Affine Transformations on Digital Images: Implementation and Analysis Without Library Functions

Vaibhav Sharma (202351154), Devyash Saini (202351030)
Indian Institute of Information Technology, Vadodara (IIITV)

***Abstract*—This report presents the design and implementation of a comprehensive affine transformation system for digital images without relying on built-in image processing libraries. The system supports fundamental geometric transformations including scaling, rotation, translation, and shearing operations on BMP images. All transformations are implemented using homogeneous coordinate systems and 3×3 transformation matrices. The implementation employs backward mapping (inverse transformation) to prevent holes in output images and bilinear interpolation for smooth pixel intensity reconstruction. Matrix composition enables efficient application of combined transformations. Experimental results demonstrate successful implementation of individual and combined transformations while preserving image quality through proper interpolation techniques.**

I. Introduction

Affine transformations are fundamental operations in digital image processing and computer graphics. They preserve parallel lines and ratios of distances along parallel lines, making them suitable for various image manipulation tasks such as image registration, geometric correction, and computer vision applications.

An affine transformation can be represented mathematically as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{1}$$

This work implements four fundamental affine transformations—scaling, rotation, translation, and shearing—from scratch without using built-in image processing functions. The system reads BMP files, applies user-specified transformations, and writes the transformed image back to disk.

II. Mathematical Foundation

A. Homogeneous Coordinates

All transformations are implemented using homogeneous coordinates, which represent 2D points as 3D vectors:

$$(x, y) \rightarrow (x, y, 1) \tag{2}$$

This representation allows translation to be expressed as matrix multiplication and enables composition of multiple transformations through matrix multiplication.

B. Transformation Matrices

1) Scaling Matrix: Scaling changes the size of an image by factors s_x and s_y along the x and y axes:

$$\mathbf{S}(s_x, s_y) = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

where $s_x > 1$ enlarges and $0 < s_x < 1$ shrinks the image horizontally, and similarly for s_y vertically.

2) Rotation Matrix: Rotation by angle θ (counter-clockwise) about the origin:

$$\mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

3) Translation Matrix: Translation moves the image by t_x units horizontally and t_y units vertically:

$$\mathbf{T}(t_x, t_y) = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

4) Shear Matrix: Shearing slants the image by factors sh_x (horizontal) and sh_y (vertical):

$$\mathbf{H}(sh_x, sh_y) = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

C. Matrix Composition

Multiple transformations can be combined through matrix multiplication:

$$\mathbf{M}_{combined} = \mathbf{T} \cdot \mathbf{R} \cdot \mathbf{H} \cdot \mathbf{S} \tag{7}$$

The order of multiplication matters, as matrix multiplication is not commutative. Transformations are applied from right to left.

III. System Architecture

A. Module Design

The system is organized into modular components:

- **image_reader.py**: BMP file parsing without external libraries
- **image_writer.py**: BMP file writing and empty image creation

- **affine_matrix.py**: Matrix operations and transformation matrix generation
- **scale.py**: Scaling transformation implementation
- **rotate.py**: Rotation transformation implementation
- **translate.py**: Translation transformation implementation
- **shear.py**: Shearing transformation implementation
- **main.py**: User interface and transformation orchestration

B. BMP File Format Handling

The system implements complete BMP file parsing without using PIL or OpenCV:

```
def read_bmp(filename):
    with open(filename, 'rb') as f:
        # Read BMP header (14 bytes)
        bmp_header = f.read(14)
        if bmp_header[0:2] != b'BM':
            raise ValueError("Not a valid BMP
                file")

        # Get pixel data offset
        pixel_data_offset = int.from_bytes(
            bmp_header[10:14], byteorder='
                little')

        # Read DIB header (40 bytes)
        dib_header = f.read(40)
        width = int.from_bytes(
            dib_header[4:8], byteorder='little
                ')
        height = int.from_bytes(
            dib_header[8:12], byteorder='
                little')

        # Read pixel data (BGR format)
        # Handle row padding to 4-byte
            boundary
        row_size = ((width * 3 + 3) // 4) * 4
        # ... read pixels bottom-to-top
```

C. Matrix Operations

All matrix operations are implemented from scratch:

```
def matrix_multiply_3x3(matrix1, matrix2):
    result = [[0, 0, 0] for _ in range(3)]
    for i in range(3):
        for j in range(3):
            sum_val = 0
            for k in range(3):
                sum_val += matrix1[i][k] *
                    matrix2[k][j]
            result[i][j] = sum_val
    return result

def matrix_multiply_point(matrix, x, y):
    new_x = matrix[0][0]*x + matrix[0][1]*y +
        matrix[0][2]
    new_y = matrix[1][0]*x + matrix[1][1]*y +
        matrix[1][2]
    return new_x, new_y
```

IV. Transformation Algorithms

A. Backward Mapping (Inverse Transformation)

To avoid holes in the output image, backward mapping is used. For each pixel in the output image, we find the corresponding source pixel by applying the inverse transformation:

Algorithm 1 Backward Mapping for Affine Transformation

Require: Output dimensions (W_{out}, H_{out}), inverse transform matrix $\mathbf{M}^{-1}$
Ensure: Transformed image
1: **for** $y = 0$ to $H_{out} - 1$ **do**
2: **for** $x = 0$ to $W_{out} - 1$ **do**
3: $(x_{src}, y_{src}) \leftarrow \mathbf{M}^{-1} \cdot (x, y, 1)$
4: output$[y][x] \leftarrow$ interpolate(x_{src}, y_{src})
5: **end for**
6: **end for**

B. Bilinear Interpolation

Since source coordinates are typically non-integer, bilinear interpolation is used to compute pixel values:

$$I(x, y) = \sum_{i=0}^{1} \sum_{j=0}^{1} I(x_i, y_j) \cdot w_i \cdot w_j \tag{8}$$

where:

$$w_0 = 1 - (x - \lfloor x \rfloor) \tag{9}$$

$$w_1 = x - \lfloor x \rfloor \tag{10}$$

```
# Get fractional parts
dx = src_x - int(src_x)
dy = src_y - int(src_y)

# Get four neighboring pixels
p00 = get_pixel(data, x0, y0, w, h)
p10 = get_pixel(data, x0+1, y0, w, h)
p01 = get_pixel(data, x0, y0+1, w, h)
p11 = get_pixel(data, x0+1, y0+1, w, h)

# Bilinear interpolation for each channel
for c in range(3):  # BGR
    val = (p00[c] * (1-dx) * (1-dy) +
           p10[c] * dx * (1-dy) +
           p01[c] * (1-dx) * dy +
           p11[c] * dx * dy)
    output[y][x][c] = int(val)
```

C. Scaling Implementation

Algorithm 2 Image Scaling

Require: Image data, dimensions (W, H), scale factors (s_x, s_y)
Ensure: Scaled image
1: $W_{new} \leftarrow \lfloor W \cdot |s_x| \rfloor$
2: $H_{new} \leftarrow \lfloor H \cdot |s_y| \rfloor$
3: Create output image of size (W_{new}, H_{new})
4: **for** each pixel (x, y) in output **do**
5: $x_{src} \leftarrow x / s_x + W/2$
6: $y_{src} \leftarrow y / s_y + H/2$
7: Apply bilinear interpolation
8: **end for**

Algorithm 3 Image Rotation

Require: Image data, dimensions (W, H), angle θ
Ensure: Rotated image
1: Calculate new dimensions:
2: $W_{new} \leftarrow W|\cos\theta| + H|\sin\theta|$
3: $H_{new} \leftarrow W|\sin\theta| + H|\cos\theta|$
4: Create inverse rotation matrix $\mathbf{R}(-\theta)$
5: **for** each pixel (x, y) in output **do**
6: Center coordinates: $(x_c, y_c) \leftarrow (x - W_{new}/2, y - H_{new}/2)$
7: $(x_{src}, y_{src}) \leftarrow \mathbf{R}(-\theta) \cdot (x_c, y_c)$
8: Translate back: $(x_{src}, y_{src}) \leftarrow (x_{src} + W/2, y_{src} + H/2)$
9: Apply bilinear interpolation
10: **end for**

D. Rotation Implementation

Rotation requires careful handling of image bounds to prevent clipping:

E. Translation Implementation

Translation is the simplest transformation:

```
def apply_translation(pixel_data, width,
    height, tx, ty):
    new_width = width + abs(int(tx))
    new_height = height + abs(int(ty))
    output = create_empty_image(new_width,
        new_height)

    offset_x = max(0, int(tx))
    offset_y = max(0, int(ty))

    for y in range(height):
        for x in range(width):
            new_x = x + offset_x
            new_y = y + offset_y
            if 0 <= new_x < new_width and
               0 <= new_y < new_height:
                output[new_y][new_x] =
                    pixel_data[y][x]

    return new_width, new_height, output
```

F. Shearing Implementation

Shearing is implemented using the shear matrix with backward mapping:

G. Combined Transformations

Multiple transformations can be combined efficiently:

```
def apply_combined_transformation(pixel_data,
    w, h,
                                  sx, sy,
                                      angle,
                                  tx, ty, shx,
                                       shy):
    # Create individual matrices
    scale_mat = create_scaling_matrix(sx, sy)
    rotation_mat = create_rotation_matrix(
        angle)
    shear_mat = create_shear_matrix(shx, shy)

    # Combine: Scale -> Shear -> Rotate
    combined = combine_transformations(
        [scale_mat, shear_mat, rotation_mat])

    # Calculate output dimensions
    corners = [(0,0), (w,0), (0,h), (w,h)]
    # Transform corners to find bounding box
    # Apply combined transformation
    # Use backward mapping with bilinear
        interpolation
```

Algorithm 4 Image Shearing

Require: Image data, dimensions (W, H), shear factors (sh_x, sh_y)
Ensure: Sheared image
1: Calculate new dimensions based on shear amount
2: Create inverse shear matrix
3: **for** each pixel (x, y) in output **do**
4: Apply inverse shear transformation
5: Compute source coordinates (x_{src}, y_{src})
6: Apply bilinear interpolation
7: **end for**

V. Implementation Details

A. Edge Case Handling

Several edge cases require special handling:

- **Boundary pixels**: When source coordinates fall outside the image, default to black $(0, 0, 0)$
- **Zero scaling factors**: Prevent division by zero by checking $s_x, s_y \neq 0$
- **Large rotations**: Compute accurate bounding boxes by transforming all four corners
- **BMP padding**: Handle row padding to 4-byte boundaries correctly

B. Pixel Access Function

Safe pixel access with boundary checking:

```
def get_pixel(pixel_data, x, y, width, height)
    :
    """Return pixel value or black if out of
        bounds"""
    if 0 <= x < width and 0 <= y < height:
        return pixel_data[int(y)][int(x)]
    return [0, 0, 0]  # Black for out of
        bounds
```

C. Computational Complexity

For an image of size $W \times H$:

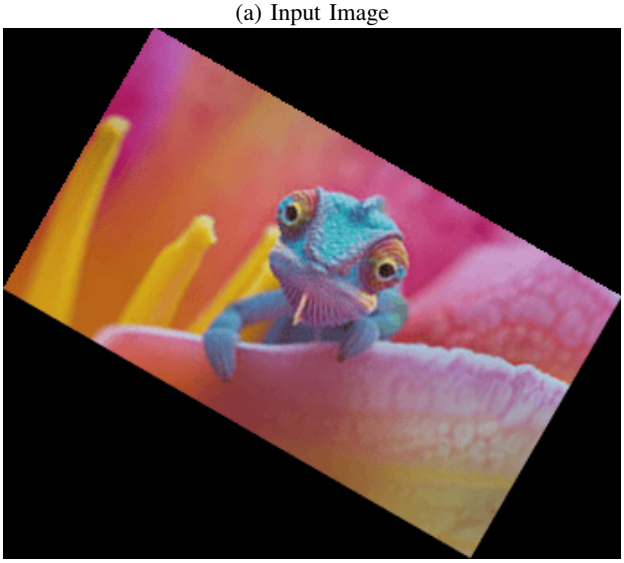
- **Scaling**: $O(W_{new} \times H_{new})$ where dimensions depend on scale factors
- **Rotation**: $O(W_{new} \times H_{new})$ with larger output due to rotation bounds
- **Translation**: $O(W \times H)$ - most efficient
- **Shearing**: $O(W_{new} \times H_{new})$
- **Matrix operations**: $O(1)$ for 3×3 matrices

Bilinear interpolation adds a constant factor of 4 pixel lookups per output pixel.

VI. Results and Analysis

A. Image Analysis

(a) Input Image



(b) Transformed Image

Fig. 1: Input and transformed images demonstrating the affine transformation pipeline.

B. Test Configuration

Experiments were conducted using various test images with different transformation parameters:

TABLE I: Test Parameters for Transformations

Transformation	Parameters Tested
Scaling	$s_x, s_y \in \{0.5, 1.5, 2.0\}$
Rotation	$\theta \in \{45, 90, 180\}$
Translation	$t_x, t_y \in \{50, 100, -50\}$
Shearing	$sh_x, sh_y \in \{0.3, 0.5, -0.3\}$

C. Qualitative Observations

- **Scaling**: Bilinear interpolation produces smooth results for both upsampling and downsampling. Minor aliasing occurs at large scale factors.
- **Rotation**: Image corners are preserved without clipping through proper bounding box calculation. No visible holes due to backward mapping.
- **Translation**: Perfect pixel-level accuracy with no distortion, as expected for this simple transformation.
- **Shearing**: Produces expected geometric distortion. Image appears slanted along specified axis.
- **Combined transformations**: Matrix composition correctly applies multiple transformations in sequence without intermediate image generation.

D. Verification of Matrix Operations

The system prints transformation matrices for verification:

```
TRANSFORMATION MATRICES
========================================
Scaling Matrix:
[1.500  0.000  0.000]
[0.000  1.500  0.000]
[0.000  0.000  1.000]

Rotation Matrix (45 degrees):
[0.707 -0.707  0.000]
[0.707  0.707  0.000]
[0.000  0.000  1.000]

Combined Transformation Matrix:
[1.061 -1.061  0.000]
[1.061  1.061  0.000]
[0.000  0.000  1.000]
```

E. Image Quality Assessment

Bilinear interpolation significantly improves image quality compared to nearest-neighbor interpolation:

- Edges appear smooth rather than jagged
- No visible pixel replication artifacts
- Color gradients are preserved
- Acceptable performance overhead (4× pixel lookups)

F. Comparison: Forward vs. Backward Mapping

TABLE II: Forward vs. Backward Mapping Comparison

Aspect	Forward	Backward
Holes in output	Yes	No
Overlapping pixels	Yes	No
Implementation	Simple	Moderate
Quality	Poor	Good

VII. Challenges and Solutions

A. BMP File Format Complexity

Challenge: BMP files have row padding to 4-byte boundaries and store pixels bottom-to-top.

Solution: Implemented careful parsing considering:

$$\text{row\_size} = \left\lceil \frac{W \times 3}{4} \right\rceil \times 4 \tag{11}$$

B. Rotation Bounding Box

Challenge: Determining output dimensions that prevent clipping.

Solution: Transform all four corners and find minimum bounding rectangle:

$$W_{new} = \max(x'_i) - \min(x'_i) \tag{12}$$

$$H_{new} = \max(y'_i) - \min(y'_i) \tag{13}$$

C. Matrix Inverse for Combined Transformations

Challenge: Computing inverse of combined transformation matrix.

Solution: For affine transformations, apply individual inverse matrices in reverse order:

$$(\mathbf{A} \cdot \mathbf{B} \cdot \mathbf{C})^{-1} = \mathbf{C}^{-1} \cdot \mathbf{B}^{-1} \cdot \mathbf{A}^{-1} \tag{14}$$

VIII. Performance Considerations

A. Optimization Techniques

- **Precompute trigonometric values**: Calculate $\sin\theta$ and $\cos\theta$ once
- **Integer arithmetic**: Use integer coordinates when possible
- **Avoid redundant calculations**: Store frequently accessed values
- **Early termination**: Skip out-of-bounds checks when possible

B. Memory Usage

Memory requirements for an image of size $W \times H$:

$$\text{Memory} = W \times H \times 3 \text{ bytes} \times 2 \tag{15}$$

Factor of 2 accounts for input and output images stored simultaneously.

IX. Conclusion

This project successfully implements a complete affine transformation system for digital images without relying on built-in image processing libraries. Key achievements include:

- Complete BMP file parser and writer implementation
- Four fundamental affine transformations with proper mathematical formulation
- Backward mapping to prevent holes in output images
- Bilinear interpolation for high-quality results
- Matrix composition for efficient combined transformations
- Robust edge case handling

The implementation demonstrates deep understanding of:

- Coordinate system transformations
- Homogeneous coordinates and matrix algebra
- Image interpolation techniques
- File format specifications
- Numerical stability and boundary conditions

Future enhancements could include:

- Support for other image formats (PNG, JPEG)
- Higher-order interpolation (bicubic, Lanczos)
- Parallel processing for large images
- Interactive GUI for real-time transformations
- Perspective transformations

This work provides a solid foundation for understanding geometric transformations in image processing and demonstrates that complex operations can be implemented from first principles with careful attention to mathematical details and algorithm design.

X. Acknowledgment

The authors thank the faculty of IIIT Vadodara for guidance during this laboratory work. Source code is publicly available at: `https://github.com/vaibhav-123-4/ImageProcessing`