EE 046746 - Technion - Computer Vision

Elias Nehme

Tutorial 11 - Stereo Imaging

Agenda

Triangulation

Reconstructing 3D points

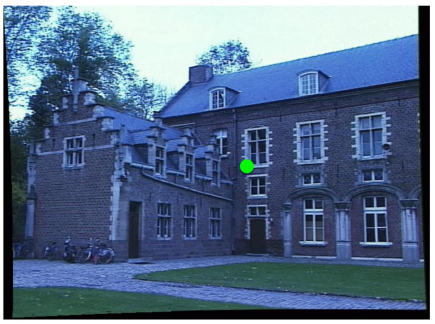
How would you reconstruct 3D points?

Left image

Right image

Reconstructing 3D points

How would you reconstruct 3D points?



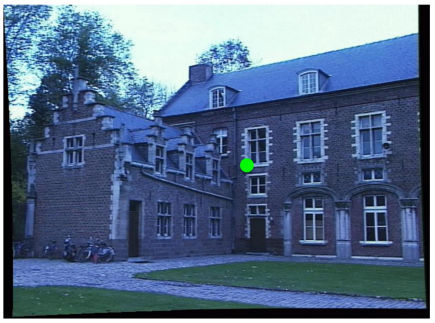
Left image

Right image

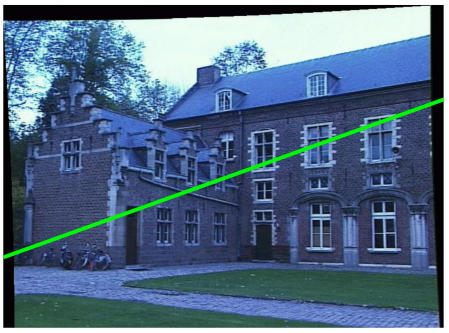
1. Select point in one image (how?)

Reconstructing 3D points

How would you reconstruct 3D points?



Left image

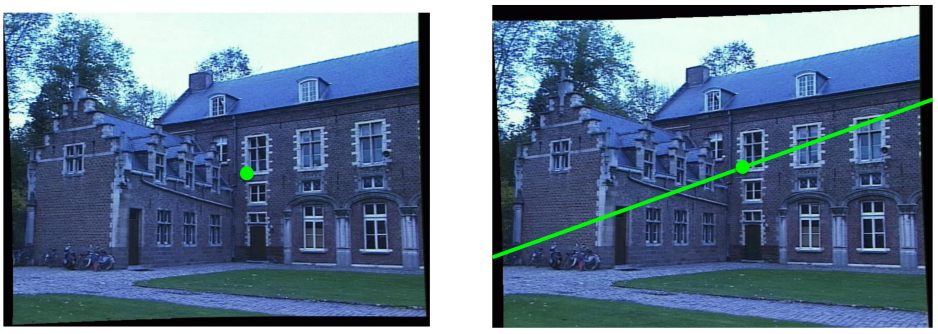


Right image

1. Select point in one image (how?)
2. Form epipolar line for that point in second image (how?)

Reconstructing 3D points

How would you reconstruct 3D points?

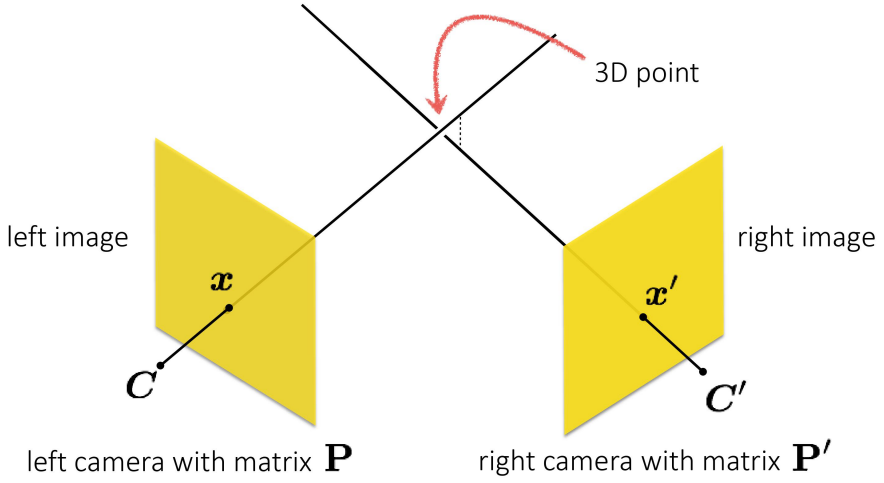


Left image Right image

1. Select point in one image (how?)
2. Form epipolar line for that point in second image (how?)
3. Find matching point along line (how?)

Reconstructing 3D points

Triangulation



Reconstructing 3D points

- Switch to row-wise representation of the projection matrix:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \leftrightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} - & m_1^T & - \\ - & m_2^T & - \\ - & m_3^T & - \end{bmatrix} P$$

- Use the fact that the cross product should be zero:

$$p \times MP = 0$$

Reconstructing 3D points

- Leads to two independent equations:

$$p \times MP = 0 \rightarrow \begin{bmatrix} ym_3^T P - m_2^T P \\ m_1^T P - xm_3^T P \end{bmatrix} = 0$$

- Using the matched point in image 2 as well $\rightarrow$ Solution by SVD:

$$\begin{bmatrix} ym_3^T - m_2^T \\ m_1^T - xm_3^T \\ \tilde{y}\tilde{m}_3^T - \tilde{m}_2^T \\ \tilde{m}_1^T - \tilde{x}\tilde{m}_3^T \end{bmatrix} P = 0 \leftrightarrow AP = 0$$

Stereo Concept

Depth vs Disparity

What's different between these two images?

Depth vs Disparity

Depth vs Disparity

Depth vs Disparity


Objects that are close move more or less?

Depth vs Disparity

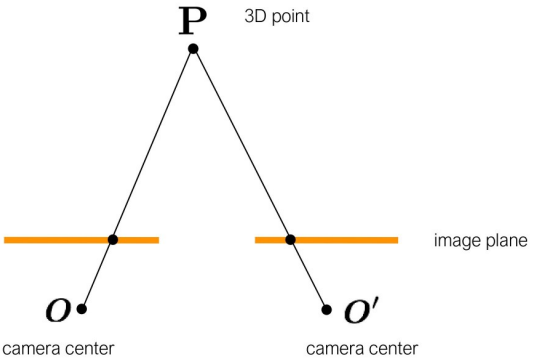
The amount of horizontal movement is
inversely proportional to …



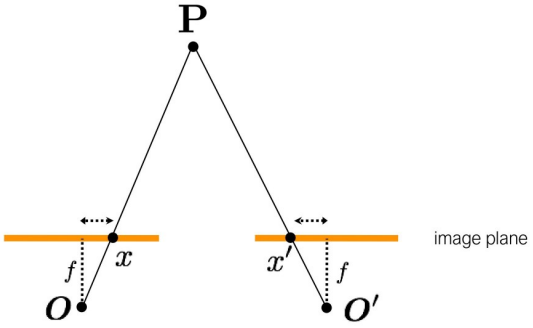
… the distance from the camera.

More formally…

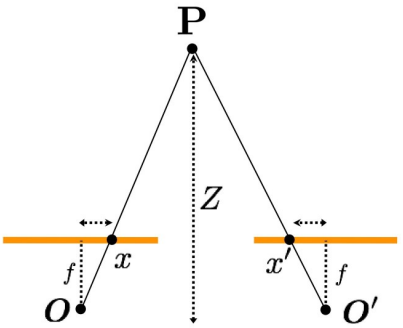
Depth vs Disparity



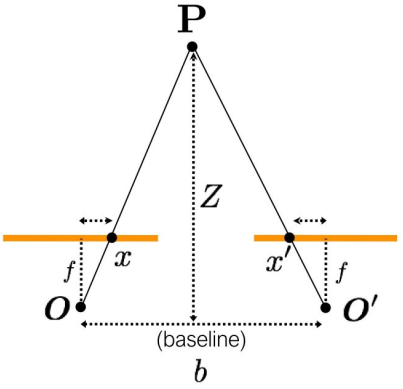
Depth vs Disparity



Depth vs Disparity

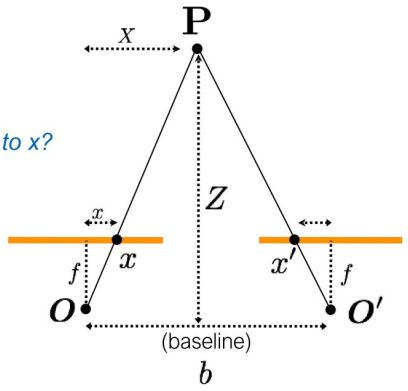


Depth vs Disparity



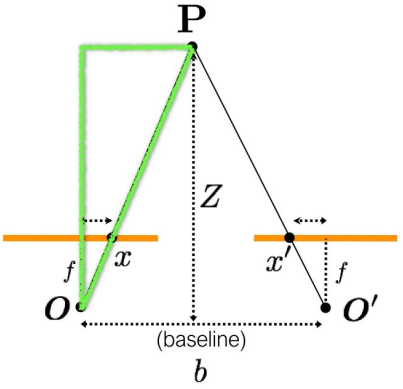
Depth vs Disparity

How is X related to x?



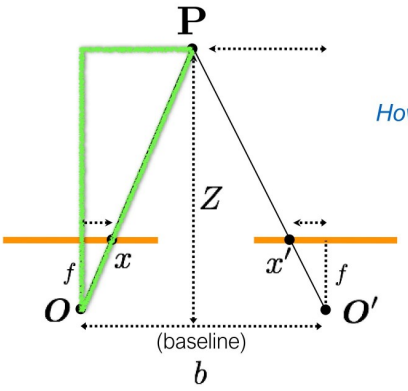
Depth vs Disparity

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



Depth vs Disparity

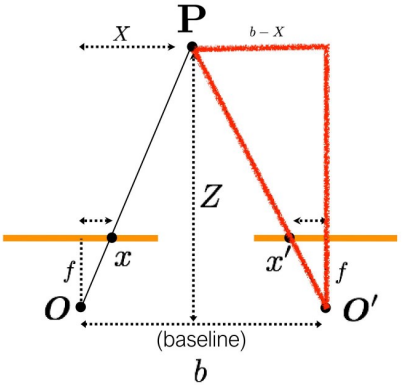
$$\frac{X}{Z} = \frac{x}{f}$$



How is X related to x'?

Depth vs Disparity

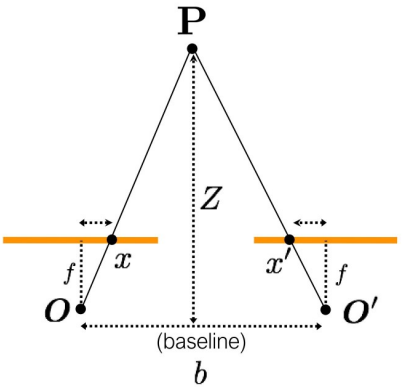
$$\frac{X}{Z} = \frac{x}{f}$$



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

Depth vs Disparity

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



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

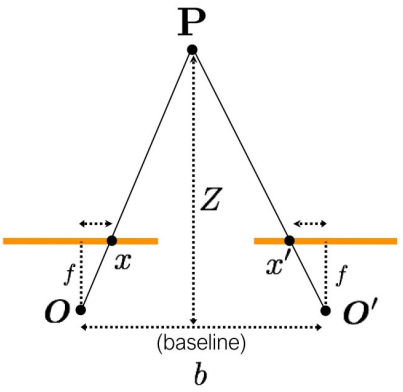
Disparity

$$d = x - x'$$
$$= \frac{bf}{Z}$$

inversely proportional to depth

Depth vs Disparity

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



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

Disparity

$$d = x - x'$$
$$= \frac{bf}{Z}$$

inversely proportional to depth

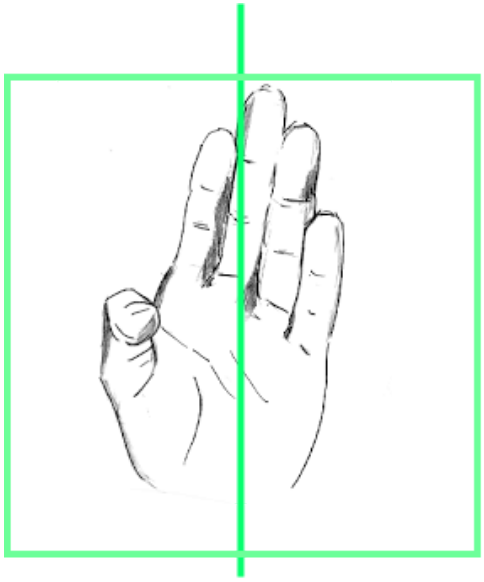
Depth vs Disparity: Application 1

Subaru
Eyesight system

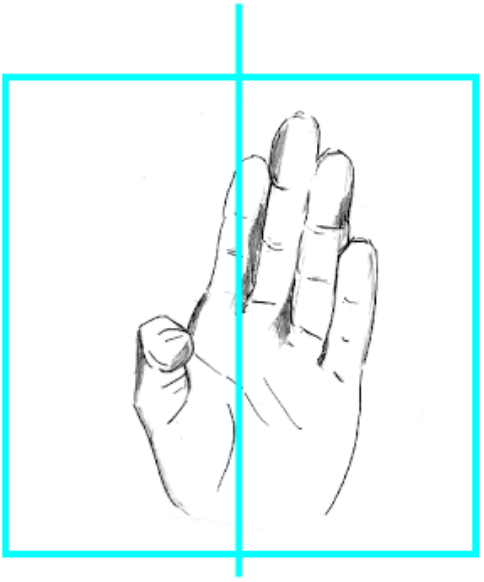
Pre-collision
braking

Depth vs Disparity: Application 1

Depth vs Disparity: Intuition?



Left eye closed. Right Eye open

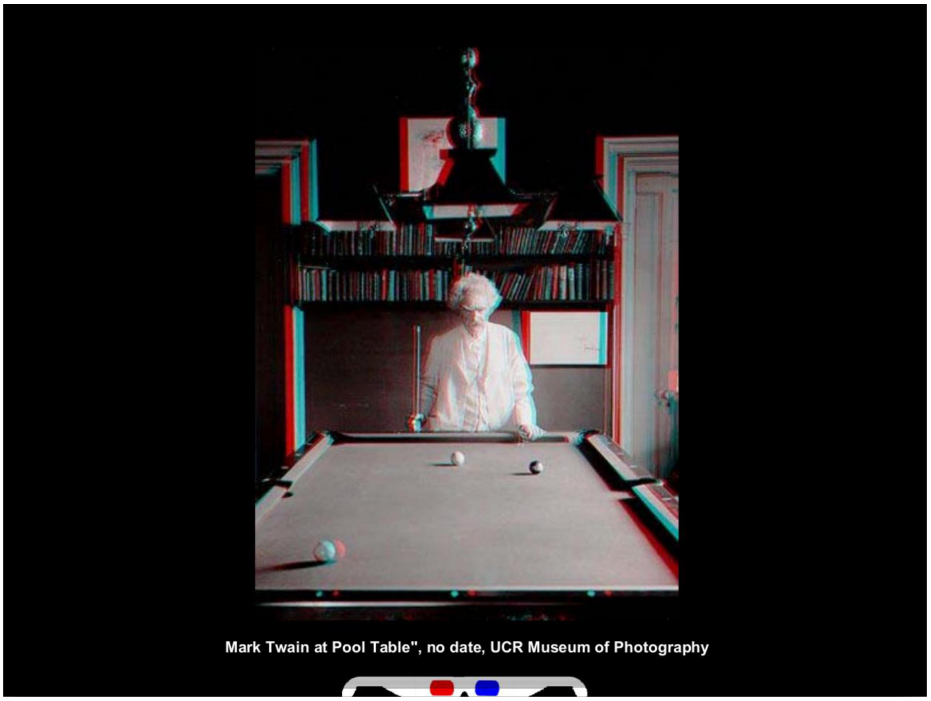


Right eye closed. Left eye open

- Image Source

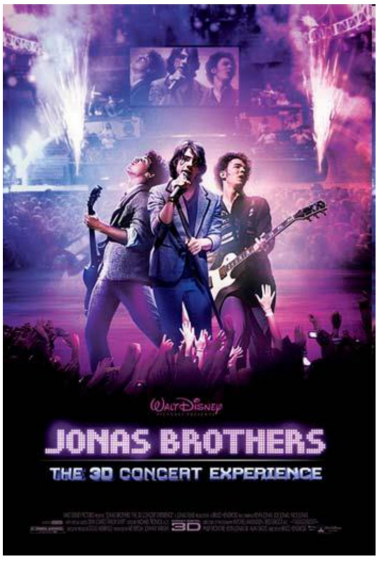
Depth vs Disparity: Application 2

Depth vs Disparity: Application 2



Depth vs Disparity: Application 2

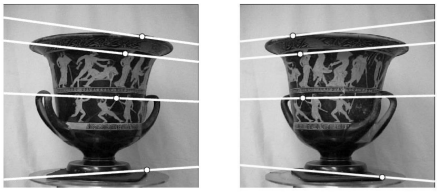
This is how 3D movies work



Stereo Rectification

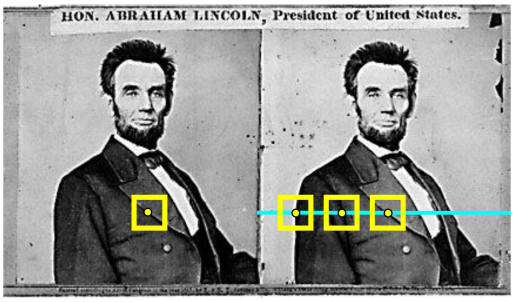
Stereo Rectification

So can I compute depth from any two images of the same object?



1. Need sufficient baseline
2. Images need to be 'rectified' first (make epipolar lines horizontal)

Stereo Rectification

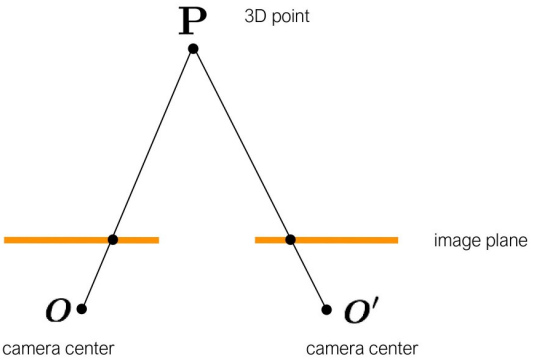


1. Rectify images
   (make epipolar lines horizontal)
2. For each pixel
   a. Find epipolar line
   b. Scan line for best match
   c. Compute depth from disparity

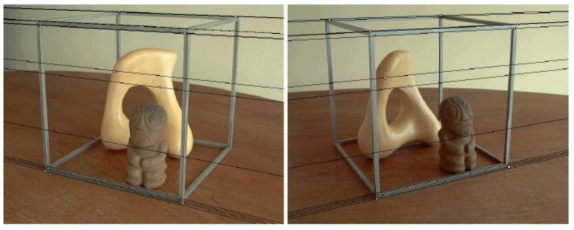
$$Z = \frac{bf}{d}$$

Stereo Rectification

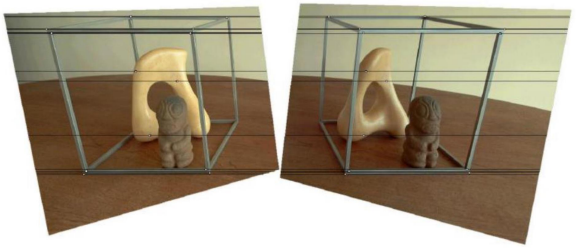
- What is special about these two cameras?
  - they have the same image plane: No rotation, only horizontal translation.



Stereo Rectification



How can you make the epipolar lines horizontal?



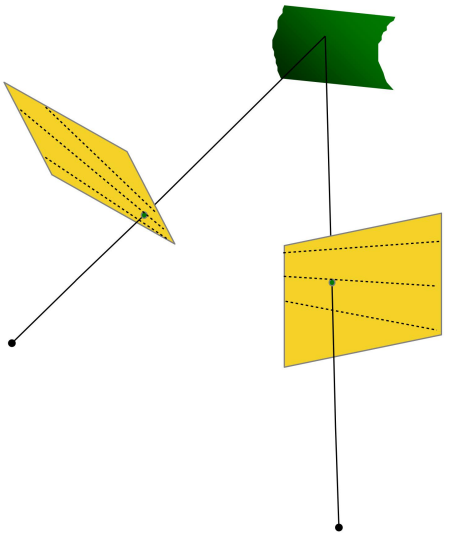
Stereo Rectification

Stereo Rectification

1. **Rotate** the right camera by $\mathbf{R}^T$
   (aligns camera coordinate system orientation only)
2. Rotate (**rectify**) the left camera so that the epipole is at infinity
3. Rotate (**rectify**) the right camera so that the epipole is at infinity
4. Adjust the **scale**

Stereo Rectification

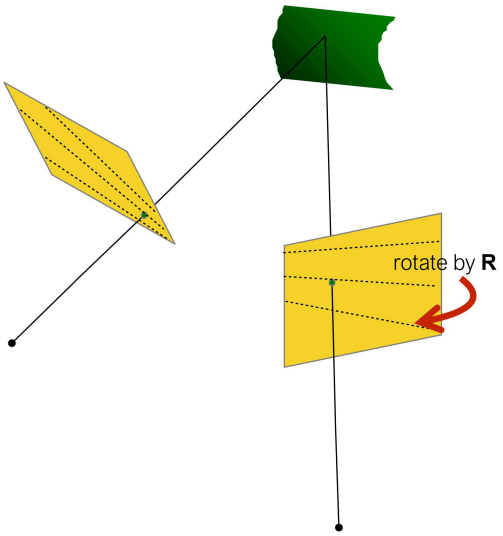
Stereo Rectification:



1. Compute **E** to get **R**
2. Rotate right image by **R**
3. Rotate both images by $\mathbf{R}_{rect}$
4. Scale both images by **H**

Stereo Rectification

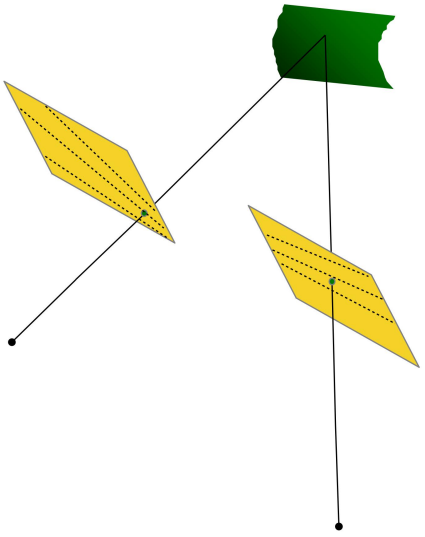
Stereo Rectification:



1. Compute **E** to get **R**
2. Rotate right image by **R**
3. Rotate both images by $\mathbf{R}_{rect}$
4. Scale both images by **H**

Stereo Rectification

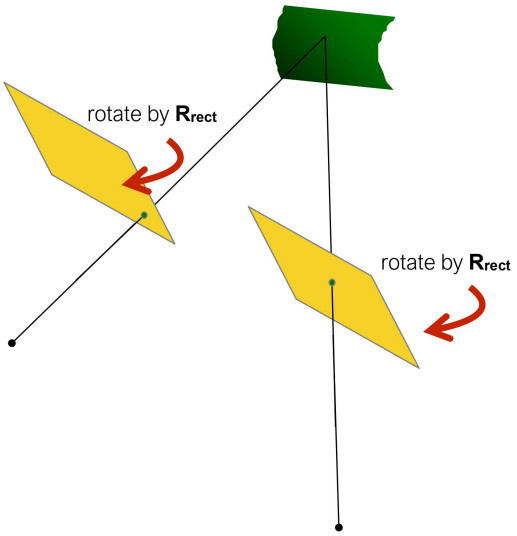
Stereo Rectification:



1. Compute **E** to get **R**
2. Rotate right image by **R**
3. Rotate both images by $\mathbf{R}_{rect}$
4. Scale both images by **H**

Stereo Rectification

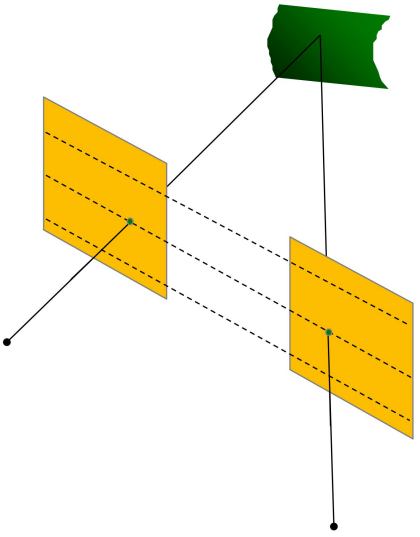
Stereo Rectification:



1. Compute **E** to get **R**
2. Rotate right image by **R**
3. Rotate both images by $\mathbf{R}_{rect}$
4. Scale both images by **H**

Stereo Rectification

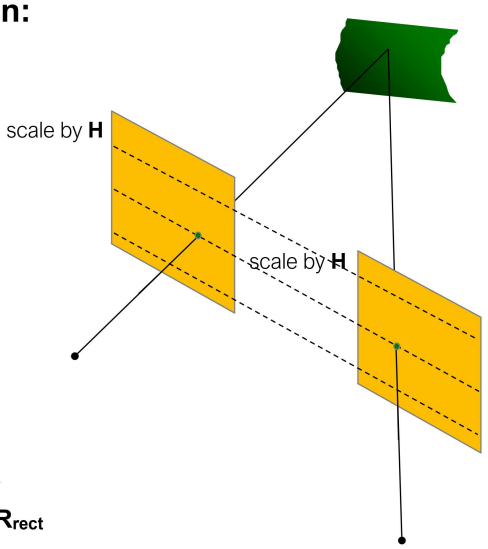
Stereo Rectification:



1. Compute **E** to get **R**
2. Rotate right image by **R**
3. Rotate both images by $\mathbf{R}_{rect}$
4. Scale both images by **H**

Stereo Rectification

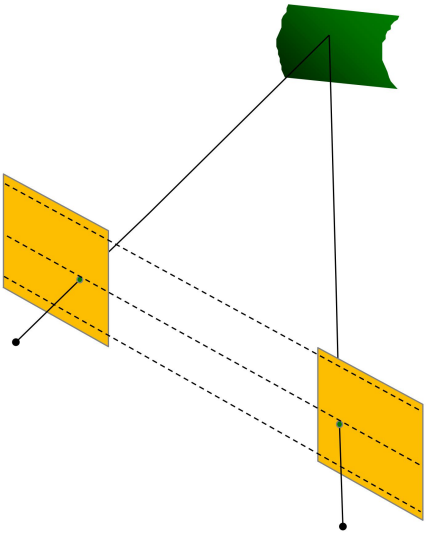
Stereo Rectification:



1. Compute **E** to get **R**
2. Rotate right image by **R**
3. Rotate both images by $\mathbf{R}_{\mathbf{rect}}$
4. Scale both images by **H**

Stereo Rectification

Stereo Rectification:



1. Compute **E** to get **R**
2. Rotate right image by **R**
3. Rotate both images by $\mathbf{R}_{\mathbf{rect}}$
4. Scale both images by **H**

Stereo Rectification

Setting the epipole to infinity

(Building $\mathbf{R}_{\text{rect}}$ from **e**)

Let $R_{\text{rect}} = \begin{bmatrix} \boldsymbol{r}_1^\top \\ \boldsymbol{r}_2^\top \\ \boldsymbol{r}_3^\top \end{bmatrix}$ Given: epipole **e** (translation from **E**)

$$\boldsymbol{r}_1 = \boldsymbol{e}_1 = \frac{T}{||T||}$$ epipole coincides with translation vector

$$\boldsymbol{r}_2 = \frac{1}{\sqrt{T_x^2 + T_y^2}} \begin{bmatrix} -T_y & T_x & 0 \end{bmatrix}$$ cross product of e and the direction vector of the optical axis

$$\boldsymbol{r}_3 = \boldsymbol{r}_1 \times \boldsymbol{r}_2$$ orthogonal vector

Stereo Rectification

If $\boldsymbol{r}_1 = \boldsymbol{e}_1 = \frac{T}{||T||}$ and $\boldsymbol{r}_2$ $\boldsymbol{r}_3$ orthogonal

then $R_{\text{rect}} \boldsymbol{e}_1 = \begin{bmatrix} \boldsymbol{r}_1^\top \boldsymbol{e}_1 \\ \boldsymbol{r}_2^\top \boldsymbol{e}_1 \\ \boldsymbol{r}_3^\top \boldsymbol{e}_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$

Where is this point located on the image plane?

At x-infinity

Stereo Rectification

Stereo Rectification Algorithm

1. Estimate **E** using the 8 point algorithm (SVD)
2. Estimate the epipole **e** (SVD of **E**)
3. Build $\mathbf{R}_{rect}$ from **e**
4. Decompose **E** into **R** and **T**
5. Set $\mathbf{R}_1=\mathbf{R}_{rect}$ and $\mathbf{R}_2 = \mathbf{R}_{rect}\mathbf{R}^T$
6. Rotate each left camera point (warp image)
   $[x' \ y' \ z'] = \mathbf{R}_1 [x \ y \ z]$
7. Rectified points as $\mathbf{p} = f/z'[x' \ y' \ z']$
8. Repeat 6 and 7 for right camera points using $\mathbf{R}_2$

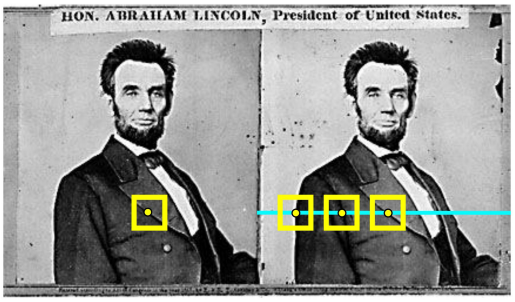
- OpenCV got you covered with `cv2.stereoRectify()` ...

Stereo Matching

Stereo Matching


Depth Estimation via Stereo Matching

Stereo Matching

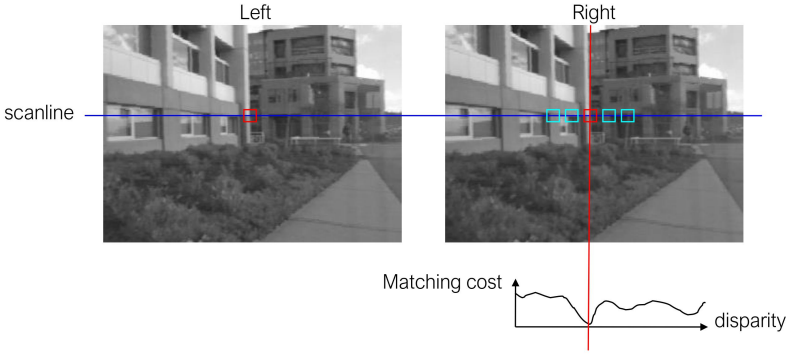


1. Rectify images
   (make epipolar lines horizontal)
2. For each pixel
   a. Find epipolar line
   b. Scan line for best match ← How would you do this?
   c. Compute depth from disparity

$$Z = \frac{bf}{d}$$

Stereo Matching

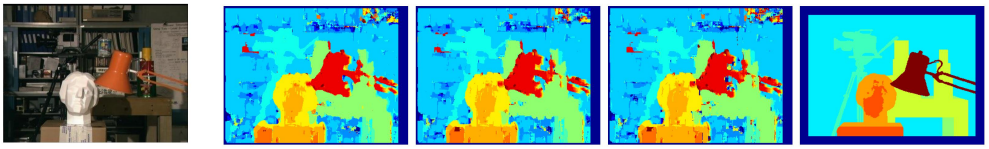
Stereo Block Matching



- Slide a window along the epipolar line and compare contents of that window with the reference window in the left image
- Matching cost: SSD or normalized correlation

Stereo Matching

Similarity Measure	Formula
Sum of Absolute Differences (SAD)	$\sum_{(i,j)\in W} \lvert I_1(i,j) - I_2(x+i,y+j)\rvert$
Sum of Squared Differences (SSD)	$\sum_{(i,j)\in W} \left(I_1(i,j) - I_2(x+i,y+j)\right)^2$
Zero-mean SAD	$\sum_{(i,j)\in W} \lvert I_1(i,j) - \bar{I}_1(i,j) - I_2(x+i,y+j) + \bar{I}_2(x+i,y+j)\rvert$
Locally scaled SAD	$\sum_{(i,j)\in W} \lvert I_1(i,j) - \frac{\bar{I}_1(i,j)}{\bar{I}_2(x+i,y+j)} I_2(x+i,y+j)\rvert$
Normalized Cross Correlation (NCC)	$\frac{\sum_{(i,j)\in W} I_1(i,j).I_2(x+i,y+j)}{\sqrt{\sum_{(i,j)\in W} I_1^2(i,j).\sum_{(i,j)\in W} I_2^2(x+i,y+j)}}$



SAD SSD NCC Ground truth

Stereo Matching

Effect of window size

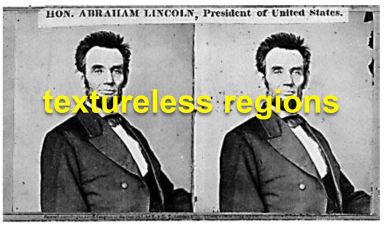
W = 3 W = 20

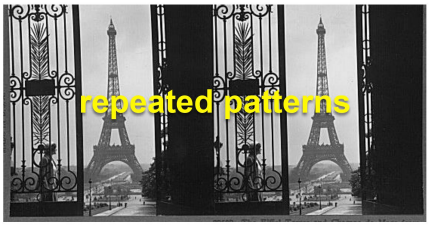
Smaller window
+ More detail
- More noise

Larger window
+ Smoother disparity maps
- Less detail
- Fails near boundaries

Stereo Matching

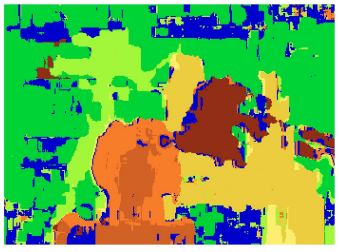
When will stereo block matching fail?





Depth Smoothing

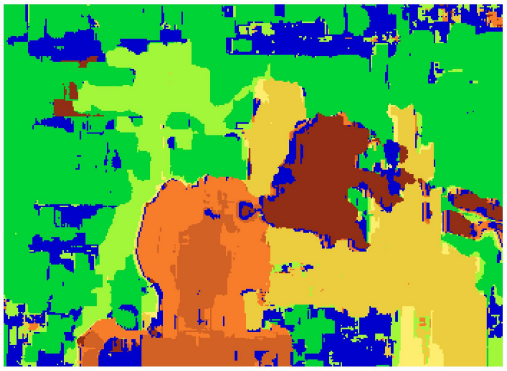
Block matching



Ground truth

What are some problems with the result?

Depth Smoothing



How can we improve depth estimation?

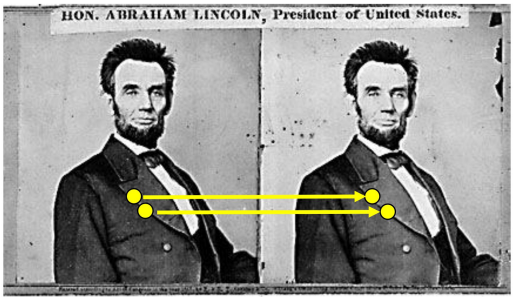
Too many discontinuities.
We expect disparity values to change slowly.

Let's make an assumption:
depth should change smoothly

Depth Smoothing

Stereo matching as …

Energy Minimization



What defines a good stereo correspondence?

1. **Match quality**
   - Want each pixel to find a good match in the other image
2. **Smoothness**
   - If two pixels are adjacent, they should (usually) move about the same amount

Depth Smoothing

energy function
(for one pixel)

$$E(d) = \underbrace{E_d(d)}_{\text{data term}} + \lambda \underbrace{E_s(d)}_{\text{smoothness term}}$$

Want each pixel to find a good match in the other image
(block matching result)

Adjacent pixels should (usually) move about the same amount
(smoothness function)

Depth Smoothing

$$E(d) = \boxed{E_d(d)} + \lambda E_s(d)$$

$$E_d(d) = \sum_{(x,y)\in I} C(x, y, d(x, y))$$

data term

SSD distance between windows centered at I(x, y) and J(x+ d(x,y), y)

Depth Smoothing

$$E(d) = E_d(d) + \lambda E_s(d)$$

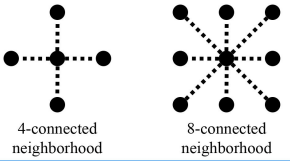
$$E_d(d) = \sum_{(x,y)\in I} C(x, y, d(x, y))$$

SSD distance between windows centered at I(x, y) and J(x+ d(x,y), y)

$$E_s(d) = \sum_{(p,q)\in\mathcal{E}} V(d_p, d_q)$$

smoothness term

$\mathcal{E}$: set of neighboring pixels



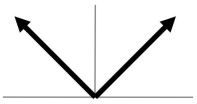
4-connected neighborhood | 8-connected neighborhood

Depth Smoothing

smoothness term $E_s(d) = \sum_{(p,q)\in\mathcal{E}} V(d_p, d_q)$

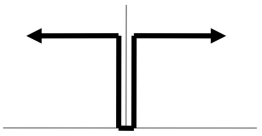
$$V(d_p, d_q) = aff(I_p, I_q)|d_p - d_q|$$

L_1 distance



$$V(d_p, d_q) = aff(I_p, I_q)\begin{cases} 0 \; if \; d_p = d_q \\ 1 \; if \; d_p \neq d_q \end{cases}$$

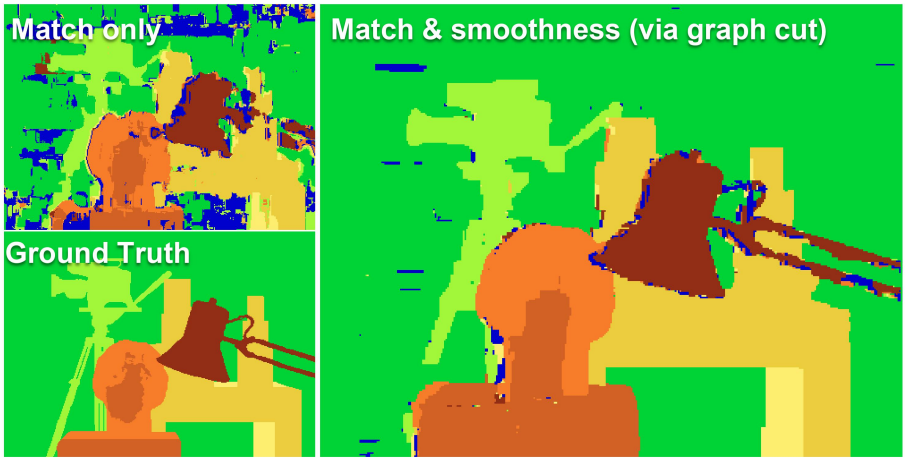
"Potts model"



$$aff(I_p, I_q) = e^{\frac{-|I_p - I_q|^2}{\sigma}}$$

Penalty for discontinuities is lower if edge in the image exists

Depth Smoothing



Y. Boykov, O. Veksler, and R. Zabih, Fast Approximate Energy Minimization via Graph Cuts, PAMI 2001

Code Example

- Code based on example by OpenCV

Point Cloud from Stereo Images

- Resulting `.ply` file can be easily viewed using MeshLab

In [1]:

```
import numpy as np
import cv2
import matplotlib.pyplot as plt

# function that writes a disparity to a point cloud
def write_ply(fn, verts, colors):
    ply_header = '''ply
    format ascii 1.0
    element vertex %(vert_num)d
    property float x
    property float y
    property float z
    property uchar red
    property uchar green
    property uchar blue
    end_header
    '''
    verts = verts.reshape(-1, 3)
    colors = colors.reshape(-1, 3)
    verts = np.hstack([verts, colors])
    with open(fn, 'wb') as f:
        f.write((ply_header % dict(vert_num=len(verts))).encode('utf-8'))
        np.savetxt(f, verts, fmt='%f %f %f %d %d %d ')
```

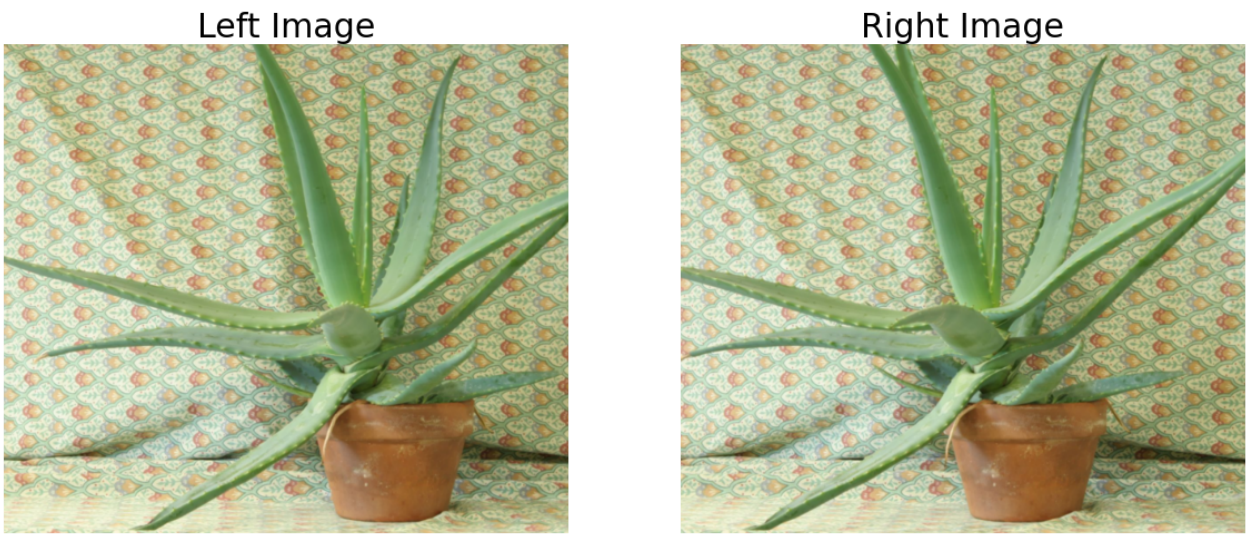
In [2]:

```
# load the input images and downscale for faster processing
imgL = cv2.pyrDown(cv2.imread('./assets/aloeL.jpg'))
imgR = cv2.pyrDown(cv2.imread('./assets/aloeR.jpg'))

# load also the GT depth for comparison
disp_gt = cv2.pyrDown(cv2.imread('./assets/aloeGT.png'))

# look at the inputs
plt.figure(figsize=(20,10))
```

```
plt.subplot(121)
plt.imshow(cv2.cvtColor(imgL,cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title('Left Image', fontsize=30)
plt.subplot(122)
plt.imshow(cv2.cvtColor(imgR,cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.title('Right Image', fontsize=30)
plt.show()
```



In [3]:

```
# disparity range is tuned for 'aloe' image pair
window_size = 3
min_disp = 16
num_disp = 112-min_disp
stereo = cv2.StereoSGBM_create(minDisparity = min_disp,
    numDisparities = num_disp,
    blockSize = 16,
    P1 = 8*3*window_size**2,
    P2 = 32*3*window_size**2,
    disp12MaxDiff = 1,
    uniquenessRatio = 10,
    speckleWindowSize = 100,
    speckleRange = 32
)

# compute the disparity using opencv func.
disp = stereo.compute(imgL, imgR).astype(np.float32) / 16.0
```

In [4]:

```
# compare the result to the ground truth
# look at the inputs
plt.figure(figsize=(20,10))
plt.subplot(121)
plt.imshow((disp-min_disp)/num_disp, cmap='gray')
plt.axis('off')
plt.title('Estimated Disparity', fontsize=30)
plt.subplot(122)
plt.imshow(cv2.cvtColor(disp_gt,cv2.COLOR_BGR2GRAY), cmap='gray')
plt.axis('off')
plt.title('GT Disparity', fontsize=30)
plt.show()
```

Estimated Disparity

GT Disparity

In [5]:

```
# generate a colored 3D point cloud by reprojecting to 3D for Meshlab
print('generating 3d point cloud...',)
h, w = imgL.shape[:2]
f = 0.8*w                          # guess for focal length
Q = np.float32([[1, 0, 0, -0.5*w],
                [0,-1, 0,  0.5*h], # turn points 180 deg around x-axis,
                [0, 0, 0,     -f], # so that y-axis looks up
                [0, 0, 1,      0]])
points = cv2.reprojectImageTo3D(disp, Q)
colors = cv2.cvtColor(imgL, cv2.COLOR_BGR2RGB)
mask = disp > disp.min()
out_points = points[mask]
out_colors = colors[mask]
out_fn = 'out.ply'
write_ply(out_fn, out_points, out_colors)
print('%s saved' % out_fn)
```

```
generating 3d point cloud...
out.ply saved
```

Recommended Videos

Warning!

- These videos do not replace the lectures and tutorials.
- Please use these to get a better understanding of the material, and not as an alternative to the written material.

Video By Subject

- Student Competition - Computer Vision Training
- CVFX Lecture 15 - Stereo Correspondence

Credits

- Slides (CMU) - Ioannis Gkioulekas, Kris Kitani, Srinivasa Narasimhan
- Multiple View Geometry in Computer Vision - Hartley and Zisserman - Chapter 6
- Computer vision: models, learning and inference , Simon J.D. Prince - Chapter 15
- Computer Vision: Algorithms and Applications - Richard Szeliski - Sections 2.1.5, 6.2. , 7.1, 7.2, 11.1

- Icons from Icon8.com - https://icons8.com