Computer Version Assignment 2

Yunming Hui(5334977)
Graduate School of Natural Sciences
Utrecht University

Yuqi Liu(5383986)
Graduate School of Natural Sciences
Utrecht University

The assignment is based on the Windows operating system, OpenCV 4.2.0 and Visual Studio 2022, with the sample code framework provided as the most basic.

I. Basic tasks

In this section, we will describe in detail how we completed the basic tasks and show the results obtained.

A. Intrinsics and extrinsics calibration

The images used to calibrate the intrinsics were captured from each camera's "intrinsics.avi" file. The goal is to capture as many images as possible with different relative positions and angles of the calibration plate to the camera.

We tried two methods to capture images. The first method uses SSIM (structural similarity index) as a measure of the similarity of two images. The video is read frame by frame. The first image is captured by default. When the similarity between current frame and the last acquired image is less than 0.9, the current image is considered to be an image that should be acquired. The results of this method are not ideal. When the calibration plate is tilted at a small angle relative to the camera, a slight movement results in a large variability, but when the tilt angle is large, a larger change is required to produce a recognized variability. This method is implemented by function SSIM().

In the second method, we consider that every 1 second the person in the video causes a large change in the position and angle of the calibration plate relative to the camera. The rate of the video is 50 frames per second, so we capture the image every 50 frames. Compared with the first method, this method is more effective and can obtain more images with different positions and angles of the calibration plate with respect to the camera. This method is implemented by function fixedInterval().

Therefore, we used the second method. After acquiring all the images, we use the same way and code as in Assignment1 to calibrate intrinsics and extrinsics.

* The code of this part is not merged into the sample code framework and can be found in the submitted project named "extra".

B. Background subtraction

In "video.avi", the people in the frame hardly move, making the difference between frames insignificant. Therefore, we chose to perform background subtraction by analyzing the difference between the frames and the reference image.

1) Reference image: To obtain a reference image, we calculated the mean and variance of each pixel in each channel for the first 100 frames, and then regenerated an image using the mean, which is the reference image. This part is implemented by function getbgimg() in the submitted project named "extra".

2) Implementation: The implementation of background subtraction is performed based on the sample code framework. In the process of whether a pixel belongs to the foreground or the background, we implement automatic setting of the threshold value.

We judge the goodness of the segmentation based on the assumption that the ideal segmentation should be coherent. For the resulting segmented image, we use the function findContours provided by OpenCV to detect the number of contours in the image (only external contours are calculated). The segmented image is then eroded and dilated, and the number of contours in the processed image is detected again. The absolute value of the difference between the number of contours before and after processing is used as an indicator of good segmentation, the lower the value the better the segmentation.

It is worth noting that we found that if only a very small number of pixels in the segmented result image are identified as foreground , like figure 1, good results will also be obtained. In order to avoid this situation, we put a limit on the mean value of the result image, and we set that the mean value of the result image should be between 8.5 and 14. This part is implemented by the function diff_imgprocessed().

Fig. 1: Incorrectly evaluated segmentation result image

Since the frame hardly changes in the video, we use the first frame of each camera to find the best parameters for each of the four cameras. We first search for the three parameters in steps of 15. After finding the rough parameters, we search for parameters in the range of five before and after them in steps of 1 to find the optimal parameter value (which may be locally optimal). This part is implemented by the function find_opt.

C. Volume reconstruction

The main function for Volume reconstruction is the "update" method in the "Reconstructor.cpp" file. The main idea is that all voxels are traversed. If a voxel appears in the foreground of all cameras, then the value of that voxel in the look-up table is "on".
In the original code, instead of using a look up table to record the values of voxels, the voxels with the value "on" are stored directly in a vector. This saves memory on look up tables, but limits the implementation of subsequent operations such as de-noising. We have therefore added the "look up table" as a private member of the "Reconstructor" class, which can be manipulated via the get/set methods.
In addition to adding the 'look up table', we have improved the parallel computation on the basis of the source code, added noise reduction to the voxels, optimised the construction of the voxel model and implemented the colouring of the voxels. The noise reduction is described in the following section and the rest is described in the next section. The results of the voxel reconstruction are presented in section 3.

1) Noise reduction in voxel reconstruction: We use "Erosion" and "Dilation" in 3D. The idea is to set a threshold "x" and if a voxel has more than "x" voxels adjacent to it with the value "on", then that voxel is also set to "(a voxel has a total of 6 neighbouring voxels). Accordingly, if most of the surrounding voxels are "off", the voxel is also set to "off".
This is achieved by iterating through the look up table once after the values of all voxels have been determined based on the foreground image, and changing the values of the voxels that are considered noisy.
This code is in the "update" method. It is commented out because the noise reduction process requires another traversal of the look up table, so the time overhead is significant and the code is slowed down.

II. Choice tasks

A. Extrinsics calibration acceleration

According to the OpenCV interpretation[1] of the function solvePnP, a minimum of four points are needed to find the optimal extrinsics. The first three points are used to estimate all solutions to the problem and the last one is used to find the best solution. According to our experiments, the accuracy of the external reference obtained by using four points is also sufficient, and in the program we are now using this way. That is, only the points 1, 8, 13, 20 in the figure 2 need to be annotated.

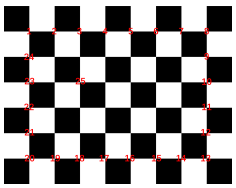

Fig. 2: Checkerboard image

[1]https://docs.opencv.org/4.2.0/d9/d0c/group__calib3d.html

If all the points are used, annotation can be speed up by annotating only all the peripheral points(Point 1-24 in figure 2). The internal points can be inferred from the points being annotated. A line should still be a line after projection, and the coordinates of a point can be determined by two intersecting lines. Therefore, all points inside can be determined by the already annotated points. For example, the coordinates of point 25 can be obtained from the line determined by points 3 and 28 and the line determined by 10 and 23.

B. Voxel modeling optimization

The idea is to first determine which part of the two foreground images has changed, then calculate the value of only the voxel corresponding to the changed part, and change the look up tabl accordingly.
Firstly, the "camera" class in the source code only has the member "m_foreground_image" which holds the current foreground image. So we added a member to the "camera" class that holds the last foreground image, named "m_pre_foreground_image". And when the code wants to change the current foreground image, the current foreground image will be saved to "m_pre_foreground_image", to achieve the need to record the previous foreground image.
We then use OpenCV's "absdiff" method to calculate the changed parts of the two foreground images. Since the voxel class holds the coordinates of the voxel in 2D, we iterate through all the voxels to see if the coordinates are part of the changed part, and if they are, we recalculate the look up table value, and if they are not, we do not. This code is also implemented in the "update" method.
It is worth noting that the optimization does not actually improve code execution speed significantly. This is because in the original algorithm, we traverse the voxels to update the look up table values. After the optimization we still have to traverse all the voxels on one side to determine if they are part of the changed section, and we have to look at all four cameras. This optimization may be more effective in an algorithm that traverses by view.

C. Voxel model coloring

We also try to colour all voxels with the value "on". One is how to determine whether a point is the surface of an object (which voxel is visible in 3D for a point in a 2D image, taking into account the depth order). The other problem is what is the colour of this point.
The basic idea of determining the visibility of voxels is that for a 2D point on a view, first find all voxels projected to this point and calculate the Euclidean distance of these points from the camera corresponding to that view. The voxel with the smallest distance is the visible voxel for this 2D point (the voxel that should be coloured). The position of the camera is recorded in the "m_camera_location" of the "camera" class, the 3D position of each voxel is stored by its "x", "y", "z" and the 2D coordinates of each voxel projected onto the corresponding camera are also recorded in the "camera_projection" of the voxel class. So the above idea can be implemented.

The second problem is to determine the colour of these surface voxels. For voxels that are only visible in one view, the colour of the voxel is the colour of the corresponding 2D point in that view. For voxels visible in multiple views, the colour is the weighted average of the colours in these views. The colour of the voxels is determined by the "m_frame" member of the "camera" class of the four cameras.
The code to implement this is in the "color" method of "Rconstructor.cpp".

D. Parallel computing for look up table

We use "OpenMp" to implement parallel computation of look up tables. Since we update the index table by traversing the voxels, we use parallel computation for the loop that traverses the voxels. The code is described in the "update" and "colour" methods.

III. Results

A. Intrinsics and extrinsics calibration

The number of images obtained, number of valid images after optimization, and calibration error are shown in Table I.

TABLE I: **Intrinsics calibration results**

Camera ID	Number of images acquired	Number of valid images	RMS re-projection error
1	75	40	0.313
2	85	58	0.255
3	74	33	0.196
4	124	65	0.262

Figure 3 shows the calibration results of four cameras. Intrinsics and extrinsics of each camera can be found in the submitted "config.xml" files.

Fig. 3: Calibration result images of 4 cameras

B. Background subtraction

Figure 4 shows the reference images of the four cameras that we obtained by calculating the mean values of each pixel.

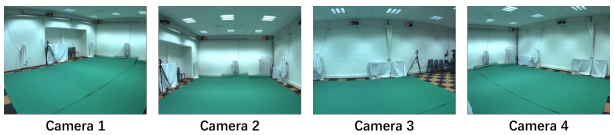

Fig. 4: Reference images of 4 cameras

Figure 5 shows a example result of foreground extraction obtained by automatically setting the threshold value.

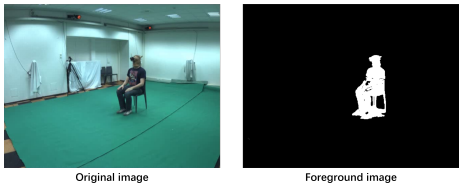

Fig. 5: Background subtraction example

C. Volume reconstruction

Figure 6 shows the volume reconstruction and coloring results. What can be seen is that there are no shadows after the volume reconstruction. A link to the volume reconstruction demonstration video can then be found in the supplement material.

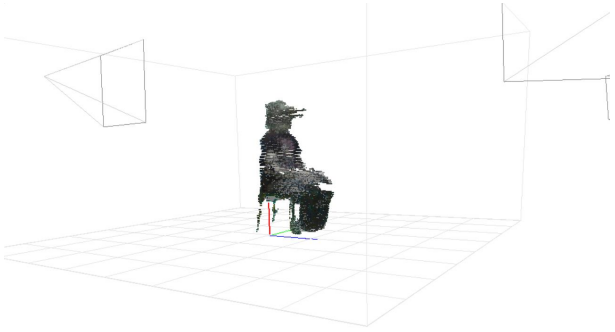

Fig. 6: The volume reconstruction and coloring results

IV. Supplement material

Result Video: https://youtu.be/fQjXRBcqCLg