Course Title (과목명)	디지털영상처리개론 (01)
HW Number (HW 번호)	HW01
Submit Date (제출일)	2021-04-01
Grade (학년)	4th Grade
ID (학번)	20161482
Name (이름)	박준용

Affine transform of digital images

Affine matrix를 이용한 geometric transformation을 진행함

- **openCV의 Mat class method 및 입/출력 함수(imread와 imwrite)만을 사용함. (즉, openCV에서 제공하는 영상처리 관련 함수들은 사용을 하지 않음.)**
- **제공된 HW01.cpp 및 utils.h파일은 수정하지 않고, utils.cpp파일을 작성하여 제출함.**

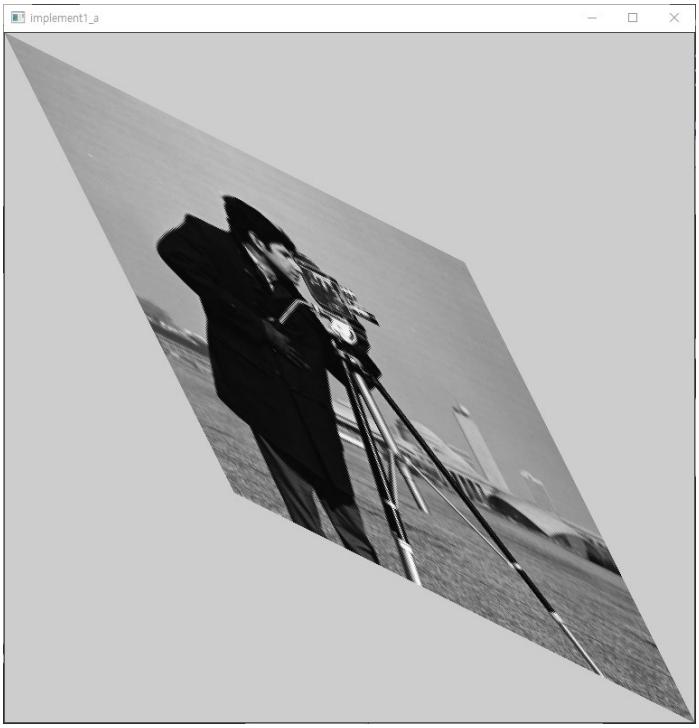
(1) 구현 1

- Affine matrix $\mathbf{A}, \mathbf{B}$를 다음과 같이 정함.

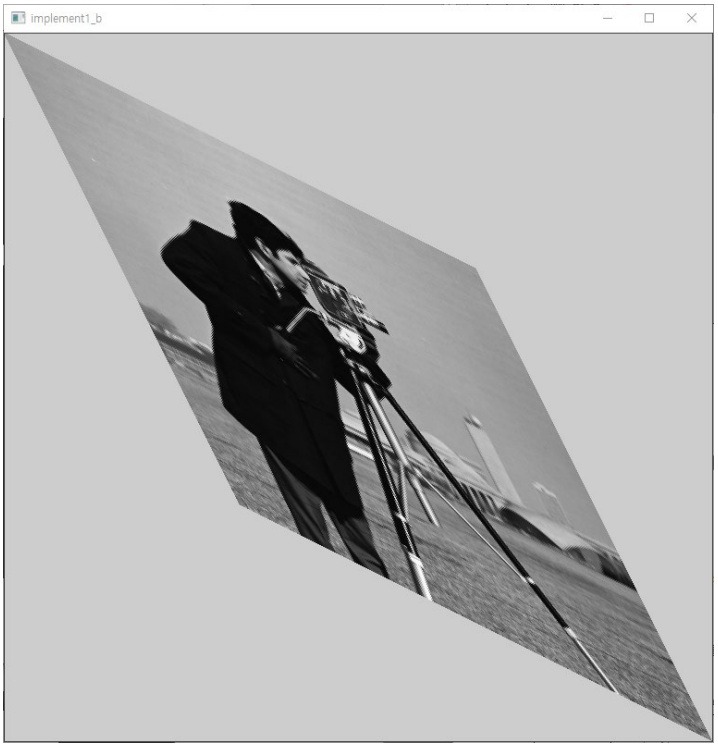
$$\mathbf{A} = \begin{bmatrix} 2.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}, \mathbf{B} = \begin{bmatrix} 1.0 & 0.5 & 0.0 \\ 0.5 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

Affine Matrix A는 이미지를 2배로 resizing하는 matrix이며, Affine Matrix B의 경우에는 각 0.5만큼 동시에 vertical, horizontal shear 하는 matrix이다.

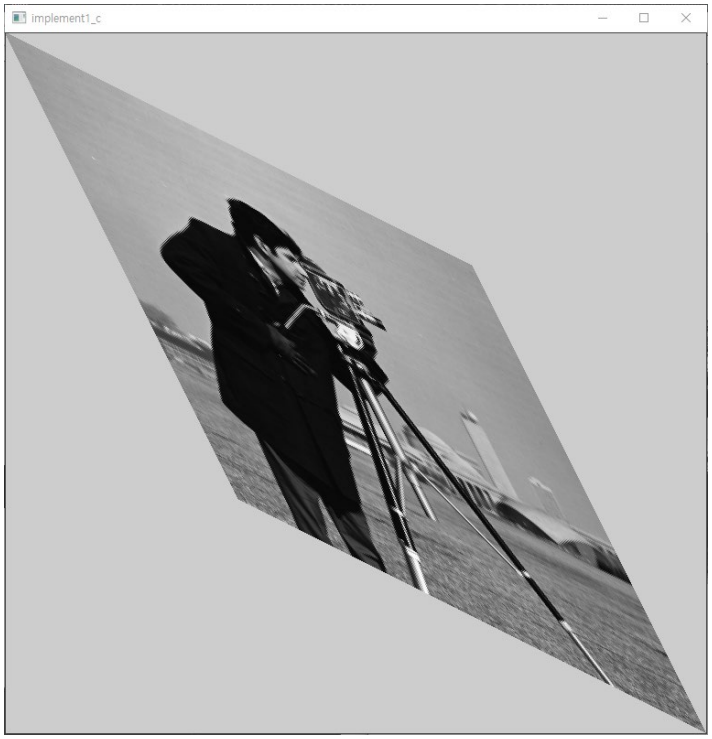
a) *.zip에 포함된 **"cameramen.tif"**에 **A**를 먼저 적용한 후 **B**를 적용한 transform action 결과를 보이시오.



b) $\mathbf{C}(=\mathbf{BA})$를 이용하여 transform한 결과를 보이시오.



c) 순서를 바꾸어 $\mathbf{C}'(=\mathbf{AB})$를 이용하여 transform한 결과를 보이시오.



- a)와 b)의 결과를 **비교**하시오.

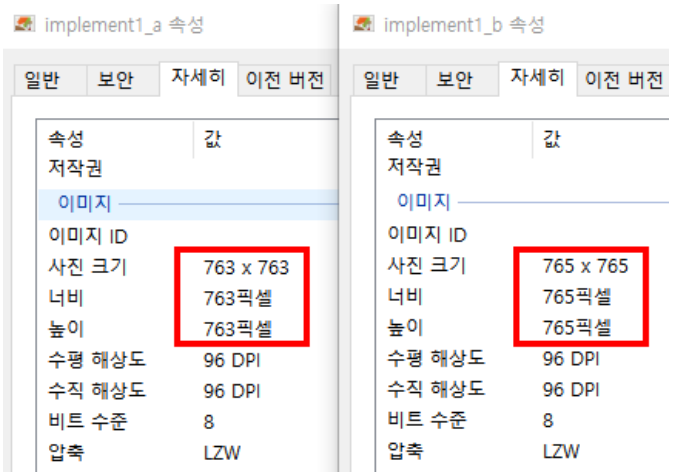
우선, a) 와 같이 Affine matrix A를 먼저 적용하여 transformation을 한 뒤, matrix B를 적용한 결과는 다음과 같다.

$$A * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 2.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 2x \\ 2y \\ 1 \end{bmatrix}$$

$$B * \begin{bmatrix} 2x \\ 2y \\ 1 \end{bmatrix} = \begin{bmatrix} 1.0 & 0.5 & 0.0 \\ 0.5 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} 2x \\ 2y \\ 1 \end{bmatrix} = \begin{bmatrix} 2x + y \\ x + 2y \\ 1 \end{bmatrix}$$

같은 형태로 b)의 transformation 결과는 다음과 같다.

$$C * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = (BA) * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 2.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} 1.0 & 0.5 & 0.0 \\ 0.5 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 2.0 & 1.0 & 0.0 \\ 1.0 & 2.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 2x + y \\ x + 2y \\ 1 \end{bmatrix}$$



이러한 형식으로 계산되므로 수학적으로는 a)의 결과와 b)의 결과가 동일하다. 그러나 a)와 b)의 image size를 확인할 경우 a)의 image가 b)보다 2px 줄어든 것을 확인할 수 있다. 이는 Affine Transformation을 시행할 때, matrix multiplication을 2회 적용하였기 때문에 일어난 변경점이라고 생각할 수 있다.

- b)와 c)의 결과에 **차이점**이 있는지 확인하고 차이점이 발생하는(또는 발생하지 않는) 이유를 설명하시오.

위 구현 결과와 비교할 수 있도록 c)의 transformation 결과를 계산할 수 있다.

$$C' * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = (AB) * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 2.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} 1.0 & 0.5 & 0.0 \\ 0.5 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 2.0 & 1.0 & 0.0 \\ 1.0 & 2.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 2x + y \\ x + 2y \\ 1 \end{bmatrix}$$

두 수식의 결과가 같아 수학적으로 b)와 c)의 결과가 동일한 것을 확인할 수 있으며, 두 경우 모두 matrix multiplication을 1회만 적용하였으므로 동일한 image size를 가지는 것 역시 확인하였다.

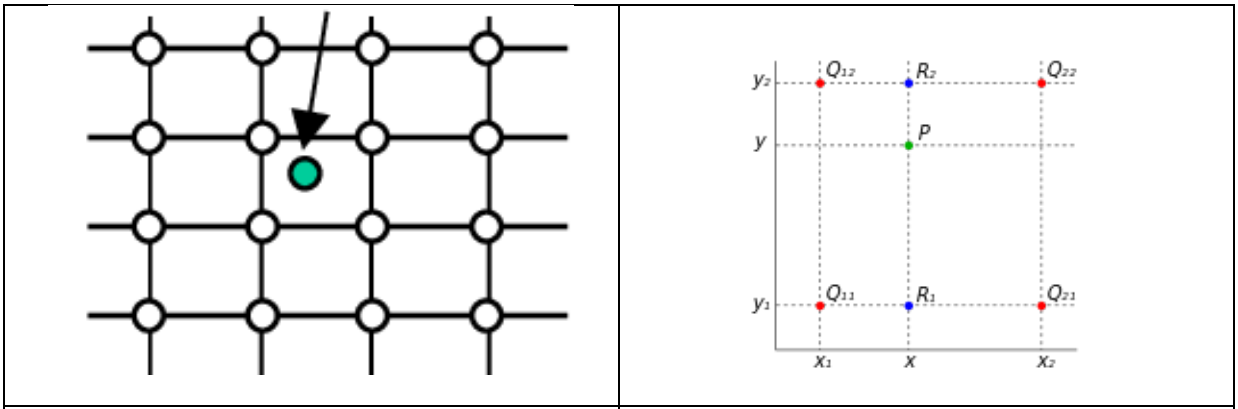
✓ 아래 6.3)를 참고하여 warpAffine()을 작성하시오.

```
float interp2D(Mat src, float x, float y, string interp_type) {
        if (interp_type == "nearest") {
                if (x - (int)x < 0.5 && y - (int)y < 0.5) {
                        return src.at<uchar>(y, x);
                }
                else if (x - (int)x < 0.5 && y - (int)y >= 0.5) {
                        return src.at<uchar>(y + 1, x);
                }
                else if (x - (int)x >= 0.5 && y - (int)y < 0.5) {
                        return src.at<uchar>(y, x + 1);
                }
                else if (x - (int)x >= 0.5 && y - (int)y >= 0.5) {
                        return src.at<uchar>(y + 1, x + 1);
                }
        }
        else if (interp_type == "bilinear") {
                if (y - int(y) == 0 && x - int(x) == 0)
                        return src.at<uchar>(y, x);
                else {
                        return src.at<uchar>(int(y), int(x)) * (int(x) - x + 1) * (int(y) - y
+ 1) + src.at<uchar>(int(y) + 1, int(x)) * (int(x) - x + 1) * (y - (int)y)
                                + src.at<uchar>(int(y), int(x) + 1) * (x - (int)x) * (int(y)
- y + 1) + +src.at<uchar>(int(y) + 1, int(x) + 1) * (x - (int)x) * (y - (int)y);
                }
        }
}

Mat warpAffine(Mat src, Mat affmat, string interp_type) {
        Mat original = Mat::zeros(Size(2, 2), CV_32F);
        Mat invorg = Mat::zeros(Size(2, 2), CV_32F);
        Mat implement = Mat::zeros(Size(1, 3), CV_32F);
        float x, y, x_min=INF, y_min=INF, x_max = -INF, y_max = -INF;
        float* orig = (float*)original.data;
        orig[0] = affmat.at<float>(0, 0); orig[1] = affmat.at<float>(1, 0);
        orig[2] = affmat.at<float>(0, 1); orig[3] = affmat.at<float>(1, 1);

        for (int x_prime = 0; x_prime < src.rows; x_prime++) {
                for (int y_prime = 0; y_prime < src.cols; y_prime++) {
                        float* imp_ele = (float*)implement.data;
                        imp_ele[0] = y_prime * affmat.at<float>(0, 0) + x_prime *
affmat.at<float>(0, 1);
                        imp_ele[1] = y_prime * affmat.at<float>(1, 0) + x_prime *
affmat.at<float>(1, 1);
                        imp_ele[2] = 1;
                        if (imp_ele[0] < x_min)
                                x_min = imp_ele[0];
                        if (imp_ele[0] > x_max)
                                x_max = imp_ele[0];
                        if (imp_ele[1] < y_min)
                                y_min = imp_ele[1];
                        if (imp_ele[1] > y_max)
                                y_max = imp_ele[1];
                }
        }
```

```
        Mat dst(y_max - y_min + affmat.at<float>(1, 2),
                x_max - x_min + affmat.at<float>(0, 2),
                src.type());
        for (int x_prime = 0; x_prime < dst.rows; x_prime++) {
                for (int y_prime = 0; y_prime < dst.cols; y_prime++) {
                        invorg = original.inv();
                        x = (y_prime + x_min - affmat.at<float>(0, 2)) * invorg.at<float>(1,
1) + (x_prime + y_min - affmat.at<float>(1, 2)) * invorg.at<float>(1, 0);
                        y = (y_prime + x_min - affmat.at<float>(0, 2)) * invorg.at<float>(0,
1) + (x_prime + y_min - affmat.at<float>(1, 2)) * invorg.at<float>(0, 0);
                        if (int(y) < 0 || int(x) < 0 || (int(x)+1) > src.cols - 1 ||
(int(y)+1) > src.rows - 1)
                                continue;
                        dst.at<uchar>(x_prime, y_prime) = interp2D(src, x,y, interp_type);
                }
        }
        return dst;
};
```



Nearest-Neighbor Interpolation	Bilinear Interpolation

우선, warpAffine() 함수를 구현하기 전에 interpolation 형식을 구현하기 위한 interp2D() 함수를 작성하였다. Interp2D는 원 image, x좌표, y좌표, interpolation 방법을 input으로 받는다.

먼저 nearest interpolation의 경우, x와 y좌표의 소수점에 가장 인접한 integer pixel의 intensity를 출력하는 interpolation 방법이다. 이를 code에서는 x값과 y값의 float값에서 해당 int 값을 빼 주어 0~1 사이의 값으로 만든 뒤, 0.5를 기준으로 Euclidian Distance가 가까운 쪽의 pixel로 return하는 code를 작성하는 것으로 구현하였다.

또한, bilinear interpolation의 경우는 인접 4개의 pixel의 가중치에 따라 intensity를 계산하는 interpolation 방법이다. 위의 figure에서 pixel P의 경우, intensity를 $f(Q_{11}) * (x_2 - x) * (y_2 - y) + f(Q_{12}) * (x_2 - x) * (y - y_1) + f(Q_{21}) * (x - x_1) * (y_2 - y) + f(Q_{22}) * (x - x_1) * (y - y_1)$ 로 계산할 수 있으며, 이는 x1과 x2의 사이를 int(x)와 int(x)+1, y1과 y2의 사이를 int(y)와 int(y)+1 로 구현한 뒤 이를 사용하여 Q의 값을 나타내어 intensity를 return하는 code를 작성할 수 있었다.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \mathbf{A} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

다음으로, warpAffine() 함수는 affine transform을 시행하기 위한 함수로, 원 image, affine matrix, interpolation 방법을 input으로 받는다. 우선 Mat original의 경우, affine transform 대상의 2-by-2 element 값을 받으며, 이를 Mat invorg을 통하여 inverse matrix 값을 구한다.

이 때, image가 rotate 또는 translation되어도 모두 표시되게 하기 위해서는 추가적으로 image의 길이와 넓이에 대한 설정이 필요하다. 위의 식과 같이, 원본 영상의 size만큼 Mat Implement에 x'과 y'의 값을 통하여 길이를 받아들이며 수식에서 구현한 Mat implement element 는 다음과 같다.

$$\begin{bmatrix} y' * a_{11} + x' * a_{12} \\ y' * a_{21} + x' * a_{22} \\ 1 \end{bmatrix}$$

이 값을 모든 x'과 y'의 값에 대하여 iteration을 통해 x_min, x_max, y_min, y_max를 최종 결정할 수 있으며, 이를 통하여 변환된 영상의 matrix인 Mat dst의 길이와 넓이를 설정할 수 있다. (translation을 고려하여 a_{13}과 a_{23}의 값 역시 합산)

이후, x'과 y'의 값에 대하여 loop를 통하여 최종 output이 되는 x와 y의 값을 결정한다.

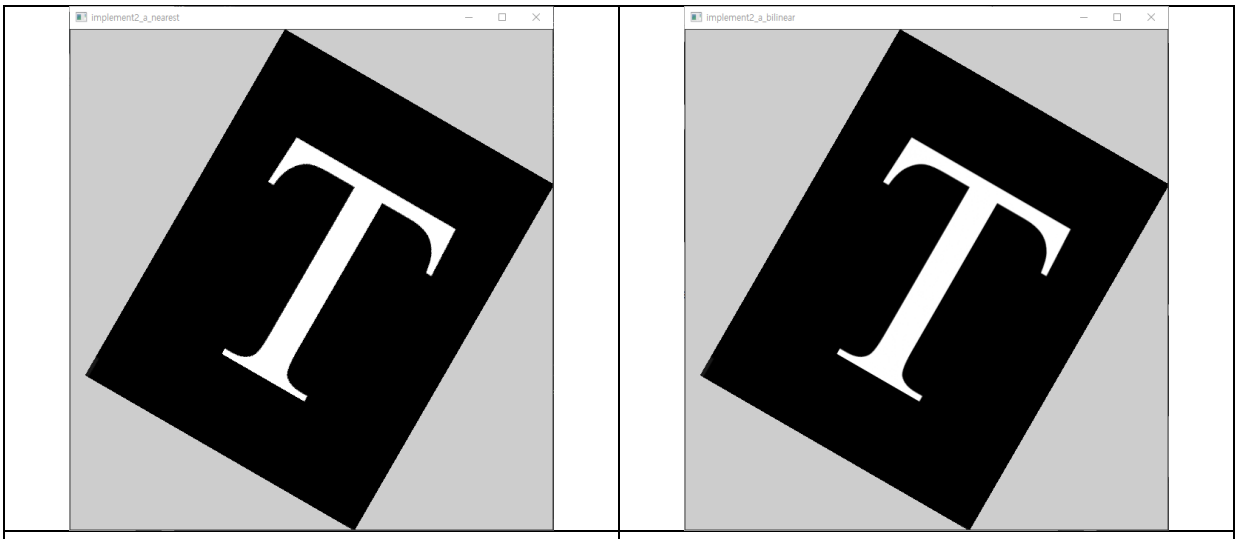
$$x = (y' + x_{min} - a_{13}) * A_{22}^{-1} + (y' + x_{min} - a_{13}) * A_{21}^{-1}$$
$$y = (y' + x_{min} - a_{13}) * A_{12}^{-1} + (y' + x_{min} - a_{13}) * A_{11}^{-1}$$

이 때 A^{-1} 값은 Mat invorg로 구현하였다. 이렇게 구현한 x, y값을 interp2D() 함수에 입력하여 최종적으로 x', y'의 값을 결정할 수 있다.

2) 구현 2

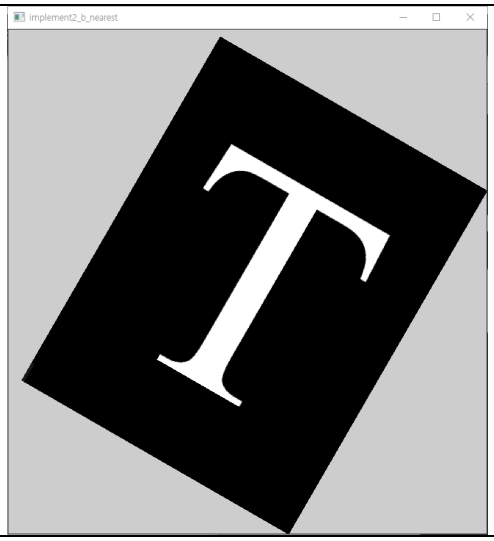
- **A** : 원점을 중심으로 +30°만큼 회전하는 transform matrix.
  B : x축 방향으로 +20pixel만큼 이동하는 transform matrix.

a) *.zip에 포함된 **"Letter_T.tif"**에 **A**를 먼저 적용한 후 **B**를 적용한 transform 한 결과를 보이시오.

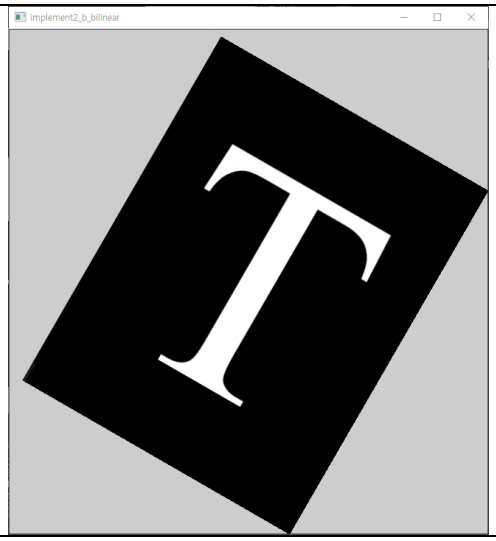


Nearest-Neighbor Interpolation	Bilinear Interpolation

b) 순서를 바꾸어 **B**를 먼저 적용한 후 **A**를 이용하여 transform한 결과를 보이시오.



Nearest-Neighbor Interpolation



Bilinear Interpolation

- a)와 b)의 결과가 **차이점**이 있는지 확인하고 차이점이 발생하는(또는 발생하지 않는) 이유를 설명하시오.

우선, a)와 같이 rotation matrix A를 먼저 적용하여 transformation을 한 뒤, translation matrix B를 적용한 결과는 다음과 같다.

$$A * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0.0 \\ \sin\theta & \cos\theta & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta * x - \sin\theta * y \\ \sin\theta * x + \cos\theta * y \\ 1 \end{bmatrix}$$

$$B * \begin{bmatrix} \cos\theta * x - \sin\theta * y \\ \sin\theta * x + \cos\theta * y \\ 1 \end{bmatrix} = \begin{bmatrix} 1.0 & 0.0 & 20.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} \cos\theta * x - \sin\theta * y \\ \sin\theta * x + \cos\theta * y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta * x - \sin\theta * y + 20 \\ \sin\theta * x + \cos\theta * y \\ 1 \end{bmatrix}$$

같은 형태로 b)의 transformation 결과는 다음과 같다.

$$A * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1.0 & 0.0 & 20.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + 20 \\ y \\ 1 \end{bmatrix}$$

$$B * \begin{bmatrix} x + 20 \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0.0 \\ \sin\theta & \cos\theta & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} x + 20 \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta * (x + 20) - \sin\theta * y \\ \sin\theta * (x + 20) + \cos\theta * y \\ 1 \end{bmatrix}$$

따라서 a)와 b)의 결과는 같지 않다.

✓ 아래 6.3)를 참고하여 `rotate()`와 `translation()`을 작성하시오.

```
Mat rotate(Mat src, float radian, string interp_type) {
        Mat rotation_matrix = Mat::zeros(Size(3, 3), CV_32F);
        float* rot = (float*) rotation_matrix.data;
```

```
        rot[0] = (float)cos(radian);      rot[1] = (float)-sin(radian);     rot[2] = 0.0;
        rot[3] = (float)sin(radian);      rot[4] = (float)cos(radian);      rot[5] = 0.0;
        rot[6] = 0.0;                     rot[7] = 0.0;                     rot[8] = 1.0;

        Mat dst_rotation = warpAffine(src, rotation_matrix, interp_type);
        return dst_rotation;
}

Mat translation(Mat src, float x, float y, string interp_type) {
        Mat translation_matrix = Mat::zeros(Size(3, 3), CV_32F);
        float* trans = (float*)translation_matrix.data;
        trans[0] = 1.0;  trans[1] = 0.0;  trans[2] = x;
        trans[3] = 0.0;  trans[4] = 1.0;  trans[5] = y;
        trans[6] = 0.0;  trans[7] = 0.0;  trans[8] = 1.0;

        Mat dst_translation = warpAffine(src, translation_matrix, interp_type);
        return dst_translation;
}
```

Rotation (about the origin)	$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x\cos\theta - y\sin\theta$ $y' = x\sin\theta + y\cos\theta$
Translation	$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$	$x' = x + t_x$ $y' = y + t_y$

강의자료의 rotation과 translation matrix를 참고하여 해당 식을 matrix의 element로 할당할 수 있다. 이 때, Mat rotate는 원본 영상과 radian 값, interpolation 방법을 input으로 받으며, Mat translation의 경우는 원본 영상과 x이동, y이동, interpolation 방법을 input으로 받게 된다. 이러한 matrix를 토대로 warpAffine()을 수행할 경우, Mat rotate의 경우 원점을 기준으로 회전하는 영상을 출력할 수 있으며, Mat translation은 해당 x, y축으로 이동한 영상을 출력할 수 있다.

4. 검토사항

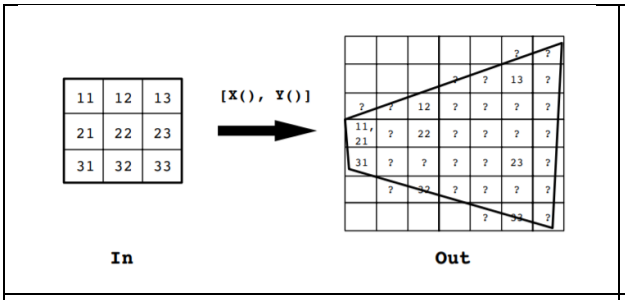
1) 구현 1과 구현 2에 대한 결과를 보고서에 첨부하시오.

2) 구현 1과 구현 2에서 **forward mapping**으로 transform을 진행했을 때 발생하는 **문제점**에 대하여 설명하시오. 해당 문제를 **inverse mapping**으로 어떻게 해결할 수 있었는 지 **interpolation**과 함께 설명하시오.

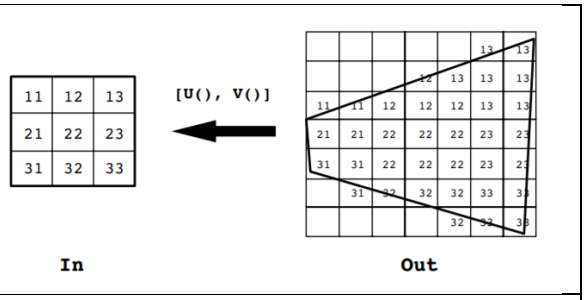
Forward mapping의 경우, 원본 image에서 pixel 좌표가 (x, y)일 때 transform된 image에서의 대응하는 pixel 좌표는 (x′, y′)이며, 이를 식으로 표현한다면 다음과 같다.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0.0 & 0.0 & 1.0 \end{bmatrix} * \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

위와 같이 forward mapping을 하는 경우 input image의 두 개 이상의 pixel이 동일하게 변환될 수 있다는 문제점이 존재한다. 또한, 일부 출력 위치에 pixel이 전혀 할당되지 않아 hole이 발생하는 문제점 역시 존재한다. Output matrix의 hole이 연속적이지 않다면 변환된 영상의 hole에서 인접한 pixel들의 값으로 interpolation 방법을 통해 hole의 intensity를 계산을 할 수 있다. 그러나 두 개 이상의 pixel이 단일 값으로 변환되거나, output matrix에 연속적이거나 불규칙하게 hole이 발생한 경우 output intensity를 예측하거나 계산하는 것이 어려우며, 변환된 영상이 왜곡될 수 있다.



Forward Mapping



Inverse Mapping

이를 inverse mapping을 통하여 효율적으로 해결할 수 있다. Inverse mapping의 경우, transform된 output image에서의 pixel 좌표인 (x′, y′)에 대응하는 원본 image의 pixel 좌표 (x, y)를 구하는 방식이다. 이를 식으로 표현한다면 다음과 같다.

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

이 때 Inverse mapping된 좌표가 원래 image에 있다면, 그 좌표의 주변 pixel로부터 intensity를 예측하기 위하여 interpolation 기법을 사용할 수 있다. Interpolation 기법은 코드 구현 단계에서 설명하였듯 x와 y좌표의 소수점에 가장 인접한 integer pixel의 intensity를 출력하는 nearest interpolation, 인접 4개의 pixel의 가중치에 따라 intensity를 계산하는 bilinear interpolation 방법 등이 존재한다.

3) Affine matrix를 homogeneous형태로 표현하면 다음과 같다.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

이러한 표현이 교과서 식(2.44)에 대하여 갖는 장점을 적으시오.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{T} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Affine Transformation (2-44)

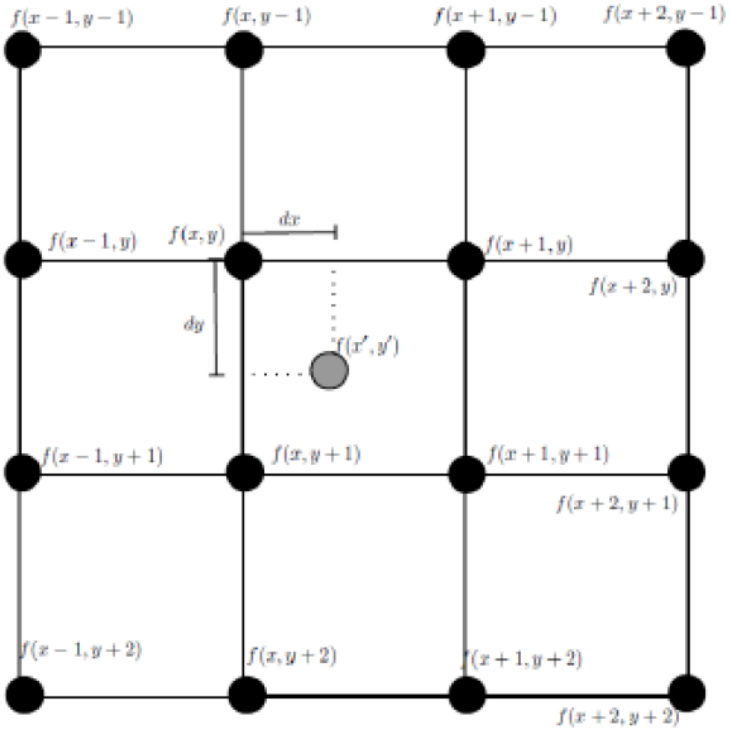
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \mathbf{A} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Homogeneous Affine Transformation (2-45)

교과서 식 (2-44)의 경우 Affine Transformation 중에서 scaling, rotation, shearing을 표현할 수 있지만, translation의 경우에는 우변에 constant 2D vector를 추가해야만 표현이 가능하다.

이러한 형태를 3x3 matrix을 사용하여 일반적인 형식으로 통합한 homogeneous한 식 (2-45)를 사용한다면 하나의 행렬을 사용하여 모든 변환을 수행할 수 있다. 예를 들어, translation의 경우에는 a_{13}과 a_{23}을 사용하여 x, y translation이 가능하다.

4) **Bicubic interpolation** 구현 방법에 대해 설명을 하시오.



Bicubic interpolation은 pixel 주변의 인접 16개의 pixel을 고려한 가중치를 통하여 intensity를 계산하는 interpolation 방법이다. 이 때, interpolated surface를 식으로 표현하면 다음과 같다.

$$v(x,y) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij} x^i y^j$$

우선, 함수값 f를 (0, 0)의 값에 배치하며, 함수값의 미분값 f_x, f_y, f_{xy}을 각각 unit square 내의 (1, 0), (0, 1), (1, 1)로 mapping한다. 이 때, 16개의 p(x, y)를 구하기 위하여 다음 수식을 적용할 수 있다.

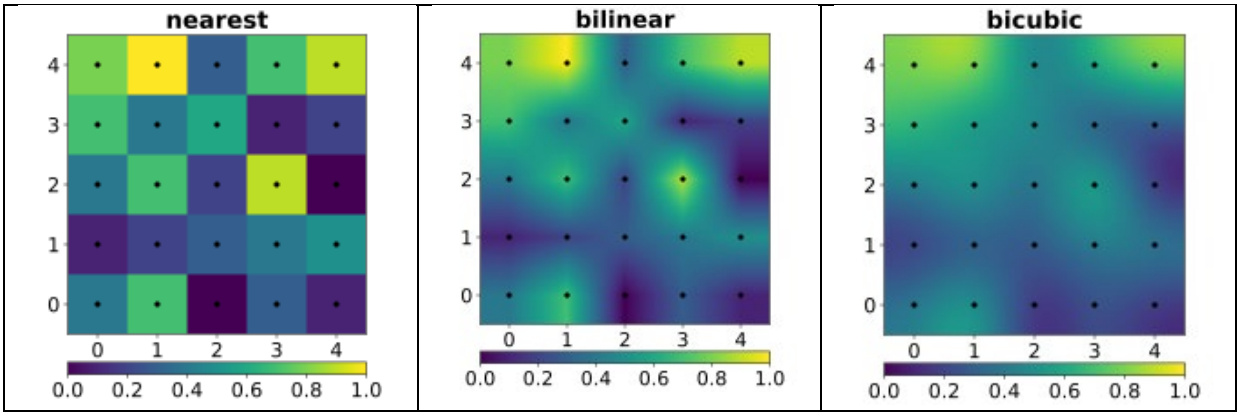
$f(0,0) = p(0,0) = a_{00}$, $f(1,0) = p(1,0) = a_{00} + a_{10} + a_{20} + a_{30}$, $f(0,1) = p(0,1) = a_{00} + a_{01} + a_{02} + a_{03}$, $f(1,1) = p(1,1) = \sum_{i=0}^{3} \sum_{j=0}^{3} a_{ij}$.	$f_x(0,0) = p_x(0,0) = a_{10}$, $f_x(1,0) = p_x(1,0) = a_{10} + 2a_{20} + 3a_{30}$, $f_x(0,1) = p_x(0,1) = a_{10} + a_{11} + a_{12} + a_{13}$, $f_x(1,1) = p_x(1,1) = \sum_{i=1}^{3} \sum_{j=0}^{3} a_{ij} i$,
Function f	Derivative f_x

$f_y(0,0) = p_y(0,0) = a_{01},$ $f_y(1,0) = p_y(1,0) = a_{01} + a_{11} + a_{21} + a_{31},$ $f_y(0,1) = p_y(0,1) = a_{01} + 2a_{02} + 3a_{03},$ $f_y(1,1) = p_y(1,1) = \sum_{i=0}^{3}\sum_{j=1}^{3} a_{ij}j.$	$f_{xy}(0,0) = p_{xy}(0,0) = a_{11},$ $f_{xy}(1,0) = p_{xy}(1,0) = a_{11} + 2a_{21} + 3a_{31},$ $f_{xy}(0,1) = p_{xy}(0,1) = a_{11} + 2a_{12} + 3a_{13},$ $f_{xy}(1,1) = p_{xy}(1,1) = \sum_{i=1}^{3}\sum_{j=1}^{3} a_{ij}ij.$
Derivative f_y	Partial Derivative f_{xy}

위의 식을 matrix form으로 정리하면 다음과 같다.

$$\begin{bmatrix} f(0,0) & f(0,1) & f_y(0,0) & f_y(0,1) \\ f(1,0) & f(1,1) & f_y(1,0) & f_y(1,1) \\ f_x(0,0) & f_x(0,1) & f_{xy}(0,0) & f_{xy}(0,1) \\ f_x(1,0) & f_x(1,1) & f_{xy}(1,0) & f_{xy}(1,1) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 3 \end{bmatrix}$$

무작위의 길이와 넓이를 가진 dataset에서의 Bicubic interpolation의 경우, 위와 같은 형식의 unit square를 연속적으로 적용시켜 bicubic surface를 만들 수 있다.



위의 figure에서 확인할 수 있듯이, pixel 값을 round하여 integer pixel의 intensity를 출력하는 nearest interpolation과 pixel 주변의 인접 4개만을 고려한 가중치로 intensity를 계산하는 bilinear interpolation과 다르게 더욱 부드럽고 detail을 보존하는 interpolation 성능을 보여준다. 그러나 kernel의 negative lobe의 영향으로, haloing, clipping, apparent sharpness 증가 등으로 이어질 수 있다.

참조문헌

DIP_Homework1.pdf, 서강대학교 MMI Lab

OpenCV Documentation (https://docs.opencv.org/), Intel Corporation

Gonzalez, Rafael C. Woods, Richard E. - Digital image processing, 4th Edition, Pearson, 2018.

Bicubic interpolation (https://en.wikipedia.org/wiki/Bicubic_interpolation), Wikimedia Foundation, Inc.