# CSC317 Class Note 5 – Geometric Transformations

Outline:

1. Motivation
2. Homogeneous Coordinates
3. Linear Transformations
4. Euclidean and Affine Transformations
   1. Translations
   2. Rotation
   3. Scaling
   4. Shear
5. Some Observations
6. Textbook Chapter 6 Image Geometry
7. OpenCV Programming

## I. Motivation

## Reference – Chapter 6. A. Majumder et al., “Introduction to Visual Computing: Core Concepts in Computer Vision, Graphics, and Image Processing,” CRC Press, 2018. (ISBN 978-1-4822-4491-5).

**The material in this lecture note was taken from Chapter 6 of the reference book; and the OpenCV tutorial.**

*Geometric Transformation* means transforming a geometric entity (e.g. point, line, object) to another. This can happen in any dimension. For example, a 2D image can be transformed to another with translation, rotation, scaling, or shear.

## II. Homogeneous Coordinates

Before we discuss about Geometric Transformation, we need to take a look at an important concept about coordinates. Often, we believe the claim, “If two lines that are in parallel, they will never intersect.” However, in the world of Computer Vision, a close view from the person standing in the middle of a railway track shows that the two lines that form the railway will intersect. How can we represent this concept with a regular x- and y-axis?

When we begin to learn Geometry, we learn how to represent the location in Cartesian Coordinates. For a 2D position, we use (x, y) to represent a point, for a 3D position, we use (x, y, z) to represent a point. However, in the Cartesian world, two lines in parallel will not intersect with each other.

As matter of fact, in the world of Computer Vision, a 3D object will be viewed as a 2D image on the retina of the eye. Our brain identifies each point of the image in the retina as a ray into the 3D world. If we stand in the middle of two rails of a railway track, the two rails that are in parallel will intersect at the far end. The introduction of Homogeneous Coordinates solves the issue.

We use the concept of Homogeneous Coordinates to represent a point in a 2D world (x, y) as (x, y, 1). In general, we use (x/w, y/w) in Cartesian Coordinates to represent (x, y, w) in Homogeneous Coordinates.

Cartesian Coordinates Homogeneous Coordinates

1D: (3) (3, 1)

2D: (4, 6) => (4, 6, 1)

3D: (4, 6, 8) (4, 6, 8, 1)

Converting from the Cartesian space to the Homogeneous space, we just add a constant “1” as another parameter. It does not change the “dimension” of the point.

The 2D points (8, 12), (4, 6), (2, 3) in Cartesian Coordinates can be connected with a straight line. The Homogeneous Coordinates (8, 12, 4), (4, 6, 2), and (2, 3, 1) are represented as (2, 3) in the Cartesian (or Euclidean) space. That is exactly why we call these points “homogeneous” in the Homogeneous Space.

Cartesian Coordinates Homogeneous Coordinates

1D: (1/3) 2D: (1, 3)

2D: (2/6, 4/6) <= 3D: (2, 4, 6)

3D: (2, 4, 6) 4D: (4, 8, 12, 2)

2D: (x/w, y/w) 3D: (x, y, w)

2D: () 3D: (x, y, 0)

For a point at the far end of the railway track, we can use (x, y, 0) in Homogeneous Coordinates or ( in Cartesian Coordinates. The value of infinity is derived from x/0 and y/0. In the case of the intersection point of the two lines of a railway, we can represent the point as (0, 0, 0). Using the homogeneous coordinate to represent a 3D point (x, y, z) as a 4D point (x, y, z, 1). For the point at infinity, we represent it as (x, y, z, 0) where (x, y, z) is the direction of the point from the origin.

The reason we introduce the concept of Homogeneous Coordinates before talking about geometric transformations is because **we can represent the geometric operations as matrix and vector multiplications**. **Without using Homogeneous Coordinates, we can only represent the operations with systems of equations without using “transformation matrices”.**

In the rest of the chapter, we will represent points or vectors as 4 X 1 column vectors. Therefore, a 3D Cartesian point P = (x, y, z) will be written in the Homogeneous space as

x

y

P = z

1

Note: We always use a constant “1” to convert from Cartesian space to Homogeneous space.

## III. Linear Transformations

Linear transformation is a special kind of transformation. Given two points P and Q, the transformation L is considered a linear transformation if

L (aP + bQ) = a x L(p) + b x L(Q)

There are three types of linear transformations: *Euclidean, affine*, and *projective (or Perspective, or Homograph)*.

1. *Euclidean* – preserves length and angles.

A square cannot be changed to a rectangle by an Euclidean transformation. Translation and rotation are Euclidean transformations.

1. *Affine* – preserves the ratios of lengths and angles. A square can be transformed to a rectangle or rhombus by an affine transformation but cannot be formed to a general quadrilateral. Examples of affine transformations are shear and scaling. This means that parallel lines will remain parallel and intersecting lines will remain intersecting with Euclidean or affine transformations.

Based on the description online at the [Wikipedia website](https://en.wikipedia.org/wiki/Affine_transformation), an affine transformation preserves the following:

1. [**collinearity**](https://en.wikipedia.org/wiki/Collinearity) between points: three or more points which lie on the same line (called collinear points) continue to be collinear after the transformation.
2. [**parallelism**](https://en.wikipedia.org/wiki/Parallel_(geometry)): two or more lines which are parallel, continue to be parallel after the transformation.
3. [**convexity**](https://en.wikipedia.org/wiki/Convex_set)**of sets**: a convex set continues to be convex after the transformation. Moreover, the [extreme points](https://en.wikipedia.org/wiki/Extreme_point) of the original set are mapped to the extreme points of the transformed set.
4. The **ratios of lengths of parallel line segments** are preserved.
5. *Projective/Perspective transformation* – parallel lines become non-parallel. An example is shown in Fig. 6.2 of the handout. The projective transformation of a camera captured image and the relevant vanishing points.

In the following, we represent the concepts of geometric transformations with matrix forms.

## Euclidean and Affine Transformations

### Translation: P (x, y) -> P’ (x’, y’)

X’ = x + tx

Y’ = y + ty

x’ x + tx 1 0 tx x

P’ = Y’ = y + ty = 0 1 ty y = T (tx, ty) x P

1 1 0 0 1 1

Note: The last row is always (0 0 1) in the transformation matrix due to the use of Homogeneous Coordinates. As we mentioned previously, a translation operation can be represented as a matrix multiplication with a vector. The matrix T (tx, ty) is known as the “transformation matrix” of translation. There are two parameters that can change in the matrix. Thus, we sometimes call that the “*degree of freedom*” of a *translation operation* is 2.

### Rotation – We will illustrate this later in this lecture note

Section 6.3.2 of the reference book & the OpenCV Tutorial

### Scaling - Section 6.3.3 of the reference book & the OpenCV Tutorial

We will discuss the scaling operation later in the programming section.

### Shear – Section 6.3.4 of the reference book & the OpenCV Tutorial

### Shearing is to change the value of a point (in a 2D space) (x, y) only in one direction based on another dimension. For example, the following is a shear in the x direction. It is also called y-shearing. Intuitively, the original shape is shifted toward x-axis without changing y.

We can represent the transformation of y-shearing as follows:

X’ = x + a y

Y’ = y

(0,1)

(1,1)

(0,0) (1,0)

## Before shearing

If a = 0.5,

(0, 1) -> ( 0.5, 1) (x’= x + a; y=0 + 0.5= 0.5, y’= y = 1)

(1, 1) -> (1.5, 1)

(1, 0) -> (1, 0)

(0, 0) -> (0, 0)

(1,1)

(0,1)

(0,0) (1,0)

After shearing in the x director with the sharing parameter 0.5, the square turned into a “tilted” square. Note that the parallel lines are still parallel. *Shearing* is an example of *affine transformations* because the shape may change but the parallelism of lines is kept.

### Exercise

Draw a shearing in the y direction with the parameter of 0.5 using the original shape of a square.

## (Summary) Linear Transformations

L(aP+bQ) = a L(P) + b L(Q)

1. Euclidean Transformations: translation, rotation

Euclidean transformations preserve the lengths and angles.

1. Affine Transformations: scaling, shearing

Affine transformations preserve the ratios of lengths and angles.

## Some Observations

1. Euclidean transformations are a subset of affine transformations.
2. Euclidean transformations (translation and rotation) are called rigid body transformations that retain the shape of the original object.
3. Affine (shearing and scaling) can change the shape of the original object, but the parallel lines are still in parallel.
4. Prospective or perspective transformations change the shape, the parallelism of lines.
5. The translation matrix cannot be expressed as a 3 x 3 matrix multiplication with a vector in Cartesian Space, while scaling, rotation, or shear can be. We need to use the Homogeneous Space to represent the translation matrix in matrix and vector multiplication.
6. For rotation around any point other than the origin, we also need to use the homogeneous representation for the transformation matrix.

## VI. Chapter 6 (Textbook) Image Geometry

The material hereto after was taken from Chapter 6 in our textbook. OpenCV Tutorial does not cover this part about Data Interpolation and Scaling. The concept is very useful in practice. Think about this: how can we enlarge an image or shrink an image? In the original image, there are fewer number of pixel data. In our textbook, the concept of data interpolation, scaling, and rotation are discussed.

## 6.1 – 6.3 Data Interpolation

There are many situations in which we may want to change the shape, orientation, size of an image. We may want to increase the sampling size of an image so that it may improve the clarity. We may also want to decrease the sampling size without losing the clarity. Here we provide the explanation on how to increase the sampling size for a given 1D and 2D data. Suppose we want to increase the sampling size from 4 pixels to 8 pixels for a given range. Then, we can enlarge the image. We call the operation to fit more pixels into the given range ***interpolation***. For example, **the original sample range for 4 pixels will be enlarged to 8 pixels.** The “guessing” at the function values f(xi’) is called “***interpolation***”. We can use two methods to do the interpolation:

1. Nearest Neighbor Scaling
2. Linear (1D) Scaling or Bilinear (2D) Scaling

### Method 1 – Nearest Neighbor Scaling







In the process of adding more pixels, we need to decide (1) the value of the pixel coordinates, and (2) the gray levels. We ignore the first step because it is intuitively straight forward. We still need to determine the grayscale vaues for each new pixel. We can use the grayscale value of the nearest neighbor as the diagram illustrated. The disadvange is that the image may look “jagged”.

### Method 2 – Linear Scaling

In the following , we illustrate another method that makes the processed image smoother. The grayscale value of the pixel at F can be derived using the slope of the straight line formed by connecting the two pixels at the original image. We need to assume that we know the value of below. We can find the grayscale value of F as follows:

[F – f(x1)] / = [f(x2) – f(x1)] / 1

(Here we also assume that the distance between two pixels is 1.)

After simplifying the equation, we can conclude that

F = [f(x2) – f(x1)] + f(x1)

This is computable if both grayscale values of x1 and x2 are known. This method is known as a linear interpolation in a 1D image. For 2D images, we call the interpolation method Bilinear.





Applying Linear Scaling to an image, we call it Bilinear Interpolation.

We ignore the details below.



In this digram, the open circles are the original image pixels, while the solid dots are the interpolated pixels. This diagram illustrates the concept of Bilinear method in a 2D image. In the following image, the above diagram is annotated with coordinates and scales, e.g.,



If you are interested in the derivation, please refer to the textbook.

Here are the resulting images applying these two methods, respectively.



Original Image



As you can see that the Nearest Neighbor Scaling method generates a somewhat “jagged” edges, whereas Bilinear Interpolation generates a smoother image.

6.3 General Data Scaling (optional) - Skipped

This method is a generalization of the interpolation methods we mentioned previously, using the interpolated pixel as the origin. We may ignore the details.





6.4 Scaling of Images

These are examples taken from our textbook. The first image is the original image. You can easily conclude that the Nearest Neighbor generates an image with jagged edges comparing with Bilinear.

A picture containing computer

Description automatically generatedAn old photo of a person

Description automatically generatedoriginal image


Bilinear

Nearest Neighbor

Original Image

6.5 Scaling Smaller - We can ignore this section.

6.6 Rotation

we will derive the formula for rotating with the center at the origin.

Prove the following:

x’ cos -sin x

y’ = sin cos y





Proof:

sin(

Let the angle between the x-axis and the line from the origin to (x, y) be called . We can conclude that

and

Can you simplify the representation for x’ and y’ to get rid of the representation with

= r [

Exercise

Can you complete the proof?

We can now take a look at programming about rotation. We can skip section 6.7 Correcting Image Distortion in our textbook.

## **OpenCV Programming - Geometric Transformations of Images**

## Goal

* Learn to apply different geometric transformation to images like translation, rotation, affine transformation etc.

## Transformations

## OpenCV provides two transformation functions, cv2.warpAffine and cv2.warpPerspective, with which you can have all kinds of transformations. cv2.warpAffine takes a 2x3 transformation matrix while cv2.warpPerspective takes a 3x3 transformation matrix as input.

## Scaling

## Scaling is just resizing of the image. OpenCV comes with a function cv2.resize() for this purpose. The size of the image can be specified manually, or you can specify the scaling factor. Different interpolation methods are used. Preferable interpolation methods are cv2.INTER\_AREA for shrinking and cv2.INTER\_CUBIC (slow) & cv2.INTER\_LINEAR for zooming. By default, interpolation method used is cv2.INTER\_LINEAR for all resizing purposes. You can resize an input image either of following methods:

**import** **cv2**

**import** **numpy** **as** **np**

img = cv2.imread('messi5.jpg')

res = cv2.resize(img,None,fx=2, fy=2, interpolation = cv2.INTER\_CUBIC)

*#OR*

height, width = img.shape[:2]

res = cv2.resize(img,(2\*width, 2\*height), interpolation = cv2.INTER\_CUBIC)

## Translation

## Translation is the shifting of object’s location. If you know the shift in (x,y) direction, let it be (tx, ty) , you can create the transformation matrix M as follows:

1 0 tx

M = 0 1 ty

0 0 1

## (Note: The original document of OpenCV Tutorial does not have the third row for the matrix M. It will not be able to represent the translation operation as a matrix multiplication that way.)

## You can take make it into a Numpy array of type np.float32 and pass it into cv2.warpAffine() function. See below example for a shift of (100,50):

**import** **cv2**

**import** **numpy** **as** **np**

img = cv2.imread('messi5.jpg',0)

rows,cols = img.shape

M = np.float32([[1,0,100],[0,1,50]])

dst = cv2.warpAffine(img,M,(cols,rows))

cv2.imshow('img',dst)

cv2.waitKey(0)

cv2.destroyAllWindows()

## Warning

## Third argument of the **cv2.warpAffine()** function is the size of the output image, which should be in the form of **(width, height)**. Remember width = number of columns, and height = number of rows.

## TranslationSee the result below:

## Rotation

Rotation of an image for an angle  is achieved by the transformation matrix of the form

M =

Note that the third row is [ 0 0 1] because M is represented as a homogeneous representation.

Also, OpenCV provides scaled rotation with adjustable center of rotation so that you can rotate at any location you prefer. Modified transformation matrix is given by

-

0 0 1

\begin{array}{l} \alpha =  scale \cdot \cos \theta , \\ \beta =  scale \cdot \sin \theta \end{array},where

To find this transformation matrix with programming, OpenCV provides a function, **cv2.getRotationMatrix2D**. Check below example which rotates the image by 90 degree with respect to center without any scaling.

img = cv2.imread('messi5.jpg',0)

rows,cols = img.shape

M = cv2.getRotationMatrix2D((cols/2,rows/2),90,1)

dst = cv2.warpAffine(img,M,(cols,rows))

See the result:



## Affine Transformation

In affine transformation, all parallel lines in the original image will still be parallel in the output image. To find the transformation matrix, we need three points from input image and their corresponding locations in output image. Then **cv2.getAffineTransform** will create a 2x3 matrix which is to be passed to **cv2.warpAffine**.

Check below example, and also look at the points I selected (which are marked in Green color):

img = cv2.imread('drawing.png')

rows,cols,ch = img.shape

pts1 = np.float32([[50,50],[200,50],[50,200]])

pts2 = np.float32([[10,100],[200,50],[100,250]])

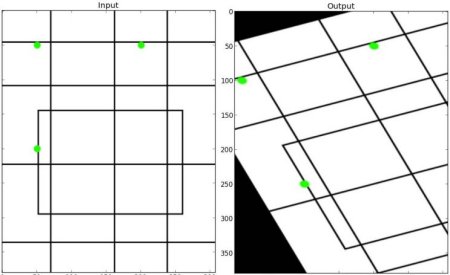
M = cv2.getAffineTransform(pts1,pts2)

dst = cv2.warpAffine(img,M,(cols,rows))

plt.subplot(121),plt.imshow(img),plt.title('Input')

plt.subplot(122),plt.imshow(dst),plt.title('Output')

plt.show()

See the result:

## Perspective Transformation

For perspective transformation, you need a 3x3 transformation matrix. Straight lines will remain straight even after the transformation. To find this transformation matrix, you need 4 points on the input image and corresponding points on the output image. Among these 4 points, 3 of them should not be collinear. Then transformation matrix can be found by the function **cv2.getPerspectiveTransform**. Then apply **cv2.warpPerspective** with this 3x3 transformation matrix.

See the code below:

img = cv2.imread('sudokusmall.png')

rows,cols,ch = img.shape

pts1 = np.float32([[56,65],[368,52],[28,387],[389,390]])

pts2 = np.float32([[0,0],[300,0],[0,300],[300,300]])

M = cv2.getPerspectiveTransform(pts1,pts2)

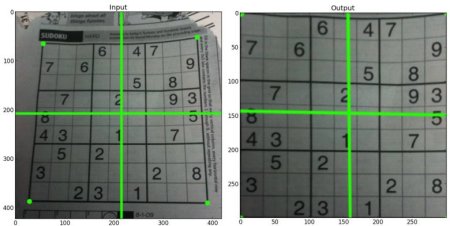
dst = cv2.warpPerspective(img,M,(300,300))

plt.subplot(121),plt.imshow(img),plt.title('Input')

plt.subplot(122),plt.imshow(dst),plt.title('Output')

plt.show()

Result:



There are 4 reference points in the left image. These four pixels become the four corners of the transformed image. Note that the non-parallel lines in the original image may become parallel, and vice versa. That is, parallel lines may become non-parallel.