Project 1: Transformations

Matias Cinera
Computer Science Department, *University of South Florida*
Tampa, Florida, 33612, USA
cinera@usf.edu

***Abstract*— This paper describes my implementation and understanding of geometric transformations and pixel operations of images. A geometric transformation is a mathematical relationship between two planar surfaces (images) which hold some type of relationship. Pixel operations modify the value of pixels to alter how the image looks without affecting its shape. This paper will implement (in code) and test the fundamental geometric transformations and pixel operations over a batch of images.**

***Keywords*—Geometric transformations, Pixel operations**

I. Introduction

This document is the report of the 1st project from my fall 2021 class (CAP4410 – Computer Vision), as an undergraduate student. The two individuals who are overviewing this project are our professor: Sudeep Sarkar (Professor and Department Chair CS & CSE Department) and our Teaching Assistant: Daniel Sawyer (Ph.D. Student). For this project, Mr. Sawyer provided a template python file. This file contained code that set up the data (batch of images) and some supplementary functions.

The project required students to implement five geometric transformations (Translation, Rotation, Scaling, Affine and Projection) and three pixel-operations (Color Brightness and Contrast, Gamma Correction, and Histogram equalization). The project also required two normalization operations which are commonly used for deep learning purposes (Mean & Standard Deviation and Batch Normalization). Each image will be processed by all geometric transformation and pixel operation functions (each function outputting an image). Finally, the batches are processed by the two normalization functions and stored in an output folder.

II. Homogeneous Coordinates.

A. Relation to Geometric transformations and Pixel operations

A geometric transformation is a mathematical relationship between two planar surfaces (images in this project) which hold some type of relationship. A pixel is composed of three integers (red, green, blue) in a range from 0 to 255. These integers represent the intensity of the given values (r, g, b). Pixel operations are essentially just modifying these values.

Since in this project we are dealing with images, we could think of a pixel as a point in a 2-dimensional coordinate plane (our image). For this project, we use homogeneous coordinates to represent 2d points (pixels).

B. Homogeneous Coordinates of points

A 2-dimensional point: $\mathbf{x} = (x, y)$

Can be represented as a 3-dimensional line through the origin.

$$\tilde{\mathbf{x}} = w \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Where $\tilde{x}$ is the homogeneous representation of the point. By this logic, we could also transform a 3D line into a 2D point. However, this is a theoretical 3D representation, thus, homogeneous coordinates are only used to simplify operations. This is because many transformations can be described as matrix multiplication.

III. Geometric Transformations

A. Translation

The translation is shifting the 2-dimensional plane by a percent of the plane. After translation, the image retains the same orientation, angles, and length. This is the least disruptive transformation in this project. This transformation can be achieved by adding the t vector (shift x, shift y).$\mathbf{x} = (x, y)$, $\mathbf{t} = (t_x, t_y)$

$$\mathbf{x} + \mathbf{t} = (x, y) + (t_x, t_y)$$

The same expression in homogeneous coordinates:

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

$$\widetilde{\mathbf{x}'} = \begin{bmatrix} \mathbf{I}^{2\times 2} & \mathbf{t}^{2\times 1} \\ \mathbf{0}^{1\times 2} & 1 \end{bmatrix} \tilde{\mathbf{x}}$$

Pseudo-Code:

```
Translation(img, x_shift, y_shift):
    h, w = img_height, img_width
    new_img = image of size(height*width)
    for gx to height:
        for gy to width:
            fx = gx + w*x_shift
            fy = gy + h*y_shift
            if((fx,fy) exists in img):
                img_new[fx][fy] = img[gx][gy]
    return img_new
```

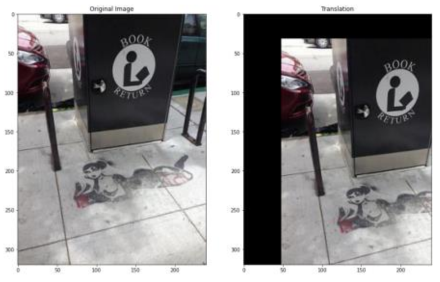

Figure 1. Translation, shift x = 0.2, shift y = 0.1

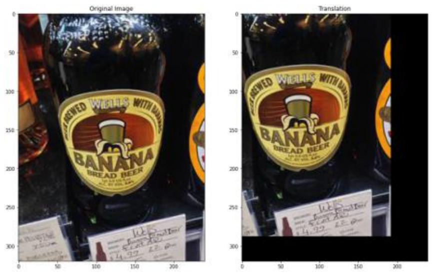

Figure 2. Translation, shift x = -0.2, shift y = -0.1

B. *Rotation*

This transformation will rotate the image over theta. Theta is a 2D rotation angle specified in degrees. After translation, the image will preserve orientation, angles, and length. This transformation can be achieved by multiplying the rotational matrix with the coordinates of the image. To do this, we transform the image coordinates into homogeneous coordinates.

$$\mathbf{x}' = \mathbf{R}^{2\times 2}\mathbf{x} + \mathbf{t}$$

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

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & t_x \\ \sin(\theta) & \cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\widetilde{\mathbf{x}'} = \begin{bmatrix} \mathbf{R}^{2\times 2} & \mathbf{t}^{2\times 1} \\ \mathbf{0}^{1\times 2} & 1 \end{bmatrix} \widetilde{\mathbf{x}}$$

The homogeneous representation of this transformation includes $\mathbf{t}_x$ and $\mathbf{t}_y$ which are the values translation values. For this project, $\mathbf{t}_x$ and $\mathbf{t}_y$ will be hard coded to avoid a translation.

Pseudo-Code(t is fixed for this project):

```
1. Rotate(img, Theta):
2.     Theta = -Theta
3.     new_img = image of size(height*width)
4.     def h (x, y):
5.         Rotational_Matrix =  $R^{2\times2}x + t$
6.         return R • array([[x],[y],[1]])
7.     return inverse_warp(h, img.shape)
```

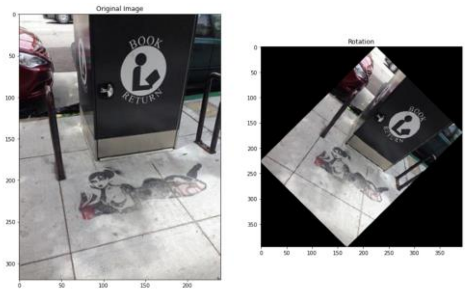

Figure 3. Rotation, Θ = -45 °

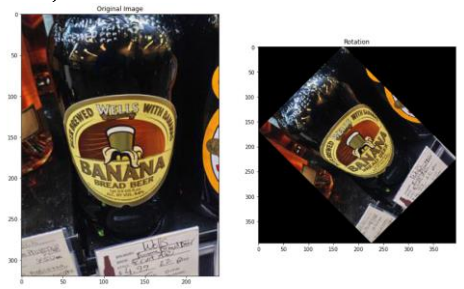

Figure 4. Rotation, Θ = 45 °

C. *Scaling*

Conceptually scaling an image based on a percent is self-descriptive. Given a value larger than 100% the image will expand, a value less than 100% will shrink the image. Scaling will only retain the angles of the original image. To achieve this, we can modify the rotation operation, however, now we will multiply the rotational matrix by our scaling value.

$$\mathbf{x}' = s\mathbf{R}\mathbf{x} + \mathbf{t}$$

$$\widetilde{\mathbf{x}'} = \begin{bmatrix} \begin{bmatrix} a & -b \\ b & a \end{bmatrix} & \mathbf{t}^{2\times 1} \\ \mathbf{0}^{1\times 2} & 1 \end{bmatrix} \widetilde{\mathbf{x}}$$

Since we only multiplied the scaling ratio to the rotational matrix the pseudocode will be the same as the pseudocode of *Rotation*. The only difference will be line 5:

```
Rotational_Matrix =  $sR^{2\times2}x + t$
```

For the purpose of only testing scaling, the values of **theta** and **t** will remain fixed in this project.

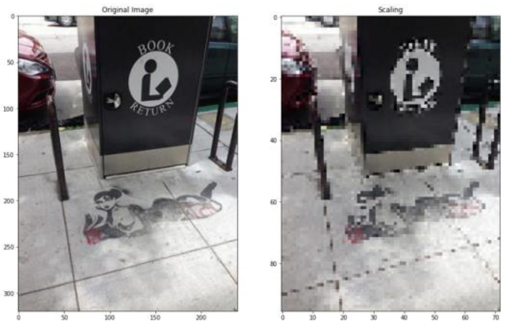

Figure 5. Scaling, 30%

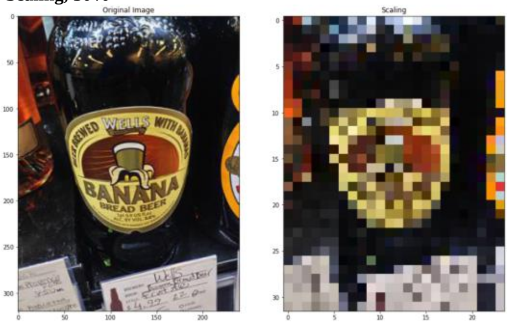

Figure 6. Scaling, 10%

D. *Affine*

Affine is achieved by stretching the image while conceptually remaining in a 2-dimensional plane. This transformation allows you to shape the image in any if parallel lines are preserved. To achieve this transformation **A** (a 2 by 2 vector) is needed to represent the stretching, and a **t** (fixed translation value).

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & t_x \\ a_{10} & a_{11} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Pseudo-Code:

```
Affine(img, A):
    new_img = image of size(height*width)
    def
    def h (x, y):
            Rotational_Matrix =
    [A[0,0], A[0,1], t[0]]
    [A[1,0], A[1,1], t[1]]
    [ 0 , 0 , 1 ]
            return R • array([[x],[y],[1]])
    return inverse_warp(h, img.shape)
```

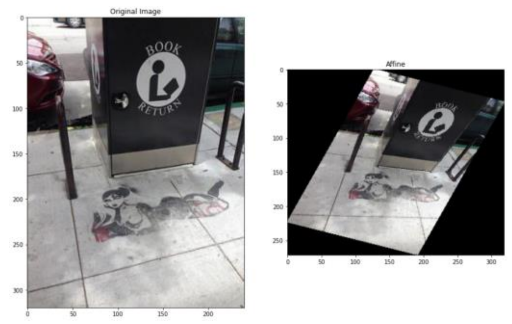

Figure 7. Affine, A = [[0.8, 0.4], [0.2, 0.7]] T = [0.0, 0.0]

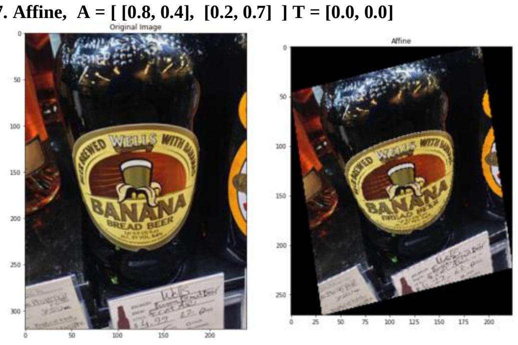

Figure 8. Affine, A = [[0.8, 0.1], [0.2, 0.7]] T = [0.0, 0.0]

E. *Projection*

A very common transformation in the field, projection only required the image to retain straight lines. However, all the other properties of the image are lost in the process. Conceptually, I like to think of projection as rotating a 2d plane on a 3d plane. Since we are modifying all 9 values of the homogeneous representation of the image, we will use H (a 3 by 3 vector) to achieve this. This geometric model represents two perspective views of a plane.

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\widetilde{\mathbf{x}'} = \widetilde{\mathbf{H}}\widetilde{\mathbf{x}}$$

$$\mathbf{x} = \begin{bmatrix} a/c \\ b/c \end{bmatrix}$$

Pseudo-Code:

```
Projective(img, H):
    new_img = image of size(height*width)
    def h (x, y):
            out =  H • [[x],[y],[1]]
            return (out[0]/out[2], out[1]/out[2])
    def h_inv (x, y):
            out =  inverse(H) • [[x],[y],[1]]
            return (out[0]/out[2], out[1]/out[2])
    return inverse_warp(h, h_inv)
```

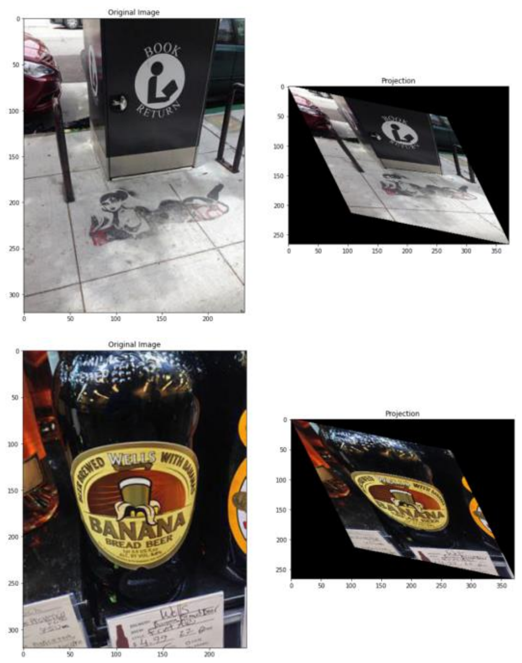

Figure 9 & 10. Projection, H = [[1.0, 0.3, 0.0],
[0.2, 0.6, 0.0],
[0.0, 0.0, 0.9]]

IV. Pixel Operations

As explained before, a pixel is a point on a 2d plane (image). However, each pixel has 3 different values (R, G, B), essentially, making an image a 3d vector of size (width, height, 3). [1]To alter the luminosity of each pixel, we would first need to transform the image to lab (Lightness, a = Red/Green, b = Blue/Yellow). The luminosity channel will basically let us adjust how the color looks, a process that doesn't affect the (R, G, B) values. For example, if a red pixel (255, 0, 0) has an L value of 0, we would simply not be able to tell the color of the pixel.

A. *Brightness and contrast*

In this project we are only manipulating the L channel, meaning that the pixel values are unchanged in this process. To achieve this, we use the image processor operator ***h*** which is a function that takes an input image ***f(i, j)*** and produces an output image ***g(i, j)***

$$g(i,j) = h(f(i,j))$$

Pseudo-Code:

```
brightness_contrast (a, b):
    img /= max(original)
    out = rgb to lab (img)
    out[:,:,0] = change_contrast(change_brightness(out[:,:,0]/100, b), a) * 100.0
    return uint8(out[:,:,0])
```

Note: Change_contrast/brightness are functions that will alter the R, G, B values. However, the transformation on the pseudocode only works on the L channel.

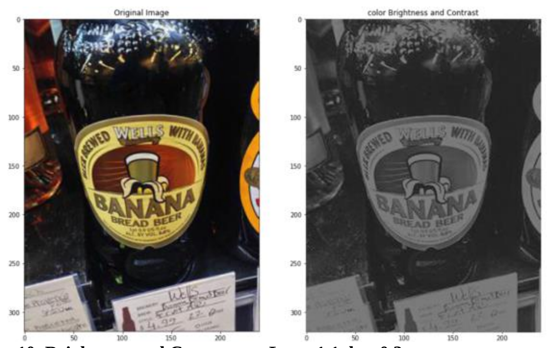
Figure 10. Brightness and Contrast on L a = 1.1, b = 0.2

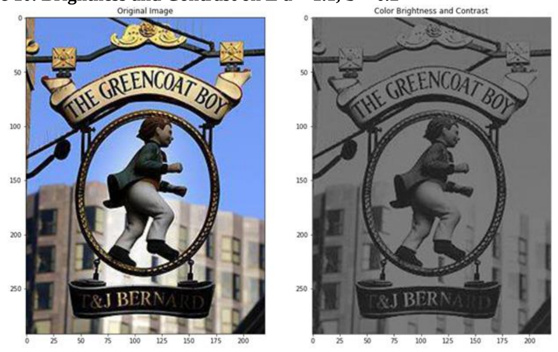
Figure 11. Brightness and Contrast on L a = 1.1, b = 0.2

B. *Gamma Correction*

Gamma correction is an operation commonly used to remove the non-linear mapping between input radiance and quantized pixel values. Is commonly represented by the given function(a gamma value of 2.2 is a good fit for most modern cameras):

$$g(i,j) = [f(i,j)]^{\frac{1}{\gamma}}$$

Pseudo-Code:

```
gamma_correction ():
     img /= max(original)
     out = rgb to lab (img)
     def gc (input, gamma):
       return (power(input, 1/gamma))
     out[:,:,0] =  gc (out[:,:,0]/100, 2.2) * 100.0
     out_img = lab 2 rgb(out)*255
     return uint8(out_img[:,:,0])
```

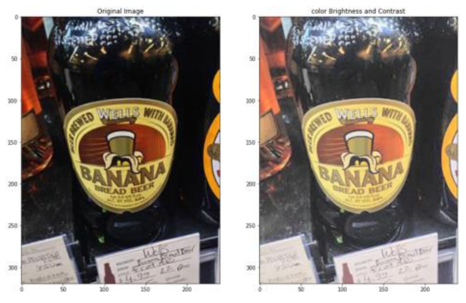
Figure 12. Gamma Correction 1st sample

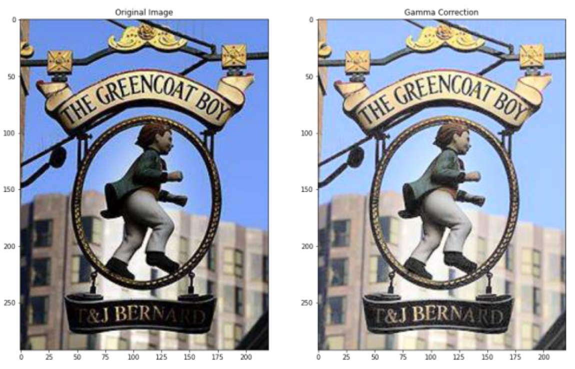
Figure 13. Gamma Correction 1st sample

C. *Histogram Equalization*

The goal of this operation is to turn the histogram shape (from the RGB values of a picture) into a flatter one.

Pseudo-Code:

```
gamma_correction ():
    out = rgb to lab (img)
    def gc (out, gamma):
   channel_l = in_im[:, :, 0].astype(np.uint8)
   histogram_l, bin_edges_l = histogram(out [:, :, 0], bins=101)
   histogram_l = histogram_l/sum(histogram_l)
   cummulative_histogram_l = cumulative sum(histogram_l)
   hist_norm_im = image of size out
   hist_norm_im[:, :, 0] = cummulative_histogram_l[int(out[:, :, 0])] * 100
   hist_norm_im[:, :, 1] = out[:, :, 1]
   hist_norm_im[:, :, 2] = out[:, :, 2]
   img_new = lab 2rgb (hist_norm_im)
   img_new *= 255
   return uint8(img_new)
```

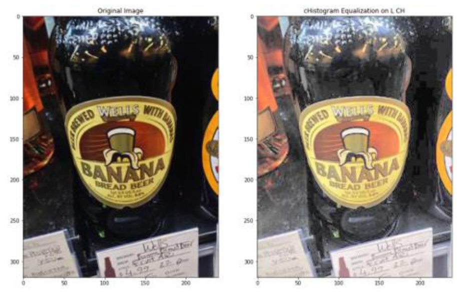
Figure 14. Histogram Equalization on L CH, sample 1

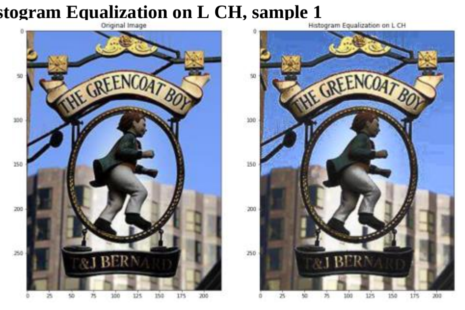
Figure 15. Histogram Equalization on L CH, sample 2

V. IMAGE NORMALIZATION OPEREATIONS FOR DEEP LEARNING

A. *Mean and Standard Deviation*

For this operation we were given a batch of images, from these batch we would get the overall mean and standard deviation from the RGB values, then return two vectors containing the overall result of both operations. These vectors will only have 3 values, values for RGB.

Pseudo-Code:

```
1. mean_sd (batch):
2.      mean = [[],[],[]]
3.      sd = [[],[],[]]
4.      for i in batch:
5.              mean[0].append(mean(item[0]))
6.              mean[1].append(mean(item[1]))
7.              mean[2].append(mean(item[2]))
8.      out_mean = [average[mean[0]],[ mean[1]],[ mean[2]]]
9.      out_sd = [average[sd[0]],[ sd[1]],[ sd[2]]]
10.     return out_mean and out_sd
```

B. *Normalization*

For this operation we will divide each pixel of the inputs image *f(i ,j)*, by the maximum possible value (255 in RGB), this is one so each value of RGB is between 0 and 1. Then compute the norm by subtracting the mean value from each pixel in the image and divide the standard deviation. This transformation aligns the mean and standard deviation method of the color channels and accelerates the convergence of deep learning.

Pseudo-Code:

```
1. batch_norm (batch, resize_shape):
2.      new_batch = []
3.      for img in batch:
4.              img = float(img)
5.              img_norm = = img/max(img)
6.              img_norm = = resize(resize_shape)
7.              sd = std(img_norm, axis=(0,1))
8.              img_sd = (img_norm - mean[None, None, :])/std[None, None, :]
9.              clipped = = (img_sd + 3)/6
10.             return clipped np.max(img)
```

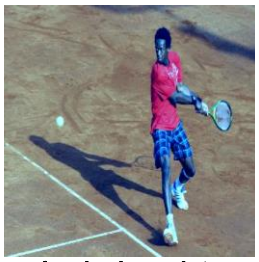
Figure 16. Normalized image from batch, sample 1

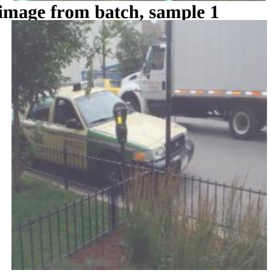
Figure 16. Normalized image from batch, sample 1

VI. Results

Overall, the results of this project were successful. Almost all of the functions acted as expected. The only function that didn't work as expected was the mean and normal deviation function. The function is supposed to return two vectors of size 3, with the mean and sd of RGB values. Even though my function successfully returned two vectors that met this requirement, the function didn't seem to work as expected.

VII. Conclusion

At first, this project seemed to be more complicated than that it was. Most of the functions to be implemented were given by the professor, we just needed to adapt them to the boundaries of the project. Only knowing the basics of python gave me a chance to unravel the power of this programming language. As for the concepts discussed in this paper, most of them are still challenging to me. However, now I have a general understanding of them and a powerful tool to implement them.

References

[1] "Lab color space and values: X-rite color blog," X. [Online]. Available: https://www.xrite.com/blog/lab-color-space. [Accessed: 15-Sep-2021].

[2] T. Mouw, "Lab color space and values: X-rite color blog," X. [Online]. Available: https://www.xrite.com/blog/lab-color-space. [Accessed: 16-Sep-2021].

[3] S.Sarkar, "CAP4410 Lecture 3 Pixel operations color_histogram," Pixel operations color histogram. [Online]. Available: CAP_4410_Lecture_3 [Accessed: 10-Sep-2021].

[4] S.Sarkar, "CAP4410 Lecture 2 2D Geometric Primitives and Transformation [Online]. Available: CAP_4410_Lecture 2 [Accessed: 10-Sep-2021].