CS 188-2
Midterm Review

Albert Zhao

10/25/2019

Image Filtering

- Image: represented as 2D array of intensity values I(x, y) (or for RGB images, three intensity values)
- Filtering is used to process an image in various ways
- Treating image as a 2D signal:
  - Low-pass filter: keeps low-frequency portion, removes high-frequency portion -> blurs (smoothes) the image
  - High-pass filter: keeps high-frequency portion, removes low-frequency portion -> sharpens image, can be used to detect edges
  - As image is sum of low and high-frequency components, a high-pass filter can be implemented by difference of image and low-pass filtered image (and vice versa)

Convolutions

- Image filtering implemented via convolutions
- Convolution:
  - Flips the convolution kernel (filter)
  - Perform dot product between elements of kernel and pixels of image
  - Slide it around, repeatedly computing the dot product using different image patches

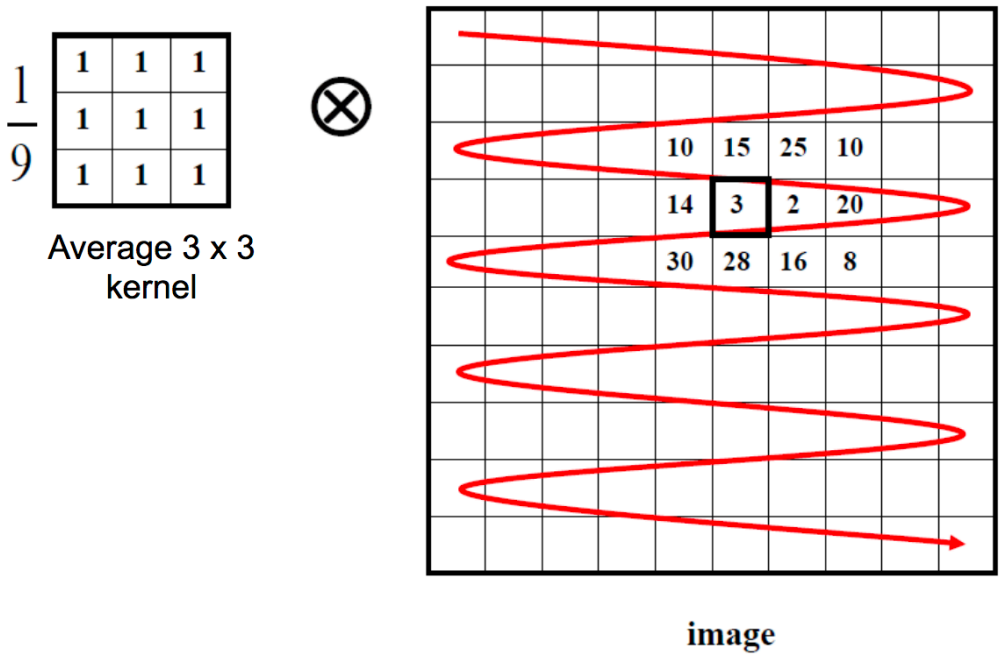
$$I'(x,y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} I(x+i, y+j) \cdot \text{filter}^{\text{flipped}}(i, j)$$

Padding

- Without any padding, the output of convolution is typically smaller than image
- To keep convolution output the same size as image, start convolution with filter center at top left corner of image and make some assumption about image values outside the borders
- Generally, assume values outside image are 0 (zero padding)

Other Filters

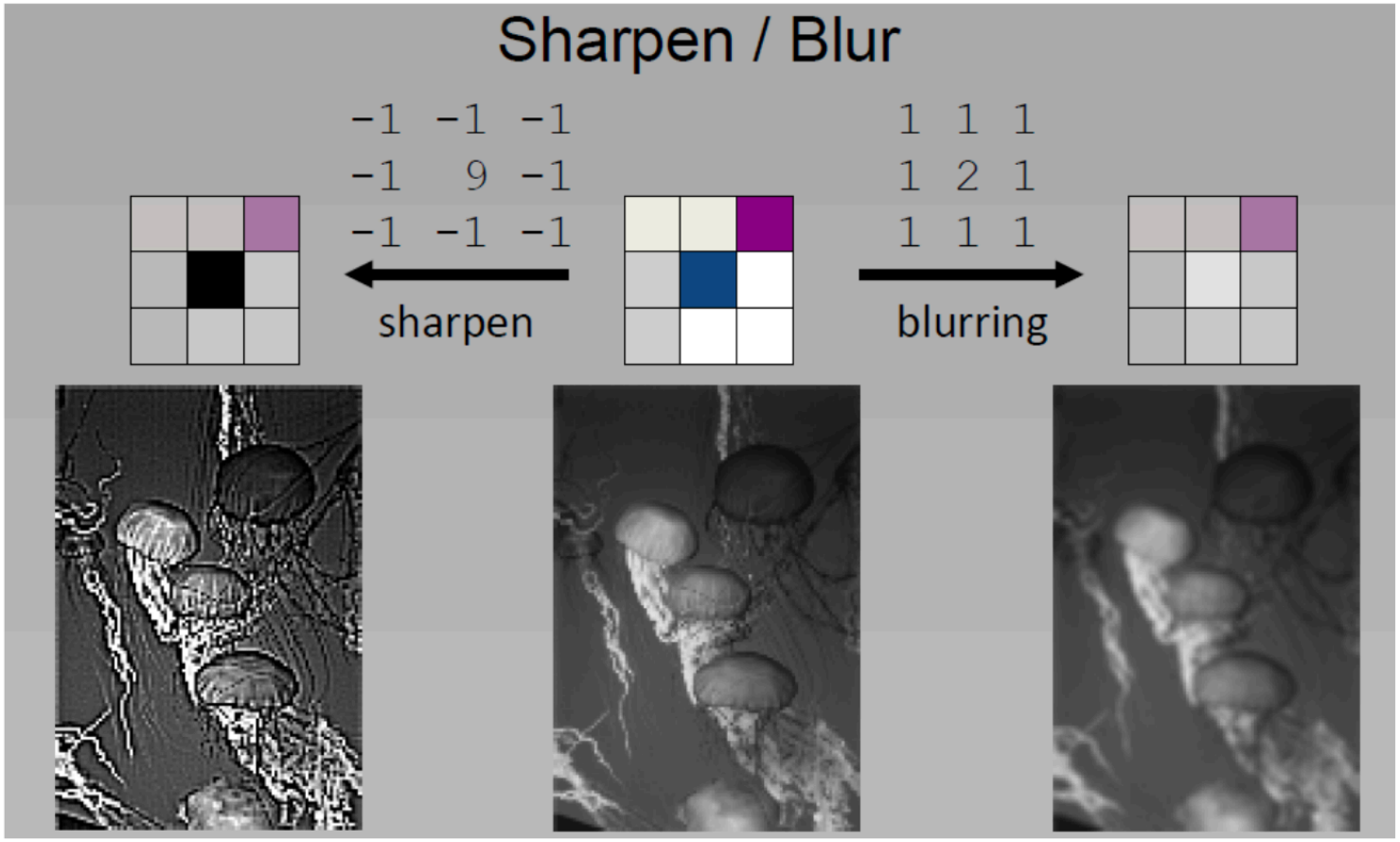
- Mean/median/max/min filter takes mean/median/max/min. respectively, of corresponding image patch; apply each filter by sliding along image similar to convolution



Sharpen vs Blur (High-pass vs Low-pass)

- Sharpening / high-pass filters try to magnify differences in intensity
- Blurring / low-pass filters try to reduce differences in intensity (generally via some sort of weighted sum)
- Examples (without normalization) on next slide

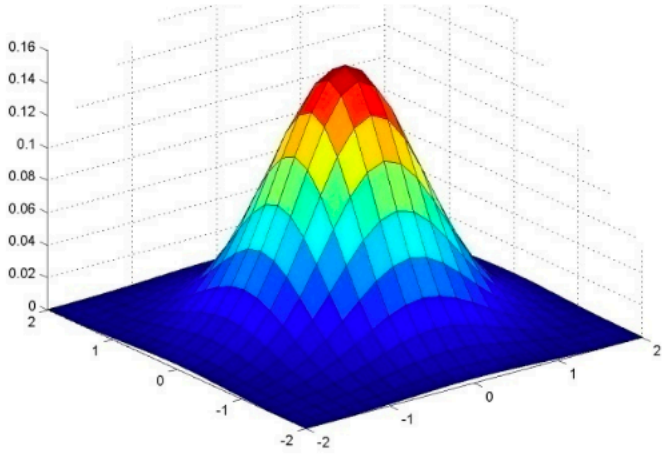
Sharpen vs Blur (High-pass vs Low-pass)



Gaussian Filter

- Special example of low-pass filter often used for blurring / removing noise
- Kernel is based on values of Gaussian distribution with some scale but with finite support

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$



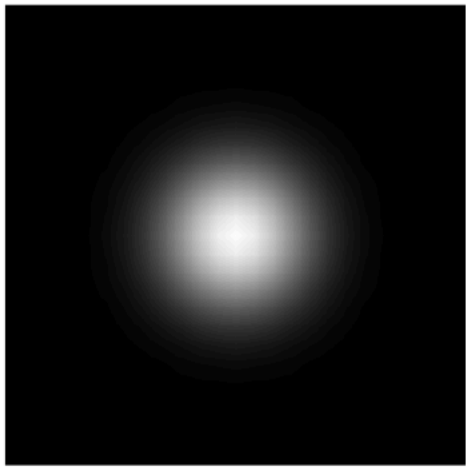


Image Gradients

- Image gradients are useful for a variety of applications such as edge detection, feature extraction (SIFT for example)
- Can be computed via convolution
- We start from the approximation below to derive some filters

Discrete approximation

$$\frac{\partial I}{\partial x} \approx \frac{I(x+1, y) - I(x-1, y)}{2}$$

Image Gradients

- Sample filters for computing gradients in y and x directions (y direction is top, x direction is bottom), simplest version on left, Prewitt filter in middle, Sobel filter on right (Prewitt and Sobel are more robust)

Filter

0	-1	0
0	0	0
0	1	0

-1	-1	-1
0	0	0
1	1	1

-1	-2	-1
0	0	0
1	2	1

0	0	0
-1	0	1
0	0	0

-1	0	1
-1	0	1
-1	0	1

-1	0	1
-2	0	2
-1	0	1

Image Gradients

- Gradient is really vector of x and y-derivative with magnitude and direction given below

Gradient

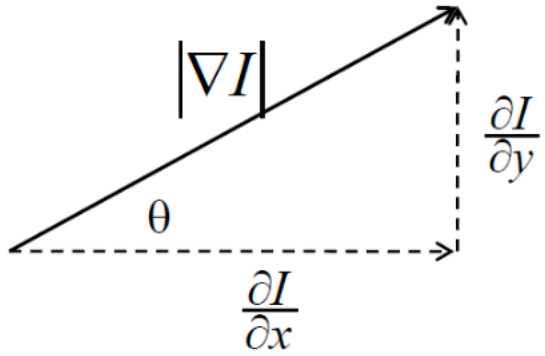
$$\nabla I = \begin{bmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{bmatrix}$$

Magnitude

$$|\nabla I| = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2} \approx \left|\frac{\partial I}{\partial x}\right| + \left|\frac{\partial I}{\partial y}\right|$$

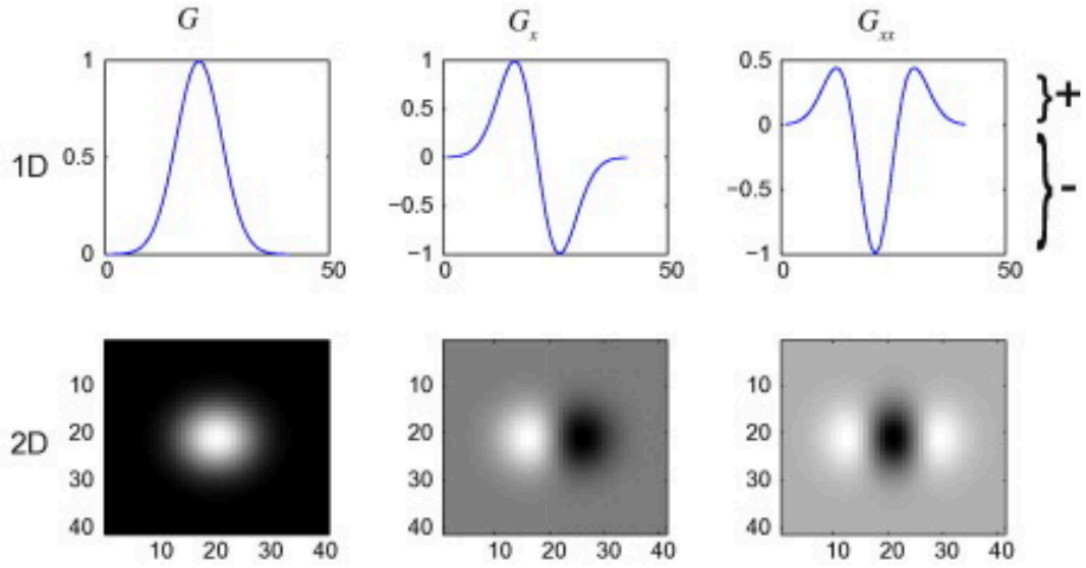
Direction

$$\theta = \tan^{-1} \frac{\left(\frac{\partial I}{\partial y}\right)}{\left(\frac{\partial I}{\partial x}\right)}$$



Gaussian Derivative and Laplacian

- More robust way to compute derivative: apply Gaussian blur and then take derivative
- Second derivative is also useful (used for edge detection and keypoint detection); Laplacian is applying Gaussian blur and then derivative twice



Separable Convolutions

- Separable convolution: its kernel can be expressed as outer product of two vectors (two 1-D kernels)
- Sobel kernel:

$$\begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} +1 & 0 & -1 \end{bmatrix}$$

- Gaussian kernel is also separable

Separable Convolutions

- Since convolution is associative, can quickly evaluate separable convolution using two 1-D convolutions
- For an M x M kernel, takes $O(M^2)$ operations to evaluate convolution, O(M) operations to evaluate separable convolution (since we evaluate two 1-D convolutions)

$$f * (g * h) = (f * g) * h$$

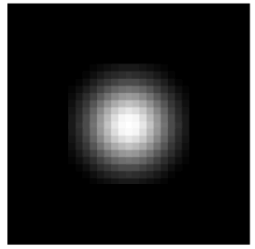
Bilateral Filtering

- Gaussian filter smoothes image but also smoothes edges
- How to avoid this?
- Add Gaussian factor that uses difference in intensities
- Now two parameters: standard deviations for each Gaussian factor

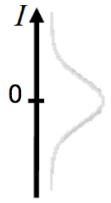
$$BF[I]_{\mathbf{p}} = \underbrace{\frac{1}{W_{\mathbf{p}}}}_{\text{new}} \sum_{\mathbf{q}\in S} \underbrace{G_{\sigma_s}(\|\mathbf{p}-\mathbf{q}\|)}_{\text{not new}} \underbrace{G_{\sigma_r}(|I_{\mathbf{p}}-I_{\mathbf{q}}|)}_{\text{new}} I_{\mathbf{q}}$$

normalization factor

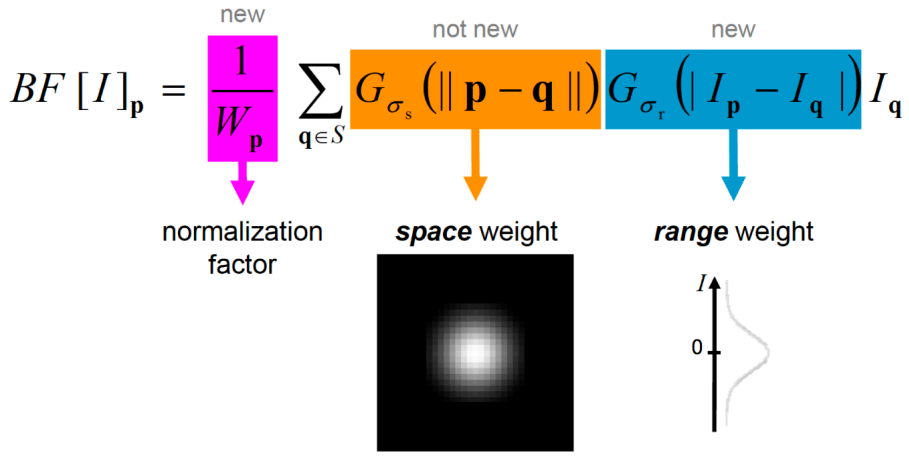
space weight



range weight



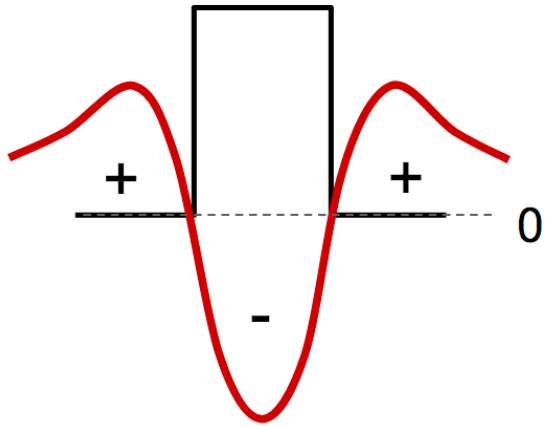
Bilateral Filtering



- If intensities in small patch all the same (i.e not an edge), this new Gaussian factor is high and it is considered for smoothing
- If large intensity change (i.e. filter centered at edge and currently considering non-edge pixel), this Gaussian factor is low and non-edge pixel is ignored in smoothing -> edges not smoothed

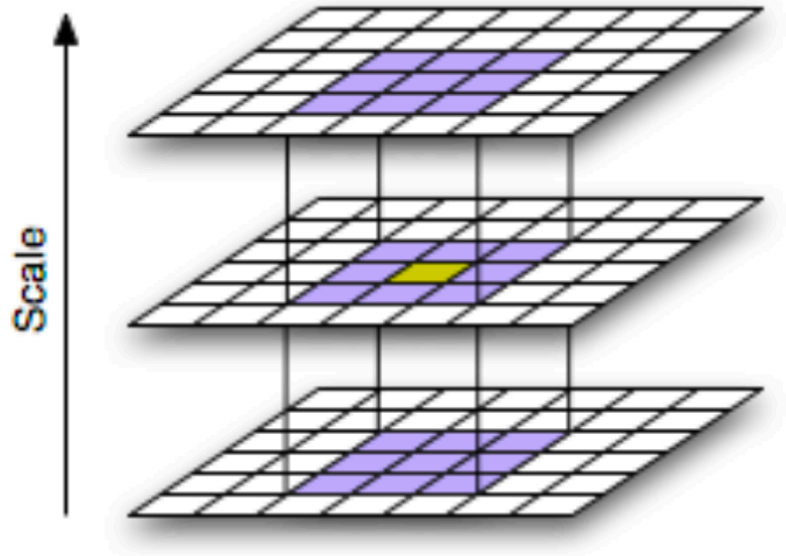
Blob Detection

- Consider the problem of detecting circular blob of radius r
- To detect a blob of radius r, we use Laplacian filter with scale $\sigma = r / \sqrt{2}$
- See which location leads to largest magnitude response
- Why this scale? when we slide Laplacian over image, we need negative part of Laplacian completely aligned with blob to produce largest magnitude response from convolution



Blob Detection

- However, we don’t know blob size ahead of time
- Convolve with normalized Laplacians of various scale and see which locations produce largest magnitude response, then scale of blob can be found from scale of corresponding Laplacian



Blob Detection

- Need to normalize Laplacian since its magnitude decreases with larger scale (g below is Gaussian)

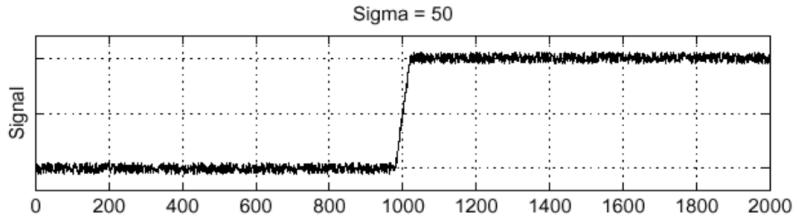
$$\nabla^2_{\text{norm}} g = \sigma^2 \left(\frac{\partial^2 g}{\partial x^2} + \frac{\partial^2 g}{\partial y^2} \right)$$

Edge Detection

- Edge is discontinuity in image intensity; ideal edge has very small width
- To find ideal edge, can find where derivative has maximum magnitude
- Equivalently, can find where second derivative crosses 0

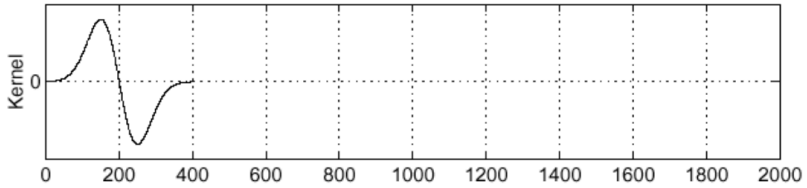
Edge Detection

f



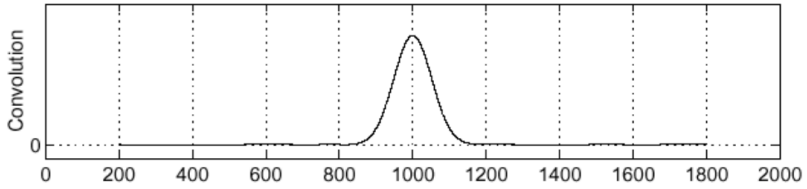
Edge

$\frac{d}{dx} g$



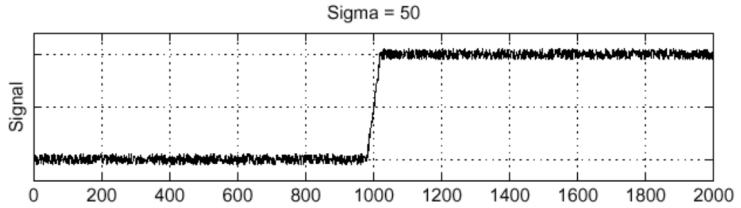
Derivative of Gaussian

$f * \frac{d}{dx} g$



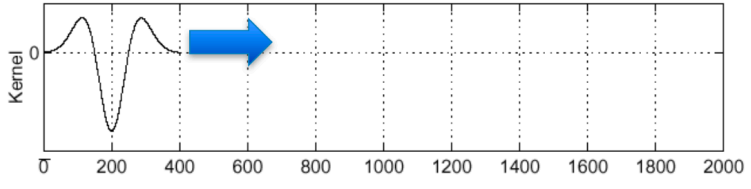
Edge = extrema of derivative

f



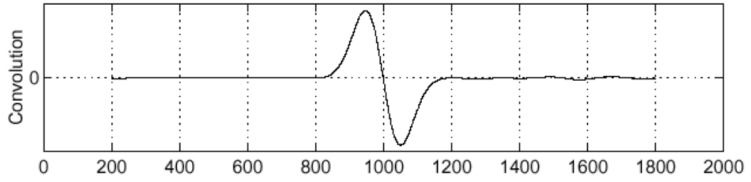
Edge

$\frac{d^2}{dx^2} g$



Second derivative of Gaussian (Laplacian)

$f * \frac{d^2}{dx^2} g$



Edge = zero crossing of second derivative

Why Image Features

- Blob / edge detection can be used to find image features (example: SIFT)
- But why do we need image features?
- We want image features to have certain properties for applications like classification
- Invariance to translations, rotations, scale, illumination
- Small shifts, rotations, scale changes, and intensity changes do not change the overall meaning of an image
- Also would like features to have some sense of spatial arrangement (more on this in a bit)

Examples of Bad Image Features

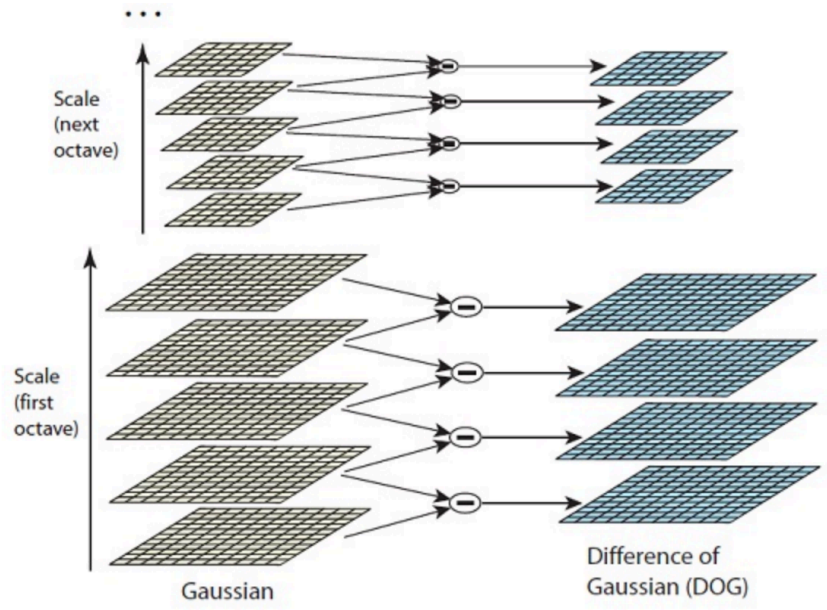
- Using downsampled image as image feature is bad idea as downsampled image has none of these invariances
- Another attempt at features: use the edges; edges are not invariant to scale or rotation
- Yet another attempt: histogram of colors in image patch; this loses spatial arrangement; we can rearrange the pixels to get different semantic meaning but same features
- Fourth attempt: combine histograms over various small patches in a big image patch (spatial histogram), but not invariant to rotation anymore -> correct for orientation to get rotation invariance

SIFT Features

- Good set of image features (finally): SIFT (scale-invariant feature transform)
- SIFT features possess invariance to translation, scale, rotation, illuminance as well as preserving information about spatial arrangement
- Useful for various tasks such as classification (HW 1), panorama (discussed in later lectures), etc.

SIFT Keypoints

- Extracting SIFT Features:
- Step 1: Apply difference of Gaussians at neighboring scales and find local maxima along with corresponding scale
  - These local maxima along with scale will form the keypoints ((x, y, σ) tuples)



SIFT Keypoints

- Why difference of Gaussians (DoG)? It's an approximation to the Laplacian, useful for finding edges
- Why does local maxima of DoG find edges?
- As a Gaussian filter is a low-pass filter, DoG allows a narrow band of frequencies through
- DoG at certain scale finds edges of certain size; thicker edges are lower frequency spatially and will be found by DoG at larger scale
- DoG is approximation to Laplacian and local maxima of Laplacian finds blobs; real edges have a certain thickness and in a small image patch, they will look like blobs of various sizes, so DoG can find edges

SIFT Keypoint Refinement

- Step 2: Taylor series approximation of DoG
- Use quadratic Taylor series approximation of DoG function to refine the locations and scales of keypoints (which has been found to pixel-level accuracy and some discrete scale)

$$D(\mathbf{x}) = D + \frac{\partial D}{\partial \mathbf{x}}^T \mathbf{x} + \frac{1}{2}\mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}$$

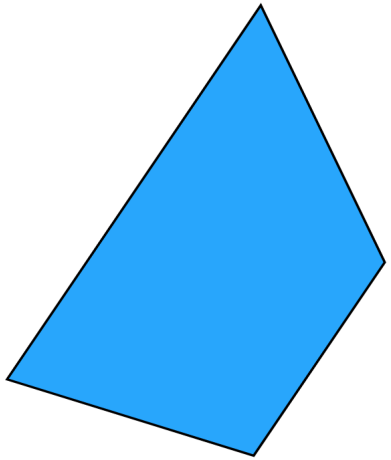
- Take derivative of Taylor series approximation, set to 0, and solve to refine keypoints

SIFT Keypoint Refinement

- Step 2 (continued): remove noisy keypoints and keypoints along edges (these keypoints can be anywhere along edge and have noisy location)
- The keypoints should generally correspond to corners

Why Corners?

- Why do we want to find corners (for both SIFT and later on, Harris corner detector)?
- Corners are distinctive (for example for image matching) even more so than edges (since corners are intersection of multiple edges)
- If you wanted to find the same point in a matching image of the polygon below, which point would you choose?

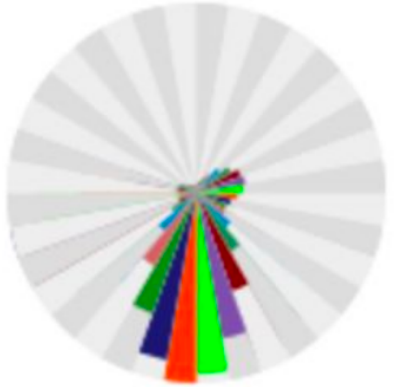


SIFT Keypoint Orientation

- Step 3: find dominant orientation of each keypoint (x, y, σ)
- Take Gaussian-smoothed image L(x, y, σ) (Gaussian-smoothed using Gaussian at closest scale to that of keypoint), and compute gradient magnitude and orientation in small patch around keypoint
- Compute histogram of gradient orientations (weighted by gradient magnitudes so that stronger gradients are more important) -> get keypoint (x, y, σ, θ)

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}$$

$$\theta(x,y) = \operatorname{atan2}(L(x,y+1) - L(x,y-1), L(x+1,y) - L(x-1,y))$$

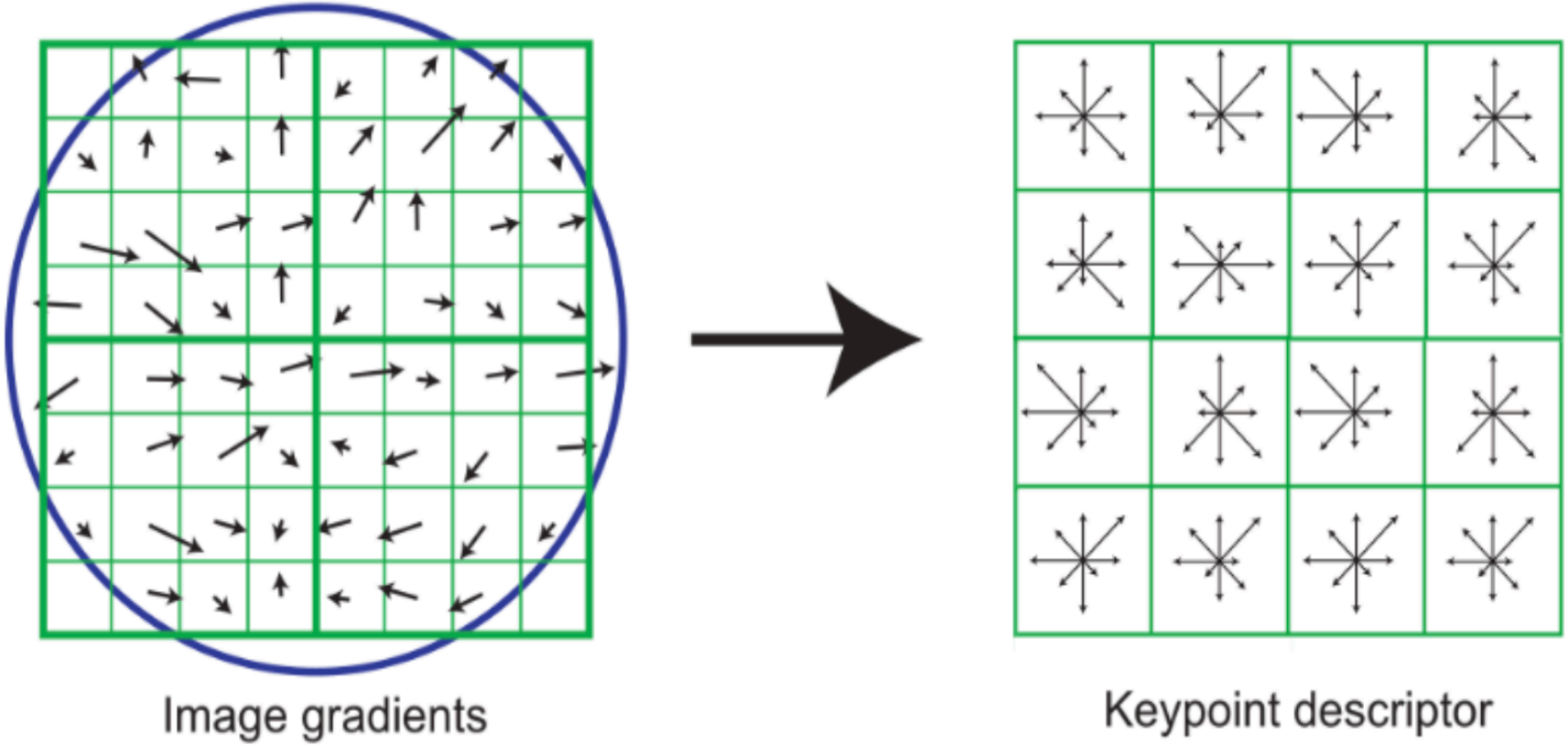


SIFT Descriptors

- Step 4: compute SIFT descriptors, i.e spatial histogram of patch around each keypoint (x, y, σ, θ); these are the final features
- I.e take larger size patch (size 16x16) and compute gradient orientation histograms for smaller patches (size 4x4) in big patch
- These gradient orientation histograms have 8 bins (coarser than in step 3) -> final size of descriptor is (16x16/(4x4))*8=128
- Again, weight orientation histograms by gradient magnitude and normalize histograms
- Normalize orientation using dominant orientation θ found from step 3 (i.e subtract θ from orientation bin edges)

SIFT Descriptors

- Graphically, step 4 looks like this



SIFT Properties

- Does SIFT have the desirable properties we want? Yes
- Invariance to scale: keypoints are found whatever scale they occur at due to DoG pyramid; gradient orientation computation is invariant to scale due to using Gaussian-smoothed image at keypoint scale
- Invariance to translation: keypoints are found at whatever locations they occur at and then we look at patch around keypoint, wherever it is
- Invariance to rotation: keypoints are found independent of orientation; then, by normalizing dominant orientation, SIFT descriptors are invariant to rotation

SIFT Properties

- Invariance to illumination? Yes, as gradient orientation histograms are normalized at the end so uniform intensity scaling will be normalized out; gradient is invariant to constant intensity shift
- Keeps sense of spatial arrangement? Yes as we use multiple histograms of smaller patches around keypoint so info about arrangement of smaller patches preserved

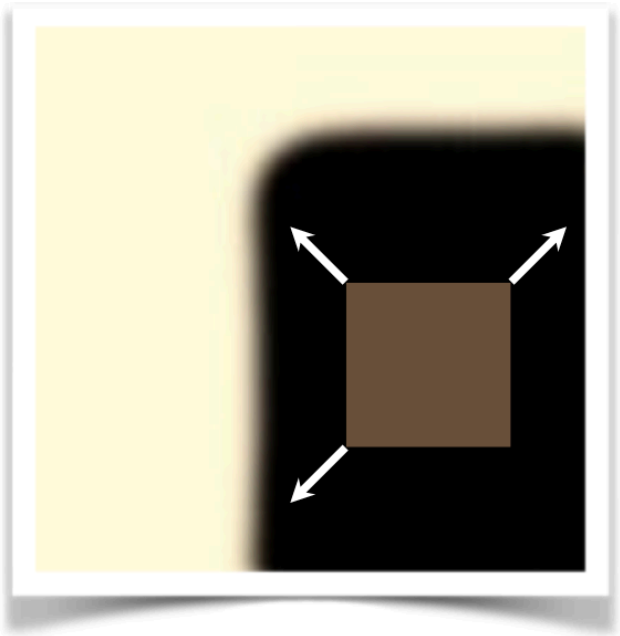
Alternative Set of Features

- Can detect corners using Harris corner detector combined with multi-scale detection (to find corners of different scale)
- Then, for each of these keypoints, extract MOPS (multi-scale oriented patches) features
- MOPS is an alternative set of image features

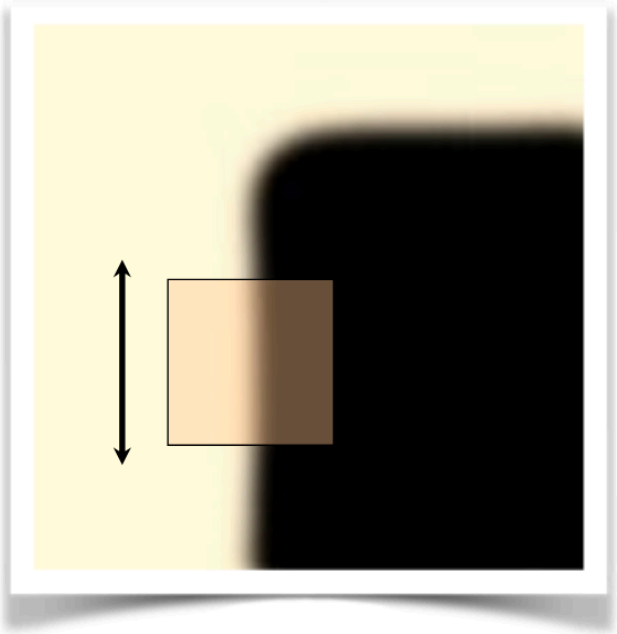
Harris Corner Detector

- Another way to find corners, which are then used as keypoints
- Intuition: at a corner, computing the image gradient at the corner vs using a slightly shifted window should result in big difference in gradient
- Importantly, this should hold for shifting the window in multiple directions
- For an edge, shifting the window only changes gradient significantly for certain shift directions

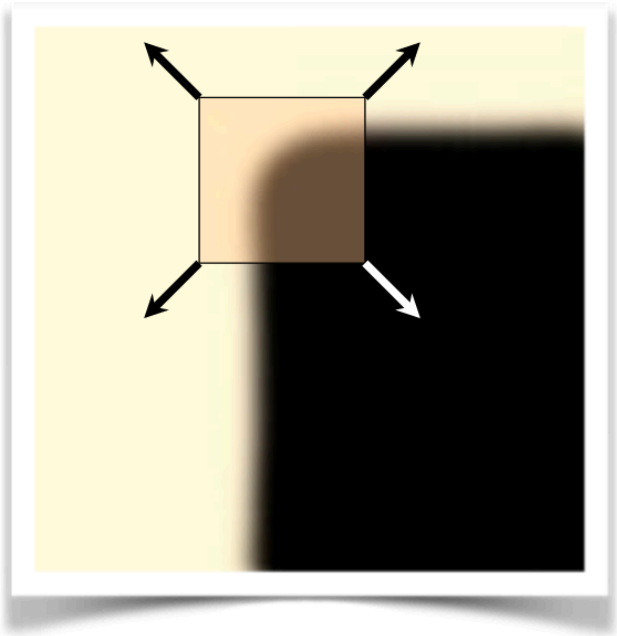
Intuition for Harris Corner Detector



"flat" region:
no change in all directions



"edge":
no change along the edge direction

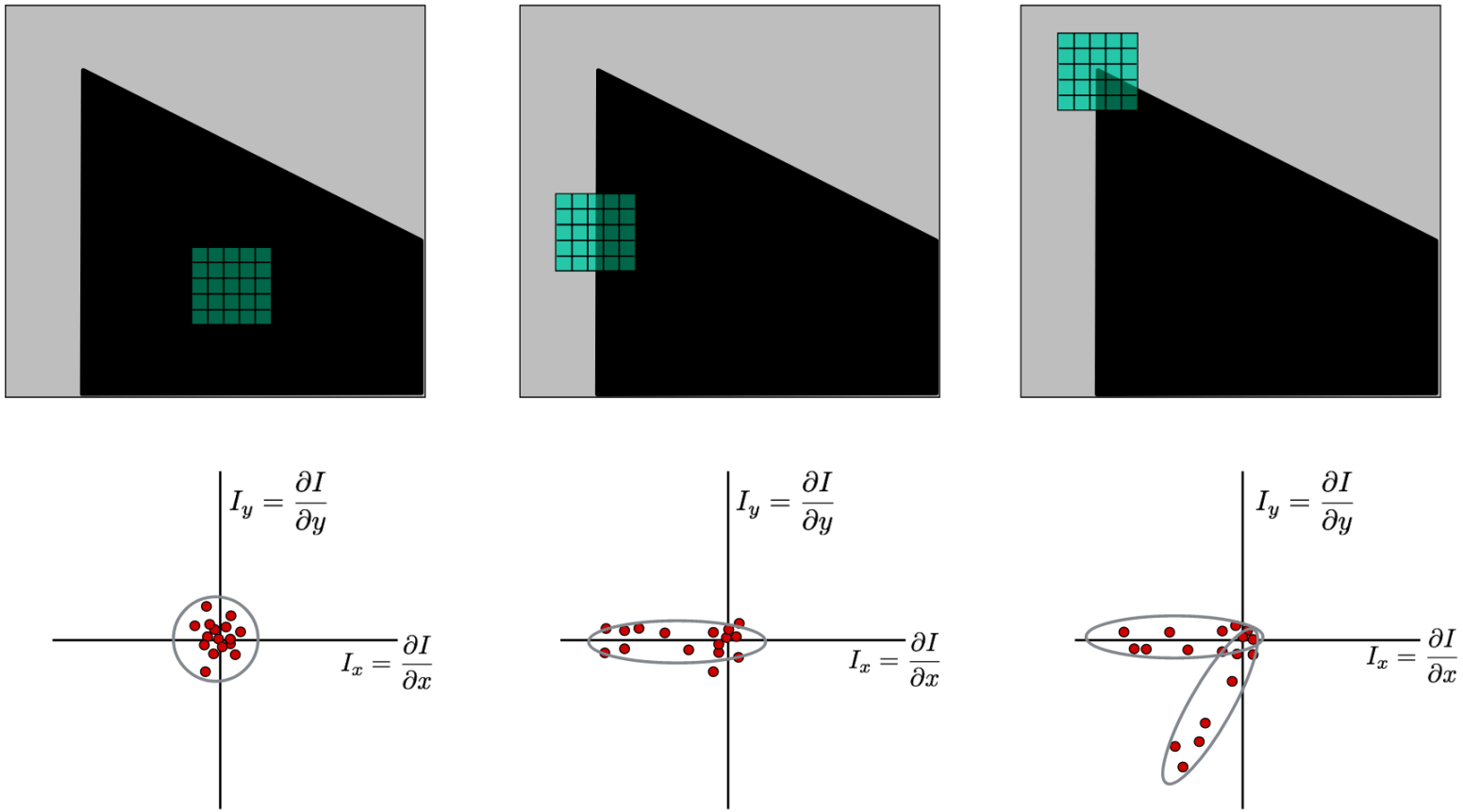


"corner":
significant change in all directions

Steps for Harris Corner Detector

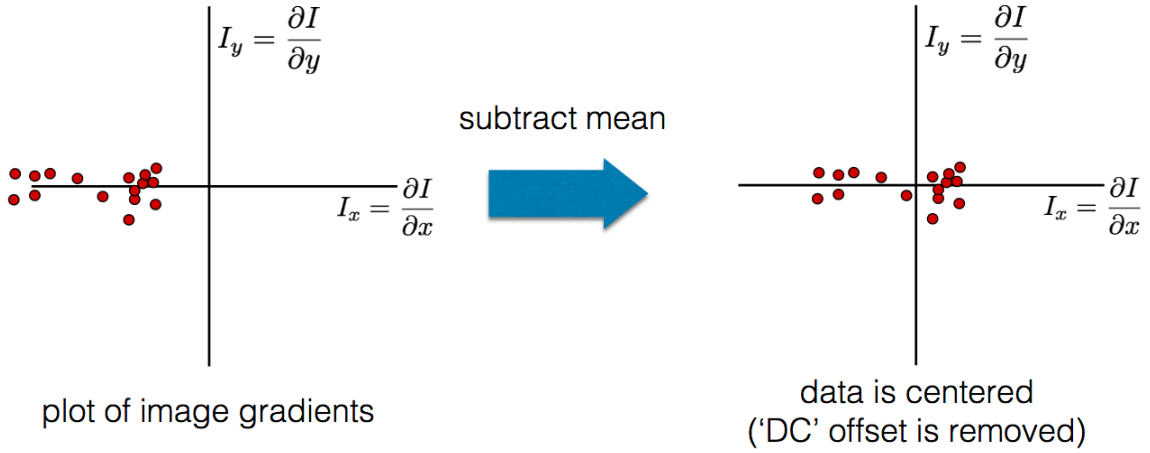
- Step 1: Compute image gradients in small window around every point
- Why small window? We want distribution of gradients as we shift away from hypothetical corner
- For corner, we should see strong gradients in multiple directions

Steps for Harris Corner Detector



Steps for Harris Corner Detector

- Step 2: subtract mean of gradients found from all the gradients
- Why? We are interested in the variation of the distribution of the gradients as we care about the change in the gradient
- Shifting the point we take gradient at should change gradient significantly for multiple shift directions for a corner



Steps for Harris Corner Detector

- Step 3: compute covariance matrix of gradients
- We want to know variation in distribution of gradients so compute covariance

$$\begin{bmatrix} \sum_{p \in P} I_x I_x & \sum_{p \in P} I_x I_y \\ \sum_{p \in P} I_y I_x & \sum_{p \in P} I_y I_y \end{bmatrix}$$

$$\sum_{p \in P} I_x I_y = \text{sum}(\quad .* \quad)$$

$I_x = \frac{\partial I}{\partial x}$ array of x gradients

$I_y = \frac{\partial I}{\partial y}$ array of y gradients

Harris Error Function

- Another way to motivate using covariance matrix M (where M is weighted covariance here):
- Error function for intensity change with shift direction

$$E(u,v) = \sum_{x,y} w(x,y)\left[I(x+u,y+v) - I(x,y)\right]^2$$

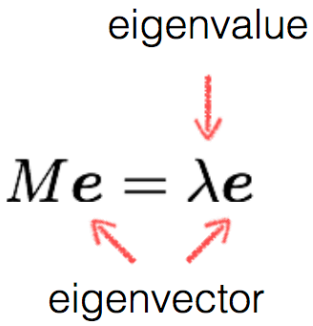
- Approximation for small shifts:

$$E(u,v) \cong \begin{bmatrix} u, v \end{bmatrix} \; M \; \begin{bmatrix} u \\ v \end{bmatrix}$$

Steps for Harris Corner Detector

- Step 4: compute eigenvalues and eigenvectors of covariance matrix
- Why eigenvalues and eigenvectors? Eigenvectors of covariance matrix represent principal directions of variance (eigenvectors are orthogonal) and eigenvalues represent how strong variance is in the principal directions

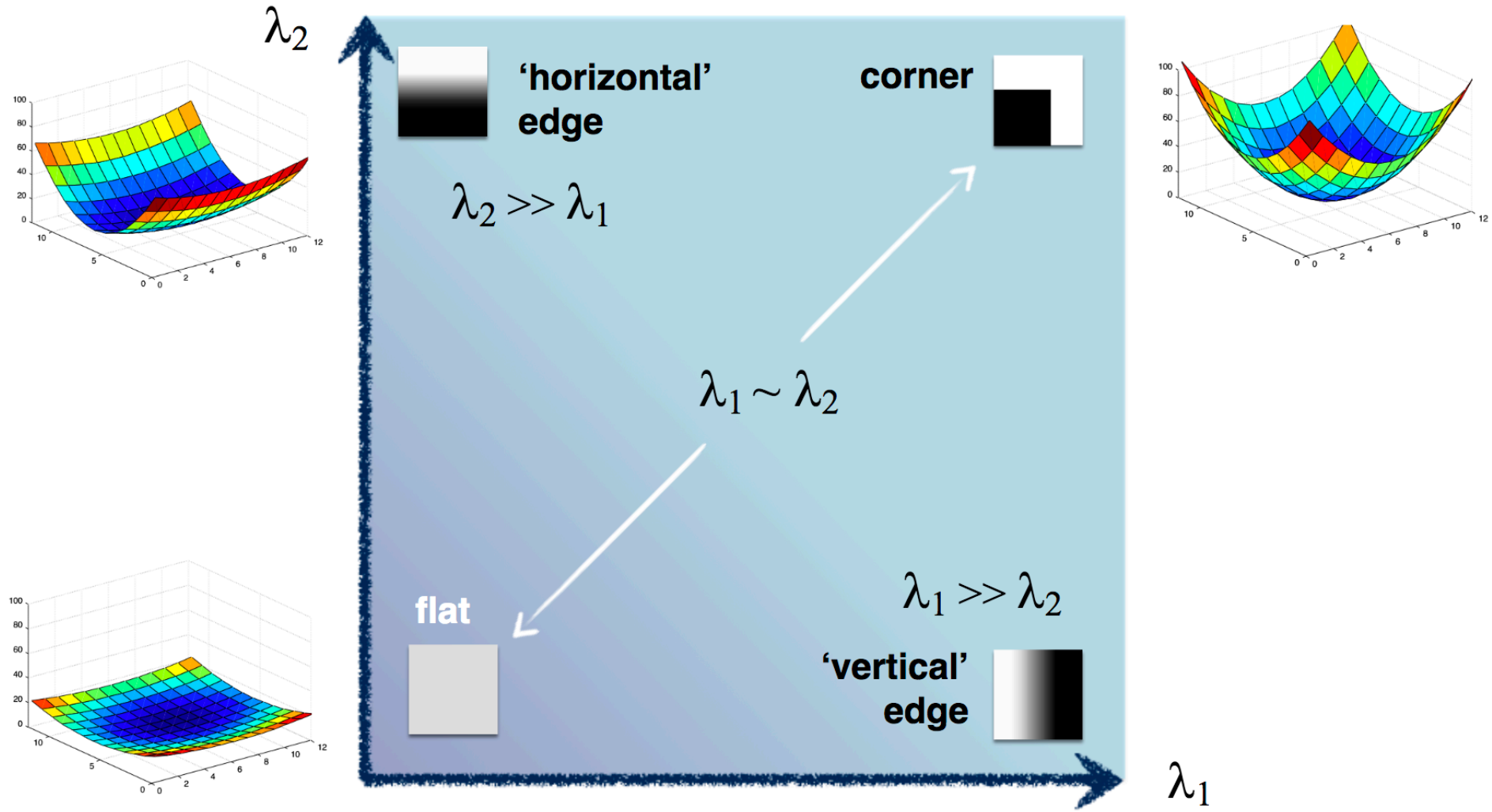
Steps for Harris Corner Detector



$$(M - \lambda I)e = 0$$

1. Compute the determinant of $M - \lambda I$
   (returns a polynomial)
2. Find the roots of polynomial $\det(M - \lambda I) = 0$
   (returns eigenvalues)
3. For each eigenvalue, solve $(M - \lambda I)e = 0$
   (returns eigenvectors)

Steps for Harris Corner Detector



Steps for Harris Corner Detector

- Step 5: compute function of eigenvalues (Harris response function) and threshold to detect corners
- Strong corners should have strong shift directions, i.e directions where gradient changes a lot due to small shift -> strong eigenvalues

Harris & Stephens (1988)

$$R = \det(M) - \kappa \text{trace}^2(M)$$

Kanade & Tomasi (1994)

$$R = \min(\lambda_1, \lambda_2)$$

Nobel (1998)

$$R = \frac{\det(M)}{\text{trace}(M) + \epsilon}$$

$$\det M = \lambda_1 \lambda_2$$

$$\text{trace } M = \lambda_1 + \lambda_2$$

$$det\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc$$

$$trace\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = a + d$$

Harris Corner Detector Properties

- Harris corner detector invariant to rotation: eigenvectors of covariance matrix change; eigenvalues (used for Harris response function) do not
- Invariant to intensity shifts (only derivatives of intensity are used) and intensity scaling by small factors (as we threshold function of eigenvalues, as long as intensity scaling factor around 1, no new eigenvalues will meet/no longer meet threshold)

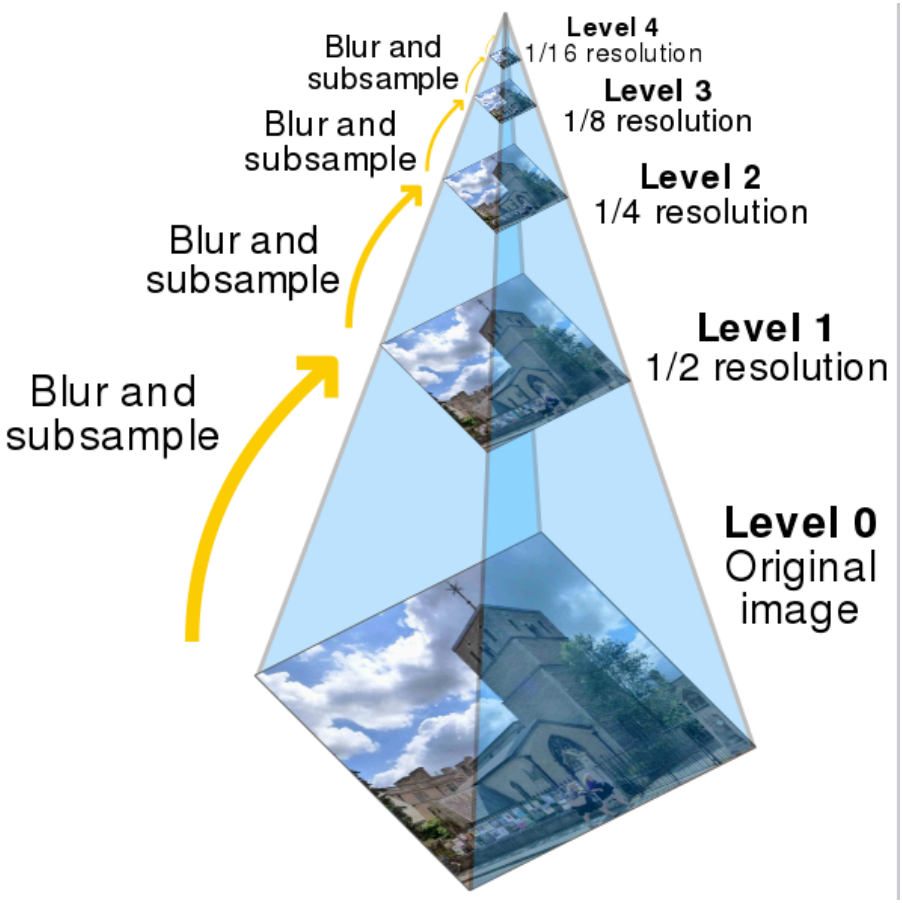
Harris Corner Detector Properties

- Not invariant to scale: since Harris corner detector computes distribution of gradients in fixed patch, scaling can change overall gradient distribution

Multi-scale + Harris detector

- To detect corners at multiple scales, apply multi-scale detection
- This can be done in several ways:
- 1) downsample image with several different sizes and apply feature extractor (in this case, Harris detector)
- 2) apply feature extractor (Harris detector here) to image smoothed with Gaussians of different scale
- Save location and scale that produces greatest response (for example local maxima across scale and location)

Multi-scale + Harris detector



MOPS

- MOPS (multi-scale oriented patches)
- Uses Harris corner detector + scale pyramid to detect keypoints (x, y, s), where s is the scale
- Need descriptors so features more discriminative

MOPS Steps

- Step 1: given feature (x, y, s, θ), get 40 x 40 image patch and subsample every fifth pixel to get 8 x 8 patch
- Note: patch is extracted from pyramid with scale s to obtain scale invariance and oriented using θ to obtain rotation invariance
- How is orientation θ computed? Smoothed gradient around keypoint (similar in concept to Gaussian + gradient)
- We use image patch to get context around keypoint
- Subsample to reduce effect of localization errors (subsampled patch varies less if keypoint off by a pixel, removes some noise)

MOPS Steps

- Step 2: Normalize image patch by subtracting mean and dividing by standard deviation
- This leads to invariance to intensity changes (intensity shifts and scaling)
- Step 3: Compute Haar wavelet transform of normalized image patch
- This step computes responses when applying low-frequency filters to patch
- Haar wavelet is a way to get low frequency info -> we have seen that components at various frequencies can be indicative of useful features like edges and corners at various scales

Classification Pipeline

- A brief review of classification pipeline from HW 1
- Step 1: extract SIFT/SURF/ORB descriptors from all images in training set; these describe image patches
- Step 2: cluster descriptors from step 1 using Kmeans or agglomerative clustering and obtain vocabulary (cluster centers); this step summarizes the many descriptors into a few cluster centers
- Step 3: get BoW representation for all images in training set by finding for each image, how many descriptors are closest to each cluster center, and then normalizing the resulting histogram

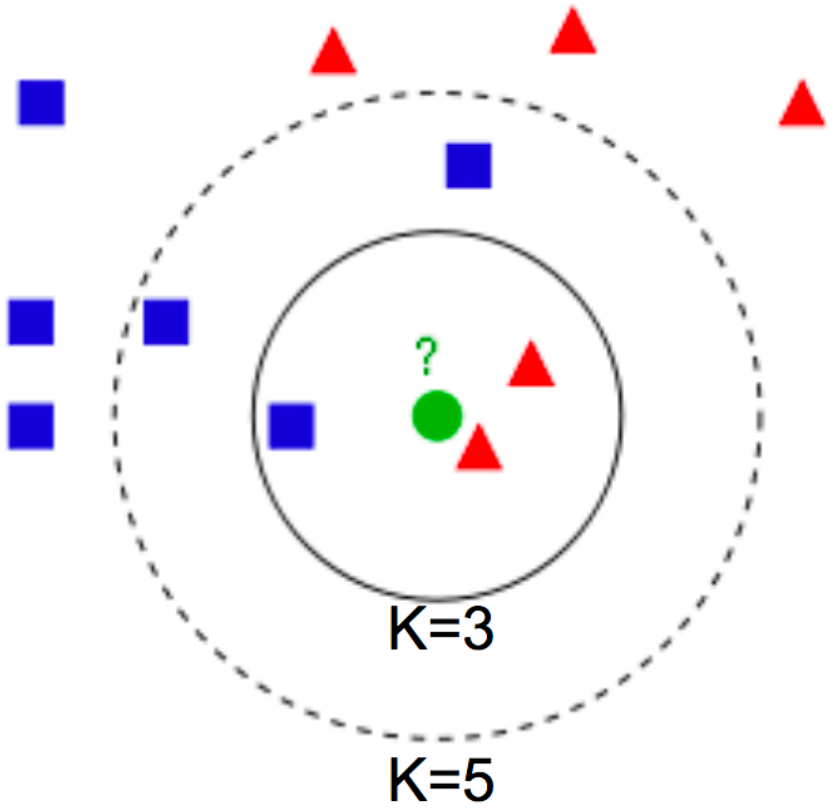
Classification Pipeline

- Step 4: train classifiers to classify images in training set
  - In case of KNN, can just use one classifier for all labels
  - In case of SVM, train one classifier for each class to classify whether each image belongs to class or not; take class whose SVM is most confident that the image belongs to respective class
- Step 5: to classify a test image, extract BoW for test image (follow steps 1 and 3 for test image using vocabulary from step 2) and run classifier from step 4

K-Nearest Neighbor

- So how do the ML algorithms (K-nearest neighbor, Kmeans, etc.) in the classification pipeline work? Start from K-Nearest neighbor
- K-Nearest Neighbor is a simple classifier
- For a point in test set, take K nearest neighbors in training set (using some distance metric like L2) and majority vote based on the labels of the nearest neighbors
- Can also weight the votes based on distance

K-Nearest Neighbor



K-Means Clustering

- Goal of K-means clustering: find K clusters with minimal distances between points in a cluster and cluster center
- We want cluster center to be close to all points in its cluster
- Step 1: Initialize cluster centers randomly
- Step 2: alternate below 2 steps repeatedly:
  - Step 2(a): Set cluster centers to mean of points in their respective clusters; this minimizes distance between cluster center and points currently in cluster
  - Step 2(b): Assign each point to cluster whose center is closest; this minimizes distance between the points to be clustered and the current cluster centers

K-Means Clustering

1. Initialize **cluster centroids** $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$ randomly.
2. Repeat until convergence: {

   For every i, set

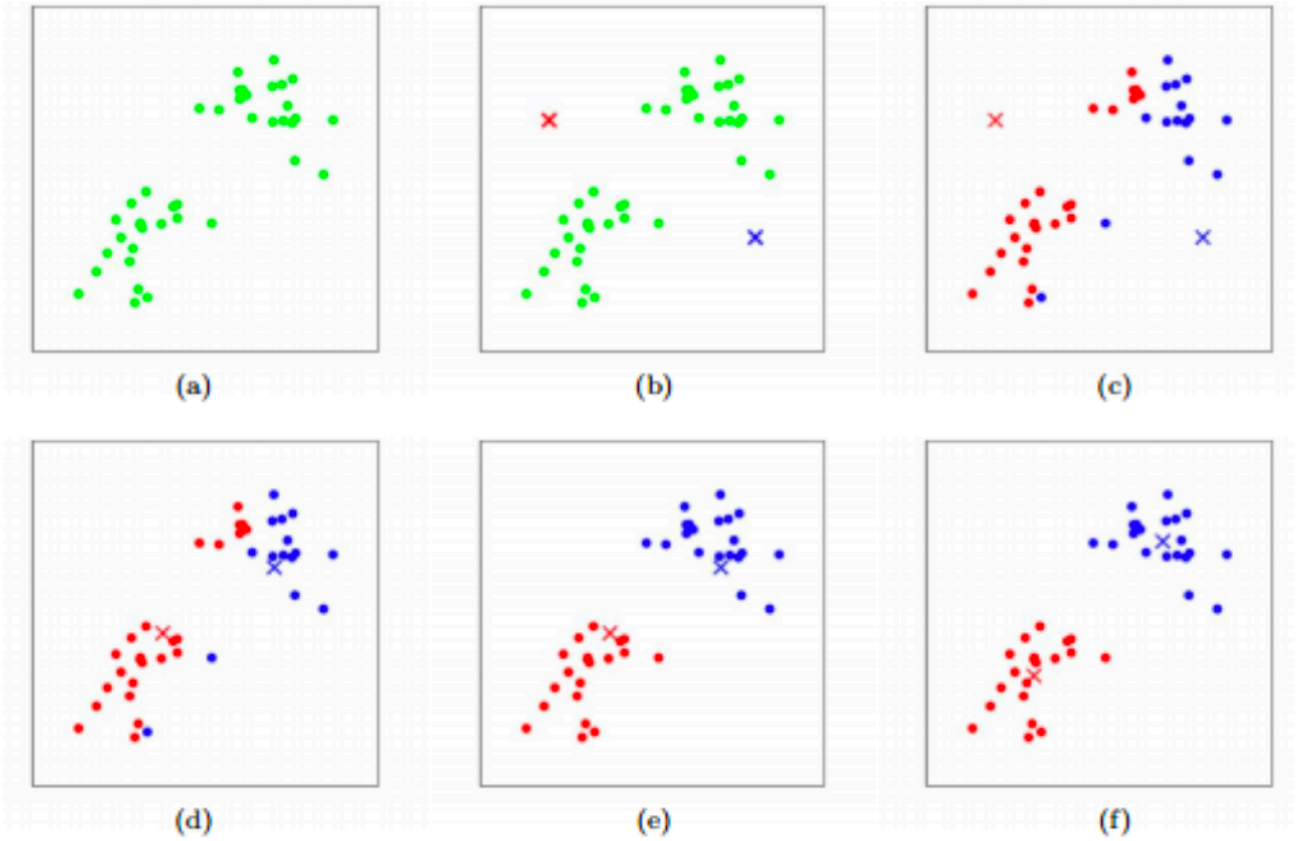
   $$c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2.$$

   For each j, set

   $$\mu_j := \frac{\sum_{i=1}^{m} 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^{m} 1\{c^{(i)} = j\}}.$$

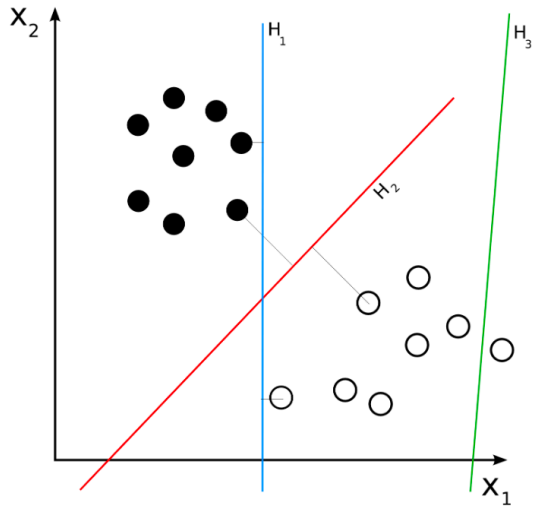
   }

K-Means Clustering



SVM

- SVM learns a linear classifier
- Can be viewed as maximizing margin between points in training set and classification hyperplane or as minimizing hinge loss
- Weight on penalty term controls misclassification penalty
- Kernel allows for SVM to use nonlinear features



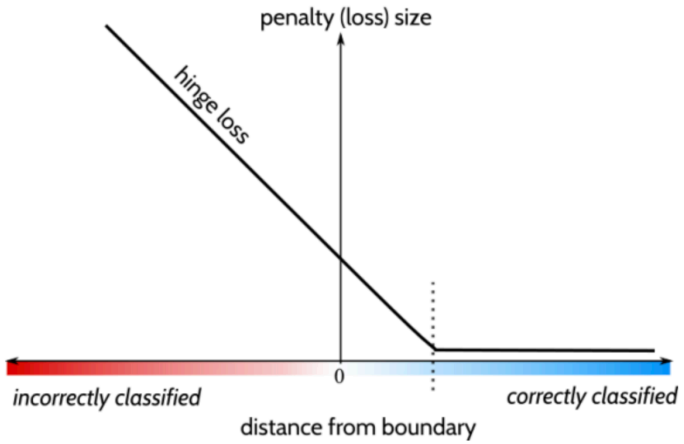


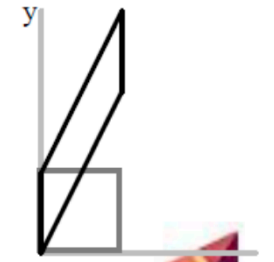
Image Transformations

- Various ways we can transform an image
- Translation: shift image up, down, left, right
- Rotation: rotate image by an angle
- Scaling: make image smaller or bigger
- Flipping: flip image around axis

Image Transformations

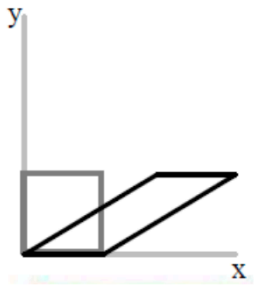
- Shearing: hard to describe, see picture

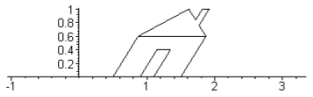
$$\begin{bmatrix} x' = x \\ y' = y + sh_y \cdot x \end{bmatrix}$$



along Y-axis

$$\begin{bmatrix} x' = x + sh_x \cdot y \\ y' = y \end{bmatrix}$$





along X-axis

Image Transformations

- Most of these transformations are linear transformations; however, translation is not, making it hard to represent with matrix multiplication

- Rotation

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad \begin{aligned} x' &= x\cos(\theta) - y\sin(\theta) \\ y' &= y\cos(\theta) + x\sin(\theta) \end{aligned}$$

- Scaling

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

- Shearing

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & sh_x \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad \begin{bmatrix} x' = x + sh_x \cdot y \\ y' = y \end{bmatrix}$$

- Translation

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

Homogeneous Coordinates

- To represent all these transformations as matrix multiplication, use homogeneous coordinates
- Homogeneous coordinates: add a coordinate fixed to be 1 to (x, y) coordinates of point

$$(x, y) \Rightarrow \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Image Transformations Rewritten

Translation

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

Scaling

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Rotation

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Shearing

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Affine Transformations

- Space of homoegeneous coordinates forms an affine space, which is similar to a vector space but shifted (so it generally does not necessarily contain the origin and is generally not closed under linear operations)
- The image transformations rewritten are affine transformations, which act linearly on the vectors (differences) between points in homogeneous coordinates
- Affine transformations can be written as linear transformation combined with translation; hence, they preserve straightness of lines and parallelness of lines (i.e straight lines are still straight and parallel lines are still parallel after affine transformations)
- In general, image transformations are not linear transformations (due to translation)

Applying Multiple Transformations

- Apply the matrix multiplications corresponding to transformations in opposite order of the transformations

When we write transformations using standard math notation, the closest transformation to the point is applied first:

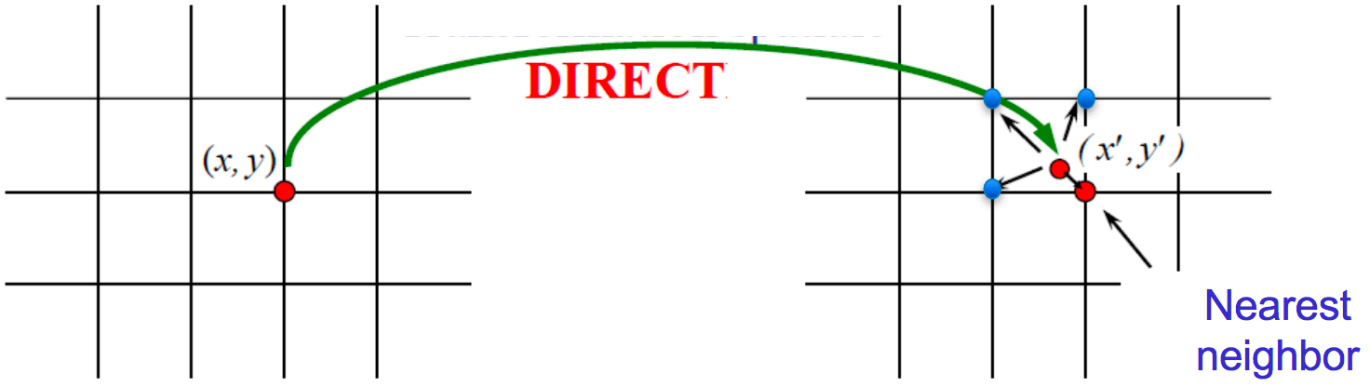
$$T\ R\ S\ p = T(R(Sp))$$

first, the object is scaled,
then rotated,
then translated

Issues with Applying Transformations

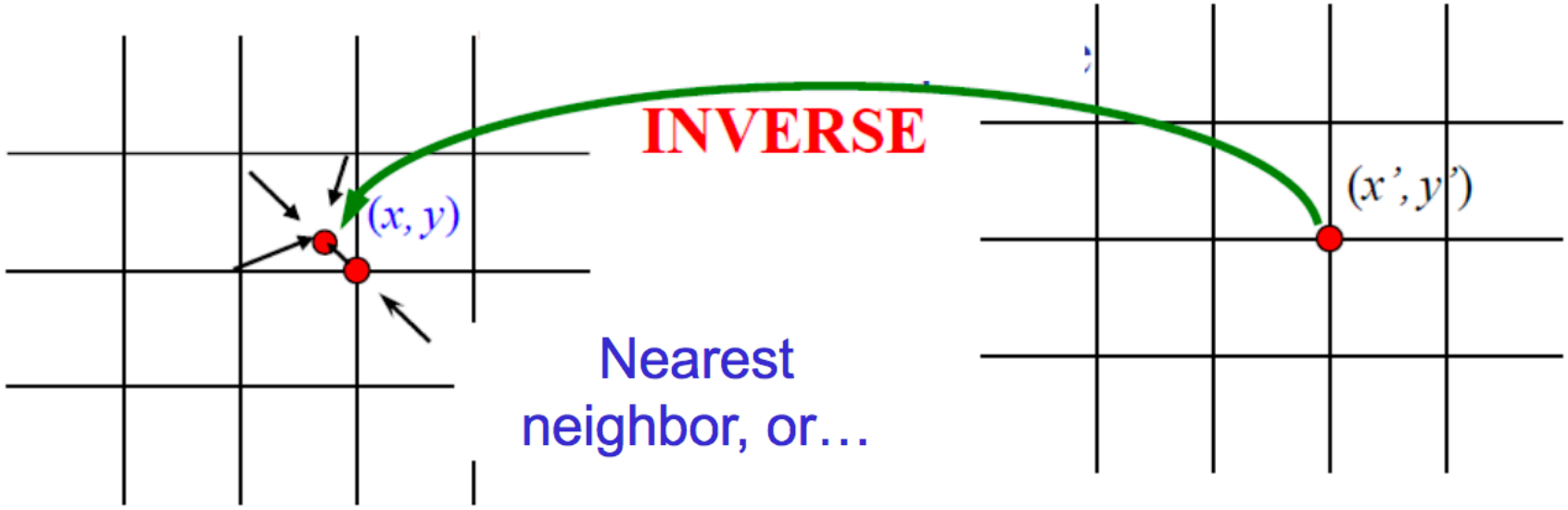
- If we apply image transformations directly, we will have to interpolate to find pixel values of output image at integer pixel locations
- This leads to holes and multiple values for some pixels and is bad

How not to do it



Correct Way to Apply Transformations

- Better way to apply image transformation is to take output image apply the inverse transformation, and then interpolate in input image to get pixel values for output image



Inverse Transformations

- Inverse transformations fortunately are just as easy to compute; just use different parameters
- I.e rotate by –θ instead of θ

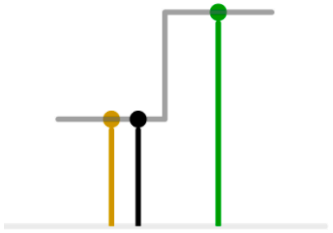
- Translation: $t_x, t_y \rightarrow -t_x, -t_y$
- Rotation: $\theta \rightarrow -\theta$
- Scaling: $s_x, s_y \rightarrow 1/s_x, 1/s_y$
- Shearing: $sh_x, sh_y \rightarrow -sh_x, -sh_y$

Interpolation

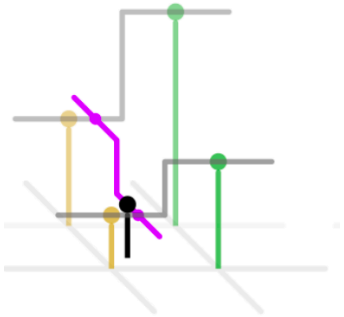
- However, we still have to interpolate in source image to apply the transformation
- Image interpolation is also useful for resizing an image (what pixel values should downsampled image have?)
- Various ways to interpolate (in increasing order of complexity and accuracy): nearest neighbor, bilinear interpolation, bicubic interpolation

Nearest Neighbor Interpolation

- For nearest neighbor interpolation, just select pixel value of nearest neighbor pixel value



1D nearest-neighbour

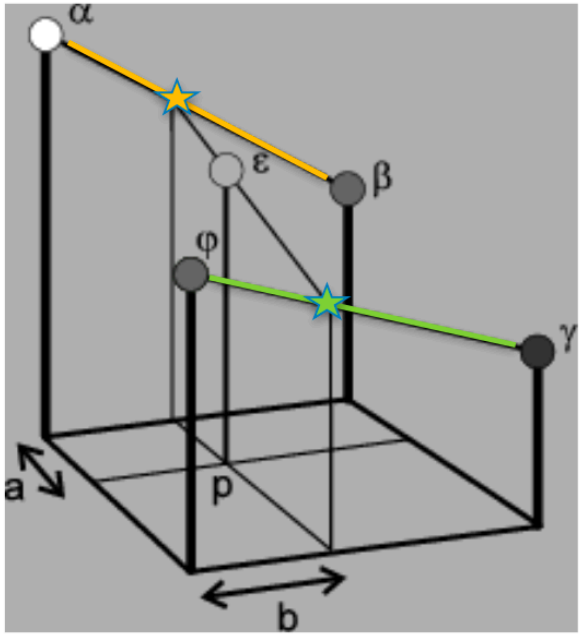


2D nearest-neighbour

Bilinear Interpolation

- We want to find interpolated value I(x', y'), where both x' and y' may be non-integers
- Solve this problem by first interpolating in x twice and then interpolate in y using the interpolated points found before
- Bilinear interpolation requires four neighboring pixel values (as seen in picture in next slide)

Bilinear Interpolation

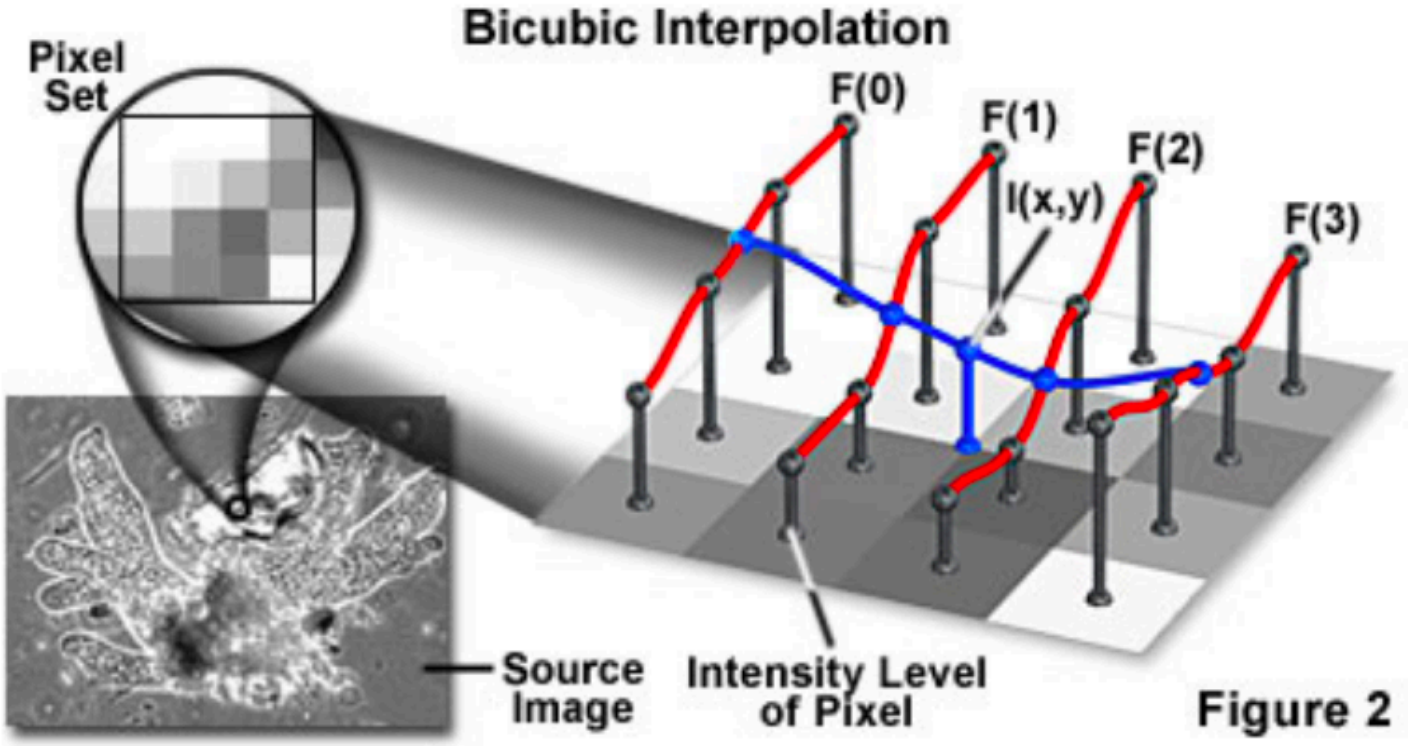


$$\varepsilon = a(b\gamma + (1-b)\varphi) + (1-a)(b\beta + (1-b)\alpha)$$

Bicubic Interpolation

- More accurate to use cubic interpolations instead of linear interpolations
- For bicubic interpolation, perform four cubic interpolations in x direction and then use these interpolated points to perform cubic interpolation in y direction
- Need 16 neighboring pixel values (4 for each cubic interpolation and we have four cubic interpolations; the last cubic interpolation uses interpolated points)

Bicubic Interpolation

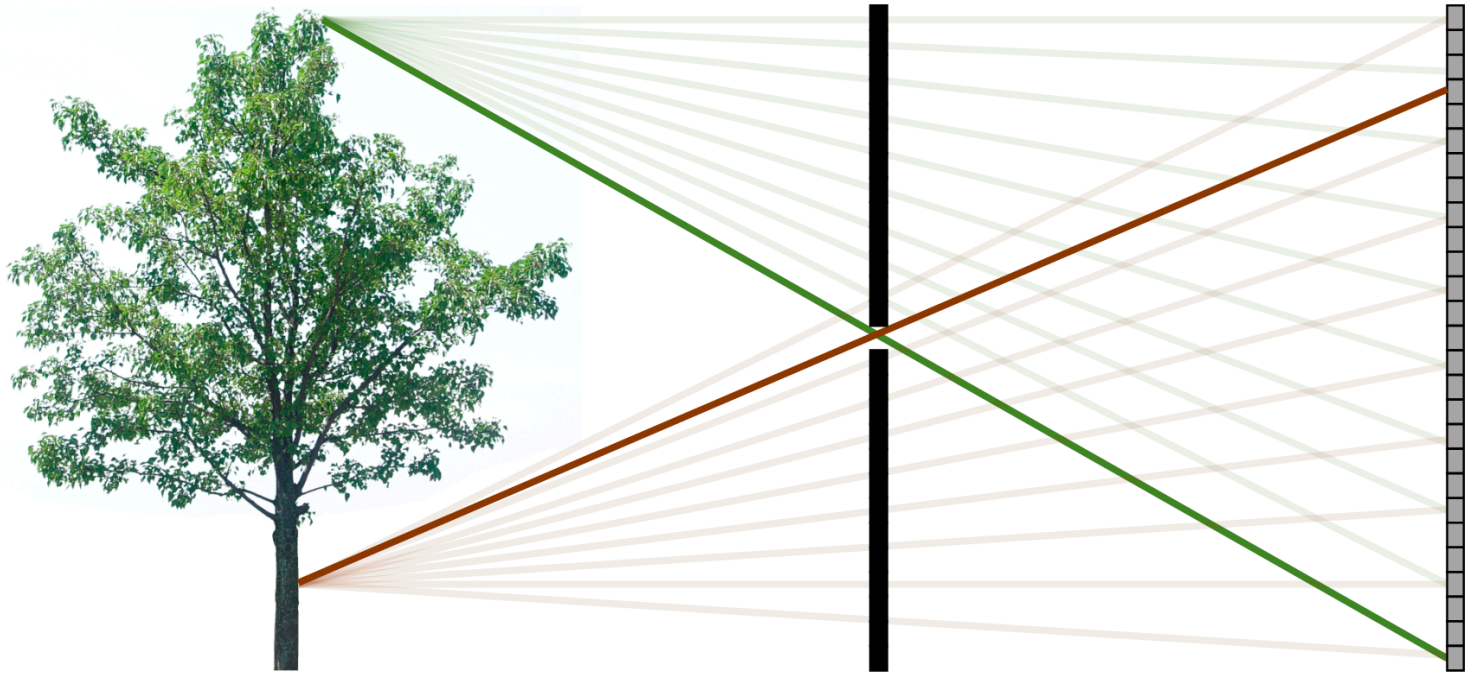


Pinhole Camera Model

- How is an image produced from a scene?
- There are multiple models (geometric and photometric); we will start with the geometric pinhole camera model
- For pinhole camera model, assume we have an infinitely small pinhole in center of our camera, so each scene point contributes to only one point in image (only one light ray is let through)
- If we take image plane to be behind the pinhole, we get an upside down image scaled depending on how far image plane is from camera

Pinhole Camera Model

real-world object



digital sensor (CCD or CMOS)

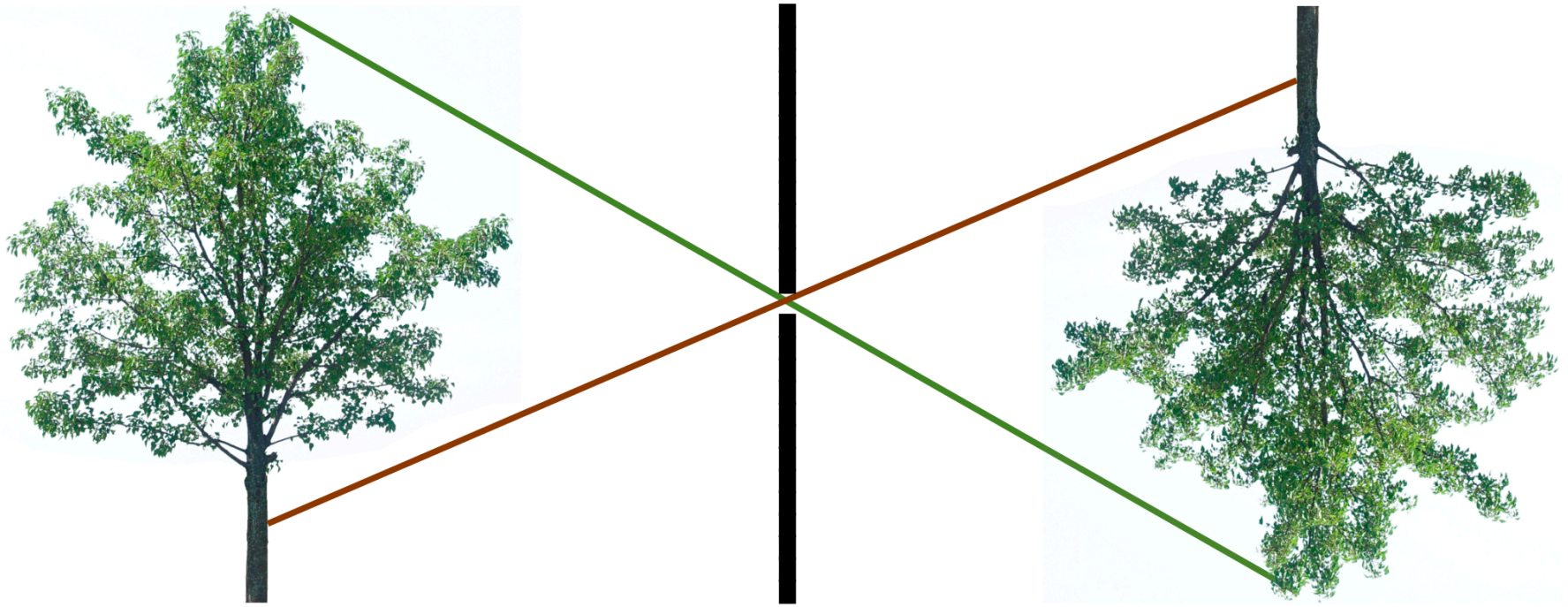
Each scene point contributes to only one sensor pixel

What does the image on the sensor look like?

Pinhole Camera Model

- Object on sensor looks inverted and scaled on the sensor

real-world object



copy of real-world object (inverted and scaled)

Pinhole Camera Model

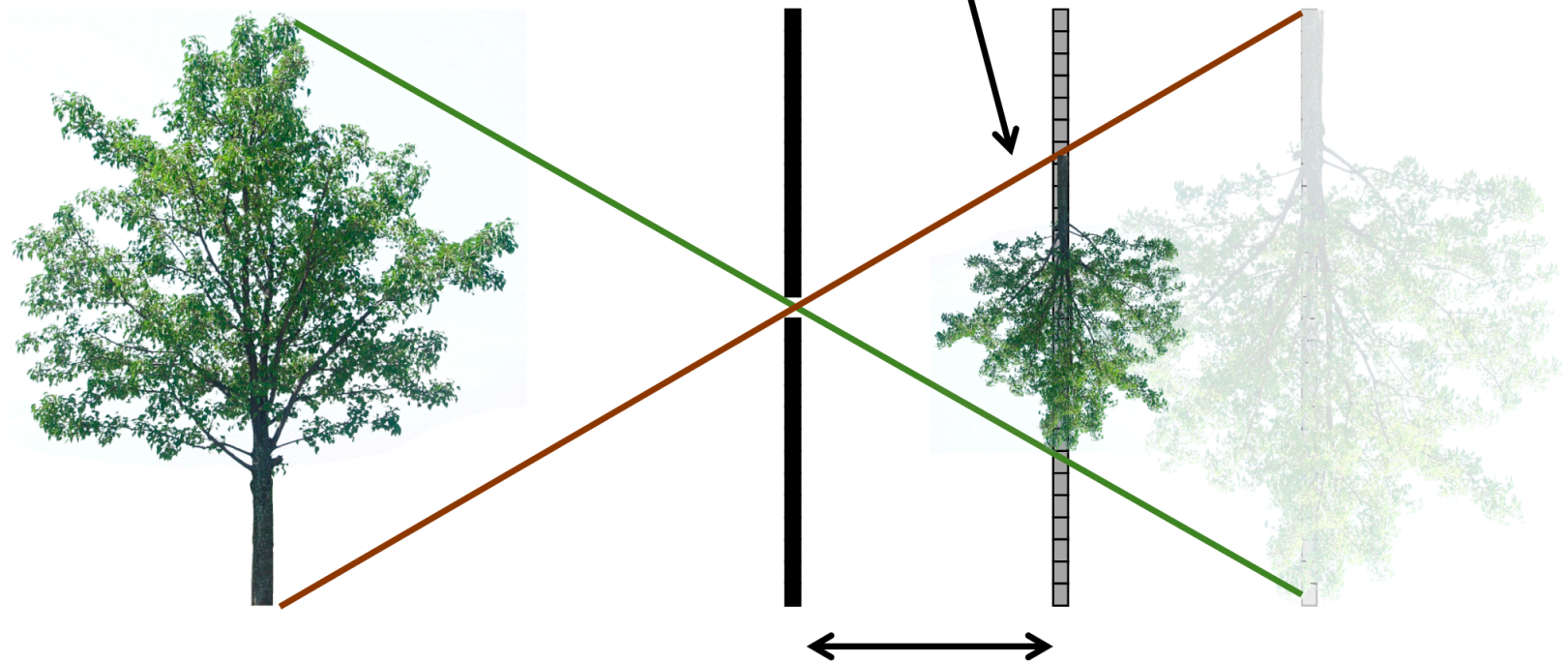
- Focal length of pinhole camera: distance of image sensor to camera pinhole (camera center)
- Changing the focal length by some factor changes the size of the object in the image by the same factor
- I.e. halving the focal length makes the object half as large in the image

Pinhole Camera Model

What happens as we change the focal length?

object projection is half the size

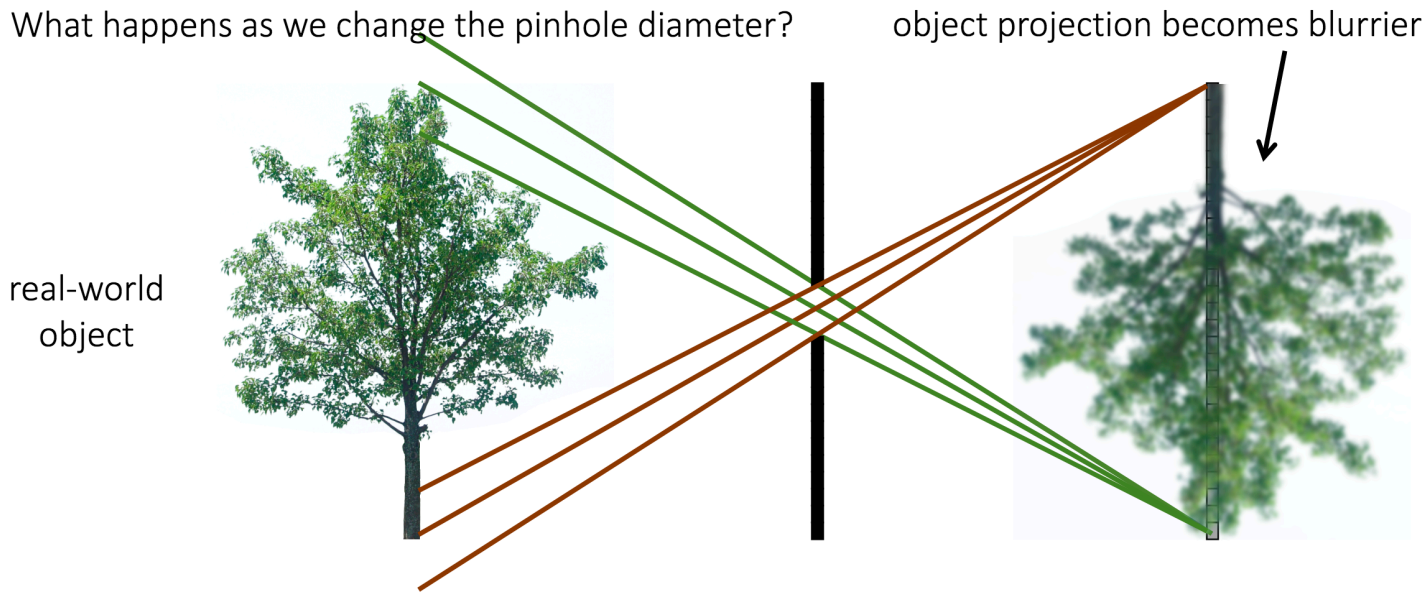
real-world object



focal length 0.5 f

Pinhole Camera Model

- We have assumed pinhole has infinitesimal size
- In practice, this is impossible
- If pinhole diameter increased, object will appear blurrier as light rays from multiple points in scene contribute to a point in the image



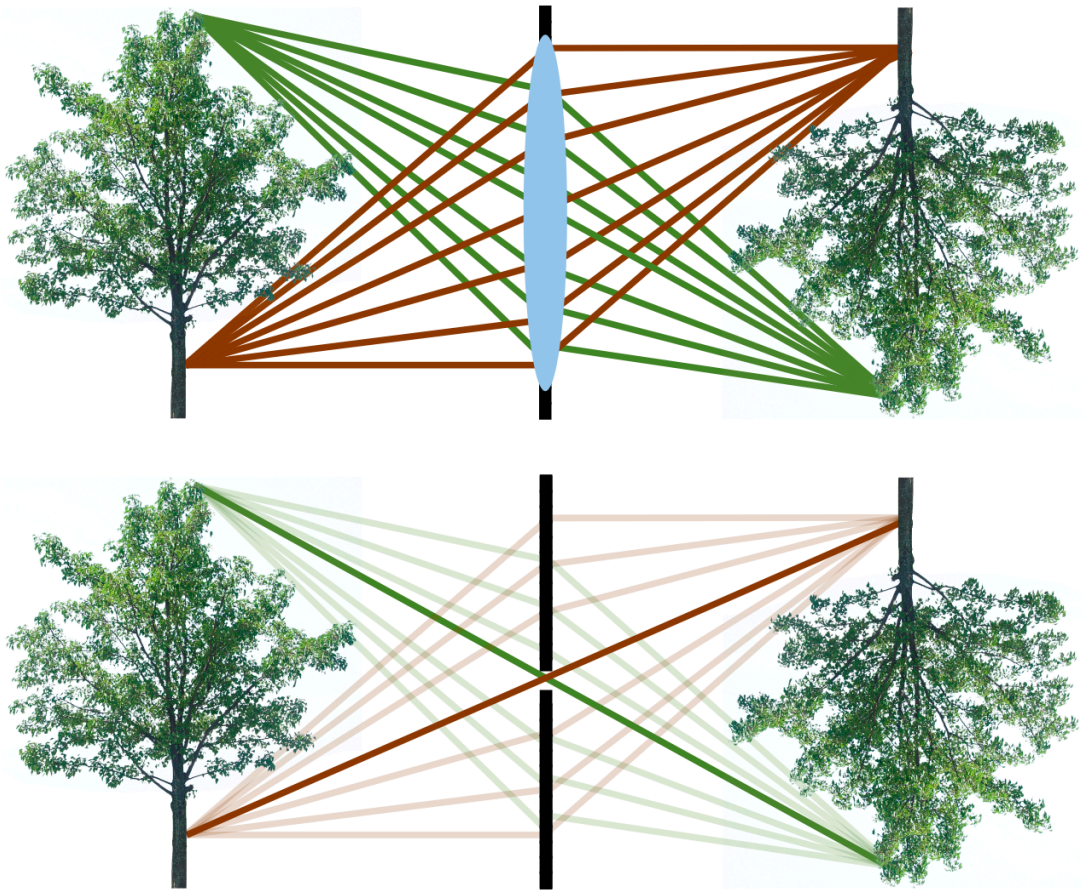
Pinhole Camera Model

- What happens to the light efficiency we change the pinhole diameter or focal length?
- Light efficiency increases as square of pinhole diameter (i.e. 2x pinhole diameter -> 4x light)
- Light efficiency decreases as square of focal length (i.e. 2x focal length -> 0.25x pinhole diameter)

Lens Camera

- In reality, most cameras use a lens not a pinhole
- With a lens, multiple rays of light from scene may converge to same point in image
- Make some assumptions so conclusions hold for both lens and pinhole cameras:
  - Only consider rays that go through center of lens
  - Lens camera is in focus
  - Focal distance (defined a bit later) of lens camera = focal length of pinhole camera
- Hence, after next two slides, we will focus on pinhole cameras

Lens Camera

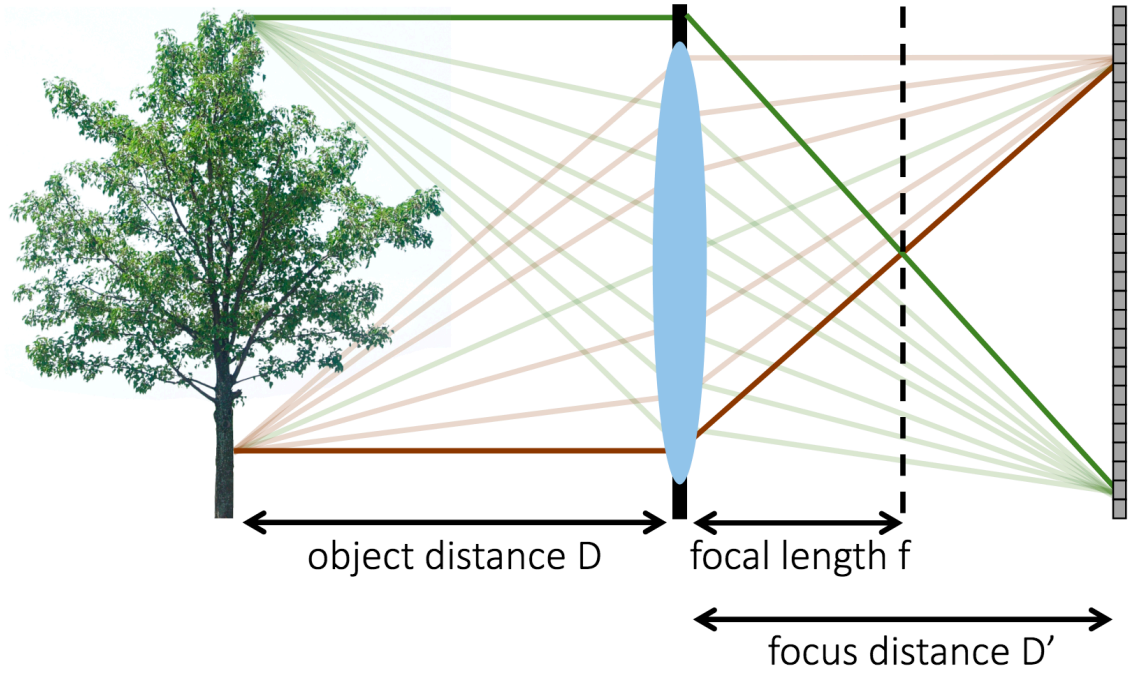


We can derive properties and descriptions that hold for both camera models if:

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

Focal Length and Distance of Lens Camera

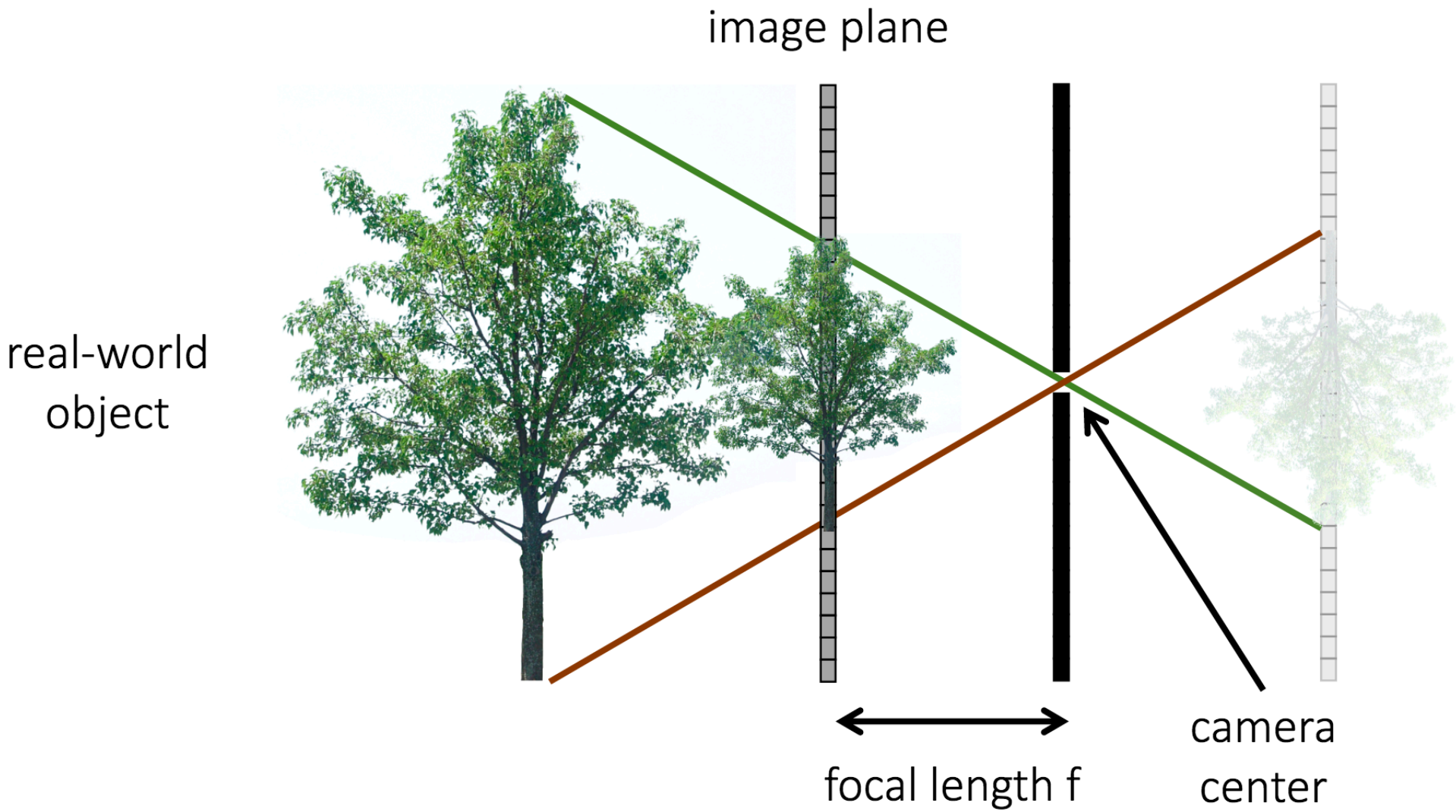
- Focal length is distance where parallel light rays intersect after hitting lens
- Focal distance is distance between lens and camera sensor



Rearranged Pinhole Camera

- Let’s change the location of the image plane so we don’t have to worry about the image being flipped
- Put image plane on same side of pinhole as scene instead of on opposite side
- We get rearranged pinhole camera now

Rearranged Pinhole Camera



Camera Matrix

- Camera matrix (dimensions 3 x 4) transforms points in scene (homogeneous world coordinates, 4 x 1) to points in image (homogeneous image coordinates, 3 x 1)

$$\boldsymbol{x} = \mathbf{P}\mathbf{X}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} p_1 & p_2 & p_3 & p_4 \\ p_5 & p_6 & p_7 & p_8 \\ p_9 & p_{10} & p_{11} & p_{12} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

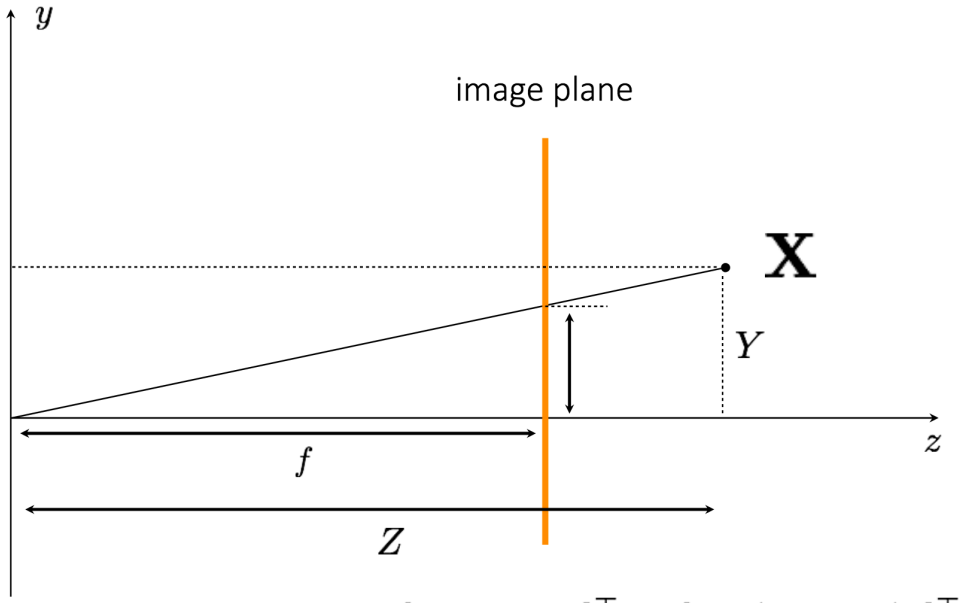
homogeneous image coordinates 3 x 1

camera matrix 3 x 4

homogeneous world coordinates 4 x 1

Camera Matrix Derivation

- We first assume camera origin, image origin, and world origin are the same
- Use similar triangles to derive image coordinates in terms of world coordinates and pinhole camera's focal length f



$$[X \quad Y \quad Z]^\top \mapsto [fX/Z \quad fY/Z]^\top$$

Camera Matrix Derivation

- For this derivation, we include the Z coordinate for the image homogeneous coordinate, so the camera matrix is scaled by factor of Z

Relationship from similar triangles:

$$[X \quad Y \quad Z]^\top \mapsto [fX/Z \quad fY/Z]^\top$$

General camera model:

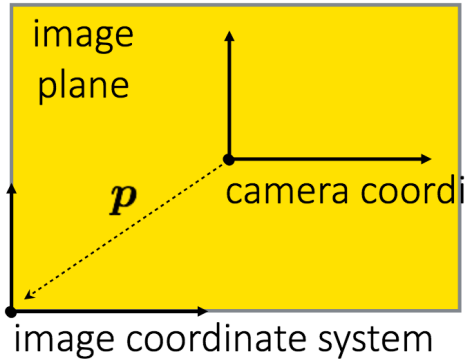
$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} p_1 & p_2 & p_3 & p_4 \\ p_5 & p_6 & p_7 & p_8 \\ p_9 & p_{10} & p_{11} & p_{12} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

What does the pinhole camera projection look like?

$$\mathbf{P} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Camera to Image Frame

- We now relax assumption that camera origin and image origin are the same
- Just add a vector to translate camera origin to image origin



How does the camera matrix change?

shift vector transforming camera origin to image origin

$$\mathbf{P} = \begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Camera Matrix Decomposition

- Can now divide camera matrix into 2 parts
  - one transforming 2D coordinates in camera coordinates to image coordinates, accounting for focal length and origin translation
  - One component projecting from 3D to 2D assuming camera and image origins are the same and focal length is 1 (i.e. image plane at z = 1)

Camera Matrix Decomposition

$$\mathbf{P} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

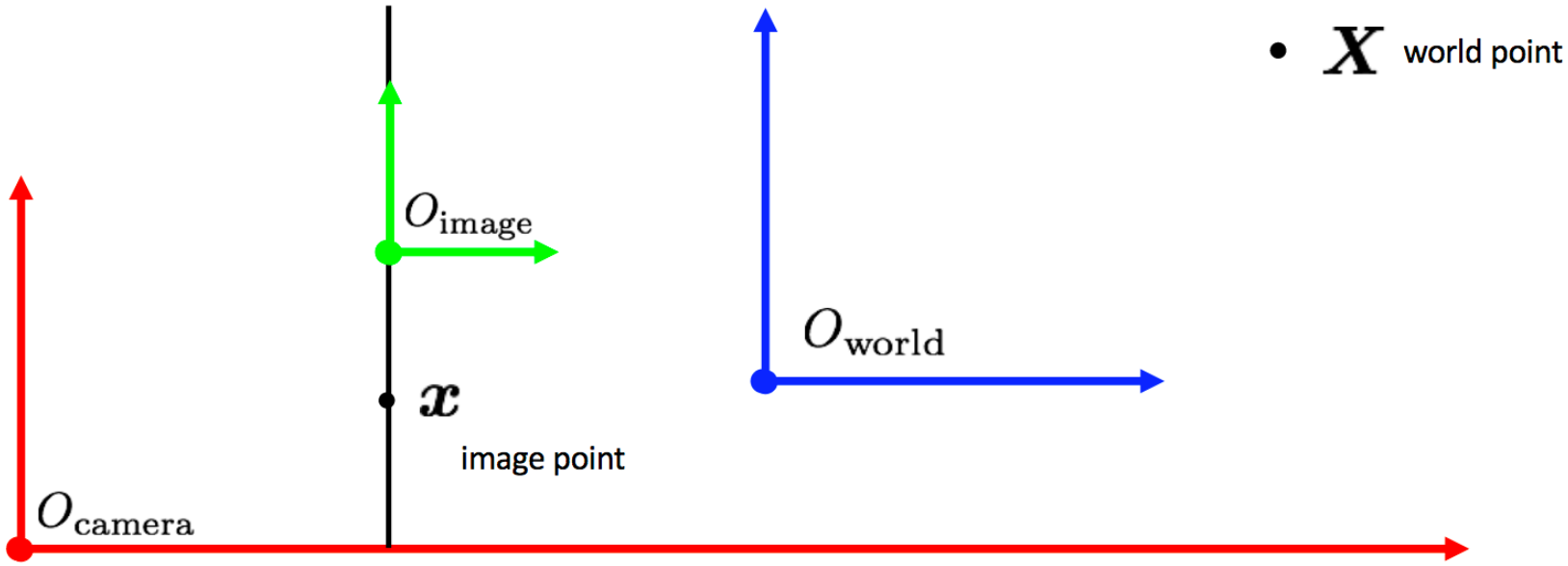
(homogeneous) transformation from 2D to 2D, accounting for not unit focal length and origin shift

(homogeneous) projection from 3D to 2D, assuming image plane at z = 1 and shared camera/image origin

Also written as: $\mathbf{P} = \mathbf{K}[\mathbf{I}|\mathbf{0}]$ where $\mathbf{K} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix}$

Different Coordinate Systems

- We generally have 3 coordinate systems (world, camera, image)

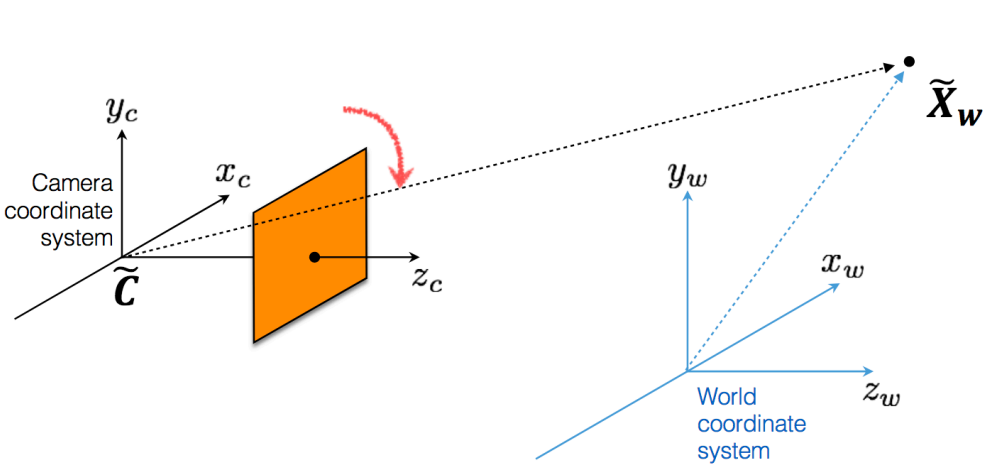


World to Camera Transform

- We accounted for transformation between camera coordinates and image coordinates
- Now account for transformation between world coordinates and camera coordinates
- First translate world coordinates so centered at camera coordinates
- Then rotate world coordinates to have same orientation as camera coordinates

World to Camera Transform

- First, we consider transformation in heterogeneous (not homogeneous coordinates; hence, why points below have tildes)



points now coincide

$$R \cdot (\widetilde{X}_w - \widetilde{C})$$

rotate translate

World to Camera Transform

- Rewrite world to camera transform in homogeneous coordinates

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R} & -\mathbf{RC} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \quad \text{or} \quad \mathbf{X_c} = \begin{bmatrix} \mathbf{R} & -\mathbf{R}\tilde{\mathbf{C}} \\ \mathbf{0} & 1 \end{bmatrix} \mathbf{X_w}$$

Combined Camera Matrix

- Combine world to camera transform with previous camera matrix
- Previous camera matrix (camera to image)

$$\mathbf{P} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \left[\begin{array}{ccc|c} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{array}\right]$$

- World to camera: $\mathbf{X_c} = \begin{bmatrix} \mathbf{R} & -\mathbf{R}\tilde{\mathbf{C}} \\ \mathbf{0} & 1 \end{bmatrix} \mathbf{X_w}$

Combined Camera Matrix

- Combined camera matrix (where K is intrinsic camera matrix):

$$\mathbf{P} = \begin{bmatrix} f & 0 & p_x \\ 0 & f & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} & -\mathbf{RC} \end{bmatrix}$$

intrinsic parameters (3 x 3):
correspond to camera internals
(sensor not at f = 1 and origin shift)

extrinsic parameters (3 x 4):
correspond to camera externals
(world-to-image transformation)

Combined Camera Matrix

- Breaking down components of combined camera matrix, which relates 3D world coordinates to 2D image coordinates

$$\mathbf{P} = \mathbf{K}\mathbf{R}[\mathbf{I}|-\mathbf{C}]$$

3x3	3x3	3x3	3x1
intrinsics	3D rotation	identity	3D translation

Corrections to Camera Matrix

- Since camera pixels may not be square and as camera sensor may be skewed, we allow intrinsic matrix to have different factors for x and y coordinates and add a skew term
- Total: 11 degrees of freedom (5 from intrinsics, 3 from 3D rotation, and 3 from 3D translation)

$$\mathbf{P} = \begin{bmatrix} \alpha_x & s & p_x \\ 0 & \alpha_y & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} & -\mathbf{RC} \end{bmatrix}$$

Projective Geometry

- We note that parallel lines that have a depth component (i.e the direction of the line has a Z component) eventually intersect in the image at a single point
- Example: take 2 parallel lines whose X and Y components are constant but which move in the Z direction
- From the derivation of the camera matrix (slide 91), ignoring the effects of different world, camera, image coordinate frames, we have

$$[X \quad Y \quad Z]^\top \mapsto [fX/Z \quad fY/Z]^\top$$

- As Z->∞, we see that fX/Z, fY/Z->0; hence, as the Z components (depth) of points in the 2 parallel lines increases, the points get closer together

Projective Geometry

- We can see this phenomenon of parallel lines moving in Z direction intersecting occur in this image

Projective Geometry

- We want to model this phenomenon geometrically; however, we cannot do this with Euclidean geometry as parallel lines cannot intersect in Euclidean geometry
- We instead use projective geometry, where we replace an axiom of Euclidean geometry
- Euclidean geometry: two parallel lines do not intersect
- Projective geometry: two parallel lines intersect in exactly one point

Projective Geometry

- The next few slides may be confusing; for the main takeaways, skip to slide 112
- A motivation of the definition of the projective plane P^2:
- From the pinhole camera model (slide 76), each light ray passing through the camera center contributes to exactly one point in the image plane
- Hence, we can associate every point in the image plane to a light ray
- Let the camera center be the origin (0, 0, 0) point in R^3
- A light ray passing through the camera center (these are the only light rays that contribute to the image) can be identified with a vector (x, y, z) representing its direction

Projective Geometry

- First observation: the vector (0, 0, 0) is not a direction vector for any light ray so we can exclude it as identifying a light ray
- Second observation: the scale of the vector (x, y, z) does not matter for identifying the light ray since we care about the direction of this vector (in other words the vectors (1, 2, 3) and (4, 5, 6) both identify the same light ray with the corresponding direction)
- Hence, to identify light rays passing through the camera center, we can consider the space R^3, exclude the origin, and then disregard scale

Projective Geometry

- This leads us to the definition of the projective plane $P^2 = (R^3 – \{\mathbf{0}\}) / R$, where $\mathbf{0}$ is the origin and dividing by R means to ignore scale
- Then, elements of the projective plane correspond to light rays passing through the camera center (more specifically, to the direction vectors of the light rays)

Projective Geometry

- We then observe that not all light rays passing through the image center actually hit the image plane
- Specifically, the light rays that do not move in the Z direction at all do not hit the image plane
- These light rays have direction vectors (x, y, 0), where x and y are not both 0
- We exclude these light rays as they do not affect the resulting image
- Another way to view points (x, y, 0) in projective space is points at infinity where parallel lines intersect (see ”Dis4_2B” slides on CCLE)

Projective Geometry

- Hence, we are left with light rays with direction vectors (x, y, z), where z is not 0 and we don’t care about scale
- As each light ray passing through the image center contributes to exactly one point in the image plane, we can identify points in the image plane with these same direction vectors (x, y, z), z not 0 and not caring about scale
- As we don’t care about scale, we can set z = 1 to normalize to get (x, y, 1)
- This is homogeneous coordinates!

Projective Geometry

- Main takeaways:
- Projective geometry assumes parallel lines intersect in exactly 1 point (vs 0 points for euclidean geometry)
- Projective plane $P^2 = (R^3 – \{\mathbf{0}\}) / R$, where $\mathbf{0}$ is the origin and dividing by R means to ignore scale
- Homogeneous coordinates, used to represent image coordinates, represent elements in the projective plane
- Projective plane is not a vector space (since projective plane does not contain origin); similarly, transformations acting on homogeneous coordinates / in projective plane are generally not linear transformations