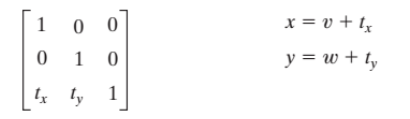
**Class Project 3**

Use two images for each operation to do the following operations and write down their advantages and disadvantages and explain your results:

1. **Image translation(lena, noise):**

**Algorithm:**



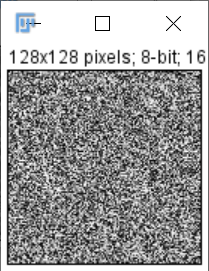
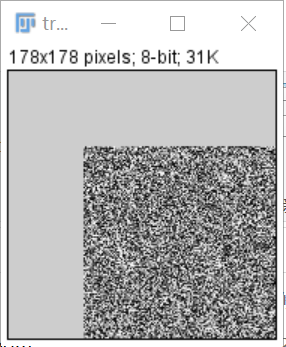
outData[(i+tx)\*width + j+ty] = inData[ i\*image->Width + j];

**Results (including pictures):**

Source: result:

Source: result:

**Discussion:**

The translation is quite simple, we just need to change the coordinate of the intensity. The key is to enlarge the width and height to adjust the new image.

**Codes:**

Image \*Image\_translation(Image \* image)

{

unsigned char \*tempin, \*tempout;

int i, j, size;

Image \*outimage;

int width = image->Width + tx;

int height = image->Height + ty;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin = image->data;

tempout = outimage->data;

for (i = 0; i < image->Width; i++)

{

for (j = 0; j < image->Height; j++)

{

tempout[(i+tx)\*width + j+ty] = tempin[ i\*image->Width + j];

}

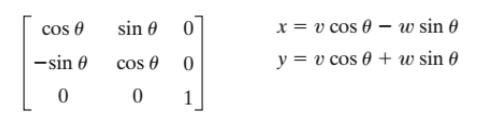
}

return(outimage);

}

1. **Image rotation (lena, noise):**

**Algorithm:**



Rotate coordinate of the new height and new width. then match with the old image. if it is in the area of the old image then make the value to the new image. And then translate the new image to the center.

a = floor((i\*cosA - (j - image->Height\*sinA)\*sinA));

b = floor((i\*sinA + (j - image->Height\*sinA) \*cosA));

If it is in the area of the old image then make the value to the new image. And then translate the new image to the center.

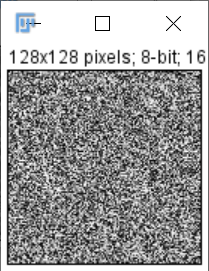
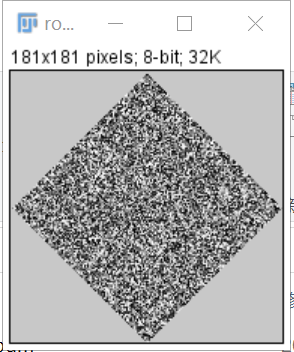
outData[i\*width + j] = inData[a\*image->Width + b];}

**Results (including pictures):**

Source: result:

Source: result:

**Discussion:**

At first enlarge the width and height make the new width and new height. Then rotate coordinate of the new height and new width. then match with the old image, if it is in the area of the old image then make the value to the new image. And then translate the new image to the center.

**Codes:**

Image \*Image\_rotation(Image \*image)

{

unsigned char \*tempin, \*tempout,\*tempout1;

int i, j;

float M = 2;

Image \*outimage;

float RotaryAngle = 45 \* 3.1415926535898 / 180;

float sinA = sin(RotaryAngle);

float cosA = cos(RotaryAngle);

int width = pow(2,0.5)\*image->Width;

int height = pow(2, 0.5)\* image->Height;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin = image->data;

tempout = outimage->data;

int p, q,a,b;

for (i = 0; i<height; i++)

{

for (j = 0; j< width; j++)

{

int a = 0, b = 0;

a = floor((i\*cosA - (j - image->Height\*sinA)\*sinA));

b = floor((i\*sinA + (j - image->Height\*sinA) \*cosA));

if (a> 0 && a <=255 && b >0 && b <=255)

{

tempout[i\*width + j] = tempin[a\*image->Width + b];

}

else

{

tempout[i\*width + j] = 200;

}

}

}

return(outimage);

}

1. **Shear operations (lena, noise):**

* vertical

**Algorithm:**



Define center of rotation

x0 = image->Width\*0.5

y0 = image->Height\*0.5

Based on the center, rotate coordinate of the new height and new width.

a = ((i - x0) +(j - y0) + width\*0.5);

b = ((j - y0) + height\*0.5);

Then match with the old image.

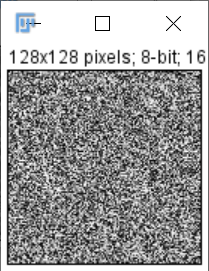
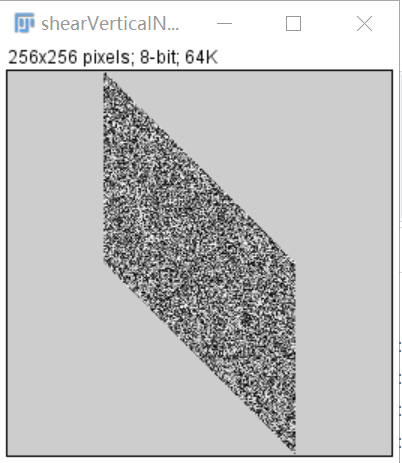
outData[a\*width + b] = inData[i\*image->Width + j];

**Results (including pictures):**

Source: result:

Source: result:

**Discussion:**

The shear operation the main is also to define the center just like the rotation and it also need to enlarge the scale of the image.

**Codes:**

Image \*Shear\_operations1(Image \*image)

{

unsigned char \*tempin, \*tempout;

int i, j, size;

Image \*outimage;

int width = image->Width\*2;

int height = image->Height\*2;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin = image->data;

tempout = outimage->data;

if (image->Type == GRAY)

size = width \* height;

else if (image->Type == COLOR)

size = width \* height \* 3;

//Define center of rotation

int x0 = image->Width\*0.5;

int y0 = image->Height\*0.5;

for (i = 0; i< image->Width; i++)

{

for (j = 0; j< image->Height; j++)

{

int a = ((i - x0) +(j - y0) + width\*0.5);

int b = ((j - y0) + height\*0.5);

tempout[a\*width + b] = tempin[i\*image->Width + j];

}

}

return(outimage);

}

* horizontal

**Algorithm:**



Define center of rotation

x0 = image->Width\*0.5

y0 = image->Height\*0.5

Based on the center, rotate coordinate of the new height and new width. The only difference between vertical and horizontal shear operation is that the coordinates are reversed.

a = ((i - x0) + width\*0.5);

b = ((j - y0) + (i - x0)\*0.5+ height\*0.5);

Then match with the old image.

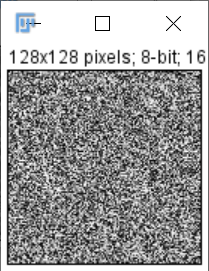
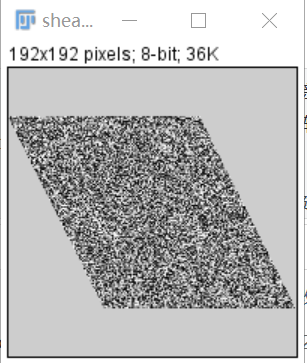
outData[a\*width + b] = inData[i\*image->Width + j];

**Results (including pictures):**

Source: result:

Source: result:

**Discussion:**

The shear operation the main is also to define the center just like the rotation and it also need to enlarge the scale of the image. The only difference between vertical and horizontal shear operation is that the coordinates are reversed.

**Codes:**

Image \*Shear\_operations2(Image \*image)

{

unsigned char \*tempin, \*tempout;

int i, j, size;

Image \*outimage;

int width = image->Width\*1.5;

int height = image->Height\*1.5;

float RotaryAngle = 45 \* 3.14 / 180;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin = image->data;

tempout = outimage->data;

if (image->Type == GRAY)

size = width \* height;

else if (image->Type == COLOR)

size = width \* height \* 3;

//Define center of rotation

int x0 = image->Width\*0.5;

int y0 = image->Height\*0.5;

for (i = 0; i< image->Width; i++)

{

for (j = 0; j< image->Height; j++)

{

int a = ((i - x0) + width\*0.5);

int b = ((j - y0) + (i - x0)\*0.5+ height\*0.5);

tempout[a\*width + b] = tempin[i\*image->Width + j];

}

}

return(outimage);

}

1. **filters (lena, noise):**

* 3×3 average

**Algorithm:**

y

x

...... ......

(0, width-1)

(1, 0)

(2, 0)

...... ......

(height-1, 0)

(0, 0)

(0, 1)

(0, 2)

(i, j-1)

(i+1, j-1)

(i-1, j-1)

(i, j)

(i+1, j)

(i-1, j)

(i, j+1)

(i+1, j+1)

(i-1, j+1)

(1.1)

(1.2)

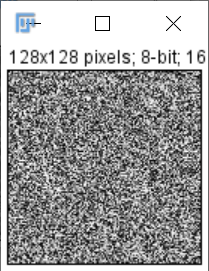
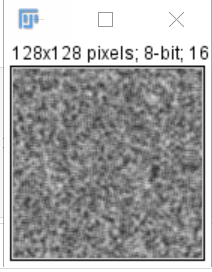
(1.3)

**Results (including pictures):**

Source: result:

Source: result:

**Discussion:**

From equation (1.1), we can conclude that differences between neighbor pixels become smaller by averaging, which leads to the smoother of image.

The result images do turn out to be smoother. For example, the edges between objects are less obvious.

**Codes:**

Image \*Average\_fliter3times3(Image \* image)

{

unsigned char \*tempin, \*tempout;

int size, i, j, k, t;

Image \*outimage;

int width = image->Width;

int height = image->Height;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin=image->data;

tempout=outimage->data;

if(image->Type==GRAY) size = image->Width \* image->Height;

else if(image->Type==COLOR) size = image->Width \* image->Height \* 3;

for(i = 1; i < height - 1; ++i){

for(j = 1; j < width - 1; ++j){

int sum = 0;

for(k = -1; k < 2; ++k)

for(t = -1; t < 2; ++t)

sum += tempin[width \* (i + k) + (j + t)];

tempout[width \* i + j] = sum/9;

}

}

return(outimage);

}

* 5×5 average

**Algorithm:**

Start from (2,2) and end with (width-3, height-3).

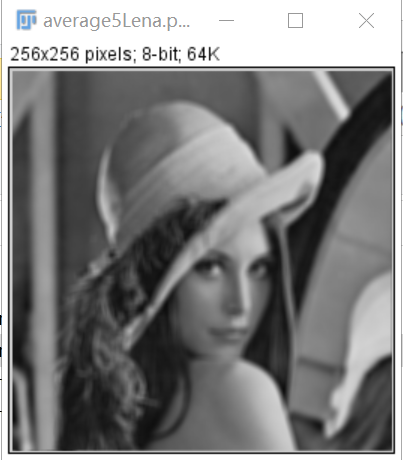
(1.1)

(1.2)

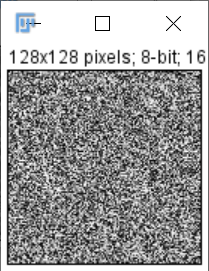
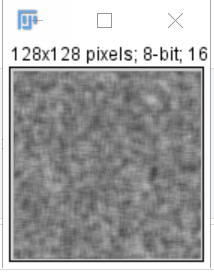
(1.3)

**Results (including pictures):**

Source: result:

Source: result:

**Discussion:**

From equation (1.1), we can conclude that differences between neighbor pixels become smaller by averaging, which leads to the smoother of image.

The result images do turn out to be smoother. For example, the edges between objects are less obvious.

The image after 5×5 average filter seems to become fuzzier than the 3×3 average filter.

**Codes:**

Image \*Average\_fliter5times5(Image \* image)

{

unsigned char \*tempin, \*tempout;

int size, i, j, k, t;

Image \*outimage;

int width = image->Width;

int height = image->Height;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin=image->data;

tempout=outimage->data;

if(image->Type==GRAY) size = image->Width \* image->Height;

else if(image->Type==COLOR) size = image->Width \* image->Height \* 3;

for(i = 2; i < height - 1; ++i){

for(j = 2; j < width - 1; ++j){

int sum = 0;

for(k = -2; k < 3; ++k)

for(t = -2; t < 3; ++t)

sum += tempin[width \* (i + k) + (j + t)];

tempout[width \* i + j] = sum/25;

}

}

return(outimage);

}

* 3×3 median

**Algorithm:**

Input: a unsigned char array that stores the source image data

Output: a unsigned char array that stores the output image data

For inImage(i, j):

Store its’ 3 x 3 neighbor including itself in a unsigned char array **local**[9];

Find out the median of **local**;

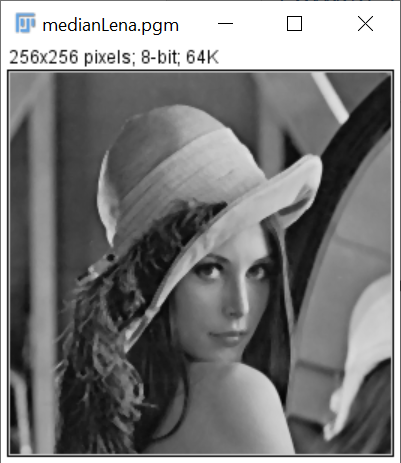
Assign the median to outImage(i,j) =>

END

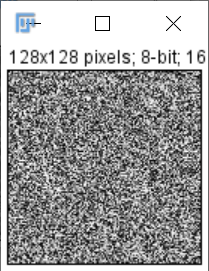
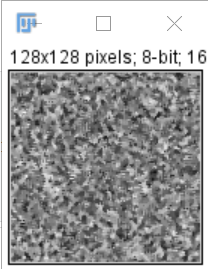
Function findMedian() use bubble sort to find the 5th smallest value of **local**.

**Results (including pictures):**

Source: result:

Source: result:

**Discussion:**

The median filter also smooths the input image. But the result of median filter is sharper than that of average filter. Median filter simply substitutes the target pixel with the median of its 3 x 3 neighbor, that is, the operation does not operate all pixels in the neighbor. So the differences between pixels in the output image by median filter is larger than that by average filter.

**Codes:**

Image \*Median\_fliter3times3(Image \*image)

{

unsigned char \*tempin, \*tempout, local[9];

int size, i, j, k, t;

Image \*outimage;

int width = image->Width;

int height = image->Height;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin=image->data;

tempout=outimage->data;

if(image->Type==GRAY) size = image->Width \* image->Height;

else if(image->Type==COLOR) size = image->Width \* image->Height \* 3;

for(i = 1; i < height - 1; ++i){

for(j = 1; j < width - 1; ++j){

int pos = 0;

for(k = -1; k < 2; ++k)

for(t = -1; t < 2; ++t)

local[pos++] = tempin[width \* (i + k) + (j + t)];

tempout[width \* i + j] = findMedian(local, 9);

}

}

return(outimage);

}

* 5×5 median

**Algorithm:**

Input: a unsigned char array that stores the source image data

Output: a unsigned char array that stores the output image data

For inImage(i, j):

Store its’ 5 x 5 neighbor including itself in a unsigned char array **local**[25];

Find out the median of **local**;

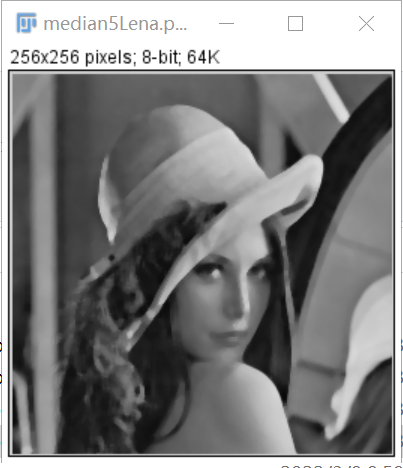
Assign the median to outImage(i,j) =>

END

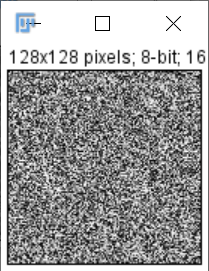
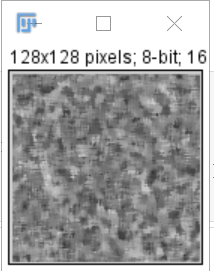
Function findMedian() use bubble sort to find the 13th smallest value of **local**.

**Results (including pictures):**

Source: result:

Source: result:

**Discussion:**

The median filter also smooths the input image. But the result of median filter is sharper than that of average filter. Median filter simply substitutes the target pixel with the median of its 5 x 5 neighbor, that is, the operation does not operate all pixels in the neighbor. So the differences between pixels in the output image by median filter is larger than that by average filter.

The image after 5×5 median filter seems to become fuzzier than the 3×3 median filter.

**Codes:**

Image \*Median\_fliter5times5(Image \*image)

{

unsigned char \*tempin, \*tempout, local[25];

int size, i, j, k, t;

Image \*outimage;

int width = image->Width;

int height = image->Height;

outimage = CreateNewImage(image, "#testing Swap", width, height);

tempin=image->data;

tempout=outimage->data;

if(image->Type==GRAY) size = image->Width \* image->Height;

else if(image->Type==COLOR) size = image->Width \* image->Height \* 3;

for(i = 2; i < height - 1; ++i){

for(j = 2; j < width - 1; ++j){

int pos = 0;

for(k = -2; k < 3; ++k)

for(t = -2; t < 3; ++t)

local[pos++] = tempin[width \* (i + k) + (j + t)];

tempout[width \* i + j] = findMedian(local, 25);

}

}

return(outimage);

}