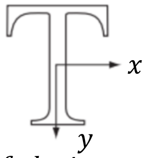
3. Image Translation, Rotation, Shear, and Smoothing

1. Image translation(lena, circles):

Algorithm:



$x_{out} = x_{in} + t_x,\ (t_x\text{: } horizontal\ offset\ of\ the\ image\ pixel)$,

$y_{out} = y_{in} + t_y,\ (t_y\text{: } vertical\ offset\ of\ the\ image\ pixel)$.

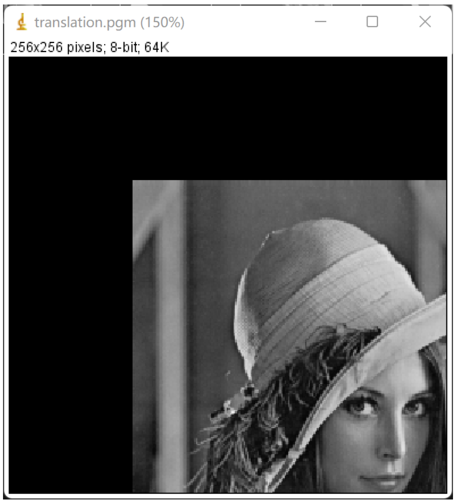
The following code will translate the image by a fixed distance to show the principle.

Results (including pictures):

Process result of “lena.pgm”:

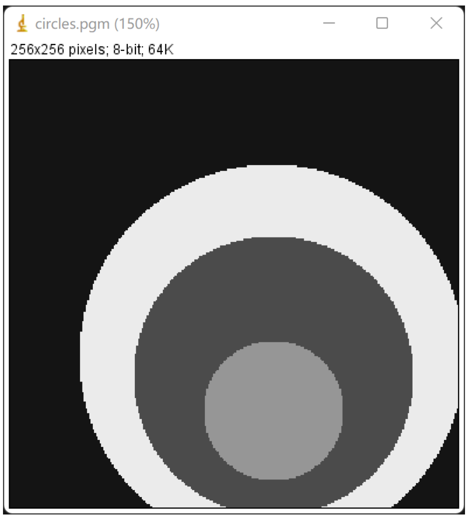
Source Image:

Result after translation:

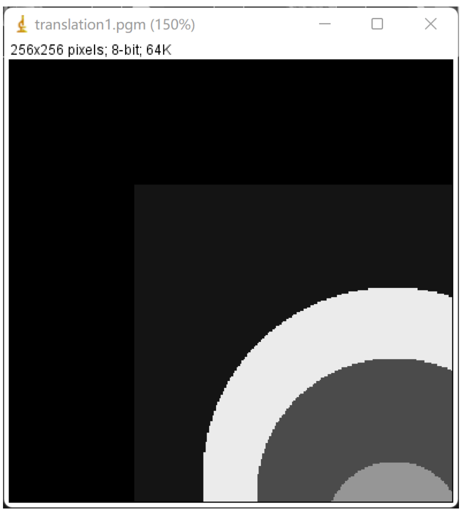


Process result of “circles.pgm”:

Source Image:



Result after translation:



Discussion:

This algorithm is to move each pixel in the image by a certain distance in the horizontal and vertical directions respectively to achieve the effect of translation. We can see that the image has been moved as a whole and the black background contains no information. This method is fast and it will not cause any pixel loss or change(if the output image is large enough).

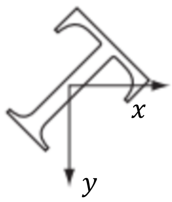
Codes:

```
// Algorithms Code:
Image *Translation(Image *image) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function", 0);
    tempin = image->data;
    tempout = outimage->data;
    // set the background of the whole image to black(0):
    for(int i = 0; i < outimage->Height; i++) {
        for(int j = 0; j < outimage->Width; j++){
            tempout[(outimage->Width)*i + j] = 0;
        }
    }

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++){
            // in case the coordinates + offsets beyond the boundary:
            if((j+72) >= outimage->Width || (i+72) >= outimage->Height) continue;
            else tempout[(outimage->Width)*(i+72) + (j+72)] = tempin[(image->Width)*i + j];
        }
    }
    return (outimage);
}
```

2. Image rotation(lena, circles):

Algorithm:



$x_{out} = round(x_{in}cos\theta + y_{in}sin\theta)$,

$y_{out} = round(x_{in}cos\theta - y_{in}sin\theta)$.

θ is the angle between the right x-axis and the clockwise rotation of the image.

For visual integrity of the output image, the code enlarges the original image size and then translates the rotated image by a distances:

$y_{out} = y_{in} + t_y$ (t_y: $vertical\ offset\ of\ the\ image\ pixel$).

And finally use the Median Filter to interpolate missing pixels in the rotated image.

Results (including pictures):

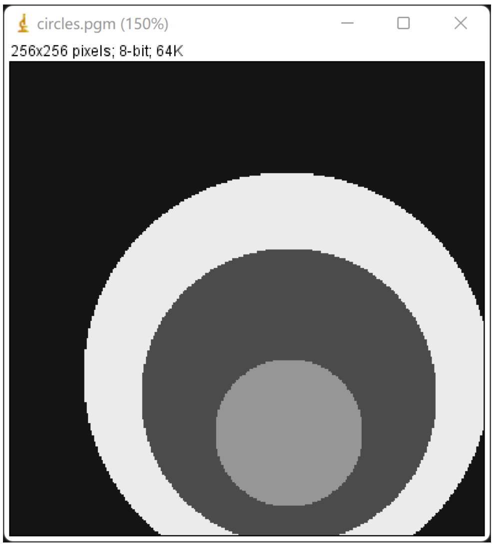
Process result of "lena.pgm":

Source Image:

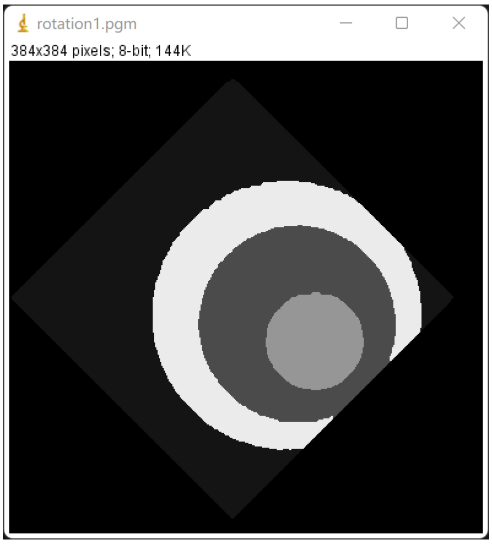
Result after rotation:

Process result of "circles.pgm":

Source Image:



Result after rotation:



Discussion:

The algorithm rotates the image 45° counterclockwise and then translates it to the center. The rotation of the image causes some of its pixels to shift and some pixels will be missing, all resulting in the rotated image being jagged, especially around the edges of objects in the image. I used median filter to interpolate the missing pixels and to smooth the image. However, the filter will also make the image a little blurry.

Codes:

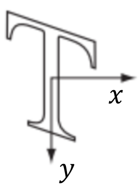
```c
Image *Rotation(Image *image) {
    unsigned char *tempin, *tempout, mask[9];
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function", 1);
    tempin = image->data;
    tempout = outimage->data;
    // set the background of the whole image to black(0):
    for(int i = 0; i < outimage->Height; i++) {
        for(int j = 0; j < outimage->Width; j++){
            tempout[(outimage->Width)*i + j] = 0;
        }
    }
    // cos(-45°) = √2/2, about 0.707.
    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++){
            int x = round(i*0.707+j*0.707);
            int y = round(i*0.707-j*0.707);
            // move the rotated image to the center:
            tempout[(outimage->Width)*(y+192) + x] = tempin[(image->Width)*(i) + (j)];
        }
    }
    // Then use 3x3 Median Filter to fill the missing pixels:
    for(int i = 0; i < outimage->Height; i++) {
        for(int j = 0; j < outimage->Width; j++){
            int num = 0;
            for(int x = -1; x <= 1; x++) {
                for(int y = -1; y <= 1; y++) {
                    mask[num++] = tempout[(outimage->Width)*(i+x) + (j+y)];
                }
            }
            // Insertion Sort:

            for(int m = 1; m < 9; m++) {
                int currNum = mask[m];
                int n = m;
                while(n >= 1 && mask[n-1] > currNum) {
                    mask[n] = mask[n-1];
                    n--;
                }
                mask[n] = currNum;
            }
            tempout[(outimage->Width)*i + j] = mask[4];
        }
    }
    return (outimage);
}
```

3. Shear operation(lena, circles):

- Vertical shear:

Algorithm:



$x_{out} = x_{in}$,

$y_{out} = y_{in} + kx_{in}$.

k is the coefficient that each pixel moves in the vertical direction according to their abscissas(x). In the following code, the value of k is set to 0.5. And the original image size is enlarged for the visual integrity.

Results (including pictures):

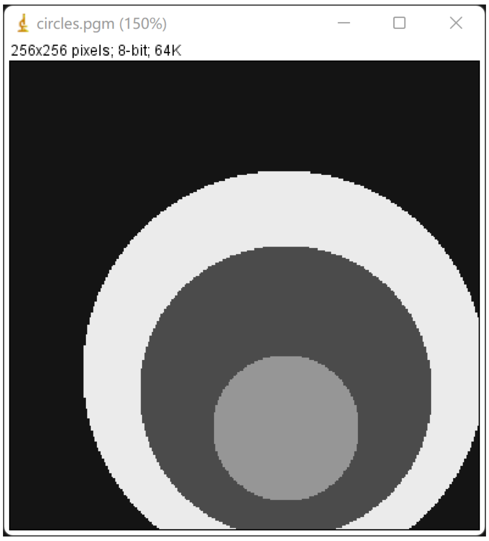
Process result of “lena.pgm”:

Source Image:

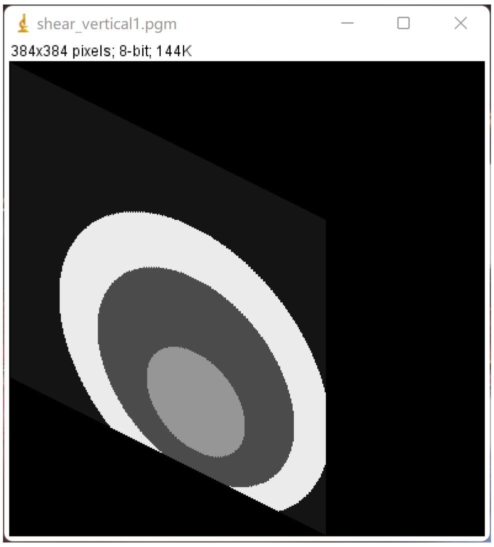
Result after vertical shear:

Process result of “circles.pgm”:

Source Image:



Result after vertical shear:



Discussion:

The algorithm keeps the abscissas (x) of the original pixels unchanged and stretches the ordinate(y) by multiplying the abscissas (x) by a certain ratio, achieving the effect of shearing into a parallelogram. And it preserves all the original pixels and does not add additional interpolation, so the image appears jagged on beveled edges.

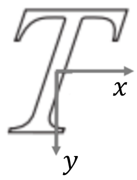
Codes:

```c
Image *Shear_vertical(Image *image) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function", 1);
    tempin = image->data;
    tempout = outimage->data;
    // set the background of the whole image to black(0):
    for(int i = 0; i < outimage->Height; i++) {
        for(int j = 0; j < outimage->Width; j++){
            tempout[(outimage->Width)*i + j] = 0;
        }
    }

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++){
            int y = round(i + (float)j*0.5);
            // in case the vertical coordinate + offset beyond the boundarys:
            if(y >= outimage->Height) continue;
            tempout[(outimage->Width)*y + j] = tempin[(image->Width)*i + j];
        }
    }
    return (outimage);
}
```

- Horizontal shear:

Algorithm:



$x_{out} = x_{in} + ky_{in},$

$y_{out} = y_{in}.$

k is the coefficient that each pixel moves in the horizontal direction according to their ordinates(y). In the following code, the value of k is set to 0.5. And the original image size is enlarged for the visual integrity.

Results (including pictures):

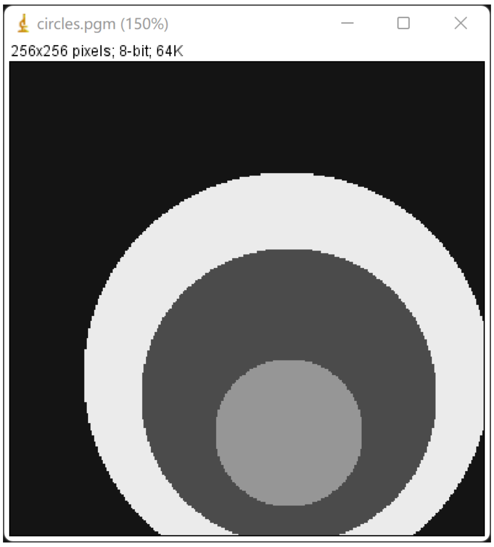
Process result of "lena.pgm":

Source Image:

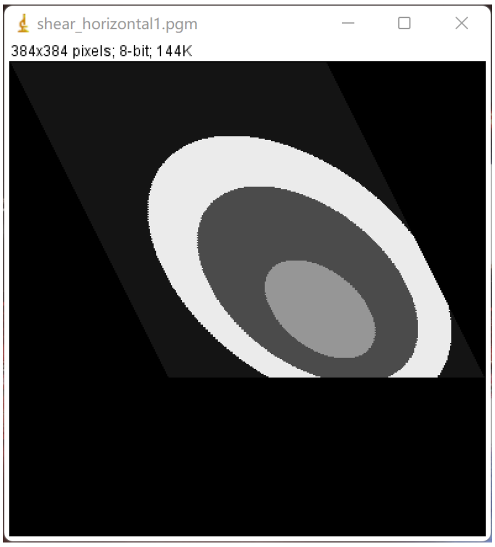
Result after horizonal shear:

Process result of “circles.pgm”:

Source Image:



Result after horizonal shear:



Discussion:

This algorithm is similar to the vertical shearing, but keeps the ordinate(y) unchanged and stretches the abscissas(x), implementing a parallelogram in the horizontal direction. And it also preserves all the original pixels and does not add additional interpolation, so the image appears jagged on beveled edges.

Codes:

```
Image *Shear_horizontal(Image *image) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function", 1);
    tempin = image->data;
    tempout = outimage->data;
    // set the background of the whole image to black(0):
    for(int i = 0; i < outimage->Height; i++) {
        for(int j = 0; j < outimage->Width; j++){
            tempout[(outimage->Width)*i + j] = 0;
        }
    }

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++){
            int x = round(j + (float)i*0.5);
            // in case the horizontal coordinate + offset beyond the current row:
            if(x >= outimage->Width) continue;
            else tempout[(outimage->Width)*i + x] = tempin[(image->Width)*i + j];
        }
    }
    return (outimage);
}
```

4. Smoothing(lena, circles):

- 3x3 average Filter

Algorithm:

(i-1, j-1)	**(i-1, j)**	**(i-1, j+1)**
(i, j-1)	**(i, j)**	**(i, j+1)**
(i+1, j-1)	**(i+1, j)**	**(i+1, j+1)**

Each pixel in the image is surrounded by **eight** pixels except those are ignored at the edges of the image. And we recalculate the value of each pixel by taking the average of the **nine** pixels in the figure. So we have the following algorithm:

$$Pixel(i,j) = data[i * Width + j], \quad i,j \geq 1, i < Height - 1, and\ j < Weight - 1.$$

$$newPixel(i,j) = \sum_{x=-1}^{1}\sum_{y=-1}^{1} originalPixel(i+x, j+y)/9 = \sum_{x=-1}^{1}\sum_{y=-1}^{1} data[(i+x) * Width + (j+y)]/9$$

Results (including pictures):

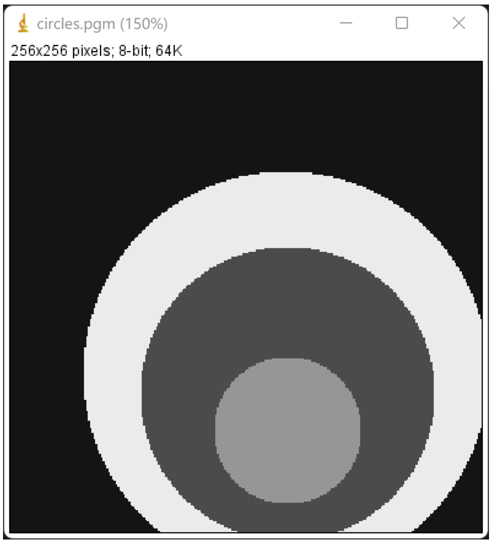
Process result of “lena.pgm”:

Source Image:

Result after 3x3 average filter:

Process result of “circles.pgm”:

Source Image:



Result after 3x3 average filter:

Discussion:

Each pixel's value is replaced by the average value of the surrounding 8 pixels and itself, so the differences between all pixels are reduced and the image will look smoother. However, it does not protect the image details well, and it also destroys the details of the image while denoising the image, so that the image becomes blurred.

Codes:

```
Image *AverageImage_3x3(Image *image) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function", 0);
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++){
            // Boundary check:
            if(i == 0 || j == 0 || i == image->Height-1 || j == image->Width-1) {
                tempout[(image->Width)*i + j] = tempin[(image->Width)*i + j];
                continue;
            }
            int sum = 0;
            for(int x = -1; x <= 1; x++) {
                for(int y = -1; y <= 1; y++) {
                    sum += tempin[(image->Width)*(i+x) + (j+y)];
                }
            }
            tempout[(image->Width)*i + j] = sum/9;
        }
    }
    return (outimage);
}
```

- 5x5 average Filter

Algorithm:

(i-2, j-2)	(i-2, j-1)	(i-2, j)	(i-2, j+1)	(i-2, j+2)
(i-1, j-2)	(i-1, j-1)	(i-1, j)	(i-1, j+1)	(i-1, j+2)
(i, j-2)	(i, j-1)	(i, j)	(i, j+1)	(i, j+2)
(i+1, j-2)	(i+1, j-1)	(i+1, j)	(i+1,j+1)	(i+1,j+2)
(i+2, j-2)	(i+2, j-1)	(i+2, j)	(i+2,j+1)	(i+2,j+2)

Each pixel in the image is surrounded by **24** pixels except those are ignored at the edges of the image. And we recalculate the value of each pixel by taking the average of the **25** pixels in the figure. So we have the following algorithm:

$$Pixel(i,j) = data[i * Width + j], \qquad i,j \geq 2, i < Height - 2, and\ j < Weight - 2.$$

$$newPixel(i,j) = \sum_{x=-2}^{2} \sum_{y=-2}^{2} originalPixel(i+x, j+y)/25 = \sum_{x=-2}^{2} \sum_{y=-2}^{2} data[(i+x) * Width + (j+y)]/25.$$

Results (including pictures):

Process result of “lena.pgm”:

Result after **3x3** average filter:

Result after **5x5** average filter:

Process result of “circles.pgm”:

Result after **3x3** average filter:

Result after **5x5** average filter:

Discussion:

Because each pixel is now averaged over the surrounding 24 pixels and itself, the difference in density between pixels across the entire image becomes smaller. Therefore, the image after the processing of the 5x5 mask is obviously more blurred and smoother than the 3x3 mask.

Codes:

```c
Image *AverageImage_5x5(Image *image) {
    unsigned char *tempin, *tempout;
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function", 0);
    tempin = image->data;
    tempout = outimage->data;
    // ignore the edge of the image:
    for(int i = 1; i < image->Height-1; i++) {
        for(int j = 1; j < image->Width-1; j++){
            int sum = 0;
            for(int x = -2; x <= 2; x++) {
                for(int y = -2; y <= 2; y++) {
                    sum += tempin[(image->Width)*(i+x) + (j+y)];
                }
            }
            tempout[(image->Width)*i + j] = sum/25;
        }
    }
    return (outimage);
}
```

- 3x3 median Filter

Algorithm:

Similar to the average filter, but the value of each pixel is replaced by the **median** of the nine-square grid pixels instead of the average. The values of the **8** surrounded pixels and itself are stored into an array and use the **Insertion Sort** method to find their median, $array[4]$, which will be assigned to $Pixel(i,j)$.

Results (including pictures):

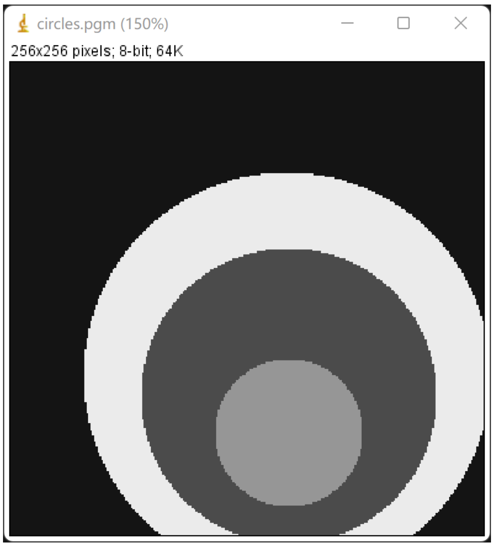
Process result of “lena.pgm”:

Source Image:

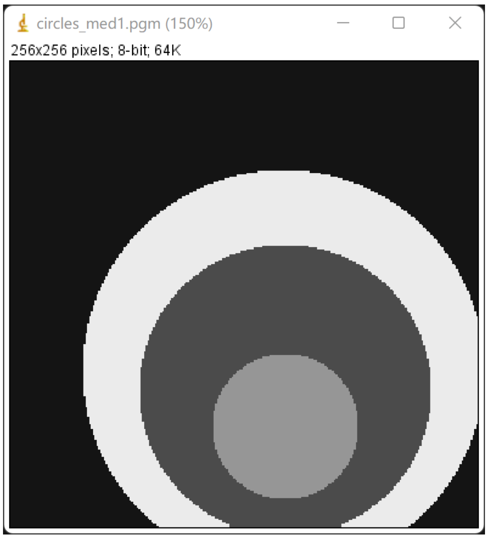
Result after 3x3 median filter:

Process result of “circles.pgm”:

Source Image:



Result after 3x3 median filter:



Discussion:

Each pixel's value is replaced by the median of the surrounding 8 pixels and itself, and the differences between all pixels are also reduced and make the image smoother. Compared to the average filter, the median filter has sharper edges of objects after noise reduction. And the difference between pixels is bigger because it only processes the target pixel each time.

Codes:

```
Image *MedianImage_3x3(Image *image) {
    unsigned char *tempin, *tempout, mask[9];
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function", 0);
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++){
            int num = 0;
            for(int x = -1; x <= 1; x++) {
                for(int y = -1; y <= 1; y++) {
                    mask[num++] = tempin[(image->Width)*(i+x) + (j+y)];
                }
            }
            // Use Insertion Sort:
            for(int m = 1; m < 9; m++) {
                int currNum = mask[m];
                int n = m;
                while(n >= 1 && mask[n-1] > currNum) {
                    mask[n] = mask[n-1];
                    n--;
                }
                mask[n] = currNum;
            }
            tempout[(image->Width)*i + j] = mask[4];
        }
    }
    return (outimage);
}
```

- 5x5 median Filter

Algorithm:

Similar to above, the value of each pixel is replaced by the median of the **25**-square grid pixels. The values of the **24** surrounded pixels and itself are stored into an array and use the **Insertion Sort** method to find their median, $array[12]$, which will be assigned to $Pixel(i,j)$.

Results (including pictures):

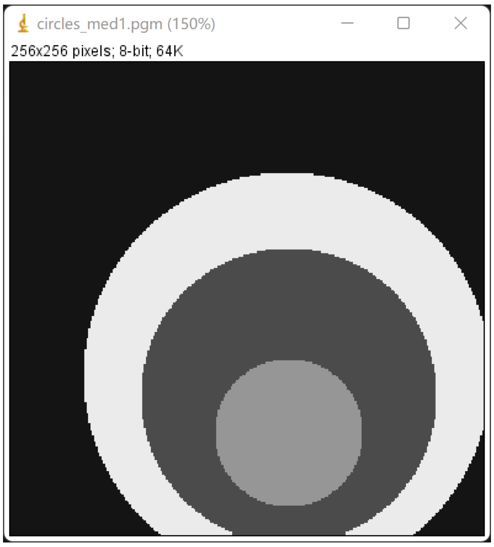
Process result of “lena.pgm”:

Result after **3x3** median filter:

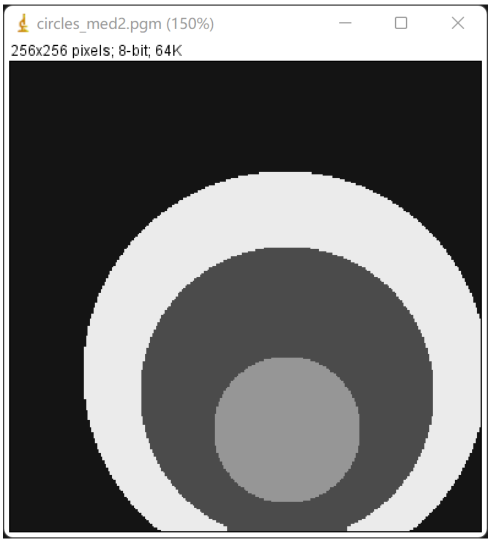
Result after **5x5** median filter:

Process result of “circles.pgm”:

Result after **3x3** median filter:



Result after **5x5** median filter:



Discussion:

Because each pixel now takes the median of the surrounding 24 pixels and itself, the difference in density between pixels in the entire image becomes smaller. Therefore, the image after the processing of the 5x5 mask is obviously more blurred and smoother than the 3x3 mask. But it is still sharper than the output of the 5x5 average filter.

Codes:

```
Image *MedianImage_5x5(Image *image) {
    unsigned char *tempin, *tempout, mask[25];
    Image *outimage;
    outimage = CreateNewImage(image, (char*)"#testing function", 0);
    tempin = image->data;
    tempout = outimage->data;

    for(int i = 0; i < image->Height; i++) {
        for(int j = 0; j < image->Width; j++){
            int num = 0;
            for(int x = -2; x <= 2; x++) {
                for(int y = -2; y <= 2; y++) {
                    mask[num++] = tempin[(image->Width)*(i+x) + (j+y)];
                }
            }
            // Use Insertion Sort:
            for(int m = 1; m < 25; m++) {
                int currNum = mask[m];
                int n = m;
                while(n >= 1 && mask[n-1] > currNum) {
                    mask[n] = mask[n-1];
                    n--;
                }
                mask[n] = currNum;
            }
            tempout[(image->Width)*i + j] = mask[12];
        }
    }
    return (outimage);
}
```