Computer Vision

CSE 455
Matching and Blending

Linda Shapiro
Professor of Computer Science & Engineering
Professor of Electrical & Computer Engineering

Review

- Descriptors
- Matching
- Computing Transformation

Simple Normalized Descriptor

interest point	neighborhood around interest point	normalized neighborhood around interest point
201	45 56 200 46 201 200 85 101 105	156 145 1 155 0 1 116 100 96

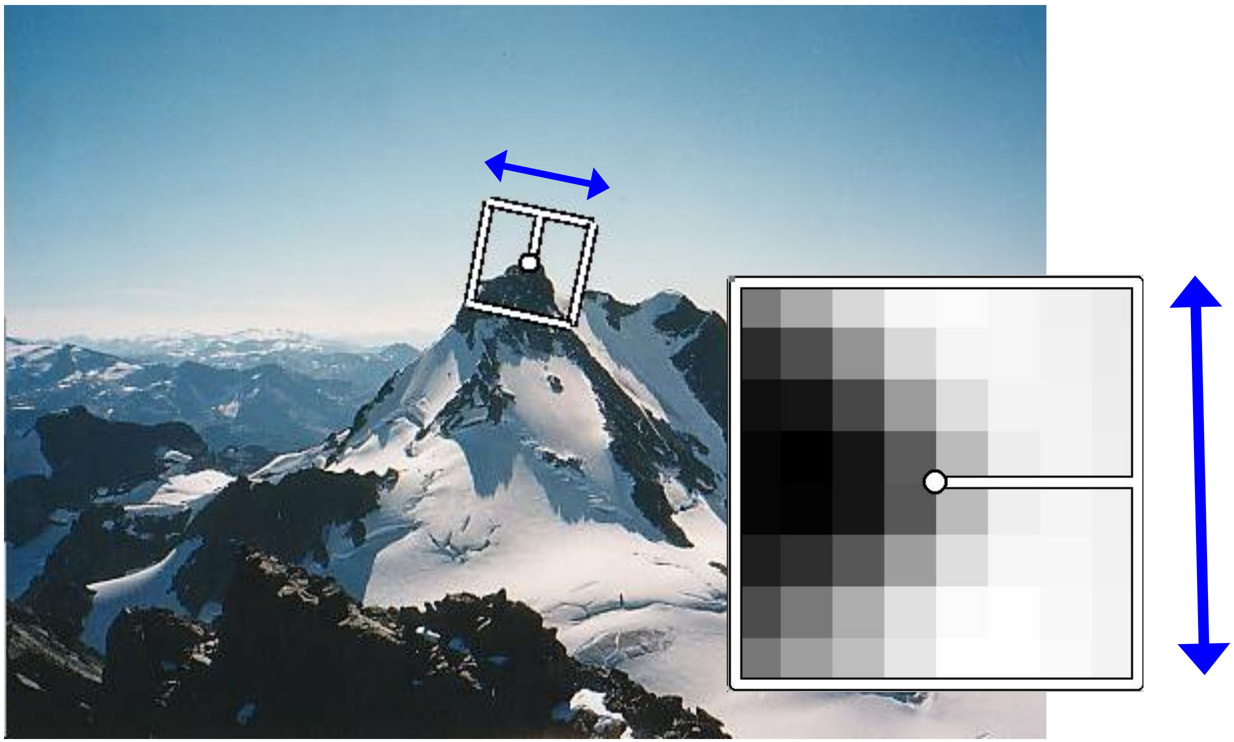
- The simple descriptor just subtracts the center value from each of the neighbors, including itself to normalize for lighting and exposure.
- We can store this as a 1D vector to be efficient:
  156 145 1 155 0 1 116 100 96

Properties of our Descriptor

- Translation Invariant
- Not scale invariant
- Not rotation invariant
- Somewhat invariant to lighting changes

- Let's look at the SIFT descriptor, because it is heavily used, even without using the SIFT key point detector.
- It already solves the scale problem by computing at multiple scales and keeping track.

Rotation invariance



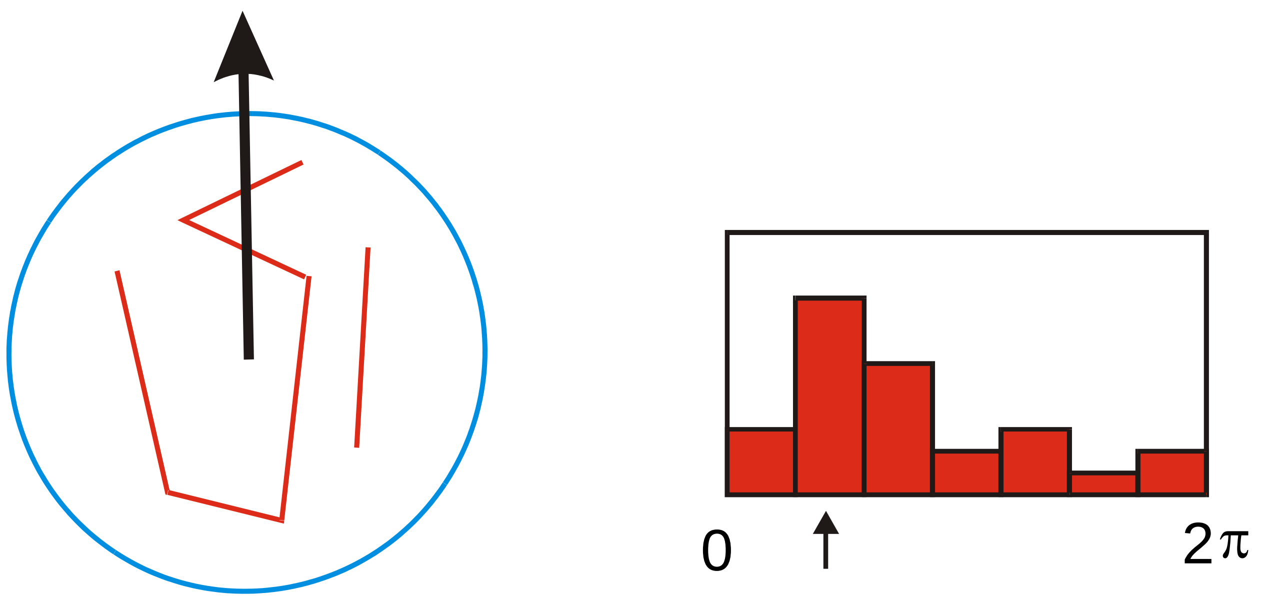
- Rotate patch according to its dominant gradient orientation
- This puts the patches into a canonical orientation.

Image from Matthew Brown

Orientation Normalization

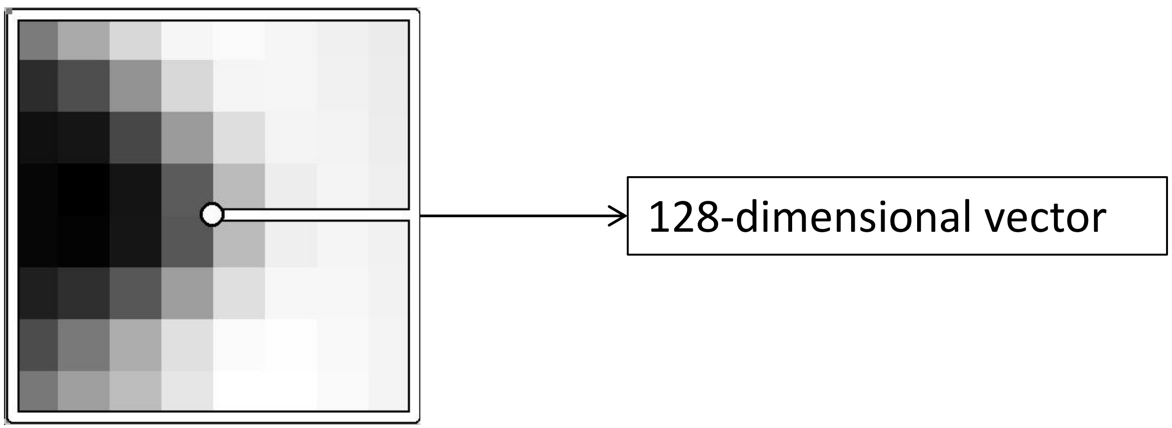
- Compute orientation histogram
- Select dominant orientation
- Normalize: rotate to fixed orientation

[Lowe, SIFT, 1999]



Once we have found the key points and a dominant orientation for each,

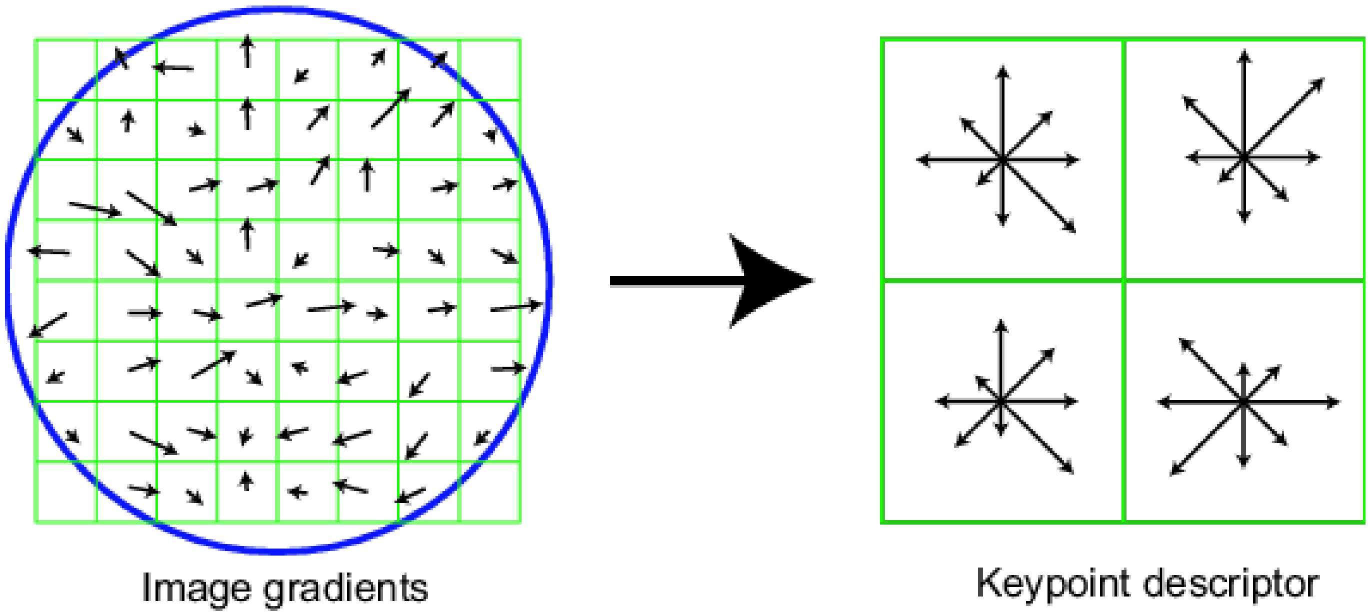
we need to describe the (rotated and scaled) neighborhood about each.



SIFT descriptor

Full version

- Divide the 16x16 window into a 4x4 grid of cells (2x2 case shown below)
- Compute an orientation histogram for each cell
- 16 cells * 8 orientations = 128 dimensional descriptor

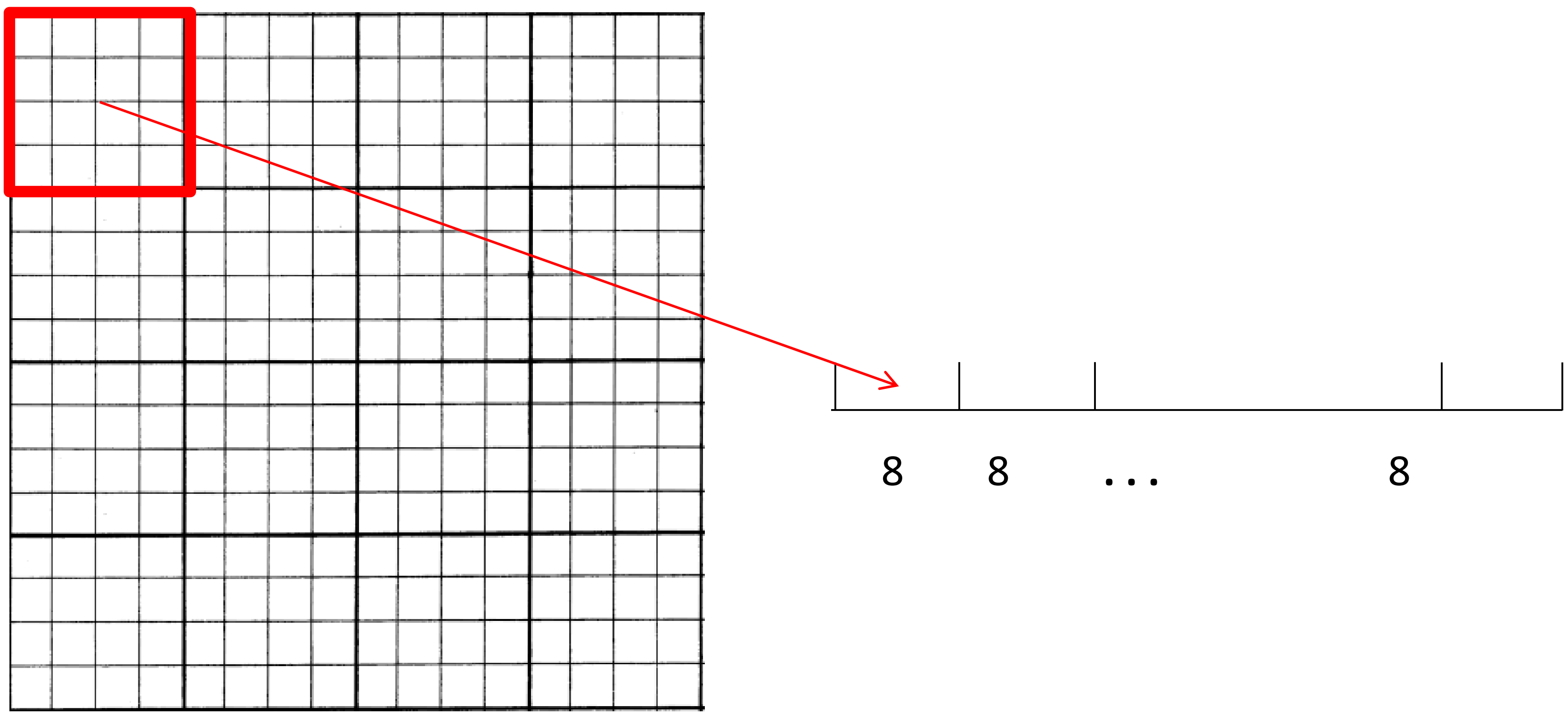


Adapted from slide by David Lowe

SIFT descriptor

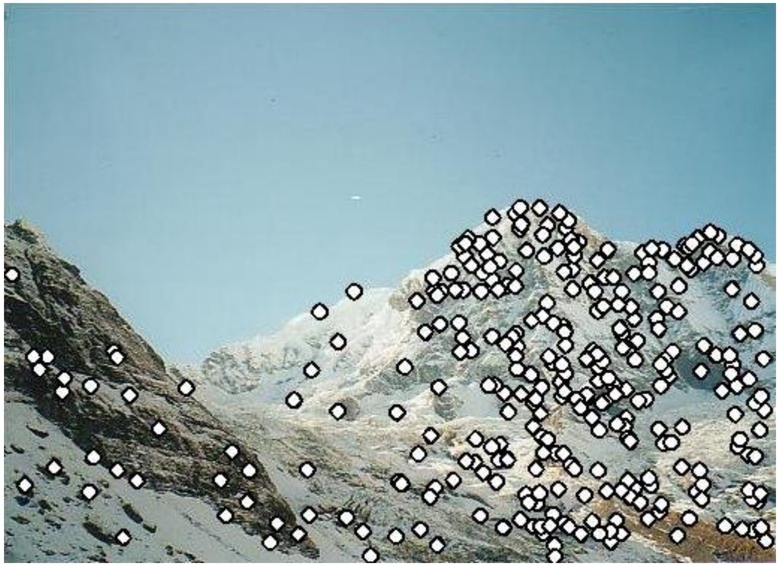
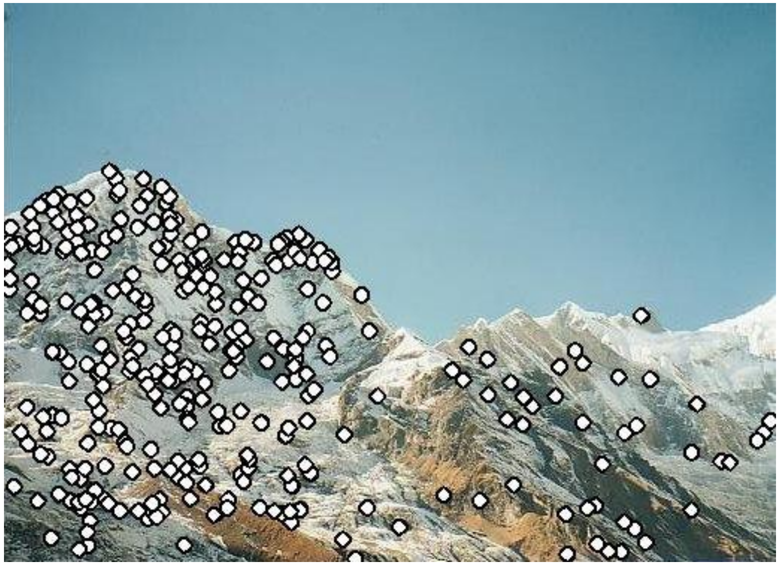
Full version

- Divide the 16x16 window into a 4x4 grid of cells
- Compute an orientation histogram for each cell
- 16 cells * 8 orientations = 128 dimensional descriptor



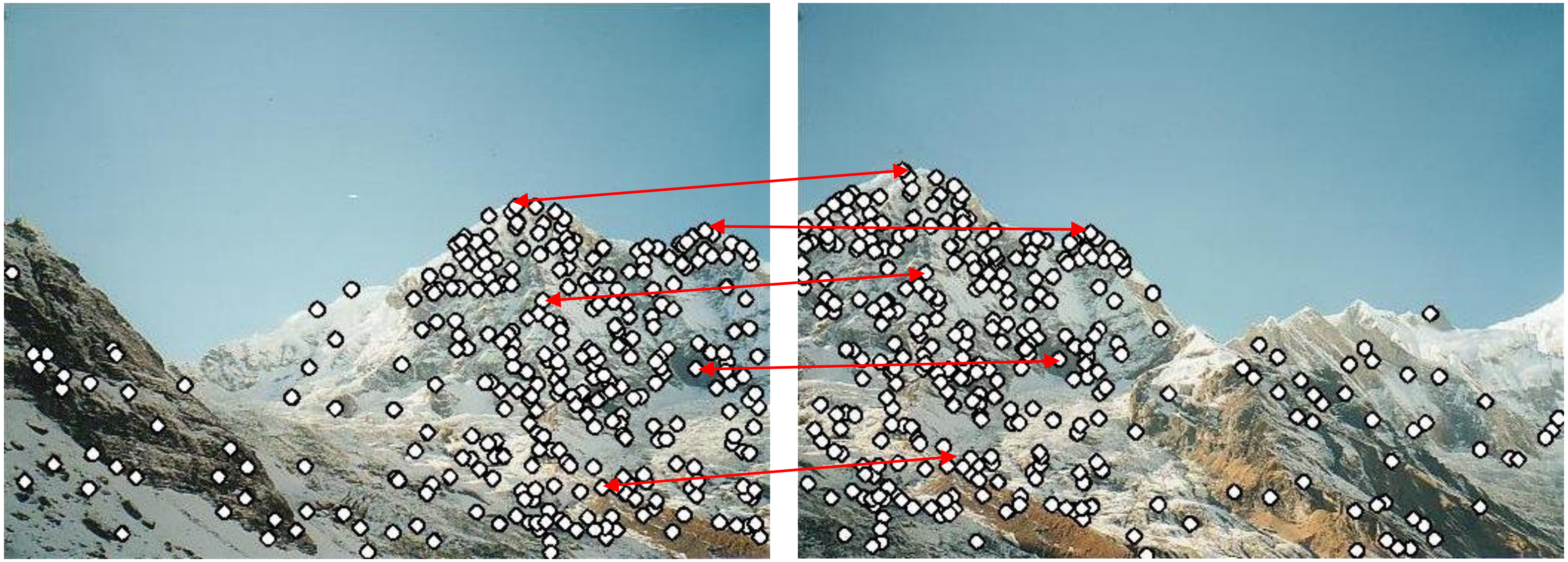
Matching with Features

•Detect feature points in both images

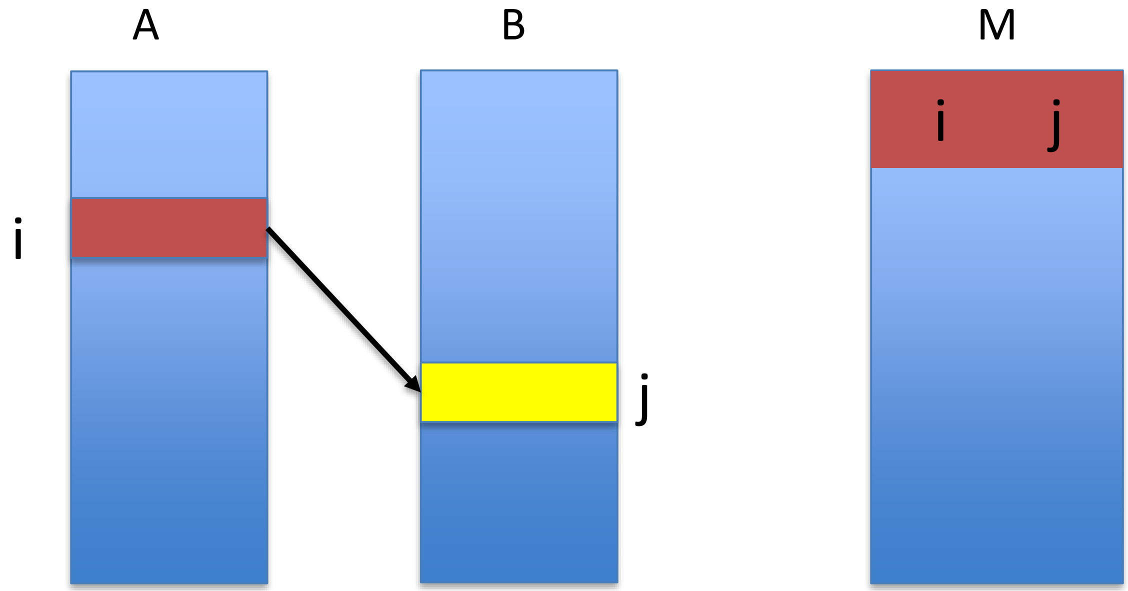
Matching with Features

- Detect feature points in both images
- Find corresponding pairs



Find the best matches

- For each descriptor a in A, find its best match b in B

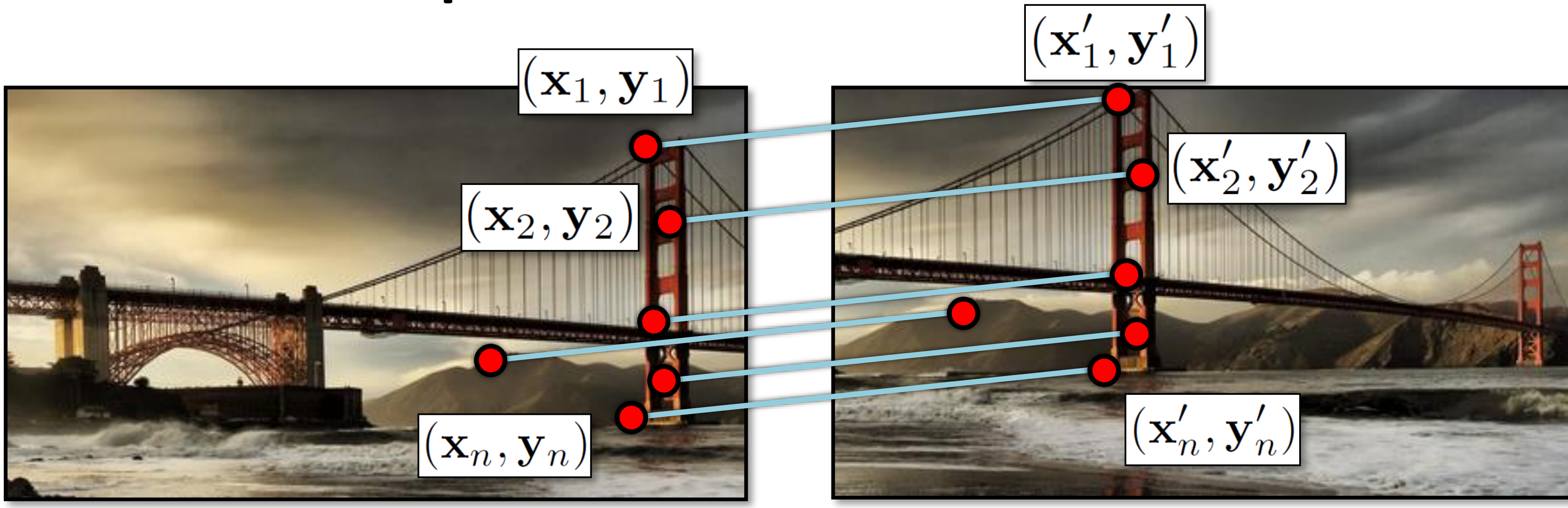


- And store it in a vector of matches
- Note: this is abstract; see code for details.

- Larger Goal: Combine two or more overlapping images to make one larger image

Slide credit: Vaibhav Vaish

Simple case: translations



Displacement of match *i* = $(\mathbf{x}'_i - \mathbf{x}_i, \mathbf{y}'_i - \mathbf{y}_i)$

$$(\mathbf{x}_t, \mathbf{y}_t) = \left(\frac{1}{n} \sum_{i=1}^{n} \mathbf{x}'_i - \mathbf{x}_i, \frac{1}{n} \sum_{i=1}^{n} \mathbf{y}'_i - \mathbf{y}_i \right)$$

Solving for homographies

$$\begin{bmatrix} x_i' \\ y_i' \\ 1 \end{bmatrix} \cong \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

Why is this now a variable and not just 1?

- A homography is a projective object, in that it has no scale. It is represented by the above matrix, up to scale.
- One way of fixing the scale is to set one of the coordinates to 1, though that choice is arbitrary.
- But that's what most people do and your assignment code does.

Solving for homographies

$$\begin{bmatrix} x'_i \\ y'_i \\ 1 \end{bmatrix} \cong \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

$$x'_i = \frac{h_{00}x_i + h_{01}y_i + h_{02}}{h_{20}x_i + h_{21}y_i + h_{22}}$$

$$y'_i = \frac{h_{10}x_i + h_{11}y_i + h_{12}}{h_{20}x_i + h_{21}y_i + h_{22}}$$

Why the division?

$$\begin{aligned} x'_i(h_{20}x_i + h_{21}y_i + h_{22}) &= h_{00}x_i + h_{01}y_i + h_{02} \\ y'_i(h_{20}x_i + h_{21}y_i + h_{22}) &= h_{10}x_i + h_{11}y_i + h_{12} \end{aligned}$$

Solving for homographies

$$x'_i(h_{20}x_i + h_{21}y_i + h_{22}) = h_{00}x_i + h_{01}y_i + h_{02}$$
$$y'_i(h_{20}x_i + h_{21}y_i + h_{22}) = h_{10}x_i + h_{11}y_i + h_{12}$$

$$\begin{bmatrix} x_i & y_i & 1 & 0 & 0 & 0 & -x'_i x_i & -x'_i y_i & -x'_i \\ 0 & 0 & 0 & x_i & y_i & 1 & -y'_i x_i & -y'_i y_i & -y'_i \end{bmatrix} \begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

This is just for one pair of points.

Direct Linear Transforms (n points)

$$\underbrace{\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x'_1 x_1 & -x'_1 y_1 & -x'_1 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -y'_1 x_1 & -y'_1 y_1 & -y'_1 \\ & & & & & \vdots & & & \\ x_n & y_n & 1 & 0 & 0 & 0 & -x'_n x_n & -x'_n y_n & -x'_n \\ 0 & 0 & 0 & x_n & y_n & 1 & -y'_n x_n & -y'_n y_n & -y'_n \end{bmatrix}}_{\substack{\mathbf{A} \\ 2n \times 9}} \underbrace{\begin{bmatrix} h_{00} \\ h_{01} \\ h_{02} \\ h_{10} \\ h_{11} \\ h_{12} \\ h_{20} \\ h_{21} \\ h_{22} \end{bmatrix}}_{\substack{\mathbf{h} \\ 9}} = \underbrace{\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}}_{\substack{\mathbf{0} \\ 2n}}$$

Defines a least squares problem: minimize $\|\mathbf{Ah} - \mathbf{0}\|^2$

- Since $\mathbf{h}$ is only defined up to scale, solve for unit vector $\hat{\mathbf{h}}$
- Solution: $\hat{\mathbf{h}}$ = eigenvector of $\mathbf{A}^T\mathbf{A}$ with smallest eigenvalue
- Works with 4 or more points

Direct Linear Transforms

- Why could we not solve for the homography in exactly the same way we did for the affine transform, ie.

$$\mathbf{t} = \left(\mathbf{A}^{\mathrm{T}}\mathbf{A}\right)^{-1}\mathbf{A}^{\mathrm{T}}\mathbf{b}$$

Answer from Sameer Agarwal (Dr. Rome in a Day)

- For an affine transform, we have equations of the form $Ax_i + b = y_i$, solvable by linear regression.
- For the homography, the equation is of the form

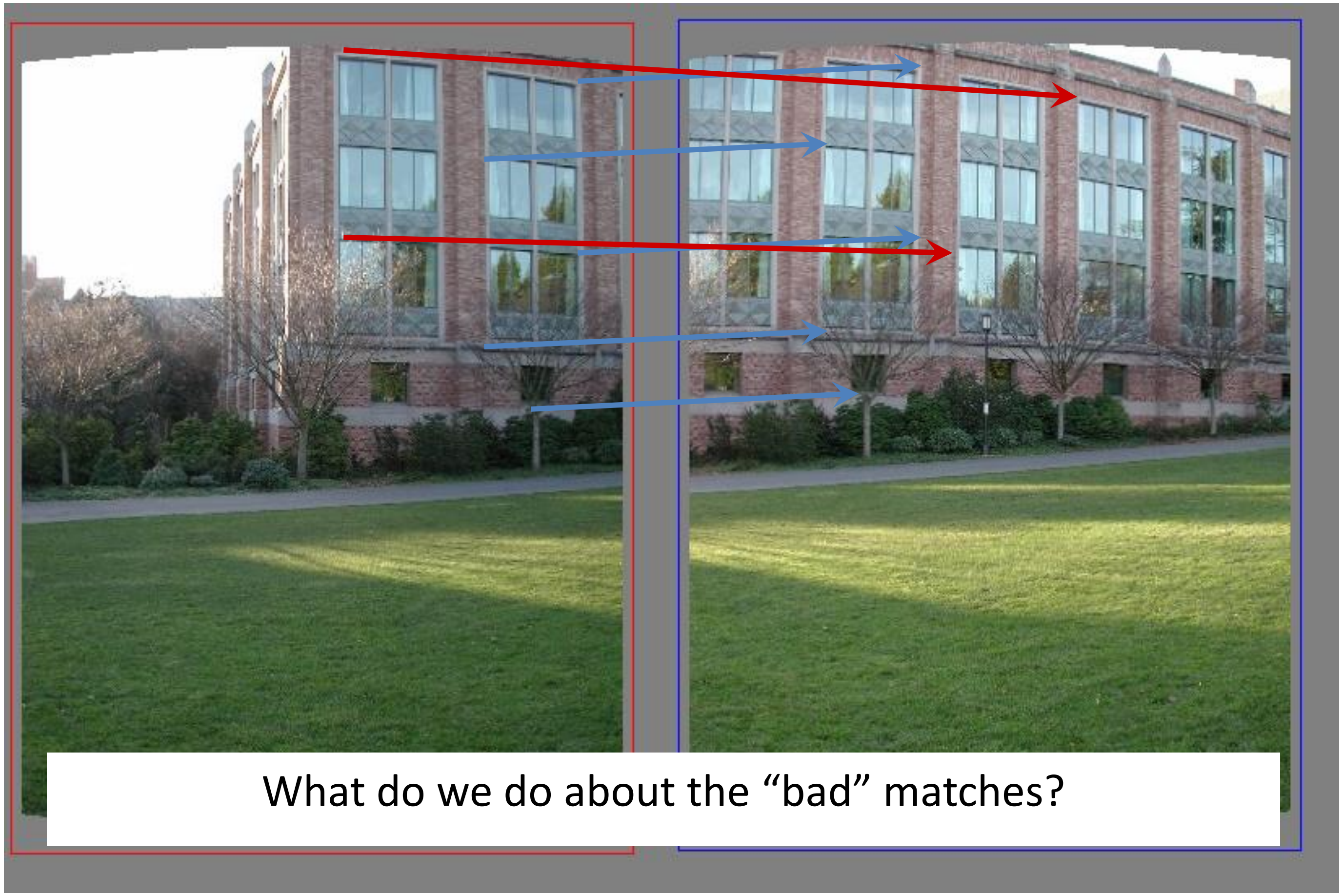
$H\tilde{x}_i \sim \tilde{y}_i$ (homogeneous coordinates)

and the ~ means it holds only up to scale. The affine solution does not hold.

Colosseum: 2,097 images, 819,242 points

Trevi Fountain: 1,935 images, 1,055,153 points

Matching features



RANSAC for estimating homography

- RANSAC loop:

1. Select four feature pairs (at random)
2. Compute homography $\boldsymbol{H}$ (exact)
3. Compute inliers where $\|p_i{'}, \boldsymbol{H}\, p_i\| < \varepsilon$

- Keep largest set of inliers
- Re-compute least-squares $\boldsymbol{H}$ estimate using all of the inliers

Panorama algorithm:

Find corners in both images

Calculate descriptors

Match descriptors

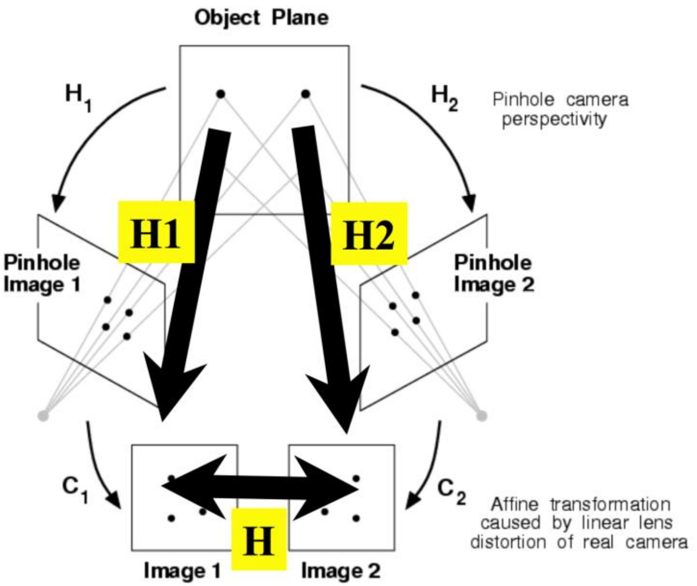
RANSAC to find homography

Stitch together images with homography

Stitching panoramas:

- We know homography is right choice under certain assumption:
  - Assume we are taking multiple images of planar object

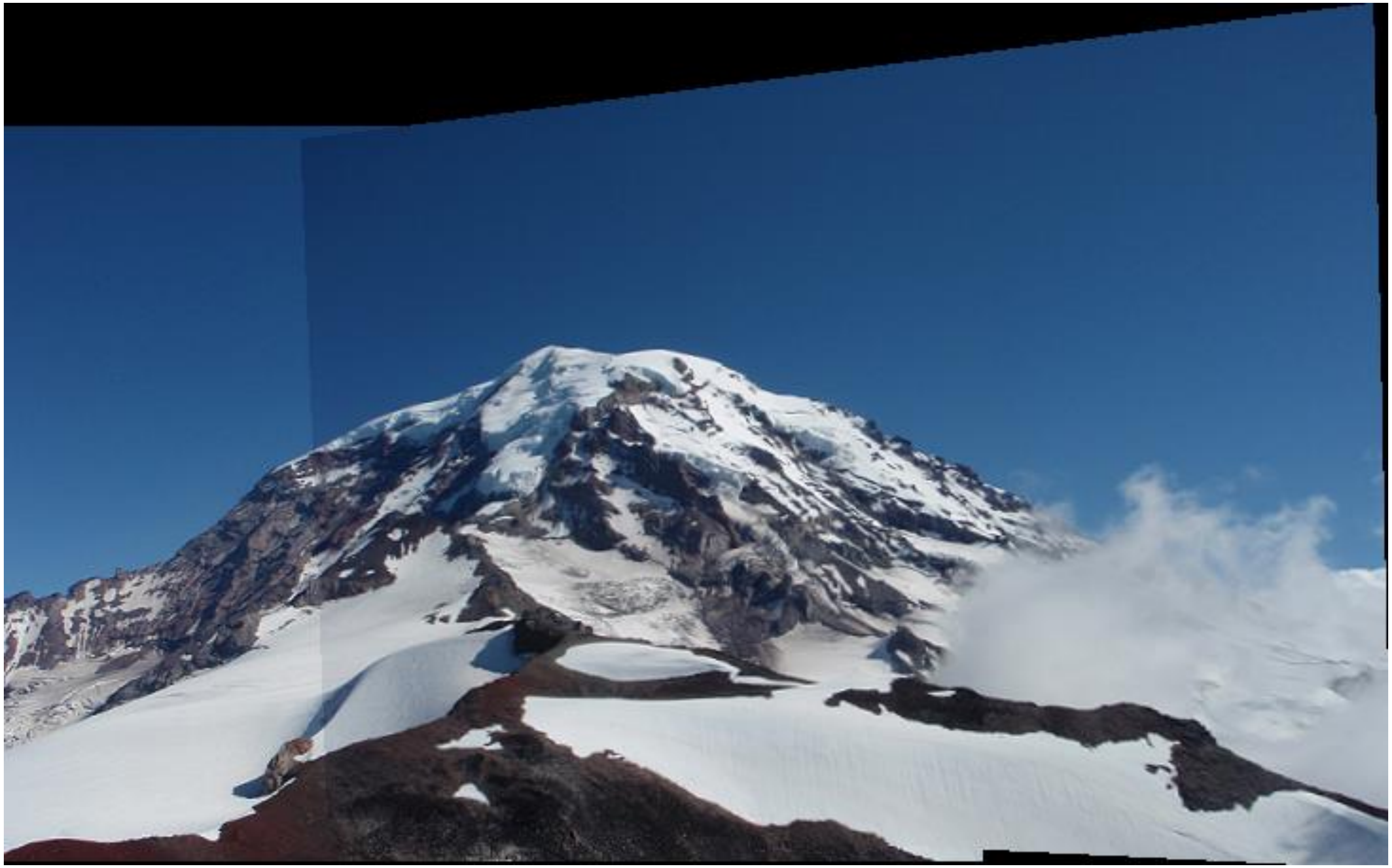
3D



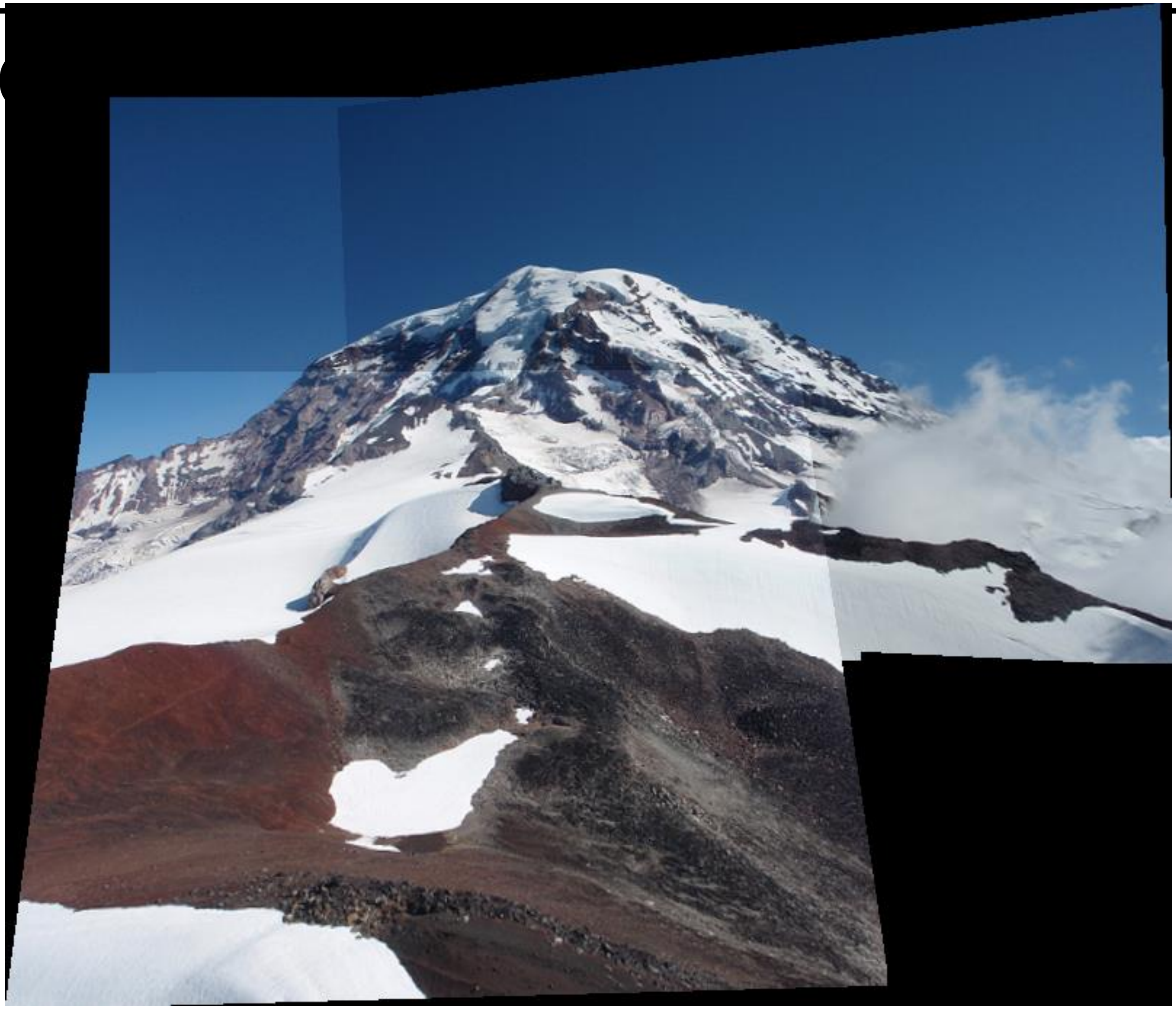
homography H

In practice:

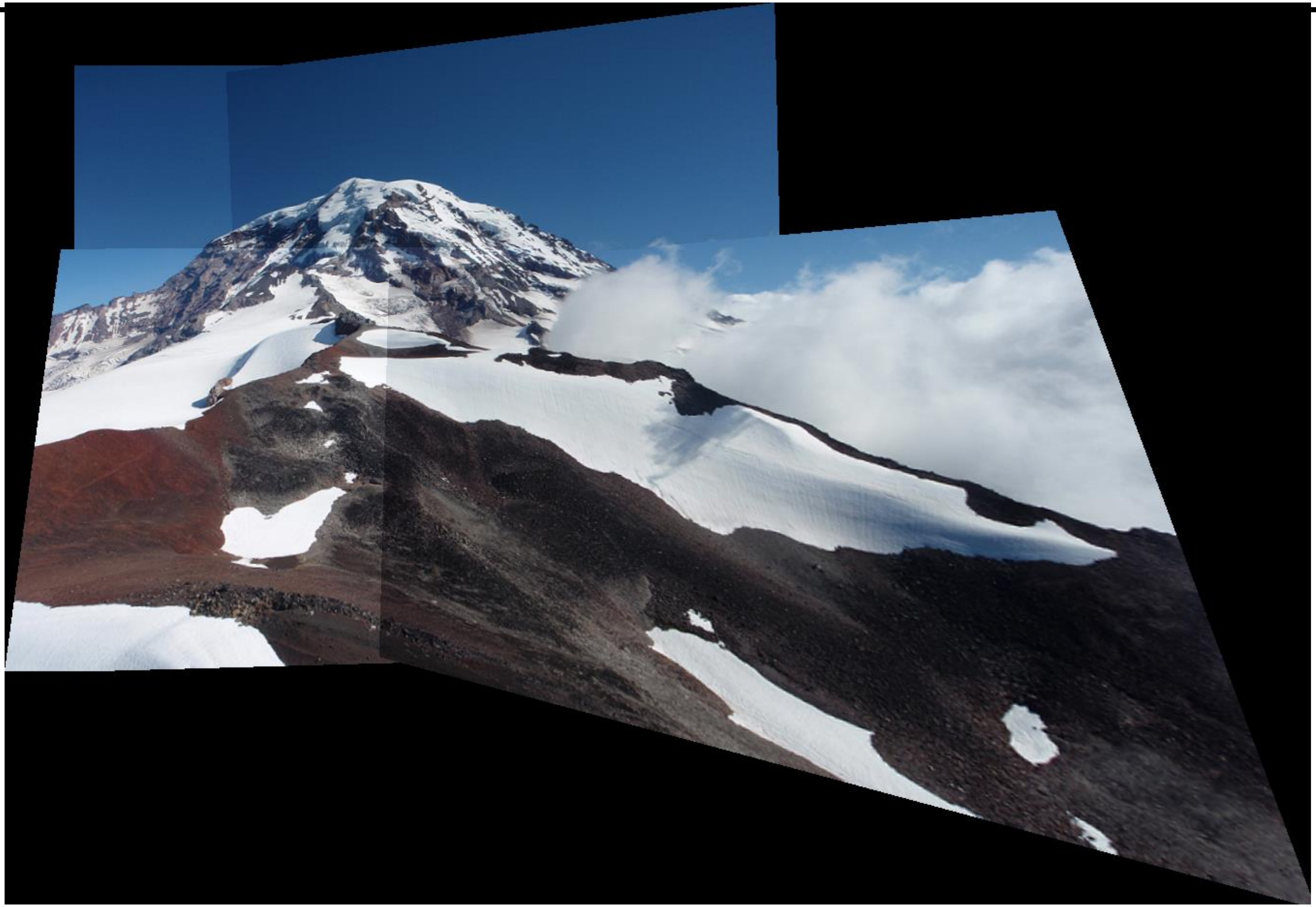
In practice:

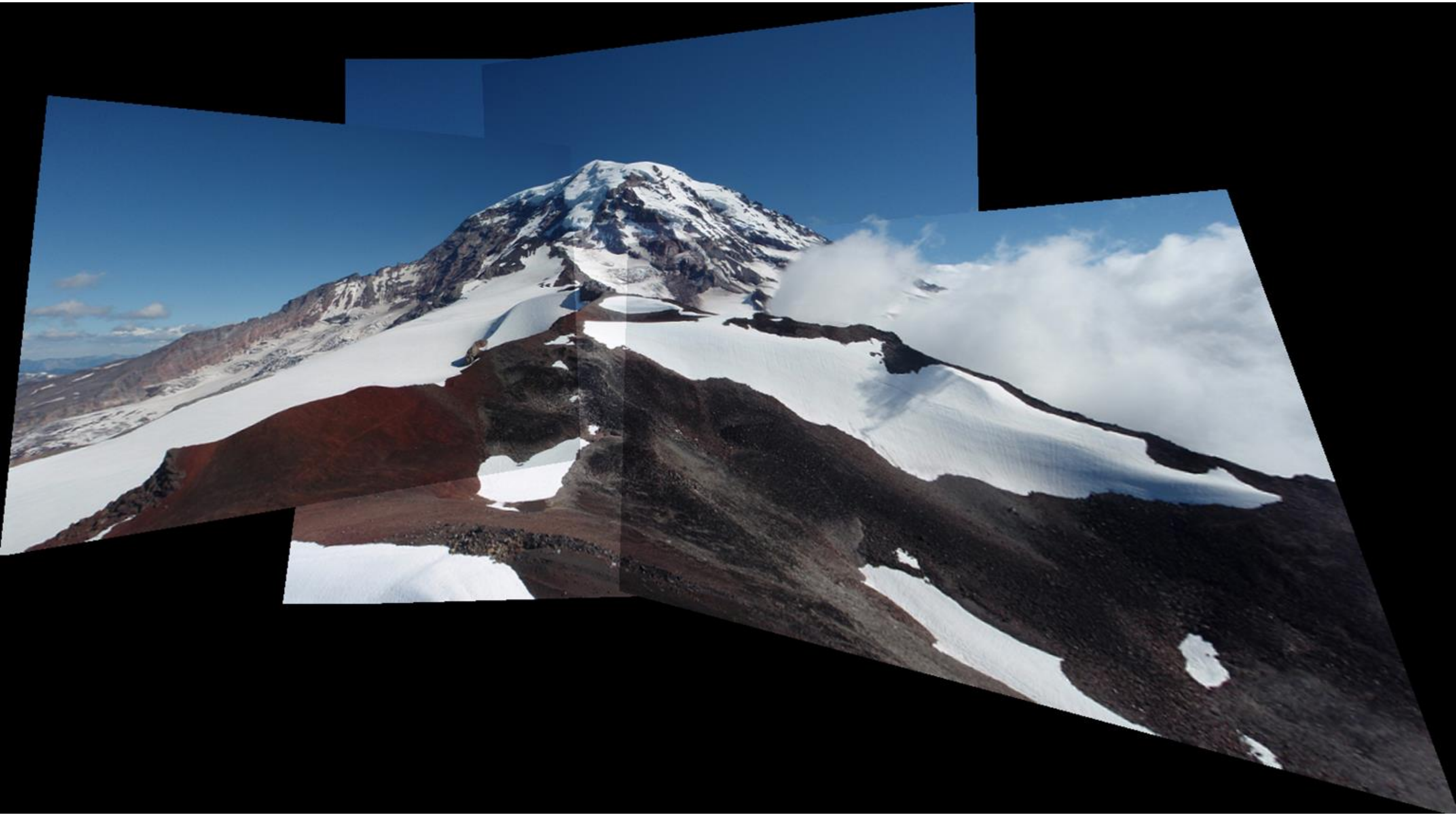


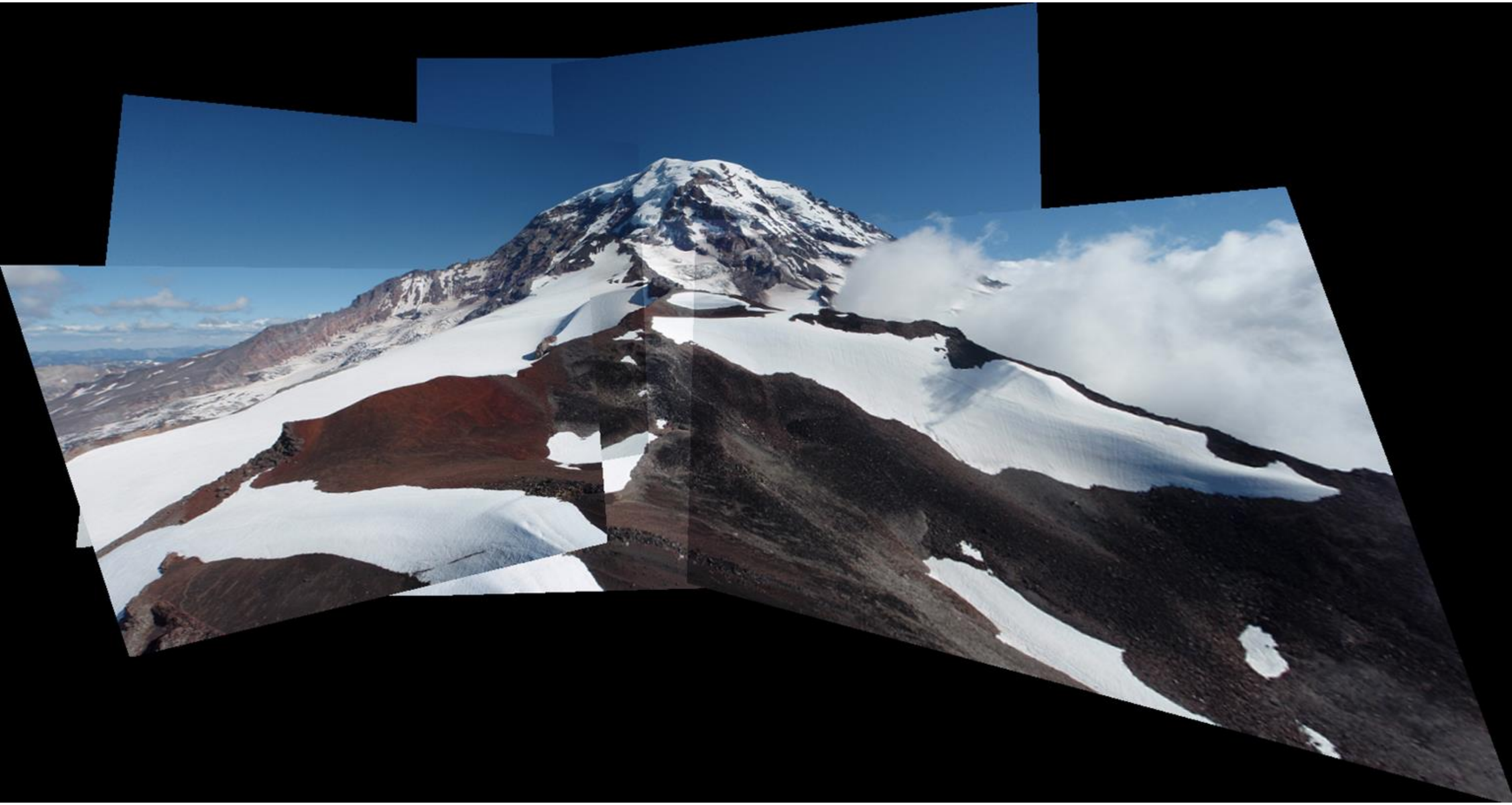
In prac



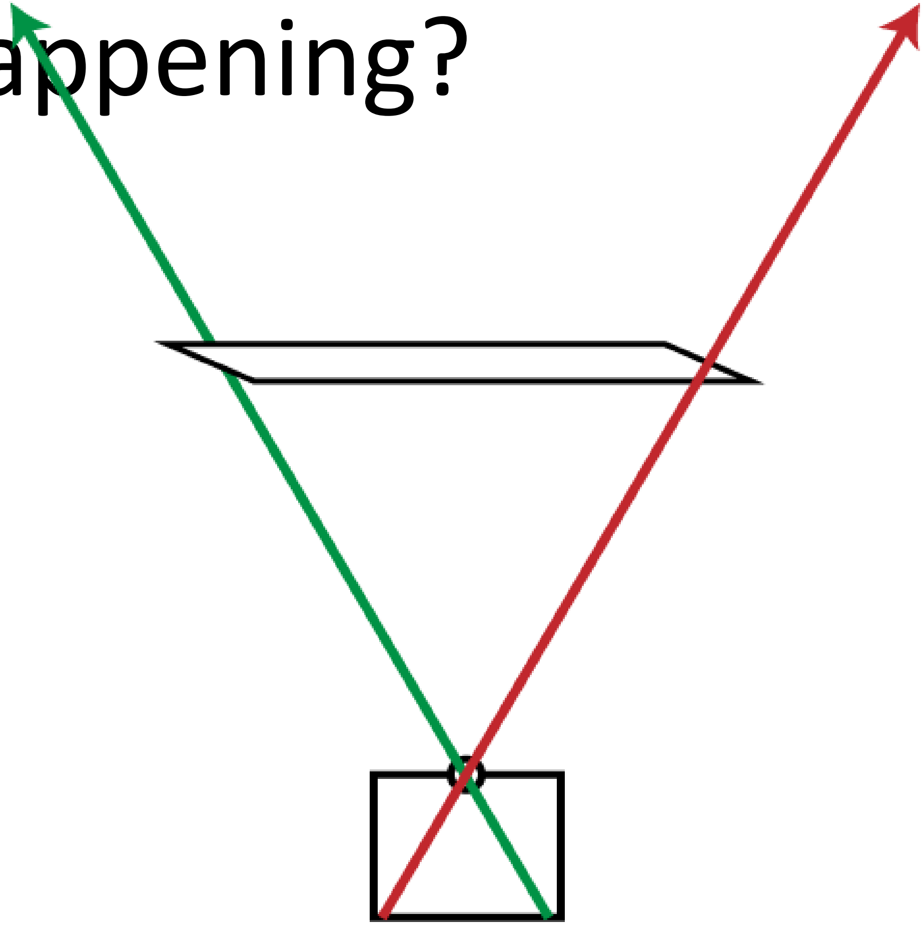
In



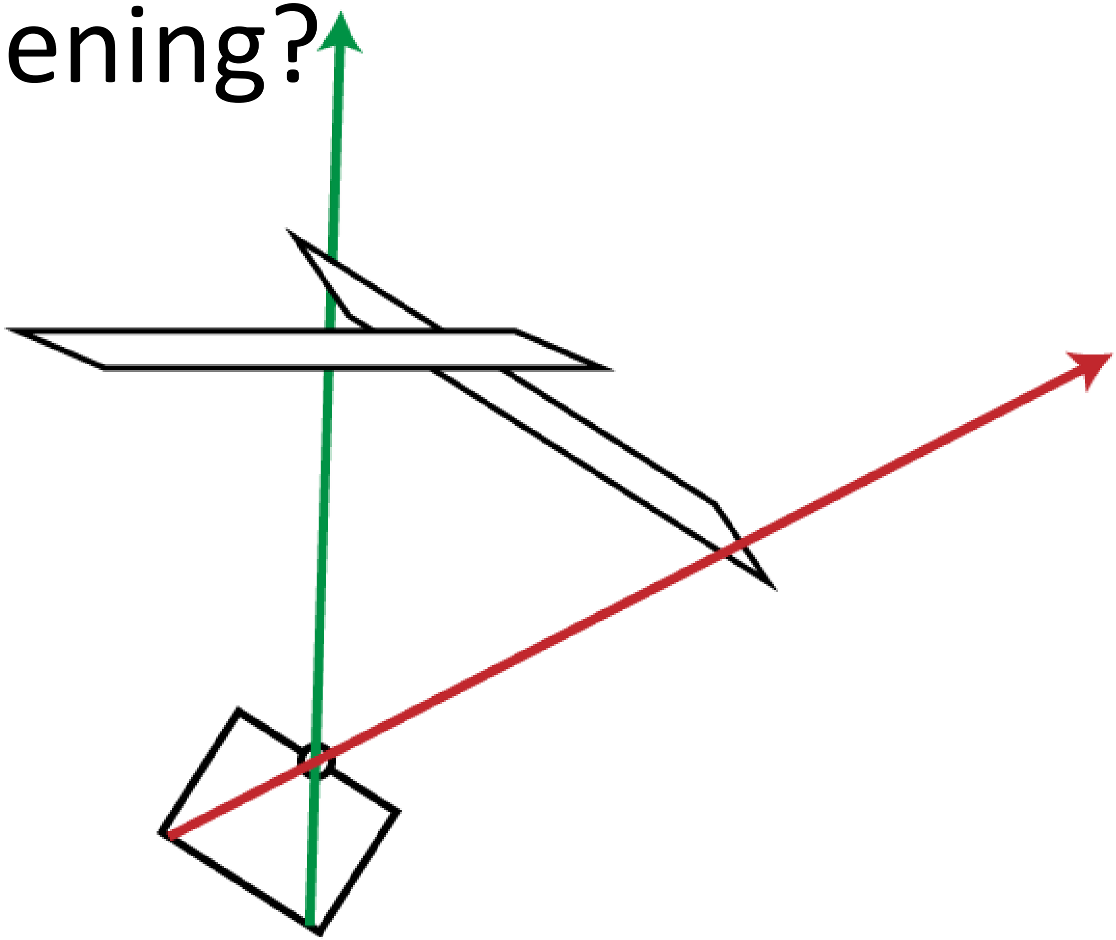




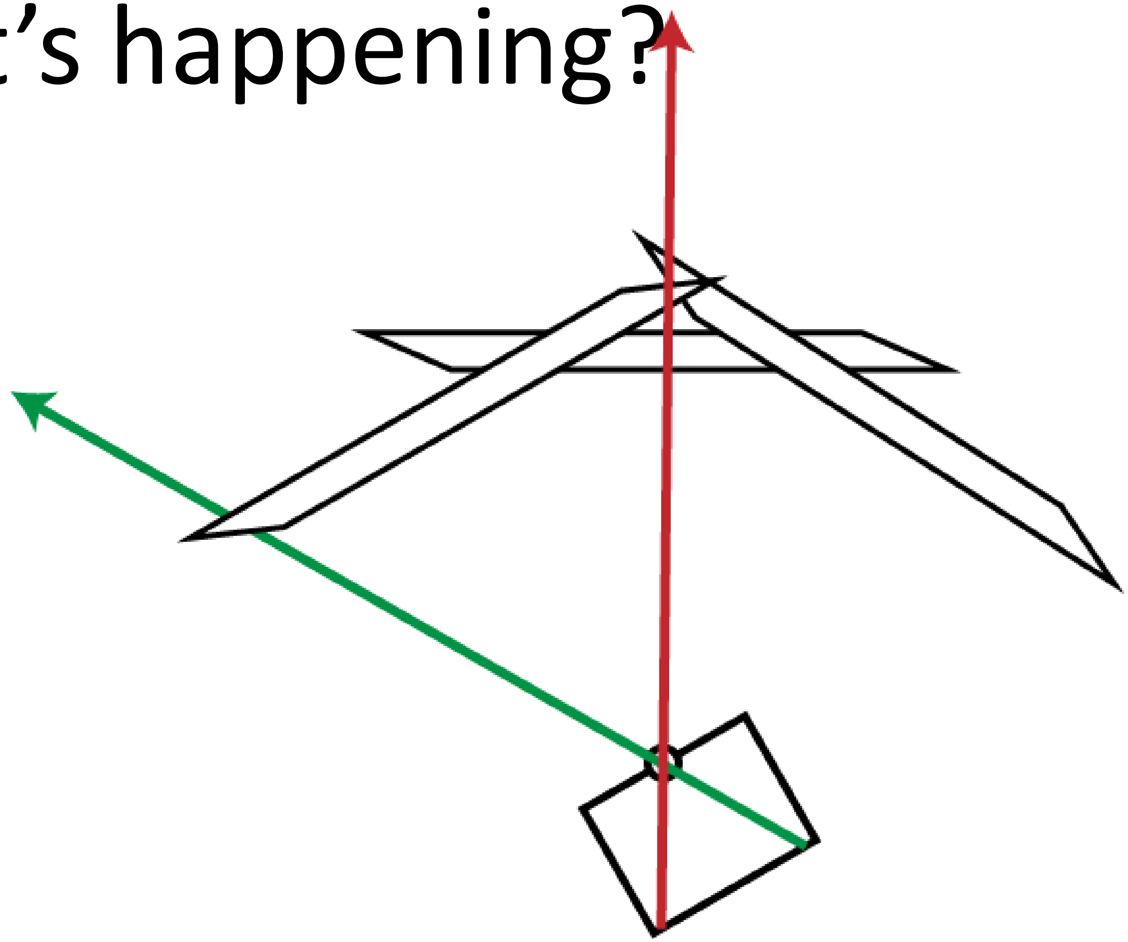
What’s happening?



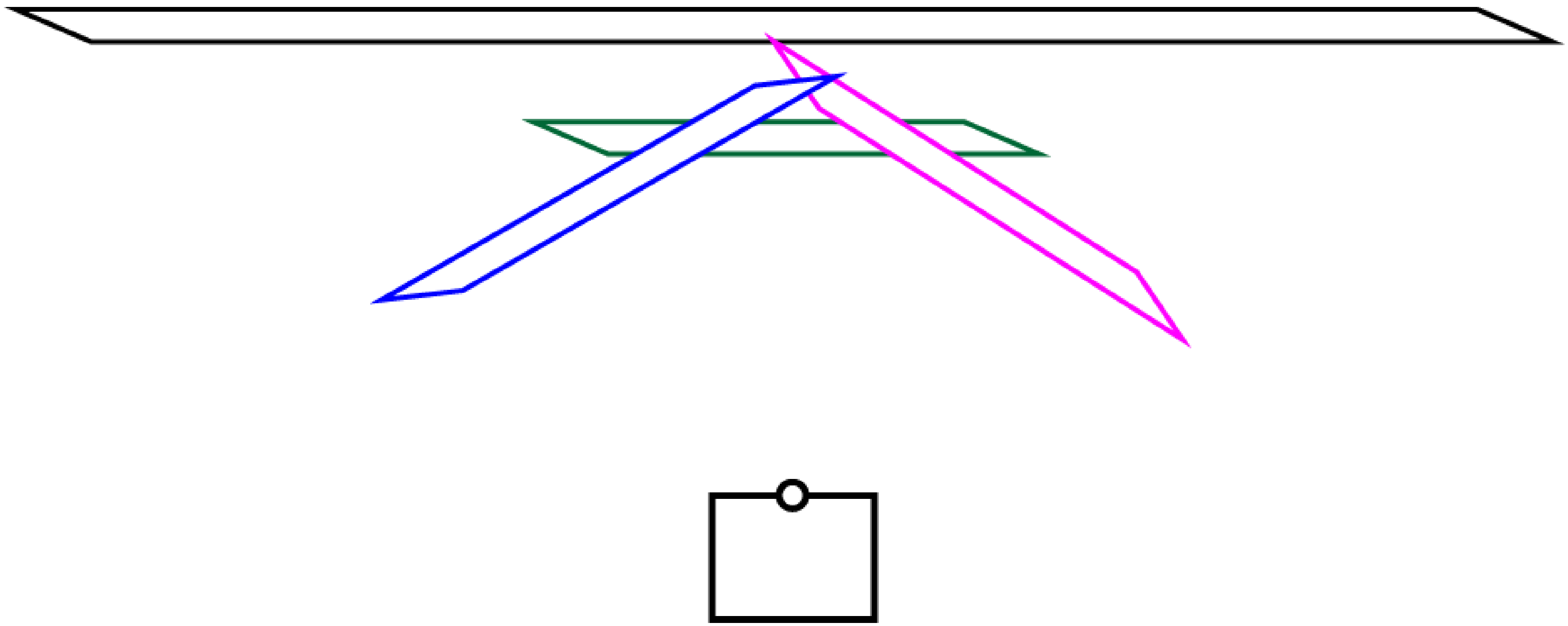
What’s happening?



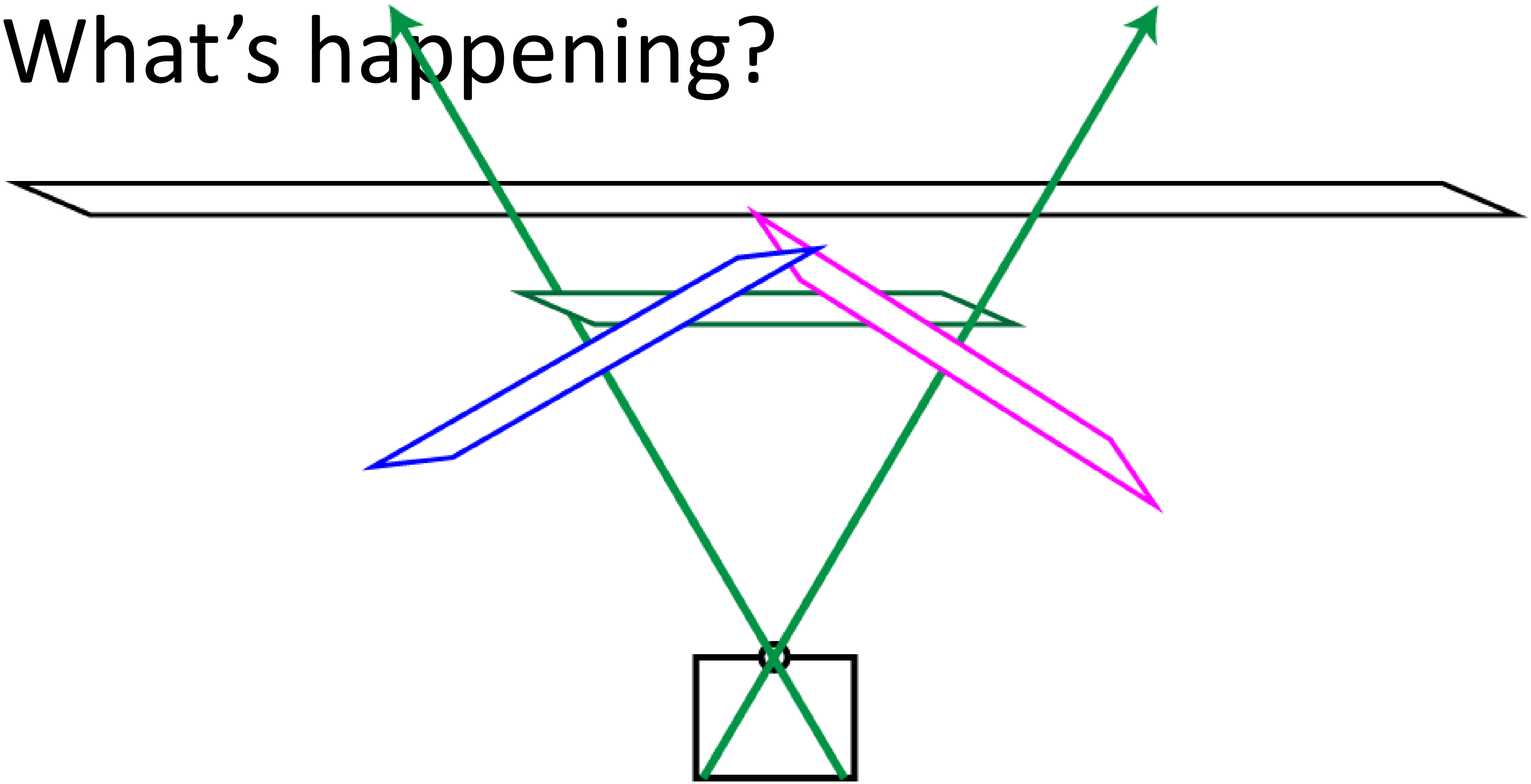
What’s happening?



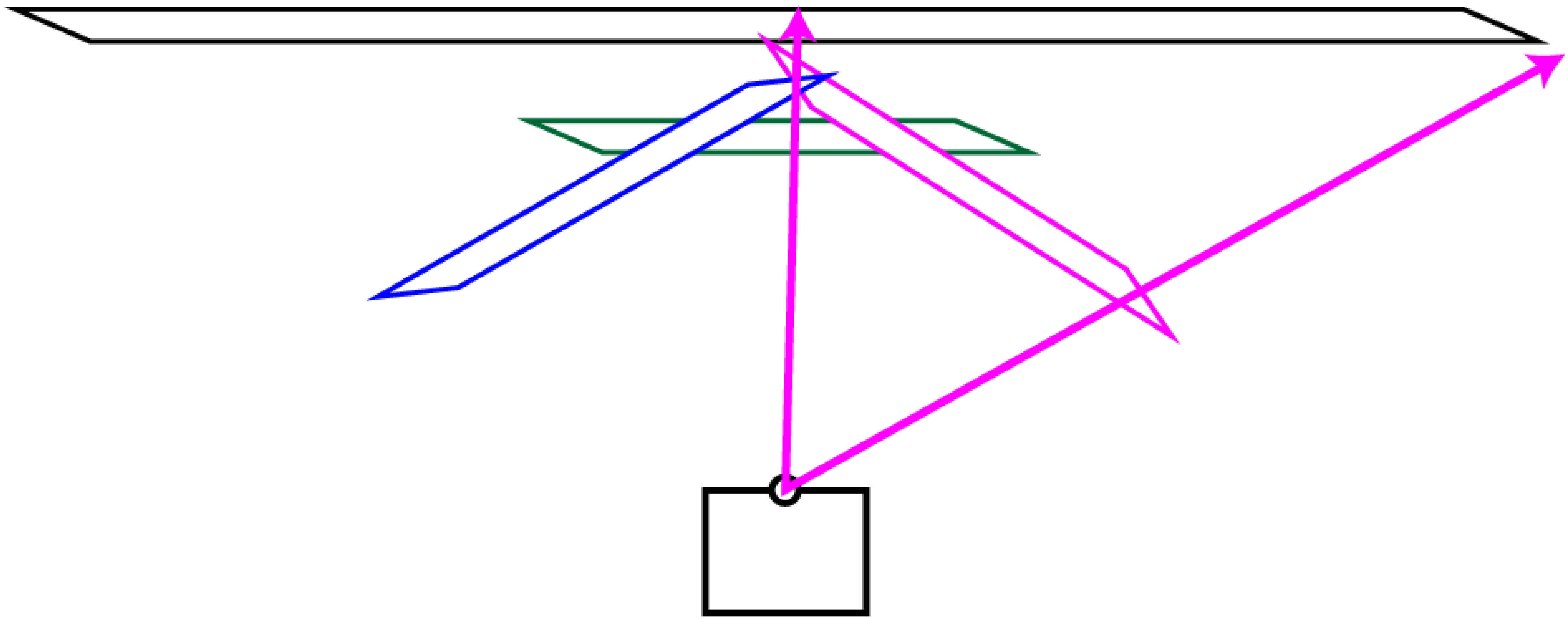
What’s happening?



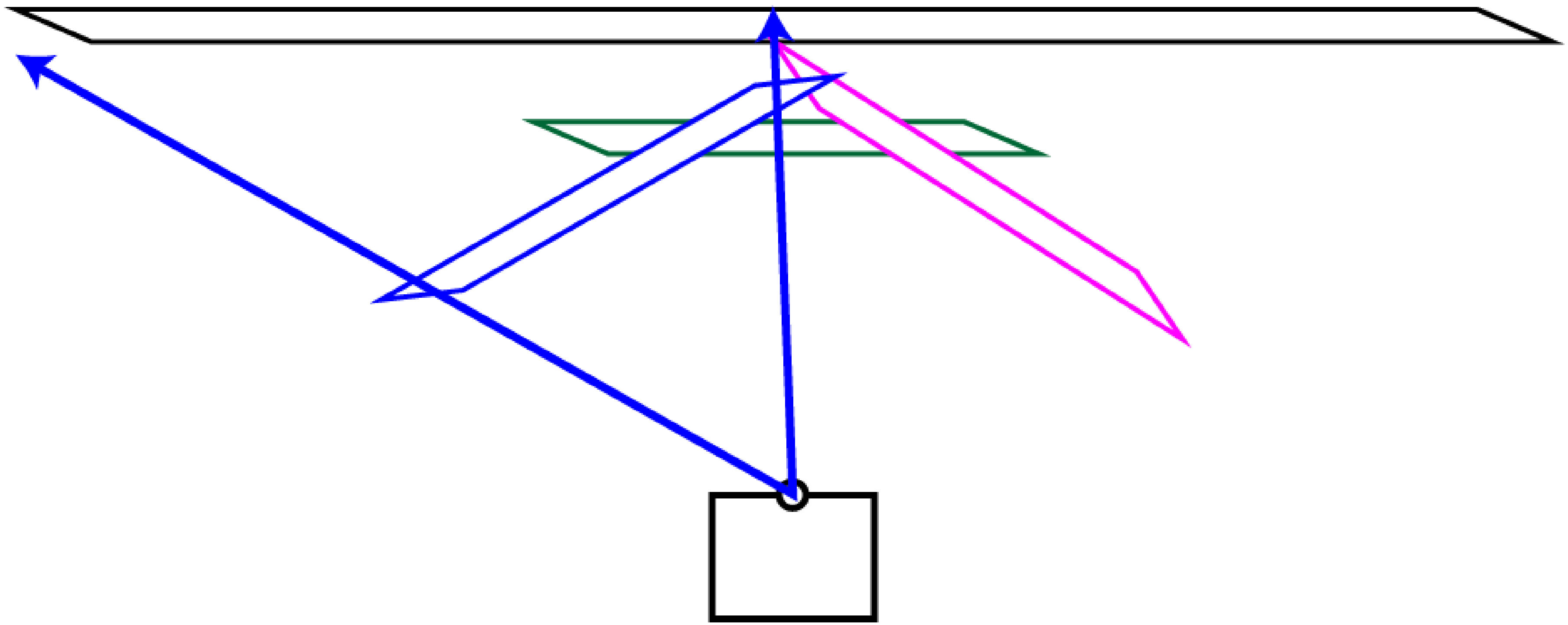
What's happening?



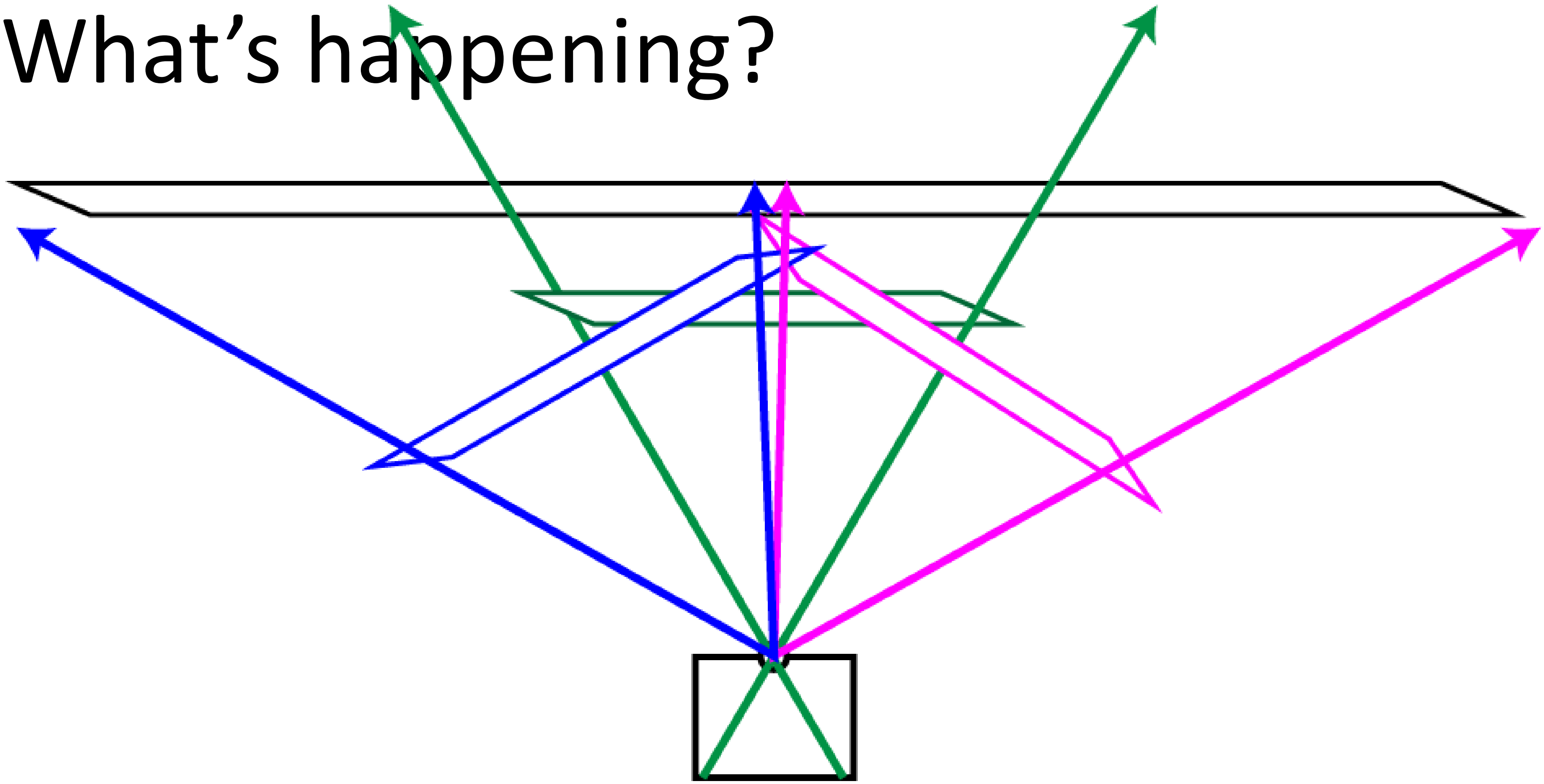
What’s happening?



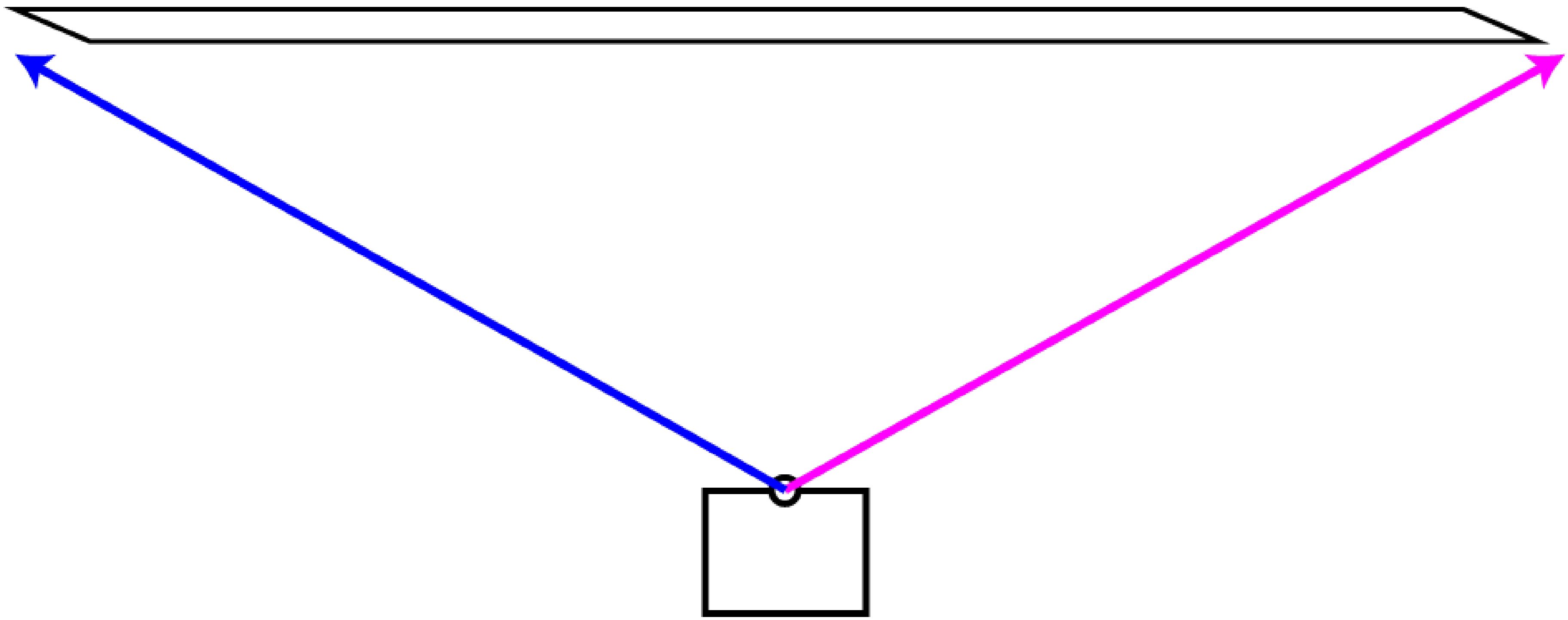
What’s happening?



What’s happening?

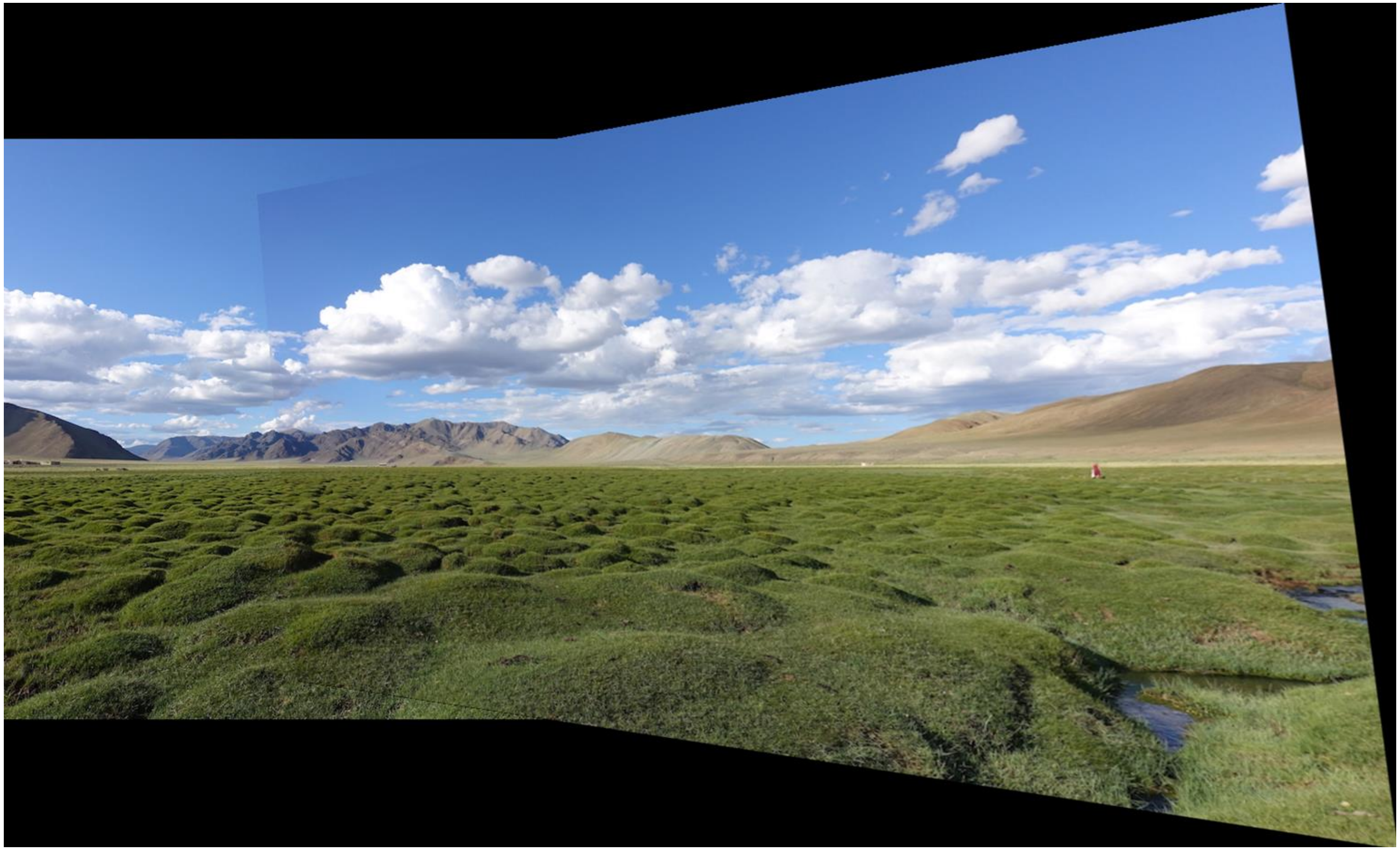


What’s happening?



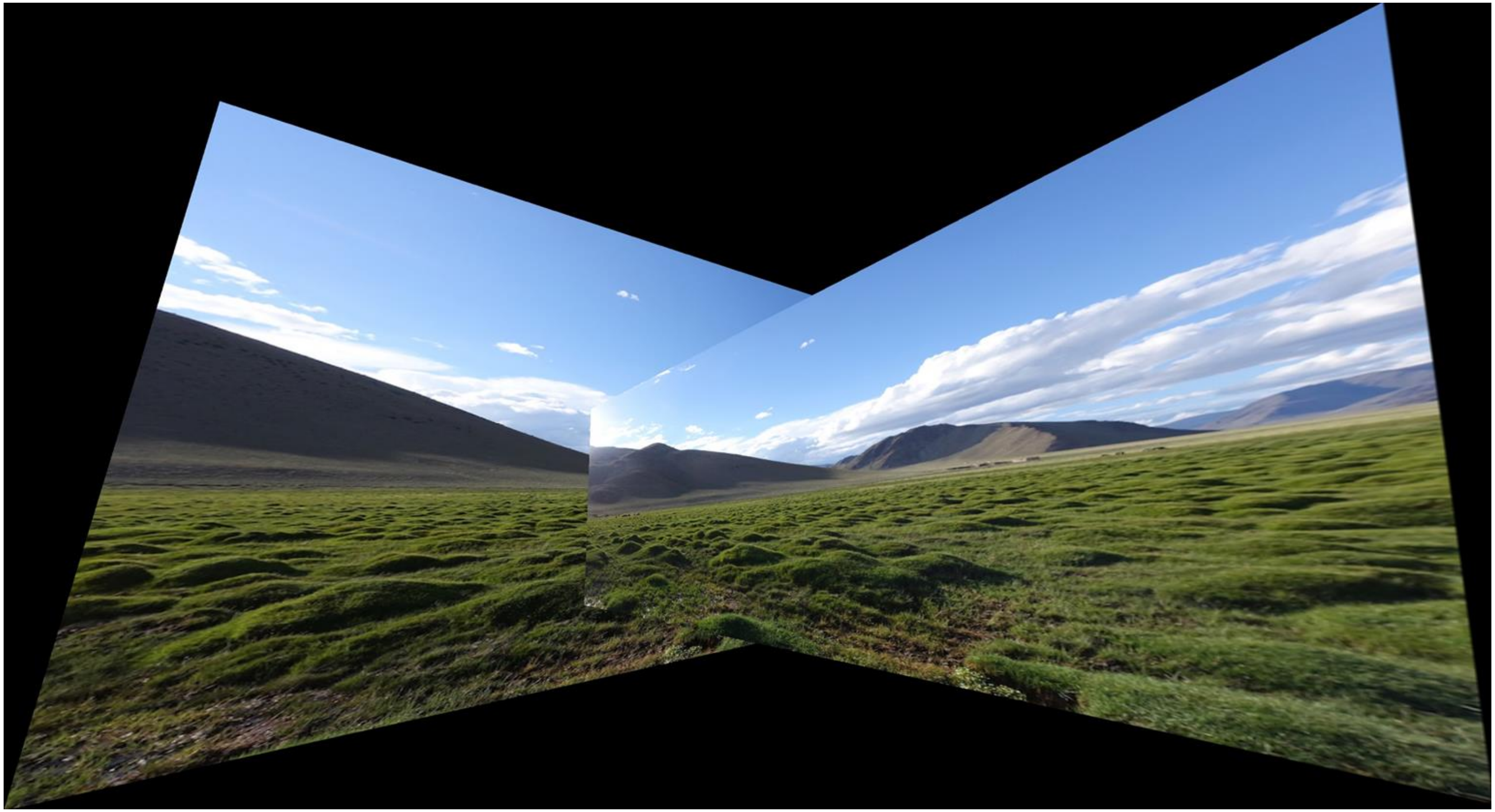
Very bad for big panoramas!

Very bad for big panoramas!



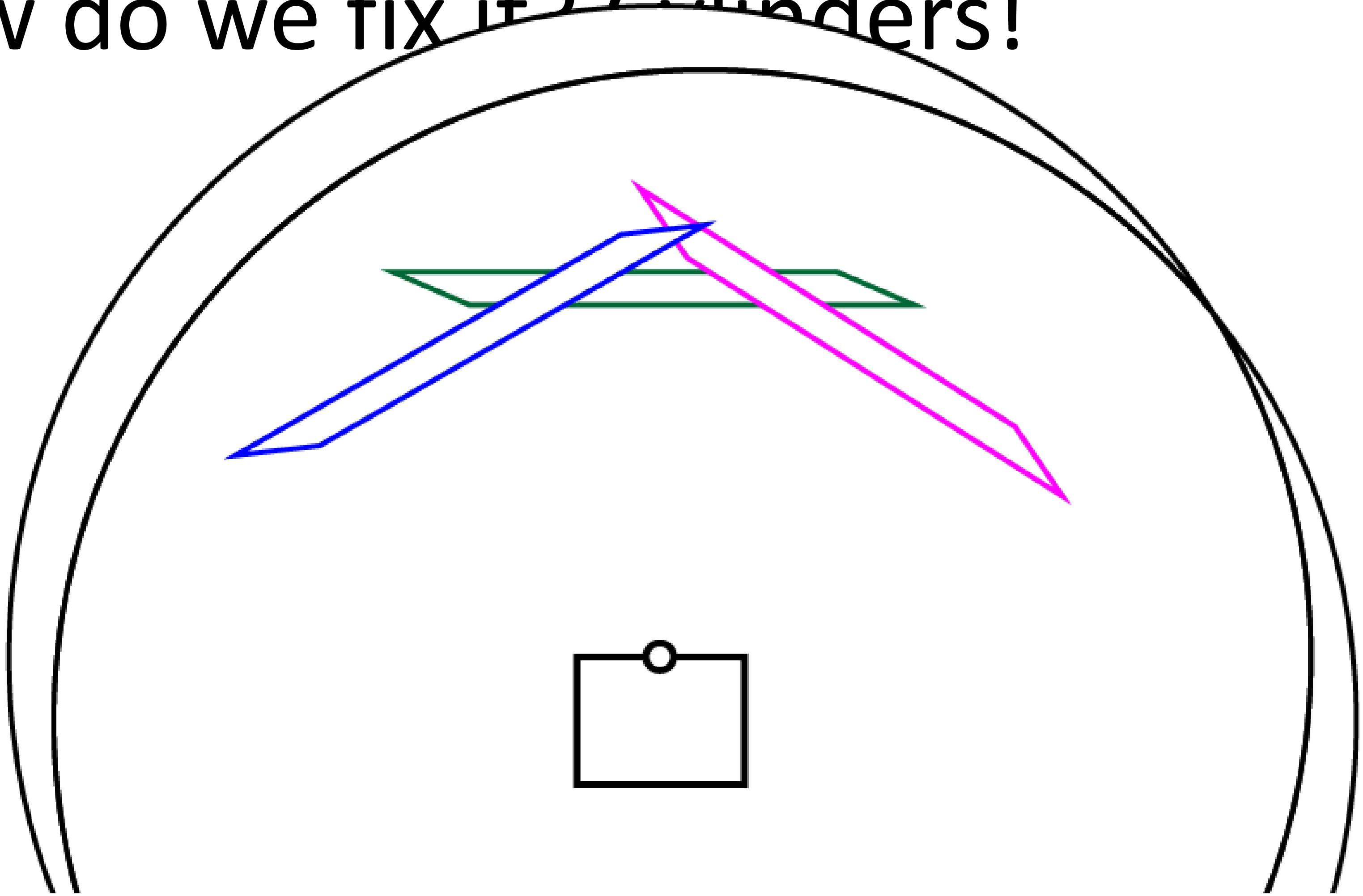
Very bad for big panoramas!

Fails :-(

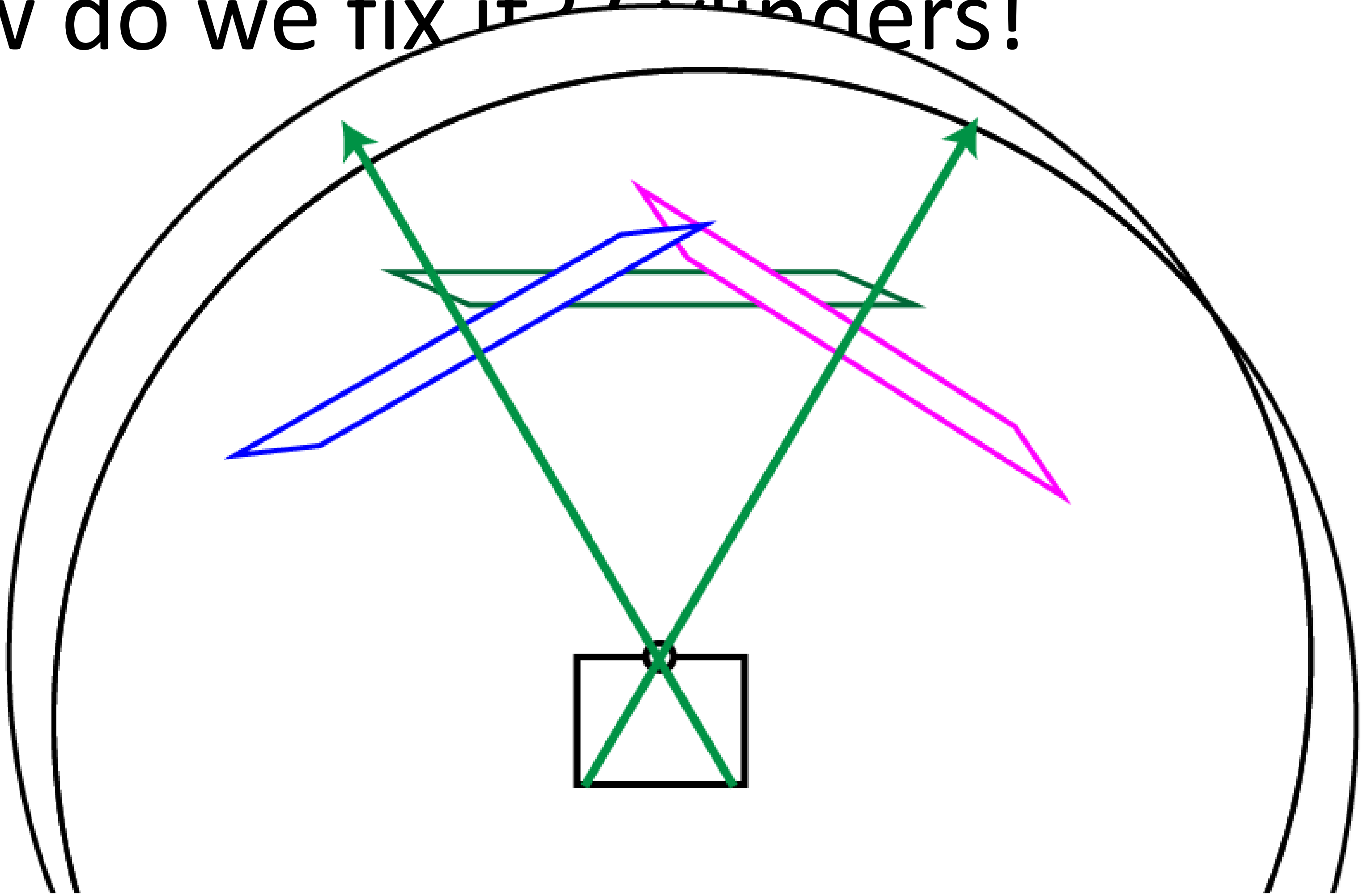


How do we fix it? Cylinders!

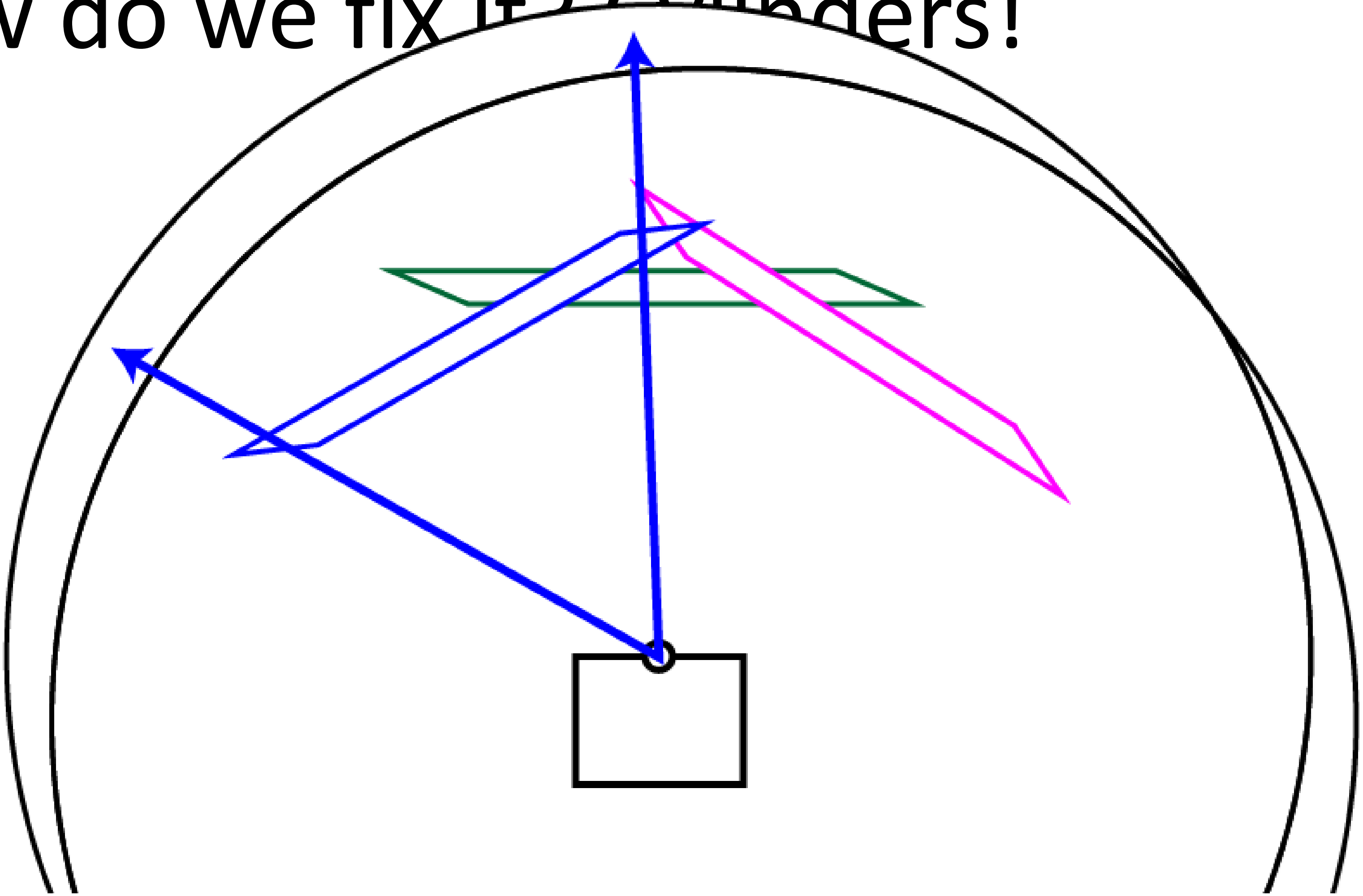
How do we fix it? Cylinders!



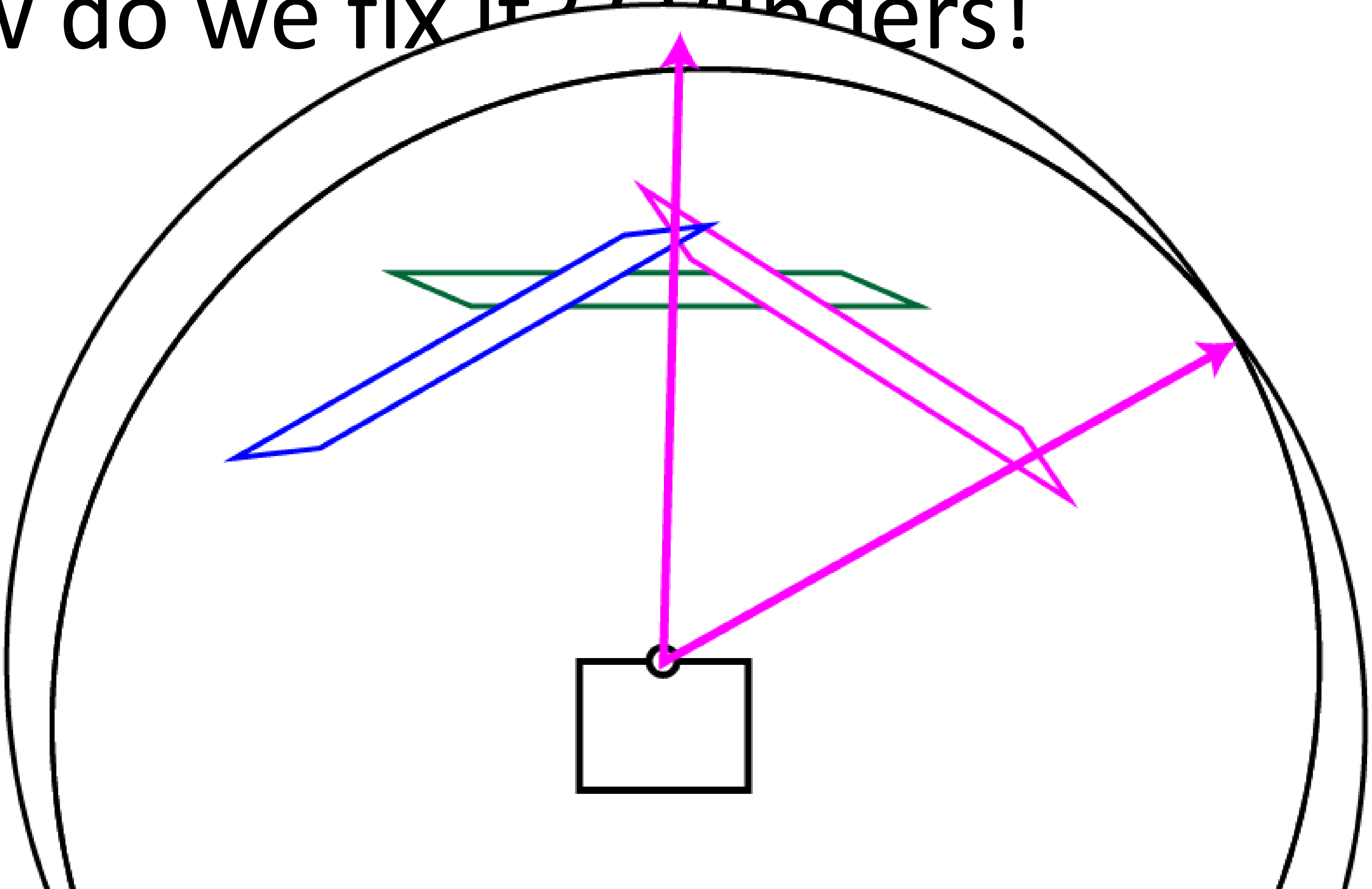
How do we fix it? Cylinders!



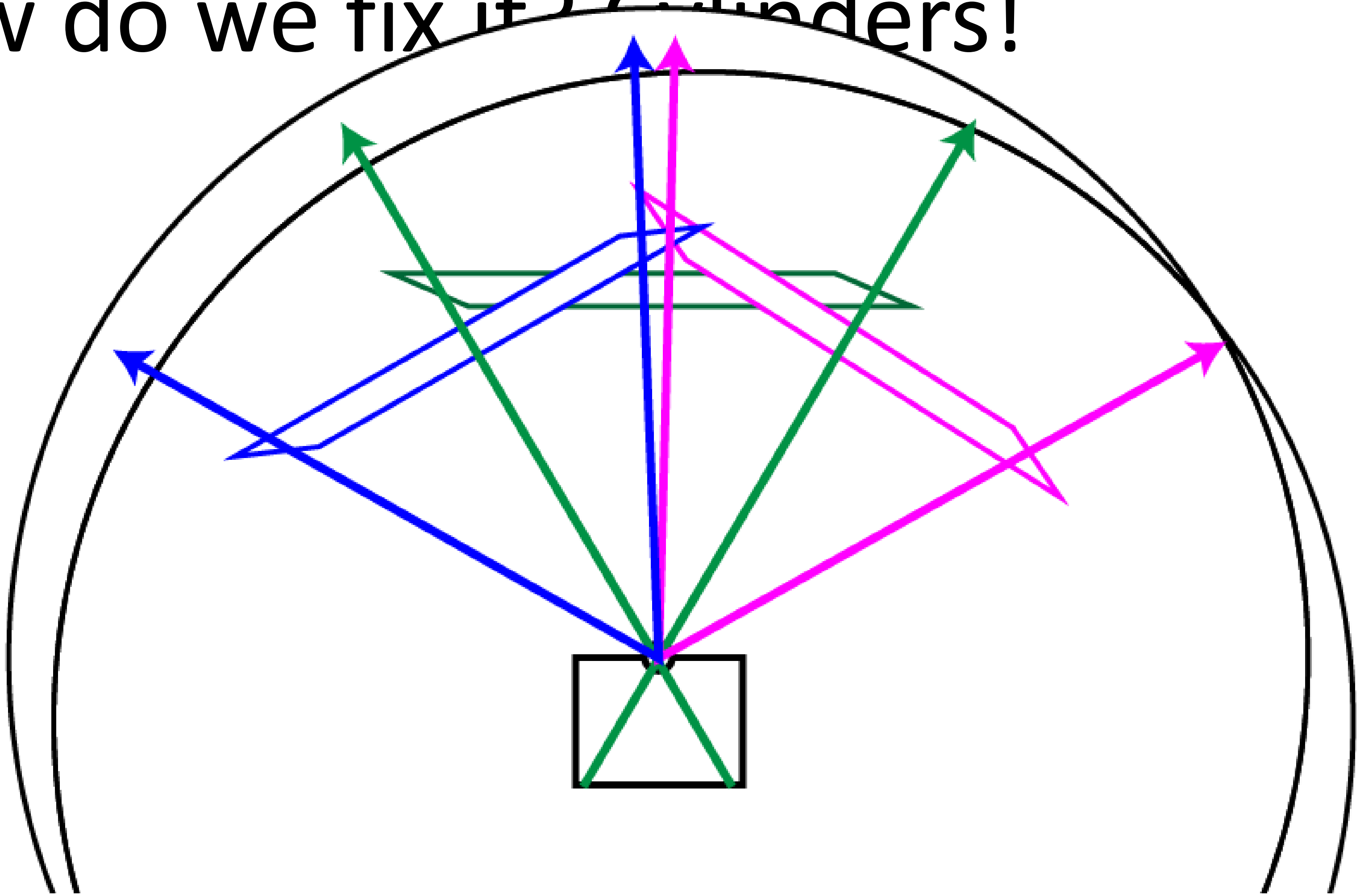
How do we fix it? Cylinders!



How do we fix it? Cylinders!



How do we fix it? Cylinders!



How do we fix it? Cylinders!

Calculate angle and height:

$\theta = (x - xc) / f$

$h = (y - yc) / f$

Find unit cylindrical coords:

$X' = \sin(\theta)$

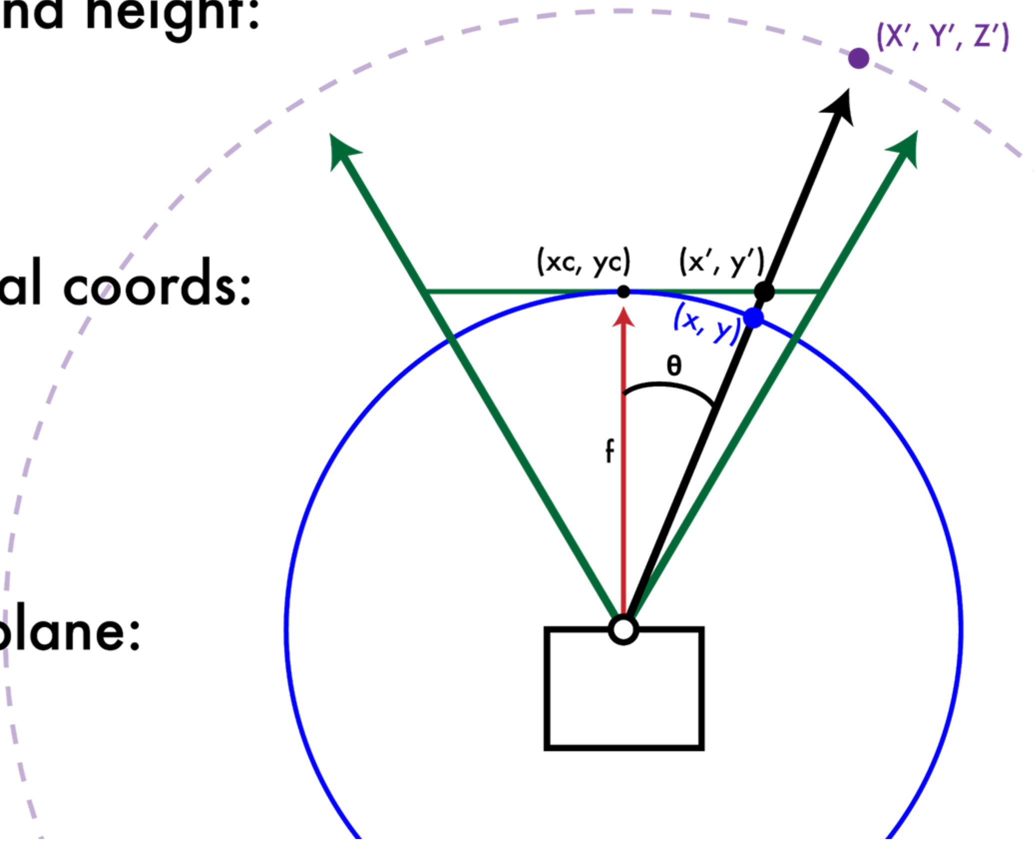
$Y' = h$

$Z' = \cos(\theta)$

Project to image plane:

$x' = f\, X'/Z' + xc$

$y' = f\, Y'/Z' + yc$



(xc,yc) = center of projection and f = focal length of camera

Dependent on focal length!

f = 300

f = 500

f = 1000

f = 1400

f = 10,000

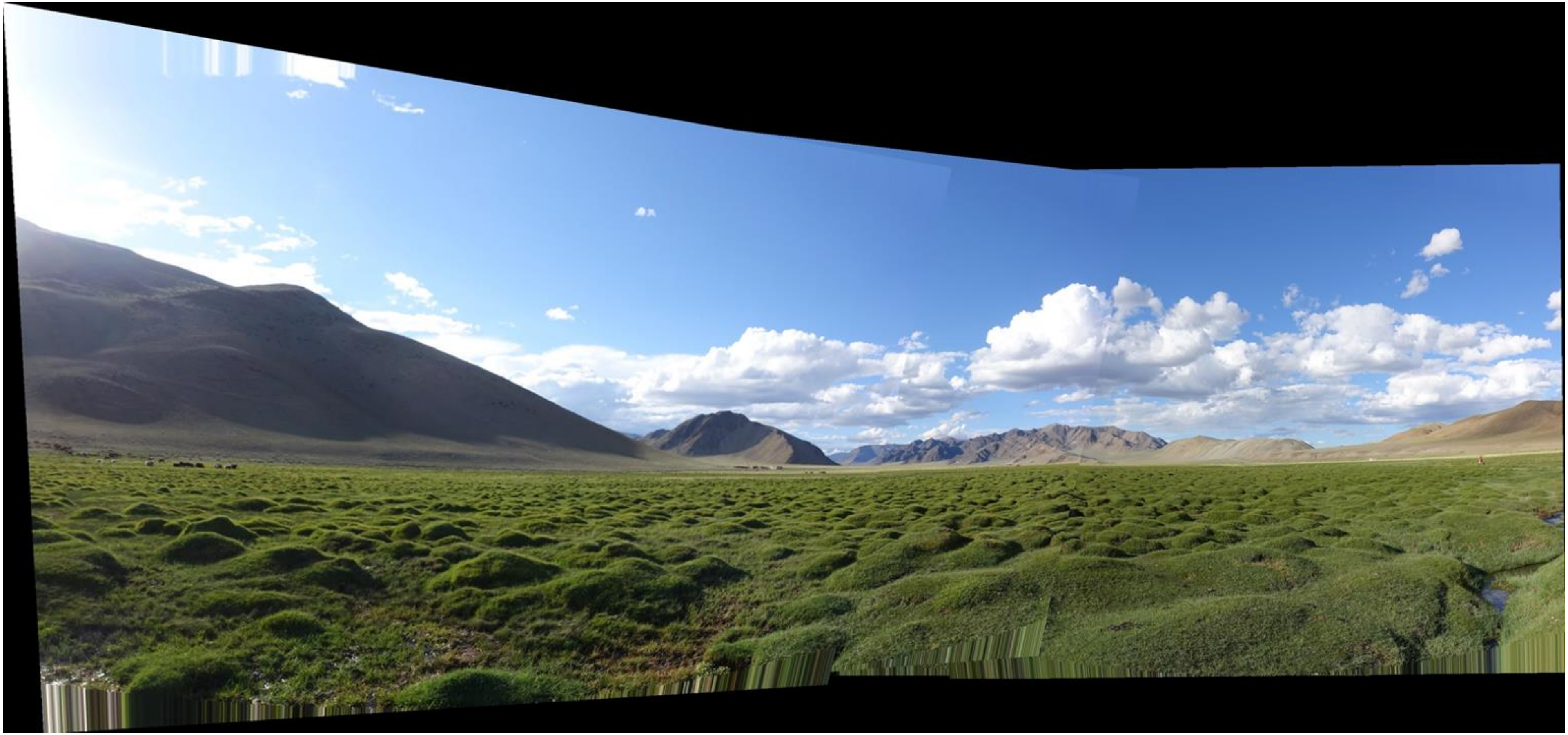
f = 10,000

Does it work?

Does it work?

Does it work?

Does it work?



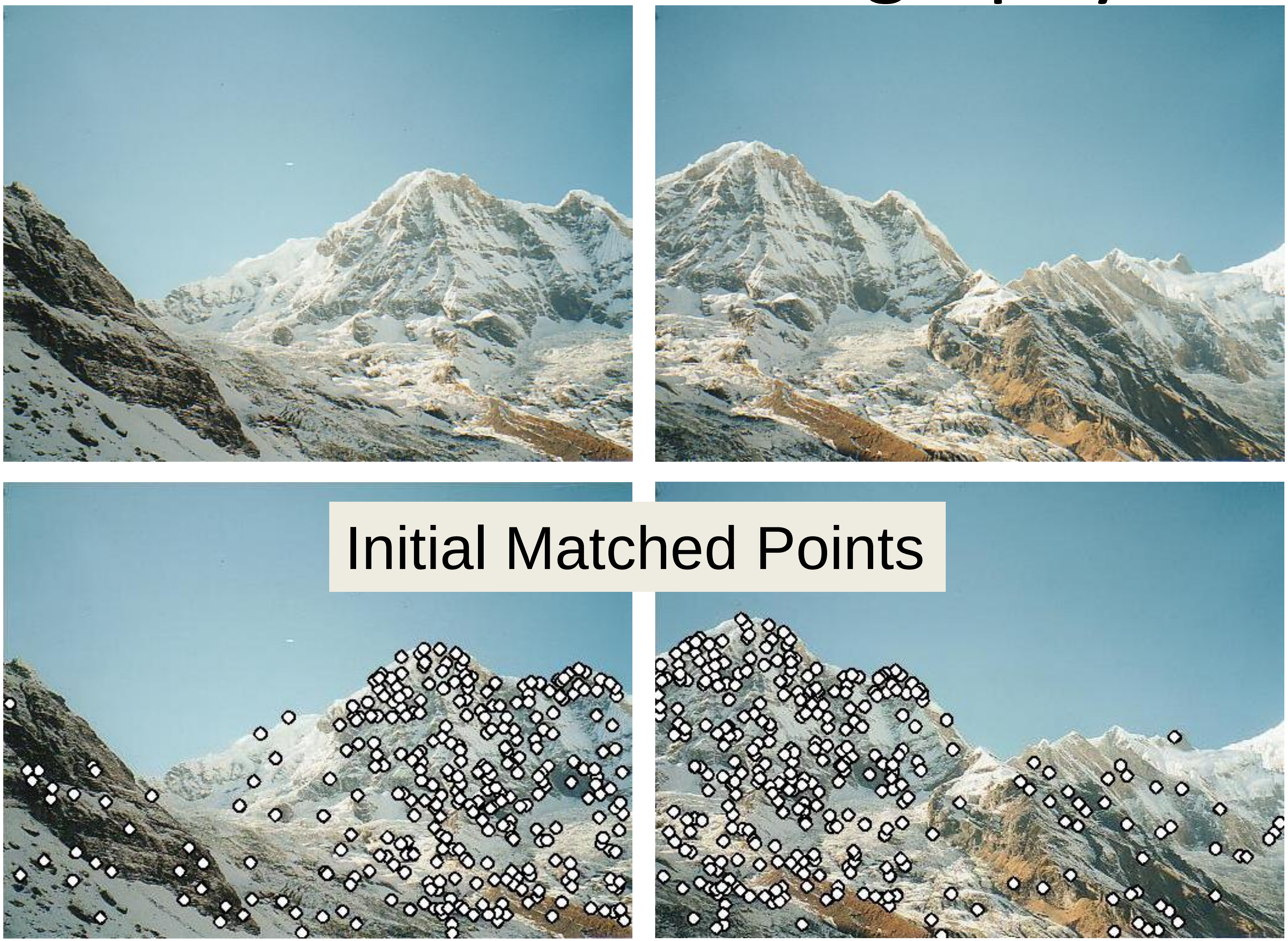
Does it work?

Does it work? Yay!

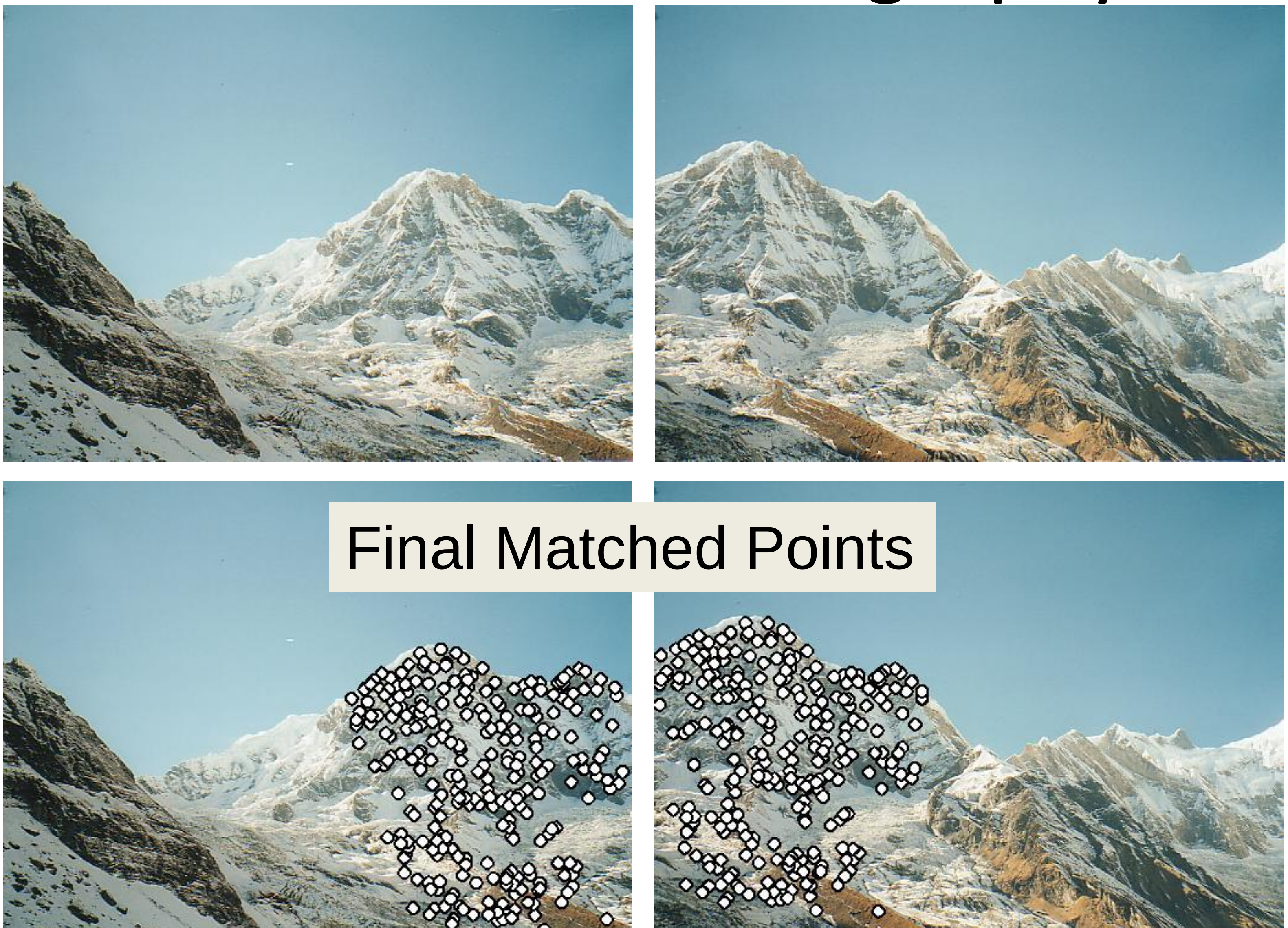
Where are we?

- We are going to build a panorama from two (or more) images.
- We need to learn about
  - Finding interest points
  - Describing small patches about such points
  - Finding matches between pairs of such points on two images, using the descriptors
  - Selecting the best set of matches and saving them
  - Constructing homographies (transformations) from one image to the other and picking the best one
  - Stitching the images together to make the panorama

RANSAC for Homography



RANSAC for Homography



RANSAC for Homography

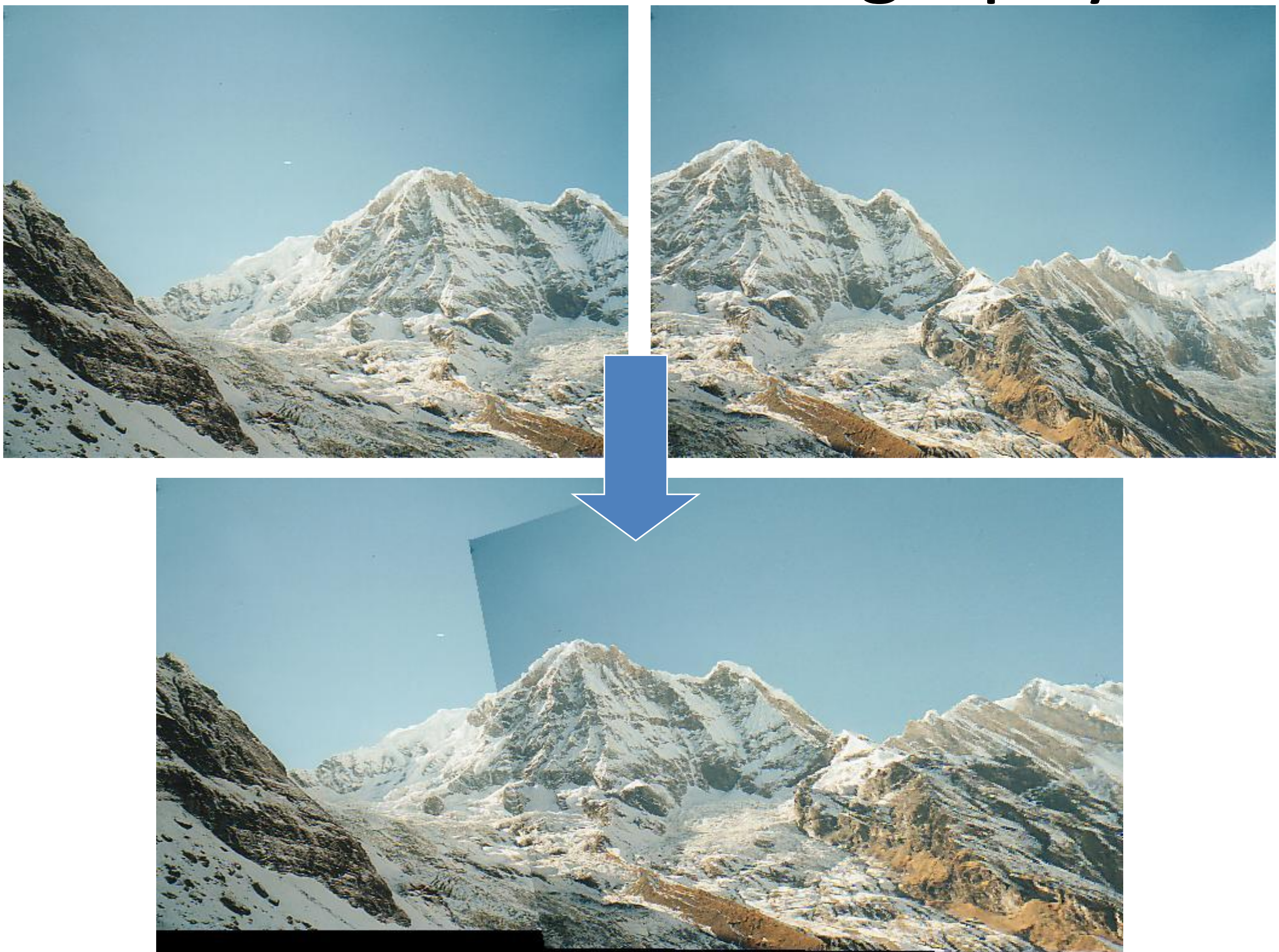
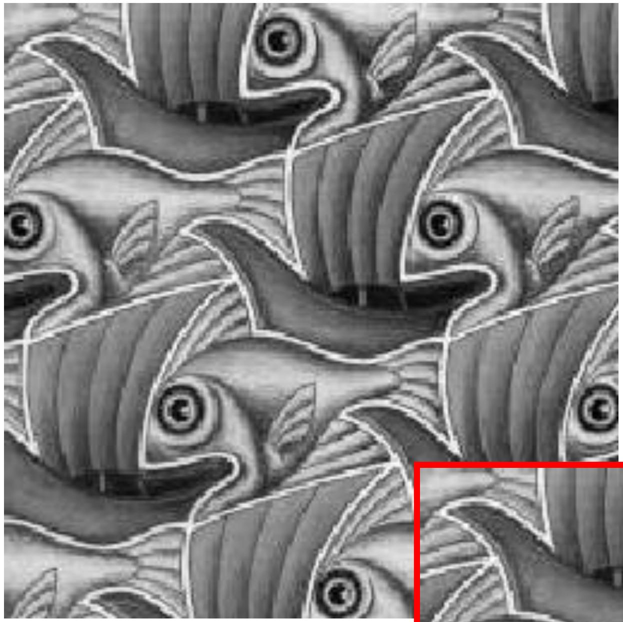
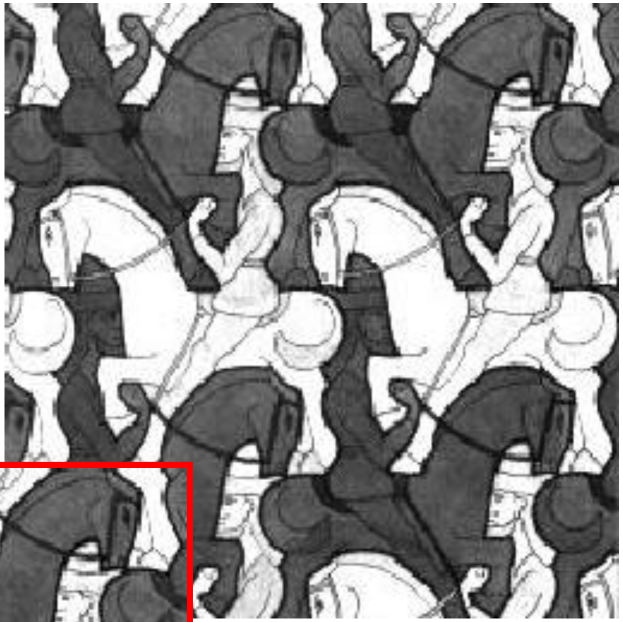


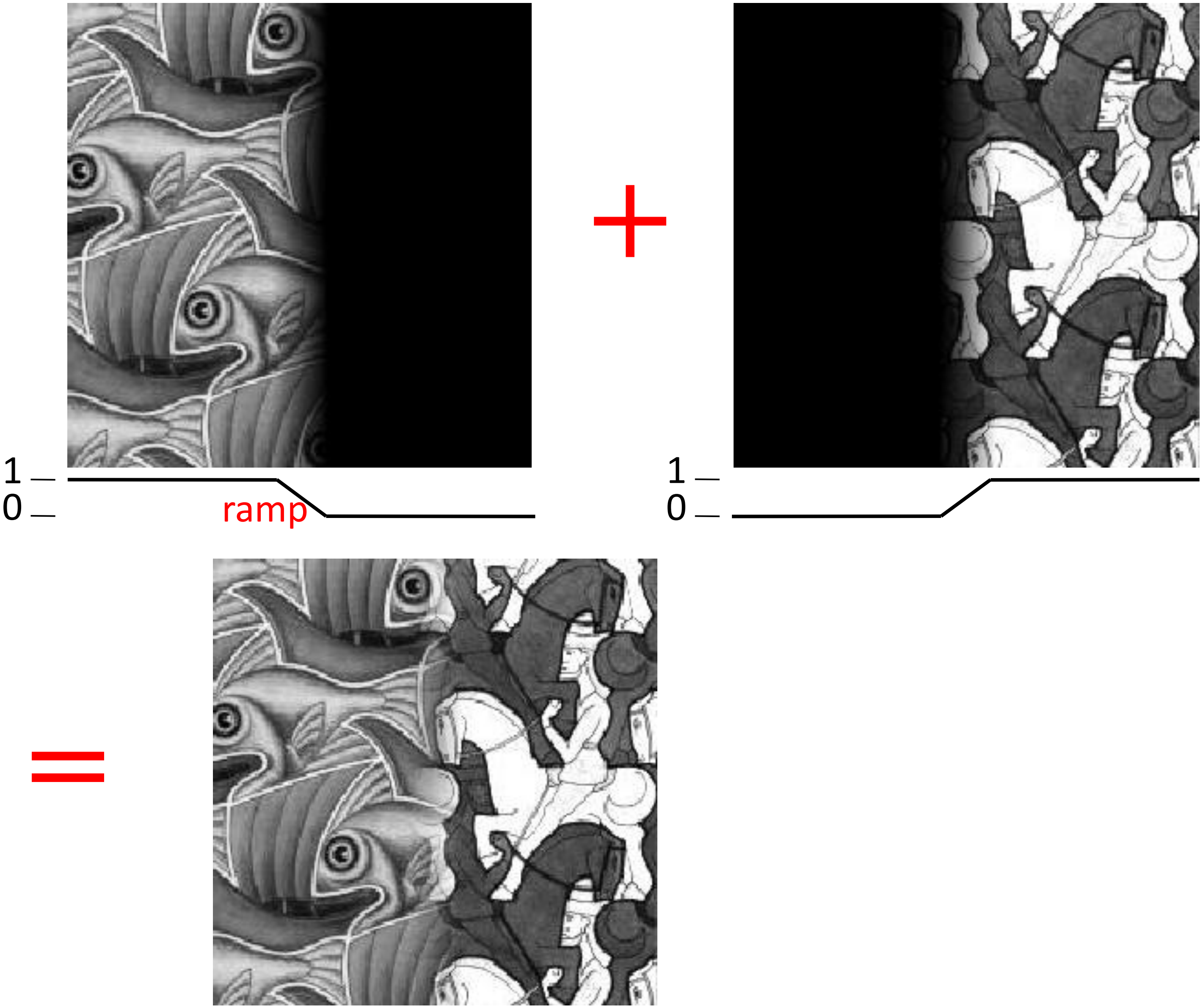
Image Blending



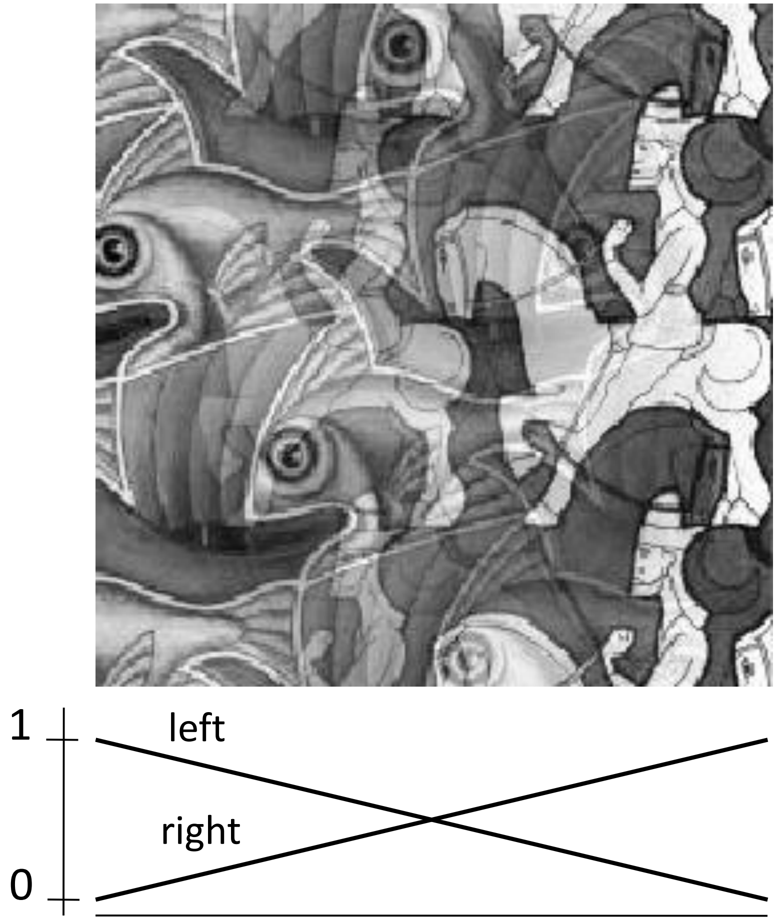


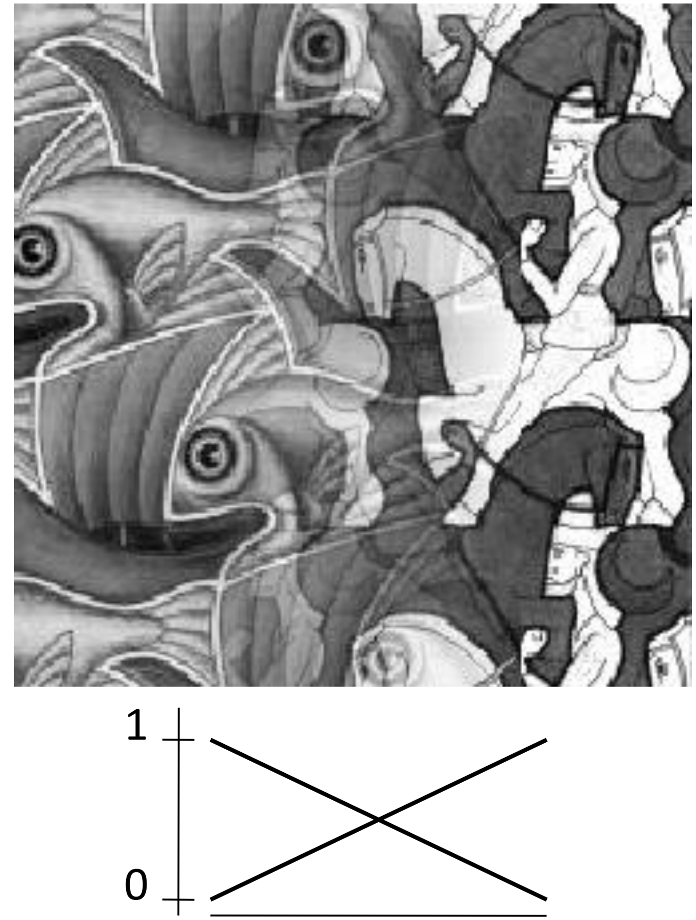
What's wrong?

Feathering

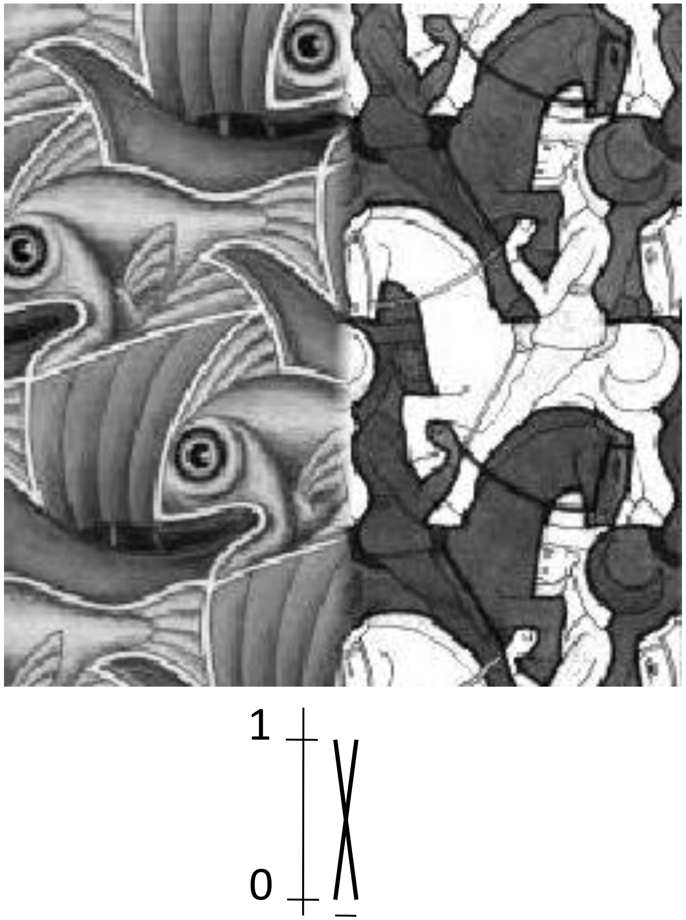


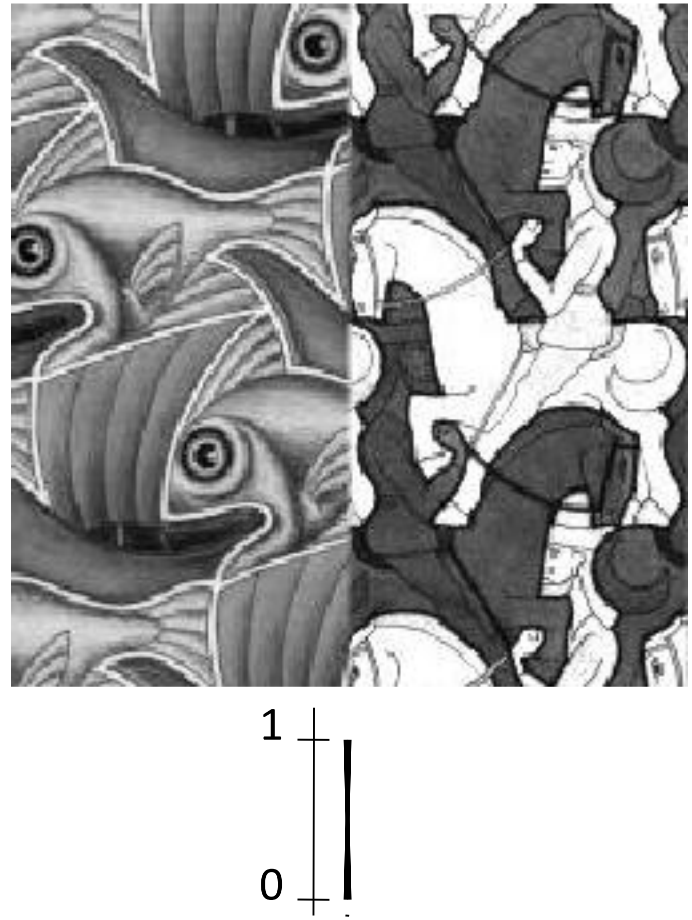
Effect of window (ramp-width) size



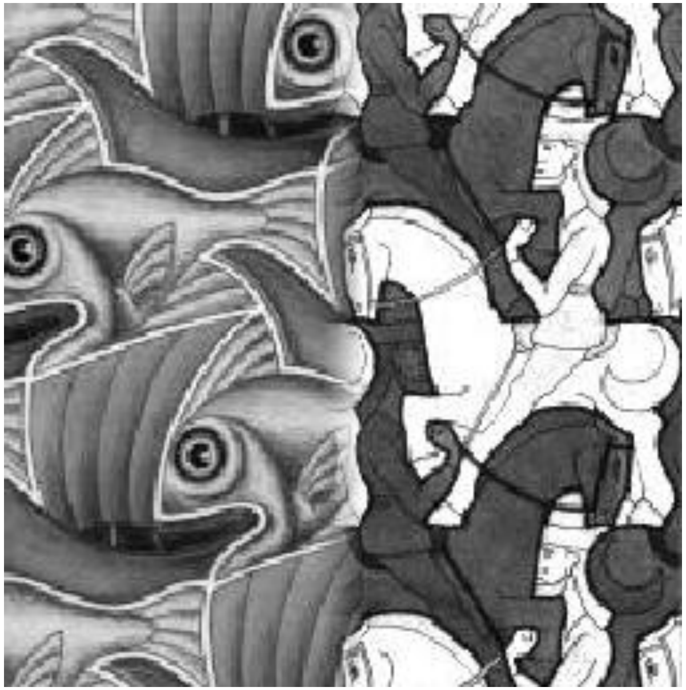


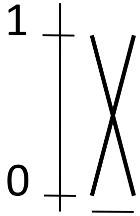
Effect of window size





Good window size





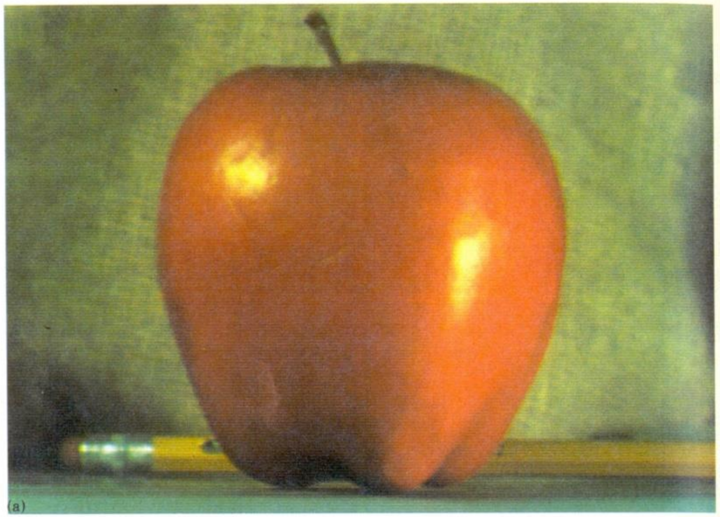
"Optimal" window: smooth but not ghosted

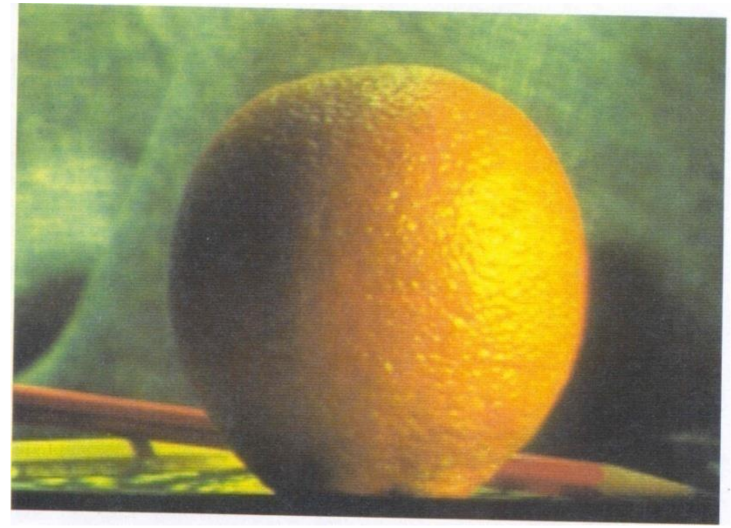
- Doesn't always work...

What can we do instead?

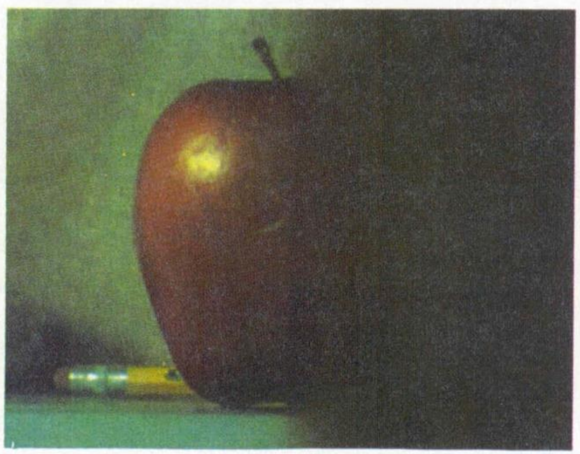
Pyramid blending

apple

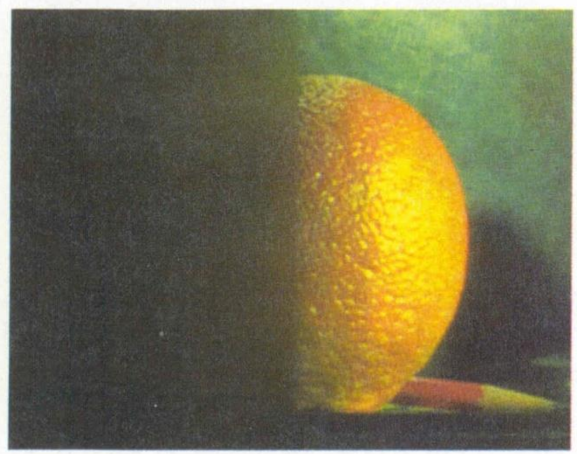




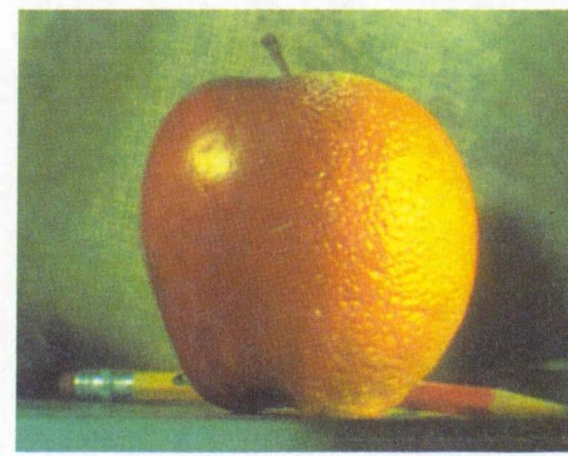
orange



(d)



(h)

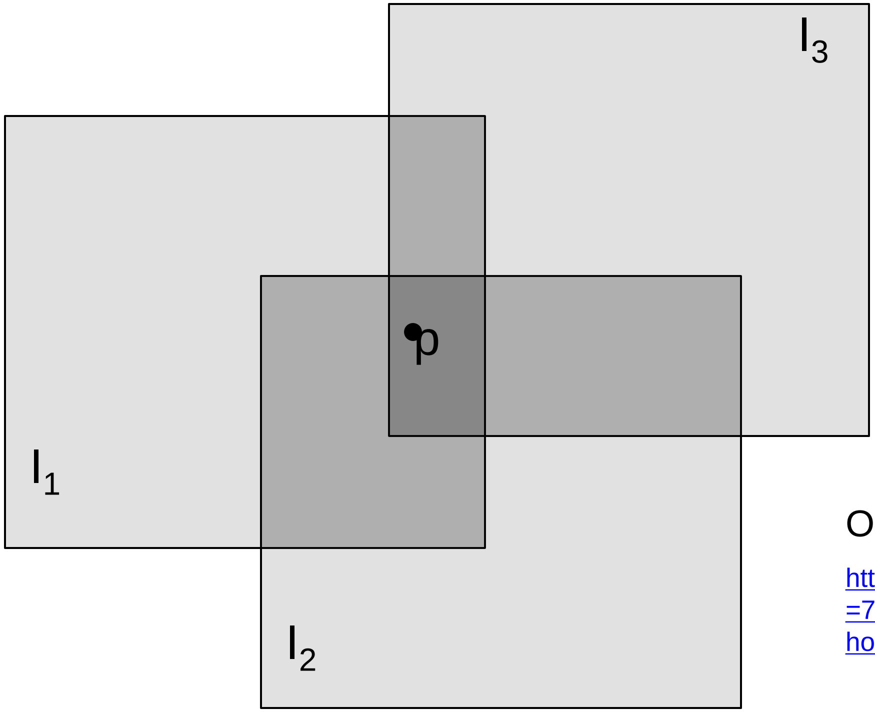


(l)

Create a Laplacian pyramid, blend each level

- Burt, P. J. and Adelson, E. H., A Multiresolution Spline with Application to Image Mosaics, ACM Transactions on Graphics, 42(4), October 1983, 217-236. http://persci.mit.edu/pub_pdfs/spline83.pdf

Alpha Blending



Optional: see Blinn (CGA, 1994) for details:

http://ieeexplore.ieee.org/iel1/38/7531/00310740.pdf?isNumber=7531&prod=JNL&arnumber=310740&arSt=83&ared=87&arAuhor=Blinn%2C+J.F.

Encoding blend weights: $I(x,y) = (\alpha R, \alpha G, \alpha B, \alpha)$

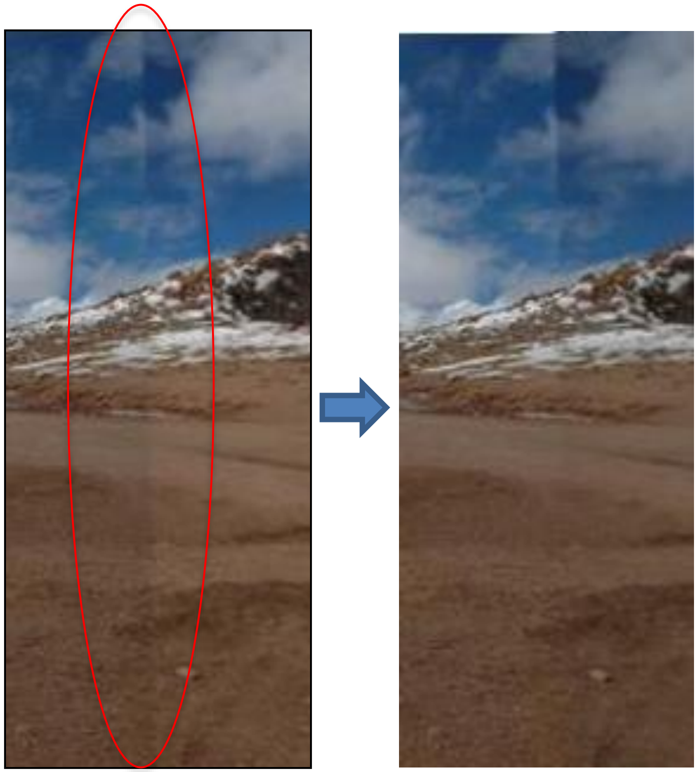
color at p = $$\frac{(\alpha_1 R_1,\ \alpha_1 G_1,\ \alpha_1 B_1) + (\alpha_2 R_2,\ \alpha_2 G_2,\ \alpha_2 B_2) + (\alpha_3 R_3,\ \alpha_3 G_3,\ \alpha_3 B_3)}{\alpha_1 + \alpha_2 + \alpha_3}$$

Implement this in two steps:

1. accumulate: add up the (α premultiplied) RGB values at each pixel
2. normalize: divide each pixel's accumulated RGB by its α value

Gain Compensation: Getting rid of artifacts

- Simple gain adjustment
  - Compute average RGB intensity of each image in overlapping region
  - Normalize intensities by ratio of averages



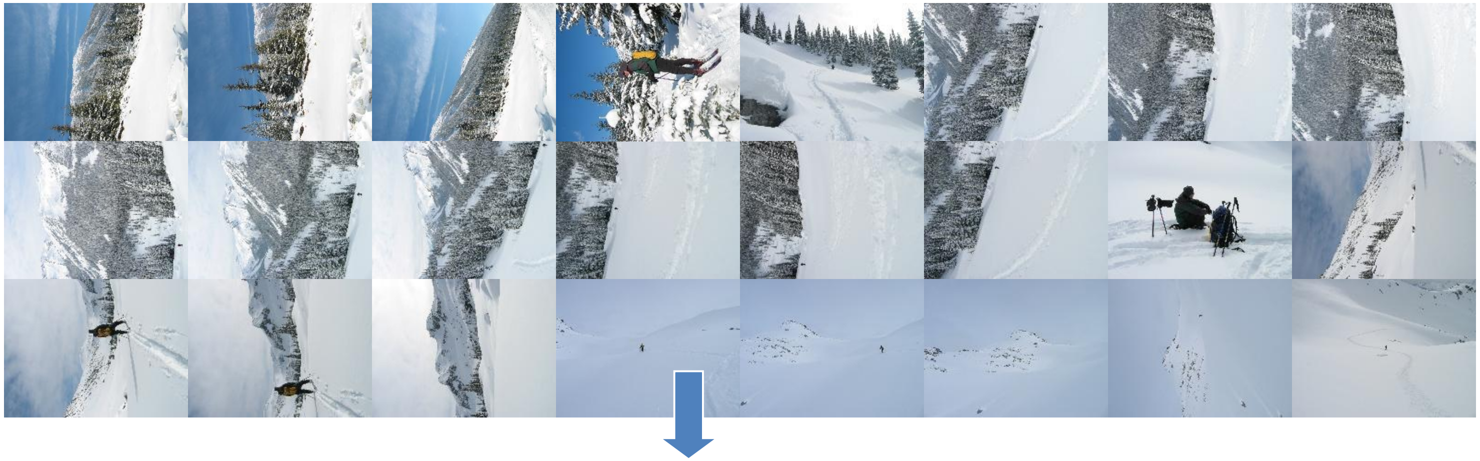
Blending Comparison

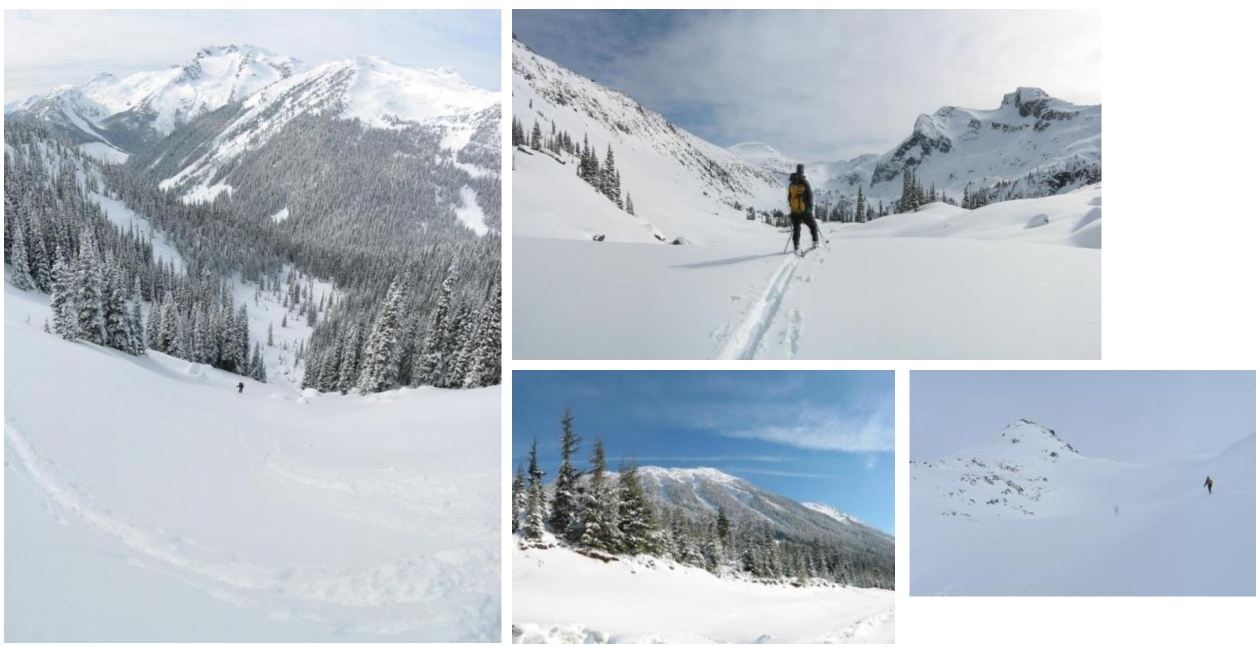
(b) Without gain compensation

(c) With gain compensation

(d) With gain compensation and multi-band blending

Recognizing Panoramas





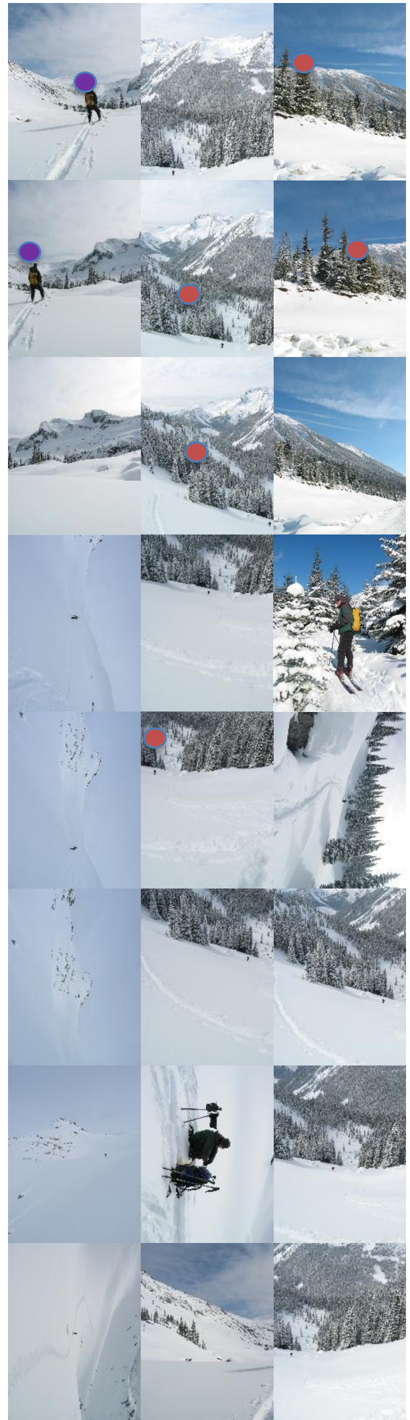
Some of following material from Brown and Lowe 2003 talk

Brown and Lowe 2003, 2007

Recognizing Panoramas

Input: N images

1. Extract SIFT points, descriptors from all images
2. Find K-nearest neighbors for each point (K=4)
3. For each image
   a) Select M candidate matching images by counting matched keypoints (m=6)
   b) Solve homography $\mathbf{H}_{ij}$ for each matched image



Recognizing Panoramas

Input: N images

1. Extract SIFT points, descriptors from all images
2. Find K-nearest neighbors for each point (K=4)
3. For each image
   a) Select M candidate matching images by counting matched keypoints (m=6)
   b) Solve homography $\mathbf{H}_{ij}$ for each matched image
   c) Decide if match is valid ($n_i > 8 + 0.3 \; n_f$)

inliers

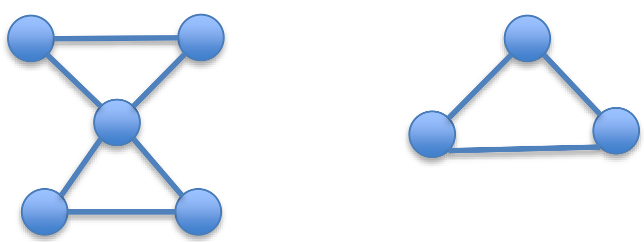
keypoints in overlapping area

Recognizing Panoramas (cont.)

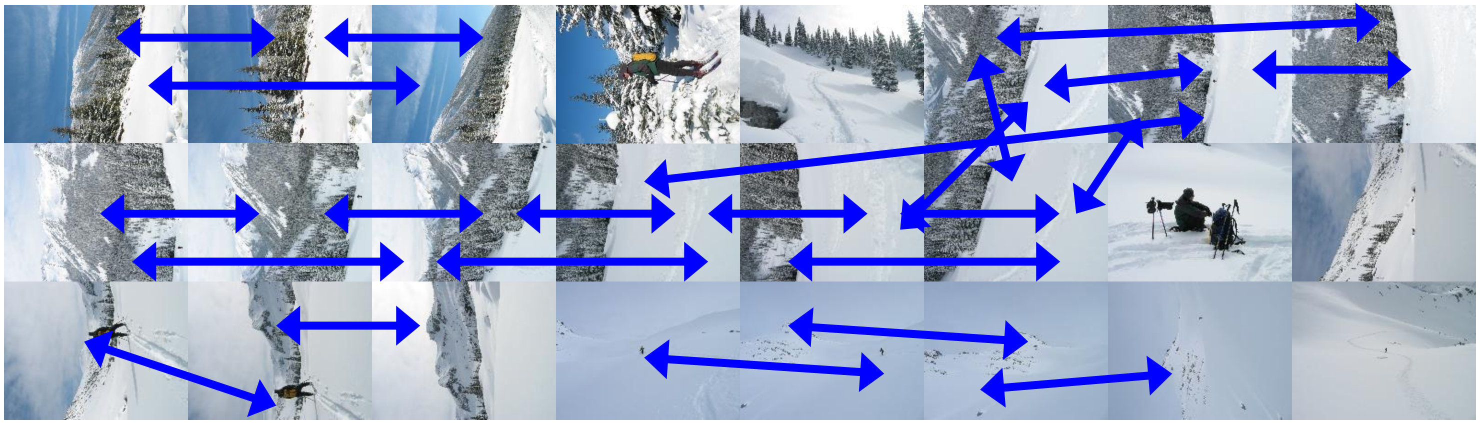
(now we have matched pairs of images)

4. Make a graph of matched pairs

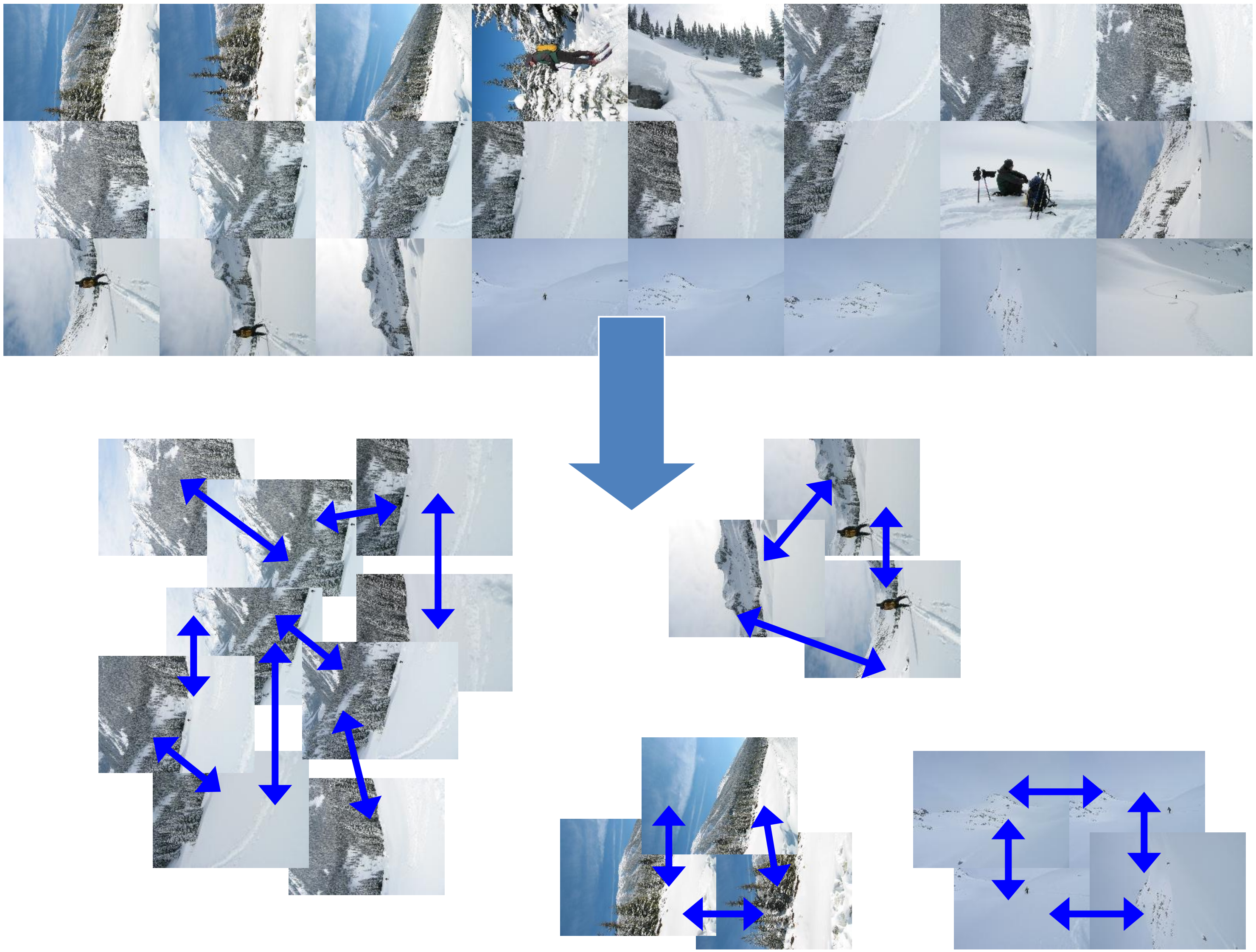
   Find connected components of the graph



Finding the panoramas



Finding the panoramas



Recognizing Panoramas (cont.)

(now we have matched pairs of images)

4. Find connected components
5. For each connected component
   a) Solve for rotation and f
   b) Project to a surface (plane, cylinder, or sphere)
   c) Render with multiband blending

Finding the panoramas

Homework 3

CREATING PANORAMAS!

Useful structures (defined in image.h)

- **Data structure for an point**

```
typedef struct{
    float x, y;
} point;
```

- **Data structure for a descriptor**

```
typedef struct{
    point p; <-pixel location
    int n; <-size of data
    float *data;
} descriptor;
```

- **Data structure for a match**

```
typedef struct{
    point p, q; <-matching points
    int ai, bi; <-matching indices of descriptor arrays
    float distance; <-dist. between matching descriptors
} match;
```

Overall algorithm

```
image panorama_image(image a, image b, float sigma, float thresh, int
nms, float inlier_thresh, int iters, int cutoff)
{
    // Calculate corners and descriptors
    descriptor *ad = harris_corner_detector(a, sigma, thresh, nms, &an);
    descriptor *bd = harris_corner_detector(b, sigma, thresh, nms, &bn);

    // Find matches
    match *m = match_descriptors(ad, an, bd, bn, &mn);

    // Run RANSAC to find the homography
    matrix H = RANSAC(m, mn, inlier_thresh, iters, cutoff);

    // Stitch the images together with the homography
    image combine = combine_images(a, b, H);

return combine;
}
```

1. Harris corner detection

- TODO #1.1: Compute structure matrix S
- TODO #1.2: Compute cornerness response map R from structure matrix S
- TODO #1.3: Find local maxes in map R using non-maximum suppression
- TODO #1.4: Compute descriptors for final corners

TODO #1.1: structure matrix

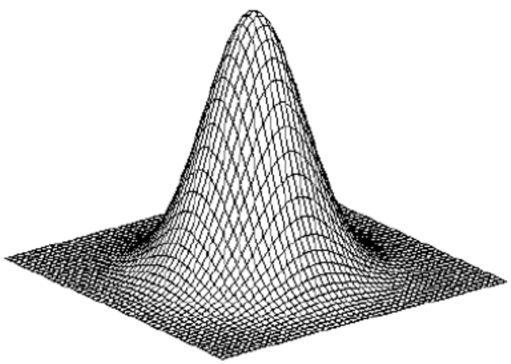
- Compute Ix and Iy using Sobel filters from HW2

- Create an empty image of 3 channels
  - Assign channel 1 to Ix^2
  - Assign channel 2 to Iy^2
  - Assign channel 3 to Ix * Iy

- Compute weighted sum of neighbors
  - smooth the image with a gaussian of given sigma

TODO #1.1.1: make a fast smoother

- Decompose a 2D gaussian to 2 1D convolutions.



Gaussian

Separable kernel

- Factors into product of two 1D Gaussians
- Discrete example:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$$

$$h_\sigma(u,v) = \frac{1}{2\pi\sigma^2} e^{-\frac{u^2+v^2}{2\sigma^2}} = \left(\frac{1}{\sqrt{2\pi}\sigma} \exp^{-\frac{x^2}{2\sigma^2}} \right) \left(\frac{1}{\sqrt{2\pi}\sigma} \exp^{-\frac{y^2}{2\sigma^2}} \right)$$

TODO #1.2: response map

- For each pixel of the given structure matrix S:
  - Get Ix^2, Iy^2 and IxIy from the 3 channels
  - Compute $Det(S) = Ix^2 * Iy^2 – IxIy * IxIy$
  - Compute $Tr(S) = Ix^2 + Iy^2$
  - Compute $R = Det(S) – 0.06 * Tr(S) * Tr(S)$

TODO #1.3: NMS

- For each pixel ‘p’ of the given response map R
  - get value(p)
  - loop over all neighboring pixels ‘q’ in a 2w+1 window
    - +/- w around the current pixel location
    - if value(q) > value (p), value(p) = -999999 (very low)
  - set ‘p’ to value(p)

TODO #1.4: corner descriptors

- Given: Response map after NMS
- Initialize count; loop over each pixel
  - if pixel value > threshold, increment count

- Initialize descriptor array of size ‘count’
- Loop over each pixel again
  - if pixel value > threshold, create descriptor for that pixel
    - use describe_index() defined in harris_image.c
  - add this new descriptor to the array

2. Matching descriptors

- TODO #2.1: Implement L1 distance
- TODO #2.2.1: Find best matches from descriptor array “a” to descriptor array “b”
- TODO #2.2.2: Eliminate duplicate matches to ensure one-to-one match between “a” and “b”
- TODO #2.3: Project points given a homography and compute inliers from an array of matches
- TODO #2.4: Implement RANSAC algorithm
- TODO #2.5: Combine images

TODO #2.1: Distance Metrics

- For comparing patches we'll use L1 distance.

```
// Calculates L1 distance between to floating point arrays.
// float *a, *b: arrays to compare.
// int n: number of values in each array.
// returns: l1 distance between arrays (sum of absolute differences).
float l1_distance(float *a, float *b, int n)
{
    // TODO: return the correct number.
    return 0;
}
```

TODO #2.2.1: best matches

- For each descriptor ‘a_r’ in array ‘a’:
  - initialize min_distance and best_index
  - for each descriptor ‘b_s’ in array ‘b’:
  - compute L1 distance between a_r and b_s
    - sum of absolute differences
  - if distance < min_distance:
    - update min_distance and best_index

TODO #2.2.2: remove duplicates

- Sort the matches based on distance (shortest is first)
- Initialize an array of 0s called ‘seen’

- Loop over all matches:
  - if b-index of current match is ≠1 in ‘seen’
    - set the corresponding value in ‘seen’ to 1
    - retain the match
  - else, discard the match

TODO #2.3.1: point projection

- Given point p, set matrix c_{3x1} = [x-coord, y-coord,1]

- Compute $M_{3x1} = H_{3x3} * c_{3x1}$ with given Homography

- Compute x,y coordinates of a point ’q’:
  - x-coord: M[0] / M[2]
  - y-coord: M[1] / M[2]

- Return point ‘q’

TODO #2.3.2, 2.3.3: L2 distance and model inliers

- Loop over each match from array of matches (starting from end):
  - project point ‘p’ of match using given ‘H’
  - compute L2 distance between point ‘q’ of match and the projected point
  - if distance < given threshold:
    - it is an inlier; bring match to the front of array (swap)
    - update inlier count

TODO #2.3.4:Fitting the homography

- Use the matrix operations discussed in class to solve equations like M*a = b.

- Most of this is already implemented
  - you just have to fill in the matrices M and b with our match information.

TODO #2.4-2.5: implement RANSAC

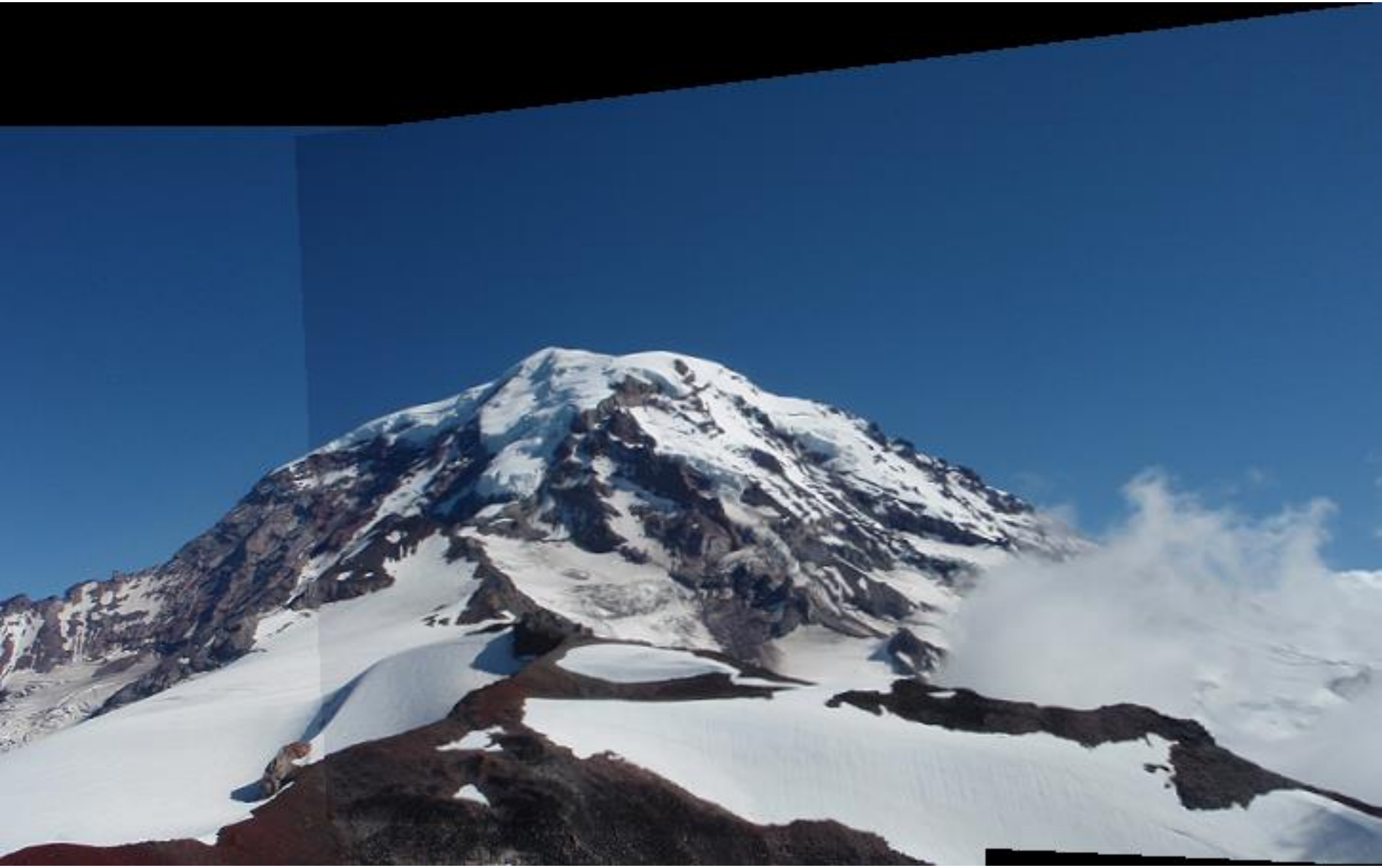
- For each iteration:
  - compute homography with 4 random matches
    - call compute_homography() with argument 4
  - if homography is empty matrix, continue
  - else compute inliers with this homography
  - if #inliers > max_inliers:
    - compute new homography with all inliers
    - update best_homography with this new homography
    - update max_inliers with #inliers computed with this new homography unless new homography is empty
    - if updated max_inliers > given cutoff: return best_homography
- Return best_homography

TODO #2.6: combine images

- Project corners of image ‘b’ and create a big empty image ‘c’ to place image ‘a’ and projected ‘b’. This part is given in the code.

- For each pixel in image ‘a’, get pixel value and assign it to ‘c’ after proper offset

- For each pixel in image ’c’ within projected bounds:
  - project to image ‘b’ using given homography
  - get pixel value at projected location using bilinear interpolation
  - assign the value to ’c’ after proper offset



3. Cylindrical Projection

- Implement cylindrical projection for an image
  - See lecture slides for the formulas
  - See Tryhw3, which will call the panorama code to do the stitching.
  - See code for the code stub you will fill in to cylinderize an image.

Have Fun