P 1

1.1 M

One common technology in image processing operation is geometrical manipulation. By modifying the positions of the pixels and keeping the colors unchanged, we can create special visual effects. The operation contains translation, scaling, rotation and so on. This method now is also widely used in computer graphics. In this part of project, I realized swirl effect by basic geometric modification and implemented 3D imaging geometry to 3D object image capture.

1.2 A

1.2.1.

In in image processing, we usually manipulate the pixel in Cartesian coordinates. So we need to translate the image coordinates to Cartesian coordinates at first. Suppose (u, v) denotes the Cartesian coordinates and (p, q) denotes the image coordinates. The relationship between the two coordinates is $u = q - \frac{1}{2}$ and $v = P + \frac{1}{2} - p$ $(P\ is\ the\ image\ height)$. After translating the input image to Cartesian coordinates, based on the desired effect, we can find the transformation matrix (address mapping) form the input image to output image in Cartesian coordinates. Then translate the output image form Cartesian coordinates to image coordinates. One thing in geometrical manipulation is when we find the transformation matrix, we need to do **.** If we do the forward mapping, the corresponding pixel address in output image may be fractional, so we can't decide which pixel it indicates. Using reverse address mapping, when we get the fractional address in input image, we can use bilinear interpolation to find the pixel value for the corresponding output image pixel.

1.2.2

R is one of the steps to realize swirl effect. With the origin as the pivot, when image rotates θ anticlockwise, the corresponding coordinates (2D) are expressed as: $x_o = \cos\ \theta\ x_i - \sin\ \theta\ y_i$, $y_o = \sin\ \theta\ x_i + \cos\ \theta\ y_i$. Details are discussed in (1.3 a-1).

1.2.3

S is another step to realize swirl effect. The swirl effect is caused by rotating different parts (different pixels) of the image by different angles. We need to translate the image to polar coordinates and then find the radial coordinate for different pixels, because the rotation angle is related to the radial coordinate. Once we get the rotation angles, we can rotate the pixels of the image by corresponding angles. Details are discussed in (1.3 a-1).

1.2.4

T is also used in swirl effect. Since we want the center of image to be the pivot (at the origin), we can translate the center of the image to the origin and then apply rotation (1.2.2) to the image. Translation of coordinates can be expressed as: $x_o = x_i + t_x$ and $y_o = y_i + t_y$.

1.2.5

To capture 3D object scene, we need to use to transform the object in world coordinates to camera coordinates and then to 2D image coordinates. These two transformations are realized by and respectively.

The extrinsic camera matrix transforms the world coordinates to the camera coordinates, it can be expressed as:

$$[R \quad |t] = \begin{pmatrix} X_c^X & X_c^Y & X_c^X & -\vec{r}\overrightarrow{X_C} \\ Y_c^X & Y_c^Y & Y_c^X & -\vec{r}\overrightarrow{Y_C} \\ Z_c^X & Z_c^Y & Z_c^X & -\vec{r}\overrightarrow{Z_C} \end{pmatrix}$$

where $\overrightarrow{X_C} = (X_c^X, X_c^Y, X_c^Z)^T, \overrightarrow{Y_C} = (Y_c^X, Y_c^Y, Y_c^Z)^T, \overrightarrow{Z_C} = (Z_c^X, Z_c^Y, Z_c^Z)^T$, $\vec{r}$ is the distance from the lens of camera to the origin.

The intrinsic camera matrix transforms the camera coordinates to the 2D image coordinates, it can be expressed as:

$$K = \begin{pmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{pmatrix}$$

where f is the focal length and c_x and c_y are translation of the image form image center to its origin, so $c_x = \frac{width}{2}$ and $c_y = \frac{height}{2}$.

So far, we can map the 3D coordinates form the world coordinates to the 2D image coordinates by

$$\begin{pmatrix} x \cdot Z_C \\ y \cdot Z_C \\ Z_C \end{pmatrix} = K \cdot [R \quad t] \cdot (X\ Y\ Z\ 1)^T$$

Details about this problem are discussed in (1.3 b-2).

1.2.6

Before capture the 3D object image, we need to construct the the 3D object first. There are six faces for a cube and we only care about the five faces, since in reality we can't see the bottom face, we can ignore it. The next thing we need to do is to put the five images on the surface of the cube. Equivalently, we need to assign the 3D coordinates to each pixel of the 5 images. Details are discussed in (1.3 b-1).

1.2.7

The methods are implemented by C++.

1.3 R & D

a-1

According to 1.2.1, I do reverse address mapping form the output image to input image. Firstly, I translate the center of image to the origin by using $t_x = t_y = \frac{(size-1)}{2}$ (1.2.4). Secondly, I calculate the the polar coordinate and the radial coordinate is $r = \sqrt{x^2 + y^2}$. Then the rotation angle should be related to r. By trying different relations and compare the different results with illusion of swirl effect about Lena in Fig.1, I finally get the same images as the illusion. The rotation angle is $\theta = \frac{\pi * r}{size}$. After getting the angle, to apply reverse address mapping, the rotation of the pixel coordinates is expressed as: $x_i = \cos\ \theta\ x_o + \sin\ \theta\ y_o$ and $y_i = -\sin\ \theta\ x_o + \cos\ \theta\ y_o$.

(a)original Lena (b)Lena with swirl effect

Figure.1 I use my code to get the same image effect as the illusion provided by the project

Since we find the rotation angles, we can obtain every output image pixel address in the input image. Then using bilinear interpolation to determine the pixel value for the corresponding output image. The bilinear interpolation method used here is:

$F(p', q') = (1-a)(1-b)F(p,q) + b(1-a)F(p,q+1) + a(1-b)F(p+1,q) + abF(p+1,q+1)$

where $a = x_i - integer(x_i), b = y - integer(y_i), p = integer(x_i), q = integer(y_i)$.

The final output image for Kate is shown in Fig.2:

(a)original Kate (b)Kate with swirl effect

Figure.2 Swirl special effect on Kate image

b-1 **P -**

The image size of the five images is 200x200 and the object (cube) size is 2x2x2, so the unit length of each pixel in the 3D coordinates (world coordinates) is 0.01. Then we can translate the image coordinate to the Cartesian coordinate. Since the the images are on the surface of the cube, it means one of the 3D coordinates is fixed. In this problem, the *baby* image has a fixed z coordinate (z=2), *baby cat* has a fixed x coordinate (x=2), *baby dog* has a fixed y coordinate (y=2), *baby panda* has a fixed x coordinate (x=0) and *baby bear* has fixed y coordinate (y=0). For each fixed coordinate, we can find the rest two coordinates corresponding to each pixel in each input image as mentioned in (1.2.1).

The input is five 2D images and the output is the 3D coordinates of points (pixels of five images) in 3D world coordinate with corresponding pixel value.

b-2 **F**

In this problem, set the center of the cube (3D object) as the origin. The camera can only observe 3 faces of the object (*baby*, *baby cat, baby dog*), so we only map these 3 image points in the world coordinate. The camera lens is centered at (5,5,5), denoted $\vec{r} = (5,5,5)$, so the viewing direction is (1,1,1). The unit vectors of the camera coordinates are $\overrightarrow{X_c} = (-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0)^T, \overrightarrow{Y_c} = (\frac{1}{\sqrt{6}}, \frac{1}{\sqrt{6}}, -\frac{2}{\sqrt{6}})^T, \overrightarrow{Z_c} = (-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}})^T$. According to (1.2.5), we can find the corresponding coordinates in camera coordinate for each points in world coordinate by the forward mapping: $x_C = -\frac{1}{\sqrt{2}}X + \frac{1}{\sqrt{2}}Y, y_C = \frac{1}{\sqrt{6}}X + \frac{1}{\sqrt{6}}Y - \frac{2}{\sqrt{6}}Z, z_C = -\frac{1}{\sqrt{3}}X - \frac{1}{\sqrt{3}}Y - \frac{1}{\sqrt{3}}Z + \frac{15}{\sqrt{3}}$.

The focal length in this problem is $\sqrt{3}$. After we get the 3D object coordinates in the camera coordinate, we can find the coordinates in the 2D image plane (1.2.5): $x = \sqrt{3} \cdot \frac{x_C}{z_C} + c_x$, $y = \sqrt{3} \cdot \frac{y_C}{z_C} + c_y$.

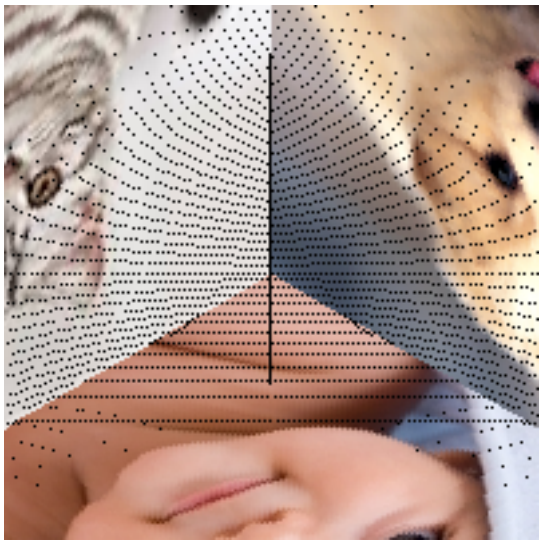
In the 2D image plane, we have the coordinates for each input pixel. One parameter we need to decide is the for the output image. It determines how large view we can see about the scene of 3D object captured by the camera and also the quality and artifacts in the output image. So I tried different density to find the optimal density, the different densities and corresponding images are shown in Fig.3.

Once the density is determined, we need to multiply the Cartesian coordinates x and y with the density. The results be fractional. To handles this problem, I the fractional address values.

(a)density=200

(b)density=280

(c)density=400



(b)density=600

Figure.3 projected image using different pixel density

From Fig.3 we can see that if the density is small, the projected image is too small and if the density is large, the projected image is so big that we can't see the entire scene. Also, if the density is very large, some pixel points in the image plane will overlap and result in some artifacts (the black dots).

By comparing different pixel density, the optimal density I pick is **280**, i.e. 280 of pixels per unit length.

There are in the resulting image. Since I use method to handle the fractional address, some pixel overlap in the same location and this leads to some artifacts. Also, in preprocessing, the edge of each image overlap, after forward mapping form the world coordinate to image plane, some information about the edge is lost. So we can see there are some distortions on the boundaries in Fig.3. To the result, I want to assign equal weights for the overlapping pixels on the boundaries. Since the overlapping pixels have the same location in the 2D image coordinates, to preserve the information and reduce distortion, we can combine the information of the overlapping pixels.

b-3 **B**

The challenges in the reverse mapping is that the **$(Z_C,\ \omega)$** is not given for each points in the 2D image plane transformed to 3D camera coordinate points. To handle this problem, I first infer the relation between the 2D points in image plane and 3D point in the world coordinates. In (1.2.5), we have:

$$Z_c \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = K * [R \quad |t] * \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

Suppose $\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = Q, \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = P, Z_c Q = K * [R \quad |t] * P = K[R * P + \quad t] = K * R[P - \vec{r}], where\ \vec{r} = (5,5,5).$

Therefore, $Z_c Q = K * R * P - K * R * \vec{r}$, $\boldsymbol{Z_c[K * R]^{-1}Q + \vec{r} = P}$. So far we find the relation between the 2D points in the image plane and 3D points in the world coordinate, K, R, $\vec{r}$ and Q, P are known to us, the only unknown is Z_c, the depth. Finding the depth is the main challenge in this problem, and it's not easy. But in this problem, the object (cube) and the camera position are fixed and known to us. Given this information, we can use the known variables to express Z_c.

Since we know **#1, #2** **#3** and we know the cube's points coordinates. We can easily find the normal vectors for these faces, i.e. $\overrightarrow{n_1} = (0,0,1), \overrightarrow{n_2} = (1,0,0), \overrightarrow{n_3} = (0,1,0)$ for face #1, #2 and #3 respectively. According to the property of normal vector, $\vec{n}$*P (eliminate the 4th component) +d= 0, so for example, $\overrightarrow{n_1}$*P(the 3D points in face #1) = Z = 1, so $d_1 = -1$. It can be shown that $d_1 = d_2 = d_3 = -1$.

So far we have $\vec{n}, d$ and $\vec{n}$*P+d = 0 → $\vec{n}$*(P- $\vec{r}$)+d+$\vec{n}$*$\vec{r}$ =0. Substitute P- $\vec{r}$ with $Z_c[K * R]^{-1}Q$:

$$\vec{n}*(Z_c[K * R]^{-1}Q)+d+\vec{n}*\vec{r} = 0$$

By transforming, $Z_c = \frac{(-d-\vec{n}*\vec{r})}{\vec{n}*([K*R]^{-1}Q)}$. Substitute Z_c into $Z_c[K * R]^{-1}Q + \vec{r} = P$. Then we can find the corresponding 3D points in world coordinate for face #1, #2 and #3.

To calculate the matrix computation, I use MATLAB for reverse mapping. The density I chose is 280 from b-2. Since we know which faces we can see, I calculate the reverse mapping for each face individually. And then integrate them together.

The result is shown in Fig.4.

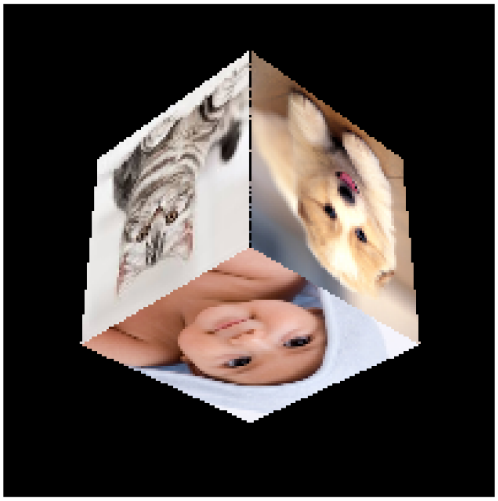


Figure.4 projected image using different pixel density

P 2

2.1 M

Half-toning can be used in print industry to reduce visual reproductions to an image that is printed with fewer inks, usually white ink (color of the paper) and black ink (color of the ink). When viewed from a distance, the dots blur together, creating the illusion of continuous lines and shapes. Using the density of dots can control the darkness of a region in the image. In this part of project, I convert input image to half-toned image by four methods: dithering, error diffusion, scalar color half-toning and vector half-toning.

2.2 A

2.2.1

Before applying half-toning, we need to normalize the image pixel value to be between 0 to 1, i.e. transform the image pixel value form emission based (display) to absorption based (printing). 255 (white) and 0 (black) in display are mapped to 0(white) and 1(black) respectively in printing.

2.2.2

D : the idea for half-toning is that we need to decide each pixel value whether to be 0 (black) and 255 (white) in emission base, this is called binary quantization. A traditional way is to set the threshold to be 128 but the resulting image is flat, with no variation. So a better way is to add noise (blue noise) to the input image and then do binary quantization. However, adding noise is not easy, so uses multiple quantization thresholds instead of adding noise, i.e. we can set different thresholds for different pixels in a region to do binary quantization. The quantization pattern used is called Bayer filter array. The formula is:

$$I_{2n}(i,j) = \begin{bmatrix} 4 \cdot I_n(x,y) + 1 & 4 \cdot I_n(x,y) + 2 \\ 4 \cdot I_n(x,y) + 3 & 4 \cdot I_n(x,y) \end{bmatrix}, \text{ where } I_2(i,j) = \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix}$$

Sliding the Bayer index matrix on the input image, we can determine the threshold for each pixel by the following formula: $T(x,y) = \frac{I(x,y)+0.5}{N^2}$, where N is the size of the Bayer index matrix. Using the thresholds, we can do binary quantization for each pixel. In absorption based view, if the pixel value

is over $T(x, y)$, it is black, i.e. set the pixel value 0 in emission based. Otherwise, set the pixel value 255 in emission based to represent white. Details are discussed in (2.3 a-1).

2.2.3

E is an elegant idea to achieve the blue noise effect. The idea is to compute the error between quantized output image with the ground truth normalized input image. Once we have the error for each pixel, we can use different kinds of diffusion matrix to diffuse the error in one pixel to its neighboring pixels. Usually, the error diffusion is from left to right. By adding the diffusion error to each pixel, we can compare this new pixel value with the threshold ($T = 0.5$) to do binary quantization. Details are discussed in (2.3 b-1).

2.2.4

CMYK : In this problem, we use CMY channels instead of RGB channels. In CMYK space, there are 8 colors: W(0,0,0), K(1,1,1), Y(0,0,1), B(1,1,0), R(1,0,1), C(0,1,0), G(0,1,1) and M(1,0,0). For each RGB value, we can find the corresponding CMY value and (W, K), (Y, B), (R, C), and (G, M) are complementary color pairs.

S **-** : This method is similar as error diffusion discussed above (2.2.3). I first transform the RGB to CMY for each RGB channel and then apply Floyd-Steinberg error diffusion matrix to each CMY channel separately. Finally, based on the CMY value I obtain the resulting half-toned color image.

2.2.5

V **-** **:** CMY color space can be partitioned into 6 Minimum Brightness Variation Quadruples (MBVQ) and at first, we set the quantization error for all pixels to be 0. For each pixel, based on the CMY value, we can find which MBVQ it belongs to and then find the vertex (CMY value) of this MBVQ closest to the CMY value plus the quantization error. After finding this vertex, the quantization error for this pixel will be the difference of CMY value plus the quantization error and the vertex CMY value. Then we can use error diffusion to distribute the quantization error to neighboring pixels. The order for this diffusion is in serpentine order [1]. Details are discussed in (2.3 d-1).

2.3 R **& D**

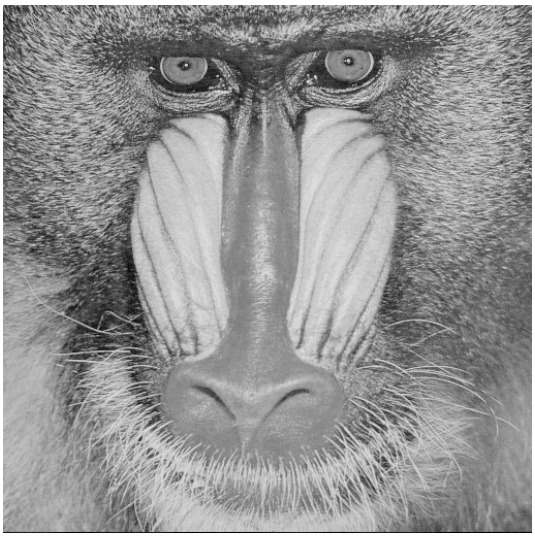
a-1

According to the formula in (2.2.2), the Bayer index matrices are as following

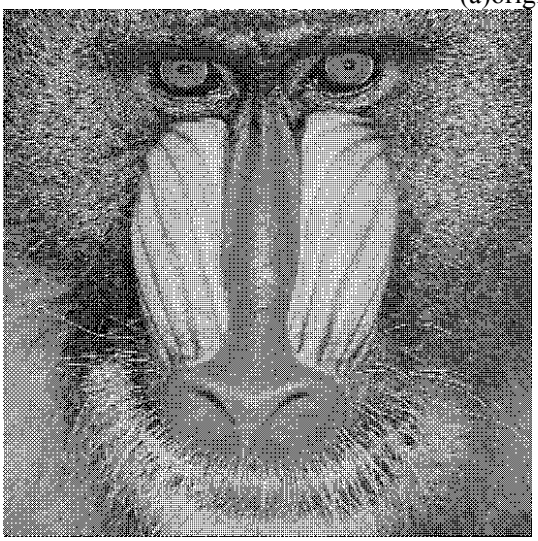
$$I_4(i,j) = \begin{pmatrix} 5 & 9 & 6 & 10 \\ 13 & 1 & 14 & 2 \\ 7 & 11 & 4 & 8 \\ 15 & 3 & 12 & 0 \end{pmatrix} \qquad I_8(i,j) = \begin{pmatrix} 21 & 37 & 25 & 41 & 22 & 38 & 26 & 42 \\ 53 & 5 & 57 & 9 & 54 & 6 & 58 & 10 \\ 29 & 45 & 17 & 33 & 30 & 46 & 18 & 34 \\ 61 & 13 & 49 & 1 & 62 & 14 & 50 & 2 \\ 23 & 39 & 27 & 43 & 20 & 36 & 24 & 40 \\ 55 & 7 & 59 & 11 & 52 & 4 & 56 & 8 \\ 31 & 47 & 19 & 35 & 28 & 44 & 16 & 32 \\ 63 & 15 & 51 & 3 & 60 & 12 & 48 & 0 \end{pmatrix}$$

Sliding the Bayer index matrix without overlapping to get the thresholds ($T(x, y) = \frac{I(x,y)+0.5}{N^2}$) for each image pixel (has been normalized in printing base) and then do quantization based on the thresholds: if the pixel value is over T, we set the pixel value 0 (black) in display base, otherwise, we set the pixel value 255 (white) in display base.

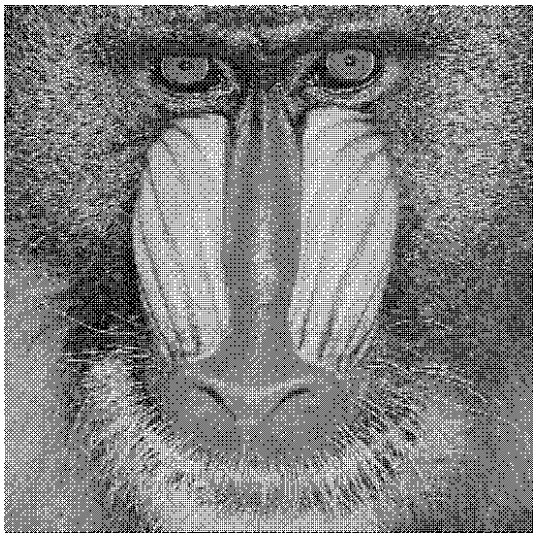
The resulting images are shown in Fig.5.



(a)original mandrill



(b)I_4 $Bayer\ index\ matrix$



(c)I_8 $Bayer\ index\ matrix$

Figure.5 half-toned images by dithering

After dithering, in Fig.5, we can see there are some black or white - in the image. Since Bayer index matrix we use has smaller value in a row or column, so it leads to all pixels in a row or column larger than the threshold and set to the black. Then we will see some black lines in the image. And the same reason results in the white lines.

a-2

F -

The idea to create four gray-level dithering is that we can threshold each pixel twice. In this problem, I use the combination of $I_4(i,j)$ and $I_8(i,j)$ in (2.3 a-1) to complete the two times threshold. For each pixel, I first use I_4 to get a threshold to decide whether the pixel should be put into black region (0 or 85) or white region (170 or 255). Then in each one region, I use I_8 to get a threshold to decide which value should this pixel picks. Using this order is because I_4 gives less thresholds compared with I_8, I_4 gives the general half-tone sets and I_8 gives precise threshold in different half-tone set.

To implement this algorithm, I first filter the image with I_4. By comparing each pixel with the threshold $T = \frac{I(x,y)+0.5}{4^2}$, if it is larger than T, I assign indicator_1(x, y)= 1 and indicator_1(x, y)= 0 otherwise. Again, I filter the image with I_8 and compare each pixel with the threshold $T = \frac{I(x,y)+0.5}{8^2}$, if it is larger than T, I assign

indicator_2(x, y) = 1 and indicator_2(x, y) = 0. Then based on indicator_1 and indicator_2, we can determine the pixel value:

$$indicator\_1\begin{cases} = 1\begin{cases} indicator\_2 = 1 \rightarrow pixel\ value = 0 \\ indicator\_2 = 0 \rightarrow pixel\ value = 85 \end{cases} \\ = 0\begin{cases} indicator\_2 = 1 \rightarrow pixel\ value = 170 \\ indicator\_2 = 0 \rightarrow pixel\ value = 255 \end{cases} \end{cases}$$

The final result is shown in Fig.6.

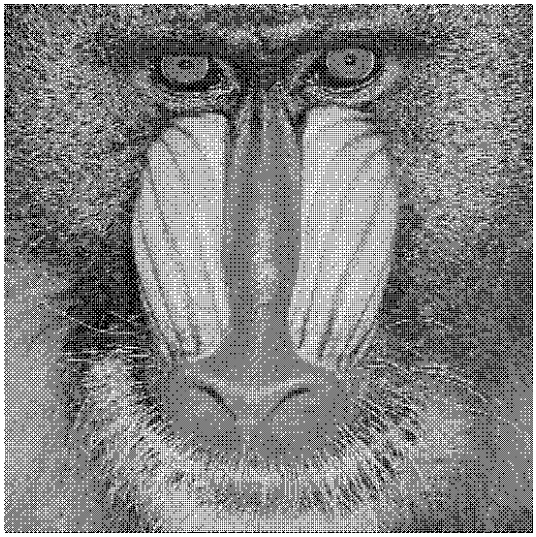


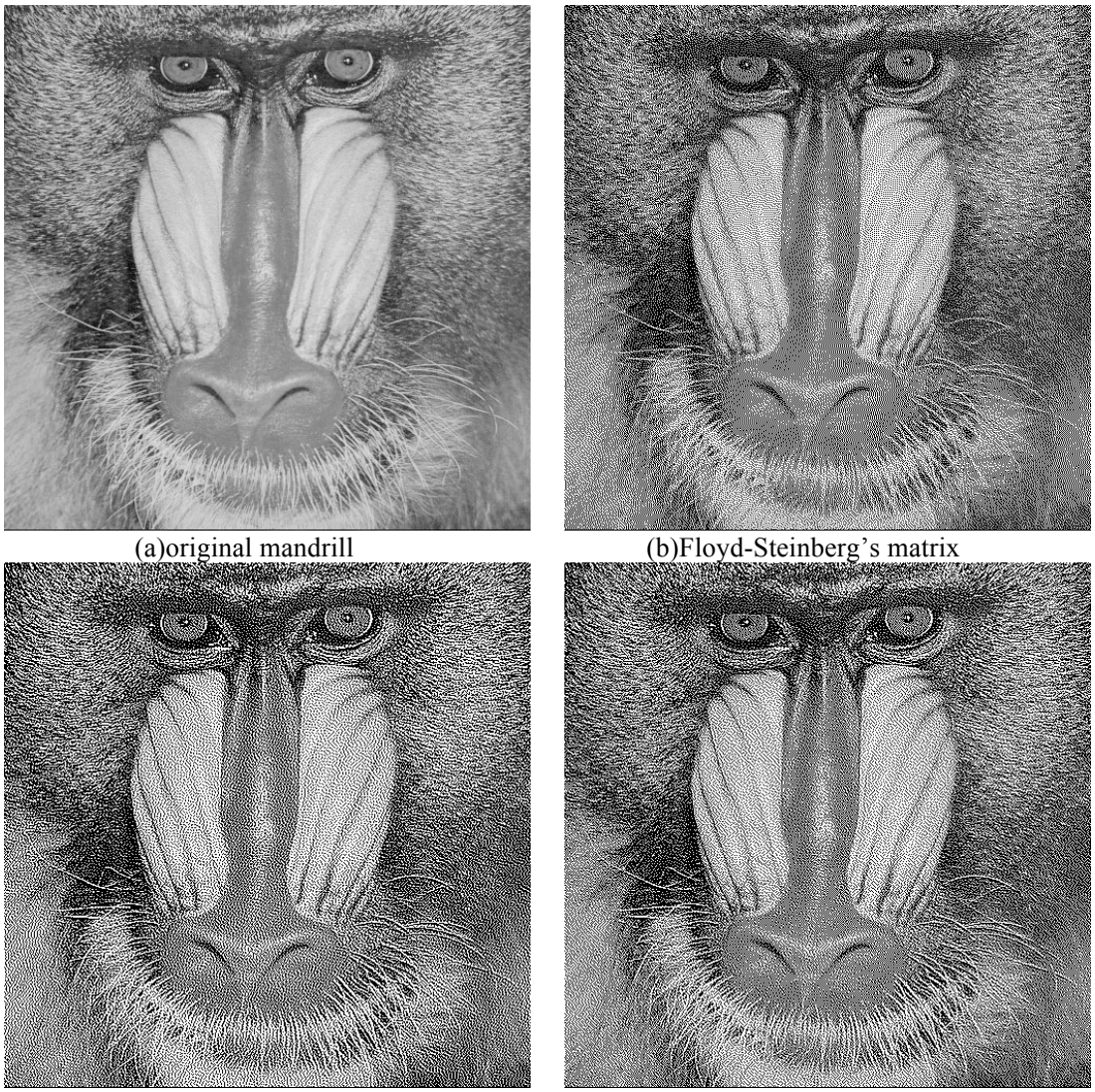
Figure.6 half-toned images by four gray-level dithering

Compared Fig.6 with Fig.5, we can see there is , i.e. the - in the image viewing from a distance. The transition of the gray level is more natural. Since we use four gray-level to quantize the image, some pixels set to 0 or 255 are set to 85 or 179, it can reduce the “line-like” distortions appeared in the image, making the image better.

b-1

For the input image, I scan the pixels in raster order. For the starting pixel, I use the threshold (T=0.5) to quantize the pixel value and compute the error for this pixel. Then use three kinds of error diffusion matrices to distribute the error to its neighboring (future) pixels respectively. For the next current pixel, I use the pixel value plus error for quantization (T=0.5), compute the error for this current pixel and then do error diffusion again. Iterate this process until all the pixels are quantized.

The half-toned images using Floyd-Steinberg's error diffusion matrix, error diffusion matrix proposed by Jarvis, Judice, and Ninke (JJN) and error diffusion matrix proposed by Stucki are shown in Fig.7.



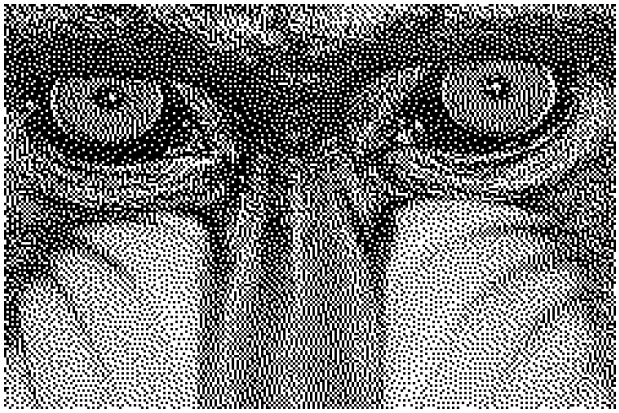
(a)original mandrill (b)Floyd-Steinberg's matrix

(c)JJN matrix (d)Stucki matrix

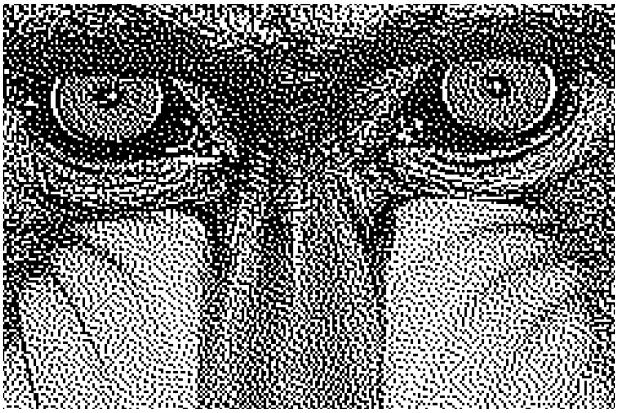
Figure.7 half-toned images by error diffusion

In Fig.8, compared with (a), (b) and (c) shows fewer structure artifacts. In (b), there are some worm-like artifacts, but in (b) and (c), these artifacts become disappeared or are highly reduced. Since JJN and Stucki matrix have which can distribute the quantization residual to more neighboring pixels which has not been processed and the error can be propagated in . By doing this, it enhances the edges more in an image. I also calculate the MSE between the half-toned images and the original images. Although the MSE is not a very useful metric, since the results are quite similar. As shown in the table below, JJN matrix gives a better result than Stucki matrix. (The pixel values have been normalized to 0 to 1 to calculate the MSE)

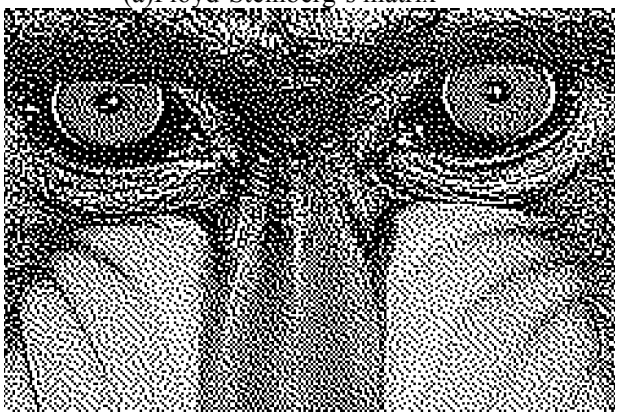
	Floyd-Steinberg's	JJN	Stucki
MSE	0.2128	0.9172	0.2000



(a)Floyd-Steinberg's matrix



(b)JJN matrix



(c)Stucki's matrix

Figure.8 detailed areas for half-toned images by error diffusion

b-2

M is that since Floyd-Steinberg can create some warm-like artifacts, we can apply the matrix in . Also, I want to change the

$$\frac{1}{16}\begin{pmatrix}0 & 0 & 0\\ 0 & 0 & 7\\ 3 & 5 & 1\end{pmatrix} \rightarrow \begin{pmatrix}0 & 0 & 0\\ 0 & x & f_1(x)\\ f_2(x) & f_3(x) & 0\end{pmatrix}$$

It means that the coefficients vary according to the input value (current pixel) and is neighboring pixels. In my method, $f_i = \frac{|x-x_i|}{\sum_{i=1}^{3}|x-x_i|}$, where i=1,2,3. Since I want the bigger the difference between the current pixel and its neighboring pixel, the more error is distributed to that neighboring pixel.

By doing this, we can based on the , since a slighter out-of-position pixel cause less negative visual effect. So it will

.

c-1

The method is the same as (2.3 b-1) and apply this method for CMY channels respectively.

The resulting half-toned color images are shown in Fig.9.

(a)original sailboat image (b)half-toned sailboat image

Figure.9 half-toned color images by scalar color half-toning

From Fig.9 we can see there are lots of in the half-toned image. Compared with the original color image, the difference about the luminance (brightness) between the original and the half-toned is significant. The half-toned is a little and **(** **)** than the original.

The of this approach is that it applies error diffusion for each the channel . By doing this, we can compensate the loss or gain (error) and therefore improve the quality for each channel, but when (since we focus on the final CMY value, the joint value), the error for the pixel can be very large. It results an imbalanced CYM value compared with the original image, so there will be lots of artifacts.

d-1

Before doing the vector color half-toning, I initialize the quantization error at all pixel to zero. I first scan the pixel of the input image in serpentine order. For each scanned pixel, I calculate which MBVQ it belongs to by calculating the R, G, B value as the pyramid MBVQ described in [1]:

$$R+G\begin{cases} >255\begin{cases} G+B>255\begin{cases} R+B+G>510 \rightarrow CMYW \\ R+B+G<=510 \rightarrow MYGC \end{cases} \\ G+B<=255 \rightarrow RGMY \end{cases} \\ <=255\begin{cases} G+B<=255\begin{cases} R+B+G<=255 \rightarrow KRGB \\ R+B+G>255 \rightarrow RGBM \end{cases} \\ G+B>255 \rightarrow CMGB \end{cases} \end{cases}$$

Once we find which MBVQ the pixel belongs to, I find the vertex of the MBVQ (one of the four vertexes) closest to the pixel's CMY value plus its error. Then subtract this vertex value from this pixel's CMY value plus its error and distribute the difference to its neighboring pixels by Floyd-Steinberg's matrix. Upgrade the quantization error at all pixels and then iterate the process discussed above to the next pixel in in serpentine order.

The resulting half-toned color image by vector color half-toning is shown in Fig.10.

half-toned sailboat image

Figure.10 half-toned color images by vector color half-toning

Compare Fig.10 with (b) in Fig.9, there is and the color of the is better. Also the brightness is more natural. We can see vector color half-toning improves the half-tone result. As discussed in (c-1), the shortcoming of the scalar color half-toning is that it applies error diffusion for each the channel , since we want to do error diffusion for three channels at the same time (because three channels account for the real color). Therefore, the idea of vector color half-toning is to - which the original color tends to render, i.e. we can have a smaller but more accurate half-tone set. In this way, we use at the same time and choosing a smaller range half-tone set before applying error diffusion can reduce the error between the quantization color and the real color a lot. And when we apply the error diffusion, we can further reduce the luminance (brightness) efficiently. Therefore, the half-tone noise is reduced.

P 3

3.1 M

Binary image usually contains lots of imperfections and pixels with redundant information. To remove these, we can account for the form and structure of the image. The form and structure of the image mean the ordering of the pixel value (0 or 1). By studying this feature, we can apply a series of nonlinear operations to the binary image to realize the morphological processing. In this part of project, we will apply shrinking, thinning and skeletonizing to the binary images.

3.2 A

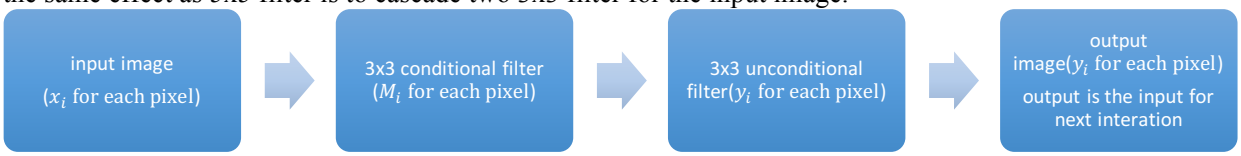
3.2.1

H - - : The idea is to use a set of predefined filters to scan each pixel in the input image with its neighboring pixels and check whether matched or not. The filters used are usually 3x3 binary patterns. For the matching pixel, we can take action to change the pixel value according to the specific problem. For the mismatching pixel, we just need to copy the original pixel value for the output image.

3.2.2

S , T S : The ideas for these three methods are similar except using different conditional and unconditional filters. By using the hit-or-miss filters discussed in (3.2.1), we can

erase the black pixels for the object. But the 3x3 filters can lead to over-erasure which will destroy the connectivity of the objects in the image since the 3x3 filters don't provide enough information. 5x5 filters can do better but the number of pattern is huge, i.e. 2^{25}. So the way to use 3x3 filter to create the same effect as 5x5 filter is to cascade two 3x3 filter for the input image.



When the input image goes through the 3x3 conditional filter, it examines each pixel whether to erase or not. If the the pixel "hits" the 3x3 conditional filter, we denote M=1 for this pixel, meaning its pixel value be erased (from 1 to 0). Otherwise, we denote M=0 for this pixel, meaning its pixel value remains. After finding all M's for pixels of input image, we form a new image (same size with the input) with M's, called *tmp*. As stated above, the pixel with M=1 may be erased, so to decide whether we will erase it, we need to use 3x3 unconditional filter to scan the pixel wit M=1 of the image *tmp*. If the pixel "hits" the 3x3 unconditional filter, we decide this pixel can't be erased (remain the pixel value), so y=1. Otherwise, the pixel should be erased, y=0. After this step, we can obtain the output image according to the information about whether erase the value or not for each pixel of the input image. To get the final result, we need to apply this process several times to the image several times until the output doesn't change anymore.

$$x_i \begin{cases} x_i = 0 \rightarrow pixel\ value\ remained \\ x_i = 1 \begin{cases} miss\ filter \rightarrow pixel\ value\ remained \\ hit\ filter \rightarrow M = 1 \begin{cases} hit\ filter \rightarrow pixel\ value\ remained \\ miss\ filter \rightarrow pixel\ value\ changed \end{cases} \end{cases} \end{cases}$$

3.2.3

C **:** to apply the filters easily and efficiently, I represent the filters in 8-bit stream form. Suppose the 3x3 filter is like:

$$\begin{pmatrix} M_3 & M_2 & M_1 \\ M_4 & M & M_0 \\ M_5 & M_6 & M_7 \end{pmatrix}$$

I represent the filter as $M_0M_1M_2M_3M_4M_5M_6M_7$. According to the pattern table provided, I use MATLAB to obtain all the 8-bit stream as following.

C for :

1000000,10000,100,1,10000000,100000,1000,10,11000000,1100000,110000,11000,1100,110,11,10000001,11000001,1110000,11100,111,10110000,10100001,1101000,11000010,11100000,111000,1110,10000011,10110001,1101100,11110000,11100001,1111000,111100,11110,1111,10000111,11000011,11110001,1111100,11111,11000111,11100011,11111000,111110,10001111,11110011,11100111,11111100,11111001,111110,11111,10011111,11001111,11110111,11111101,111111,11011111

C for :

10100000,101000,1010,10000010,11000001,1110000,11100,111,10110000,10100001,1101000,11000010,11100000,111000,1110,10000011,10110001,1101100,11110000,11100001,1111000,111100,11110,1111,10000111,11000011,11110001,1111100,11111,11000111,11100011,11111000,111110,10001111,11110011,11100111,11111100,11111001,111110,11111,10011111,11001111,11110111,11111101,111111,11011111

C for **:**

10100000,101000,1010,10000010,11000001,1110000,11100,111,11110000,11100001,1111000,111100,11110,1111,10000111,11000011,11110001,1111100,11111,11000111,11100011,11111000,111110,10001111,11110

011,11100111,11111100,11111001,111110,11111,10011111,11001111,11110111,11111101,111111,1101111
1,11111011,11111110,10111111,11101111

U :

1000000,10000,10,10000000,11000000,1100000,110000,11000,1100,110,11,10000001,1101000,10110000,10
100001,11000010,1100100,11000100,11100100,11001,110001,111001,1001100,1000110,1001110,10011,100
10001,10010011,111000,11111111,10101000,10111100,11101001,10001010,11001011,10011110,101010,11
11010,101111,10100010,10100111,11110010,10100100,10110101,101001,1101101,1001010,1011011,10010
010,11010110,1010001,1010010,1010011,1010100,1010101,1010110,1010111,11111001,11111010,1111101
1,11111100,11111101,11111110,11111111,1010100,10010100,11010100,10101,1010101,10010101,1101010
1,1111110,10111110,11111110,111111,1111111,10111111,11111111,10101,100101,110101,1000101,101010
1,1100101,1110101,10011111,10101111,10111111,11001111,11011111,11101111,11111111,1000101,10010
01,1001101,1010001,1010101,1011001,1011101,11100111,11101011,11101111,11110011,11110111,111110
11,11111111

U :

1,100,1000000,10000,10,10000000,1000,100000,10100000,101000,10000010,1010,111000,11111111,111111
11,10000011,10101000,11111111,101010,11111111,10001010,11111111,10100010,11111111,1010001,1010
010,1010011,1010100,1010101,1010110,1010111,11111001,11111010,11111011,11111100,11111101,11111
110,11111111,1010100,10010100,11010100,10101,1010101,10010101,11010101,1111110,10111110,111111
10,111111,1111111,10111111,11111111,10101,100101,110101,1000101,1010101,1100101,1110101,1001111
1,10101111,10111111,11001111,11011111,11101111,11111111,1000101,1001001,1001101,1010001,101010
1,1011001,1011101,11100111,11101011,11101111,11110011,11110111,11111011,11111111,10100100,1011
0101,101001,1101101,1001010,1011011,10010010,11010110

3.2.4

O : to count the number of objects, we need to decide how to classify an object. In this problem, an object is the pixels connected together. Here I use the 8-connectivity (the weak condition) as the principle to classify an object, we can label each point (black pixel) depending on the connectivity. The method I use: first of all, I scan the pixel . Depending on the 8-connectivity, I use this filter as my mask:

$\begin{pmatrix} X_3 & X_2 & X_1 \\ X_4 & X & X_0 \\ X_5 & X_6 & X_7 \end{pmatrix}$, where $X_0 \cup X_1 \cup X_2 \cup X_3 \cup X_4 \cup X_5 \cup X_6 \cup X_7 = 1$. When the first pixel hits the mask during the scan, I label this pixel as #1 in a new image (called labelling image) with the same size as the original image. At beginning, initial all pixels 0 in the labelling image. Then scan the next pixel. If the pixel hits the mask, then we need to look at the corresponding pixel in the labelling image. If its neighborhood has non-zero label, then assign this pixel the same label number. Otherwise, assign the label with 1 plus the latest label number. After labelling all the pixels in the image, we need to look at the label numbers in the label image and find the largest one, this will be the number of object. (3.3 a-2)

3.2.5

C : closing is an important operator from the field of mathematical morphology. It can be derived from the fundamental operations of erosion and dilation. The structuring element (H) I use is 3x3 matrix:

$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$. The closing can be a dilation followed by an erosion using this structuring element (since the reflection of H is the same as H). I can be expressed as G = $(F \oplus H) \ominus H$. For $\oplus$ and $\ominus$, I set the center of the H as the original position. $\oplus$ means moving the images to 8 directions then take the union and $\ominus$ means moving the image to 8 directions then take the joint. (3.3 a-2) and (3.3 b-1)

3.3 R & D

a-1

To count the nails (white circles) in the image, I first convert (original image) the white pixel to black pixel and vice versa. So the task is to count how many black circles in the image. Before counting, we need to apply shrinking to the image first. I implemented the method discussed in (3.2.2) and filters discussed in (3.2.3) for several times, the results are shown in Fig.11.

(a)converting the binary value

(b)shrinking 2 times

(c)shrinking 4 times

(d)shrinking 6 times

(e)shrinking 7 times

(f)shrinking 8 times

Figure.11 shrinking for several times to count nails

As shown in Fig.11, after shrinking the image 7 times, the image remained unchanged. So we can stop shrinking.

Due to the , we just want to count the number of nails and don't want to count the horse-shoe. So I don't use the the method discussed in (3.2.4) to count all the objects in the image, because we know it will lead to inaccurate number in advance. Instead, due to the prior knowledge, we know the circles will be shrink into dots and the horse-shoe will not, so I use another method to count the black dots only. I design a 5x5 filter to examine the pixels of the output image.

Counting filter = $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$, if hit, then count 1. And the is **11**.

In reality, the number of nails is 8, but the result is 11. From (e) in Fig.11, we find that two bottom nails on each side aren't shrink into one black dot but a line because in original image, (a), we can see these nails are close to the edges of the horse shoe. Due to preserving the connectivity, they are shrink into lines, it is shown in Fig.12, the right bottom part. The dots should not be erased based on (3.2.2). Also, the shrinking of the inner edge of the horse shoe results in 5 black dots, it is shown in Fig.12, the left top part. The dots should be eliminated based on (3.2.2). So the resulting number is comprised of 6 true nail dots and 5 false dots. One way to handle this problem is that if using a larger counting filter instead, for example, 7x7 or larger, the 5 false dots will not be counted. So the number will be **6.** The error dues to the 2 bottom nails only.

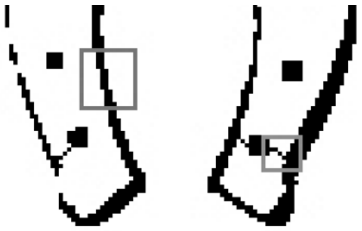
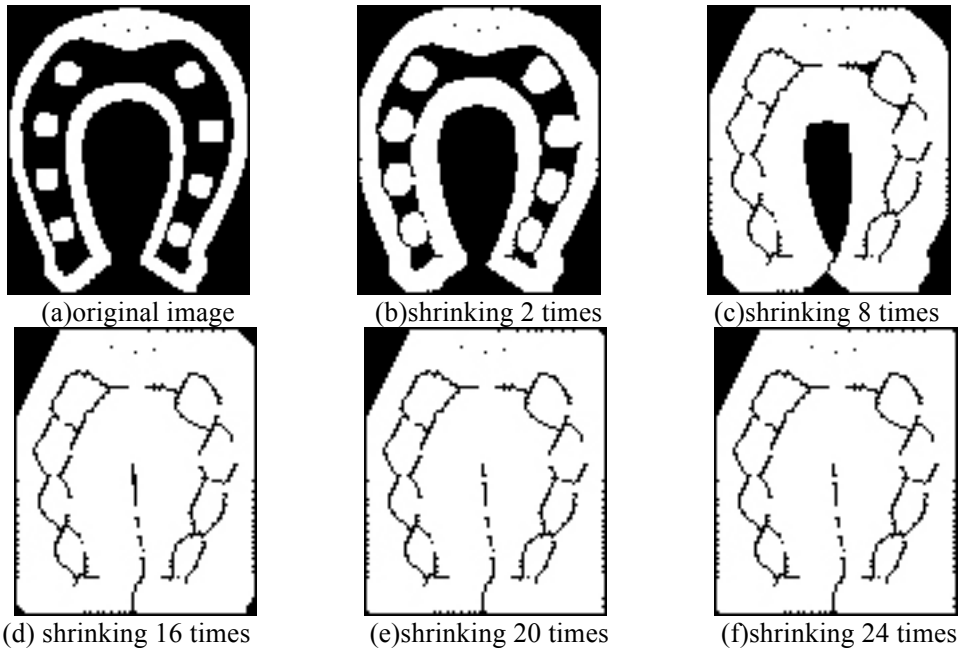


Figure.11 error parts of the image

To count the holes (black dots), I use the original image as the input image directly. The process is the same as counting the nails. The result is shown in Fig.13.



(a)original image (b)shrinking 2 times (c)shrinking 8 times

(d) shrinking 16 times (e)shrinking 20 times (f)shrinking 24 times

Figure.13 shrinking for several times to count holes

As shown in Fig.11, after shrinking 24 times, the image remains unchanged, so we can stop shrinking. To count the holes (black dots), the idea is the same as counting the nails so I use the same counting filter. The resulting (black dots) is **3.** In reality, the correct number is 3, which is consistent with the result. From (f) in Fig.13, we can see that the 3 black dots in (a) are shrink into 3 black dots and other black areas are shrink into lines, so we can get the correct number.

a-2

Before counting, we need to apply hole filling to the holes (white circles). The method I use is discussed in (3.2.5). I also did another preprocessing. I eliminate the 3 black dots by applying subtractive filter. Find the black pixels surrounded by white pixels and then change the value from 1 to 0.

Pattern filter $=\begin{pmatrix} X & 0 & X \\ 0 & 1 & 0 \\ X & 0 & X \end{pmatrix}$, $where\ X\ is\ o\ or\ 1$. If hits the filter, change the value from 1 to 0. The output is shown in (a) in Fig.13. The hole filling result is shown in (b) in Fig.14.

Then I convert the hole filling result image as what I did in (a-1) before shrinking, i.e. convert the black pixel to the white and vice verse. Depending on the shrinking times obtained from (a-1), I use 10 times to shrink the image. The final output is shown in (c) in Fig.13.


(a)filling the black dots


(b)hole filling


(c)shrinking the image 10 times

Figure.14 shrinking for several times to count holes

To count the number of objects, I use the method discussed in (3.2.4). The result is **28**. The result is disappointing because in reality we know the correct number should be 1. The reason is that the connectivity of the horse-shoe is destroyed. The reason for this situation is the same as we discussed in (3.3 a-1). We can see the top part of the image shown in Fig.15 in details.


(a)shrinking 7 times

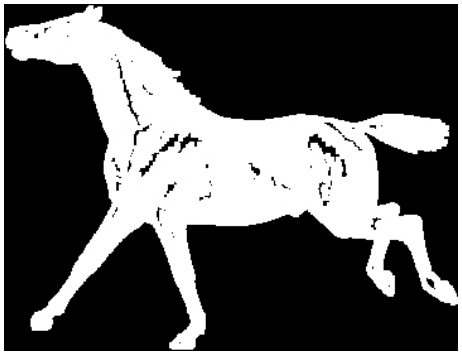

(b)shrinking 8 times

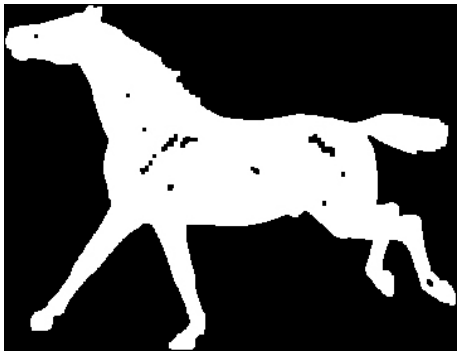
Figure.15 details of the shrinking image

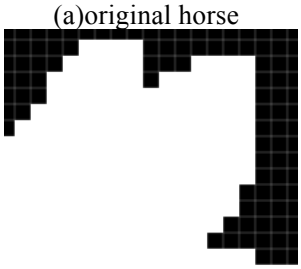
In (a), the connectivity is preserved after shrinking 7 times. But when shrinking 8 times, the connected dots should be erased due to (3.2.2). So the final result has some discontinued points and these dots are considered as a new object resulting in the inaccurate counting number.

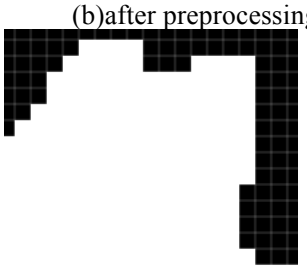
b-1

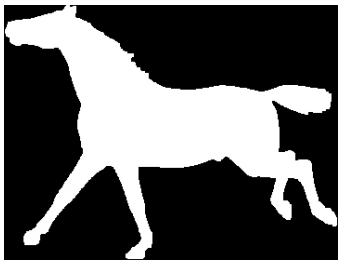
The pre-processing uses closing method discussed in (3.2.4). The resulting image after preprocessing is shown as follows:


(a)original horse


(b)after preprocessing


(c)contour of ears in original


(d)contour of ears after preprocessing

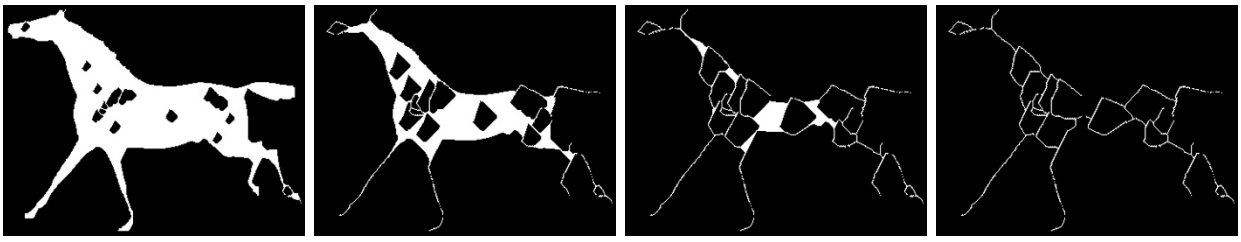


(e)improved image after preprocessing

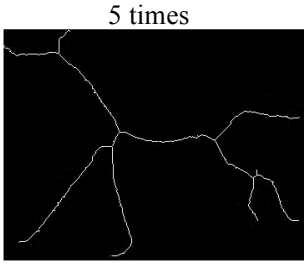
Figure.16 image after preprocessing

As shown in (a) and (b) in Fig.16, many holes are filled. But **(3 3) I** , $\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$, there still remains some black dots. Compared (d) with (c), we can see that the contour of the horse is smoothen after preprocessing. The this is that we can apply the subtractive filter to fill the single pixel hole and use other types of structure element to do closing, the improved hole-filling image is shown in (e) in Fig.16.

The thinning and skeletonizing results for image with preprocessing are shown in (a) and (b) in Fig.17.

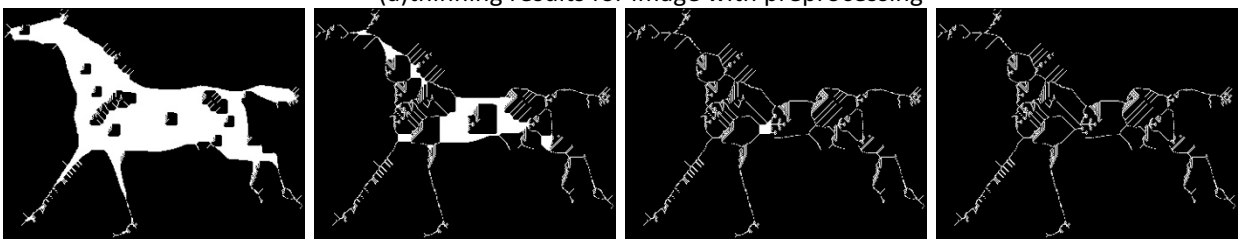


5 times 15 times 35 times 45 times

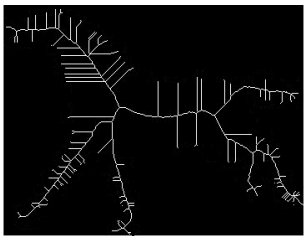


improved image after 45 times thinning

(a)thinning results for image with preprocessing

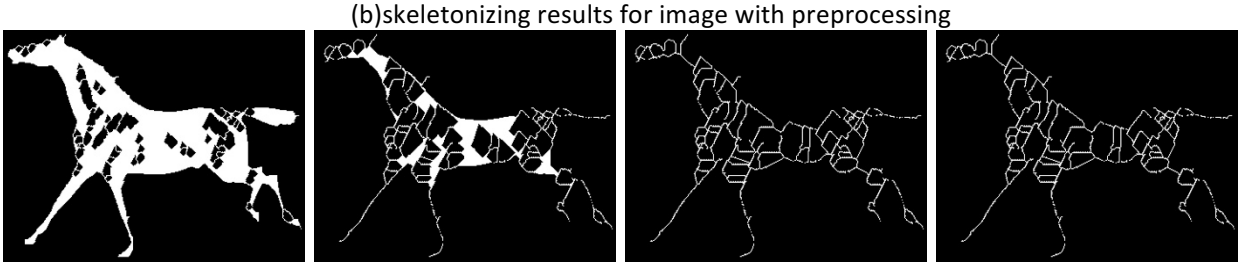


5 times 15 times 35 times 45 times



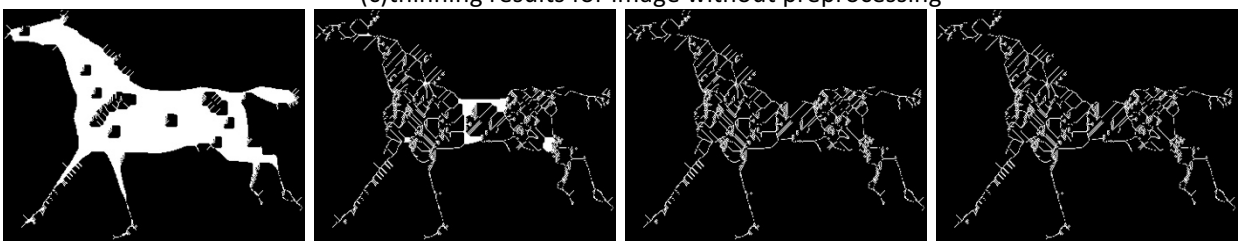
improved image after 45 times skeletonizing

(b)skeletonizing results for image with preprocessing



5 times 15 times 35 times 45 times

(c)thinning results for image without preprocessing



5 times 15 times 35 times 45 times

(d) skeletonizing results for image without preprocessing

Figure.17 thinning and skeletonizing result for the horse image

Compared with (a) and (c), (b) and (d), the holes and contours play an important role in thinning and skeletonizing. As stated in Fig.16, the image after preprocessing still has holes but it is improved a lot. From 5 times to 45 times, we can see the pixels of the object are transformed to the shape of the object (the pixels are reducing). This process not only starts at the boundary of the figure but also at the holes. Since the holes also create the boundary. So we can see that in (c) and (d), which has more holes than those in (a) and (b), the resulting images has lots of " ". Also, by smoothing the boundary, the resulting image (a) and (b) have less . The "blocks" and "radial like" lines are useless information for describing the shape of the figures and can affect the correct expression. So based on the above comparison, preprocessing is very important since we want to fill all the holes to reduce the "blocks" and smooth the boundaries to reduce the "radial like" lines. As shown in Fig.17, the improved image after 45 times thinning or skeletonizing. We just want to extract the main shape of the object in the image.

R

[1] D.Shaked, N. Arad, A.Fitzhugh, I. Sobel, "Color Diffusion: Error-Diffusion for Color Halftones", HP Labs Technical Report, HPL-96-128R1, 1996.