Chenrui's Summary of Image Processing Project

Generating books with handwritten text or special font styles is very interesting to me, not only because it has a lot of optional creative implementation details, but also because this project can be reused by others who have a need for it - anyone can upload their own manuscripts, and then generate the corresponding personal style books through this project. I was working towards this goal, but unfortunately didn't have enough time to try it out on larger datasets (my sister has this need - to organize handwritten notes on Notability into a single document).

My understanding of this project is this: the project wants to isolate the background of the book page as well as the text from an existing book page, and process them separately to make them more varied and full of detail. Then using the relative position of the original text in the book page, the newly generated text will be put into the newly generated book page, so as to realize the function of inputting text to generate corresponding book pictures in batch. These generated images can continue to be used to train downstream neural networks for tasks such as OCR and Vision-Language Multi-model.

Task 1 is to isolate the background image, de-noise it, deform it, add stains and other details to make it more realistic. Being able to provide realistic and clear background images in large quantities is the goal of task one. Friends in the group have asked me how the color fills in after removing the text, in fact, choosing the color of the blank area near the text to fill in, the effect will be very good. I also thought about the problem of chromatic aberration, I think if you really want to take chromatic aberration into account, you can first balance the background color of the whole picture, and then assume a direction of light, add a gradual change in the shadow, so the effect will be better.

Task 2 was done by me and Qinglin, the details will be described later.

Task 3 is about how to add ink to the text. I think there is a little complicated with dividing the task into grayscale images of the text first and then adding ink to it - why not just keep the RGB text images with transparent backgrounds and ink when sampling the text? I think about it this way because there is actually a pattern to how each text falls, for example, the letter n, which starts out heavier in the upper left corner and lighter in the center. And many fonts are sampled in Task 2, which also ensures the diversity of ink strokes at the same time. In the generation can also be shaped in the text at the same time, the ink color, dark and light area size and other factors, such as small-scale randomization, so that the results will be very good.

For task 4, about generating the full image, I think there are still problems with line alignment and morphing. If I were to do it, I'd try to flatten the page as much as possible, and then record the position of each line, so that it can be morphed back to something else after the text is added. As for how to "make the page as flat as possible", I think we can manually mark the four sides of the page (because we used this method to process the text, and the result is very good).

This is my understanding of the task and some ideas. I will continue my part of the assignment with the work of Qinglin.

As a result of the reprocessing 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. 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. 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. I used cv2.matchTemplate 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. 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. 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 classmates 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.