"Heaven's Light is Our Guide"

Department of Computer Science & Engineering

Rajshahi University of Engineering & Technology

Lab Report-3

Digital Image Processing Sessional

Course Code: CSE 4106

Submitted By:	Submitted To:
Name: Sajidur Rahman Tarafder Roll: 2003154 Section: C Session: 2020-21 Department: CSE	Khaled Zinnurine Lecturer Department of CSE, RUET

Contents

Geometric Transformations on Images

Objective: Implement various geometric transformations (Translation, Rotation, Scaling, Shearing, and Combined Affine Transformations) from scratch using Python.

1 Image Loading and Preprocessing

1.1 Code Snippet

```python
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Load image and convert to RGB
img = cv2.imread("bird.jpg")
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# Display original image
plt.figure(figsize=(12, 5))
plt.imshow(img)
plt.title('Original Image')
plt.axis('off')
plt.show()
```

Listing 1: Loading and Displaying Original Image

1.2 Implementation Approach

I start by loading a color image using OpenCV's `imread` function. Since OpenCV reads images in BGR format but matplotlib displays in RGB format, I convert the color space using `cvtColor`. This ensures the colors appear correctly when visualizing the results. The original image serves as the baseline for all subsequent transformations.

2 Translation Transformation

2.1 Code Snippet

```python
# Translation parameters
tx = 60  # Shift right by 60 pixels
ty = 60  # Shift down by 60 pixels

height, width = img.shape[:2]
result = np.zeros_like(img)
```

```
7
8  # Apply translation pixel by pixel
9  for y in range(height):
10     for x in range(width):
11         new_x = x + tx
12         new_y = y + ty
13
14         if new_x < width and new_y < height:
15             result[new_y, new_x] = img[y, x]
16
17 # Display results
18 plt.figure(figsize=(12, 5))
19 plt.subplot(1, 2, 1)
20 plt.imshow(img)
21 plt.title("Original Image")
22 plt.axis('off')
23
24 plt.subplot(1, 2, 2)
25 plt.imshow(result)
26 plt.title(f"Translated Image")
27 plt.axis('off')
28 plt.show()
```

Listing 2: Manual Translation Implementation

2.2 Implementation Approach

Translation is the simplest geometric transformation - it just moves the entire image to a new position. I implemented this by iterating through every pixel in the original image and placing it at a new location calculated by adding the translation offsets (tx, ty) to the original coordinates.

The formula is straightforward:

- `new_x = x + tx`
- `new_y = y + ty`

I created a blank canvas the same size as the original and copied each pixel to its new location. The boundary check ensures we don't try to write pixels outside the image bounds. Parts of the original image that would be shifted outside the canvas are simply not displayed, creating blank areas (visible as black regions) where the image moved from.

3 Rotation Transformation

3.1 Code Snippet

```python
# Rotation parameters
angle = 180  # Rotate 180 degrees

height, width = img.shape[:2]
center_x = width // 2
center_y = height // 2

# Convert angle to radians
angle_rad = np.radians(angle)
cos_theta = np.cos(angle_rad)
sin_theta = np.sin(angle_rad)

result = np.zeros_like(img)

# Apply rotation pixel by pixel
for y in range(height):
    for x in range(width):
        # Step 1: Translate to origin
        new_x = x - center_x
        new_y = y - center_y

        # Step 2: Apply rotation
        u = new_x * cos_theta - new_y * sin_theta
        v = new_x * sin_theta + new_y * cos_theta

        # Step 3: Translate back
        u = int(u + center_x)
        v = int(v + center_y)

        if 0 <= u < width and 0 <= v < height:
            result[v, u] = img[y, x]

# Display results
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.title("Original Image")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(result)
plt.title(f"Rotated {angle} degrees")
plt.axis('off')
plt.show()
```

Listing 3: Manual Rotation Around Center

3.2 Implementation Approach

Rotation is more complex than translation because we need to rotate around a specific point (usually the image center). The rotation transformation uses trigonometry with the rotation matrix formulas:

$$u = x\cos(\theta) - y\sin(\theta)$$
$$v = x\sin(\theta) + y\cos(\theta)$$

However, these formulas rotate around the origin (0,0). To rotate around the image center, I use a three-step process:

Step 1 - Translate to Origin: Shift all coordinates so the center point becomes (0,0).

Step 2 - Apply Rotation: Use the rotation formulas with pre-calculated sine and cosine values.

Step 3 - Translate Back: Shift the rotated coordinates back so the center returns to its original position.

The angle is converted from degrees to radians because trigonometric functions in programming use radians. I pre-calculate `cos(θ)` and `sin(θ)` outside the loop for efficiency since they're constant for all pixels.

4 Scaling Transformation

4.1 Code Snippet

```python
# Scaling parameters
scale_x = 0.5  # Scale down to 50% width
scale_y = 1.2  # Scale up to 120% height

height, width = img.shape[:2]

# Calculate new dimensions
new_width = int(width * scale_x)
new_height = int(height * scale_y)

result = np.zeros((new_height, new_width, 3), dtype=img.dtype)

# Apply scaling pixel by pixel
for y in range(height):
    for x in range(width):
        new_x = int(x * scale_x)
        new_y = int(y * scale_y)

        if new_x < new_width and new_y < new_height:
            result[new_y, new_x] = img[y, x]

# Display results
plt.figure(figsize=(15, 6))
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.title(f"Original ({height}x{width})")
```

```
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(result)
plt.title(f"Scaled ({new_height}x{new_width})")
plt.axis('off')
plt.show()
```

Listing 4: Manual Scaling Implementation

4.2 Implementation Approach

Scaling changes the size of the image by multiplying the coordinates by scale factors. I can scale the width and height independently, allowing for stretching or squashing in different directions.

The scaling transformation is:

- new_x = x × scale_x
- new_y = y × scale_y

First, I calculate the output image dimensions by multiplying the original dimensions by the scale factors. Then I create a blank canvas of this new size. When iterating through the original image, I multiply each pixel's coordinates by the scale factors to determine where it goes in the scaled image.

Scale factors less than 1 shrink the image, while factors greater than 1 enlarge it. In this example, I used 0.5 for width (making it narrower) and 1.2 for height (making it taller), demonstrating non-uniform scaling. This approach creates gaps in the output image (visible as black dots) because we're using forward mapping - some output pixels don't get assigned values. A production implementation would use inverse mapping or interpolation to fill these gaps.

5 Shearing Transformation

5.1 Code Snippet

```
# Shearing parameters
shear_x = 0.3  # Horizontal shear factor
shear_y = 0.5  # Vertical shear factor

height, width = img.shape[:2]

# Calculate new dimensions to accommodate shearing
new_width = width + int(abs(shear_x * height))
new_height = height + int(abs(shear_y * width))

```

```
result = np.zeros((new_height, new_width, 3), dtype=img.dtype)

# Apply shearing pixel by pixel
for y in range(height):
    for x in range(width):
        new_x = int(x + shear_x * y)
        new_y = int(y + shear_y * x)

        if new_x < new_width and new_y < new_height:
            result[new_y, new_x] = img[y, x]

# Display results
plt.figure(figsize=(15, 6))
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.title("Original Image")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(result)
plt.title(f"Sheared (shx={shear_x}, shy={shear_y})")
plt.axis('off')
plt.show()
```

Listing 5: Manual Shearing Implementation

5.2 Implementation Approach

Shearing creates a slanted or skewed appearance by shifting pixels proportionally to their position. It's like pushing the top of the image while keeping the bottom fixed, or vice versa.

The shearing transformation formulas are:

- `new_x = x + shear_x × y`
- `new_y = y + shear_y × x`

Shearing is interesting because it changes the shape of the image without changing the relative positions of points along parallel lines. The amount of shift increases linearly with position - pixels at the top of the image shift more than pixels at the bottom.

Since shearing expands the image bounds, I calculate new dimensions that are large enough to hold the sheared result. The horizontal shear factor affects the width increase, and the vertical shear factor affects the height increase. Positive shear values push the image in the positive direction, while negative values would push it the opposite way.

6 Combined Affine Transformation

6.1 Code Snippet

```python
# Combined transformation parameters
angle = 45    # Rotate 45 degrees
tx = 60       # Then shift right 60 pixels
ty = 40       # Then shift down 40 pixels

height, width = img.shape[:2]
center_x = width // 2
center_y = height // 2

angle_rad = np.radians(angle)
cos_theta = np.cos(angle_rad)
sin_theta = np.sin(angle_rad)

result = np.zeros_like(img)

# Apply combined transformation pixel by pixel
for y in range(height):
    for x in range(width):
        # Step 1: Translate to origin
        new_x = x - center_x
        new_y = y - center_y

        # Step 2: Apply rotation
        rotated_x = new_x * cos_theta - new_y * sin_theta
        rotated_y = new_x * sin_theta + new_y * cos_theta

        # Step 3: Translate back to center
        x_after_rotation = rotated_x + center_x
        y_after_rotation = rotated_y + center_y

        # Step 4: Apply translation
        final_x = int(x_after_rotation + tx)
        final_y = int(y_after_rotation + ty)

        if 0 <= final_x < width and 0 <= final_y < height:
            result[final_y, final_x] = img[y, x]

# Display results
plt.figure(figsize=(12, 7))
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.title("Original Image")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(result)
plt.title(f"Rotated {angle} degrees + Translated ({tx}, {ty})")
plt.axis('off')
plt.show()
```

Listing 6: Rotation Followed by Translation

6.2 Implementation Approach

Combined transformations show the real power of geometric operations. By applying multiple transformations in sequence, we can create complex effects. This example combines rotation and translation to both spin and move the image.

The order of operations matters! I first rotate the image around its center, then translate it. If we did translation first, the rotation would happen around a different point, giving a completely different result.

The four-step process is:

1. Translate coordinates to origin for rotation
2. Apply rotation transformation
3. Translate back from origin
4. Apply translation to final position

This demonstrates an important principle: complex transformations can be built by composing simple ones. In real applications, you'd typically use transformation matrices to combine multiple operations efficiently, but this manual implementation shows clearly what's happening at each step.

Discussion and Conclusion

In this lab, I implemented five fundamental geometric transformations from scratch: translation, rotation, scaling, shearing, and combined transformations. This hands-on implementation gave me deep insights into how these operations work at the pixel level.

Key Learnings

Translation is the simplest transformation - just adding offsets to coordinates. It's used everywhere in computer graphics, from moving game characters to panning camera views.

Rotation requires trigonometry and careful handling of the rotation center. The three-step process (translate to origin, rotate, translate back) is a fundamental pattern that appears in many graphics algorithms. Understanding this helped me see why transformation matrices are so useful in production code.

Scaling can be uniform (same factor in all directions) or non-uniform (different factors for width and height). The gaps in my scaled images showed me why interpolation is important in real image processing - professional implementations use techniques like bilinear interpolation to fill in missing pixels.

Shearing creates interesting skewed effects and is less commonly used alone but is important for understanding perspective transformations and 3D projections. It's also used in correcting image distortions.

Combined transformations demonstrated that complex effects come from composing simple operations. The order matters - rotating then translating gives different results than translating then rotating.

Implementation Challenges

The main challenge was handling the "holes" in the output images. My forward mapping approach (going from source to destination) leaves gaps because multiple source pixels might map to the same destination pixel, while other destination pixels get no source pixel. Professional implementations use:

- **Inverse mapping:** For each output pixel, calculate which input pixel it should come from
- **Interpolation:** Use bilinear or bicubic interpolation to fill gaps smoothly
- **Anti-aliasing:** Smooth the edges to avoid jagged appearance

Another challenge was maintaining image quality. Transformations can introduce artifacts, especially with multiple operations. Understanding these issues helps explain why image processing libraries have so many options for interpolation methods and quality settings.

Real-World Applications

These transformations are fundamental to:

- **Computer Vision:** Image registration, object tracking, augmented reality
- **Medical Imaging:** Aligning scans from different angles or time periods
- **Computer Graphics:** Game engines, 3D rendering, animation
- **Photography:** Image editing software, panorama stitching, perspective correction
- **Document Processing:** OCR systems, document alignment

Mathematical Foundation

All these transformations can be represented as matrix operations. In homogeneous coordinates, each transformation becomes a 3×3 matrix:

- Translation, rotation, scaling, and shearing can all be expressed as matrix multiplications
- Multiple transformations combine by multiplying their matrices
- This matrix representation is what makes modern graphics hardware so fast

Understanding the manual implementation first made me appreciate why libraries like OpenCV use optimized matrix operations - they're mathematically elegant and computationally efficient.

Conclusion

This lab showed me that geometric transformations aren't just black-box functions in a library - they're based on clear mathematical principles that can be implemented with basic operations. While my implementations were educational, they also highlighted why production code uses inverse mapping, interpolation, and hardware acceleration. The gaps and artifacts in my results aren't failures - they're learning opportunities that show exactly what professional implementations need to handle.

Most importantly, I now understand transformation composition and can debug issues in real applications because I know what's happening under the hood. This foundational knowledge will be valuable whether I'm working with computer vision, computer graphics, or any field that manipulates visual data.