Gebze Technical University
CSE 464 - Homework 1 Report

Sena Özbelen
1901042601
10 November 2023

1. Introduction

This project is about the affine transformation by transforming the spatial coordinates of the image and then assigning the intensity to the corresponding coordinates.

Transformation of spatial coordinates is done by using affine matrices. Assigning the intensity of corresponding coordinates is done by forward or backward mapping. For backward mapping, interpolation can also be used.

This project implements those affine transformations : scaling, shearing, rotating. Zooming is also implemented by using scaling.

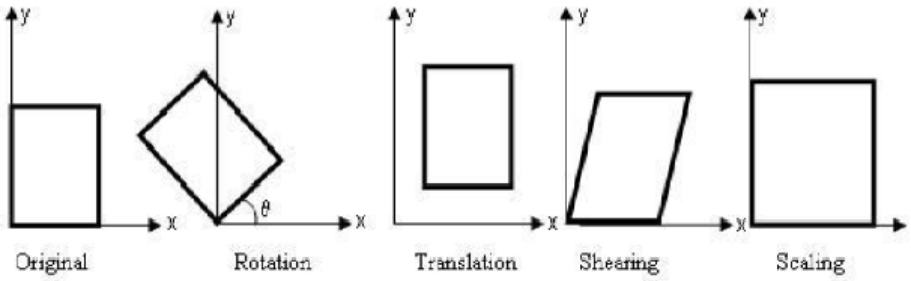


Fig. 1. Basic affine transform operations

2. Implementation Details

The first step is transforming the spatial coordinates of the image. To do that, affine matrices are used.

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

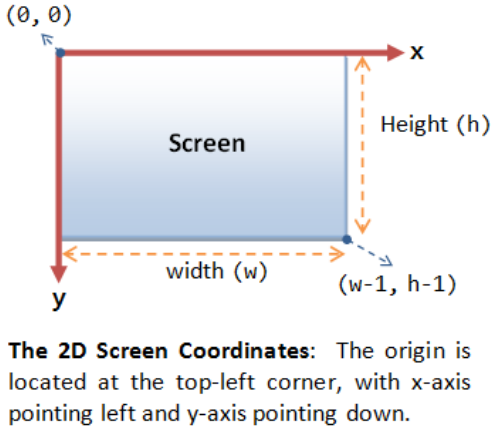
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \lambda_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Shear-Y

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

Fig. 2. Affine Matrices

Figure 2 shows the affine matrices which are used in affine transformation. This project utilizes scale, shear-x, shear-y and rotation matrices.



The 2D Screen Coordinates: The origin is located at the top-left corner, with x-axis pointing left and y-axis pointing down.

Fig. 3. The Coordinate System

Figure 3 shows the representation of the coordinate system in which the image is represented. The image is a matrix and the row of the matrix represents the y-axis, the column of the matrix represents the x-axis.

Firstly, the image is converted to a python list. To do that, it is read by OpenCV “imread” function and converted to a list.

```python
# load the image
image = cv2.imread('istanbul.jpg')

# convert the image to an array
if image is not None:
    image_array = image.tolist()
else:
    print("Image not available")
    exit()
```

For each affine transformation, their functions are implemented separately. For scaling, rotating and shearing, it starts with defining the matrix of the corresponding transformation.

```python
# Define the affine matrix for scaling
scale_affine_matrix = [[scale_x, 0, 0], [0, scale_y, 0], [0, 0, 1]]
```

After that, it calculates the original dimensions of the image by using len function and the transformed dimensions. Scaling makes the dimensions double. For shearing, the right-down corner's coordinates are calculated by using affine matrix of shearing. For rotating, by using geometry, the new height equals to old_height*cos(degree) + old_width*sin(degree) and the new width equals to old_width*cos(degree) + old_height*sin(degree)

```
# Calculate the new dimensions using old dimensions
old_width = len(image[0])
old_height = len(image)
new_width = int(old_width * scale_x)
new_height = int(old_height * scale_y)
```

Then, it creates a new matrix for the transformed image using the calculated dimensions.

```
# Create a new matrix for the new image
new_image = [[[0 for k in range(3)] for j in range(new_width)] for i in range(new_height)]
```

For forward mapping, the input image is scanned and for each pixel, the new coordinate is calculated. The calculation is done by matrix-vector multiplication. The vector is the corresponding pixel in the input image and the matrix is the affine matrix. The result of this multiplication is the new coordinate of the pixel on the output image. After the calculation of new coordinate, the intensity is directly assigned to the new coordinate. For rotating, it adds the value of "old_height*sin(degree)" to transformed_x to show the whole image.

```
for y in range(old_height):
    for x in range(old_width):
        # Calculate the new coordinates for the current coordinates
        point = [x, y, 1]
        result = matrix_vector_multiplication(scale_affine_matrix, point)
        transformed_x = int(result[0])
        transformed_y = int(result[1])

        # Use forward mapping to transform the intensity
        if transformed_x >= 0 and transformed_x < new_width and transformed_y >= 0 and transformed_y < new_height:
            new_image[transformed_y][transformed_x][0] = image[y][x][0]
            new_image[transformed_y][transformed_x][1] = image[y][x][1]
            new_image[transformed_y][transformed_x][2] = image[y][x][2]
```

For backward mapping, instead of scanning the input, the output is scanned and the inverse transformation is applied. To do that, the inverse of the affine matrix is needed. When the inverse affine matrix is multiplied by the output coordinate, it produces the input coordinate for the corresponding pixel. Again, for rotating, if it starts scanning from 0, we cannot see the whole image since (0,0) is equal to (0,0) in the input image

```
for y in range(new_height):
    for x in range(new_width):
    # Calculate the new coordinates for the current coordinates
        point = [x, y, 1]
        inverse_matrix_result = inverse_matrix(scale_affine_matrix)
        result = matrix_vector_multiplication(inverse_matrix_result, point)
        transformed_x = result[0]
        transformed_y = result[1]
```

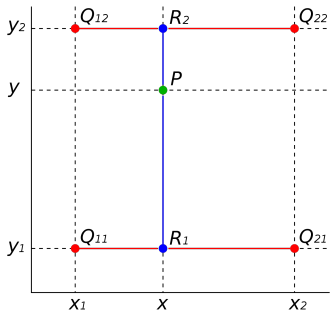
too and the rotated image has some parts in negative side of x-axis. Therefore, “old_height*sin(degree)” value is subtracted from the x value. “old_height*sin(degree)” value is the length of the part in negative side.

```
point = [x-int(old_height*math.sin(rotate_degree)), y, 1]
```

If interpolation is not used, then the intensity is directly assigned to the output image.

```
# Use backward mapping to transform the intensity
if interpolation == False:
    transformed_x = int(transformed_x)
    transformed_y = int(transformed_y)
    if transformed_x >= 0 and transformed_x < old_width and transformed_y >= 0 and transformed_y < old_height:
        new_image[y][x][0] = image[transformed_y][transformed_x][0]
        new_image[y][x][1] = image[transformed_y][transformed_x][1]
        new_image[y][x][2] = image[transformed_y][transformed_x][2]
```

If interpolation is used, the neighbor pixels’ intensity values are used. This project uses bilinear interpolation so only 4 neighbor pixels are considered.



$$f(x,y) = \frac{y_2 - y}{y_2 - y_1} f(x, y_1) + \frac{y - y_1}{y_2 - y_1} f(x, y_2)$$
$$= \frac{y_2 - y}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21}) \right) + \frac{y - y_1}{y_2 - y_1} \left(\frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22}) \right)$$

Fig. 4. Bilinear Interpolation

The transformed coordinates are sent to the interpolation calculation and the returned result is assigned to the pixel's intensity.

```
else:
    if transformed_x >= 0 and transformed_x < old_width and transformed_y >= 0 and transformed_y < old_height:
        results = interpolation_calc(transformed_x, transformed_y, image)
        new_image[y][x][0] = results[0]
        new_image[y][x][1] = results[1]
        new_image[y][x][2] = results[2]
```

For zooming, the program uses the scaling feature and it scales the image by a factor of 1.6. Since the image has to preserve the original dimensions, it is cropped from all the sides.

```
# Calculate the differences between dimensions and divide them by 2
# This is used to crop the image
width_diff = new_width - old_width
height_diff = new_height - old_height

width_diff = width_diff // 2
height_diff = height_diff // 2

for y in range(old_height):
    for x in range(old_width):
        # Crop the image by removing the outer parts from all sides equally
        original_dimension_image[y][x][0] = new_image_array[height_diff + y][width_diff + x][0]
        original_dimension_image[y][x][1] = new_image_array[height_diff + y][width_diff + x][1]
        original_dimension_image[y][x][2] = new_image_array[height_diff + y][width_diff + x][2]
```

3. Results

-Scaling

Forward mapping method with 0.5 and 2 for both x and y-axis is applied.

Fig. 5. Scaling with 0.5

Fig. 6. Scaling with 2

Scaling with 2 has a lot of missing pixels since the program scans the input and the output is larger than the input. Scaling with 0.5 does not have a problem with forward mapping since the output is smaller.

Fig. 7. Without interpolation

Fig. 8. With interpolation

Figure 7 and Figure 8 are the examples for scaling with 2 but the program uses backward mapping. The outputs do not have missing pixels anymore because the program scans the output image so every pixel gets an intensity value. Normally, these images don't have a huge difference but the details show that interpolation makes the image smoother.

-Shearing

Fig. 9.

Fig. 10.

Figure 9 and Figure 10 shows shearing on x-axis with 2 by using forward mapping and backward mapping. There is no difference between these two.

Fig. 11.

Figure 11 shows the same picture using interpolation. Again, there is no difference between these three images.

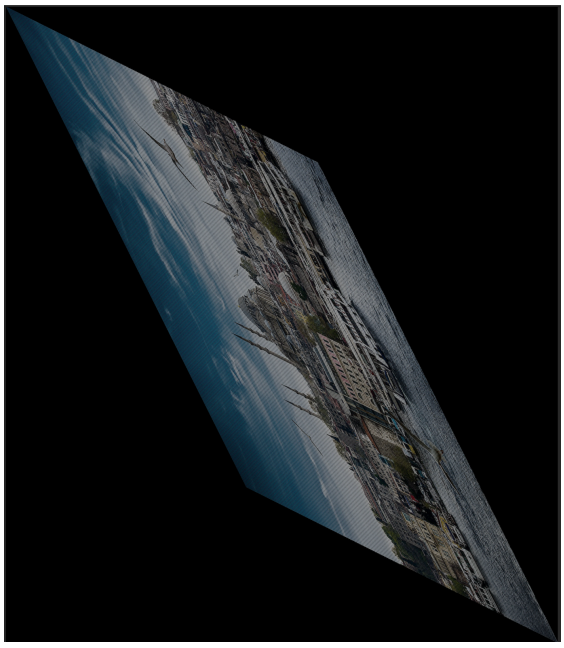


Fig. 12.

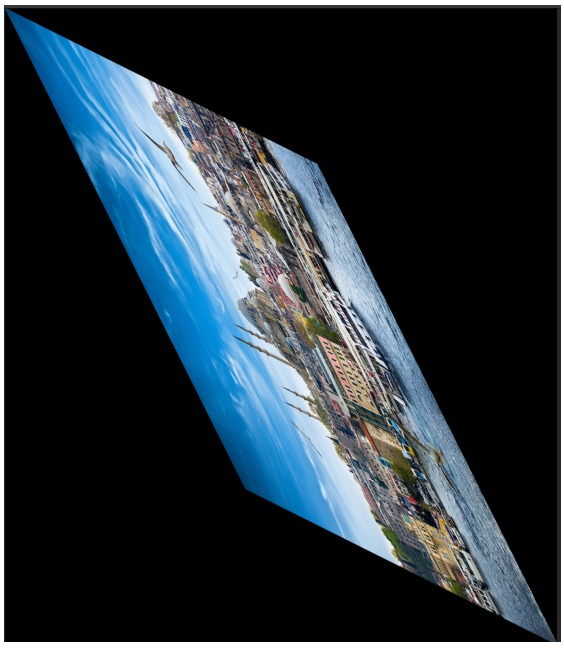


Fig. 13.

We can see the difference when shearing is applied to both x and y-axis. Figure 12 uses forward mapping and it gives a similar result to scaling.

Backward mapping in Figure 13, again, can handle all the pixels. However, lower factors in shearing does not have a problem of missing pixels with forward mapping for this case. It depends on the factor value.

-Rotating

Fig. 14.

Fig. 15.

Fig. 16.

Figure 14 shows forward mapping result of rotating 30 degrees. There are some missing pixels. Figure 15 shows backward mapping and we can see that there is no missing pixel. Figure 16 additionally uses interpolation. The only difference between Figure 15 an 16 is that interpolation makes Figure 16 much smoother.

-Zooming

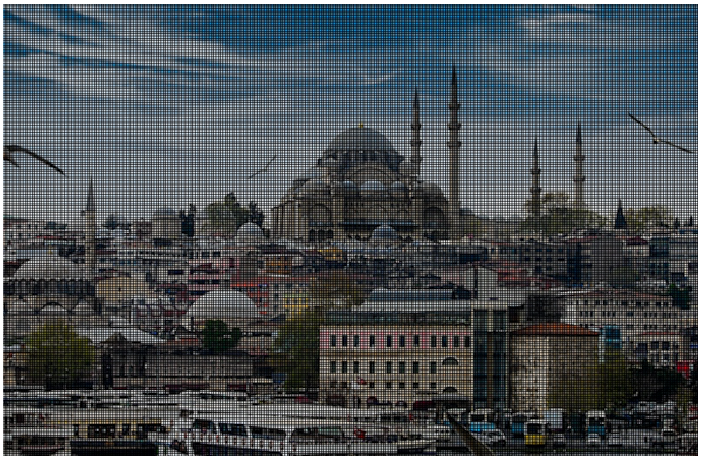


Fig. 17.

Fig. 18.

Fig. 19.

Zooming uses scaling. Therefore it gives the same results as scaling gives. Zooming uses scaling for both x and y-axis with a factor of 1.6 and then it crops the image and preserves the dimensions. Figure 17 shows a forward mapping result and the missing pixels can be seen easily on this image since the output image size is bigger than the original one. Figure 18 shows the backward mapping and Figure 19 shows the backward mapping with interpolation. Again, with interpolation, the image becomes a smooth, complete image.

-The Main Result

The results show that forward mapping cannot handle all the pixels on the output image so there are some missing pixels in some cases. Backward

mapping gives a better solution by scanning the output image so that it can handle all the pixels and the result is more accurate. Using interpolation makes the image smoother. It removes the sharpness coming from the direct assignment of intensity values.

-Additional Code Segments

Interpolation calculation for one color matrix.

```
# Find the corners
x1 = int(math.floor(x))
y1 = int(math.floor(y))
x2 = min(math.ceil(x), len(image[0])-1)
y2 = min(math.ceil(y), len(image)-1)

# Apply for all z dimensions (for 3)

# Get the neighbor pixels' values
q11, q21, q12, q22 = image[y1][x1][0], image[y1][x2][0], image[y2][x1][0], image[y2][x2][0]

# Get the final intensity regarding the neighbor pixels
if (x2 == x1) and (y2 == y1): # If there is no need for neighbors (integers)
    P_0 = image[int(y)][int(x)][0]
elif (x2 == x1):  # If there are only two neighbors vertically
    P_0 =  image[y1][int(x)][0] * (y2-y) / (y2-y1) + image[y2][int(x)][0] * (y-y1) / (y2-y1)
elif (y2 == y1):  # If there are only two neighbors horizontally
    P_0 =  image[int(y)][x1][0] * (x2-x) / (x2-x1) + image[int(y)][x2][0] * (x-x1) / (x2-x1)
else: # If all neighbors are available
    R1_0 =  q11 * (x2-x) / (x2-x1) + q21 * (x-x1) / (x2-x1)
    R2_0 = q12 * (x2-x) / (x2-x1) + q22 * (x-x1) / (x2-x1)
    P_0 = R1_0 * (y2-y) / (y2-y1) + R2_0 * (y-y1) / (y2-y1)
```

Getting the inverse of a matrix

```
def inverse_matrix(matrix):

    a, b, c = matrix[0]
    d, e, f = matrix[1]
    g, h, i = matrix[2]

    # Calculate the determinant
    det = a * (e * i - f * h) - b * (d * i - f * g) + c * (d * h - e * g)

    # Check whether the determinant is zero. If it is zero, this matrix does not have an inverse version
    if det == 0:
        print("This matrix does not have an inverse version")
    else:
        a, b, c = matrix[0]
        d, e, f = matrix[1]
        g, h, i = matrix[2]

        inverse_matrix = [[(e * i - f * h) / det, (c * h - b * i) / det, (b * f - c * e) / det],
                          [(f * g - d * i) / det, (a * i - c * g) / det, (c * d - a * f) / det],
                          [(d * h - e * g) / det, (b * g - a * h) / det, (a * e - b * d) / det]]


    return inverse_matrix
```

Matrix vector multiplication

```
def matrix_vector_multiplication(matrix, vector):
    result = [0] * len(matrix)

    # Perform matrix-vector multiplication
    for i in range(len(matrix)):
        for j in range(len(vector)):
            result[i] += matrix[i][j] * vector[j]

    return result
```

4. References

- [1] https://www.semanticscholar.org/paper/Acceleration-of-Affine-Transform-for-Multiplane-in-Mondal-Biswal/5096e06918b21b2163a59d80f8fe226b05de08d9
- [2] https://subscription.packtpub.com/book/data/9781789537147/1/ch01lvl1sec04/applying-affine-transformation
- [3] https://www3.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
- [4] https://en.wikipedia.org/wiki/Bilinear_interpolation