# **Computer Vision Decoration program flow**

## **Step 1: Identify blocks on the conveyor (“Main1\_DetectBlocksOnConveyor”)**

This function Identified whether blocks exists on the conveyor to be picked up. This step did not detect shape or colour of the block, hence if blocks with letters were placed on the conveyor, it was detected. The program flow was as follow:

1. The camera on top of the conveyor was calibrated with the function below. Definition of the function has been included in Appendix A.

CameraCalibrationConveyor;

1. Image was read into the file and calibrated based on the parameters calculated from the previous function.

conveyorImg = imread('Proper\_Pics\conveyor5.jpg');

conveyorImg = undistortImage(conveyorImg,...

cameraParamsConveyor);

1. A colour mask was applied to the image to filter out the blocks from the background of the colour (the definition of the mask function has been included in appendix B). The result was inverted (to have the blocks appear as white and background as black) and segmented to only include the section of the conveyor reachable by the IRB-120 (definition of segmentSection is included in Appendix C). Additional image processing methods were also applied to remove any white noises from the image.

[BW,maskedRGBImage] = createConveyorMask3(conveyorImg);

BW = ~BW;

BW = segmentSection(BW, 555, 1155, 10, 586);

BW = bwareaopen(BW,2000); % remove white noise

BW = bwmorph(BW, 'hbreak');

1. The centroid and area of the blocks detected were calculated using “regionprops”. If blocks were detected, any areas larger or smaller than the average blocks were removed (potentially came from additional white noise) and the remaining centroids were converted to real life coordinates and returned (definition of “conveyorPxlToReal” function has been included in Appendix D). If no blocks were detected, the code simply returns a message indicating and returns an empty array, indicated no detection of blocks.

% use regionprops to calculate centroids

blockProps = regionprops(BW, ‘Centroid’,’Area’);

centroids = vertcat(blockProps.Centroid);

areas = vertcat(blockProps.Area);

if size(centroids,1) > 0

removeIdx = find(areas > 7000 | areas < 1000);

areas(removeIdx) = [];

centroids(removeIdx,:) = [];

% convert to real life

conveyorCentroids = conveyorPxlToReal(centroids(:,1), centroids(:,2));

figure();

imshow(BW); hold on;

plot(centroids(:,1), centroids(:,2), ‘c\*’, ‘MarkerSize’, 10);

else

display(“No blocks detected on the conveyor”);

conveyorCentroids = [];

end

The results are shown in the table below

|  |  |
| --- | --- |
| **Step and Description** | **Figure** |
| Image after calibration was completed |  |
| Colour filter applied to extract the blocks. |  |
| The black and white colour was inverted, segmented to sections reachable by the IRB-120 and white noise was removed. |  |
| “Regionprops” was used to detect the centroid and image was marked. |  |

## **Step 2: Identify the block picked up (“Main2\_ IdentifyBlock”)**

Once the block was moved from the conveyer to a designated area on the table (predetermined by the team), this function analysed the block and identified its shape, colour, coordinates and orientation (angle relative to the y-axis of the table). The program flow was as follow:

1. Variables allocated for later use and Camera Calibration was done similar to Step 1 (definition of “CameraCalibration” was similar to “CameraCalibrationConveyor” as shown in Appendix A).

red = 1.1;

green = 1.2;

blue = 1.3;

yellow = 1.4;

circle = 2.1;

flower = 2.2;

diamond = 2.3;

square = 2.4;

star4 = 2.5;

star6 = 2.6;

recognise = 0;

CameraCalibration;

1. The image was read and calibrated with parameters calculated in “CameraCalibration” and segmented to only view the designated area for analysis. A black and white version of the image was also stored for later use in the code.

% Undistort and segment

transfer\_Img = imread('Proper\_Pics\Shapes\blueSquare.jpg');

transfer\_Img = undistortImage(transfer\_Img, cameraParams);

transfer\_Img = segmentSection(transfer\_Img, 1238, size(transfer\_Img,2), 290, 783);

figure; imshow(transfer\_Img);

% Get black and white version of the image

transfer\_ImgBW = im2bw(transfer\_Img);

transfer\_ImgBW = ~transfer\_ImgBW;

transfer\_ImgBW = segmentSection(transfer\_ImgBW, 1238, size(transfer\_Img,2), 290, 783);

transfer\_ImgBW = bwareaopen(transfer\_ImgBW,400);

figure; imshow(transfer\_ImgBW);

1. The block was filtered with all 4 possible colours (red, green, blue, yellow), white noise was removed and the code checks which of the 4 colours exists (e.g. if the block was of red colour, the red mask would return a white spot/blob whilst the other 3 colour filters would return black).

% Colour masking

[myRedShpBW,myRedShp] = createTableRedMask(transfer\_Img);

[myGreenShpBW,myGreenShp] = createTableGreenMask(transfer\_Img);

[myBlueShpBW,myBlueShp] = createTableBlueMask(transfer\_Img);

[myYellowShpBW,myYellowShp] = createTableYellowMask(transfer\_Img);

% Remove any white noise from all the masks

myRedShpBW = bwareaopen(myRedShpBW,400);

myGreenShpBW = bwareaopen(myGreenShpBW,400);

myBlueShpBW = bwareaopen(myBlueShpBW,400);

myYellowShpBW = bwareaopen(myYellowShpBW,400);

% Determine if the colour is red, green, blue or yellow

redExist = find(myRedShpBW);

greenExist = find(myGreenShpBW);

blueExist = find(myBlueShpBW);

yellowExist = find(myYellowShpBW);

1. The code detected the colour and recorded it for later use. If none of the 4 filters returned anything (i.e. the shape was of unrecognisable colour), the function returned an empty struct for its “Shape” and “Colour” fields.

if length(redExist) > 50

display("It's a red shape");

Colour = 'Red';

blockColour = red;

myShpGray = rgb2gray(myRedShp);

myShpBW = myRedShpBW;

myFinalShp = myRedShp;

recognise = 1;

elseif length(greenExist) > 50

display("It's a green shape");

Colour = 'Green';

blockColour = green;

myShpGray = rgb2gray(myGreenShp);

myShpBW = myGreenShpBW;

myFinalShp = myGreenShp;

recognise = 1;

elseif length(blueExist) > 50

display("It's a blue shape");

Colour = 'Blue';

blockColour = blue;

myShpGray = rgb2gray(myBlueShp);

myShpBW = myBlueShpBW;

myFinalShp = myBlueShp;

recognise = 1;

elseif length(yellowExist) > 50

display("It's a yellow shape");

Colour = 'Yellow';

blockColour = yellow;

myShpGray = rgb2gray(myYellowShp);

myShpBW = myYellowShpBW;

myFinalShp = myYellowShp;

recognise = 1;

else

% Unrecognisable shape

%display("The shape and colour are unrecognisable");

shapeProps.Colour = [];

shapeProps.Shape = [];

end

1. If the shape was recognisable, “Regionprops” was used once again to determine the properties of the shape given. These properties were used to filter the likelihood of a shape the block belongs to. E.g. if the area was greater than 1200 pixels, it was likely it was a circle shape. If the area was less than 1200 pixels and greater than 850, it was either flower, a diamond or a square. Further filter was applied to differentiate between the 3 using properties such as perimeter and angle of the shape relative to the angle of the block [1], etc. The process was repeated until the shape was determined. Once determined, the colour, shape, location of centroid and orientation was returned.
2. If the shape was not recognisable, the code analysed the black and white image of the block to see if any block had been placed on the transfer section. If “Regionprops” returned an empty array, it indicated an empty transfer section hence the “Centroid” and “Orientation” fields of the result was empty. If the transfer section has a block, the function would return values for Centroid and Orientation whilst the shape and colour fields were left empty.

if recognise == 1

% Determine the shape property

s = regionprops(myShpBW, 'Area', 'MajorAxisLength', 'MinorAxisLength','Eccentricity', 'Orientation', 'EulerNumber', 'EquivDiameter', 'Perimeter', 'ConvexArea', 'Extent', 'FilledArea', 'Solidity', 'Centroid');

shpStats = [s.Area s.MajorAxisLength

s.MinorAxisLength s.Eccentricity s.Orientation

s.EulerNumber s.EquivDiameter s.Perimeter s.ConvexArea s.Extent s.FilledArea s.Solidity];

% determine the angle of the shape

shpAngle = calculateAngle(myShpBW);

blockAngle = calculateAngle(transfer\_ImgBW);

% Determine the shape

if s.Area > 1200

display("It's a circle");

Shape = 'Circle';

blockShape = circle;

elseif s.Area > 850

if s.Perimeter > 150

display("It's a flower");

Shape = 'Flower';

blockShape = flower;

else

if abs(shpAngle - blockAngle) < 10

display("It's a square");

Shape = 'Square';

blockShape = square;

else

display("It's a diamond");

Shape = 'Diamond';

blockShape = diamond;

end

end

else

if s.MajorAxisLength > 34.5

display("It's a 4star");

Shape = '4star';

blockShape = star4;

else

display("It's a 6star");

Shape = '6star';

blockShape = star6;

end

end

% record final properties

transferCentroid = tablePxlToReal(s.Centroid(1), s.Centroid(2));

transferOrientation = blockAngle;

shapeProps.Colour = blockColour;

shapeProps.Shape = blockShape;

shapeProps.Centroid = transferCentroid;

shapeProps.Orientation = transferOrientation;

else

% see if there are any blocks at all

s = regionprops(transfer\_ImgBW, 'Centroid');

centroid = vertcat(s.Centroid);

if size(centroid, 1) > 0

% block does exist

shapeProps.Centroid = tablePxlToReal(s.Centroid(1), s.Centroid(2));

shapeProps.Orientation = calculateAngle(transfer\_ImgBW);

else

% transfer section is empty

%display("The transfer section is empty");

shapeProps.Centroid = [];

shapeProps.Orientation = [];

end

The results are as shown below

|  |  |
| --- | --- |
| **Step and Description** | **Figure** |
| Image after calibration and segmented. |  |
| Black and white picture of the block |  |
| Colour filter mask was applied and indicated it was of blue colour. |  |
| The result once the shape was recognised |  |
| If the colour was unrecognisable and the transfer section was empty, the command prompt returned a message to indicate such event and return an empty struct |  |

## **Step 3: Identify the shapes in the pattern given by the customer and its properties (“Main3\_IdentifyShapesInPattern”)**

This function analysed the desired pattern given by the customer in any orientation and positioned in the 9x9 designated grid. The function returned a collection of all the shapes, colours, and corresponding coordinates of its centroid and orientation. The code was as follow:

1. Similar to step 2 numbers 1 and 2, variables were declared for later use and the camera was calibrated. The image was read, undistorted using parameters calculated from “CameraCalibration” and segmented to the 9x9 grid area. A black and white version of the image was also stored for later use.

red = 1.1;

green = 1.2;

blue = 1.3;

yellow = 1.4;

CameraCalibration;

%% Identify shapes and colour of the pattern

table\_Img = imread('Proper\_Pics\Patterns\Pattern1.jpg');

table\_Img = undistortImage(table\_Img, cameraParams);

table\_Img = segmentSection(table\_Img, 552, 1043, 288, 782);

figure; imshow(table\_Img);

table\_ImgBW = ~im2bw(table\_Img);

table\_ImgBW = segmentSection(table\_ImgBW, 552, 1043, 288, 782);

table\_ImgBW = removeLettersAndNumbers(table\_ImgBW);

table\_ImgBW = bwareaopen(table\_ImgBW,100);

figure; imshow(table\_ImgBW);

patternProps.Colour = [];

patternProps.Shape = [];

patternProps.Centroid = [];

patternProps.Orientation = [];

1. Colour filters were applied to the pattern to extract all the desired colours. Extra image processing were also done to remove any white noise, fill in any holes and cater for different shades of glare that affected different colours.

%% Filter all colours

% red

[myPatternRedBW,myPatternRed] = createPatternRedMask(table\_Img);

myPatternRedBW = imfill(myPatternRedBW,'holes');

myPatternRedBW = bwareaopen(myPatternRedBW,100);

%myPatternRedBW = bwmorph(myPatternRedBW, 'majority');

figure; imshow(myPatternRed);

figure; imshow(myPatternRedBW);

% green

% green's bit more sensitive

table\_Img = imsharpen(table\_Img, 'Radius', 5, 'Amount', 1.5);

[myPatternGreenBW,myPatternGreen] = createPatternGreenMask(table\_Img);

myPatternGreenBW = bwareaopen(myPatternGreenBW,100);

myPatternGreenBW = imfill(myPatternGreenBW,'holes');

figure; imshow(myPatternGreen);

figure; imshow(myPatternGreenBW);

% Blue

[myPatternBlueBW,myPatternBlue] = createPatternBlueMask3(table\_Img);

myPatternBlueBW = bwareaopen(myPatternBlueBW,200);

figure; imshow(myPatternBlue);

figure; imshow(myPatternBlueBW);

% Yellow

[myPatternYellowBW,myPatternYellow] = createPatternYellowMask3(table\_Img);

myPatternYellowBW = bwareaopen(myPatternYellowBW,350);

figure; imshow(myPatternYellow);

figure; imshow(myPatternYellowBW);

1. For all the colours detected, the shape was identified using a similar method as in step 2. For all the shapes detected, the shape, colour, coordinates of centroid and orientation was recorded and returned. A section of the code for processing the red colour is as shown below, the full section is included in Appendix E and the definition of “identifyAllShapes” is shown in Appendix F.

%% See if the colours exists in the pattern

redExists = find(myPatternRedBW);

greenExists = find(myPatternGreenBW);

blueExists = find(myPatternBlueBW);

yellowExists = find(myPatternYellowBW);

% process red

if length(redExists) > 50

cicleAreaThreshold = 1150;

area2Threshold = 720;

imopenSquareDim = 20;

[redShape, redCentroid, redOrientation] = identifyAllShapes(myPatternRedBW, ...

table\_ImgBW, cicleAreaThreshold, area2Threshold);

redCentroidWorld = tablePxlToReal(redCentroid(:,1), redCentroid(:,2));

else

display("No red exists in the pattern");

redShape = [];

redCentroid = [];

redOrientation = [];

redCentroidWorld = [];

end

.

.

.

patternProps.Colour = [ones(size(redShape, 1), 1)\*red; ones(size(greenShape, 1), 1)\*green;...

ones(size(blueShape, 1), 1)\*blue; ones(size(yellowShape, 1), 1)\*yellow];

patternProps.Shape = [redShape; greenShape; blueShape; yellowShape];

patternProps.Centroid = [redCentroidWorld; greenCentroidWorld; blueCentroidWorld; yellowCentroidWorld];

patternProps.Orientation = [redOrientation; greenOrientation; blueOrientation; yellowOrientation];

## **Step 4: Match the block to the pattern (“Main4\_MatchBlocks”)**

Once the shape of the block in the transfer section of the table and the shapes on the desired pattern were identified, it was compared to find a match. Given the list of properties of the shape and list of properties of the pattern, the code found the index of the pattern properties that matched its shape and colour with the shape and colour of the block in the transfer section of the table. The function returned the corresponding coordinates of the centroid and orientation on the pattern.

desiredIdx = find(patternProps.Colour == shapeProps.Colour & patternProps.Shape == shapeProps.Shape);

if length(desiredIdx) > 0

display("Match found!");

coordinates.Centroid = patternProps.Centroid(desiredIdx, :);

coordinates.Orientation = patternProps.Orientation(desiredIdx);

else

display("Match not found... Sorry :(");

coordinates.Centroid = [];

coordinates.Orientation = [];

end

## **Step 5: Remove the matched shape from the list of shapes (“Main5\_RemoveFromList”)**

The purpose of this function was to remove the matched shape from the list of remaining shapes to be matched on the pattern and remove the block from the list of blocks on the conveyor for pick up. The code checked whether there were still items left on the list and if there were, the function compares all the coordinates in the list and finds the index of the closest one to the coordinates taken in by the function (which was passed in to indicate completion of moving the block from the transfer section to the pattern) and removed it from the list.

% remove from patternProps

if size(patternProps.Shape, 2) > 0

distance = sqrt( (patternProps.Centroid(:,1) - coordinatePattern(1)).^2 + (patternProps.Centroid(:,2) - coordinatePattern(2)).^2 );

furthestIdx = find(distance == min(distance));

patternProps.Colour(furthestIdx) = [];

patternProps.Shape(furthestIdx) = [];

patternProps.Centroid(furthestIdx,:) = [];

patternProps.Orientation(furthestIdx) = [];

else

display("The pattern list is empty!");

end

% remove from conveyors list

if size(conveyorList,1) > 0

distance = sqrt( (conveyorList(:,1) - coordinateConveyor(1)).^2 + (conveyorList(:,2) - coordinateConveyor(2)).^2 );

furthestIdx = find(distance == min(distance));

conveyorList(furthestIdx,:) = [];

else

display("The conveyor list is empty!");

end

# Bibliography

|  |  |
| --- | --- |
| [1] | “Why does the REGIONPROPS command not return the correct orientation for a square region?,” [Online]. Available: https://au.mathworks.com/matlabcentral/answers/97460-why-does-the-regionprops-command-not-return-the-correct-orientation-for-a-square-region. [Accessed 15 08 2019]. |
| [2] | “matchFeatures,” [Online]. Available: https://au.mathworks.com/help/vision/ref/matchfeatures.html. [Accessed 12 08 2019]. |
| [3] | “detectFASTFeatures,” [Online]. Available: https://au.mathworks.com/help/vision/ref/detectfastfeatures.html. [Accessed 12 08 2019]. |
| [4] | “detectHarrisFeatures,” [Online]. Available: https://au.mathworks.com/help/vision/ref/detectharrisfeatures.html. [Accessed 12 08 2019]. |
| [5] | “Local feature detection and extraction,” [Online]. Available: https://au.mathworks.com/help/vision/ug/local-feature-detection-and-extraction.html. [Accessed 12 08 2019]. |
| [6] | “Object Detection in a Cluttered Scene Using Point Feature Matching,” [Online]. Available: https://au.mathworks.com/help/vision/examples/object-detection-in-a-cluttered-scene-using-point-feature-matching.html. [Accessed 12 08 2019]. |
| [7] | L. y. Ku, “Object Matching,” [Online]. Available: https://au.mathworks.com/matlabcentral/fileexchange/34626-object-matching. [Accessed 15 08 2019]. |
| [8] | “Matching signs Using SURF,” [Online]. Available: https://au.mathworks.com/matlabcentral/answers/114176-matching-signs-using-surf. [Accessed 15 08 2019]. |
| [9] | “detectMinEigenFeatures,” [Online]. Available: https://au.mathworks.com/help/vision/ref/detectmineigenfeatures.html. [Accessed 12 08 2019]. |
| [10] | “detectORBFeatures,” [Online]. Available: https://au.mathworks.com/help/vision/ref/detectorbfeatures.html. [Accessed 12 08 2019]. |
| [11] | “detectMSERFeatures,” [Online]. Available: https://au.mathworks.com/help/vision/ref/detectmserfeatures.html. [Accessed 12 08 2019]. |

## **Appendix A**

% Auto-generated by cameraCalibrator app on 13-Aug-2019

%-------------------------------------------------------

% Define images to process

imageFileNames = {‘D:\Kevinly Santoso\Documents\UNI\y4s2\MTRN4230\Assignments\Assignment 2\Conveyor\_Calibration\conveyor\_\_08\_07\_18\_46\_52.jpg’,...

‘D:\Kevinly Santoso\Documents\UNI\y4s2\MTRN4230\Assignments\Assignment 2\Conveyor\_Calibration\conveyor\_\_08\_07\_18\_46\_57.jpg’,...

‘D:\Kevinly Santoso\Documents\UNI\y4s2\MTRN4230\Assignments\Assignment 2\Conveyor\_Calibration\conveyor\_\_08\_07\_18\_47\_04.jpg’,...

‘D:\Kevinly Santoso\Documents\UNI\y4s2\MTRN4230\Assignments\Assignment 2\Conveyor\_Calibration\conveyor\_\_08\_07\_18\_47\_17.jpg’,...

‘D:\Kevinly Santoso\Documents\UNI\y4s2\MTRN4230\Assignments\Assignment 2\Conveyor\_Calibration\conveyor\_\_08\_07\_18\_47\_21.jpg’,...

‘D:\Kevinly Santoso\Documents\UNI\y4s2\MTRN4230\Assignments\Assignment 2\Conveyor\_Calibration\conveyor\_\_08\_07\_18\_47\_24.jpg’,...

‘D:\Kevinly Santoso\Documents\UNI\y4s2\MTRN4230\Assignments\Assignment 2\Conveyor\_Calibration\conveyor\_\_08\_07\_18\_47\_33.jpg’,...

‘D:\Kevinly Santoso\Documents\UNI\y4s2\MTRN4230\Assignments\Assignment 2\Conveyor\_Calibration\conveyor\_\_08\_07\_18\_47\_36.jpg’,...

‘D:\Kevinly Santoso\Documents\UNI\y4s2\MTRN4230\Assignments\Assignment 2\Conveyor\_Calibration\conveyor\_\_08\_07\_18\_48\_11.jpg’,...

‘D:\Kevinly Santoso\Documents\UNI\y4s2\MTRN4230\Assignments\Assignment 2\Conveyor\_Calibration\conveyor\_\_08\_07\_18\_48\_17.jpg’,...

};

% Detect checkerboards in images

[imagePoints, boardSize, imagesUsed] = detectCheckerboardPoints(imageFileNames);

imageFileNames = imageFileNames(imagesUsed);

% Read the first image to obtain image size

originalImage = imread(imageFileNames{1});

[mrows, ncols, ~] = size(originalImage);

% Generate world coordinates of the corners of the squares

squareSize = 25; % in units of ‘millimeters’

worldPoints = generateCheckerboardPoints(boardSize, squareSize);

% Calibrate the camera

[cameraParamsConveyor, imagesUsed, estimationErrors] = estimateCameraParameters(imagePoints, worldPoints, ...

‘EstimateSkew’, false, ‘EstimateTangentialDistortion’, false, ...

‘NumRadialDistortionCoefficients’, 2, ‘WorldUnits’, ‘millimeters’, ...

‘InitialIntrinsicMatrix’, [], ‘InitialRadialDistortion’, [], ...

‘ImageSize’, [mrows, ncols]);

% View reprojection errors

%h1=figure; showReprojectionErrors(cameraParamsConveyor);

% Visualize pattern locations

%h2=figure; showExtrinsics(cameraParamsConveyor, ‘CameraCentric’);

% Display parameter estimation errors

displayErrors(estimationErrors, cameraParamsConveyor);

% For example, you can use the calibration data to remove effects of lens distortion.

undistortedImage = undistortImage(originalImage, cameraParamsConveyor);

% See additional examples of how to use the calibration data. At the prompt type:

% showdemo(‘MeasuringPlanarObjectsExample’)

% showdemo(‘StructureFromMotionExample’)

## **Appendix B**

function [BW,maskedRGBImage] = createConveyorMask3(RGB)

%createMask Threshold RGB image using auto-generated code from colorThresholder app.

% [BW,MASKEDRGBIMAGE] = createMask(RGB) thresholds image RGB using

% auto-generated code from the colorThresholder app. The colorspace and

% range for each channel of the colorspace were set within the app. The

% segmentation mask is returned in BW, and a composite of the mask and

% original RGB images is returned in maskedRGBImage.

% Auto-generated by colorThresholder app on 20-Aug-2019

%------------------------------------------------------

% Convert RGB image to chosen color space

I = rgb2ycbcr(RGB);

% Define thresholds for channel 1 based on histogram settings

channel1Min = 34.000;

channel1Max = 65.000;

% Define thresholds for channel 2 based on histogram settings

channel2Min = 110.000;

channel2Max = 125.000;

% Define thresholds for channel 3 based on histogram settings

channel3Min = 134.000;

channel3Max = 150.000;

% Create mask based on chosen histogram thresholds

sliderBW = (I(:,:,1) >= channel1Min ) & (I(:,:,1) <= channel1Max) & ...

(I(:,:,2) >= channel2Min ) & (I(:,:,2) <= channel2Max) & ...

(I(:,:,3) >= channel3Min ) & (I(:,:,3) <= channel3Max);

BW = sliderBW;

% Initialize output masked image based on input image.

maskedRGBImage = RGB;

% Set background pixels where BW is false to zero.

maskedRGBImage(repmat(~BW,[1 1 3])) = 0;

end

## **Appendix C**

function conveyerImg = segmentSection(conveyerImg, xmin, xmax, ymin, ymax)

if xmin > 1

conveyerImg(:,1:xmin,:) = 0;

end

if ymin > 1

conveyerImg(1:ymin,:,:) = 0;

end

if xmax < size(conveyerImg, 2)

conveyerImg(:,xmax:size(conveyerImg, 2),:) = 0;

end

if ymax < size(conveyerImg, 1)

conveyerImg(ymax:size(conveyerImg, 1),:,:) = 0;

end

end

## **Appendix D**

Using 4 coordinates that were already known both in pixel world and real world, a formula was calculated to linearly convert from a pixel coordinate to a real world coordinate.

function realPts = conveyorPxlToReal(xPxl, yPxl)

%% Calibrate the conveyor camera

C2\_world = [20, 146]; % bottom left

C3\_world = [-270, 151]; % top left

C4\_world = [-266, 667]; % top right

C5\_world = [24, 677]; % bottpm right

C2\_pxl = [485, 541];

C3\_pxl = [500, 159];

C4\_pxl = [1181, 165];

C5\_pxl = [1188, 549];

Mx = (C2\_world(1)-C3\_world(1))/(C2\_pxl(2) - C3\_pxl(2));

xReal = Mx\*yPxl - Mx\*C2\_pxl(2) + C2\_world(1);

My = (C5\_world(2)-C3\_world(2))/(C5\_pxl(1)-C3\_pxl(1));

yReal = My\*xPxl - My\*C3\_pxl(1) + C3\_world(2);

display('This is the point you converted to the real world (conveyor): ');

realPts = [xReal, yReal, ones(size(xReal,1),1)\*22.1];

end

## **Appendix E**

%% See if the colours exists in the pattern

redExists = find(myPatternRedBW);

greenExists = find(myPatternGreenBW);

blueExists = find(myPatternBlueBW);

yellowExists = find(myPatternYellowBW);

% process red

if length(redExists) > 50

cicleAreaThreshold = 1150;

area2Threshold = 720;

imopenSquareDim = 20;

[redShape, redCentroid, redOrientation] = identifyAllShapes(myPatternRedBW, ...

table\_ImgBW, cicleAreaThreshold, area2Threshold);

redCentroidWorld = tablePxlToReal(redCentroid(:,1), redCentroid(:,2));

else

display("No red exists in the pattern");

redShape = [];

redCentroid = [];

redOrientation = [];

redCentroidWorld = [];

end

% process green

if length(greenExists) > 50

cicleAreaThreshold = 1115;

area2Threshold = 820;

[greenShape, greenCentroid, greenOrientation] = identifyAllShapes(myPatternGreenBW, ...

table\_ImgBW, cicleAreaThreshold, area2Threshold);

greenCentroidWorld = tablePxlToReal(greenCentroid(:,1), greenCentroid(:,2));

else

display("No green exists in the pattern");

greenShape = [];

greenCentroid = [];

greenOrientation = [];

greenCentroidWorld = [];

end

% process blue

if length(blueExists) > 50

cicleAreaThreshold = 1150;

area2Threshold = 875;

imopenSquareDim = 20;

[blueShape, blueCentroid, blueOrientation] = identifyAllShapes(myPatternBlueBW, ...

table\_ImgBW, cicleAreaThreshold, area2Threshold);

blueCentroidWorld = tablePxlToReal(blueCentroid(:,1), blueCentroid(:,2));

else

display("No blue exists in the pattern");

blueShape = [];

blueCentroid = [];

blueOrientation = [];

blueCentroidWorld = [];

end

% process yellow

if length(yellowExists) > 50

cicleAreaThreshold = 1150;

area2Threshold = 720;

imopenSquareDim = 20;

[yellowShape, yellowCentroid, yellowOrientation] = identifyAllShapes(myPatternYellowBW, ...

table\_ImgBW, cicleAreaThreshold, area2Threshold);

yellowCentroidWorld = tablePxlToReal(yellowCentroid(:,1), yellowCentroid(:,2));

else

display("No yellow exists in the pattern");

yellowShape = [];

yellowCentroid = [];

yellowOrientation = [];

yellowCentroidWorld = [];

end

patternProps.Colour = [ones(size(redShape, 1), 1)\*red; ones(size(greenShape, 1), 1)\*green;...

ones(size(blueShape, 1), 1)\*blue; ones(size(yellowShape, 1), 1)\*yellow];

patternProps.Shape = [redShape; greenShape; blueShape; yellowShape];

patternProps.Centroid = [redCentroidWorld; greenCentroidWorld; blueCentroidWorld; yellowCentroidWorld];

patternProps.Orientation = [redOrientation; greenOrientation; blueOrientation; yellowOrientation];

## **Appendix F**

function [Shape, Centroid, Orientation] = identifyAllShapes(myPatternBW, ...

table\_ImgBW, cicleAreaThreshold, area2Threshold)

circle = 2.1;

flower = 2.2;

diamond = 2.3;

square = 2.4;

star4 = 2.5;

star6 = 2.6;

% identify all the shapes that exist in the pattern for that colour

s = regionprops(myPatternBW, 'Centroid');

centroids = vertcat(s.Centroid);

Shape = [];

Centroid = [];

Orientation = [];

for i = 1: size(centroids,1)

blockBW = removeOtherShapes(table\_ImgBW, centroids(i,:));

shapeBW = removeOtherShapes(myPatternBW, centroids(i,:));

s = regionprops(shapeBW, 'Area', 'Perimeter', 'MajorAxisLength');

if s.Area > cicleAreaThreshold

% circle

Shape = [Shape; circle];

elseif s.Area > area2Threshold

% either flower, diamond or square

if s.Perimeter > 150

Shape = [Shape; flower];

else

shapeBW = rotateToOriginal(shapeBW, blockBW);

seSquare = strel('Square', 25);

shapeBW2 = imopen(shapeBW, seSquare);

a = find(shapeBW2);

if length(a) > 0

Shape = [Shape; square];

else

Shape = [Shape; diamond];

end

end

else

% 4 star or 6 star

if s.MajorAxisLength > 34.2

Shape = [Shape; star4];

else

Shape = [Shape; star6];

end

end

Centroid = [Centroid; centroids(i,:)];

angle = calculateAngle(blockBW);

Orientation = [Orientation; angle];

end

end