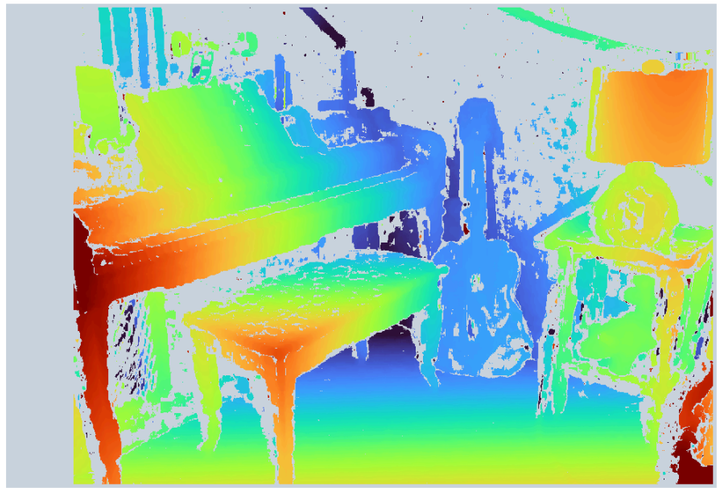
Bad d estimate (> 2 px)	W = 5	W = 11	W = 25
Flat regions	46%	38%	34%
Near object boundaries	41%	35%	39%

Block Matching With OpenCV

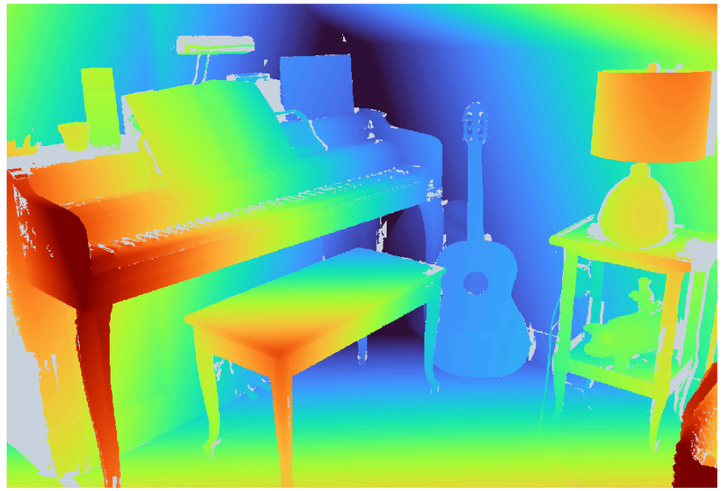
left image



StereoBM: 39% bad



ground truth

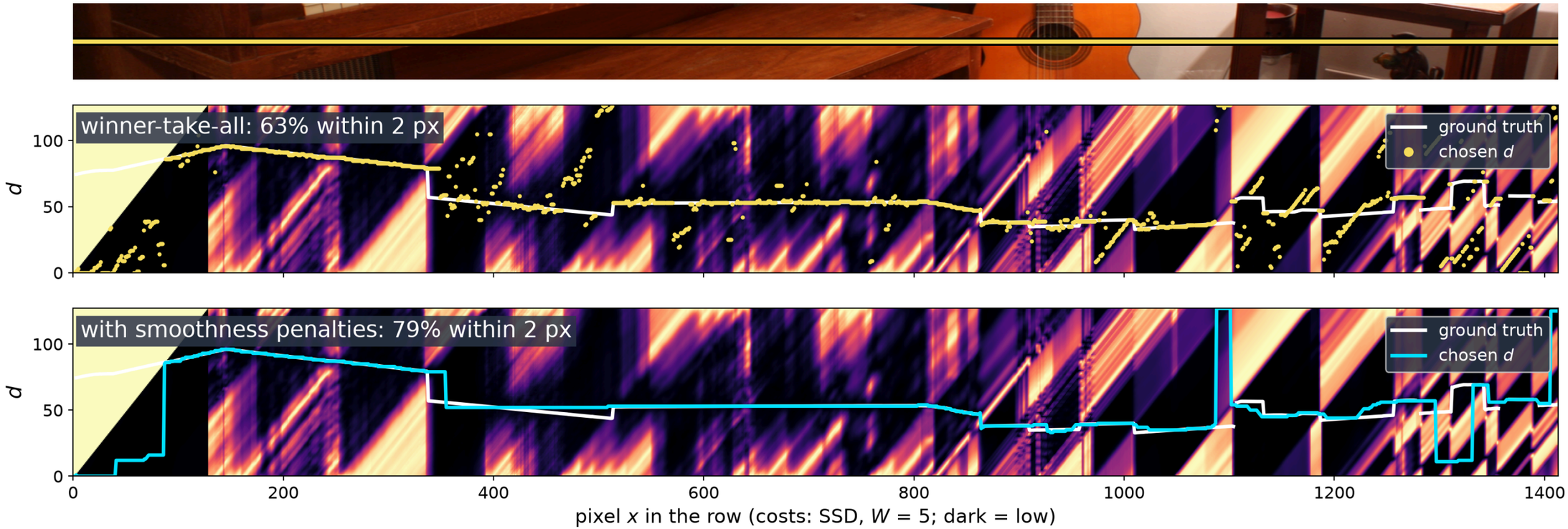


```python
left = cv2.imread("piano_im0.png", cv2.IMREAD_GRAYSCALE)
right = cv2.imread("piano_im1.png", cv2.IMREAD_GRAYSCALE)

bm = cv2.StereoBM_create(numDisparities=128,
                         blockSize=15)
disp = bm.compute(left, right).astype(np.float32) / 16
disp[disp < 0] = np.nan          # no match found
```

- `numDisparities`: search range D; OpenCV requires a multiple of 16. `blockSize`: window W (odd).
- Output is an integer equal to 16 × disparity (1/16 px precision): divide by 16.
- Gray: no match (flat or unreliable areas).
- 16 ms for a 1414 × 962 image.

Adding Smoothness



$$E = \sum_{\mathbf{p}} C(\mathbf{p}, d_{\mathbf{p}}) \; + \sum_{\text{neighbors } \mathbf{p},\mathbf{q}} \begin{cases} 0 & d_{\mathbf{p}} = d_{\mathbf{q}} \\ P_1 & |d_{\mathbf{p}} - d_{\mathbf{q}}| = 1 \\ P_2 & |d_{\mathbf{p}} - d_{\mathbf{q}}| > 1 \end{cases}$$

- Choose all d **together**: E is total disparity map cost; pick the map with the lowest E.
- $C(\mathbf{p}, d_{\mathbf{p}})$: matching cost of pixel $\mathbf{p}$ at its chosen disparity $d_{\mathbf{p}}$. $\mathbf{q}$: a neighbor of $\mathbf{p}$.
- Small P_1: allows gradual changes (slanted surfaces). Large P_2: a jump needs strong evidence (object edges, not noise).

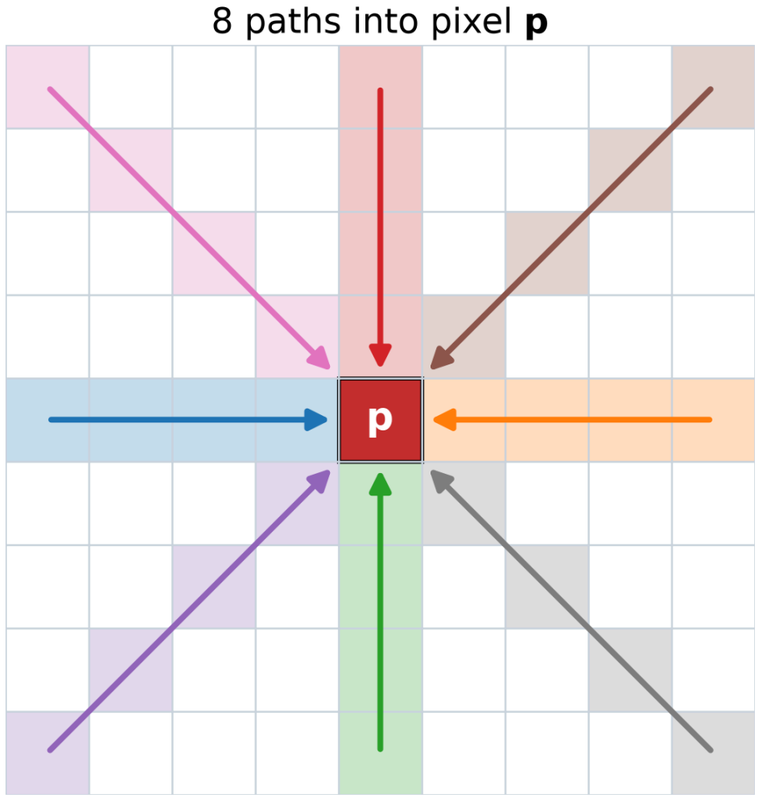
Semi-Global Matching (SGM)

Semi-Global Matching

1. Compute the **cost volume** $C(\mathbf{p}, d)$.
2. Along each of **8 straight paths** into every pixel (left → right, top → bottom, diagonals, …), add up the costs with the P_1 & P_2 penalties.
3. **Sum** the 8 results: $S(\mathbf{p}, d)$.
4. **Winner-take-all** on S: $d_{\mathbf{p}} = \arg\min_d S(\mathbf{p}, d)$.

H. Hirschmüller, *Stereo Processing by Semiglobal Matching and Mutual Information*, IEEE TPAMI, 2008.

- The best d for the **whole image at once** is too expensive to find; 8 one-dimensional paths are a fast approximation: **semi**-global.

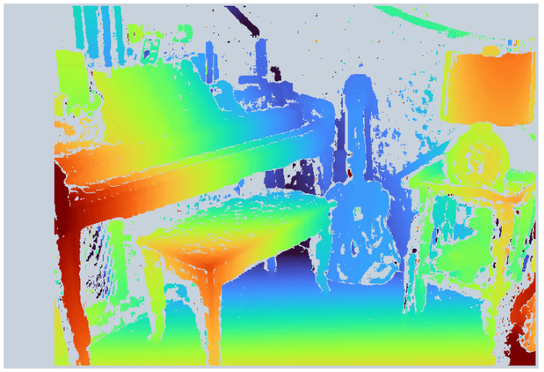


StereoBM vs. StereoSGBM in OpenCV

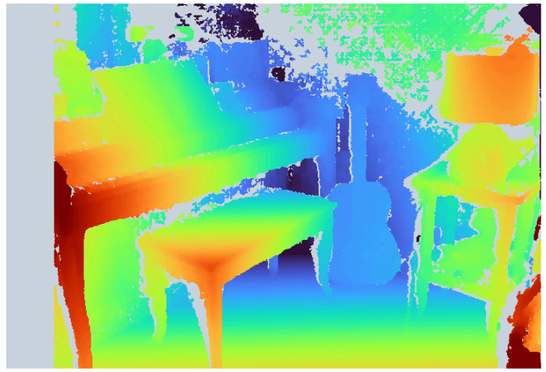
left image



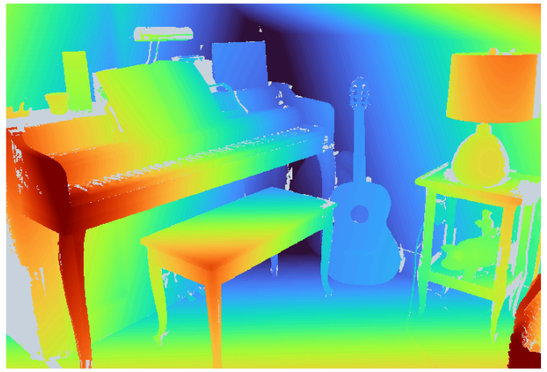
StereoBM: 39% bad



StereoSGBM: 26% bad



ground truth



```python
left = cv2.imread("piano_im0.png")       # color: 3 channels
right = cv2.imread("piano_im1.png")
W = 5
sgbm = cv2.StereoSGBM_create(
    minDisparity=0, numDisparities=128, blockSize=W,
    P1=8 * 3 * W**2, P2=32 * 3 * W**2,  # penalties
    uniquenessRatio=10, speckleWindowSize=100,
    speckleRange=2, mode=cv2.STEREO_SGBM_MODE_HH)
disp = sgbm.compute(left, right).astype(np.float32) / 16
disp[disp < 0] = np.nan                  # no match found
```

	Bad pixels	Runtime
StereoBM	39%	16 ms
StereoSGBM	**26%**	**660** ms

- `minDisparity`, `numDisparities`: search range $d = 0, \ldots, 127$ px; must cover the nearest and farthest points.
- `P1`, `P2`: the smoothness penalties ($\times$ channels $\times W^2$, as suggested by OpenCV).
- `MODE_HH`: all 8 paths (the default mode uses 5, faster).
- `uniquenessRatio`, `speckle…`: drop unreliable matches (gray).

Dense Stereo With OpenCV: Full Code

```python
import time
import cv2
import numpy as np

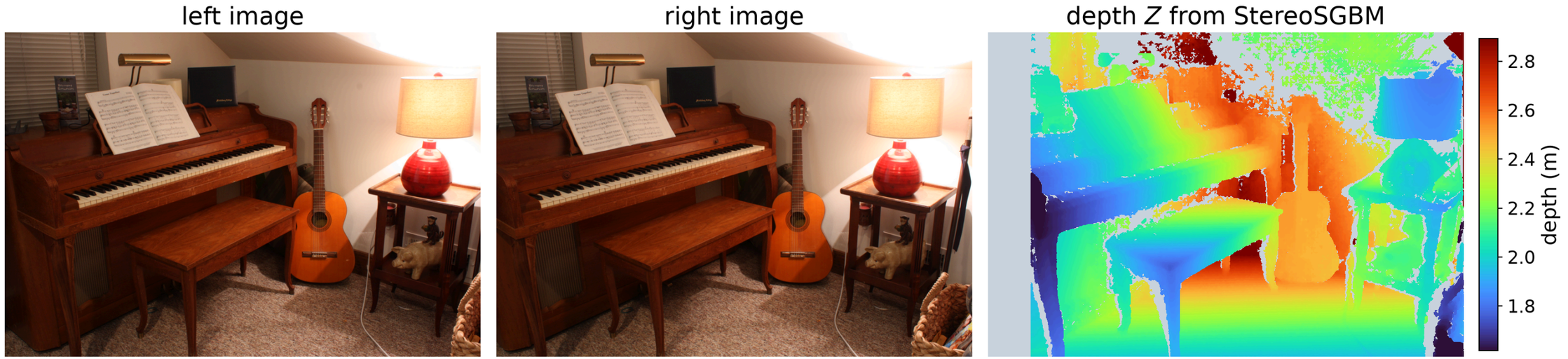
def read_pfm(path):
    # Middlebury ground truth: float32, bottom row first; unknown = inf.
    with open(path, "rb") as f:
        f.readline()                                    # "Pf": one channel
        width, height = map(int, f.readline().split())
        scale = float(f.readline())                     # negative: little-endian
        data = np.fromfile(f, "<f4" if scale < 0 else ">f4").reshape(height, width)
    return np.flipud(data)

# A rectified pair, its ground-truth disparity, and the mask of pixels both cameras see.
left = cv2.imread("assets/14_stereo/piano_im0.png")
right = cv2.imread("assets/14_stereo/piano_im1.png")
gt = read_pfm("assets/14_stereo/piano_disp0GT.pfm")
visible = cv2.imread("assets/14_stereo/piano_mask0nocc.png", cv2.IMREAD_GRAYSCALE) == 255
scored = visible & np.isfinite(gt)

def bad_percent(disp):
    # % of scored pixels whose disparity is off by more than 2 px (or missing).
    return 100 * np.mean(~(np.abs(disp - gt) <= 2)[scored])

# 1. Block matching (grayscale).
bm = cv2.StereoBM_create(numDisparities=128, blockSize=15)
start = time.perf_counter()
```

From Disparity to 3D



$$Z = \frac{f\,b}{d}, \quad X = \frac{(x - c_x)\,Z}{f}, \quad Y = \frac{(y - c_y)\,Z}{f}; \quad [\,X' \quad Y' \quad Z' \quad W\,]^\top = \mathbf{Q}[\,x \quad y \quad d \quad 1\,]^\top$$

- Go out along each pixel's ray to depth Z: one 3D point per pixel.
- The 4×4 $\mathbf{Q}$ comes from `cv2.stereoRectify`.
- $(X, Y, Z) = (X', Y', Z')/W$.

Point Cloud With OpenCV

```
# disp: StereoSGBM (px); Q: stereoR
points = cv2.reprojectImageTo3D(dis
colors = left[..., ::-1]  # BGR ->
valid = ~np.isnan(disp)   # matched
xyz = points[valid]       # N x 3 (
rgb = colors[valid]       # N x 3
```

- 1,059,037 colored 3D points.
- **Holes:** no match found.
- **Shadows** behind objects: seen by only one camera.



Stereo Reconstruction: Full Code

```python
import cv2
import numpy as np

left = cv2.imread("assets/14_stereo/piano_im0.png")
right = cv2.imread("assets/14_stereo/piano_im1.png")
h, w = left.shape[:2]

# Calibration: camera matrices of both cameras, and the baseline (mm in the file).
calib = dict(line.strip().split("=", 1) for line in open("assets/14_stereo/piano_calib.txt") if "=" in line
def to_matrix(text):
    return np.array([[float(v) for v in row.split()] for row in text.strip("[]").split(";")])
K1, K2 = to_matrix(calib["cam0"]), to_matrix(calib["cam1"])
b = float(calib["baseline"]) / 1000
dist = np.zeros(5)

# 1. Rectify. This rig is already side by side (R = I, right camera b to the right), so with flags=0
#    the warp leaves the images unchanged, and stereoRectify mainly provides Q.
R, t = np.eye(3), np.array([[-b], [0.0], [0.0]])
R1, R2, P1, P2, Q, _, _ = cv2.stereoRectify(K1, dist, K2, dist, (w, h), R, t, flags=0)
map1 = cv2.initUndistortRectifyMap(K1, dist, R1, P1, (w, h), cv2.CV_32FC1)
map2 = cv2.initUndistortRectifyMap(K2, dist, R2, P2, (w, h), cv2.CV_32FC1)
rect1 = cv2.remap(left, *map1, cv2.INTER_LINEAR)
rect2 = cv2.remap(right, *map2, cv2.INTER_LINEAR)

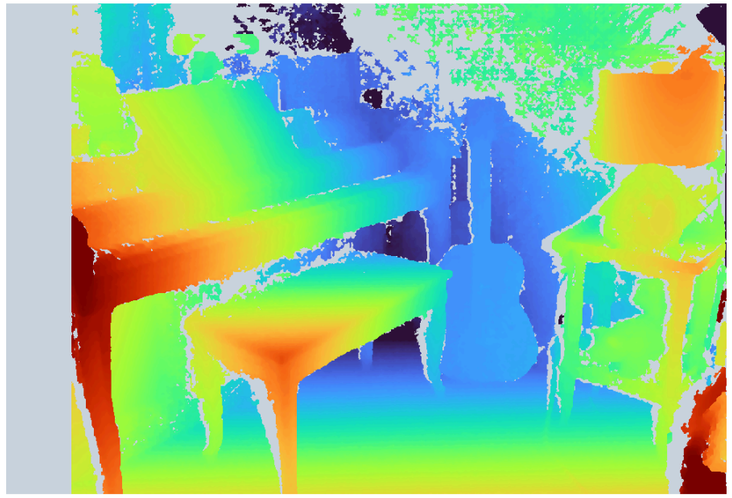
# 2. Match along rows: disparity in pixels.
W = 5
sgbm = cv2.StereoSGBM_create(minDisparity=0, numDisparities=128, blockSize=W
```

Limitations of Stereo

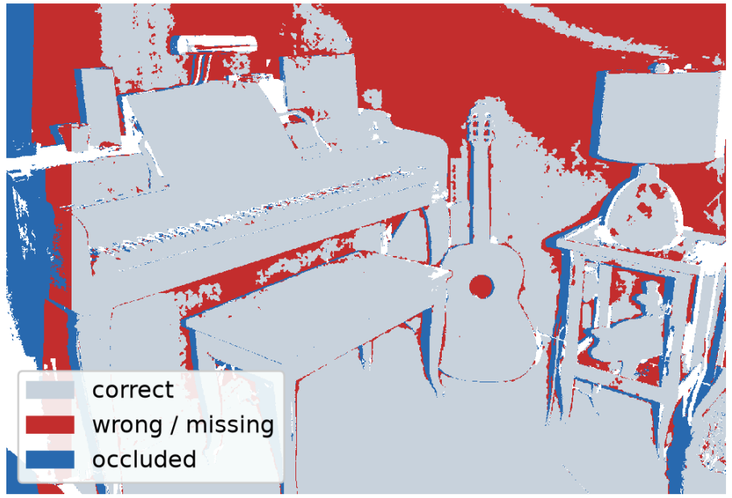
left image



StereoSGBM disparity



where it fails



StereoSGBM on Piano	Pixels	Bad
Textured	32%	**9%**
Textureless	38%	**41%**
Near object boundaries	17%	**23%**
Occluded (one camera only)	8%	**93%**

- **No texture, repeated patterns** (plain walls, piano keys): many d match equally well.
- **Occlusions**: no match exists. **Reflections**, glare: the two views differ.
- **Boundaries**: windows mix two depths.
- **Far points**: error grows with Z^2 .
- **Deep learning** stereo learns the matching and now lead the Middlebury benchmark.

Key Takeaways

- **Stereo** = two calibrated cameras side by side: a match for **every pixel** gives a dense depth map.
- **Disparity** is the sideways shift of a point between the images: $Z = f\,b\,/\,d$. Near points shift more; depth error grows with Z^2.
- **Rectification** rotates both cameras virtually (one homography per image) so every epipolar line becomes the **same row**.
- **Block matching** slides a window along the row and keeps the lowest cost (winner-take-all). NCC handles lighting changes; window size trades detail for smoothness.
- **SGM** adds smoothness penalties (P_1, P_2) along 8 paths: far fewer bad pixels than block matching, at a higher cost.
- $\mathbf{Q}$ and `reprojectImageTo3D` turn disparity into a 3D point cloud. **Holes**: textureless regions, reflections, and areas seen by only one camera.