EXERCISE 2:
MOVING LEAST SQUARES FOR IMAGE MANIPULATION

GCT722 MATHEMATICAL METHODS FOR VISUAL COMPUTING

20183151 Chaelin Kim

PART 1: IMAGE DEFORMATION USING MOVING LEAST SQUARES

Description of implementation

There are 1 script file & 8 function files for the exercise image deformation.

- **Script file**
  - **main.m**

    : This is the script for doing image deformation. It calls functions for making parameters (calWeight, calStar, calHat), doing several deformations (doAffineDeform, doSimilarityDeform, doRigidDeform) and making deformed image (makeDefImg, makeDefImgBack). Details of these functions are explained below. We can set the initial settings in this script.

```
%% Initial Setting
imgFilePath = '../materials/ginger.png';
srcImg = imread(imgFilePath);

[rows, columns, ~] = size(srcImg);

% Create a mesh grid for all the coordinate in the matrix
[X, Y] = meshgrid(1:columns, 1:rows);

% Make the image to pixel point list (rows x columns) x 1
v = reshape([X Y], [], 2);
vLen = size(v,1);
```

To select points in the image, it shows the original image. You can select source control points in the image and press the 'enter' key to end it. Then, you have to select target points to match the number of source points.

```
%% Show the original image
% Select some input and output control points
figure('units','pixels','pos',[100 100 ((columns * 4)) ((rows * 2))])
subplot(2, 4, [1,5]);
imshow(srcImg)
title('Original Image')
hold on;
sourceCP = ginput;
plot(sourceCP(:, 1), sourceCP(:, 2), 'o', 'Color', 'g')
targetCP = ginput(size(sourceCP, 1));
plot(targetCP(:, 1), targetCP(:, 2), 'x', 'Color', 'r')
hold off;
```

After that, functions that used to set parameters (weight, star, hat) and do deformation are called in order. When process of deformations is over, it shows the result images shown next to the original image.

- **Function files**

  ➢ **calWeight.m**

  : This function is used to calculate weight values. The weights have the form

$$w_i = \frac{1}{|p_i - v|^{2\alpha}}$$

The denominator of weight is the 2α square value of distance between source control points p and points v in the image. Because the weights w_i in this least squares problem are dependent on the point of evaluation v, we call this a *Moving Least Squares* minimization. Therefore, we obtain a different transformation for each v.

```
function [ weight ] = calWeight( v, sourceCP, alpha )
```

- Input Value
  - *v*: The array of data points (x, y) in the original image.
    (size: (rows * columns) x 2)
  - *sourceCP*: The array of source control points (x, y)
  - *alpha*: The alpha value used in calculating 2α
- Output
  - *weight*: The array of weights for each points of original image.

- **calStar.m**

  : This function is used to calculate weighted centroids of source and target control points.

$$p_* = \frac{\sum_i w_i p_i}{\sum_i w_i} \quad q_* = \frac{\sum_i w_i q_i}{\sum_i w_i}$$

  In main.m, this function is called for p_* and q_*.

```
function [ resultStar ] = calStar( weight, vLength, controlPoint )
```

  - Input Value
    - *weight*: The array of weights for each points of original image
    - *vLength*: The length of data points (x, y) array in the original image
    - *controlPoint*: The array of control points.
  - Output
    - *resultStar*: The array of computed centroid data for each points of original image. (size: (rows * columns) x 2)

- **calHat.m**

  : This function is used to calculate hat values of source and target control points. The hat value means the difference of control point and weighted centroid.

$$\hat{p}_i = p_i - p_* \quad \hat{q}_i = q_i - q_*$$

  In main.m, this function is called for $\hat{p}_i$ and $\hat{q}_i$.

```
function [ resultHat ] = calHat( vLength, controlPoint, cpStar )
```

  - Input Value
    - *vLength*: The length of data points (x, y) array in the original image
    - *controlPoint*: The array of control points
    - *cpStar*: The array of weighted centroids
  - Output
    - *resultHat*: The array of computed hat values for each points of original image. (size: (rows * columns) x 2 x (length of control points))

- **doAffineDeform.m**

  : This function is used to deform the image with affine transformation. Affine transformations contain shear and non-uniform scaling. In the paper [Schaefer et al. 2006], the deformation function can be expressed like:

$$f_a(v) = \sum_j A_j \hat{q}_j + q_* \ where \ A_j = (v - p_*) \left(\sum_i \hat{p}_i^T w_i \hat{p}_i \right)^{-1} w_j \hat{p}_j^T$$

  It returns the coordinates array deformed with affine transformation.

For backward warping, I implemented the code for it using inverse matrix but it is not used because of accordance with similarity and rigid. In main.m, this function is called same as forward warping except exchanging source and target.

```
function [ affineDef ] = doAffineDeform( weight, v, sourceCP, targetCP, pstar, phat, qstar, qhat )
```

- Input Value
  - *weight*: The array of weights for each points of original image
  - *v*: The array of data points (x, y) in the original image.
    (size: (rows * columns) x 2)
  - *sourceCP*: The array of source control points (x, y)
  - *targetCP*: The array of target control points (x, y)
  - *pstar*: The array of weighted centroids of source control points
  - *phat*: The array of difference of source control points and pstar
  - *qstar*: The array of weighted centroids of target control points
  - *qhat*: The array of difference of target control points and qstar
- Output
  - *affineDef*: The array of result coordinates (x, y) for deformed image with affine transformation. (size: (rows * columns) x 2)

doSimilarityDeform.m

: This function is used to deform the image with similarity transformation. Similarity transformations are a subset of affine transformations and contain translation, rotation and uniform scaling. In the paper [Schaefer et al. 2006], the deformation function can be expressed like:

$$f_s(v) = \sum_i \hat{q}_i(\frac{1}{\mu_s}A_i) + q_* \ where\ A_i = w_i \begin{pmatrix} \hat{p}_i \\ -\hat{p}_i^{\perp} \end{pmatrix} \begin{pmatrix} (v - p_*) \\ -(v - p_*)^{\perp} \end{pmatrix}^T$$

It returns the coordinates array deformed with similarity transformation.

For backward warping, it is called same as forward warping except exchanging source and target in main.m.

```
function [ similarityDef ] = doSimilarityDeform( weight, v, sourceCP, targetCP, pstar, phat, qstar, qhat )
```

- Input Value
  - *weight*: The array of weights for each points of original image
  - *v*: The array of data points (x, y) in the original image.
    (size: (rows * columns) x 2)
  - *sourceCP*: The array of source control points (x, y)
  - *targetCP*: The array of target control points (x, y)
  - *pstar*: The array of weighted centroids of source control points

    - *phat*: The array of difference of source control points and pstar
    - *qstar*: The array of weighted centroids of target control points
    - *qhat*: The array of difference of target control points and qstar
- Output
    - *similarityDef*: The array of result coordinates (x, y) for deformed image with similarity transformation. (size: (rows * columns) x 2)

➢ doRigidDeform.m

: This function is used to deform the image with rigid transformation. Rigid transformations are related to similarity transformations and contain translation and rotation. In the paper [Schaefer et al. 2006], the deformation function can be expressed like:

$$f_s(v) = |v - p_*| \frac{\vec{f}_r(v)}{|\vec{f}_r(v)|} + q_* \; where \; \vec{f}_r(v) = \sum_i \hat{q}_i A_I$$

It returns the coordinates array deformed with rigid transformation.

For backward warping, it is called same as forward warping except exchanging source and target in main.m

```
function [ rigidDef ] = doRigidDeform( weight, v, sourceCP, targetCP, pstar, phat, qstar, qhat )
```

- Input Value
    - *weight*: The array of weights for each points of original image
    - *v*: The array of data points (x, y) in the original image.
      (size: (rows * columns) x 2)
    - *sourceCP*: The array of source control points (x, y)
    - *targetCP*: The array of target control points (x, y)
    - *pstar*: The array of weighted centroids of source control points
    - *phat*: The array of difference of source control points and pstar
    - *qstar*: The array of weighted centroids of target control points
    - *qhat*: The array of difference of target control points and qstar
- Output
    - *rigidDef*: The array of result coordinates (x, y) for deformed image with rigid transformation. (size: (rows * columns) x 2)

➢ makeDefImg.m

: This function makes the deformed image with deformed coordinates for forward warping. Color values at each points in the original image are assigned to the result points in the result image.

```
function [defImg] = makeDefImg( originImg, defCoord, boolInterp )
```

- Input Value
  - *originImg*: The origin image called by imread
    (size: rows x columns x (number of color channels))
  - *defCoord*: The array of computed coordinates (x, y) for target image. It is resulted from forward warping (affine, similarity or rigid transformation)
    (size: (rows * columns) x 2)
- Output
  - *defImg*: The array of computed centroid data for each points of original image.
    (size: rows x columns x (number of color channels))

➢ **makeDefImgBack.m**

: This function makes the deformed image with deformed coordinates for backward warping. It is divided into three parts depending on whether or not interpolation is performed and using *griddata* function or not. If the parameter *interpM* is 'T', the interpolation is performed using implementation of bilinear interpolation. Bilinear interpolation is a linear interpolation in 2-dimension. It interpolates four points in each of the height and width, and then interpolates two results to obtain the result point. If the result coordinates are out of range, color value in that coordinate is set to white.
If the parameter *interpM* is 'F', color values at computed points in the original image are assigned to the points in the result image. If *interpM* is 'G', the interpolation is performed using *griddata*.

```
function [ defImg ] = makeDefImgBack( originImg, bDefCoord, interpM )
```

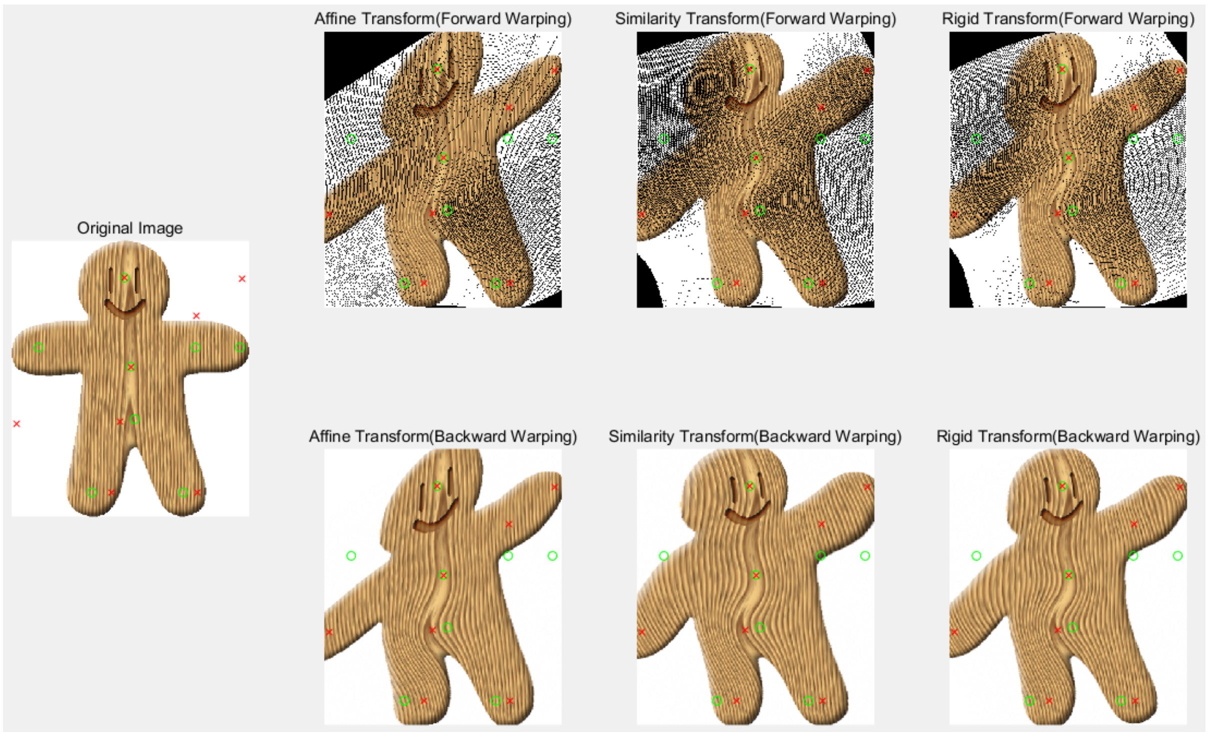
- Input Value
  - *originImg*: The origin image called by imread.
    (size: rows x columns x (number of color channels))
  - *bDefCoord*: The array of computed coordinates (x, y) for source image. It is resulted from backward warping (affine, similarity or rigid transformation)
    (size: (rows * columns) x 2)
  - *interpM*: The variable that determines whether or not to interpolate. It can have three values, 'T', 'F' and 'G' depending on whether interpolation is used and using griddata or not.
    ('T': use interpolation, 'F': not use interpolation, 'G': use interpolation with griddata)
- Output
  - *defImg*: The array of computed centroid data for each points of original image.
    (size: rows x columns x (number of color channels))

Instructions for running

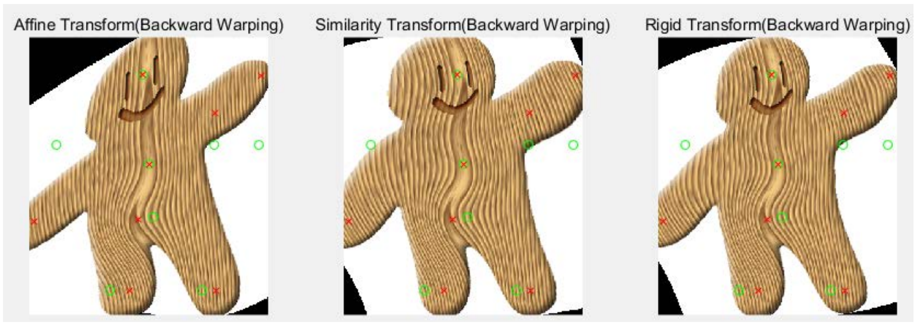
1. Open the file "main.m" in Matlab.
2. Execute that file.
    a. The window that shows the original image is opened.
3. Select source control points in the original image and if you are done, press the enter key.
4. Select target control points in the original image. You have to select them to match the order of selecting source control points and the number of them.
5. The result images of affine, similarity and rigid deformation is shown next to the original image. Result images are composed of the result of forward warping and backward warping for each deformation method.
    a. Result images are saved in './materials/resultImage'.

Screenshots

- **Forward & Backward warping (with interpolation) (Ginger man)**

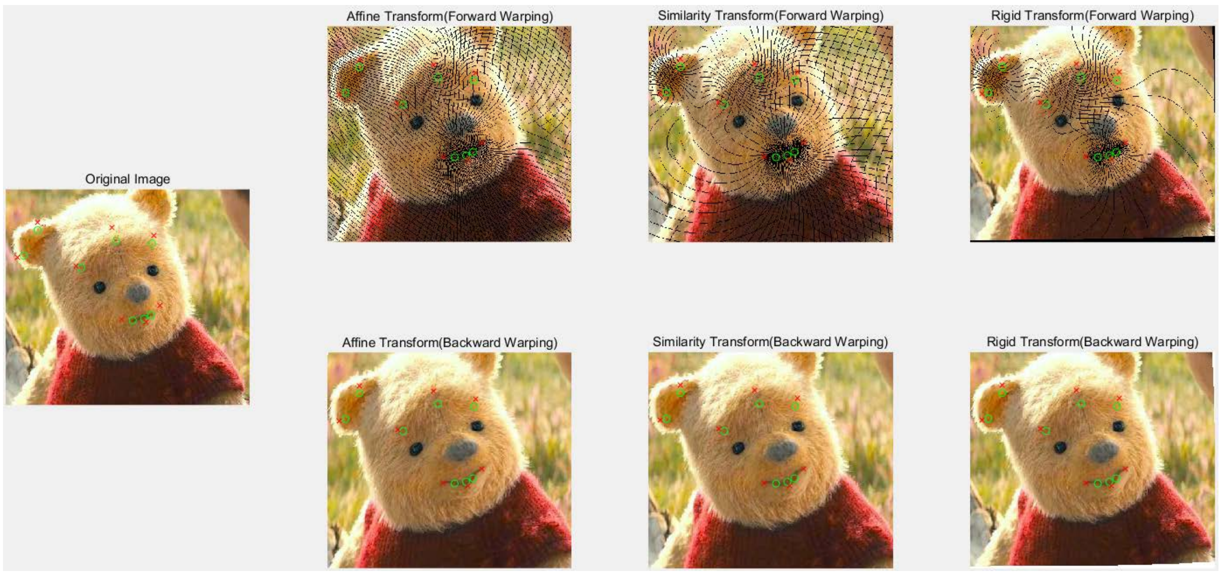

Affine Transform(Forward Warping)
Similarity Transform(Forward Warping)
Rigid Transform(Forward Warping)
Original Image
Affine Transform(Backward Warping)
Similarity Transform(Backward Warping)
Rigid Transform(Backward Warping)

- **Interpolation with griddata function in backward warping (Ginger man)**

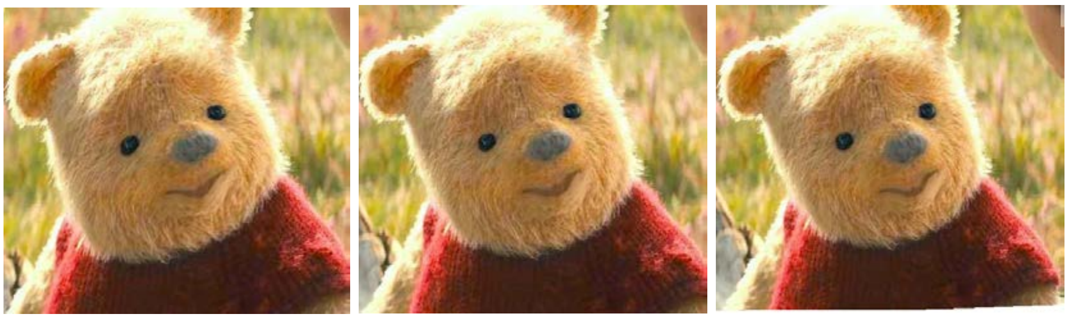

Affine Transform(Backward Warping)
Similarity Transform(Backward Warping)
Rigid Transform(Backward Warping)

→ Same as above result except unknown pixels are filled with black color

- **Forward & Backward warping with another image (make the Pooh happy :D)**


Original Image
Affine Transform(Forward Warping)
Similarity Transform(Forward Warping)
Rigid Transform(Forward Warping)
Affine Transform(Backward Warping)
Similarity Transform(Backward Warping)
Rigid Transform(Backward Warping)

(Original)



(Affine) **(Similarity)** **(Rigid)**

Discuss the results

I have implemented image deformation with the affine, similarity and rigid transformations according to the reference paper “Image Deformation Using Moving Least Squares [Schaefer et al. 2006]” with Matlab.

Affine transformations contain shear and non-uniform scaling. Because it is simple than other transformations, the points except the control points are also very affected by direction of transformation for control points. So the result image appears to be larger and stretched globally. Similarity transformations contain translation, rotation and uniform scaling, so it preserves angles in the original image better than affine. The result is more natural than Affine, but the result is stretched a little over the original because of scaling. Rigid transformations, which removes the scaling from the similarity transformation, show the most realistic result image among them. The parts except control points are almost not deformed.

Through the above transformations, I have implemented forward warping and backward warping. Forward warping is a method of calculating the coordinates of the target with respect to the source coordinates to fill the color. Because the pixels in the image are integer, they need to be rounded. Then some pixels in the result image cannot be computed and holes (unknown values) are shown in the image. In order to correct, backward warping can be used. Backward warping is the reverse of forward warping. Since this can be seen as a transformation from target points to source points, I implemented it by reversing the source and target in the forward warping. The results are a little different with those of forward warping, but each result are performed well in each transformation. In addition, the bilinear interpolation is needed for antialiasing.