ASSIGNMENT THREE

Robotics and Automation 282.762

Caelan Vandermey, 14087877

Table of Contents

Table of Figures

1. Introduction

This report details the code and output for assignment three for the Robotics and Automation (282.762) course at Massey University where a program that transforms the pixels of one letter into the location of an empty space in the image. Section two explains the code and section three shows and explains the output.

2. Program Explanation

The program for this project can be split into three major parts; the Global Setup (Section 2.1), the Main Function (Section 2.2) and the Transformation Finding Function (Section 2.3).

2.1. Global Setup

The global setup code for the program is shown in Figure 1 and has four small parts. The first is the includes which enable the use of library files. This program uses the OpenCV, cmath and Windows libraries (lines one, two and three respectively) for image processing, mathematical functions and a waiting function.

Lines five and six declare that the “std” and “cv” namespaces are being used in the program. Line eight globally declares pi for use in conversions from degrees to radians. Line 10 is the function prototype for the “transform” function that will be explained in Section 2.3.

```
1  #include <opencv2\opencv.hpp>
2  #include <cmath>
3
4  using namespace std;
5  using namespace cv;
6
7  long double pi = 3.14159265358979323846264338327795;
8
9  void transform(Mat image, int num);
```

Figure 1 Global Setup Code

2.2. Main Function

The main function is responsible for reading the images into the program and for calling the “transform” function to find the transformation matrix and this is shown in Figure 2. Line 12 creates the matrices that will hold the five images being read into the program. Lines 14 to 27 read the image into the program and convert the image from the RGB (Red Green Blue) colour space to the HSV (Hue Saturation Value). Lines 29 to 33 call the “transform” function, passing the HSV image and the number of the image. Line 35 uses the batch command “pause”. This makes the command window wait until any key is pressed, which allows the user to read the output of the program.

```
11 int main() {
12     Mat image1, image2, image3, image4, image5;
13
14     image1 = imread("Image 1.jpg", CV_LOAD_IMAGE_COLOR);
15     cvtColor(image1, image1, COLOR_BGR2HSV);
16
17     image2 = imread("Image 2.jpg", CV_LOAD_IMAGE_COLOR);
18     cvtColor(image2, image2, COLOR_BGR2HSV);
19
20     image3 = imread("Image 3.jpg", CV_LOAD_IMAGE_COLOR);
21     cvtColor(image3, image3, COLOR_BGR2HSV);
22
23     image4 = imread("Image 4.jpg", CV_LOAD_IMAGE_COLOR);
24     cvtColor(image4, image4, COLOR_BGR2HSV);
25
26     image5 = imread("Image 5.jpg", CV_LOAD_IMAGE_COLOR);
27     cvtColor(image5, image5, COLOR_BGR2HSV);
28
29     transform(image1, 1);
30     transform(image2, 2);
31     transform(image3, 3);
32     transform(image4, 4);
33     transform(image5, 5);
34
35     system("pause");
36     return 0;
37 }
```

Figure 2 Main Fuction Code

2.3. Transformation Finding Function

The “transform” can be broken down into eight parts; the Setup Code, the Contour Finding, the Minimum Bounding Circle, the Finding the Correct Circles, the Translational Change, the Taking Sections of the Image, the Finding Angle Change and the Matrix Output. These eight sections are detailed below.

2.3.1. Setup Code

The setup code for the “transform” function creates all the matrices for storing the image (lines 40 and 41 in Figure 3). Lines 43 and 44 (Figure 3) create vector arrays to hold the contour and hierarchy data that the “findContours” function will require for outputs. Line 45 creates a point to use as a centre for later transformations. Lines 46 and 47 are created to act as sizes for the vector arrays made in Figure 5. Lines 49 to 52 are made to store and hold values for use later. Line 53 and 54 hold an index for the best bounding circles. Line 55 holds the value that will be used to adjust the widths of the matrix output.

```
39  void transform (Mat image, int num){
40      Mat imageObj, imageTar;
41      Mat TarImg, ObjImg, ObjImgCheck, TarImgCheck;
42
43      vector<vector<Point>> conObj, conTar;
44      vector<Vec4i> hieObj, hieTar;
45      Point cenObj;
46      size_t conObjSize;
47      size_t conTarSize;
48
49      double white = 0;
50      double angle = 0;
51      float deltaY = 0;
52      float deltaX = 0;
53      size_t maxRObj = 0;
54      size_t closestRTar = 0;
55      int adj = 13;
```

Figure 3 Transformation Setup Code

2.3.2. Contour Finding

To find the contours, the image is masked (using the “inRange” function) to only show the correct colours, splitting the one image into two. These follow a notation of “Obj” (object or the blue letter) and “Tar” (target or the white letter(s)). Both images are passed through the “findContours” function to get the contours for the blue and white letters. These steps are shown in Figure 4.

```
67      inRange(image, Scalar(0, 0, 0), Scalar(179, 6, 255), imageTar);
68      inRange(image, Scalar(100, 35, 95), Scalar(126, 255, 255), imageObj);
69
70      findContours(imageObj, conObj, hieObj, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0));
71      findContours(imageTar, conTar, hieTar, CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE, Point(0, 0));
```

Figure 4 Contour Finding Code

2.3.3. Minimum Bounding Circle

The sizes of the objects are now calculated so that the vector arrays can be created with enough space to fill with no chances of overflow, this occurs on lines 73 and 74 in Figure 5. Lines 76 to 81 create the vectors to hold the contour polygons and circle centre and radii. Lines 83 to 86 and 88 to 91 have the same function, to determine the minimum bounding circles for the contours found in the image using "approxPolyDP" and "minEnclosingCircle", the only difference is that one finds the object's circles and the other finds the target circles (variables are named with the "Obj" and "Tar" notation).

```
73      conObjSize = conObj.size();
74      conTarSize = conTar.size();
75
76      vector<vector<Point>> conPolyObj(conObjSize);
77      vector<Point2f> cObj(conObjSize);
78      vector<float> rObj(conObjSize);
79      vector<vector<Point>> conPolyTar(conTarSize);
80      vector<Point2f> cTar(conTarSize);
81      vector<float> rTar(conTarSize);
82
83      for (size_t i = 0; i < conObjSize; i++) {
84          approxPolyDP(Mat(conObj[i]), conPolyObj[i], 3, true);
85          minEnclosingCircle((Mat)conPolyObj[i], cObj[i], rObj[i]);
86      }
87
88      for (size_t i = 0; i < conTarSize; i++) {
89          approxPolyDP(Mat(conTar[i]), conPolyTar[i], 3, true);
90          minEnclosingCircle((Mat)conPolyTar[i], cTar[i], rTar[i]);
91      }
```

Figure 5 Minimum Bounding Circle Finding Code

2.3.4. Finding the Correct Circles

The process of finding the best bounding circles from each group (object and target) is different. The best object bounding circle is the largest circle as there is only one object letter, so a for loop is run to find the largest radius and this is stored (lines 93 to 95 of Figure 6). This check is required as an "A" would have at least two circles, one around the letter and one around the opening in the middle of it (shown in the left of Figure 7). The best target circle is the one with the closest radius to the object circle as a situation like what is shown in Figure 7 may occur. The circle for the "A" is substantially more different than the circle for the "S", meaning that the two letters would not be mixed up if the expected letter was an "A" or a "S". This is achieved by looking for the smallest absolute difference in the radii and is shown in lines 97 to 100 in Figure 6.

```
93      for (size_t i = 0; i < conObjSize; i++) {
94          maxRObj = ((rObj[i] > rObj[maxRObj]) ? i : maxRObj);
95      }
96
97      for (size_t i = 0; i< conTarSize; i++) {
98          closestRTar = ((abs(rObj[maxRObj] - rTar[closestRTar])
99              > abs(rObj[maxRObj] - rTar[i])) ? i : closestRTar);
100     }
```

Figure 6 Best Circle Finding Code

Figure 7 Minimuim Bounding Circles for Target Image 3

2.3.5. Translational Change

Finding the translational change from the object circle to the target circle is as easy as finding the differences in the "x" and "y" components of the centre points. This is shown in Figure 8 and the target values are reduced by the object values to find the difference.

```
102     deltaX = cTar[closestRTar].x - cObj[maxRObj].x;
103     deltaY = cTar[closestRTar].y - cObj[maxRObj].y;
```

Figure 8 Translational Change Code

2.3.6. Taking Sections of the Image

With the best circles selected, the program now takes copies of the masked images that is just large enough to hold the bounding circles. Lines 105 to 108 of Figure 9 take the object circle and lines 109 to 112 take the target circle. The ranges are based on the centre points and are the radius' length away in the "x" and "y" directions, ensuring that the entire letter is taken to the new image.

```
105     ObjImg = imageObj(Range((cObj[maxRObj].y - rObj[maxRObj]),
106         (cObj[maxRObj].y + rObj[maxRObj])),
107         Range((cObj[maxRObj].x - rObj[maxRObj]),
108         (cObj[maxRObj].x + rObj[maxRObj])));
109     TarImg = imageTar(Range((cTar[closestRTar].y - rTar[closestRTar]),
110         (cTar[closestRTar].y + rTar[closestRTar])),
111         Range((cTar[closestRTar].x - rTar[closestRTar]),
112         (cTar[closestRTar].x + rTar[closestRTar])));
```

Figure 9 Image Cropping Code

2.3.7. Finding Angle Change

The angle is found through rotating the object image and comparing the number of uncovered pixels. The angle that results in the least number of covered pixels is the closest angle of rotation. The process that achieves this is shown in Figure 10.

Lines 107 and 108 find the centre of the cropped object image, which is the point that it will be rotated around. Line 110 sets the current lowest count of white pixels to be the number of pixels in the image, this ensures that once the value is compared against, that value will then be the lowest and is not some number that is so small that no angle results in less white pixels. Lines 112 to 127 are a for loop that will run through each angle (degree by degree).

Line 113 sets a local count of the white pixels to zero so that counting can be done from zero again. Line 115 is the line that rotates the image by using the "warpAffine" function. This uses a rotational matrix found by using the "getRotationMatrix2D" function and outputs an image that is the size of the cropped target image, to ensure that the images (target and object) are the same size as each other. Lines 118 to 122 are nested for loops that will pass through all pixels in the image and line 120 checks if the current pixel has a magnitude greater than zero (a not black pixel) and adds one to the local white count if true.

Lines 123 to 126 are an if statement that checks if the local white count is smaller than the lowest white count, and if it is lower, the angle and white count is stored.

```
107     cenObj.x = ObjImg.cols / 2;
108     cenObj.y = ObjImg.rows / 2;
109
110     white = ObjImg.cols * ObjImg.rows;
111
112     for (double ang = 0; ang < 360; ang++) {
113         double whiteCount = 0;
114
115         warpAffine(ObjImg, ObjImgCheck, getRotationMatrix2D(cenObj, ang, 1), Size(TarImg.cols, TarImg.rows));
116         TarImgCheck = TarImg - ObjImgCheck;
117
118         for (int x = 0; x < TarImgCheck.cols; x++) {
119             for (int y = 0; y < TarImgCheck.rows; y++) {
120                 whiteCount = ((TarImgCheck.at<uchar>(y, x) > 0) ? whiteCount + 1 : whiteCount);
121             }
122         }
123         if (whiteCount < white) {
124             white = whiteCount;
125             angle = ang;
126         }
127     }
```

Figure 10 Angle Finding Code

2.3.8. Matrix Output

The homogenous transformation matrix is outputted through the command window buy using the standard output stream, “cout”, and the code for this is shown in Figure 11. The rotational output is a number, not “sin “ and the angle.

```
129     cout << "Image " << num << endl;
130
131     cout << '|' << setw(adj) << cos(angle * pi / 180);
132     cout << setw(adj) << -sin(angle * pi / 180);
133     cout << setw(adj) << deltaX << " |" << endl;
134
135     cout << '|' << setw(adj) << sin(angle * pi / 180);
136     cout << setw(adj) << cos(angle * pi / 180);
137     cout << setw(adj) << deltaY << " |" << endl;
138
139     cout << '|' << setw(adj) << 0;
140     cout << setw(adj) << 0;
141     cout << setw(adj) << 1;
142     cout << " |" << endl;
143
144     for (int i = 0; i < (3 * adj + 3); i++) cout << '_';
145     cout << endl;
```

Figure 11 Matrix Output Writing Code

Line 129 states the number of the image being shown. Lines 131 to 133 output the first line of the matrix and uses the “setw” function to create a set width for the numbers to be written into so that the matrix is formatted to have all numbers in the same columns. Lines 135 to 137 and 139 to 142 show the writing of the second and third lines respectively. Line 144 creates a for loop that prints enough “_” characters to be the same width as the numbers in the matrix. This is based on the “adj” value which determines the width of the space for the numbers in the matrix. Line 145 adds an extra line to ensure that the outputs have a space between each matrix, for ease of reading.

3. Results

The output of the program is shown in Figure 12 and has been labelled to show which matrix is for which image. The matrix used is a three by three matrix as this is a two-dimensional application, not a three-dimensional one which would require a four by four matrix. Image one has an angle of 0°, image two has an angle of 45°, image three has 270°, image four has 314° and image five has 134°. These angles are based on a Z axis passing into the page.

```
Image 1
|           1           -0      369.517 |
|           0            1      800.025 |
|           0            0            1 |
_____________________________________________
Image 2
|    0.707107    -0.707107      723.892 |
|    0.707107     0.707107      752.473 |
|           0            0            1 |
_____________________________________________
Image 3
| -1.83697e-16           1      1388.03 |
|          -1 -1.83697e-16      361.648 |
|           0            0            1 |
_____________________________________________
Image 4
|     0.71934     0.694658       1736.5 |
|   -0.694658      0.71934          861 |
|           0            0            1 |
_____________________________________________
Image 5
|   -0.694658     -0.71934      1819.98 |
|     0.71934    -0.694658       115.88 |
|           0            0            1 |
_____________________________________________
Press any key to continue . . .
```

Figure 12 Output Matrices of the Program