EE569 HW3

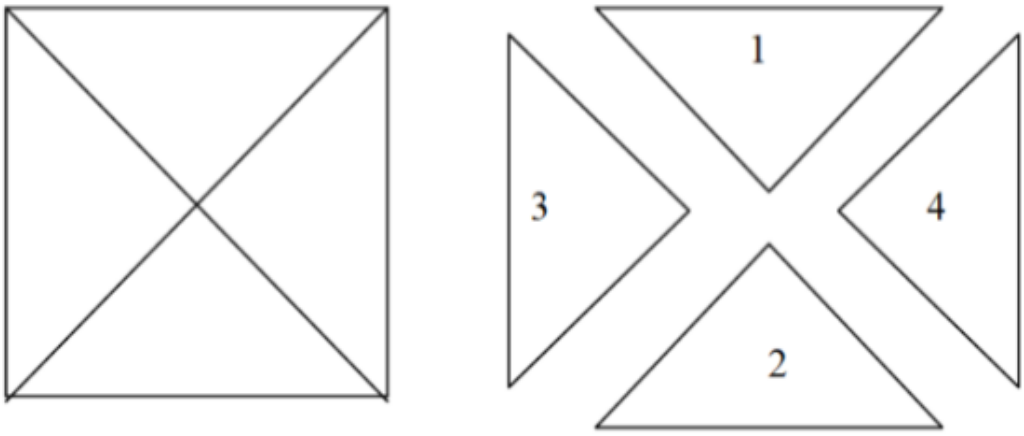
Problem1 Geometric Image Modification

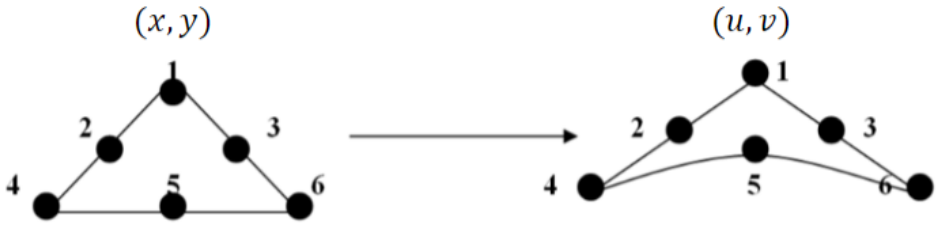
Motivation

image warping is a spatial manipulation technique that can change the shape of the image. It can be used to repair distorted image, as well as creating certain artistic effect. Theoretically, by setting some critical points on the original image and the desired output effect, and calculate the transformation matrix, we can reshape a polygon into another one. And by separating the original image into multiple polygon and calculating each's transformation matrix, we can warp the whole picture.

Procedure

We cut the image into four triangle, within each triangle we calculate a transform matrix, which satisfy the quadratic mapping shown below. And only with quadratic mapping we can generate an arc in the output image. I pick 6 points and then solve the transform matrix with Mathematica.





$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ y \\ x^2 \\ xy \\ y^2 \end{bmatrix}$$

For example, the transform matrix for pixels in triangle 2 is

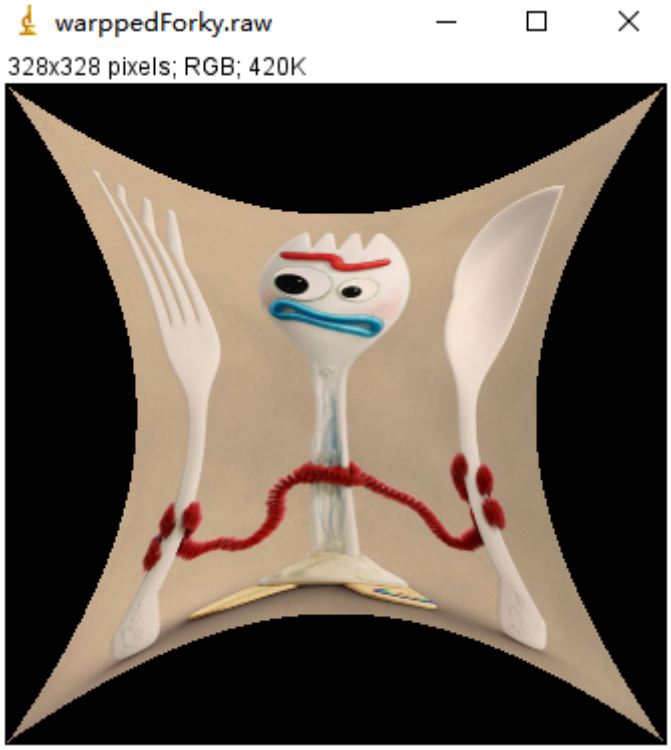
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0.785269 & 0.214731 & -0.00239411 & 0 & 0.00239411 \end{pmatrix}$$

For the inverse mapping, I plan to take the pseudoinverse of this tranform matrix to get the inverse transform matrix. But it doesn't work well. I will discuss the reason and the alternative method in the discussion section.

Result

g++ -o warping warping.cpp

warping raw_images/Forky.raw Output/warppedForky.raw



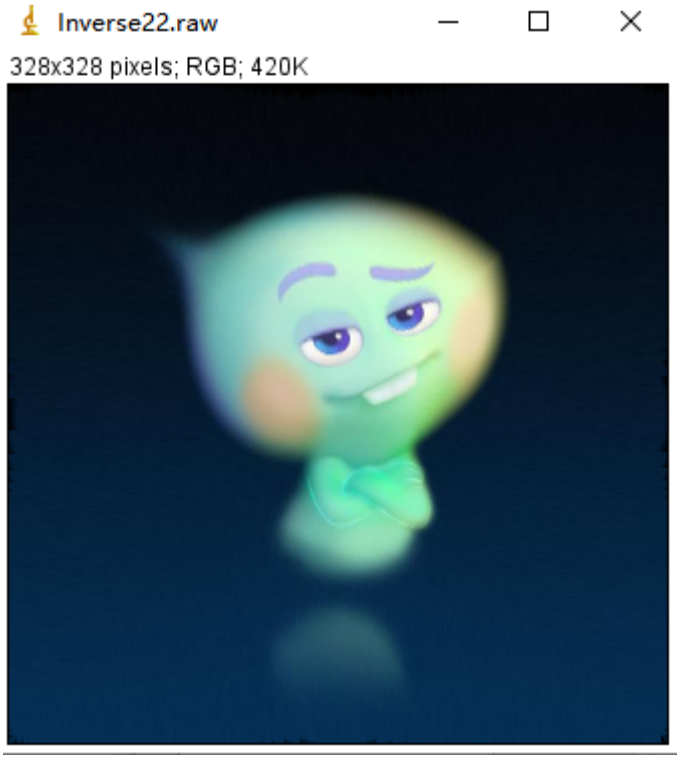
warping raw_images/22.raw Output/warpped22.raw

Inverse mapping:

g++ -o InverseWarping InverseWarping.cpp

InverseWarping Output/warppedForky.raw Output/InverseForky.raw

InverseWarping Output/warpped22.raw Output/Inverse22.raw



Discussion

When following the procedure to find the transform matrix, I met with some problem. Suppose Xi indicate the six supporting vectors (1, x, y, x^2 ...) from input image, and Xo is six coordinates from output image. Our function is then Xo=T.Xi. We can get a transform matrix T using pesudoinverse. But later when I try to find the position in input image correspond to a position in warpped image, I got into trouble. Theoretically we can get it by $T^{+} \cdot (u, v)$, but this actually leads to a lot of distortions. This transform can map unwarpped image coordinate to warpped image coordinate correctly, but the pesudoinverse cannot correctly map it back. Even some critical points are missing.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 \\ b_0 & b_1 & b_2 & b_3 & b_4 & b_5 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ y \\ x^2 \\ xy \\ y^2 \end{bmatrix}$$

Therefore, I created another mapping, which use coordinates from warpped image to compose the supporting vector. And the function becomes Xi=T.Xo.

The transform matrix, for example, the bottom triangle, becomes

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -2.12035 & 3.12035 & 0.00646448 & 0 & -0.00646448 \end{pmatrix}$$

When doing recovery, using just pseudoinverse performs so bad (similar reason as above), so I use the same method in forward mapping, and calculate a new transform matrix instead of pseudoinverse. Only in this way the result is acceptable, or it will lose lots of areas.

Compare the recovered image and the original image, the recovered image contains most of the information but still seems inward hollow, althougth I've got rid of pseudoinverse and generate two separate transformation in both direction. The first possible reason is, when recovering, the actual transformation rule to map an arc to a triangle may be more complicated than our transformation matrix generated by six control points. Maybe adding more control points on the arc and calculate a bigger transformation matrix will perform better.

The second possible reason is when doing forward warping, the star-shape image has already lost may pixel information. When recovering, more pixels should be padded using interpolation. So there'll be more false information in backward warping compared with forward warping. And the image is not clear as the original image, as you can see there're sawtooth on the shoes of Forky.

Problem 2 Homographic Transformation and Image Stitching

Motivation

Homography means the relation between any two image of a same planar surface in space. By calculating proper homographic matrix, we can adjust the 'sight' of the image, as it appears from another perspective. It can be applied to image stitching, which combines multiple image together and make them become a whole picture. It can be done by tranforming all images to a same 'perspective'.

Procedure

First I use SIFT from OpenCV to select some control points. Calculate a homographic transformation shown below by solving 8 linear equation, which correspond to 9-1=8 parameters in the matrix H (H has 8 D.O.F).

$$\begin{bmatrix} x_2' \\ y_2' \\ w_2' \end{bmatrix} = \begin{bmatrix} H_{11} & H_{12} & H_{13} \\ H_{21} & H_{22} & H_{23} \\ H_{31} & H_{32} & H_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

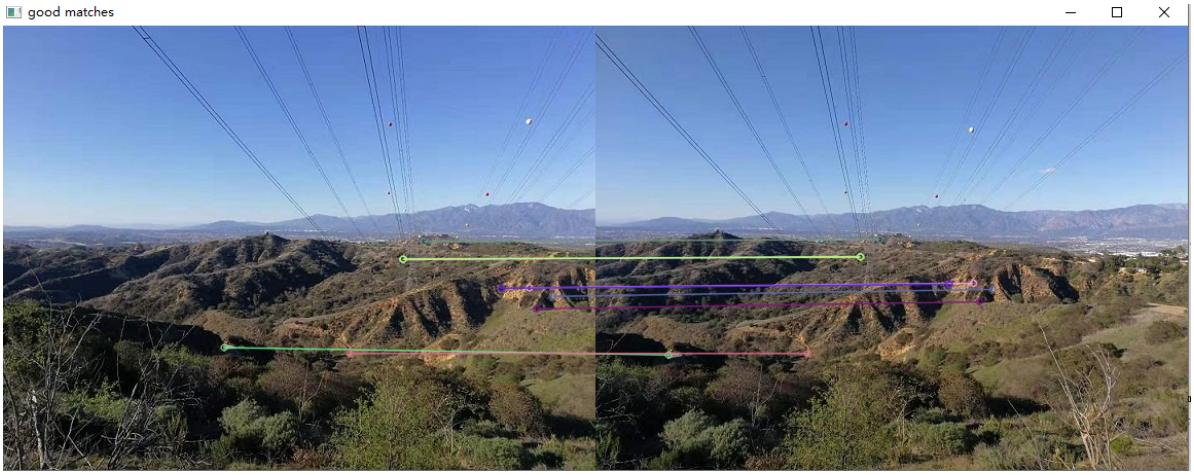
$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \frac{x_2'}{w_2'} \\ \frac{y_2'}{w_2'} \end{bmatrix}$$

Then apply the transform matrix to the scene. Pay attention to the source scene and target scene.

Finally create a new screen and put all images into it. In the area that two picture overlaps, take their average to make it looks smooth.

Result

Totally 1386 control points were found, and 9 of them are considered as good. Matches between left and middle (the color is very light, but there're really 9 pairs).



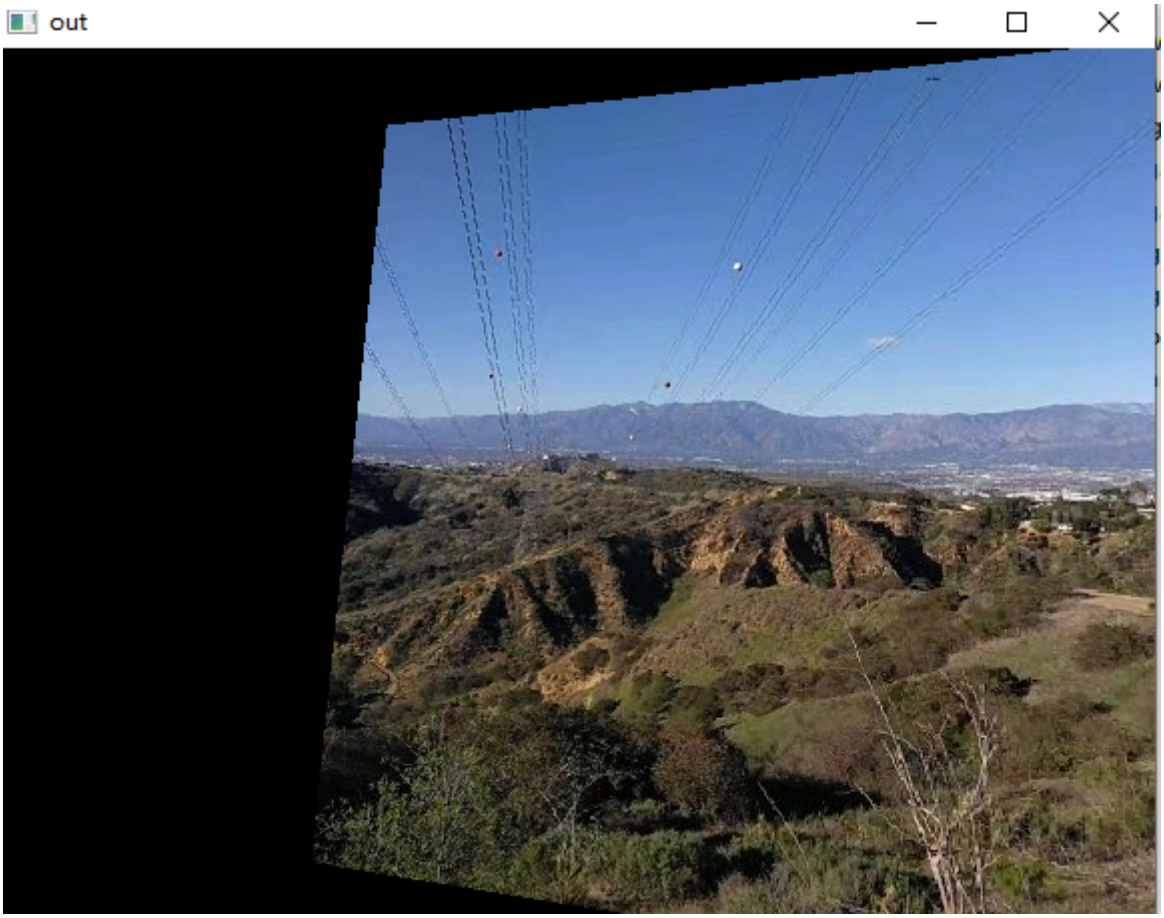
The homographic transformation matrix, for the left scene to middle scene, for example, is

[1.387, -0.0286, -209.6045;
0.158, 1.209, -53.9027;
0.00072, -0.000115, 1]

And the transformed left scene look like this.

The right scene is similar. 1185 control points are found, and 7 of them are considered as good match.

Transformed right scene



Finally the stitched image with Opencv stitching algorithm look like this.

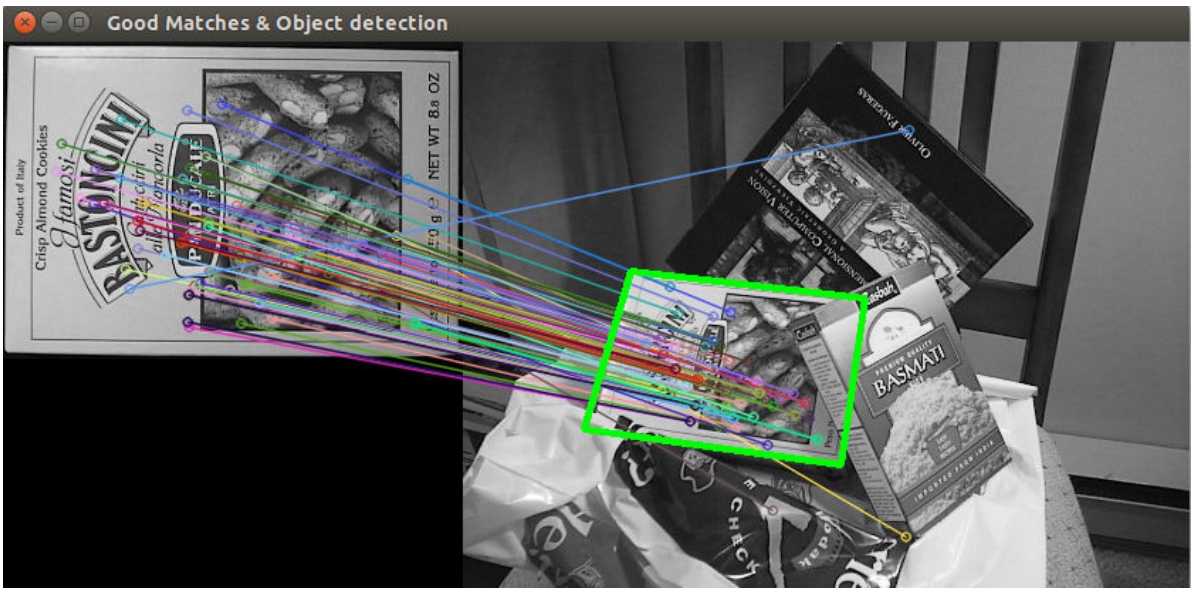
Discussion

As mentioned above, 9 out of 1386 control points were used in [left-middle] transformation, and 7 out of 1185 were used in [right-middle] transformation.

To select the control points, I first use *SiftFeatureDetector* and *DescriptorExtractor* to find feature points. Then I use *BFMatcher.knnMatch* to pair up feature points in two images. Every feature points in image1 is paired with multiple points in image2. Among all the matches, I define that only matches with descriptor difference less than 0.5 is considered as good match (Lowe's ratio test). Then only a few were left.

I met with great problem when trying to display the full transformed scene and failed. You can see in the transformed left scene above, it actually lose some parts on the left side. I tried to move the scene around but it just did not display a full picture. I think the full picture actually exist, and the transformation itself is correct, because the transformed left scene's control points (you can see the wire) has been aligned with the middle scene.

This method can not only be useful in stitching, but can also extract object image from a scene image. For example, the last year homework, extracting the Trojan logo from the playground image. And below is another example provided by OpenCV.



Problem 3 Morphological processing

Motivation

Procedure

a. First we binarize the image. Then we scan the image using two stage mask. First stage is to select candidate to erase, and second is to confirm. The most tiredsome part is hard coding the masks.

b. I use the method of 'counting island' (similar as Leetcode) to count defects as well as count their sizes. For fixing the defect, I apply closing algorithm.

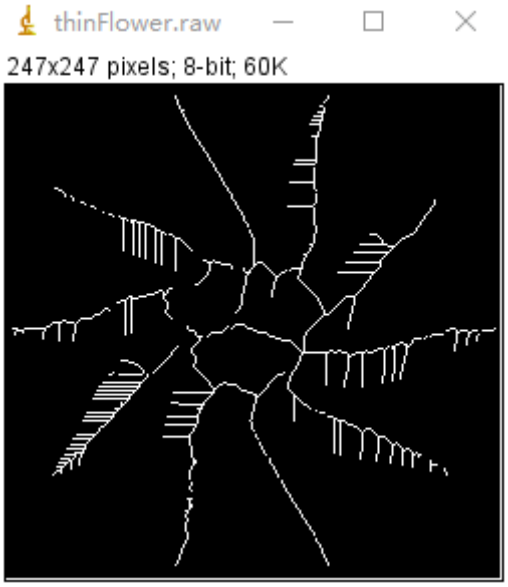
c. First change the image to grey scale by averaging the three channel. Next set a fixed threshold to binarize the picture. Then run closing algorithm from opencv to remove the defects around each beans. Finally, to just the size, I will still use the island counting algorithm as in part b.

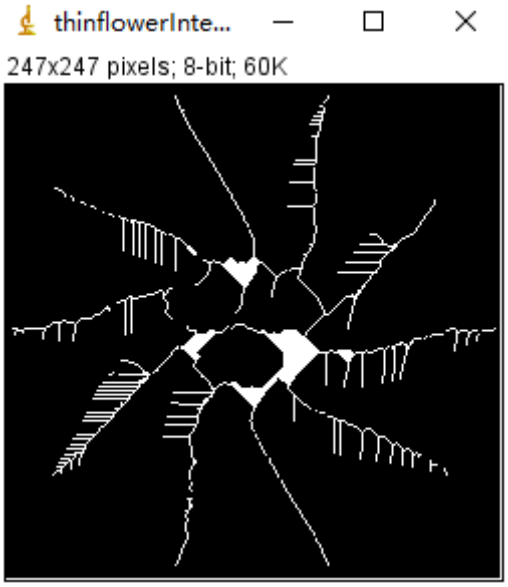
Result

g++ -o thinning thinning.cpp

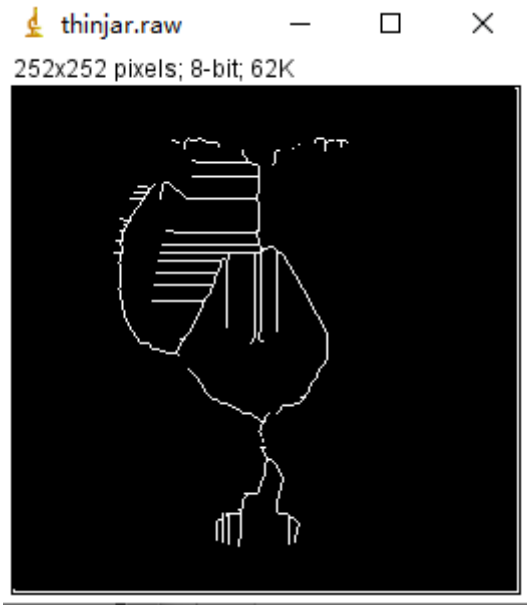
thinning raw_images/flower.raw Output/thinFlower.raw 247

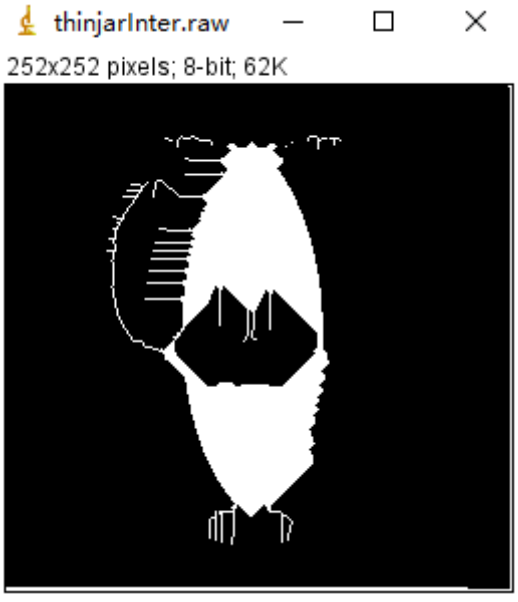
The first image is final output, and second is intermediate output at 20th iteration.



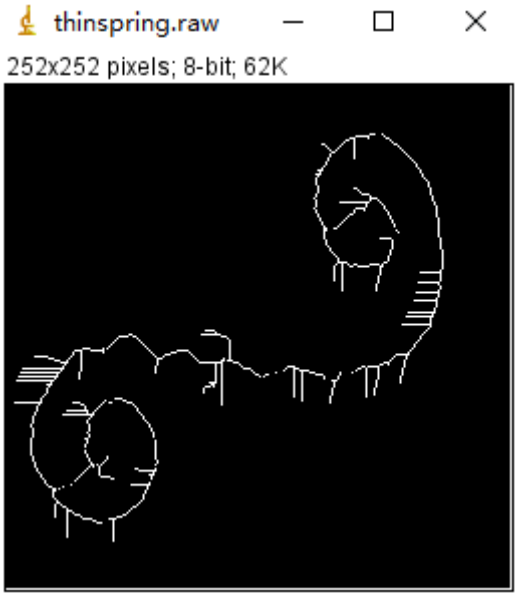


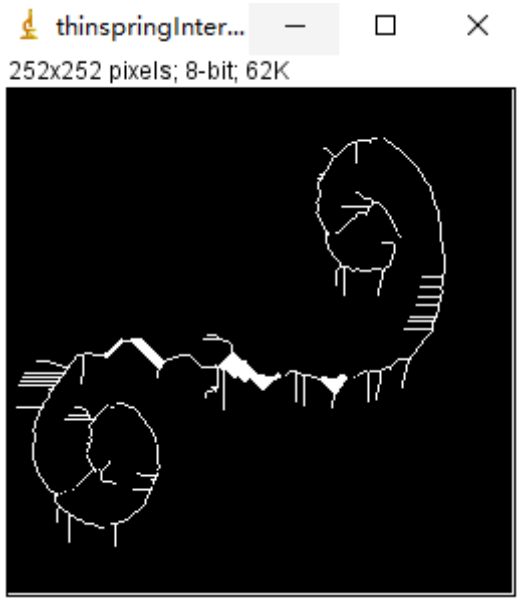
thinning raw_images/jar.raw Output/thinjar.raw 252





thinning raw_images/spring.raw Output/thinspring.raw 252





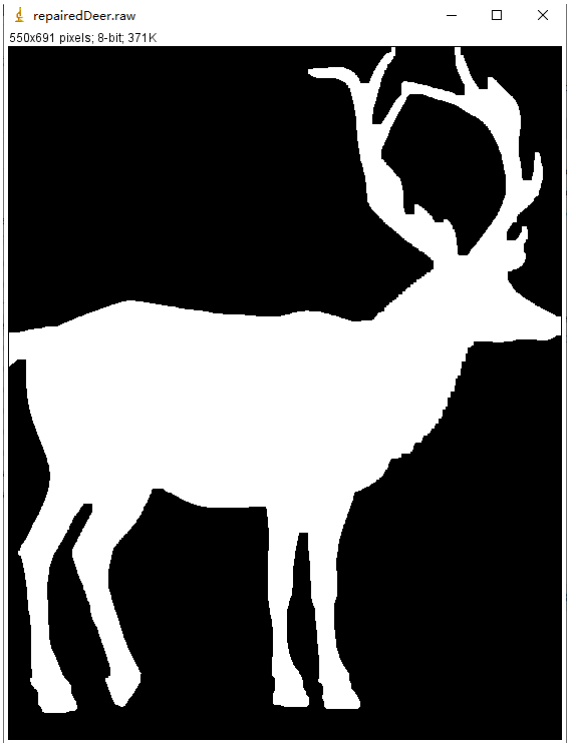
b.

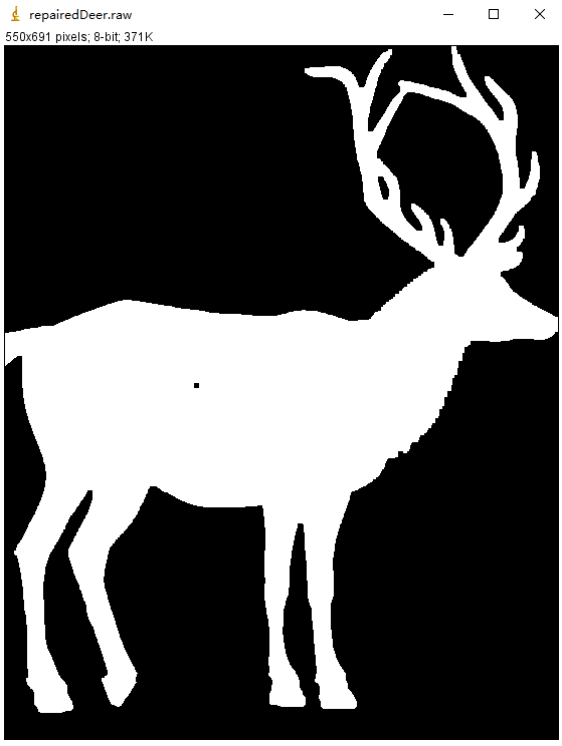
g++ -o defect defect.cpp

defect raw_images/deer.raw

There're totally 6 defects. Five of them are with size of 1, and one of them has size of 40 pixels.

I repaired the deer in two ways





c.

segmentation mask

There're totally five beans. Their size are 2768, 2814, 1042, 3820, and 779 pixels respectively. Therefore, bean 4 > bean 2 > bean 1 > bean 3 > bean 5.

Discussion

a. It becomes thin, and is in the shape of the original pattern, but there's still some problem. First is the skeleton-like shape. I'm sure I only use masks from thinning part, but it just happens. Maybe there's typo, or the dark area inside the original shape lead to unexpected result. Second is some lines break.

b. The defects are found perfectly, but it's hard to remove them without damage other features. There's a hole on the horn, and by adjusting closing parameters, it will either erase this hole, or leave the big defect in the body unremoved. I think the problem will still exist even with other problem, when the defect size is close to non-defect object.

c. This part is quite straight forward. The thresholding during binarization almost complete all the task. And closing successfully removes all the surrounding defects. One debatable point is the white area on bean 2. Direct thresholding leaves there a hole, while closing will cut it off.