CSE464-DIGITAL IMAGE PROCESSING

GENERATING AFFINE TRANSFORMATION-HW1

Adem Murat ÖZDEMİR-200104004110

Abstract

This study explores affine transformations and mapping techniques as applied to a digital image to investigate the effects of various transformation matrices and mapping methods. The primary objective is to implement scaling, rotation, horizontal shear, and zoom transformations in sequence using an affine transformation matrix. The transformed images are analyzed through both forward and backward mapping techniques. Additionally, backward mapping is examined with and without bilinear interpolation to assess its impact on image quality and pixel continuity. Results demonstrate how each transformation matrix affects image geometry and detail, as well as how interpolation methods enhance or diminish the smoothness of the resulting images. This study provides insights into the practical implementation of image transformations in digital processing and the advantages of interpolation for achieving visual accuracy.

Keywords: Image Processing, Affine Transformation, Forward Mapping, Backward Mapping, Bilinear Interpolation, Scaling, Rotation, Shearing, Zoom

Introduction

Affine transformations are essential in digital image processing, allowing images to be scaled, rotated, sheared, and zoomed while preserving linearity and geometry. These transformations have broad applications in fields such as computer vision and graphics, enabling precise adjustments to image orientation and size. This study examines the impact of applying sequential affine transformations on an image, utilizing both forward and backward mapping techniques. To further enhance image quality, bilinear interpolation is used in backward mapping, reducing pixelation effects.

An affine transformation is a linear mapping method that preserves points, straight lines, and planes in an image while allowing transformations such as scaling, rotation, shearing, and reflection. These transformations maintain the geometric structure, keeping lines parallel and proportions consistent, making them fundamental for image manipulation in computer graphics and vision. An affine transformation can be represented by a 3x3 matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} & b_0 \\ A_{10} & A_{11} & b_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Methodology

This study involves a series of affine transformations applied to a digital image using translation, scaling, rotation, horizontal shearing, and zooming. The methods used are as follows:

- **Scaling**: Scaling alters the size of the image by expanding or shrinking it proportionally along the x and y axes. This is achieved through a scaling matrix that defines the scaling factors for each axis, preserving the image's geometry.
- **Rotation**: Rotation involves rotating the image around a defined central point (here, the center of the image) by a specified angle. This transformation changes the orientation of the image while maintaining its structural integrity.
- **Shearing**: Shearing skews the image in the x or y direction by a given factor, distorting the shape while keeping lines straight. In this study, horizontal shear is applied to introduce a perspective effect.
- **Zooming**: Zooming scales the image around its center, enlarging or reducing it to a specified factor. This zoom operation is a type of scaling but is applied as an additional transformation after other adjustments.
- **Forward Mapping**: Forward mapping involves mapping each pixel from the original image to a new location in the transformed image according to the transformation matrix. This method can leave gaps or overlaps in the transformed image, as it does not cover all target pixels directly.
- **Backward Mapping**: In backward mapping, each pixel in the target image is mapped back to a corresponding location in the original image using the inverse transformation matrix. This approach minimizes gaps and overlaps, as each target pixel is calculated based on its position in the original image.
- **Bilinear Interpolation**: Bilinear interpolation is used within backward mapping to smooth out transitions between pixels. This method estimates pixel values by considering the four nearest neighbors, resulting in a more visually continuous image by reducing pixelation and improving image quality.

These transformations and mappings are combined sequentially to create a single affine transformation matrix, applied to the image to analyze the effects on structure and quality across different mapping methods. The resulting images are then compared to evaluate the effectiveness of each method in achieving smooth and accurate transformations.

Implementation

This study was implemented in Python due to its versatility and the extensive libraries available for image processing and matrix operations. The main tools and functions used in the implementation are as follows:

1. **Programming Language: Python**
   Python was chosen for its readability and the support of various image processing and scientific libraries, making it well-suited for affine transformations and mapping techniques.
2. **Libraries and Tools**
   - **OpenCV (cv2)**: OpenCV, a robust computer vision library, was used for reading, processing, and displaying images. It provides fundamental functions for image manipulation, making it ideal for implementing affine transformations.
   - **NumPy**: NumPy was used for matrix operations, such as creating and multiplying transformation matrices. Its efficient handling of arrays and mathematical functions simplifies complex transformations.
3. **Functions and Implementation Steps**
   - **Transformation Matrices**
     Custom functions were developed to create the transformation matrices for scaling, rotation, shearing, and zooming. Each transformation matrix was applied sequentially to form a single affine transformation matrix, preserving the intended effects of each transformation in order.
     - def scale_matrix(scale_x, scale_y): Generates a 3x3 matrix for scaling the image along x and y axes.
     - def rotate_matrix(angle_degrees, center_x, center_y): Creates a rotation matrix to rotate the image by a specified angle around the center point.
     - def horizontal_shear_matrix(shear_factor): Defines a horizontal shear matrix to skew the image along the x-axis by a specified factor.
     - def zoom_matrix(zoom_factor): Constructs a zoom matrix to enlarge or reduce the image around its center.
     - def affine_transformation_matrix(scale_x, scale_y, angle, shear_factor, zoom_factor, center_x, center_y) function generates a composite affine transformation matrix by sequentially applying scaling, rotation, shearing, and zooming transformations. Each transformation matrix—scale_matrix, rotate_matrix, horizontal_shear_matrix, and zoom_matrix—is calculated based on input parameters and multiplied in the specified order (Scale → Rotate → Shear → Zoom). The final affine matrix is returned, ready to apply the combined transformations to an image.
   - **Forward Mapping**
     A forward mapping function was created to apply the affine transformation to each pixel in the original image and map it to the new location in the transformed image.

However, this method can result in empty spaces or overlapping pixels in the transformed image, as it does not guarantee coverage of every pixel in the output.

- **Backward Mapping**
  In backward mapping, each pixel in the output image is mapped back to its source location in the original image using the inverse affine transformation matrix. This approach helps avoid gaps or overlaps in the final image and ensures a more accurate transformation.
- **Bilinear Interpolation**
  Bilinear interpolation was implemented to improve the smoothness and visual quality of the transformed image when using backward mapping. This function calculates the target pixel's value by interpolating between the values of the four nearest neighboring pixels, reducing pixelation and providing a more natural look.

Results

The transformation factors are set as:

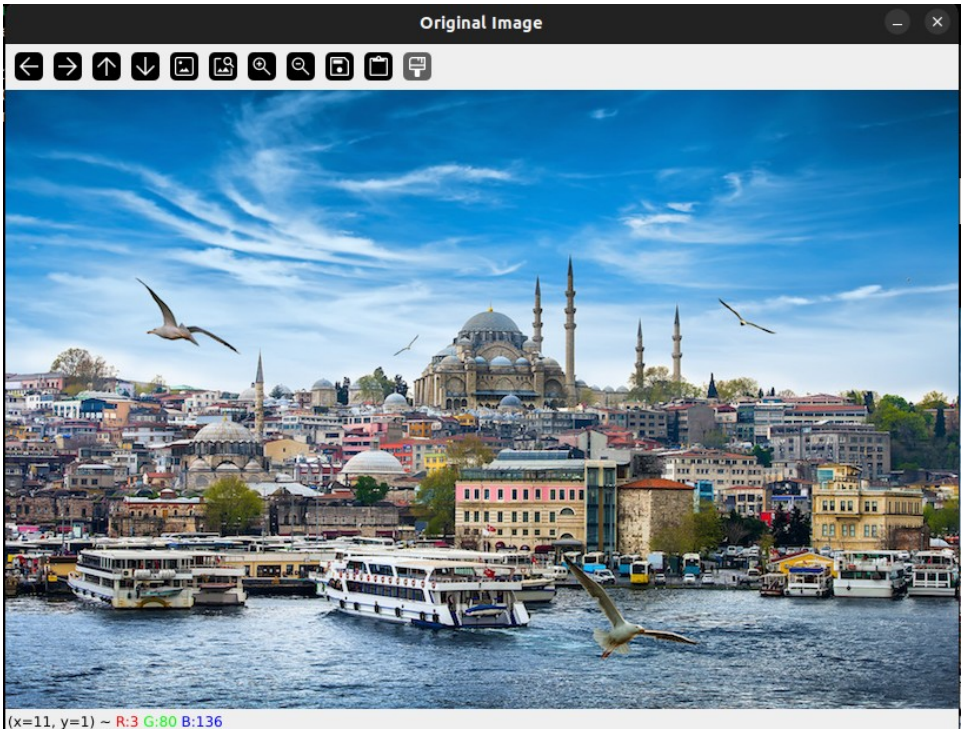
scale_factor = 1.0

rotation_angle = 60

shear = 1.4

zooming = 1.4

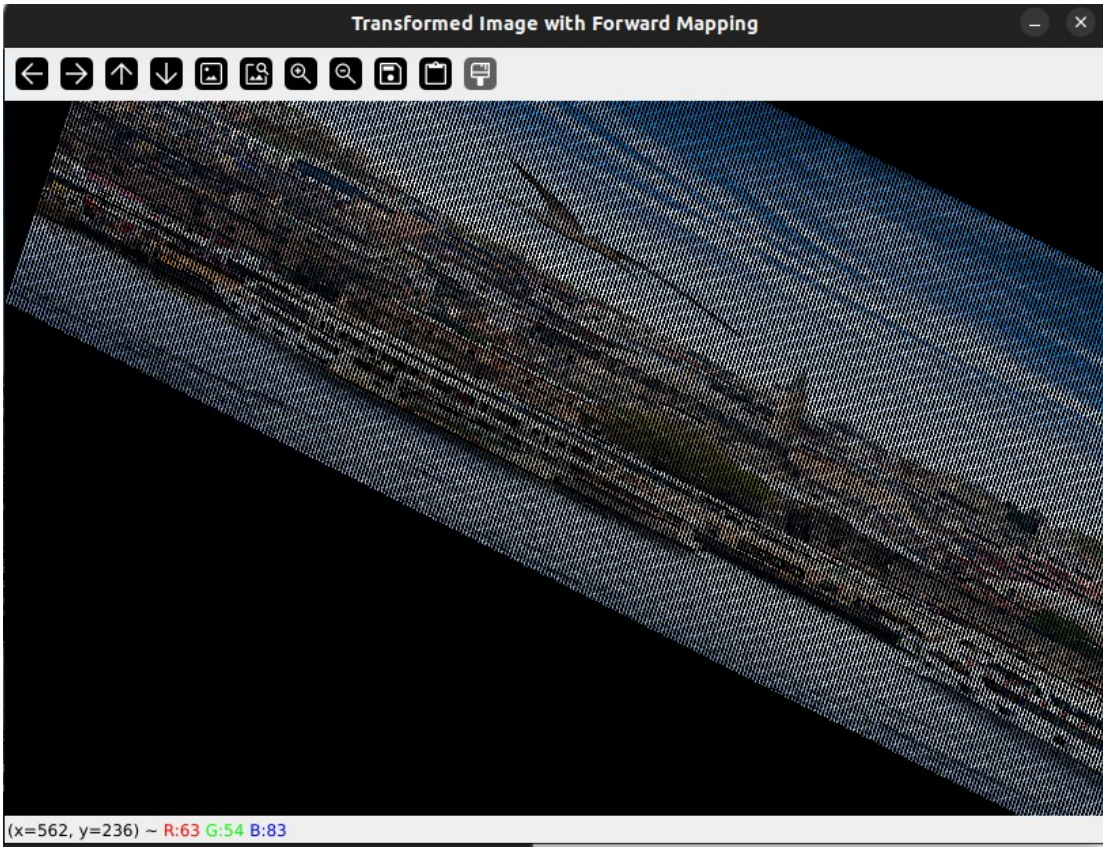
Original Image



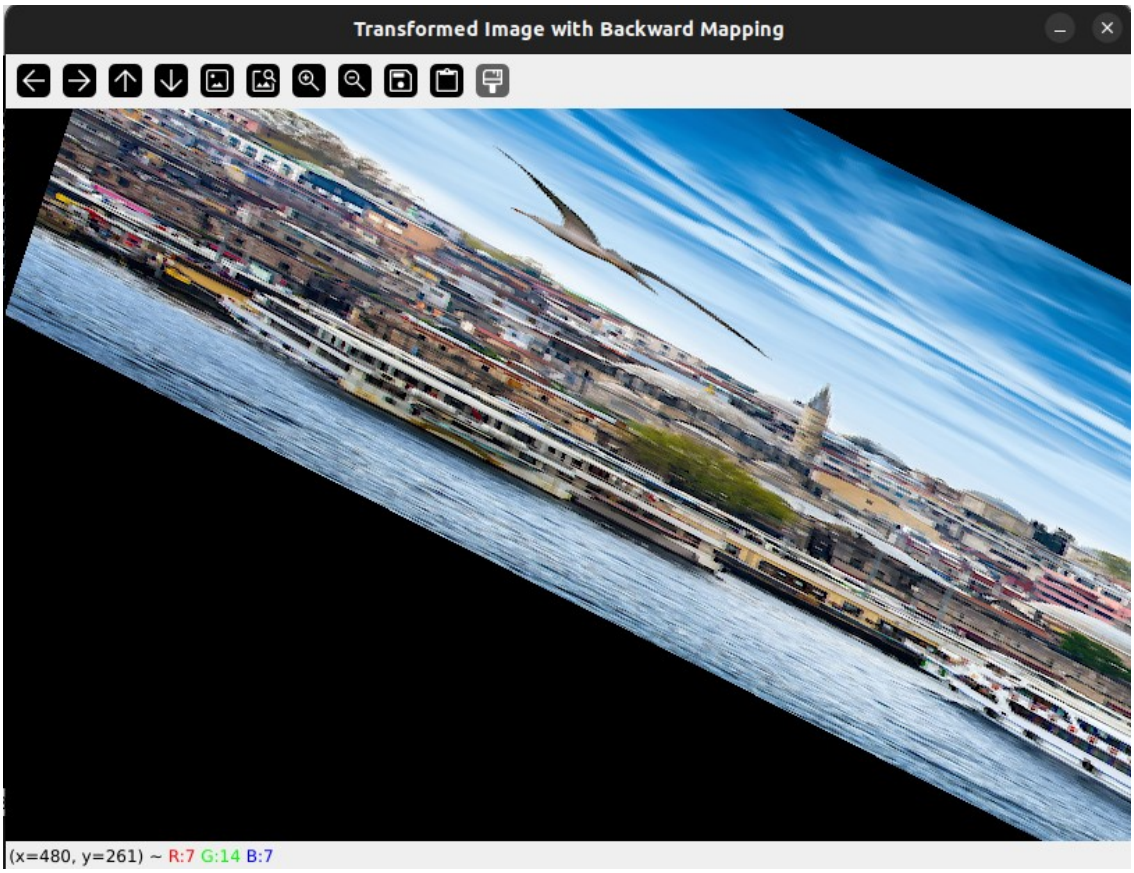
Affine Transformation Matrix :

```
ademmurat@ademmurat-GL553VD: ~/Desktop/imageHW1
ademmurat@ademmurat-GL553VD:~/Desktop/imageHW1$ python3 affine_transformation.py
Affine Transformation Matrix:
 [[   2.3974097    -0.23243555  123.32827   ]
 [   1.2124355     0.7        -219.2403    ]
 [   0.            0.            1.        ]]
```

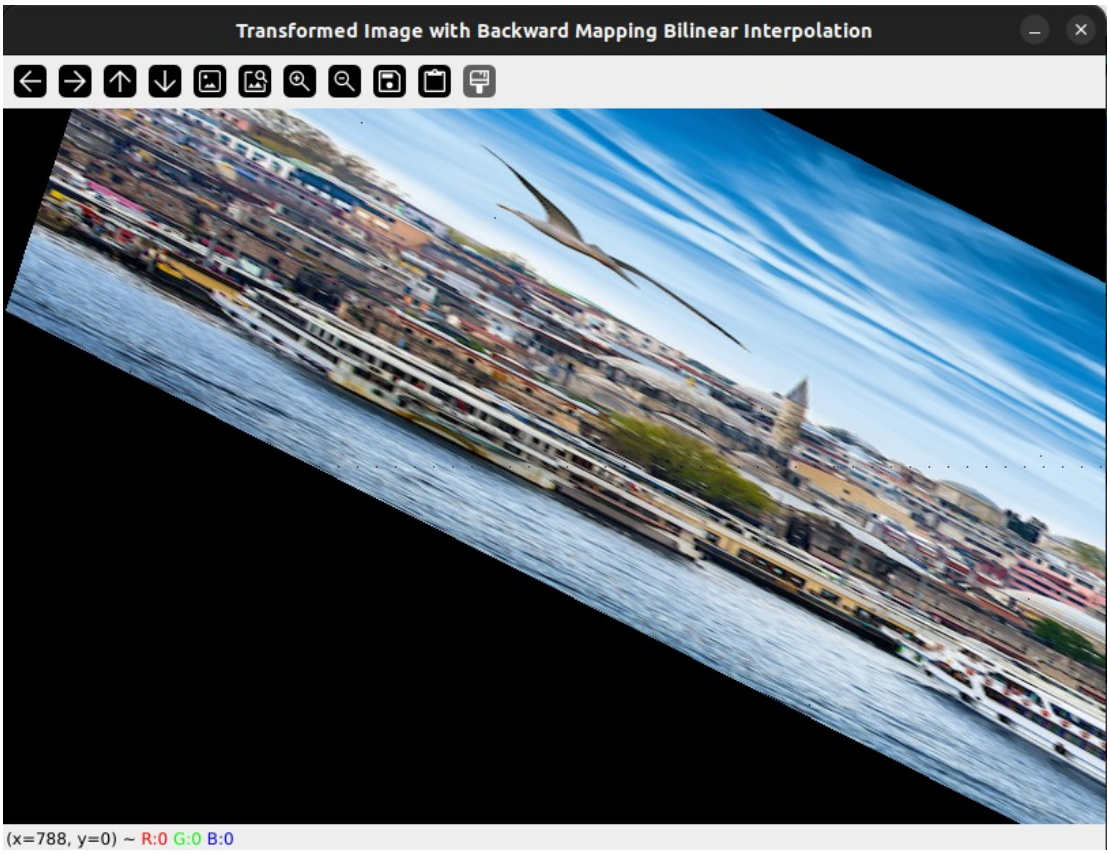
Forward Mapping with Affine Transformation Matrix



Backward Mapping with Affine Transformation Matrix :



Backward Mapping Using Bilinear Interpolation with Affine Transformation Matrix :



Compared Results :

Forward Mapping with Affine Transformation

In my implementation of forward mapping with affine transformation, I noticed that while this method straightforwardly maps each pixel from the original image to a new position, it leads to some significant issues. Specifically, there are often gaps or empty spaces in the transformed image. This happens because not every pixel from the original image finds a corresponding position in the output, resulting in areas that lack color or detail. Although forward mapping is computationally simpler, I found that the visual quality of the resulting images suffers, showing noticeable discontinuities and artifacts.

Backward Mapping with Affine Transformation Matrix

When I switched to backward mapping, I could see a marked improvement in the output. This method calculates where each pixel in the target image should come from in the original image, which means that every pixel in the output is derived from the original. This significantly reduces gaps and overlaps, leading to a much more coherent and continuous image. However, I did notice some artifacts in regions with high distortion, but overall, backward mapping provided a more accurate representation of the transformed image compared to forward mapping.

Backward Mapping by Using Bilinear Interpolation

Finally, using bilinear interpolation in backward mapping took the image quality to another level. This method estimates pixel values based on the four nearest neighbors, which really helps to smooth out the transitions. As a result, the transformed images look much better with less pixelation and fewer noticeable artifacts. I found that bilinear interpolation offered the best visual quality among the three methods, making it the preferred choice when I aimed for high-quality results in my image processing tasks.

Appendix

Scale , Rotate , Horizontal Shear , Zoom and Affine Transformations Functions

```python
# Scale Matrix
def scale_matrix(scale_x, scale_y):
    return np.array([[scale_x, 0, 0],
                     [0, scale_y, 0],
                     [0, 0, 1]], dtype=np.float32)
```

```python
# Rotate Matrix
def rotate_matrix(angle_degrees, center_x, center_y):
    angle_radians = np.radians(angle_degrees)
    cos_a = np.cos(angle_radians)
    sin_a = np.sin(angle_radians)

    return np.array([[cos_a, -sin_a, center_x * (1 - cos_a) + center_y * sin_a],
                     [sin_a, cos_a, center_y * (1 - cos_a) - center_x * sin_a],
                     [0, 0, 1]], dtype=np.float32)
```

```python
# Horizontal Shear Matrix
def horizontal_shear_matrix(shear_factor):
    return np.array([[1, shear_factor, 0],
                     [0, 1, 0],
                     [0, 0, 1]], dtype=np.float32)
```

```python
# Zoom Matrix
def zoom_matrix(zoom_factor):
    return np.array([[zoom_factor, 0, 0],
                     [0, zoom_factor, 0],
                     [0, 0, 1]], dtype=np.float32)
```

```python
# Combine Transformations into Affine Transformation Matrix
def affine_transformation_matrix(scale_x, scale_y, angle, shear_factor, zoom_factor, center_x, center_y):
    scale_mat = scale_matrix(scale_x, scale_y)
    rotate_mat = rotate_matrix(angle, center_x, center_y)
    shear_mat = horizontal_shear_matrix(shear_factor)
    zoom_mat = zoom_matrix(zoom_factor)

    # Order of matrices: Scale -> Rotate -> Shear -> Zoom
    affine_matrix = shear_mat @ rotate_mat @ scale_mat @ zoom_mat
    return affine_matrix
```

Mapping Functions

```python
# Forward Mapping
def forward_mapping(image, transformation_matrix):
    height, width = image.shape[:2]
    transformed_image = np.zeros_like(image)

    for y in range(height):
        for x in range(width):
            original_pos = np.array([x, y, 1], dtype=np.float32)
            new_pos = transformation_matrix @ original_pos
            new_x, new_y = int(new_pos[0]), int(new_pos[1])

            if 0 <= new_x < width and 0 <= new_y < height:
                transformed_image[new_y, new_x] = image[y, x]

    return transformed_image
```

```python
def backward_mapping(image, transformation_matrix):
    height, width = image.shape[:2]

    transformed_image = np.zeros_like(image)

    for y in range(height):
        for x in range(width):
            new_pos = np.array([x, y, 1], dtype=np.float32)

            original_pos = np.linalg.inv(transformation_matrix) @ new_pos
            original_x, original_y = int(original_pos[0]), int(original_pos[1])

            if 0 <= original_x < width and 0 <= original_y < height:
                transformed_image[y, x] = image[original_y, original_x]

    return transformed_image
```

```python
def bilinear_interpolation(image, x, y):
    x1 = int(np.floor(x))
    x2 = int(np.ceil(x))
    y1 = int(np.floor(y))
    y2 = int(np.ceil(y))

    height, width = image.shape[:2]

    if x1 < 0 or x2 < 0 or y1 < 0 or x1 >= width or x2 >= width or y1 >= height or y2 >= height:
        return [0, 0, 0]

    I11 = image[y1, x1]
    I21 = image[y1, x2]
    I12 = image[y2, x1]
    I22 = image[y2, x2]
    #Interpolation calculations
    f_x1 = (x2 - x) * I11 + (x - x1) * I21
    f_x2 = (x2 - x) * I12 + (x - x1) * I22
    value = (y2 - y) * f_x1 + (y - y1) * f_x2

    return value

def backward_mapping_bilinear(image, transformation_matrix):
    height, width = image.shape[:2]

    transformed_image = np.zeros_like(image)

    for y in range(height):
        for x in range(width):
            new_pos = np.array([x, y, 1], dtype=np.float32)

            original_pos = np.linalg.inv(transformation_matrix) @ new_pos
            original_x, original_y = original_pos[0], original_pos[1]

            transformed_image[y, x] = bilinear_interpolation(image, original_x, original_y)

    return transformed_image
```

Conclusion

In this study, I investigated affine transformations on a digital image using forward and backward mapping methods, including bilinear interpolation. I found that while forward mapping is simpler, it often results in visual artifacts and gaps. Backward mapping, however, significantly improved the output by providing a more coherent image with fewer discontinuities. Incorporating bilinear interpolation further enhanced the quality, yielding smoother transitions and reducing pixelation. Overall, this project emphasized the importance of choosing appropriate mapping techniques for achieving high-quality results in image processing.