Problem1-Geometric Image Manipulation

a. Geometrical Warping

Abstract and Motivation

Geometrical modification is one of the most common operation in image processing. It includes rotation, spatial translation and warping into a different shape. In our daily life, many girls like using some selfie applications such as B612 in order to add some special effects to their photos. These effects such as thinning their faces or changing the square photos into a disk shape is based on geometrical manipulation. Now, we have to implement this kind of processing to some images.

Task: Design and implement a spatial warping technique that transforms an input square image into an output image of a disk-shaped image. Also, we need to do reverse spatial warping to recover its original image and compare them with the original square images. .

Approaches and Procedures

Theoretical Approach: There are 3 ways to do implement this effect. Here are the figures of 3 methods:

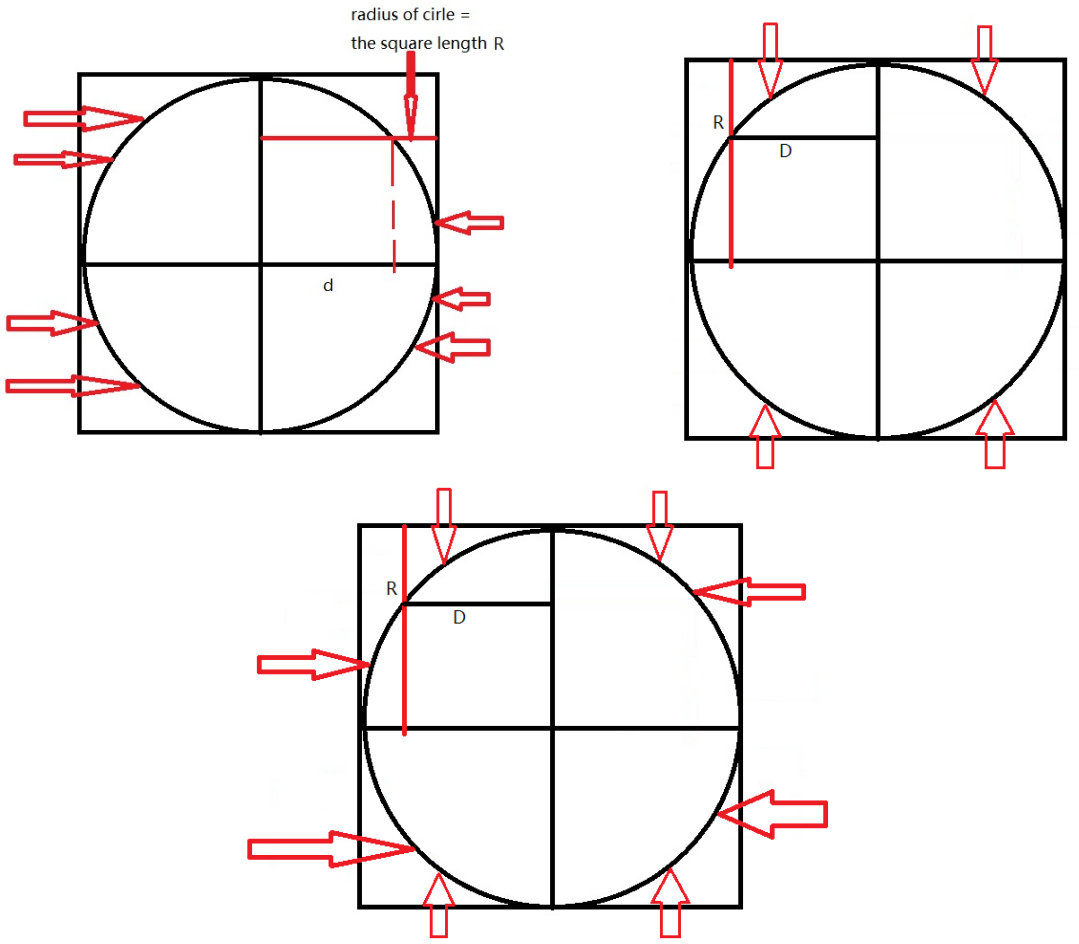

Figure1 schematic diagram of 3 methods

We can regard an image as an elastic square that can be squeezed into a disk along horizontal or vertical directions. Because the image size is 512*512, we can divide the whole image into 4 parts, which is equivalent to squeeze

4 small square into 4 sectors and stitch them together. The first way is squeezing it in the direction of horizontal. In each small square(size is 256*256), we can regard it as a matrix whose size is 256*256 so that it has 256 rows. For each row in the original image, the length is R. We want to squeeze them in to length d. Therefore, the scaling ratio is determined by d over R. It is obvious that we can express the equation of target image(disk-shaped), which is

$$x^2 + y^2 = 256^2$$

Because we only squeeze it in horizontal direction, the column index is unchanged so that for each column, the value of x is related to the column index. Then we can calculate the corresponding y in target image. Next, we have to map this (x,y) in target image into original image. If it is still in the original image, we can use bilinear interpolation to calculate its pixel value. Otherwise, the value of this pixel is 0. Similarly, if we exchange two variables x and y, we can get method 2. Also, if we take into 2 variables into consideration, we can implement method 3.

Note: in order to insure the coordinate of every pixel is nonnegative, I use a different coordinate system:

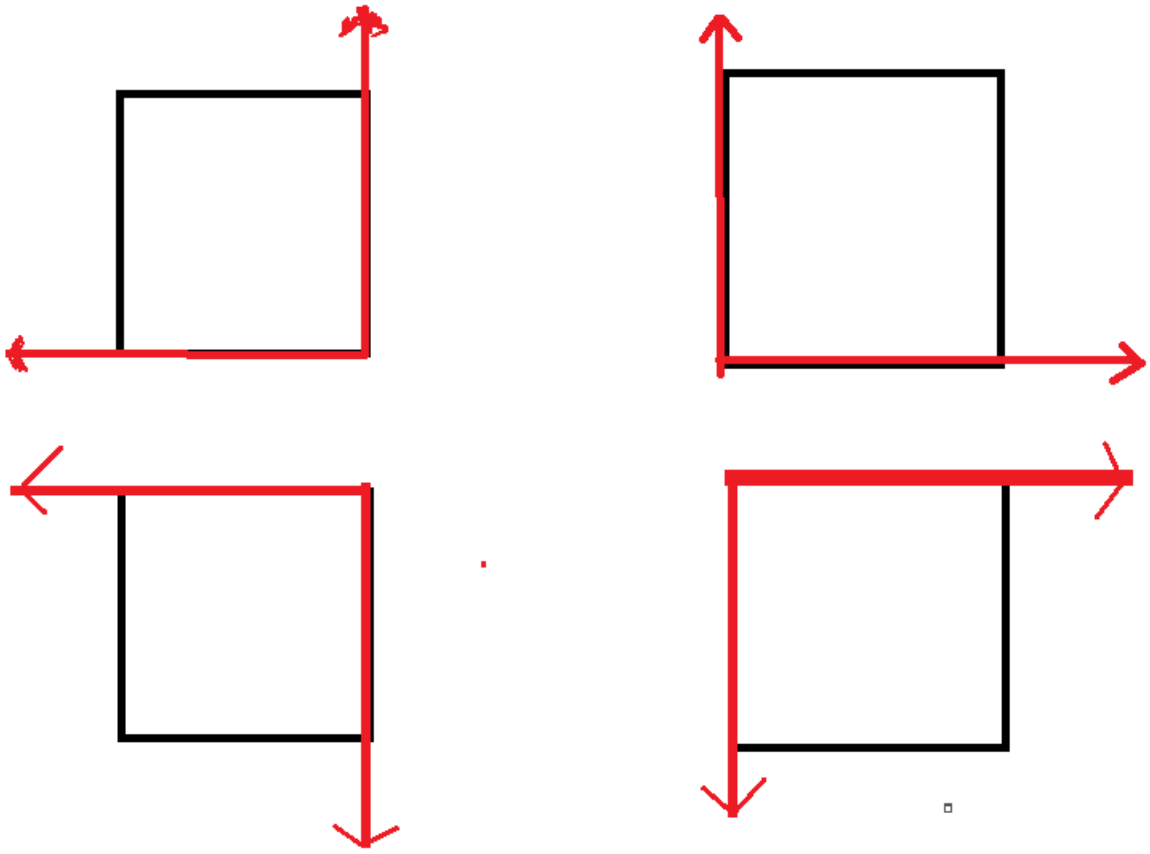

Figure2 coordinate system for each small square

The reverse operation is much easier after finishing the warping. We have to do scaling manipulation to the target image (disk-shape). The scaling factor is R over d. Then, using bilinear manipulation ,we can recover the square image.

Implementation by C++(Take panda.raw as an example)

Step1: Read input image “panda.raw” and store the image data in an unsigned char vector (1D). Read key parameters from command line including its height and width and number of channels.

Step2: Run two nested loop for getting access to every pix value. Divide this square image into 4 small squares (row0-255 col0-255, row0-255 col 256-511, row 256-511 col 0-255, row 256-511 col 256-511). Also, I stored them into 4 1-D channels. For convenience of processing, I separate their channels and store them in 2-D vectors.

Step3:For each small square, calculate the scaling ratio by previous formula and implement coordinate mapping.

If the mapping pixel position is in the original pixel, I implement bilinear interpolation to calculate the pixel value and assign it to the target image. If it is outside from the original image, the pixel value of this position in target image is 0.

Step4: Combine 4 parts together and output the processed image by a function called fwrite.

Step5: For inverse manipulation, it is similar to the forward manipulation. First, calculate the scaling ratio for each row or column. Then, do the pixel position mapping and bilinear interpolation in order to recover it.

Step6: Combine 4 parts together and output the processed image by a function called fwrite.

Results:

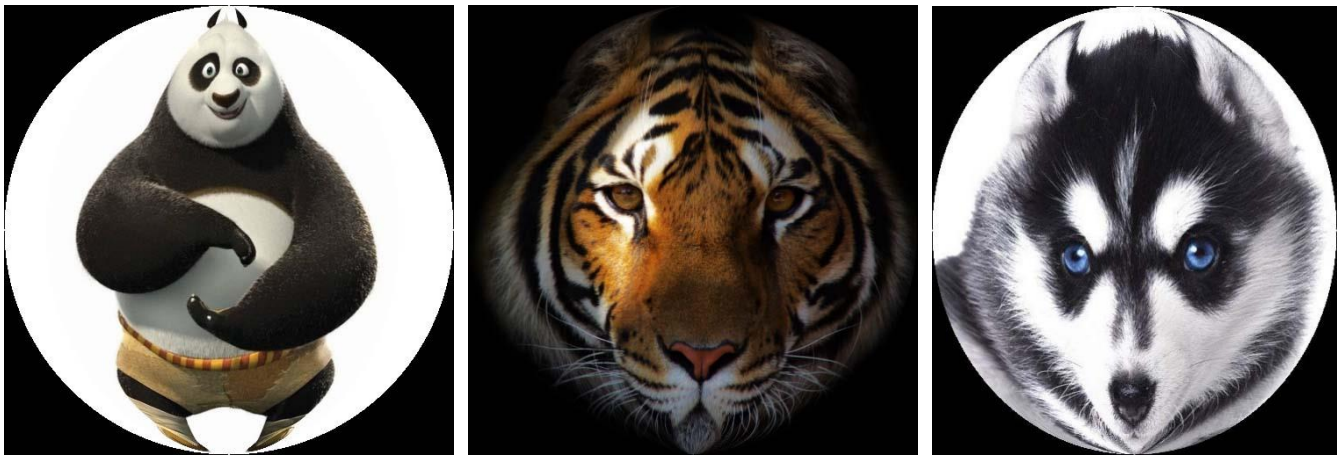

Figure3 disk-shaped images

Figure 4 recovered image.

Discussion:

Obviously, there are many differences between the edges of recovered images and original images. Because the disk-shaped images is created by the bilinear interpolation of original image. Also, the recovered image is created by doing bilinear interpolation to the disk-shaped image. Therefore, it will cause some loss of edge details. Some pixels near to edge will map into the dark area in the disk-shaped image. Therefore, we can see some black lines in 4 edges.

b. Homographic Transformation and Image Stitching

Abstract and Motivation

In present smart phones, the camera can take panorama pictures. Also we can stitch serval images into a

panorama photo. These manipulation is based on homographic transformation. In computer vision, if we want to map a pixel q in the Cartesian plane into an object plane. We can use the homographic matrix to do this:

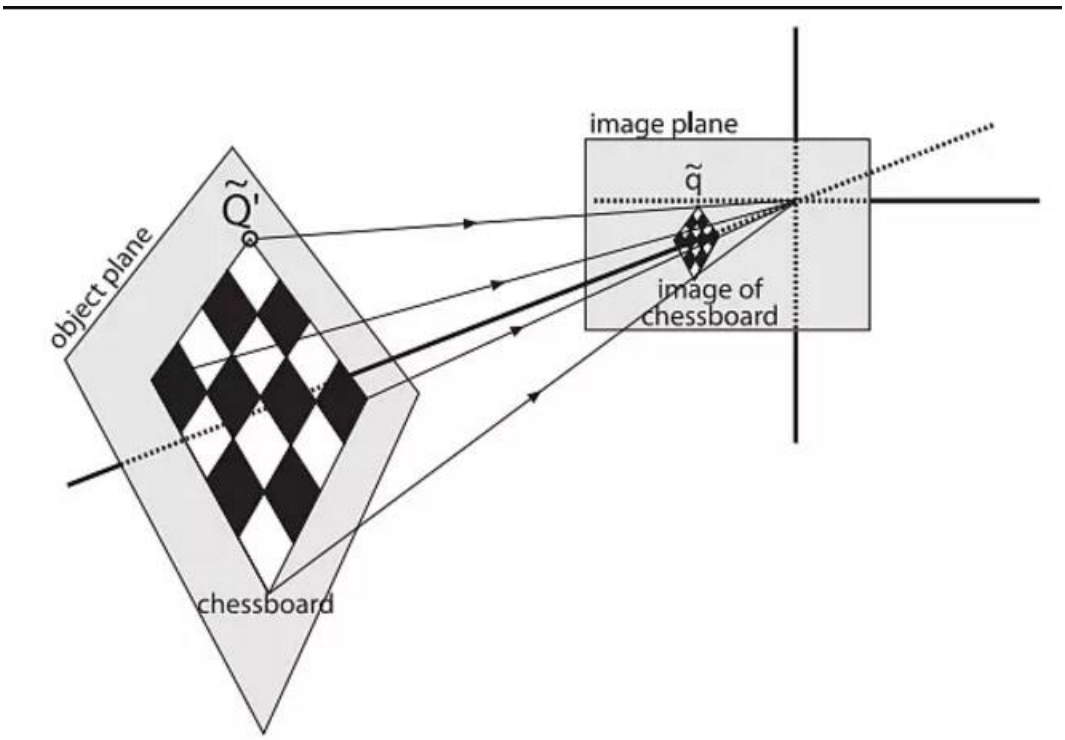

Figure3 An example of homograph.

Task: Using homographic matrix to combine 3 photos into a panorama photo.

Approaches and Procedures

Theoretical Approach: The key factor of this task is to find the correct coordinate relationship between two images. The homographic transformation procedure is stated below. Images of points in a plane, from two different camera viewpoints, under perspective projection (pin hole camera models) are related by homography:

$$P_2 = HP_1$$

where H is a 3*3 homographic transformation matrix, P1 is the coordinate before the transformation and P2 is the target coordinate. Specifically, this equation can be written in this form:

$$\begin{bmatrix} x'_2 \\ y'_2 \\ w'_2 \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} \text{ and } \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \frac{x'_2}{w'_2} \\ \frac{y'_2}{w'_2} \end{bmatrix}$$

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1X_1 & -y_1X_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2X_2 & -y_2X_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3X_3 & -y_3X_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4X_4 & -y_4X_4 \\ 0 & 0 & 0 & x_1 & y_1 & 0 & -x_1Y_1 & -y_1Y_1 \\ 0 & 0 & 0 & x_2 & y_2 & 0 & -x_2Y_2 & -y_2Y_2 \\ 0 & 0 & 0 & x_3 & y_3 & 0 & -x_3Y_3 & -y_3Y_3 \\ 0 & 0 & 0 & x_4 & y_4 & 0 & -x_4Y_4 & -y_4Y_4 \end{bmatrix} \times \begin{bmatrix} H_{11} \\ H_{12} \\ H_{13} \\ H_{21} \\ H_{22} \\ H_{23} \\ H_{31} \\ H_{32} \end{bmatrix} = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ Y_1 \\ Y_2 \\ Y_3 \\ Y_4 \end{bmatrix}$$

Figure3 specific formula[1]

$x_1, y_1 - x_4, y_4$ is 4 control points.

Note: Actually, we can select more than 4 points. We can express the expansion form of the matrix equation as AH=B. For example, if we select 8 pts, (x5,y5)-(x8,y8) can be added below the (x4,y4). Therefore, size of matrix A is equal to 16*8. We have to use the least-square error method to solve this problem. H is equal to:

$$H = (A^T A)^{-1} A^T B$$

After choosing the control points and calculate the homographic matrix, we can set the boundary of our image. I divide it into 2 stage:

First, I calculate the homographic matrix between middle image and right image. It is intuitive to think that be upper right corner and lower right corner is determined by the right image. Therefore, I plugin the coordinates of the upper right and lower right corner into matrix H to calculate the corresponding coordinates in the target image. Then I can get the new image height and new image width for the stage I. For each pixel in the target image of stage I, I use the inverse of H to multiply with the coordinates of target pixels, which can calculate its mapping of the middle image. If it is in the inside of the middle image, use bilinear interpolation to calculate its value. If not, assign 0. If the pixel is in the inside of original right image, copy the pixel value into the target image. After stage I, we can get a combination image of middle and right images.

Stage II: The control points between the left image and stitched middle image will be different from the initial points. This is not a big problem since only y coordinate will change and the new coordinates can be calculated. Therefore, we use the new pixel coordinates to calculate the homographic matrix for stage II. Then, do the similar operation as stage I. Finally, we can create a panorama of three images.

Implementation by C++:

Step1: Open 3 images in Photoshop and select 8 pairs of control points, 4 for each image.

Step2: Change the row and column index into Cartesian coordinates and using MATLAB to calculate homographic matrices. Only calculate stage I's.

Step3: Read 3 images and store the image data in an unsigned char vector (1D). Read key parameters from command line including its height and width and number of channels.

Step4: Do the manipulation of stage I.

Step5: Using the new pixel coordinates to calculate H matrix for stage II.

Step6: Do the manipulation of stage II.

Step7:Loops end and output the processed image by a function called fwrite.

Results:

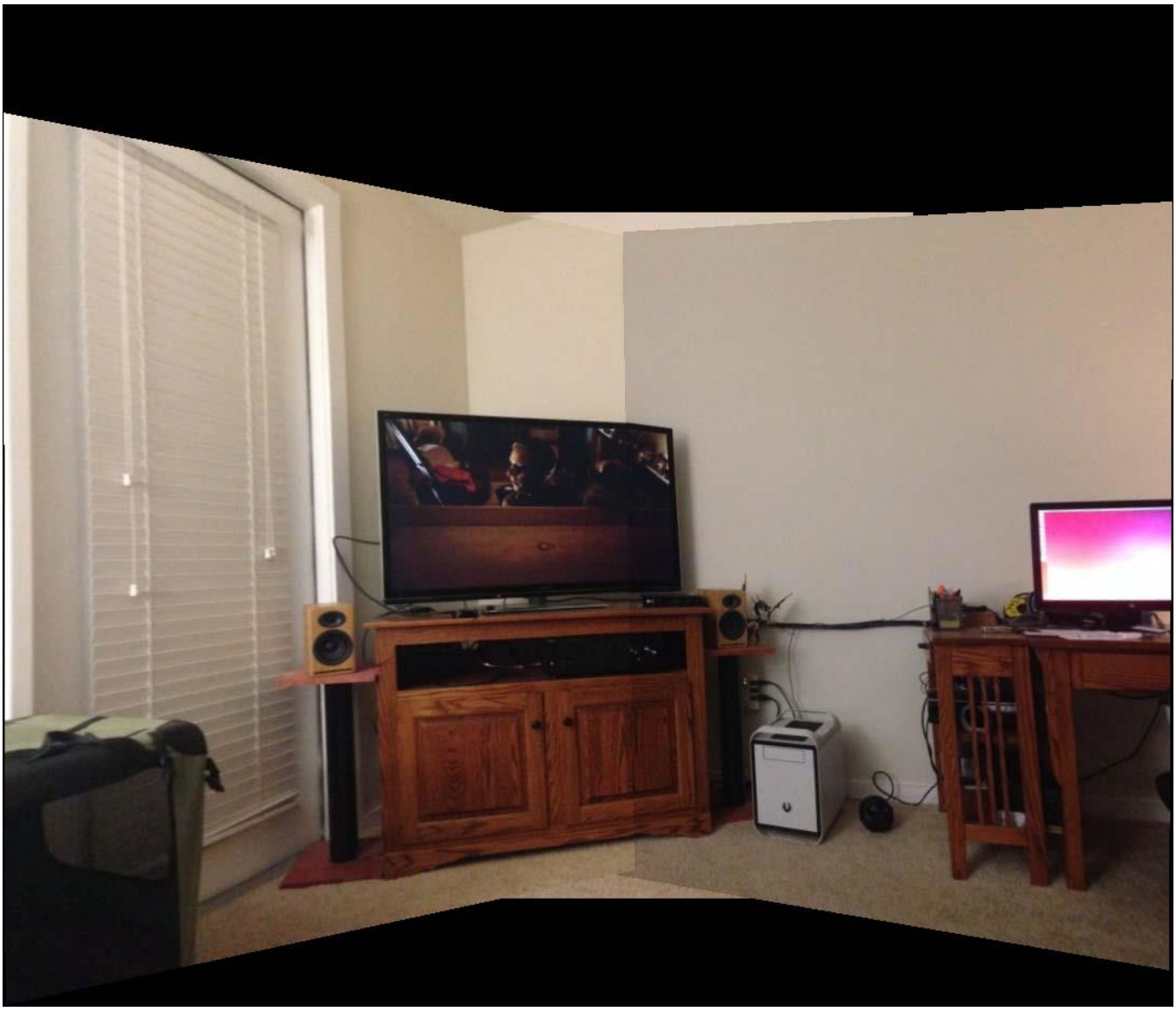

Figure4 Panorama image

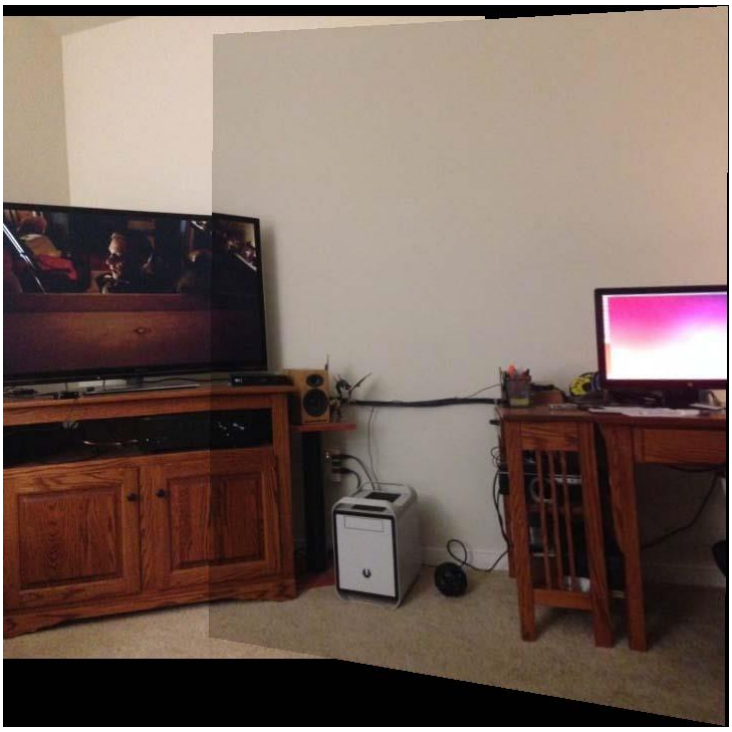

Figure5 Middle-Right combination (4) control points

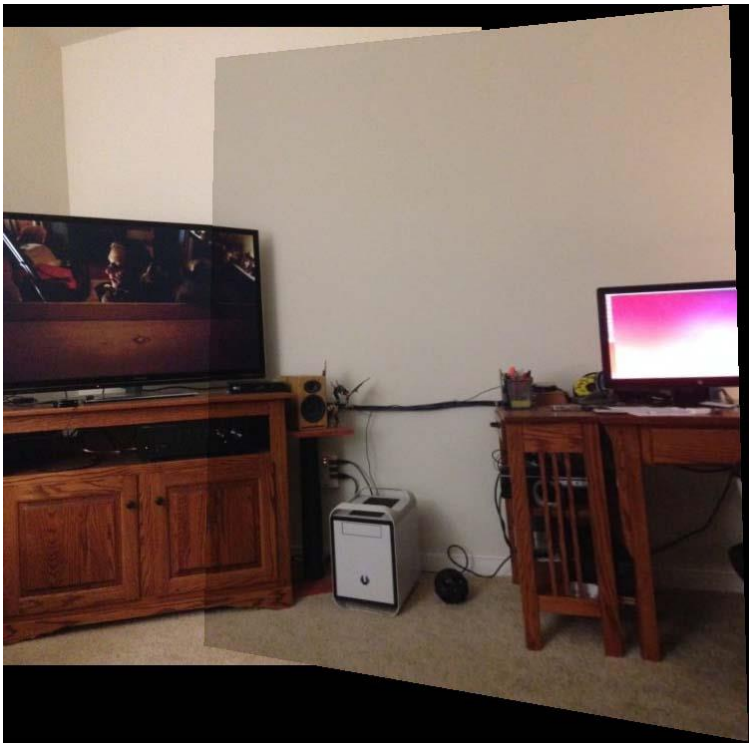

Figure6 Middle-Right combination (8) control points

Discussion:

I used 4 pairs control points to finish this task and get figure4. If we use more control points, we can get better results. Because the relationship between 2 different coordinate system is complex. If we use more control points, the mapping error will be reduced. You can compare figure5 with figure6. We can find that in figure 6, the TV matches better than figure5.

I often select points with some special feature. As we can see, the wall corner of the room and the TV screen, the two black dots on the cabinet…etc, can be regarded as our ideal control points. We can use photoshop to find their location easily and this kinds of points would fit very well.

Problem2-Digital Halftoning

a. Dithering

Abstract and Motivation:

In our daily life, the image can be divided into 2 parts: continuous-tone image and halftone image. For example, the common color photos is a continuous. Halftoning is widely used in printing industry. The intensity of the pixel value is determined by the number of dots in a particular area. If we want to some continuous-tone images, the printer often separates the image into a number of dots. In the printing of the print, the printier with a limited number of sets of ink (only black ink or cyan, magenta, yellow, black ink) to print a small number of different sizes and density. Colors and shades of the output rely on these small dots to represent, which can give the human eye to produce many gray levels or many colors illusion. Two of the most common methods to implement halftoning is dithering and error diffusion.

Task: Implementing halftoning by fixed thresholding, random thresholding, and dithering matrix.

Approaches and Procedures

Theoretical Approach:

Fixed threshoulding:

The key of this method is to change the original image to a binary image. There are only two pixel values in the image, 0 and 255. We have to select a threshold. If the pixel value is greater or equal to the threshold, the change its pixel value to 255. Otherwise, the new pixel value is 0. An intuitive choice of the threshold is 127.

Implementation by C++:

Step1: Read input image “colorchecker.raw” and store the image data in an unsigned char vector (1D). Read key parameters from command line including its height, width, number of channels.

Step2: Run two nested loop for getting access to every pixel value. The index for outer loop ranges from 0 to (height-1) and the inner one ranges from 0 to (width-1). If the pixel value is greater or equal to threshold, change it to 255. Otherwise, change it to 0.

Step3: Loops end and output the processed image by fwrite function.

Random Theresholding

Instead of choosing a fixed threshold, we have to use the random number generator to generate a threshold for each pixel. Similarly, if the pixel value is greater than the random number, set its pixel value to 255. Otherwise, set it to 0.

Implementation by C++:

Step1: Read input image “colorchecker.raw” and store the image data in an unsigned char vector (1D). Read key parameters from command line including its height, width, number of channels.

Step2: Run two nested loop for getting access to every pixel value. Use function rand (i , j) to generate threshold. The index for outer loop ranges from 0 to (height-1) and the inner one ranges from 0 to (width-1). If the pixel value is greater or equal to threshold, change it to 255. Otherwise, change it to 0.

Step3: Loops end and output the processed image by fwrite function.

Dithering matrix:

In this method, the parameters are determined by an index matrix. In this problem, given the fist index method is

$$I_2(i,j) = \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix}$$

Figure 7 I2 matrix

Bayer [2] introduced a matrix family based on the previous matrix:

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

Figure 8 I2n matrix

In this method, the pixel value is normalized to the range from 0 to 1. Therefore the threshold matrix T is defined by:

$$T(x,y) = \frac{I(x,y) + 0.5}{N^2}$$

Figure 9 T matrix

N square is the number of pixels in the matrix so that it can take the values such as 2, 4, 8… Also the decision rule is defined as :

$$G(i,j) = \begin{cases} 1 & \text{if } F(i,j) > T(i \bmod N, j \bmod N) \\ 0 & otherwise \end{cases}$$

Figure 10 Decision Rule

G(i,j) is the normalized pixel value.

Implementation by C++:

Step1: Read input image “colorchecker.raw” and store the image data in an unsigned char vector (1D). Read key parameters from command line including its height, width, number of channels.

Step2:Create the index matrix I2, I4 and I8 and calculate its corresponding threshold matrix. Run two nested loop for getting access to every pixel value and normalize it to (0, 1). Create 6 variables to store the result of modulo operation for each i and j and 3 different index matrices.

Step3: Compare the normalized value with T(i mod N, j mod N). If it is greater than the thereshold, set it to 255. Otherwise, set it to 0.

Step4: Loops end and output the processed image by fwrite function.

Four intensity levels display:

The theoretical approach of this method is quite similar to the dithering matrix method. The decision rule for the output pixel G(i,j) is:

$$0, if\ 0 \le F(i,j) \le 255 * \frac{T(i\%N, j\%N)}{2}$$

$$85, if\ 255*\frac{T(i\%N,\ j\%N)}{2} < F(i,\ j) \le 255*T(i\%N,\ j\%N)$$

$$170, if\ 255*T(i\%N,\ j\%N) < F(i,\ j) \le 255*\frac{T(i\%N,\ j\%N)}{2}+127$$

$$255, if\ 255*\frac{T(i\%N,\ j\%N)}{2}+127 < F(i,\ j)$$

F(i,j) is the input pixel.

Implementation by C++:

Step1: Read input image "colorchecker.raw" and store the image data in an unsigned char vector (1D). Read key parameters from command line including its height, width, number of channels.

Step2:Create the index matrix I2, I4 and I8 and calculate its corresponding threshold matrix. Run two nested loop for getting access to every pixel value Create 6 variables to store the result of modulo operation for each i and j and 3 different index matrices.

Step3: Compare the normalized value with T(i mod N, j mod N). Set proper pixel value for each output pixel according to the previous decision rule.

Step4: Loops end and output the processed image by fwrite function

Results:

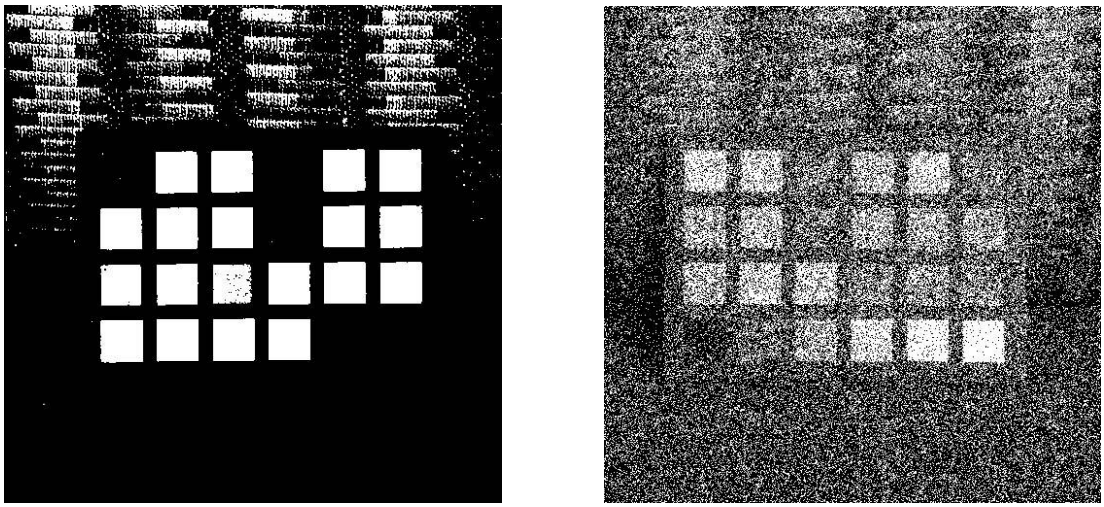

Figure11 Halftoning using fixed and random thresholding.

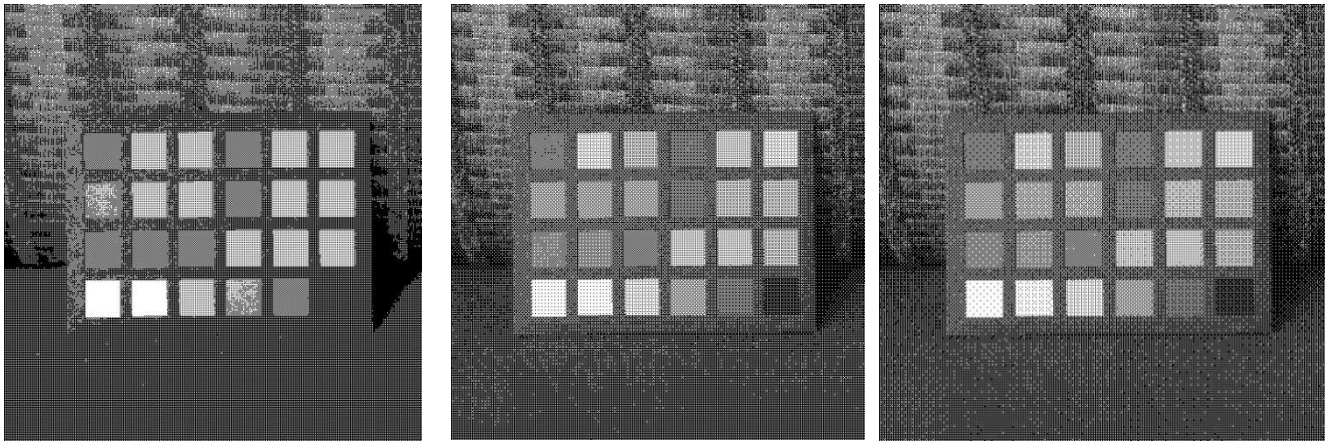

Figure12 Dithering by I2, I4 and I8.

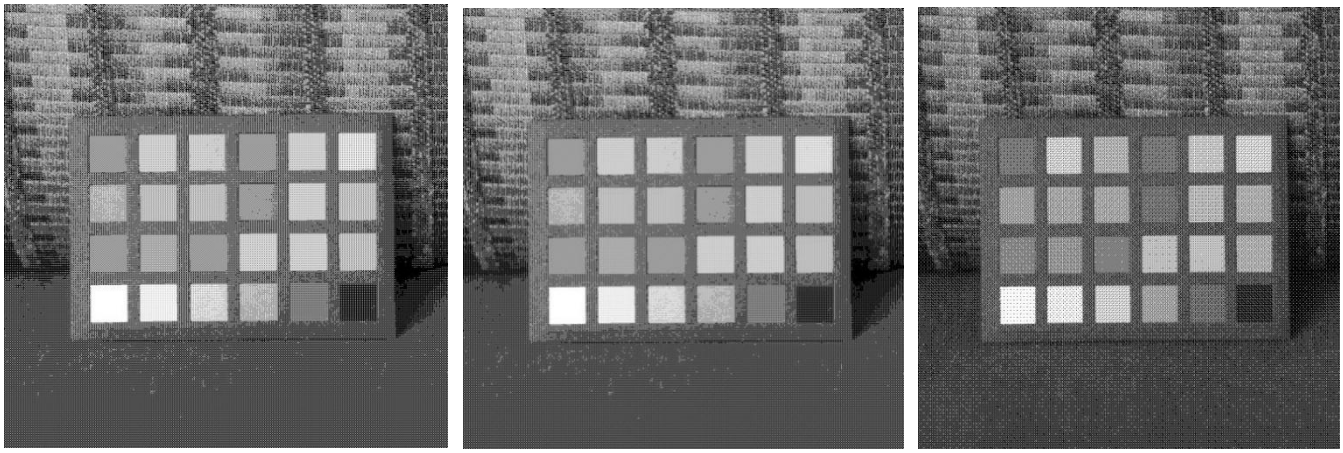

Figure13 Dithering by I2,I4 and I8 using 4-level display

b. Error Diffusion

Abstract and Motivation

After finishing problem a, we can find that the result of halftoning by previous methods is not good because we create many errors by implementing previous methods. Error diffusion will take the quantization error into consideration and it will use a special matrix to diffuse it to the "future pixels". It is popular to convert a continuous-tone image to a binary image. In this way, we can get better halftoning results.

In 1930's, this kind of technique was invented and put into practice but it is limited to analog images. Later, some scientists including Floyd, Steinberg, Jarvis, Judice, Ninke and Stucki invented various kernels which can be used in halftoning of digital images.

Task: To apply Floyd-Steinberg's, Jarvis, Judice, and Ninke's and Stucki's error diffusion matrices to a given Grayscale image and discuss on the results

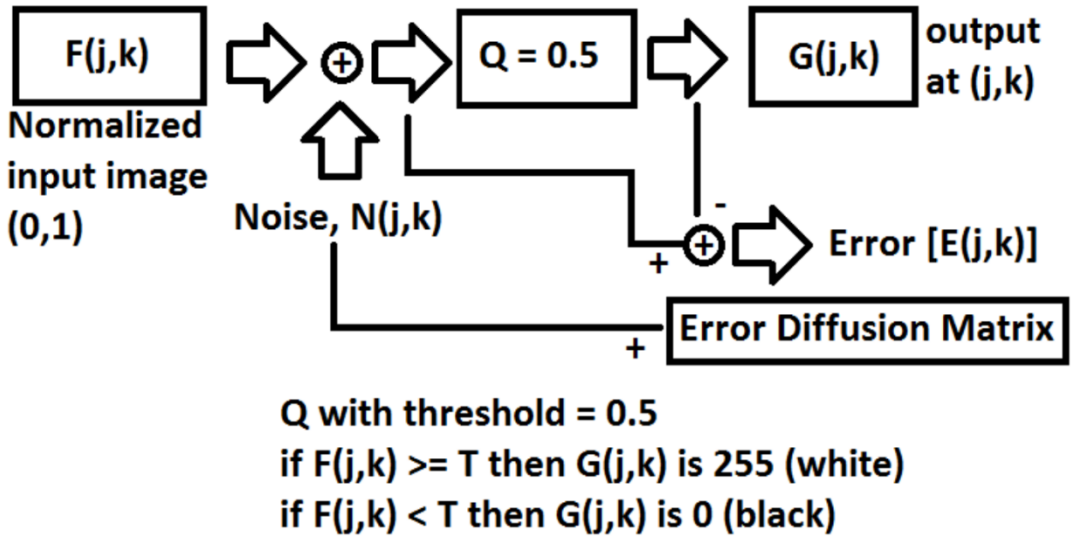

Figure 14 The flow chart of error diffusion algorithm

Approaches and Procedures

Theoretical Approach:

$$\frac{1}{16}\begin{bmatrix}0 & 0 & 0\\ 0 & 0 & 7\\ 3 & 5 & 1\end{bmatrix} \qquad \frac{1}{48}\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 7 & 5\\ 3 & 5 & 7 & 5 & 3\\ 1 & 3 & 5 & 3 & 1\end{bmatrix} \qquad \frac{1}{42}\begin{bmatrix}0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 8 & 4\\ 2 & 4 & 8 & 4 & 2\\ 1 & 2 & 4 & 2 & 1\end{bmatrix}$$

Figure 15 3 error diffusion kernels(F-S, JJN & Stucki)

The first kernel is the Floyd-Steinberg's kernel. The second one is Jarvis, Judice, and Ninke's kernel. The last one is Stucki's kernel. First, we have to expand image according to the size of kernel. Second, we can use a 2-nested loop to access each pixel in the extended image. Then, we can define a window whose size is N*N(N is the kernel size). For the center pixel, if its pixel is greater or equal to 127, set it new pixel value to be 255. Otherwise, the new pixel value is 0. The error is calculated by the old pixel value minus the new pixel value. Finally, we can use the kernel to diffuse this error to the pixel's future neighbors. Note: For the odd rows in the original image, we move the kernel forward. For even rows, we move it backward. Here are the diagrams of this operation:

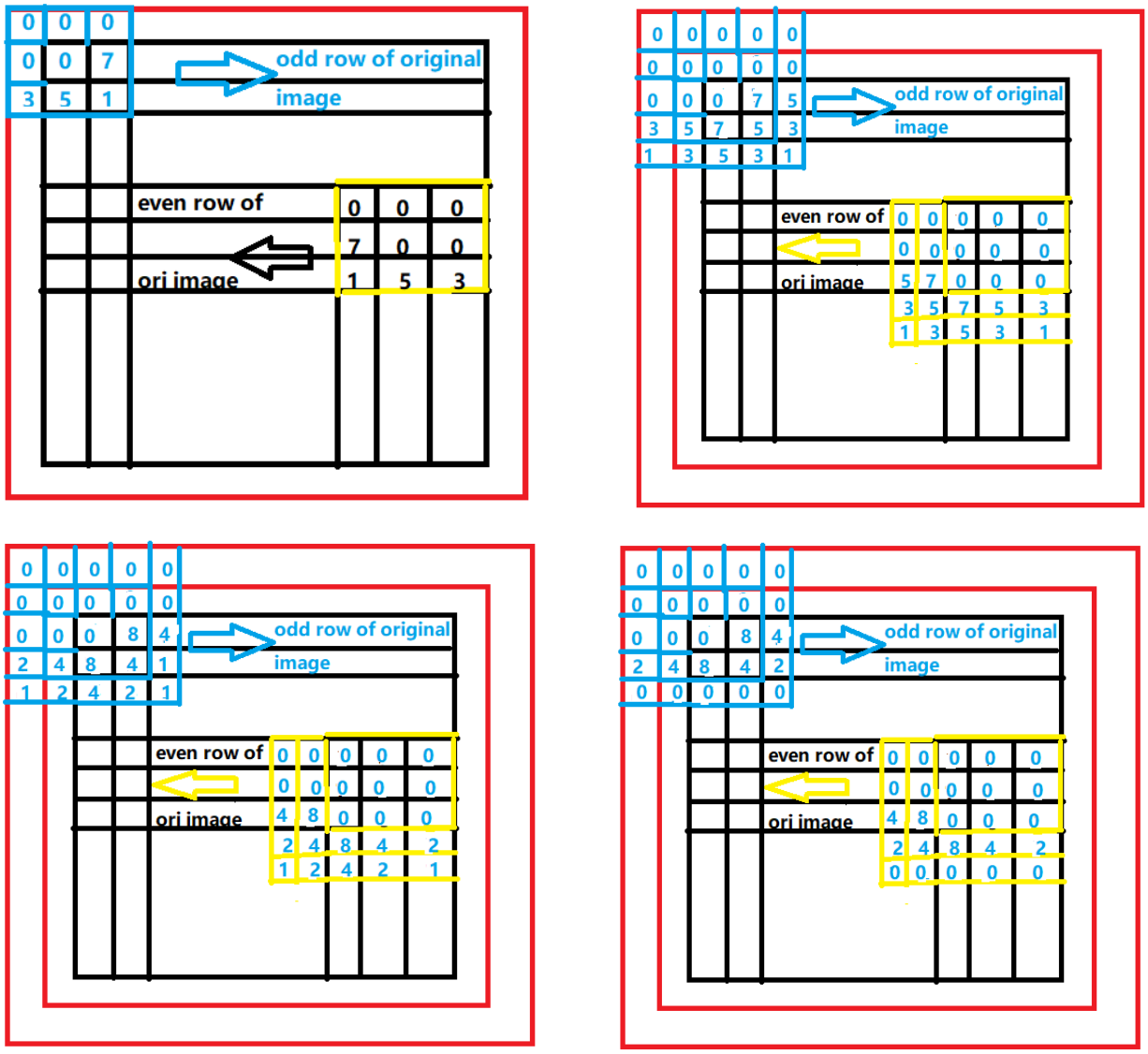

Figure 16 The detailed flow chart of each kernel. Last one is Burke's kernel

Implementation by C++ (Take F-S error diffusion as an example):

Step1: Read input image "colorchecker.raw" and store the image data in an unsigned char vector (1D). Read key parameters from command line including its height and width and number of channels.

Step2: Run two nested loop for getting access to every pix value. The index for outer loop ranges from 0 to (imageheight-1) and the inner one ranges from 0 to (imagewdith-1). If the pixel value is greater or equal to 127, the new value is 255, else is 0. Calculate the error.

Step3: Use the F-S kernel to diffuse the error to its neighbors.

Step4: Loops end and output the processed image by a function called fwrite.

Results:

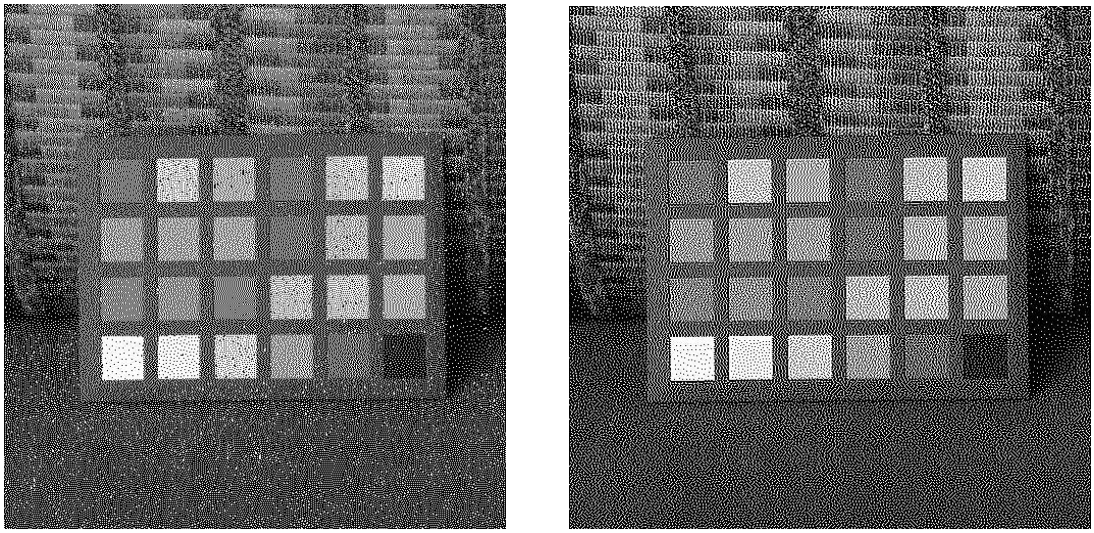

Figure 17 Error diffusion using F-S and JJN's kernel

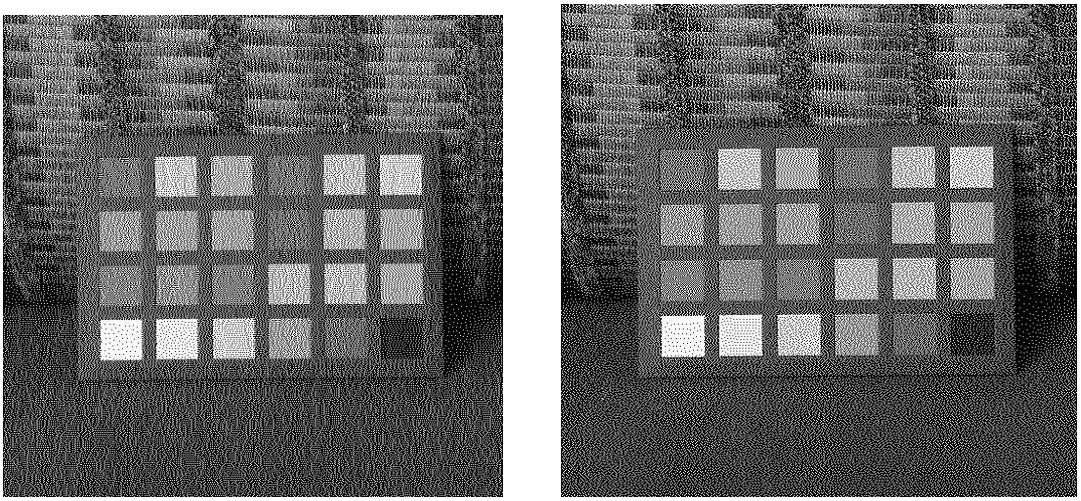

Figure 18 Error diffusion using Stuski's and Burkes's kernel

Discussion:

I found a another kernel named Burke's kernel by searching on the internet. [2]Seven years after Stucki published his improvement to Jarvis, Judice, Ninke dithering, Daniel Burkes suggested a further improvement: He thought the bottom row of Stucki's kernel is not necessary. He removed it and makes the entries of the kernel even, which can calculate the error with only a small hit to the image quality. The image is in Figure 18. Here is the Burkes' kernel:

$$\frac{1}{32}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 4 \\ 2 & 4 & 8 & 4 & 2 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

c. Color Halftoning with Error Diffusion

Abstract and Motivation

In the lecture, professor Kuo told us that we can convert a color image in the RGB domain to CMY domain. Also, it is widely used CMY domain in printing industry because of the ink. After finishing problem a and b, we mastered several skills of halftoning for single channel image. However, in our daily life, people prefer color images. Therefore, in this problem, we have to solve the problem about color image halftoing.

Approaches and Procedures

Implementation by C++:

Step1: Read input image “colorchecker.raw” and store the image data in an unsigned char vector (1D). Read key parameters from command line including its height and width and number of channels.

Step2: Run two nested loop for getting access to every pix value. The index for outer loop ranges from 0 to (imageheight-1) and the inner one ranges from 0 to (imagewdith-1). Convert this RGB image to CMY image.

Step3: For each channel, use the F-S kernel to diffuse the error to its neighbors and finish error diffusion.

Step4: Loops end and output RGB images the processed image by a function called fwrite.

Results:

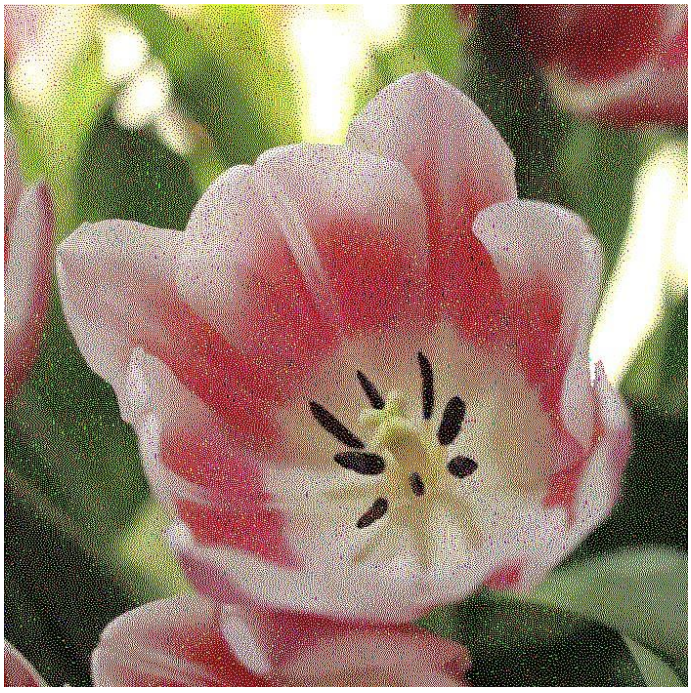

Figure19 Girl_film effect

Problem3-Morphlogical Processing

a. Shrinking

Abstract and motivation

With the development of machine learning and deep learning frame, more and more engineers would like to choose the deep learning frame such as tensorflow and use a number images to train the model such as CNN in order to realize the object recognition. However, this method need a lot of data and we have to label each data, which is a heavy burden to us. When it comes to some simple object, such as the star in a 8-bit image. Actually, we can use some morphological processing method to recognize them.

Task: Use shrinking to count the stars and record the distribution of sizes.

Approaches and Procedures

Theoretical Approach: For a binary image, we define that the background is black(pixel value =0) and the object is white(pixel value=255). Shrinking is to remove the non-background pixels. In a binary image, if an object composed by white pixels does not have a hole, it will eventually be a single pixel after shrinking. If an object with a hole, shrinking will erode it to a connected ring lying in the midway of each hole and the boundary. Here is the flowchart of this manipulation:

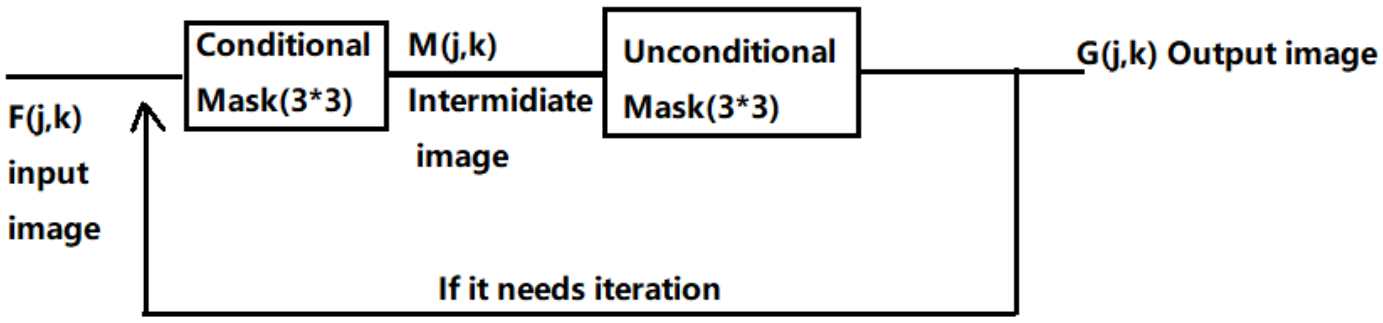

Figure20 flow chart of shrinking

For finding sizes of each star, I use the floodfill algorithm based on Breadth First Search. Generally speaking, the final output of shrinking are some points and each point is one pixel of each star. We can record these pixels' location and use floodfill algorithm to find all pixels in each star. Because the star in the binary image is fully connected, we can start at each remaining point check whether there is any point whose value is equal to 255 among its neighbor. If it exists, push it to queue. Using BFS to implement this manipulation. We can find all pixels of each star.

mean filter:

Step1: Read input image and tore the image data in an unsigned char vector (1D). Read key parameters from command line including its height, width, number of channels and window size.

Step2: Thresholding the whole image into a binary image. If the pixel value is greater or equal to 127, set it to 255, else set it to 0. Get the binary image.

Step3: Looking up the pattern tables record all conditional and unconditional mask for shrinking. Store them into several unsigned char arrays according to their corresponding bond value.

Step4: The first loop is the number of iterations. The next 2-nested loops is to get access to every pixel data in the

binary image. If the pixel value is equal to 255. Calculate its corresponding bond and record its 8 neighbor pixels into a 3*3 array. Compare this window to the corresponding conditional mask. For example, if the bond is 5. We have to compare the array composed by its 8 neighbors and itself with every conditional mask of bond5. If any of those masks matches, we assign 1 to the same location of intermediate image, else we assign 0.

Step5: Run 2-nested loop for access the intermediate image. If the pixel value of intermediate image is 0, we do not change the pixel value of the input image. If it is 1, we have to store 8 neighbors and itself in a 3*3 array. Then, we compare this array with all unconditional masks. If any of the unconditional masks matches, we do not change the pixel value of the input image. If not, we assign 0 to the corresponding location in the input image.

Step6: If the times of iteration is not enough, go back to step 4. Take the output from Step5 as the input of next iteration.

Step7: Iteration over. Then we can count the number of pixels whose values is equal to 255. The number white pixels is equal to the number of stars.

Step8: Use floodfill algorithm based on BFS to find all pixels of each star and count the number of pixels. The result is the size of corresponding star. Then record them and plot the histogram.

Step9: Loops end and output the processed image by a function called fwrite.

Results

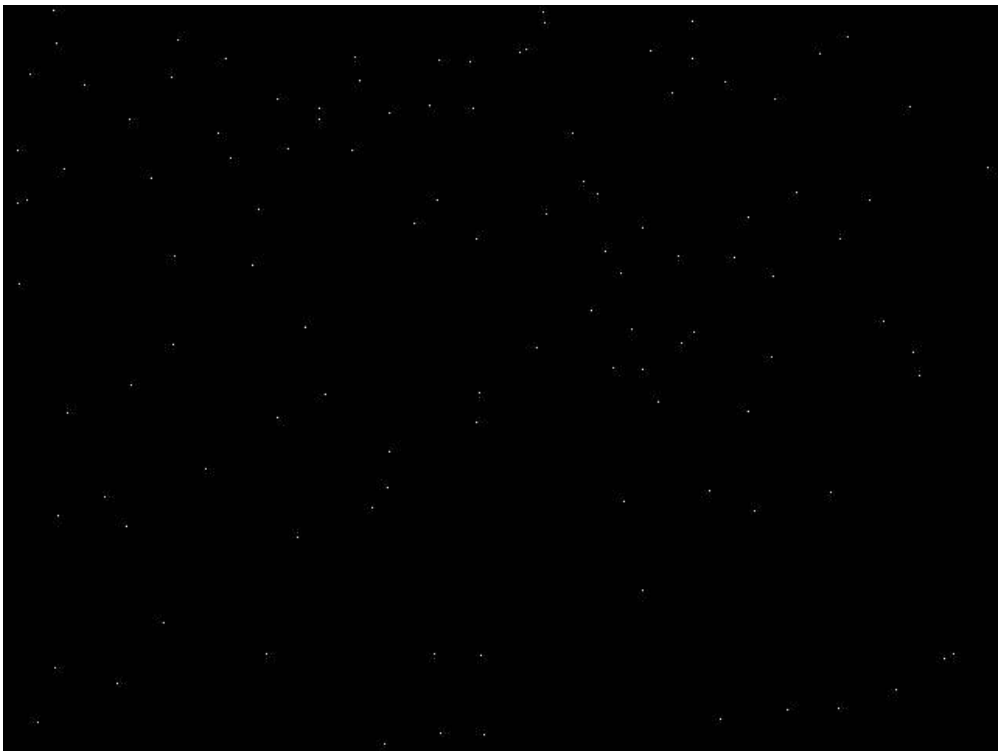

Figure21 Skrinking of star.raw

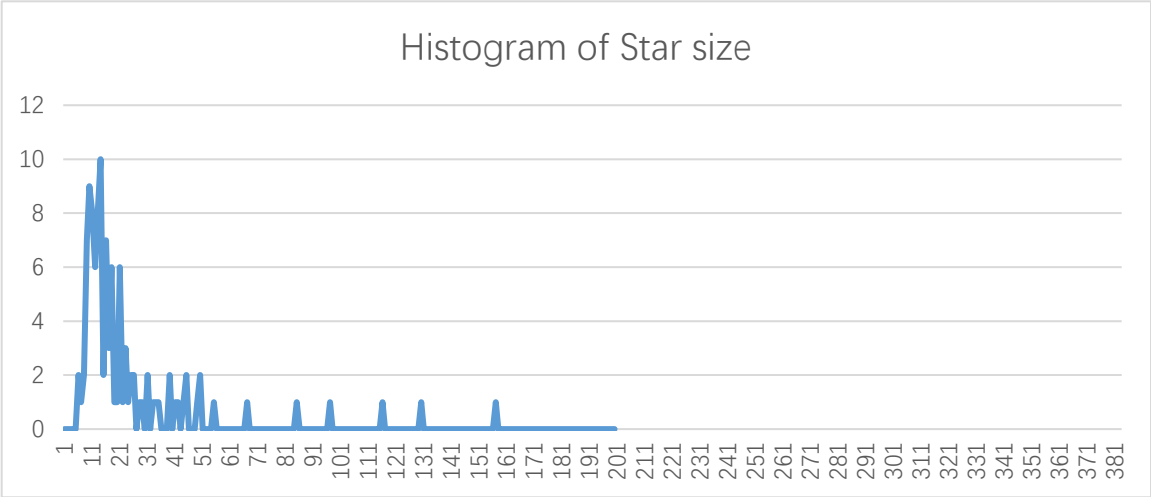

Figure22 Histogram of star size

Figure23 Screenshot of program ending

b. Thining

Approaches and Procedures

Theoretical Approach: For a binary image, we define that the background is black(pixel value =0) and the object is white(pixel value=255). Thining is to remove the non-background pixels but it is less aggressive than shrinking. In a binary image, if an object composed by white pixels does not have a hole, it will eventually be a minimally connected stroke that have same distance to the nearest boundary after thining. If an object with a hole, thining will erode it to a connected ring lying in the midway of each hole and the boundary. Here is the flowchart of this manipulation:

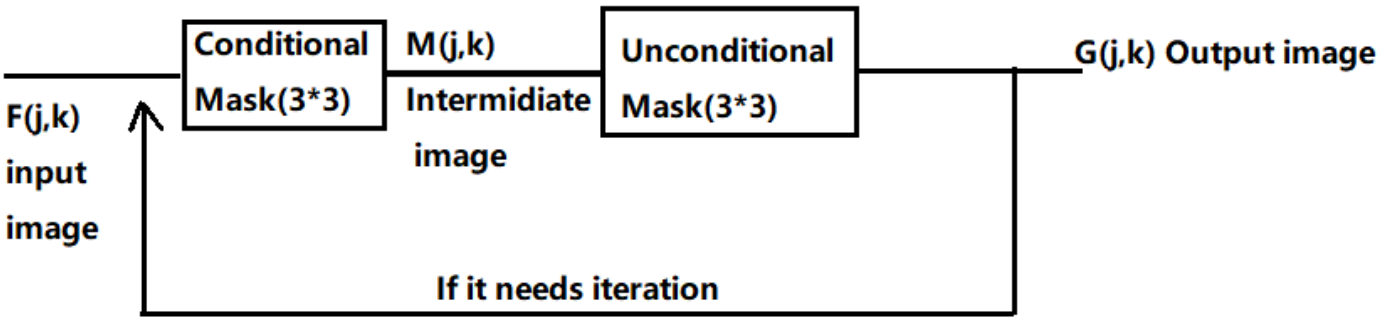

Figure24 Flowchart of thining

Step1: Read input image and tore the image data in an unsigned char vector (1D). Read key parameters from command line including its height, width, number of channels and window size.

Step2: Thresholding the whole image into a binary image. If the pixel value is greater or equal to 127, set it to 255, else set it to 0. Get the binary image.

Step3: Looking up the pattern tables record all conditional and unconditional mask for thining. Store them into several unsigned char arrays according to their corresponding bond value.

Step4: The first loop is the number of iterations. The next 2-nested loops is to get access to every pixel data in the binary image. If the pixel value is equal to 255. Calculate its corresponding bond and record its 8 neighbor pixels into a 3*3 array. Compare this window to the corresponding conditional mask. For example, if the bond is 5. We have to compare the array composed by its 8 neighbors and itself with every conditional mask of bond5. If any of those masks matches, we assign 1 to the same location of intermediate image, else we assign 0.

Step5: Run 2-nested loop for access the intermediate image. If the pixel value of intermediate image is 0, we do not change the pixel value of the input image. If it is 1, we have to store 8 neighbors and itself in a 3*3 array. Then, we compare this array with all unconditional masks. If any of the unconditional masks matches, we do not change the pixel value of the input image. If not, we assign 0 to the corresponding location in the input image.

Step6: If the times of iteration is not enough, go back to step 4. Take the output from Step5 as the input of next iteration.

Step7: Loops end and output the processed image by a function called fwrite.

Results

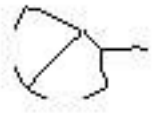

Figure 25 Thining of jigsaw_1

c. Skeletonizing

Approaches and Procedures

Theoretical Approach: For a binary image, we define that the background is black(pixel value =0) and the

object is white(pixel value=255). Thining is to remove the non-background pixels but it is least aggressive method among 3 morphological processing method. In a binary image, the result of an object after skeletonizing is its basic structure. Therefore, it can be used in many fields such as check the correction of the PCI board. Here is the flowchart of this manipulation:

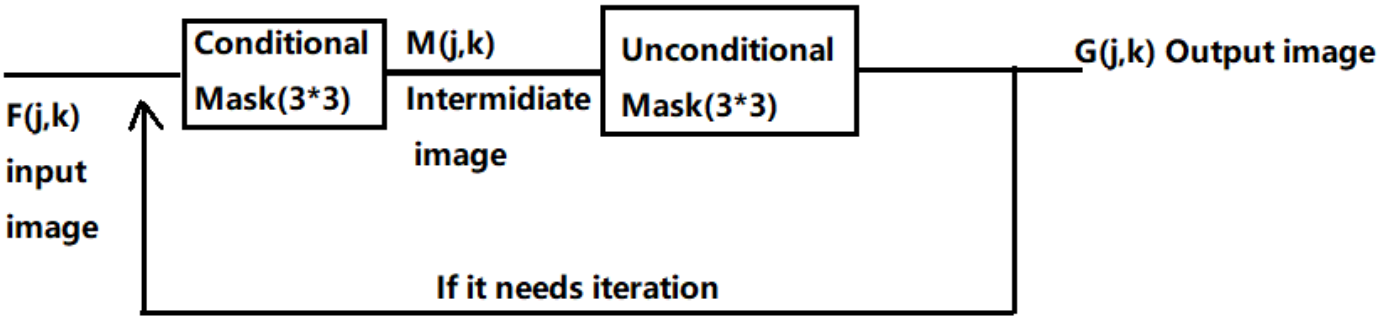

Figure26 Flowchart of skeletonizing

Step1: Read input image and tore the image data in an unsigned char vector (1D). Read key parameters from command line including its height, width, number of channels and window size.

Step2: Thresholding the whole image into a binary image. If the pixel value is greater or equal to 127, set it to 255, else set it to 0. Get the binary image.

Step3: Looking up the pattern tables record all conditional and unconditional mask for skeletonizing. Store them into several unsigned char arrays according to their corresponding bond value.

Step4: The first loop is the number of iterations. The next 2-nested loops is to get access to every pixel data in the binary image. If the pixel value is equal to 255. Calculate its corresponding bond and record its 8 neighbor pixels into a 3*3 array. Compare this window to the corresponding conditional mask. For example, if the bond is 5. We have to compare the array composed by its 8 neighbors and itself with every conditional mask of bond5. If any of those masks matches, we assign 1 to the same location of intermediate image, else we assign 0.

Step5: Run 2-nested loop for access the intermediate image. If the pixel value of intermediate image is 0, we do not change the pixel value of the input image. If it is 1, we have to store 8 neighbors and itself in a 3*3 array. Then, we compare this array with all unconditional masks. If any of the unconditional masks matches, we do not change the pixel value of the input image. If not, we assign 0 to the corresponding location in the input image.

Step6: If the times of iteration is not enough, go back to step 4. Take the output from Step5 as the input of next iteration.

Step7: Loops end and output the processed image by a function called fwrite.

Results

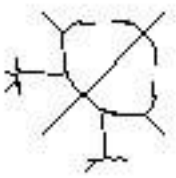

Figure 27 Skeletonizing of jigsaw_2

d. Counting Game

Abstract and motivation

We mastered the 3 basic morphological processing method by finishing previous 3 problems. Also, we do some morphological processing to the jigsaw images. Therefore, this problem will be a game related to jigsaw. We have to use the 3 basic morphological processing method to solve it.

Task: Count the number of pieces in the board image and Find the number of unique pieces in the board image.

Approaches and Procedures

For problem 1, we can apply the shrinking to the "board.raw" because each jigsaw in the board image can be regard as an object without a hole. Therefore, it will eventually be a single pixel point after shrinking. We can count the remaining pixels in the image to calculate the number of pieces. Let the number of jigsaws be k

For problem 2, it is more complex. We can divide this procedures into serval steps:

Step1: Record the remaining pixel locations of the problem1. Store them in an array.

Step2: Each pixel in that array represents a jigsaw. Therefore, I use the previous floodfill algorithm to find all pixel data of each jigsaw and store them in a vector. Note: The floodfill used here is a little different from the shrinking problem. Because the jigsaw has holes, we have to check its 8 neighbors. Also, we have to calculate their size.

Step3: Because we change the input image into a binary image, the edge of the jigsaw will also be changed. Therefore, there will be a difference in the size of jigsaws even if they are the same. I select 20 as my threshold. The initial pair is determined by their size. If the difference between 2 jigsaws' sizes is less than 20, I can regard them as "potentially matching pair". Then, store these pairs in a vector.

Step4: In step2, we find the all pixels of jigsaw. They are stored in a 2-d vector. Their pixel value is equal 255. The first size of the vector is the number of jigsaws. I want to find their boundary. Note that the shape of jigsaw is not regular so that it is hard to make comparison between each jigsaw directly. Therefore, I can restore them in a square or rectangle. The size of the square or rectangle is determined by the jigsaw's boundary. Until now, we have k rectangles or squares instead of k jigsaws. Then, we can record the shape(width, height) of each rectangular.

Step5: It is obvious that the only difference between the matching jigsaws is rotation. After we rotate it by 90 or 180 degrees, we can make them matched. Also, we notice that if the jigsaws are the same their shape must match after rotation. Therefore, we can remove some "fake matching pairs" by checking their size.

Step6: If their size are matching, we can rotate these jigsaws and compare each corresponding pixel values. Also, we need to count the number of matching pixels. Then, I define a similarity percent as the ratio of number of matching pixels and total pixels in the rectangular.

Step7: Theoretically, the similarity percent should be 100%. However, we should take the process of changing into binary image into consideration so that it will reduce a small number of same pixels. Therefore, I choose 0.93 as the threshold for similarity. If the similarity percent > 0.93, the two jigsaws are actually the same.

Step8: Calculate the number of unique jigsaws.

Results

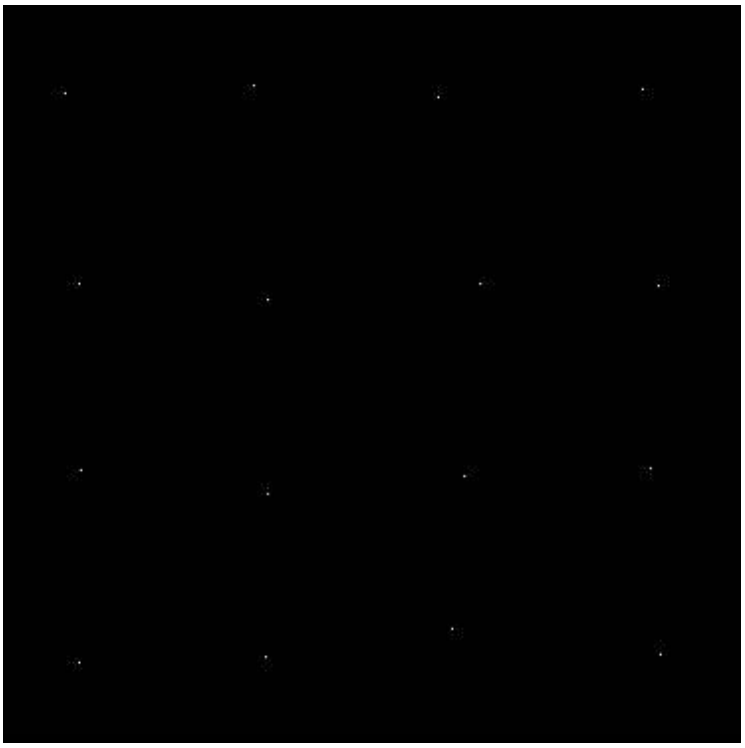

Figure 28 Shrinking of board.raw

Figure 29 Screenshot of program ending

Reference:

[1] http://www.corrmap.com/features/homography_transformation.php

[2] B. E. Bayer, “An optimum method for two-level rendition of continuous-tone pictures,” SPIE MILE- STONE SERIES MS, vol. 154, pp. 139–143, 1999

[3] http://www.tannerhelland.com/4660/dithering-eleven-algorithms-source-code/