1.Introduction

Affine transformation is a linear mapping of points in a two-dimensional space. It is a powerful tool for image processing, and can be used to perform a variety of tasks, such as scaling, rotating, shearing, and zooming images.Affine transformation matrices are typically represented by 3x3 matrices. The first two columns of the matrix represent the scaling and rotation components of the transformation, while the third column represents the shear component.

To apply an affine transformation to an image, we simply multiply the affine transformation matrix by the pixel coordinates of each pixel in the input image. The resulting pixel coordinates are the corresponding pixel coordinates in the output image.

Affine transformation can be implemented using both forward mapping and backward mapping.

In forward mapping, we start with the input image and compute the output image by applying the affine transformation to each pixel in the input image. This can produce aliasing artifacts in the output image.

In backward mapping, we start with the output image and compute the input image by inverting the affine transformation and applying it to each pixel in the output image. This method can produce better quality results than forward mapping, but it is more computationally expensive.

Also in backward mapping, applying bilinear interpolation results a higher qualith output with less aliasing but also causes blur.

In this project, we implemented both forward and backward mapping for affine transformation. We also implemented bilinear interpolation to improve the quality of the backward mapped images. Explained logic of the implementation and show the results of the affine tansformations using Figure 1.

Figure 1

2.Methods

This project aims to show 3 different methods on the example image by applying different affine transforms. Affine transformations are applied by using forward mapping, backward mapping without interpolation and backward mapping with bilinear interpolation.

I applied these three methods to scale image by 1.5 coefficient, rotate image by 30 degrees, vertically and horizontally shearing image by 0.3 shear coefficient and zoom image by 1.5 coefficient.

To perform affine transformation for all 3 different methods, coordinates of every pixel of the output are calculated by multplying coordinates of pixels of input with corresponding 3x3 affine transform matrix.For forward transform, affine transform matrices (Fig. 2) are directly multiplied by the coordinates; for backward transform, inverse of the matrices are multiplied by the coordinates.

Identity

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

Reflection

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

Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scale

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotation

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

Shear-X

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

Shear-Y

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

Figure 2

3. Operations

Under this section, this report explains 5 operations on example image (Fig 1) for 3 different operations.

3.1 Scaling

3.1.1 Forward Mapping

Firstly a new black image is generated with scaled width and height which are calculated by multiplying original image width and height with scaling factor. Then by multiplying scale matrix and coordinate of each pixel, coordinage of each corresponding pixel are calculated.Then copied from original image to blank image.Result is given at Fig 3.1.1

3.1.2 Backward Mapping Without Interpolation

Firstly a new black image is generated with scaled width and height which are calculated by multiplying original image width and height with scaling factor. Then by multiplying inverse of the scale matrix and coordinates of blank images, coordinates of the pixel which belongs to the original image are obtained. Then blank image's pixel is filled from pixel of original image. Result is given at Fig 3.1.2

3.1.3 Backward Mapping With Bilinear Interpolation

Exact same operations applied with backward mapping but weighted average of the four nearest neighbor pixels in the original image are calculated for 3 channels for pixel value of new image's corresponding index. Result is given at Fig 3.2.1

Figure 3.1.1

Figure 3.1.2

Figure 3.1.3

3.2 Rotating

3.2.1 Forward Mapping

Firstly a new black image is generated with width and height of original image by adding spaces to fit rotated. Then by multiplying rotate matrix and coordinate of each pixel, coordinage of each corresponding pixel are calculated.Then copied from original image to blank image.Result is given at Fig 3.2.1

3.2.2 Backward Mapping Without Interpolation

Firstly a new black image is generated with width and height of original image by adding spaces to fit rotated. Then by multiplying inverse of the rotate matrix and coordinates of blank image, coordinates of the pixel which belongs to the original image are obtained. Then blank image's pixel is filled from pixel of original image. Result is given at Fig 3.2.2

3.2.3 Backward Mapping With Bilinear Interpolation

Exact same operations applied with backward mapping but weighted average of the four nearest neighbor pixels in the original image are calculated for 3 channels for pixel value of new image's corresponding index. Result is given at Fig 3.2.1

Figure 3.2.1

Figure 3.2.2

Figure 3.2.3

3.3 Horizontal and Vertical Shearing

3.3.1 Forward Mapping

Firstly a new black image is generated with width and height of original image by adding spaces to fit rotated. Then by multiplying shear matrix and coordinate of each pixel, coordinage of each corresponding pixel are calculated.Then copied from original image to blank image.Result is given at Fig 3.3.1

3.3.2 Backward Mapping Without Interpolation

Firstly a new black image is generated with width and height of original image by adding spaces to fit sheared image. Then by multiplying inverse of the shear matrix and coordinates of blank image, coordinates of the pixel which belongs to the original image are obtained. Then blank image's pixel is filled from pixel of original image. Result is given at Fig 3.2.2

3.3.3 Backward Mapping With Bilinear Interpolation

Exact same operations applied with backward mapping but weighted average of the four nearest neighbor pixels in the original image are calculated for 3 channels for pixel value of new image's corresponding index. Result is given at Fig 3.3.1

Figure 3.3.1

Figure 3.3.2

Figure 3.3.3

3.4 Zoom

To zoom center of the image, output image of same size with input image is expected. Output image shows part of a input image.To achieve this, for each of 3 methods; corresponding scale function is called and part of center of scaled image is also means output of the zoom.Results are given at Fig 3.4.1 (Forward transform) ,Fig 3.4.2 (Backward transform) and Fig 3.4.3 (Backward transform with bilinear transformation)

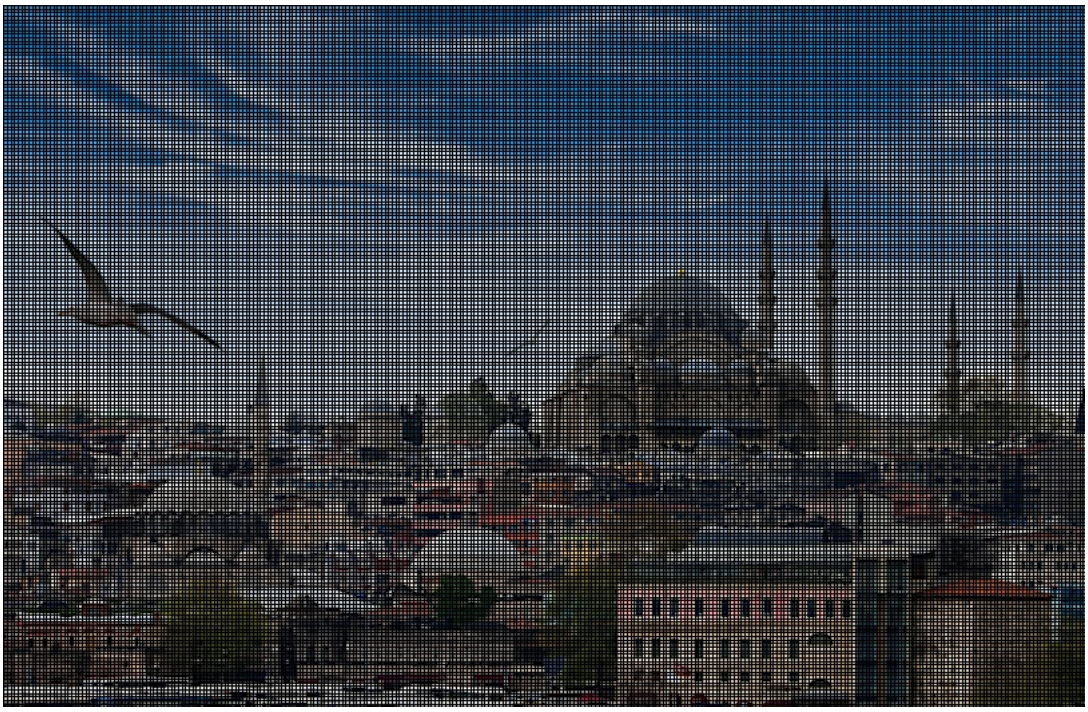

Figure 3.4.1

Figure 3.4.2

Figure 3.4.3

3.5 Code explanation

In this section, codes are explained with the comments.

3.5.1 Main script

This script runs the program.

```
#Read image
image = cv2.imread("istanbul.jpg")


#Converting image to python list to pass the functions
image = image.tolist()


#In this section,images are sent to transform functions
#Outputs of those functions are stored in numpy arrays
istanbul_scaled_forward_mapping = forward_mapping.scaleImage(image,1.5)                    #
istanbul_scaled_backward_mapping = backward_mapping.scaleImage(image,1.5)                  #scale
i_istanbul_scaled_backward_mapping = backward_mapping_w_interpolation.scaleImage(image,1.5) #


istanbul_rotated_forward_mapping = forward_mapping.rotateImage(image,30)                       #
istanbul_rotated_backward_mapping = backward_mapping.rotateImage(image,30)                     #rotate
i_istanbul_rotated_backward_mapping = backward_mapping_w_interpolation.rotateImage(image,30)   #


istanbul_vertical_chear_forward_mapping = forward_mapping.verticalShearImage(image,0.3)                  #
istanbul_vertical_chear_backward_mapping = backward_mapping.verticalShearImage(image,0.3)                #vertical shear
i_istanbul_vertical_chear_forward_mapping = backward_mapping_w_interpolation.verticalShearImage(image,0.3) #


istanbul_horizontal_shear_forward_mapping = forward_mapping.horizontalShearImage(image,0.3)                    #
istanbul_horizontal_shear_backward_mapping = backward_mapping.horizontalShearImage(image,0.3)                  #horizontal shear
i_istanbul_horizontal_shear_backward_mapping = backward_mapping_w_interpolation.horizontalShearImage(image,0.3) #


istanbul_zoomed_forward_mapping = forward_mapping.zoomImage(image,1.6)                     #
istanbul_zoomed_backward_mapping = backward_mapping.zoomImage(image,1.6)                   #zoom
i_istanbul_zoomed_backward_mapping = backward_mapping_w_interpolation.zoomImage(image,1.6) #
```

```
#Converting lists back to numpy arrays

istanbul_scaled_forward_mapping = np.array(istanbul_scaled_forward_mapping , dtype=np.uint8)
istanbul_scaled_backward_mapping = np.array(istanbul_scaled_backward_mapping , dtype=np.uint8)
i_istanbul_scaled_backward_mapping = np.array(i_istanbul_scaled_backward_mapping , dtype=np.uint8)

istanbul_rotated_forward_mapping = np.array(istanbul_rotated_forward_mapping , dtype=np.uint8)
istanbul_rotated_backward_mapping = np.array(istanbul_rotated_backward_mapping , dtype=np.uint8)
i_istanbul_rotated_backward_mapping = np.array(i_istanbul_rotated_backward_mapping , dtype=np.uint8)

istanbul_vertical_chear_forward_mapping = np.array(istanbul_vertical_chear_forward_mapping , dtype=np.uint8)
istanbul_vertical_chear_backward_mapping = np.array(istanbul_vertical_chear_backward_mapping , dtype=np.uint8)
i_istanbul_vertical_chear_forward_mapping = np.array(i_istanbul_vertical_chear_forward_mapping , dtype=np.uint8)

istanbul_horizontal_shear_forward_mapping = np.array(istanbul_horizontal_shear_forward_mapping , dtype=np.uint8)
istanbul_horizontal_shear_backward_mapping = np.array(istanbul_horizontal_shear_backward_mapping , dtype=np.uint8)
i_istanbul_horizontal_shear_backward_mapping = np.array(i_istanbul_horizontal_shear_backward_mapping , dtype=np.uint8)

istanbul_zoomed_forward_mapping = np.array(istanbul_zoomed_forward_mapping , dtype=np.uint8)
istanbul_zoomed_backward_mapping = np.array(istanbul_zoomed_backward_mapping , dtype=np.uint8)
i_istanbul_zoomed_backward_mapping = np.array(i_istanbul_zoomed_backward_mapping , dtype=np.uint8)
```

```
#Writing results to "/results" folder
cv2.imwrite("results/forward_mapping/stanbul_scaled_forward_mapping.jpg", istanbul_scaled_forward_mapping)
cv2.imwrite("results/backward_mapping/istanbul_scaled_backward_mapping.jpg", istanbul_scaled_backward_mapping)
cv2.imwrite("results/backward_with_interpolation/i_istanbul_scaled_backward_mapping.jpg", i_istanbul_scaled_backward_mapping)

cv2.imwrite("results/forward_mapping/istanbul_rotated_forward_mapping.jpg", istanbul_rotated_forward_mapping)
cv2.imwrite("results/backward_mapping/istanbul_rotated_backward_mapping.jpg", istanbul_rotated_backward_mapping)
cv2.imwrite("results/backward_with_interpolation/i_istanbul_rotated_backward_mapping.jpg", i_istanbul_rotated_backward_mapping)

cv2.imwrite("results/forward_mapping/istanbul_vertical_chear_forward_mapping.jpg", istanbul_vertical_chear_forward_mapping)
cv2.imwrite("results/backward_mapping/istanbul_vertical_chear_backward_mapping.jpg", istanbul_vertical_chear_backward_mapping)
cv2.imwrite("results/backward_with_interpolation/i_istanbul_vertical_chear_forward_mapping.jpg", i_istanbul_vertical_chear_forward_mapping)

cv2.imwrite("results/forward_mapping/istanbul_horizontal_shear_forward_mapping.jpg", istanbul_horizontal_shear_forward_mapping)
cv2.imwrite("results/backward_mapping/istanbul_horizontal_shear_backward_mapping.jpg", istanbul_horizontal_shear_backward_mapping)
cv2.imwrite("results/backward_with_interpolation/i_istanbul_horizontal_shear_backward_mapping.jpg", i_istanbul_horizontal_shear_backward_mapping)

cv2.imwrite("results/forward_mapping/istanbul_zoomed_forward_mapping.jpg", istanbul_zoomed_forward_mapping)
cv2.imwrite("results/backward_mapping/istanbul_zoomed_backward_mapping.jpg", istanbul_zoomed_backward_mapping)
cv2.imwrite("results/backward_with_interpolation/i_istanbul_zoomed_backward_mapping.jpg", i_istanbul_zoomed_backward_mapping)
```

3.5.1 Forward Mapping Functions

```
#Returns scaled image with forward mapping
def scaleImage( image , scale_factor ):


    #Scale matrix
    scale_matrix = [[scale_factor, 0,0]
                    ,[0, scale_factor,0]
                    ,[0,0,1]]


    original_width = len(image[0])  # Width and height of the input
    original_height = len(image)    # image are stored

    scaled_width = int(original_width * scale_matrix[0][0])   # Width and height of the output
    scaled_height = int(original_height * scale_matrix[1][1]) # image are calculated and stored


    new_image = [] #New image is initialized as empty list

    #New image is being initialized with black pixels
    for y in range(scaled_height):
        temp_line = []
        for x in range(scaled_width):
            temp_line.append(pixel)  #Black pixel appended

        new_image.append(temp_line)



    for y in range( original_height ):
        for x in range( original_width ):

            # #Scale matrix and input image pixel coordinates are multiplied,so new pixel coordinate found
            new_coordinates = getNewCoordinates([x,y,1],scale_matrix)
            new_x = int( new_coordinates[0] )
            new_y = int( new_coordinates[1] )


            new_image[new_y][new_x] = image[y][x]

    return new_image #Return output
```

```python
def rotateImage( image , angle ):

    radian_degree = angle * math.pi / 180 #Degree converted to radian

    #Rotate matrix is calculated
    rotate_matrix = [[ math.cos(radian_degree) , -math.sin(radian_degree), 0 ],
                     [ math.sin(radian_degree) , math.cos(radian_degree) , 0 ],
                     [0,0,1]
                     ]

    #Width and height of input image are stored
    original_width = len(image[0])
    original_height = len(image)

    #Width and height of output are calculated by calculating amount of spaces at x and y axis
    width_after_rotate = int(original_width * math.cos(radian_degree) + original_height * math.sin(radian_degree) )
    height_after_rotate = int(original_height * math.cos(radian_degree) + original_width * math.sin(radian_degree))



    new_image = [] #New image is initialized as empty list

    #New image is being initialized with black pixels
    for y in range(height_after_rotate):
        temp_line = []
        for x in range(width_after_rotate):
            temp_line.append(pixel)  #Black pixel appended

        new_image.append(temp_line)


    for y in range( original_height ):
        for x in range( original_width ):

            #Pixel coordinates are calculated by matrix multiplication
            new_coordinates = getNewCoordinates([x,y,1],rotate_matrix)
            new_x = int( new_coordinates[0] )
            new_y = int( new_coordinates[1] )

            #Shifting image, so bottom-left corner stays in border of the frame
            new_x += int( math.sin(radian_degree) * original_height )

            #Set corresponding pixel
            new_image[new_y][new_x] = image[y][x]

    return new_image
```

```python
def horizontalShearImage(image, shear_factor):

    shear_matrix = [[1, shear_factor,0]
                    ,[0, 1,0]
                    ,[0,0,1]]

    original_width = len(image[0])
    original_height = len(image)

    # Calculate the output image dimensions
    sheared_image_width = int(original_width + (original_height*shear_factor))
    sheared_height = int(original_height)



    #Initialize output image with black pixels
    new_image = []
    for y in range(sheared_height):
        temp_line = []
        for x in range(sheared_image_width):
            temp_line.append(pixel)
        new_image.append(temp_line)



    for y in range( original_height ):
        for x in range( original_width ):
            #Pixel coordinates are calculated by matrix multiplication
            new_coordinates = getNewCoordinates([x,y,1],shear_matrix)
            new_x = int( new_coordinates[0] )
            new_y = int( new_coordinates[1] )

            #Set corresponding pixel
            new_image[new_y][new_x] = image[y][x]

    return new_image
```

3.5.2 Backward Mapping Functions

```
def scaleImage( image , scale_factor ):


    scale_matrix = [[scale_factor, 0,0]
                    ,[0, scale_factor,0]
                    ,[0,0,1]]


    inverse_scale_matrix = invertAffineMatrix(scale_matrix)


    original_width = len(image[0])
    original_height = len(image)


    scaled_width = int(original_width * scale_matrix[0][0])     #Calculate width and height
    scaled_height = int(original_height * scale_matrix[1][1])   #of output image



    #Generate output image with black pixels
    new_image = []
    for y in range(scaled_height):
        temp_line = []
        for x in range(scaled_width):
            temp_line.append(pixel)

        new_image.append(temp_line)




    for y in range( scaled_height ):
        for x in range( scaled_width ):
            new_coordinates = getNewCoordinates([x,y,1],inverse_scale_matrix)

            new_x = int( new_coordinates[0] )
            new_y = int( new_coordinates[1] )

            #Get output ixel from input pixel
            new_image[y][x] = image[new_y][new_x]

    return new_image
```

```
def rotateImage( image , angle ):

    radian_degree = angle * math.pi / 180

    rotate_matrix = [[ math.cos(radian_degree) , -math.sin(radian_degree), 0 ],
                     [ math.sin(radian_degree) , math.cos(radian_degree) , 0 ],
                     [0,0,1]
                     ]

    original_width = len(image[0])
    original_height = len(image)

    #Width and height of output are calculated by calculating amount of spaces at x and y axis
    width_after_scale = int(original_width * math.cos(radian_degree) + original_height * math.sin(radian_degree) )
    height_after_scale = int(original_height * math.cos(radian_degree) + original_width * math.sin(radian_degree))

    #Output image is being initialized with black pixels
    new_image = []
    for y in range( height_after_scale ):
        temp_line = []
        for x in range( width_after_scale ):
            temp_line.append(pixel)

        new_image.append( temp_line )




    for y in range( height_after_scale ):
        for x in range( -int(math.sin(radian_degree) * original_height) , width_after_scale - int(math.sin(radian_degree) * original_height) ):

            #Pixel coordinates are calculated by matrix multiplication
            new_coordinates = getNewCoordinates([x,y,1],invertAffineMatrix(rotate_matrix))
            new_x = int( new_coordinates[0] )
            new_y = int( new_coordinates[1] )

            #Get output pixel by using input pixel
            if original_width > new_x >= 0 and  original_height > new_y >= 0:
                new_image[y][x+int(math.sin(radian_degree) * original_height)] = image[new_y][new_x]

    return new_image
```

```
def horizontalShearImage( image , shear_factor ):


    shear_matrix = [[ 1,shear_factor,0],
                    [0,1,0],
                    [0,0,1]]


    original_width = len(image[0])
    original_height = len(image)

    # Calculate the output image dimensions
    height_after_scale = original_height
    width_after_scale = original_width+int(height_after_scale*shear_factor)



    #Initialize output image with black pixels
    new_image = []
    for y in range( height_after_scale ):
        temp_line = []
        for x in range( width_after_scale ):
            temp_line.append(pixel)

        new_image.append( temp_line )

    #get inverse of shear_matrix
    inv_matrix = invertAffineMatrix(shear_matrix)


    for y in range(0, height_after_scale ):
        for x in range( 0 , width_after_scale + int(height_after_scale*shear_factor) ):

            #Pixel coordinates are calculated by matrix multiplication
            new_coordinates = getNewCoordinates([x,y,1],inv_matrix)
            new_x = int( new_coordinates[0] )
            new_y = int( new_coordinates[1] )


            #Get output pixel by using input pixel
            if original_width > new_x >= 0 and  original_height > new_y >= 0 and width_after_scale > x >= 0 and height_after_scale > y >= 0:
                new_image[y][x] = image[new_y][new_x]

    return new_image
```

3.5.3 Backward Mapping Functions with bilinear interpolation

Instead of giving all functions,functions are similar with Backward Mapping but a small difference.Nested loops at the end to get pixels are different to do bilinear interpolation.So only giving nested loop structure should be enough.

```
    new_image = np.empty((scaled_height, scaled_width, 3), dtype=np.float32)

    for y in range(scaled_height):
        for x in range(scaled_width):
            for channel in range(3):
                # Calculate the coordinates of the four nearest pixels in the original image.
                new_coordinates = getNewCoordinates([x, y, 1], inverse_scale_matrix)
                new_x0 = int(new_coordinates[0])
                new_y0 = int(new_coordinates[1])
                new_x1 = min(new_x0 + 1, original_width - 1)
                new_y1 = min(new_y0 + 1, original_height - 1)


                # Calculate the weights for the four nearest pixels.
                weight_x0 = 1 - (new_coordinates[0] - new_x0)
                weight_x1 = new_coordinates[0] - new_x0
                weight_y0 = 1 - (new_coordinates[1] - new_y0)
                weight_y1 = new_coordinates[1] - new_y0


                if 0 <= new_x0 < original_width and 0 <= new_x1 < original_width and 0 <= new_y0 < original_width and 0 <= new_y1 < original_width:
                    # Calculate the interpolated pixel value.
                    interpolated_pixel = (
                        float(image[new_y0][new_x0][channel]) * weight_x0 * weight_y0 +
                        float(image[new_y0][new_x1][channel]) * weight_x1 * weight_y0 +
                        float(image[new_y1][new_x0][channel]) * weight_x0 * weight_y1 +
                        float(image[new_y1][new_x1][channel]) * weight_x1 * weight_y1
                    )

                    new_image[y][x][channel] = interpolated_pixel
```

3.5.4 Common same functions for all 3 methods

```
def zoomImage(image, zoom_factor):
    width = len(image[0])
    height = len(image)


    #finding x and y coordinates of the top left of scaled image's zoomed part
    x_coordinate_of_zoomed_image = int((width / 2) - ((width / zoom_factor) / 2))
    y_coordinate_of_zoomed_image = int((height / 2) - ((height / zoom_factor) / 2))


    scaled_image = scaleImage(image,zoom_factor)


    #new blank image with the same width and height with input
    new_image = []
    for y in range(height):
        temp_line = []
        for x in range(width):
            temp_line.append(pixel)
        new_image.append(temp_line)


    #copying middle part of the scaled image to new image
    for y in range(height):
        for x in range(width):
            new_image[y][x] = scaled_image[y+y_coordinate_of_zoomed_image][x+x_coordinate_of_zoomed_image]

    return new_image
```

```
def invertAffineMatrix(affineMatrix):

    c1 = affineMatrix[1][1] * affineMatrix[2][2] - affineMatrix[1][2] * affineMatrix[2][1]
    c2 = -(affineMatrix[1][0] * affineMatrix[2][2] - affineMatrix[2][0] * affineMatrix[1][2])
    c3 = affineMatrix[1][0] * affineMatrix[2][1] - affineMatrix[1][1] * affineMatrix[2][0]

    c4 = -(affineMatrix[0][1] * affineMatrix[2][2] - affineMatrix[0][2] * affineMatrix[2][1])
    c5 = affineMatrix[0][0] * affineMatrix[2][2] - affineMatrix[2][0] * affineMatrix[0][2]
    c6 = -(affineMatrix[0][0] * affineMatrix[2][1] - affineMatrix[2][0] * affineMatrix[0][1])

    c7 = affineMatrix[0][1] * affineMatrix[1][2] - affineMatrix[1][1] * affineMatrix[0][2]
    c8 = -(affineMatrix[0][0] * affineMatrix[1][2] - affineMatrix[1][0] * affineMatrix[0][2])
    c9 = affineMatrix[0][0] * affineMatrix[1][1] - affineMatrix[1][0] * affineMatrix[0][1]


    determinant = (affineMatrix[0][0]*c1) + (affineMatrix[0][1]*c2)+ (affineMatrix[0][2]*c3)

    adjecency = [[c1,c4,c7],[c2,c5,c8],[c3,c6,c9]]

    inverseAffineMatrix = [[0,0,0],[0,0,0],[0,0,0]]

    if determinant == 0:
        raise ValueError("The affine matrix is not invertible.")


    for i in range(3):
        for j in range(3):
            inverseAffineMatrix[i][j] = adjecency[i][j]/determinant
    return inverseAffineMatrix


#Multiply coordinates matrix and affineMatrix
def getNewCoordinates( coordinates,affineMatrix ):

    first_row = affineMatrix[0][0]*coordinates[0] + affineMatrix[0][1]*coordinates[1] + affineMatrix[0][2]*coordinates[2]
    sec_row = affineMatrix[1][0]*coordinates[0] + affineMatrix[1][1]*coordinates[1] + affineMatrix[1][2]*coordinates[2]
    third_row = affineMatrix[2][0]*coordinates[0] + affineMatrix[2][1]*coordinates[1] + affineMatrix[2][2]*coordinates[2]

    new_coordinates = [ first_row
                      , sec_row
                      , third_row ]

    return new_coordinates
```

3.6 Conclusion

Results show that forward mapping is fastest but worst result with blank pixels at the result.Backward mapping without interpolation generates better results but it is more expensive in aspect of computation.Best results are obtained from backward mapping with interpolation with most computation power need but interpolation results output with less aliasing and more blur.At the end,this project compares three methods of affine transformations.