AINT308 - OpenCV Assignment 2 2022

Student No. 10618407
School of Engineering,
Computing and Mathematics
University of Plymouth
Plymouth, Devon

***Abstract*—Machine vision is a mature technology that is becoming more prevalent within modern engineering practises. It is being utilised more in the rapidly evolving fields of autonomy and automation. This report outlines some of the functionalities of a popular C/C++ based computer visions library *OpenCV*. The Assignment has been split into two tasks; Task 4 and Task 5. The first task, Task 4, is Disparity Mapping using a pair of stereo cameras to be able to judge distances to objects. The second task, Task 5, was Self-driving Car Lane Detection from the dashcam footage taken from a car.**

Keywords:

Computer Vision, OpenCV, Object Detection, C++, Lane Detection

I. Task 4: Disparity Mapping

A. Introduction

The first task was to perform disparity mapping to evaluate the distance of a given object in a frame to allow a robot to navigate it's environment. Disparity Mapping allows the robot to build up a 3D view of it's environment

B. Solution

The solution to this task was split up into multiple parts. Firstly, using the calibration images to calibrate the cameras. Secondly, using the calibration result to correct the stereo distance targets. Next, Combining the left and right images into a disparity map, with brightness indicating the distance from the camera. Penultimately, labelling the distance targets with their known distance to the camera. Use this to plot disparity against distance and write a formula relating distance to disparity. Finally, using the formula to calculate the distance for each of the unknown targets.

1) Camera Calibration and Stereo Image Correction

Stereo camera needs to be calibrated to one another before they can be used for stereo vision. Corrections need to be made for both physical misalignments (extrinsic error), and lens distortion(intrinsic error) [1]. This was done using the SteroCalibration program provided by *OpenCV* [2].

The program firstly reads in image pairs (taken from the left and right camera) of a checkerboard pattern target placed in different locations and orientations. This allows the program to create a list of intrinsics and extrinsic parameters to allow for these effects in the program. The checkerboard is of known size so the program can calibrate what it sees against what it is expecting. This known as the Bouquet stereo image calibration [3].

2) Disparity Mapping

The first stage of disparity mapping was to remap the images to correct for the positional/lens distortion, as mentioned above [4]. The two images were then combined into a 16 bit stereo block matcher [5]. The code for doing this can be found in Fig 1.

```
//Load images from file
Mat Left =imread("../Task4/Distance Targets/left" +to_string(ImageDistance)+"cm.jpg");
Mat Right=imread("../Task4/Distance Targets/right"+to_string(ImageDistance)+"cm.jpg");
cout<<"Loaded image: "<<ImageDistance<<endl;
//Distort image to correct for lens/positional distortion
remap(Left, Left, map11, map12, INTER_LINEAR);
remap(Right, Right, map21, map22, INTER_LINEAR);

//Match left and right images to create disparity image
Mat disp16bit, disp8bit;
// compute 16-bit greyscalse image with the stereo block matcher
sgbm->compute(Left, Right, disp16bit);
disp16bit.convertTo(disp8bit, CV_8U, 255/(numberOfDisparities*16.));
// Convert disparity map to an 8-bit greyscale image so it can be displayed
// (Only for imshow, do not use for disparity calculations)
```

Fig. 1: Remapping Left and Right Images into one image

Once this was done, a 16 bit disparity map of the two images has been created. To measure the disparity, a rectangular region of interest was created, as seen in fig 2. This was created to make sure that only the disparity of the measured object was used. Whilst in the actual test the disparity map used was the 16-bit version, it does not get outputted correctly, so the 8-bit version is used for demonstration purposes.

```
int x = 355;
int y = 210;
int rectangleWidth = 50;
int rectangleHeight = 50;

Rect rect(x, y, rectangleWidth, rectangleHeight);
rectangle(disp8bit, rect, Scalar(255, 0, 0));
```

Fig. 2: Rectangle Drawn On Disparity Map to Indicate Search Area

Fig I-B2 is the main code to check the disparity for this task. The code is searching through all of the pixels in a given area (as constructed in fig 2). If the disparity value is larger than a certain value, it is added to the list.

```
// runs through all the rows in the image
for (int i = x; i < x + rectangleWidth; i++) {
    // runs through all the columns in the image
    for (int j = y; j < y + rectangleHeight; j++) {
        // stores the RBG values in the PixelValue vector
        int PixelValue = (int)disp16bit.at<ushort>(j,i);
        if (PixelValue < 65000){
            OutputValue += PixelValue;
        }
    }
}
```

Fig. 3: Disparity Mapping Code for evaluating the disparity in a image

At the end of the run, the average value of the disparity is found, this is shown in fig 5. These values were saved to a CSV files for evaluation after execution.

3) Labelling Known Distances and Creating Disparity Function

From the values of measured distance and disparity shown in table I. The graph in fig 4 was created to map the disparity over distance.

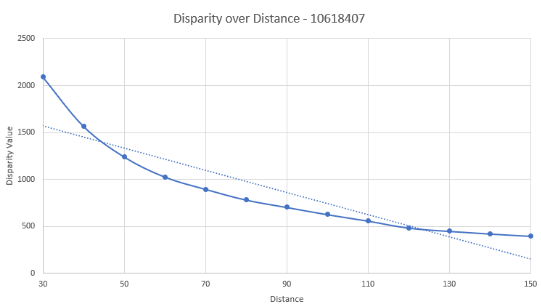

Fig. 4: Graph of Disparity over Distance

Fig 4 has been made using the data from table I. The value for Bf was calculated by rearranging the disparity equation 1 to equation 2. To do this, the average of the $Distance \cdot Disparity$ was taken, as shown in equation 3.

$$Disparity = \frac{B \cdot f}{Distance} \quad (1)$$

$$B \cdot f = Disparity \cdot Distance \quad (2)$$

$$\frac{Distance \cdot Disparity}{TotalNumberofValues} \quad (3)$$

Once the value of Bf had been found it was used to calculate the the distances for the known targets, as of test of the previous measurements accuracy this can be seen in equation 2. The code for calculating the distance and outputting the result can be seen in fig 5.

TABLE I: Distance and Disparity Table

Measured Distance	Disparity Value	Distance x Disparity
30	2089.72	62691.6
40	1566	62640
50	1240.27	62013.5
60	1026.87	61612.2
70	896.546	62758.22
80	782.35	62588
90	703.714	63334.26
100	626.804	62680.4
110	558.514	61436.54
120	483.93	58071.6
130	449.154	58390.02
140	421.226	58971.64
150	396.406	59460.9

TABLE II: Distance Calculated from Disparity Mapping with Known Distances

Measured Distance	Calculated Distance	% Difference
30	29.32	2.25
40	39.13	2.17
50	49.41	1.18
60	59.68	0.54
70	68.35	2.36
80	78.35	2.10
90	87.08	3.24
100	97.77	2.23
110	109.72	0.25
120	126.63	-5.53
130	136.43	-4.95
140	145.48	-3.91
150	154.59	-3.06

```
OutputValue = (OutputValue/(rectangleHeight * rectangleWidth));

cout << "Output Value : " << OutputValue << endl;

// using calulated BF value
double BF = 61280;

// using the disparity equation
double calcDistance = BF / OutputValue;

cout << "Calculated Distance : " << calcDistance << endl;

DataFile << OutputValue << endl;
```

Fig. 5: Code to output the distance measured to the object

From the values of measured distance and disparity shown in table II, and the accuracy of the prediction was also recorded. As the distance value for the testing images was already known and the disparity value had just been calculated, it was easy to test the output of the equation for reliability and accuracy.

4) Using Disparity Formula to calculate distances for unknown targets

To find the distance for the unknown targets, the disparity value was firstly calculated using the same method as described above. The disparity value, along with the value previously calculated using equation 2, can be used to calculated the distance of the object in the images.

The method for evaluating the distances of the objects in the images was very similar to the function used in, within the Disparity Mapping section. The main differences between the two codes are, the source of the files and the terminal output of the program. The files used in calculating the distance were images with an unknown distance to the object in the image. The output in the terminal was almost identical but the known distance of the object was not stated - as it was unknown.

It is unclear how close the the estimates of distance were to the actual values but given the accuracy on the known images (within 5.53%), it would be safe to assume that these values could be considered accurate within a 6% margin.

C. Further Improvements

Although this method was effective there are still ways in which it could be improved. The two biggest flaws with this method was how the calibration was conducted.

The distance measurement was conducted using a a box held aloft over a ruler. Whilst this did provide adequate calibration data for the disparity testing, it could have been conducted in a manor that would have made the measurements increasingly more precise.

A clearer background would have been beneficial for disparity matching. The utilised background had lots of moving objects that were not consistent between images. This required a region of interest to be used for the disparity mapping. This resulted in a smaller area for the disparity to be measured across. Ideally a plain background would have been used to allow a larger amount of the frame to be mapped. This would increase the area for the disparity mapping thus reducing the effect of outliers within the region of interest.

These were the two biggest areas of uncertainty in this method and thus would gain the best yield if improved.

D. Conclusion

Overall this task was hard to judge on the unknown data, as the results were not known and thus hard to tell the accuracy of the finalised results. Despite this, the known measurement disparity mapping was extremely successful, with a maximum error of 5.53% and an average error of only 2.9%. Although the methodology could have been improved by making the calibration data collection more precise and using a clearer background, the distances of measurement used (30cm-150cm) would negate the effects of measurement over that distance.

II. Task 5: Self-driving Car Lane Detection

A. Introduction

B. Solution

C. Further Improvements

D. Conclusion

References

[1] A. Y. Eser. (2020, Sep) The depth 1: Stereo calibration and rectification. [Online]. Available: https://python.plainenglish.io/the-depth-i-stereo-calibration-and-rectification-24da7b0fb1e0

[2] *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008.

[3] J.-y. Bouguet and P. Perona, "Camera calibration from points and lines in dual-space geometry," 10 1998.

[4] OpenCV. (2022, Apr) Remapping. [Online]. Available: https://docs.opencv.org/3.4/d1/da0/tutorial_remap.html

[5] C. McCormick. (2014, Jan) Stereo vision tutorial - part 1. [Online]. Available: http://mccormickml.com/2014/01/10/stereo-vision-tutorial-part-i/

The code can be found on GitHub here!

APPENDIX