Chenrui's Summary of Image Processing Project

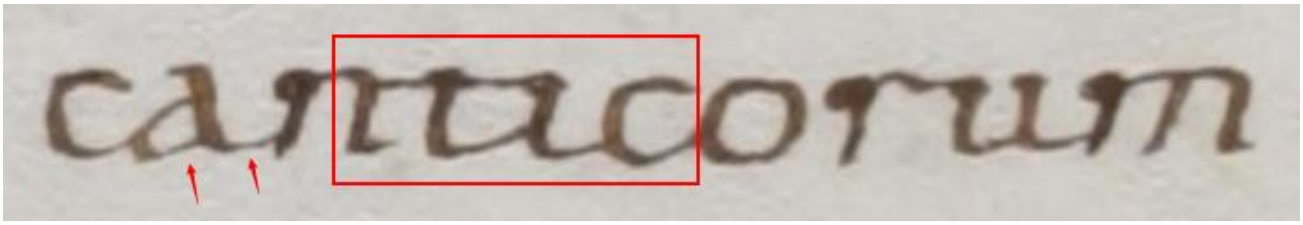

Figure 1. The arrowed portion indicates that the ink thickness inside the font is the same as that in the practiced portion; the character in the box is illegible: it can be "ntlc", or "nuc".

As a result of the preprocessing work done by Qinglin, we have obtained very good quality, clear and usable page images. However, the question of how to sample them to get a dictionary of letter pictures is a problem that needs to be determined. We had done some text segmentation tasks in Chinese or Japanese as undergraduates, but this time Latin was different - it included a large number of hyphenated fonts. There was no way to segment it directly as we had done before.

We considered the possibility of using erosion to "break" the writing in the hyphenated script, but failed - because the width of the ink in the hyphenated script is the same as the width of the letters themselves, and when the writing breaks, so do the letters, and so on, as shown in Figure 1. The inability to use corrosion is not the only problem - we don't know Latin, and even if we could successfully separate the ligatures, we wouldn't know which shapes corresponded to which letters, as shown in Figure 1. So we have to do it another way.

In the process of thinking about this, I remembered that I had implemented the helper code for a game (https://github.com/Sosekie/PocketDowntown) in my spare time this semester. In this project, I used cv2.matchTemplate and the specified images to implement the auto-finding cab logo and the goods-in logo, thus realizing the auto-driving function and the auto-feeding function. "Yeah, why can't I manually isolate individual Latin characters first and then use them as templates for use after scaling, rotating, morphing and other transformations?"

As it turns out, my idea works. I designed the following judgment logic: if the template matches a similar graphic, and the similarity is greater than threshold, then I copy the matched graphic down to the size of the template, add it to the matches array, and set the original graphic area to empty (here it's (0,0,0), which is black). Continue this process until the template no longer finds any graphics with a similarity greater than threshold. In this way, we can split the hyphenated characters with a null operation. Here's a little trick: we can match m before n, as well as match i after all characters except i have been matched. This is because m is shaped like it includes n, and the Latin i doesn't have that dot above it, so basically a lot of characters include the shape of i. And there's another advantage to using template matching: if we want to continue to add variety, we can continue to select characters from the page image that don't match as new templates, and continue to match until we're satisfied. In the process we get a large number of characters that need to be cleaned and numbered so that we can select them when we generate them. The task of filtering the characters part is done by Qinglin.

Of course, there are some characters missing from the given image, such as almost all uppercase characters as well as numbers. I used some of the fonts in carolus.ttf as replacements and scaled and distorted them during sentence generation to ensure their diversity.

Next is the sentence generation part of the task. This part consists of many elements, which I have divided into the following sections:
1. read the sentence entered by the user and find the corresponding character picture;
2. calculating the line height and line width of the sentence, as well as the spacing between letters and words;
3. process the character images one by one, deform them, and offset them horizontally and vertically;
4. generating background images and writing the processed images into the background image.

In the first step, we will encounter lowercase letters, uppercase letters, numbers, punctuation marks, in addition to special characters represented by special symbols (such as christi, prae, etc.). When encountering these characters, instead of reading them individually, we should save a consecutive batch of characters for character retrieval. Here, I have adopted the method of wrapping the special characters with "(" and ")". When reading "(", create a new empty string and add the current character; after the loop, if you do not read ")", continue to add the current character until you read ")". Then we can use this string, like (christi), to retrieve the (christi).png picture.

In the second step, the length of the line is the sum of the length of each letter, the spacing between letters, and the spacing between words. And the point about the spacing between letters is a point that needs to be carefully considered - not all letters are fixedly spaced, for example, almost any letter after the letter f will be closer to f. Qinglin and I counted the approximate distances between two by two letters, and set special horizontal offsets for some special characters. For example, when f is read, its width is the width of f minus the offset, and its current position is:

```
current_width += image.size[0] + right_offset + letter_spaces[letter_space_index]
```

Which means that the next letter will be shifted to the left when it is written. In addition, the intervals between letters and the intervals between words are randomly generated within a certain range, and I use an array to record the length of the randomly generated intervals and read them one by one when generating the sentence image and update the current position so that I can adjust where to write the characters.

The third and fourth steps continue to call calculate_bottom_offset(char) and calculate_right_offset(char) to get the vertical and horizontal offsets of the letters, record the current position, blur and write the letter images to the black background image. I have given variables such as line height, offset, and current write position here to make it easier for students doing Task 3 to get information such as baseline and word position.

In general, this project has improved my ability, especially the ability to deal with special problems - a lot of solutions are very feasible verbally, but in reality, you still have to learn to be flexible and adaptable. In addition, I hope that the code I implemented can become a reusable function, giving more people an idea of how to implement it, not just for the purpose of fulfilling assignments - work that cannot be reused by others is meaningless to me. So I tried to think of a simple, easy-to-implement way to make it easier for people who need it to run through our code without hindrance and implement what they need.