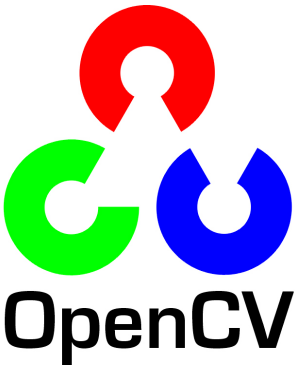
OpenCV Report 3
Robotics and Automation 282 762

Marc Alexander Sferrazza
12164165 *†

May 22, 2017



Abstract

A brief report on OpenCV to calculate spatial transformations. Each step was tested and an output image saved to the local folder. For a more detailed instruction on the stages not explained please review the source code in the appendix.

*This work was not supported by any organization
†Faculty of Mechatronics Engineering, Massey University, Albany, Auckland, New Zealand `Progress of project: https://github.com/alex1v1a/Robotics-and-Automation/`

Contents

List of Figures

1 INTRODUCTION

OpenCV (Open Source Computer Vision Library) is an open source machine learning library for visual components. It has been widely accepted as one of the worlds most used image processing tools, and has the capability of static and dynamic comprehension. Some of the basic features available in OpenCV are capable mapping environments to be used in demonstration such as self driving cars lane detection.

The process's aim in this project to achieve is to design a program in the Visual Studio environment, using the OpenCV library that will detect an characters location and find the work piece and its target location in a given workspace. The objective is to then calculate the distance and angle required to translate and rotate the work piece from its current location to it target location; and finally express the distance and angle as a homogeneous transform, and display the final results and image to the user, as described in the outline.

Unfortunately due to technical issues experienced, the final program has been compiled from source on a Mac with terminal. The shared and static lib's created in cmake then processed executables saved in the local library. Please note this when trying to re-compile as the some of the linkers may direct to other locations e.g. headers.

OpenCV is a powerful tool for image processing with more then 2500 algorithms available in its library; by using several functions within the OpenCV library, a processed image will produce these values.

2 METHOD

A few images have been provided of different characters on a workspace and their corresponding transform locations. The process will involve filtering this image to a workable standard in which the contours can be used to formulate the translations and transformations from frames of workspaces.

As there are five images the option to run the process with all outputs was first made, but then the decision of what images to check was implemented using the case keys 1, 2, 3, 4, 5, esc - A prompt will be given to the user to select which image to process and find the balls with their colours; the user can cycle back to the main image selection and select another image to process.

The program can be closed by selecting the esc key from the prompt menu. This process helps break down the information for a simpler and more specific program allowing the user to choose their results accordingly.

The image is then loaded to memory and with each step through the detection the image is saved for easy debugging, it also helps create a faster process and cleaner environment.

After the user has selected a desired image to process the transform function is called to preform the action. The sequence begins with the original image as shown below (set three from the five images are used for demo purposes in this report, as it was found one of the more complex)

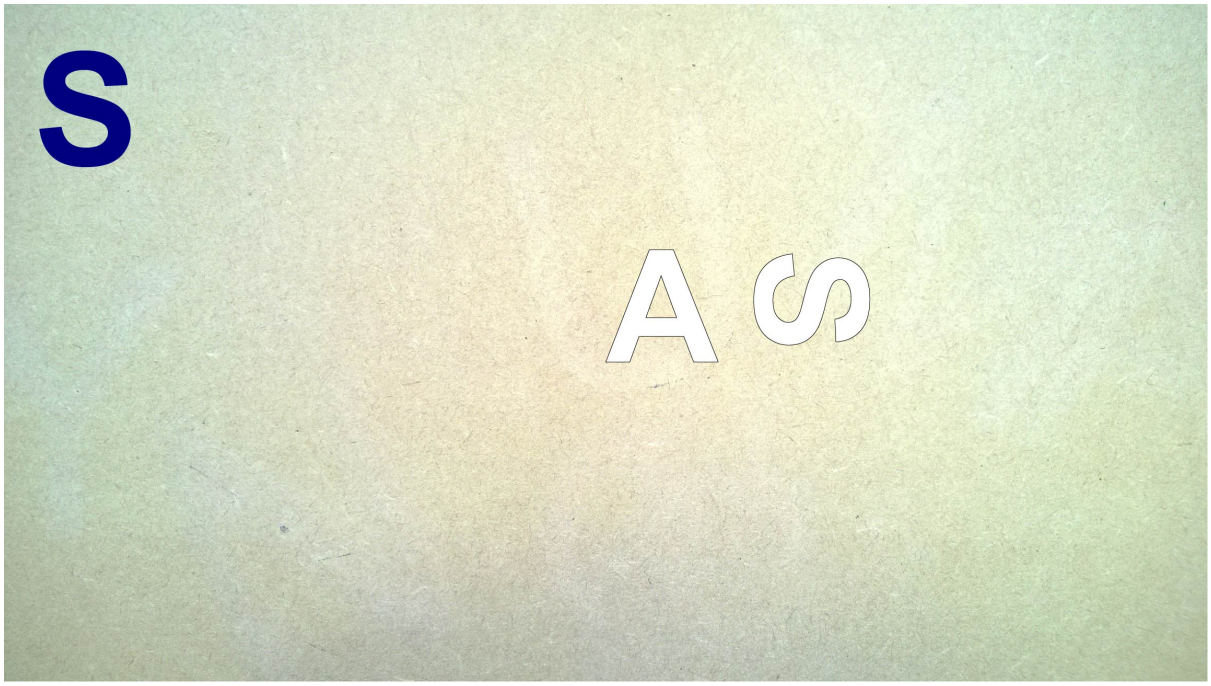


Figure 1: Initial Image

After the image has been processed and the workspace translated, and output is displayed of the resulting transformations; a homogenous matrix is also displayed.

A full break down of the process has been attached and can be found in the code in the appendix. The following code is the process for user input and the function call for transform.

```
int main() {
  cout << "Choose an image to transform: 1, 2, 3, 4, 5...\n\npress esc to
      quit\n\n
      ------------------------------------------------------------------------\
      n" << endl;
  while (1) {
    namedWindow("User Select", CV_WINDOW_NORMAL);
    switch (waitKey(0)) {
    case 27: // Exit prompt "esc"
      return 0;
    case 49: // Image 1 selected
      setpoint = 1;
      name = "1.jpg";
      transform();
      break;
    case 50: // Image 2 selected
      setpoint = 2;
      name = "2.jpg";
      transform();
      break;
    case 51: // Image 3 selected
      setpoint = 3;
      name = "3.jpg";
      transform();
      break;
    case 52: // Image 4 selected
      setpoint = 4;
      name = "4.jpg";
```

```
        transform();
        break;
      case 53: // Image 5 selected
        setpoint = 5;
        name = "5.jpg";
        transform();
        break;
      }
    }
    return 0;
}
```

2.1 BGR to HSV conversion

After the transform function has been called, the image is read into memory and resized for processing. In order to make the process more definite a conversion to greyscale is necessary, this will make the process more robust by converting a BGR. The greyscale conversion is ready to begin, using the cvtColor function to convert the image to greyscale (BGR2GRAY) will help in the detection processing.

```
// Images stored to memory for transformations
Mat image, greyscale, threshold;

// Read the image to memory and resize
image = imread(name);
resize(image, image, Size(), 0.5, 0.5);

// Convert BGR to Greyscale
cvtColor(image, greyscale, CV_BGR2GRAY);
```

2.2 Gaussian Blur

The Gaussian blur is used to reduce the noise of the image and smoothing the edges to help eliminate any external influences when finding the shape and give a more accurate average of the pixels. The function will transform the image using convolution matrix Gaussian kernel, and will give a result based on each pixel and it's surrounding pixels to contour and blur while maintaining the edges integrity.

After the Gaussian process, the image is passed to the canny operator for edge detection. The best values for Gaussian used are a 3x3 kernel matrix with a standard deviation of 1.

```
// Preform Gaussian blur to smooth the image and reduce noise
blur(greyscale, greyscale, Size(3, 3));
```

2.3 Canny edge detection

The Gaussian blur Canny edge detection function is then called to preform detection for the workspace. The frames are assessed from the greyscale smooth edged original image, and the output is assigned to out threshold image space.

After the canny edge detection the image is then passed though a dilation and erosion functions to reduce noise.

```
// Use canny to detect the edges
Canny(greyscale, threshold, 150, 300, 3);
```

2.4 Dilation & Erosion

Using the dilation function the image begins to form more solid shapes and removes discrepancies to better form solid filled in shapes, by adding pixels to the boundaries found of objects and shapes in a greyscale image. The erosion function then emphasises those shapes making it easier for the shape identifier later on by removing pixels from the boundaries of objects. The best values found for the size of the morphological operations are as shown below.

```
// Dilation and errosion used for multi level channel processing, reduced
    noise
dilate(threshold, threshold, getStructuringElement(MORPH_ELLIPSE, Size(5,
    5)));
erode(threshold, threshold, getStructuringElement(MORPH_ELLIPSE, Size(5,
    5)));
```

2.5 Contours

This algorithm finalises the edge detection and creates fine line edges for rotating the rectangles. After all the vectors containing contours are stored, and the hierarchy datatype is linked, when the function is called the parent and child contours are stored respectivly.

The algorithm finds the rectangles of each outline (character) which subsequently is used to establish the orientation by using the vertices of the contour, and in turn find the difference in angle for each object. The output of this process gives two vectors that contains instances as elements the same size of the vectors of that from contour. The significant points are then recorded to the vector matrices to find the spacial transformations.

```
// Find the contours, example linked from here http://docs.opencv.org
    /2.4/doc/tutorials/imgproc/shapedescriptors/find_contours/
    find_contours.html
findContours(threshold, contours, hierarchy, CV_RETR_EXTERNAL,
    CV_CHAIN_APPROX_NONE, Point(0, 0));

  // Find the rotated rectangles contour
vector<RotatedRect> minRect(contours.size());
for (int i = 0; i < contours.size(); i++) {
  minRect[i] = minAreaRect(Mat(contours[i]));
}
```

2.6 Bounding Rectangles & Approximate Contours to Polygons

Bounding for a particular object rectangle converts the contour to a polygon shape. The vector is that of the same size of the contour vector, but is declared to address the points of each rectangle. The algorithm used converts the bounding rectangles to a polygon shape.

```
vector<vector<Point> > contours_poly(contours.size());
vector<Rect> boundRect(contours.size());
vector<Point2f>center(contours.size());
vector<float>radius(contours.size());

for (int i = 0; i < contours.size(); i++) {
  approxPolyDP(Mat(contours[i]), contours_poly[i], 3, true);
  boundRect[i] = boundingRect(Mat(contours_poly[i]));
}
```

2.7 Drawing Contours, Rotated Rectangles & Center circles with Axis

In this process an image is then created to find the spacial transforms. The first step is to save all the points required to the image, then draw them on the final image. The process involves using contrasts to find contour lines, then looping to draw the contour lines and rectangles accordingly. The center and translation lines are then drawn to establish the difference between the two rectangles and find the coordinates for each contour.

```
for (int i = 0; i< 2; i++) {
  Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.
      uniform(0, 255));
  drawContours(final, contours, i, color, 1, 8, vector<Vec4i>(), 0, Point
      ());
  rectangle(final, boundRect[i].tl(), boundRect[i].br(), color, 2, 8, 0);

  // Rotated rectangle
  Point2f rect_points[4]; minRect[i].points(rect_points);
  for (int j = 0; j < 4; j++) {
    line(final, rect_points[j], rect_points[(j + 1) % 4], color, 1, 8);
  }

  // Final centre circle and translation arrow
  circle(final, minRect[i].center, 4, color, -1, 8, 0);

  // Display the translational arrow
  arrowedLine(final, minRect[0].center, minRect[1].center, color, 1, 8);

  // Read out the center coordinates
  cout << "Coordinates for contour " << i + 1 << " are " << minRect[i].
      center << endl;
}
```

2.8 Translation

As the z axis is ignored in this instance the x and y axis are observed to find the translation difference of the two center points of the rectangles. Using Pythagoras's trigonometry equations with the established center points a subsidy can be made to suit the algorithm and simply find the difference of the characters. Then to rotate the character it is first found between a positive or negative 180 degree value, and in the clockwise or anticlockwise direction. The angles are found for each set of 90 degrees passed, then adjusted accordingly.

```
// Preform the translation distance equation
translation = sqrt(((Bx - Ax)*(Bx - Ax)) + ((By - Ay)*(By - Ay)));

// Find the rotation of the first character
double deg0 = minRect[0].angle;
if (minRect[0].size.width < minRect[0].size.height) {
  deg0 = 90 + deg0;
}

// Find the rotation of the seccond character
double deg1 = minRect[1].angle;
if (minRect[1].size.width < minRect[1].size.height) {
  deg1 = 90 + deg1;
}

// Find the difference in rotation between characters
```

```
double rotation = (deg0 - deg1);

// Preform the transform to the matrix table in the z direction
double transform[4][4] = {
  { (cos(rotation)), (sin(rotation)), 0, Tx },
  { (-sin(rotation)), (cos(rotation)), 0, Ty },
  { 0, 0, 1, Tz },
  { 0,0,0,1 }
};
```

3 RESULTS

After the processing is complete the original image is displayed, and terminal output response given; The final result is shown below. The process is successful and the workspace transformation is complete. The user can now press any key to go back and select another image, or after closing the image hit the "esc" key to exit the program.

```
// Display the homogeneous transform matrix
cout << "\nHomogeneous Transfomation Matrix";
for (int i = 0; i < 4; i++) {
  printf("\n[");
  for (int j = 0; j < 4; j++) {
    printf("%f, ", transform[i][j]);
  }
  cout << "]";
}

// Display the rotation and translation between frames
cout << "\n\nRotation between characters " << rotation << "deg" << endl;
cout << "Translation between characters " << translation << endl;
cout << "\n\n" << endl;

// After processing and transforming is complete, display final results,
   and save for reference
// Naming the saved output files for reference
ostringstream save;
// Image save locations
string savename = "Output/Final Image ";
// Converge the string for each transformed image
save << savename << setpoint << ".jpg";

// Display the initial image
namedWindow("Initial Image", CV_WINDOW_NORMAL);
imshow("Initial Image", image);
// Display the transformed image
imshow(save.str(), final);
imwrite(save.str(), final);
```

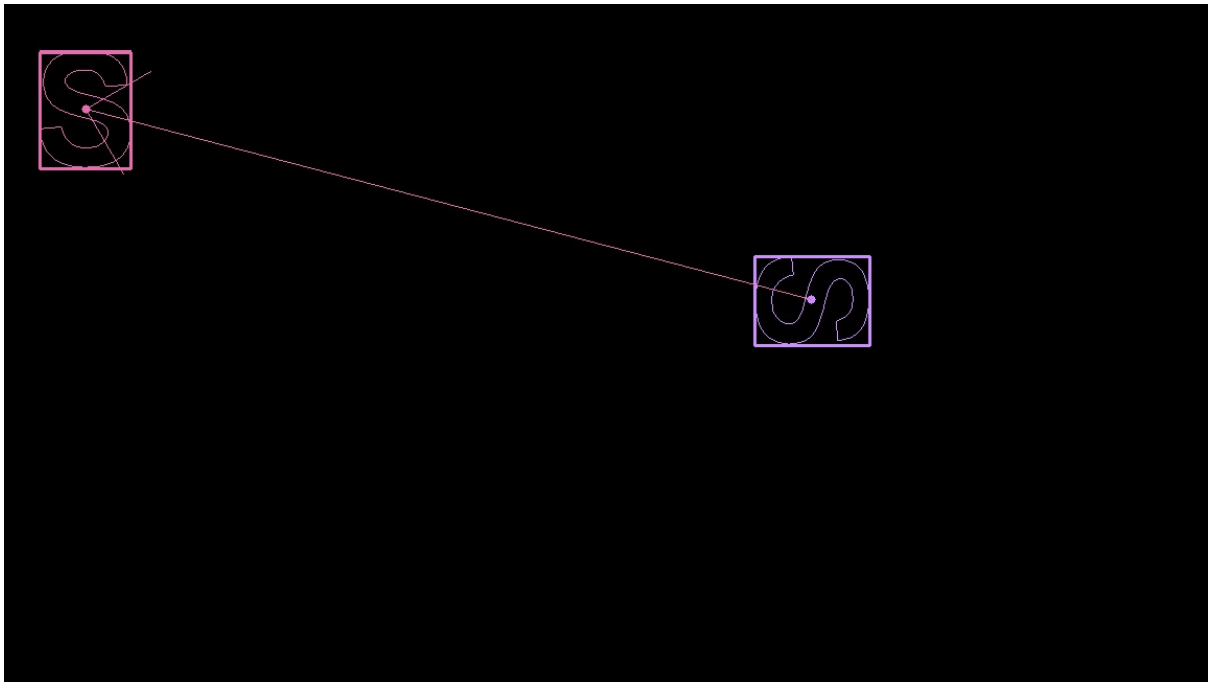


Figure 2: The final image is given with workspace results

After the transformation and translation of the frames the final image is saved to the local output folder, per process, the center coordinates, homogeneous transformation matrix, rotation and translation between characters are all displayed to the user as output in terminal. A sample of the results can be seen below.

```
Choose an image to transform: 1, 2, 3, 4, 5...

press esc to quit

--------------------------------------------------------------------------

For image 3.jpg

Coordinates for contour 1 are [772.5, 282.5]
Coordinates for contour 2 are [77.5, 100]

Homogeneous Transfomation Matrix
[-0.448074, -0.893997, 0.000000, -695.000000, ]
[0.893997, -0.448074, 0.000000, -182.500000, ]
[0.000000, 0.000000, 1.000000, 0.000000, ]
[0.000000, 0.000000, 0.000000, 1.000000, ]

Rotation between characters -90deg
Translation between characters 718.562
```

Figure 3: The final results shown of the output: center coordinates, homogeneous transformation matrix, rotation and translation between characters

4 OUTCOMES

4.1 Fine Tuning

While the process is consistent it is a clear underling of step by step operations, the procedure in which was either suitable or inadequate. This involves checking the matrix at different points like the threshold values, dilation and erosion, the Gaussian blur, and in particular the vectors and contours for find bounding rects, circles, and approximate contours to polygons, translation and transforms of each frame and combine workspace.

4.2 Testing

Ensure the target of finding translation from frames are complete and accurate, checking each function independently for a robust break down and good debugging.

Each step had an output image saved to check the processes respectively. The images are checked to make sure each stage is completing the task correctly, if not the code is then referred to and further revisions are made.

Every image was tested with the values for recognition and tweaked in the fine tuning process to get the best possible results.

4.3 Finalising

After all the results were finalised and correct the outputs and results confirmed the code was then stripped and simplified for clarity and ease of use for future development and reference.

Making the program stable and concise, and as robust as possible builds for a good design. Checking over the functions and any redundant code, ensure a good user interface is easily workable and the outputs are all correct and comprehensive.

5 CONCLUSIONS

5.1 Critical Evaluation and Methodology

The final transformation of workspaces have been produced from several steps. This demonstration gives the user a guided input and accurate clear output for, easy to use, and self guided results.

As the output for all processed images was found accurate and complete the method has been found effective. The results are clearly given show the center coordinates of both contours, a homogeneous transformation matrix, rotation and translation between characters all displayed to the user as output in terminal, and the final image saved to the output source folder.

5.2 Discussion

This process preforms and meets all aims and objects set out to achieve and can be considered a success. All required information displayed to the user in an orderly fashion and input methods are acceptable. In achieving this task the demonstration presents a practical form in which by using bounding rectangles, contours and points vectors, a total accurate reformation can be drawn accross a translational span.

The process's used were successfully able to detect and identify all characters, and have been able to take the product of matrices and find significant angles. The transformations have completed all images with speed and accuracy, and is a viable method of detection and transformation.

References

[1] http://docs.opencv.org/2.4/, "Opencv resources," 2015.

[2] http://docs.opencv.org/2.4/doc/tutorials/, "Object detection," 2015.

[3] http://docs.opencv.org/3.0 beta/doc/, "Opencv resources," 2016.

APPENDIX

```
// Marc Alexander Sferrazza 12164165
// Referenceing from http://docs.opencv.org/2.4/
// http://docs.opencv.org/2.4/doc/tutorials/imgproc/shapedescriptors/
    find_contours/find_contours.html

#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <iostream>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <sstream>

using namespace cv;
using namespace std;

void transform();

int setpoint;
string name;

int main() {
  cout << "Choose an image to transform: 1, 2, 3, 4, 5...\n\npress esc to
      quit\n\n
      ------------------------------------------------------------------------\
      n" << endl;
  while (1) {
    namedWindow("User Select", CV_WINDOW_NORMAL);
    switch (waitKey(0)) {
    case 27: // Exit prompt "esc"
      return 0;
    case 49: // Image 1 selected
      setpoint = 1;
      name = "1.jpg";
      transform();
      break;
    case 50: // Image 2 selected
      setpoint = 2;
      name = "2.jpg";
      transform();
      break;
    case 51: // Image 3 selected
      setpoint = 3;
      name = "3.jpg";
      transform();
      break;
    case 52: // Image 4 selected
      setpoint = 4;
      name = "4.jpg";
      transform();
      break;
    case 53: // Image 5 selected
      setpoint = 5;
      name = "5.jpg";
      transform();
      break;
```

```
    }
  }
  return 0;
}

void transform() {
  // Images stored to memory for transformations
  Mat image, greyscale, threshold;

  // Read the image to memory and resize
  image = imread(name);
  resize(image, image, Size(), 0.5, 0.5);

  // Convert BGR to Greyscale
  cvtColor(image, greyscale, CV_BGR2GRAY);

  // Preform Gaussian blur to smooth the image and reduce noise
  blur(greyscale, greyscale, Size(3, 3));

  // Use canny to detect the edges
  Canny(greyscale, threshold, 150, 300, 3);

  // Dilation and errosion used for multi level channel processing, reduced
      noise
  dilate(threshold, threshold, getStructuringElement(MORPH_ELLIPSE, Size(5,
      5)));
  erode(threshold, threshold, getStructuringElement(MORPH_ELLIPSE, Size(5,
     5)));

  // Finding contours variables and vectors
  RNG rng(12345);
  vector<vector<Point> > contours;
  vector<Vec4i> hierarchy;

  // Find the contours, example linked from here http://docs.opencv.org
     /2.4/doc/tutorials/imgproc/shapedescriptors/find_contours/
     find_contours.html
  findContours(threshold, contours, hierarchy, CV_RETR_EXTERNAL,
     CV_CHAIN_APPROX_NONE, Point(0, 0));

  // Remove letter A from image 3 only
  if (contours.size() > 2) {
    contours.erase(contours.begin() + 1, contours.end() - 1);
  }

  // Find the rotated rectangles contour
  vector<RotatedRect> minRect(contours.size());
  for (int i = 0; i < contours.size(); i++) {
    minRect[i] = minAreaRect(Mat(contours[i]));
  }

  // Find bounding rects, circles, and approximate contours to polygons
  vector<vector<Point> > contours_poly(contours.size());
  vector<Rect> boundRect(contours.size());
  vector<Point2f>center(contours.size());
  vector<float>radius(contours.size());

  for (int i = 0; i < contours.size(); i++) {
```

```
  approxPolyDP(Mat(contours[i]), contours_poly[i], 3, true);
  boundRect[i] = boundingRect(Mat(contours_poly[i]));
}

// Final image
Mat final = Mat::zeros(threshold.size(), CV_8UC3);

// Identify the working image
cout << "For image " << name << "\n" << endl;

// Drawing contours for each image
for (int i = 0; i< 2; i++) {
  Scalar color = Scalar(rng.uniform(0, 255), rng.uniform(0, 255), rng.
      uniform(0, 255));
  drawContours(final, contours, i, color, 1, 8, vector<Vec4i>(), 0, Point
      ());
  rectangle(final, boundRect[i].tl(), boundRect[i].br(), color, 2, 8, 0);

  // Rotated rectangle
  Point2f rect_points[4]; minRect[i].points(rect_points);
  for (int j = 0; j < 4; j++) {
    line(final, rect_points[j], rect_points[(j + 1) % 4], color, 1, 8);
  }

  // Final centre circle and translation arrow
  circle(final, minRect[i].center, 4, color, -1, 8, 0);

  // Display the translational arrow
  arrowedLine(final, minRect[0].center, minRect[1].center, color, 1, 8);

  // Read out the center coordinates
  cout << "Coordinates for contour " << i + 1 << " are " << minRect[i].
      center << endl;
}

// Find translation
// Variables for the translation coordinates
double Ax = minRect[1].center.x;
double Ay = minRect[1].center.y;
double Bx = minRect[0].center.x;
double By = minRect[0].center.y;
double translation;
double Tx = -(Bx - Ax);
double Ty = -(By - Ay);
double Tz = 0;

// Preform the translation distance equation
translation = sqrt(((Bx - Ax)*(Bx - Ax)) + ((By - Ay)*(By - Ay)));

// Find the rotation of the first character
double deg0 = minRect[0].angle;
if (minRect[0].size.width < minRect[0].size.height) {
  deg0 = 90 + deg0;
}

// Find the rotation of the seccond character
double deg1 = minRect[1].angle;
if (minRect[1].size.width < minRect[1].size.height) {
```

```
    deg1 = 90 + deg1;
  }

  // Find the difference in rotation between characters
  double rotation = (deg0 - deg1);

  // Preform the transform to the matrix table in the z direction
  double transform[4][4] = {
    { (cos(rotation)), (sin(rotation)), 0, Tx },
    { (-sin(rotation)), (cos(rotation)), 0, Ty },
    { 0, 0, 1, Tz },
    { 0,0,0,1 }
  };

  // Display the homogeneous transform matrix
  cout << "\nHomogeneous Transfomation Matrix";
  for (int i = 0; i < 4; i++) {
    printf("\n[");
    for (int j = 0; j < 4; j++) {
      printf("%f, ", transform[i][j]);
    }
    cout << "]";
  }

  // Display the rotation and translation between frames
  cout << "\n\nRotation between characters " << rotation << "deg" << endl;
  cout << "Translation between characters " << translation << endl;
  cout << "\n\n" << endl;

  // After processing and transforming is complete, display final results,
     and save for reference
  // Naming the saved output files for reference
  ostringstream save;
  // Image save locations
  string savename = "Output/Final Image ";
  // Converge the string for each transformed image
  save << savename << setpoint << ".jpg";

  // Display the initial image
  namedWindow("Initial Image", CV_WINDOW_NORMAL);
  imshow("Initial Image", image);
  // Display the transformed image
  imshow(save.str(), final);
  imwrite(save.str(), final);

  // Wait for key press before returning to main menu
  waitKey(0);
  destroyAllWindows();
}
```