

Project 4: Transformations

Nguyễn Trang Sỹ Lâm

Ngày 15 tháng 11 năm 2024

Môn học: Xử lý Ảnh số và Thị giác Máy tính
Giảng viên hướng dẫn: Võ Thanh Hùng

1 Giới thiệu

Các phép biến hình của hình ảnh (image transformations) cho phép chúng ta thay đổi không gian và cấu trúc của ảnh để thực hiện nhiều tác vụ hữu ích, được gọi là biến đổi hình ảnh hoặc xử lý ảnh dựa trên biến hình. Các tác vụ phổ biến như dịch chuyển (translation), xoay (rotation), thu phóng (scaling), và làm biến dạng hình học (warping) đều có thể được thực hiện một cách đơn giản và hiệu quả bằng cách áp dụng các phép biến đổi toán học lên tọa độ hoặc giá trị của từng điểm ảnh.

Trong bài toán này, chúng ta sẽ tập trung vào nhiều loại biến đổi khác nhau, từ những phép biến đổi đơn giản như dịch chuyển và xoay, đến các kỹ thuật phức tạp hơn như biến đổi phối cảnh (perspective transformation) và biến hình cục bộ (local warping). Mỗi kỹ thuật sẽ được sử dụng để thay đổi hoặc kết hợp hình ảnh nhằm đạt được các hiệu ứng khác nhau. Các phép biến hình này dựa trên việc áp dụng các ma trận biến đổi hoặc các phương pháp nội suy nhằm đảm bảo rằng hình ảnh sau biến đổi vẫn giữ được tính liên tục và mượt mà.

Mục tiêu là tìm hiểu và ứng dụng các phép biến đổi hình ảnh nhằm thay đổi cấu trúc không gian và độ sáng của ảnh một cách tự nhiên và liền mạch, phục vụ cho nhiều mục đích xử lý ảnh khác nhau.

2 Công thức toán

2.1 So sánh Filtering và Warping

Filtering (lọc) và **warping** (biến dạng) là hai kỹ thuật biến đổi cơ bản trong xử lý ảnh, với mục đích khác nhau thông qua việc tác động vào các yếu tố khác nhau của hình ảnh.

2.1.1 Image Filtering (Lọc ảnh)

Lọc ảnh thay đổi **dải giá trị** (range) của ảnh, tức là thay đổi giá trị cường độ của các pixel mà không thay đổi vị trí không gian của chúng. Phép biến đổi này được áp dụng trực tiếp lên giá trị của từng pixel mà không ảnh hưởng đến tọa độ của pixel. Về mặt toán học, phép lọc có thể được biểu diễn như sau:

$$g(x) = T(f(x))$$

trong đó:

- $f(x)$ là hàm ảnh gốc đại diện cho các giá trị cường độ.
- T là hàm biến đổi được áp dụng cho cường độ của từng pixel.
- $g(x)$ là ảnh kết quả sau khi lọc.

Các ví dụ về phép lọc bao gồm tăng cường độ tương phản, điều chỉnh độ sáng, và phát hiện cạnh, trong đó phép biến đổi thay đổi cường độ của các pixel dựa trên một số tiêu chí nhất định.

2.1.2 Image Warping (Biến dạng ảnh)

Ngược lại, warping thay đổi **miền không gian** (domain) của ảnh, tức là thay đổi cách sắp xếp các pixel trong không gian bằng cách biến đổi tọa độ của chúng. Warping ảnh hướng đến vị trí của các pixel thay vì giá trị cường độ của chúng, dẫn đến sự biến đổi hình học của ảnh. Phép biến đổi này có thể được biểu diễn như sau:

$$g(x) = f(T(x))$$

trong đó:

- $f(x)$ là hàm ảnh gốc, trong đó giá trị cường độ của các pixel không thay đổi.

- $T(x)$ là hàm biến đổi thay đổi tọa độ của từng pixel.
- $g(x)$ là ảnh kết quả sau khi warping.

Các ví dụ về warping bao gồm xoay, phóng to, thu nhỏ, làm xiên, và biến đổi phôi cảnh, thay đổi cấu trúc không gian của ảnh để đạt được các hiệu ứng như làm biến dạng, xoay, hoặc thay đổi kích thước.

2.2 Translation (Phép Dịch Chuyển)

Translation (dịch chuyển) là một phép biến đổi hình ảnh đơn giản, trong đó mỗi pixel trong ảnh được di chuyển theo một khoảng cách cố định trong các hướng x và y. Phép dịch chuyển này được biểu diễn bằng một ma trận affine, trong đó các giá trị dịch chuyển được thêm vào tọa độ của từng pixel.

Giả sử tọa độ của một pixel ban đầu là (x, y) và tọa độ sau khi dịch chuyển là (x', y') , phép dịch chuyển có thể được biểu diễn dưới dạng:

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

Trong đó:

- t_x và t_y lần lượt là khoảng cách dịch chuyển theo trục x và trục y.
- (x', y') là tọa độ của pixel sau khi dịch chuyển.

2.3 Rotation (Xoay)

Phép **xoay** là một phép biến hình sử dụng ma trận xoay để thay đổi vị trí của từng điểm ảnh trong ảnh dựa trên góc xoay θ . Phép xoay thường được áp dụng quanh gốc tọa độ $(0, 0)$ hoặc một điểm cho trước, sử dụng công thức sau:

Giả sử (x, y) là tọa độ ban đầu của một điểm ảnh, và (x', y') là tọa độ sau khi xoay quanh gốc với góc θ , ta có:

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

Trong đó:

- θ là góc xoay (theo đơn vị radian), tính theo chiều ngược kim đồng hồ.
- $\cos \theta$ và $\sin \theta$ lần lượt là các giá trị cosin và sin của góc xoay θ .

Khi xoay quanh một điểm (x_c, y_c) khác gốc tọa độ, công thức xoay được điều chỉnh bằng cách dịch chuyển điểm về gốc trước khi xoay và dịch chuyển lại sau khi xoay:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x - x_c \\ y - y_c \end{bmatrix} + \begin{bmatrix} x_c \\ y_c \end{bmatrix}$$

2.4 Scaling (Phóng đại và thu nhỏ)

Phép **scaling** (phóng đại và thu nhỏ) thay đổi kích thước của ảnh bằng cách nhân tọa độ của mỗi điểm ảnh với một hệ số tỷ lệ. Có hai loại scaling chính:

2.4.1 Uniform Scaling (Phóng đại đồng nhất)

Trong **uniform scaling**, cả tọa độ x và y đều được nhân với cùng một hệ số s , giữ nguyên tỷ lệ khung hình:

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

2.4.2 Non-Uniform Scaling (Phóng đại không đồng nhất)

Trong **non-uniform scaling**, các hệ số tỷ lệ khác nhau được áp dụng cho tọa độ x và y, có thể làm biến dạng ảnh:

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

Trong đó:

- s, s_x , và s_y lần lượt là các hệ số tỷ lệ cho uniform và non-uniform scaling.

2.5 Shearing (Biến dạng xiên)

Phép **shearing** làm xiên ảnh theo hướng x hoặc y, bằng cách thêm một thành phần tỷ lệ từ một tọa độ vào tọa độ kia. Có hai loại shearing:

2.5.1 X-Direction Shear (Biến dạng xiên theo trục x)

Trong x-direction shear, tọa độ x được điều chỉnh dựa trên tọa độ y:

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

2.5.2 Y-Direction Shear (Biến dạng xiên theo trục y)

Trong y-direction shear, tọa độ y được điều chỉnh dựa trên tọa độ x:

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

Trong đó:

- k là hệ số shear, xác định độ xiên của phép biến dạng.

2.6 Mirroring (Phép Đối Xứng)

Phép **mirroring** tạo ra một ảnh phản chiếu bằng cách thay đổi trật tự của các điểm ảnh theo một trong các trục. Có hai phép đối xứng phổ biến:

2.6.1 Reflection Across the x-axis (Đối xứng qua trục x)

Trong phép đối xứng qua trục x, tọa độ y được đảo ngược:

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

2.6.2 Reflection Across the y-axis (Đối xứng qua trục y)

Trong phép đối xứng qua trục y, tọa độ x được đảo ngược:

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

2.7 Affine Transformation (Phép Biến Đổi Affine)

Phép **affine transformation** kết hợp các phép biến đổi tuyến tính như scaling, rotation, shearing và translation. Ma trận affine là một ma trận 2x3 có dạng:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b & t_x \\ c & d & t_y \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Trong đó:

- a, b, c, d xác định tỷ lệ, xoay, và biến dạng xiên.
- t_x và t_y là các hằng số dịch chuyển.

2.8 Perspective (Projective) Transformation (Phép Biến Đổi Phối Cảnh)

Phép **perspective transformation** mở rộng affine transformation bằng cách thêm yếu tố phối cảnh, cho phép các đường song song hội tụ. Ma trận 3×3 biểu diễn phép biến đổi phối cảnh có dạng:

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Với tọa độ kết quả là:

$$x' = \frac{h_{11}x + h_{12}y + h_{13}}{h_{31}x + h_{32}y + 1}$$

$$y' = \frac{h_{21}x + h_{22}y + h_{23}}{h_{31}x + h_{32}y + 1}$$

Trong đó, ma trận được tính từ các điểm tương ứng giữa ảnh nguồn và ảnh đích.

2.9 Forward Warping

Forward Mapping trong phép **forward warping** là quá trình mà mỗi pixel trong ảnh nguồn được ánh xạ tới một vị trí mới trong ảnh đích bằng cách sử dụng ma trận biến đổi. Điều này có thể dẫn đến các khoảng trống trong ảnh đích nếu một số pixel không được ánh xạ trực tiếp.

Giả sử ảnh nguồn có pixel tại tọa độ (x, y) được ánh xạ tới vị trí mới (x', y') trong ảnh đích bằng ma trận biến đổi T :

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = T \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Trong đó:

- T là ma trận biến đổi, có thể là affine hoặc projective tùy theo yêu cầu của phép biến hình.
- (x', y') là tọa độ mới của pixel trong ảnh đích.

Do không phải tất cả pixel trong ảnh đích đều được ánh xạ từ ảnh nguồn, phương pháp này có thể để lại các vùng trống.

2.10 Inverse Warping

Trong **inverse warping**, mỗi pixel trong ảnh đích được ánh xạ ngược lại về một pixel tương ứng trong ảnh nguồn để xác định giá trị của nó. Phép nội suy song tuyến tính (**bilinear interpolation**) được sử dụng để ước lượng giá trị pixel từ các pixel lân cận trong ảnh nguồn, giúp đảm bảo ảnh sau khi biến đổi không có khoảng trống và mượt mà hơn.

Với tọa độ pixel (x', y') trong ảnh đích, vị trí tương ứng trong ảnh nguồn (x, y) được xác định như sau:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = T^{-1} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

Trong đó T^{-1} là ma trận nghịch đảo của ma trận biến đổi T . Phép nội suy song tuyến tính được áp dụng để xác định giá trị của (x', y') dựa trên các giá trị lân cận trong ảnh nguồn.

2.11 Morphing (Object Averaging)

Morphing là quá trình kết hợp hai ảnh bằng cách tính trung bình các vị trí và giá trị của các điểm ảnh tương ứng.

2.11.1 Linear Interpolation (Nội suy tuyến tính)

Linear Interpolation giữa các ảnh tính toán các hình dạng trung gian bằng cách tính trung bình tọa độ của các điểm đặc trưng tương ứng, tạo ra một sự chuyển đổi mượt mà giữa các hình dạng:

$$(x_{\text{morph}}, y_{\text{morph}}) = (1 - \alpha) \cdot (x_1, y_1) + \alpha \cdot (x_2, y_2)$$

Trong đó:

- (x_1, y_1) và (x_2, y_2) là các tọa độ điểm tương ứng trong hai ảnh.

- α là hệ số nội suy ($0 \leq \alpha \leq 1$), điều chỉnh mức độ pha trộn giữa hai ảnh.

2.11.2 Cross-Dissolve (Pha trộn màu)

Cross-Dissolve pha trộn màu của hai ảnh theo thời gian bằng cách nội suy giá trị pixel với hệ số pha trộn α :

$$I_{\text{morph}} = (1 - \alpha) \cdot I_1 + \alpha \cdot I_2$$

Trong đó:

- I_1 và I_2 là các giá trị pixel tương ứng trong hai ảnh.
- α là hệ số pha trộn ($0 \leq \alpha \leq 1$).

2.12 Local (Non-Parametric) Warping

Local Warping cho phép biến dạng từng phần của ảnh dựa trên việc ánh xạ các điểm đặc trưng hoặc các vùng lân cận.

2.12.1 Feature Matching (Đối sánh đặc trưng)

Trong **feature matching**, các điểm đặc trưng giữa hai ảnh được đối sánh để thực hiện biến đổi cục bộ. Các kỹ thuật như ORB hoặc SIFT giúp phát hiện và đối sánh các điểm đặc trưng, cho phép biến dạng các vùng cụ thể của ảnh.

2.12.2 Triangular Mesh Warping (Biến dạng lưới tam giác)

Ảnh được chia thành các vùng tam giác bằng cách sử dụng lưới tam giác Delaunay. Mỗi tam giác trải qua một phép biến đổi affine độc lập dựa trên các tam giác tương ứng trong ảnh nguồn và ảnh đích, cho phép biến dạng cục bộ chi tiết:

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

Trong đó A là ma trận affine được tính từ các điểm của tam giác trong ảnh nguồn và ảnh đích.

2.13 Dense and Sparse Warp Specification

2.13.1 Dense Warp (Biến dạng dày đặc)

Dense Warp định nghĩa một trường vector trong đó mỗi pixel có một vector dịch chuyển tương ứng, tạo ra các biến dạng liên tục như xoáy hoặc kéo dãn. Tọa độ mới của mỗi pixel được tính dựa trên vector dịch chuyển của nó:

$$(x', y') = (x + \Delta x, y + \Delta y)$$

Trong đó $(\Delta x, \Delta y)$ là vector dịch chuyển tại điểm (x, y) .

2.13.2 Sparse Warp (Biến dạng thưa thớt)

Trong **sparse warping**, một số điểm đặc trưng được xác định với các biến đổi đã biết. Các biến đổi trung gian cho các pixel khác được tính thông qua nội suy, như hàm nội suy ‘griddata’, đảm bảo chuyển đổi mượt mà trên toàn ảnh dù chỉ có một số điểm kiểm soát:

$$(x', y') = f(x, y)$$

Trong đó f là hàm nội suy xác định vị trí mới dựa trên các điểm kiểm soát.

3 Hiện thực

3.1 Translation (Phép Dịch Chuyển)

```
1 def translate_image(image, x_offset, y_offset, output_path):
2     width, height = image.size
3     matrix = (1, 0, x_offset, 0, 1, y_offset) # Affine
4         ↳ transformation matrix for translation
5     translated_image = image.transform((width, height),
6         ↳ Image.AFFINE, matrix)
7     translated_image.save(output_path)
8     return translated_image
```

3.2 Rotation (Xoay)

```
1 def rotate_image(image, angle, output_path):  
2     rotated_image = image.rotate(angle, expand=True)    # Expand  
    ↳ to fit the entire rotated image  
3     rotated_image.save(output_path)  
4     return rotated_image
```

3.3 Scaling (Phóng đại và thu nhỏ)

```
1 def scale_image_uniform(image, scale_factor, output_path):  
2     width, height = image.size  
3     new_size = (int(width * scale_factor), int(height *  
    ↳ scale_factor))  
4     scaled_image = image.resize(new_size)  
5     scaled_image.save(output_path)  
6     return scaled_image  
7  
8 def scale_image_non_uniform(image, x_scale_factor,  
    ↳ y_scale_factor, output_path):  
9     width, height = image.size  
10    new_size = (int(width * x_scale_factor), int(height *  
    ↳ y_scale_factor))  
11    scaled_image = image.resize(new_size)  
12    scaled_image.save(output_path)  
13    return scaled_image
```

3.4 Shearing (Biến dạng xiên)

```
1 def shear_image(image, shear_factor, direction='x',  
    ↳ output_path=None):  
2     width, height = image.size  
3     if direction == 'x':  
        # Shear in the x-direction  
        matrix = (1, shear_factor, 0, 0, 1, 0)  
5     elif direction == 'y':  
        # Shear in the y-direction  
        matrix = (1, 0, 0, shear_factor, 1, 0)  
6     else:  
        raise ValueError("Direction must be 'x' or 'y'")  
11
```

```

12     sheared_image = image.transform((width, height),
13         Image.AFFINE, matrix)
14     if output_path:
15         sheared_image.save(output_path)
16     return sheared_image

```

3.5 Affine Transformation (Phép Biến Đổi Affine)

```

1 def affine_transform(image, scale_x, scale_y, shear_x, shear_y,
2     rotation_angle, translate_x, translate_y, output_path):
3     # Convert rotation angle to radians
4     angle_rad = rotation_angle * (np.pi / 180)  # degrees to
5     # radians
6
7     # Affine transformation matrix combining scaling, rotation,
8     # shear, and translation
9     matrix = (
10         scale_x * np.cos(angle_rad) - shear_x *
11             np.sin(angle_rad),
12             shear_y * np.cos(angle_rad) + scale_x *
13                 np.sin(angle_rad),
14                 translate_x,
15                 shear_x * np.cos(angle_rad) - scale_y *
16                     np.sin(angle_rad),
17                     scale_y * np.cos(angle_rad) + shear_y *
18                         np.sin(angle_rad),
19                         translate_y
20     )
21
22     transformed_image = image.transform(image.size,
23         Image.AFFINE, matrix)
24     transformed_image.save(output_path)
25     return transformed_image

```

3.6 Perspective (Projective) Transformation (Phép Biến Đổi Phối Cảnh)

```

1 # Function to compute the projective transformation matrix
2 def compute_projective_matrix(src_points, dst_points):
3     matrix = []

```

```

4     for p1, p2 in zip(src_points, dst_points):
5         matrix.append([p1[0], p1[1], 1, 0, 0, 0, -p2[0]*p1[0],
6                         ↵ -p2[0]*p1[1], -p2[0]])
7         matrix.append([0, 0, 0, p1[0], p1[1], 1, -p2[1]*p1[0],
8                         ↵ -p2[1]*p1[1], -p2[1]])
9     matrix = np.array(matrix, dtype=np.float64)
10
11
12     # Solve the matrix to find the projective transformation
13     # coefficients
14     _, _, V = np.linalg.svd(matrix)
15     projective_matrix = V[-1, :].reshape((3, 3))
16     return projective_matrix / projective_matrix[2, 2]
17
18
19     # Function to apply projective transformation
20     def projective_transform(image, src_points, dst_points,
21                             output_path):
22         width, height = image.size
23         matrix = compute_projective_matrix(src_points,
24                                           ↵ dst_points).flatten()
25
26
27         # Apply projective transformation
28         transformed_image = image.transform((width, height),
29                                           ↵ Image.PERSPECTIVE, matrix, Image.BICUBIC)
30         transformed_image.save(output_path)
31
32     return transformed_image

```

3.7 Foward Warping

```

1  def forward_warp(source, destination, matrix):
2      src_height, src_width = source.shape[:2]
3
4      for y in range(src_height):
5          for x in range(src_width):
6              # Apply the affine transformation to (x, y)
7              new_pos = np.dot(matrix, [x, y, 1])
8              new_x, new_y = int(new_pos[0]), int(new_pos[1])
9
10             # Check if the new position is within the bounds of
11             # the destination image
12             if 0 <= new_x < destination.shape[1] and 0 <= new_y
13                 < destination.shape[0]:

```

```
12         destination[new_y, new_x] = source[y, x] # Map
→      pixel value
```

3.8 Inverse Warping

```
1  from PIL import Image
2  import numpy as np
3
4  # Load the rotated image
5  rotated_image_path = '../image/rotated_image_45.jpg'
6  rotated_image = Image.open(rotated_image_path)
7  rotated_array = np.array(rotated_image)
8  height, width = rotated_array.shape[:2]
9
10 # Create an empty array for the restored image
11 restored_array = np.zeros_like(rotated_array)
12
13 # Define the inverse rotation matrix (for a 45-degree rotation)
14 angle_rad = -45 * (np.pi / 180) # Inverse of 45 degrees in
→ radians
15 cos_theta, sin_theta = np.cos(angle_rad), np.sin(angle_rad)
16 inverse_rotation_matrix = np.array([
17     [cos_theta, -sin_theta],
18     [sin_theta, cos_theta]
19 ])
20
21 # Function to perform bilinear interpolation
22 def bilinear_interpolate(img, x, y):
23     x0, y0 = int(np.floor(x)), int(np.floor(y))
24     x1, y1 = min(x0 + 1, img.shape[1] - 1), min(y0 + 1,
→ img.shape[0] - 1)
25     dx, dy = x - x0, y - y0
26
27     # Interpolate the pixel values
28     top = (1 - dx) * img[y0, x0] + dx * img[y0, x1]
29     bottom = (1 - dx) * img[y1, x0] + dx * img[y1, x1]
30     value = (1 - dy) * top + dy * bottom
31     return value
32
33 # Inverse warping process
34 for y in range(height):
35     for x in range(width):
```

```

36     # Center the coordinates around the image center
37     centered_x, centered_y = x - width // 2, y - height //
38         ↵ 2
39
40     # Apply the inverse rotation transformation
41     src_x, src_y = np.dot(inverse_rotation_matrix,
42         ↵ [centered_x, centered_y])
43
44     # Shift the coordinates back to the original image
45     ↵ position
46     src_x += width // 2
47     src_y += height // 2
48
49     # If the source position is within bounds, apply
50     ↵ bilinear interpolation
51     if 0 <= src_x < width and 0 <= src_y < height:
52         restored_array[y, x] =
53             ↵ bilinear_interpolate(rotated_array, src_x,
54                 ↵ src_y)
55
56     # Convert back to an image and save
57     restored_image =
58         ↵ Image.fromarray(restored_array.astype('uint8'))
59     restored_image.save('../image/restored_image.jpg')
60
61 print("Restored image saved as 'restored_image.jpg'")

```

3.9 Morphing (Object Averaging)

```

1  # Function to perform linear interpolation and cross-dissolve
2      ↵ between two images
3  def morph_images(array1, array2, alpha):
4      # Linear interpolation and cross-dissolve using alpha
5      morphed_array = (1 - alpha) * array1 + alpha * array2
6      return np.clip(morphed_array, 0, 255).astype(np.uint8)
7
8  # Create a series of morphed images with different blending
9      ↵ factors
10 for i, alpha in enumerate(np.linspace(0, 1, 11)):    # 11 steps
11     ↵ from 0 to 1
12     morphed_image_array = morph_images(array1, array2, alpha)

```

```

10     morphed_image = Image.fromarray(morphed_image_array)
11     output_path =
12         ↪ f'./image/morph_sequence/morphed_image_{i}.jpg'
13     morphed_image.save(output_path)
14     print(f'Morphed image saved as '{output_path}' with
15         ↪ alpha={alpha:.2f}')

```

3.10 Local (Non-Parametric) Warping

```

1 import cv2
2 import numpy as np
3 from PIL import Image
4 from scipy.spatial import Delaunay
5
6 # Load the two images
7 image_path1 = '../image/okita.png'
8 image_path2 = '../image/kohaku.png'
9 image1 = Image.open(image_path1).convert('RGB') # Convert to
10    ↪ RGB to remove alpha channel
11 image2 = Image.open(image_path2).convert('RGB') # Convert to
12    ↪ RGB to remove alpha channel
13
14 # Resize images to the same size if they are different
15 width = min(image1.width, image2.width)
16 height = min(image1.height, image2.height)
17 image1 = np.array(image1.resize((width, height)))
18 image2 = np.array(image2.resize((width, height)))
19
20 # Example feature points manually defined for simplicity
21 points1 = np.array([[100, 100], [400, 100], [250, 400], [150,
22    ↪ 300], [350, 300]])
23 points2 = np.array([[120, 120], [420, 90], [260, 390], [160,
24    ↪ 320], [370, 280]])
25
26 # Function to warp a single triangle
27 def warp_triangle(img1, img2, t1, t2):
28     # Calculate bounding boxes for the triangles

```

```

28     r1 = cv2.boundingRect(np.float32([t1]))
29     r2 = cv2.boundingRect(np.float32([t2]))
30
31     # Offset points by the bounding box top-left corner
32     t1_offset = [(pt[0] - r1[0], pt[1] - r1[1]) for pt in t1]
33     t2_offset = [(pt[0] - r2[0], pt[1] - r2[1]) for pt in t2]
34
35     # Crop the triangular regions
36     img1_cropped = img1[r1[1]:r1[1] + r1[3], r1[0]:r1[0] +
37                         ↳ r1[2]]
38     mask = np.zeros((r2[3], r2[2], 3), dtype=np.uint8)
39     cv2.fillConvexPoly(mask, np.int32(t2_offset), (1, 1, 1))
40
41     # Compute the affine transform
42     matrix = cv2.getAffineTransform(np.float32(t1_offset),
43                                    ↳ np.float32(t2_offset))
44
45     # Apply affine transformation to the triangular region
46     img2_cropped = cv2.warpAffine(img1_cropped, matrix, (r2[2],
47                                   ↳ r2[3]), None, flags=cv2.INTER_LINEAR,
48                                   ↳ borderMode=cv2.BORDER_REFLECT_101)
49     img2_cropped = img2_cropped * mask
50
51     # Copy the warped triangle to the output image
52     img2[r2[1]:r2[1] + r2[3], r2[0]:r2[0] + r2[2]] =
53         ↳ img2[r2[1]:r2[1] + r2[3], r2[0]:r2[0] + r2[2]] * (1 -
54                         ↳ mask) + img2_cropped
55
56     # Apply the warping on each triangle
57     warped_image = np.copy(image2)
58     for simplex in tri.simplices:
59         # Get the corresponding triangles in both images
60         t1 = points1[simplex]
61         t2 = points2[simplex]
62
63         # Warp the triangle from image1 to image2
64         warp_triangle(image1, warped_image, t1, t2)
65
66         # Save the result
67         warped_image_pil = Image.fromarray(warped_image)
68         warped_image_pil.save('../image/local_warped_image.jpg')
69

```

```
64 print("Local warped image saved as 'local_warped_image.jpg'")
```

3.11 Dense and Sparse Warp Specification

```
1 # Create a dense vector field (swirl effect for demonstration)
2 def create_dense_vector_field(height, width):
3     x, y = np.meshgrid(np.arange(width), np.arange(height))
4     x_center, y_center = width // 2, height // 2
5     angle = np.arctan2(y - y_center, x - x_center)
6     radius = np.sqrt((x - x_center) ** 2 + (y - y_center) ** 2)
7     # Apply a swirl effect based on radius
8     x_displacement = radius * np.cos(angle + radius * 0.01)
9     y_displacement = radius * np.sin(angle + radius * 0.01)
10    return x + x_displacement, y + y_displacement
11
12 # Apply dense warp to the image
13 def dense_warp(image_array, x_field, y_field):
14     map_x = np.float32(x_field)
15     map_y = np.float32(y_field)
16     warped_image = cv2.remap(image_array, map_x, map_y,
17         → interpolation=cv2.INTER_LINEAR,
18         → borderMode=cv2.BORDER_REFLECT_101)
19     return warped_image
20
21 # Interpolate the transformation for sparse warp
22 def sparse_warp(image_array, points_src, points_dst):
23     # Generate a grid of coordinates covering the image
24     grid_x, grid_y = np.meshgrid(np.arange(width),
25         → np.arange(height))
26     grid_coords = np.vstack([grid_x.ravel(), grid_y.ravel()]).T
27
28     # Compute the transformations for each point
29     points_transformed = griddata(points_src, points_dst -
30         → points_src, grid_coords, method='linear', fill_value=0)
31     points_transformed = points_transformed.reshape(height,
32         → width, 2)
33
34     # Apply the transformations
35     map_x = (grid_x + points_transformed[...,
36         → 0]).astype(np.float32)
37     map_y = (grid_y + points_transformed[...,
38         → 1]).astype(np.float32)
```

```

32     sparse_warped_image = cv2.remap(image_array, map_x, map_y,
33                                     interpolation=cv2.INTER_LINEAR,
34                                     borderMode=cv2.BORDER_REFLECT_101)
35     return sparse_warped_image

```

3.12 Manual Projective

```

1 def projective_transform(target_img, src_img, src_points,
2                         dst_points):
3     matrix = cv2.getPerspectiveTransform(src_points,
4                                         dst_points)
5     warped_image = cv2.warpPerspective(src_img, matrix,
6                                         (target_img.shape[1], target_img.shape[0]))
7
8     mask = np.zeros_like(target_img, dtype=np.uint8)
9     cv2.fillConvexPoly(mask, np.int32(dst_points), (255, 255,
10                                            255))
11
12    masked_target = cv2.bitwise_and(target_img,
13                                    cv2.bitwise_not(mask))
14    blended_image = cv2.add(masked_target,
15                           cv2.bitwise_and(warped_image, mask))
16
17    return blended_image

```

3.13 Automatic Projective

```

1 # Function to find the largest rectangle in the image larger
2 # than a certain area threshold
3 def find_large_rectangle(img, area_threshold):
4     # Convert to grayscale and apply edge detection
5     gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
6     blurred = cv2.GaussianBlur(gray, (5, 5), 0)
7     edged = cv2.Canny(blurred, 50, 150)
8
9     # Find contours
10    contours, _ = cv2.findContours(edged, cv2.RETR_EXTERNAL,
11                                    cv2.CHAIN_APPROX_SIMPLE)
12
13    # Initialize variables to store the largest rectangle
14    largest_rect = None

```

```

13     largest_area = 0
14
15     # Iterate through contours to find the largest rectangle
16     for contour in contours:
17         # Approximate the contour
18         epsilon = 0.02 * cv2.arcLength(contour, True)
19         approx = cv2.approxPolyDP(contour, epsilon, True)
20
21         # Check if the contour is a rectangle (4 corners) and
22         # meets the area threshold
23         if len(approx) == 4:
24             area = cv2.contourArea(approx)
25             if area > area_threshold and area > largest_area:
26                 largest_rect = approx
27                 largest_area = area
28
29     # Return the largest rectangle points if found
30     if largest_rect is not None:
31         return np.float32([point[0] for point in largest_rect])
32     else:
33         return None
34
35     # Function to apply projective transformation using the found
36     # rectangle
37     def projective_transform_with_rectangle(target_img, src_img,
38         area_threshold):
39         # Find the largest rectangle in the target image
40         rect_points = find_large_rectangle(target_img,
41             area_threshold)
42         if rect_points is None:
43             print("No rectangle found that meets the area
44             threshold.")
45         return target_img
46
47         # Define source points (corners of the Keanu image)
48         keanu_height, keanu_width = src_img.shape[:2]
49         keanu_points = np.float32([[0, 0], [keanu_width, 0],
50             [keanu_width, keanu_height], [0, keanu_height]])
51
52         # Compute the perspective transformation matrix
53         matrix = cv2.getPerspectiveTransform(keanu_points,
54             rect_points)

```

```

48
49      # Warp the source image to fit the detected rectangle
50      warped_image = cv2.warpPerspective(src_img, matrix,
51          ↳ (target_img.shape[1], target_img.shape[0]))
52
53      # Create a mask for the warped area
54      mask = np.zeros_like(target_img, dtype=np.uint8)
55      cv2.fillConvexPoly(mask, np.int32(rect_points), (255, 255,
56          ↳ 255))
57
58      # Blend the warped image into the target image using the
59          ↳ mask
60      masked_target = cv2.bitwise_and(target_img,
61          ↳ cv2.bitwise_not(mask))
62      blended_image = cv2.add(masked_target,
63          ↳ cv2.bitwise_and(warped_image, mask))
64
65      return blended_image

```

4 Kết quả và Thảo luận

4.1 Kết quả sau khi áp dụng các phép biến đổi đã học



Hình 1: Hình ảnh gốc.



Hình 2: Hình ảnh đã áp dụng translation -30 -15.



Hình 3: Hình ảnh đã áp dụng translation 50 20.



Hình 4: Hình ảnh đã áp dụng rotation góc 45 độ.



Hình 5: Hình ảnh đã áp dụng rotation góc 90 độ.



Hình 6: Hình ảnh đã áp dụng uniform scaling 0.5.



Hình 7: Hình ảnh đã áp dụng uniform scaling 1.5.



Hình 8: Hình ảnh đã áp dụng non-uniform scaling 0.5 1.5.



Hình 9: Hình ảnh đã áp dụng non-uniform scaling 2.0 0.5.



Hình 10: Hình ảnh đã áp dụng shearing x 0.3.



Hình 11: Hình ảnh đã áp dụng shearing y 0.3.



Hình 12: Hình ảnh đã áp dụng mirroring x.



Hình 13: Hình ảnh đã áp dụng mirroring y.



Hình 14: Hình ảnh đã áp dụng forward warping.



Hình 15: Hình ảnh đã áp dụng inverse warping với hình ảnh đã áp dụng rotation gốc 45 độ.



Hình 16: Hình ảnh dùng cho morphing, local warping, dense, sparse warp.



Hình 17: Hình ảnh dùng cho morphing, local warping, dense, sparse warp.



Hình 18: Áp dụng morphing từ trên xuống trái sang.



Hình 19: Hình ảnh áp dụng local warping.



Hình 20: Hình ảnh áp dụng dense warp.



Hình 21: Hình ảnh áp dụng sparse warp.

4.2 So sánh đánh giá Affine và Projective Transformation

- **Tính bảo toàn song song:** Affine Transformation bảo toàn các đường song song, trong khi Projective Transformation có thể khiến các đường song song hội tụ, tạo ra cảm giác về phối cảnh.
- **Ứng dụng:** Affine Transformation thường được sử dụng trong các thao

tác chỉnh sửa hình học đơn giản như dịch chuyển, xoay, và phóng to/thu nhỏ, trong khi Projective Transformation được sử dụng để tạo hiệu ứng phối cảnh trong các ứng dụng như đồ họa 3D và thị giác máy tính.

- **Độ phức tạp:** Affine Transformation có độ phức tạp tính toán thấp hơn, vì nó chỉ yêu cầu một ma trận 2×3 , trong khi Projective Transformation yêu cầu ma trận 3×3 và tính toán phức tạp hơn để mô hình hóa phối cảnh.

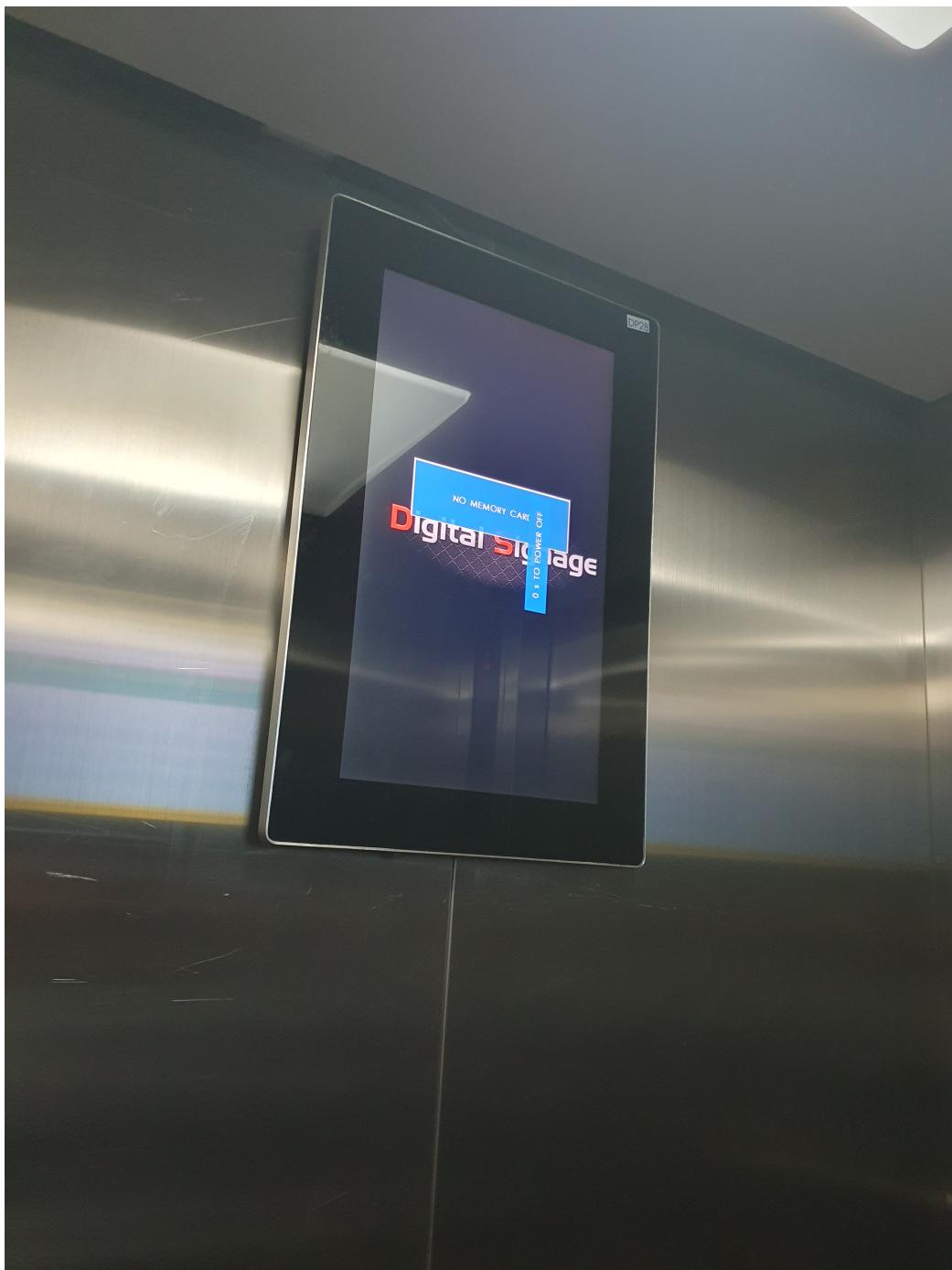


Hình 22: Hình ảnh áp dụng affine transformation.



Hình 23: Hình ảnh áp dụng projective transformation.

4.3 Thủ nghiệm dán 1 hình chữ nhật/vuông lên mặt phẳng chỉ định trước



Hình 24: Hình ảnh chứa mặt phẳng chỉ định 1.



Hình 25: Hình ảnh chứa mặt phẳng chỉ định 2.



Hình 26: Keanu Reeves.



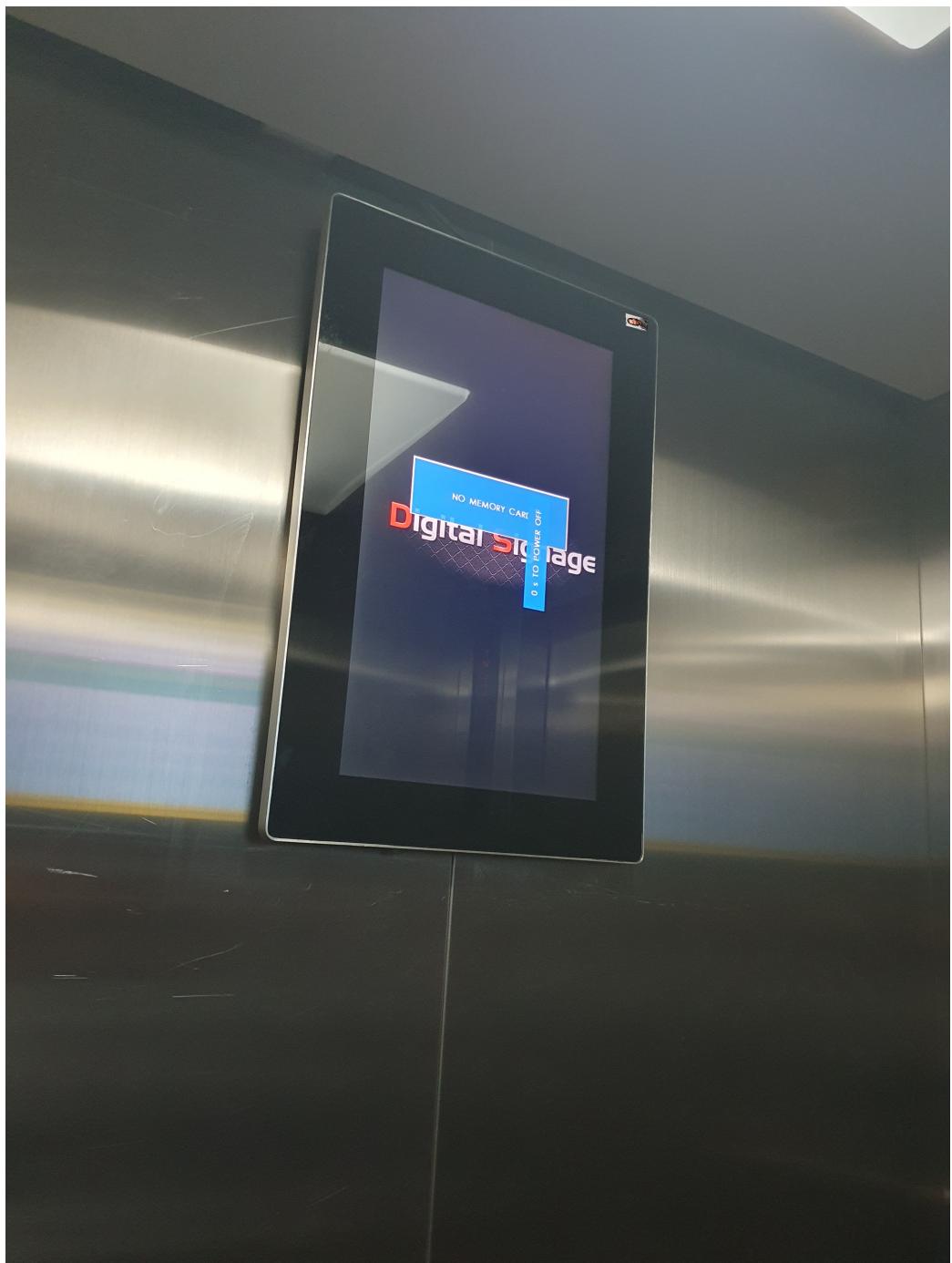
Hình 27: Hình ảnh khi dán thủ công 1.



Hình 28: Hình ảnh khi dán thủ công 2.

4.3.1 Thảo luận về Giải thuật Automatic Projective

Giải thuật `find_large_rectangle` có chức năng tìm hình chữ nhật lớn nhất trong ảnh, với diện tích lớn hơn một ngưỡng nhất định, và sau đó áp dụng phép biến đổi phối cảnh để phù hợp với đối tượng nguồn (`src_img`) trong vùng này. Tuy nhiên, giải thuật này có một hạn chế quan trọng khi không thể phát hiện đúng các hình chữ nhật hoặc hình vuông trong ảnh nếu góc chụp không duy trì sự song song của các cạnh.



Hình 29: Hình ảnh khi dán tự động 1.



Hình 30: Hình ảnh khi dán tự động 2.

Cụ thể:

- Giải thuật sử dụng hàm `cv2.approxPolyDP` để xấp xỉ các đường bao

của hình dạng và kiểm tra xem chúng có bốn cạnh hay không để xác định hình chữ nhật. Tuy nhiên, khi ảnh được chụp từ một góc nghiêng, các cạnh song song của hình chữ nhật hoặc hình vuông trong thực tế có thể không còn song song trong ảnh do hiệu ứng phối cảnh.

- Do không có các kiểm tra bổ sung để xử lý biến dạng phối cảnh, giải thuật này chỉ có thể phát hiện các hình chữ nhật có các cạnh song song trong hệ tọa độ ảnh, tức là các hình chữ nhật không bị biến dạng hoặc chỉ biến dạng nhẹ.
- Để cải thiện, có thể sử dụng thêm các phương pháp phát hiện phối cảnh như `cv2.findHomography` để xác định các điểm ảnh và cấu trúc không song song nhưng có thể khớp với hình chữ nhật nếu áp dụng biến đổi phối cảnh.

Như vậy, giải thuật này chỉ phù hợp để phát hiện các hình chữ nhật khi góc chụp gần như trực diện. Nếu ảnh được chụp từ góc nghiêng hoặc với phối cảnh mạnh, các hình chữ nhật hoặc hình vuông trong thế giới thực có thể không được phát hiện chính xác bởi giải thuật này.

5 Kết luận

Chúng ta đã khám phá và thực hiện nhiều kỹ thuật biến đổi hình ảnh khác nhau, từ các phép biến đổi cơ bản như dịch chuyển, xoay, và phóng đại/thu nhỏ, đến các kỹ thuật phức tạp hơn như biến đổi affine, biến đổi phối cảnh, và các phương pháp warping. Mỗi phép biến đổi đều có vai trò quan trọng và ứng dụng riêng trong xử lý ảnh số, giúp thay đổi cấu trúc không gian và giá trị cường độ của ảnh để đáp ứng các yêu cầu xử lý khác nhau.

Thông qua quá trình thử nghiệm, chúng ta nhận thấy rằng các phép biến đổi tuyến tính như affine transformation giữ nguyên tính song song của các đường và phù hợp cho các ứng dụng chỉnh sửa hình học đơn giản. Trong khi đó, phép biến đổi phối cảnh (projective transformation) cung cấp khả năng mô phỏng hiệu ứng chiều sâu và phối cảnh, giúp xử lý các ảnh chụp từ góc nghiêng. Tuy nhiên, thuật toán tìm hình chữ nhật tự động dựa trên projective transformation vẫn gặp hạn chế khi phải xử lý các hình chữ nhật bị biến dạng mạnh do phối cảnh.

Ngoài ra, các kỹ thuật warping như forward warping, inverse warping, morphing, và local warping cũng được áp dụng để tạo ra các hiệu ứng đặc biệt trong xử lý ảnh. Những kỹ thuật này cho phép chúng ta linh hoạt trong

việc kết hợp hoặc làm biến dạng ảnh một cách cục bộ hoặc toàn cục, tạo ra các hiệu ứng chuyển tiếp hoặc thay đổi hình dáng linh hoạt.

Tóm lại, các kỹ thuật biến đổi hình ảnh không chỉ mở ra nhiều hướng ứng dụng trong xử lý ảnh và thị giác máy tính mà còn là nền tảng cho các phương pháp xử lý phức tạp hơn. Các thử nghiệm trong báo cáo này đã minh chứng rõ ràng vai trò của các phép biến đổi hình học và cho thấy tiềm năng của chúng trong việc phát triển các ứng dụng đa dạng từ chỉnh sửa hình ảnh đến nhận dạng và theo dõi đối tượng. Những hướng phát triển tiếp theo có thể bao gồm việc kết hợp các kỹ thuật này với các phương pháp học máy và thị giác máy tính để nâng cao khả năng tự động nhận diện và xử lý các đối tượng trong các bối cảnh phức tạp.

6 Code đầy đủ

Mã nguồn đầy đủ có thể truy cập tại GitHub: ComputerVisionAssignment.