PS #6 Report

Problem #1: Object Distinction

Methods

To identify the 5 objects in the image, a combination of contour hierarchy analysis and contour shape detection was used. First, the image was binarized and cleaned up using erosion and dilation techniques in order to make the contour shapes clearer.

```
# Read image
img = cv2.imread(args.input)

# Convert to gray-scale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Binary
thr,dst = cv2.threshold(gray, 60, 255, cv2.THRESH_BINARY)

# clean up
for i in range(1):
    dst = cv2.erode(dst, None)
for i in range(4):
    dst = cv2.dilate(dst, None)
```

Figure 1: Contour detection preparation techniques included binarization and erosion/dilation.

Next, contours were detected using the cv.findContours() method. This method returns a list of contours as well as a hierarchy array, which provides information about which contours are within other contours. Using this information, the following decision tree was used to identify each of the 5 objects.

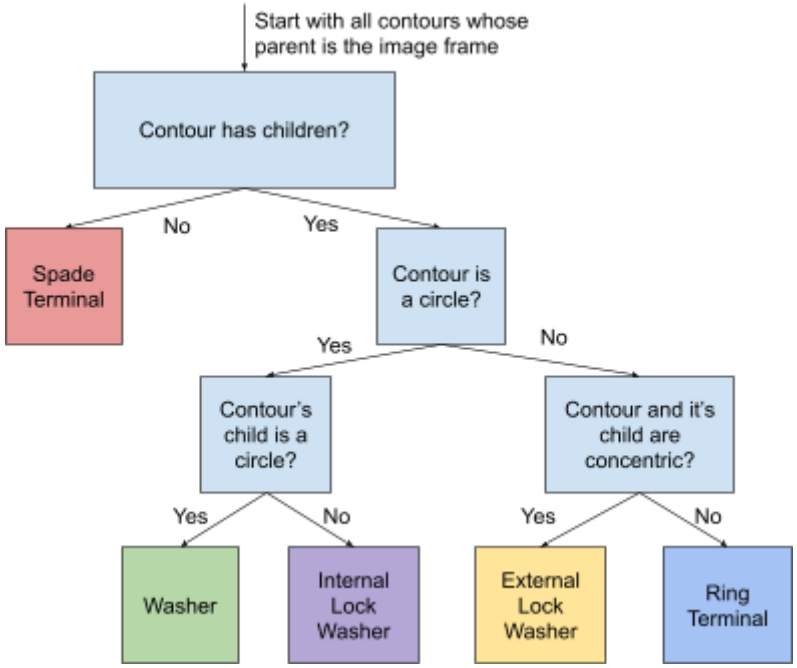

Figure 2: Decision tree for identifying object type based on contour shape and hierarchy.

```python
# find contours with hierachy
cont, hier = cv2.findContours(dst, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
# each contoure
for i in range(len(cont)):
    c = cont[i]
    h = hier[0,i]
    if h[2] == -1 and h[3] == 0:
        # no child and parent is image outer
        img = cv2.drawContours(img, cont, i, (0,0,255),-1)
    elif h[3] == 0 and hier[0,h[2]][2] == -1:
        # with child
        if isCircle(c):
            if isCircle(cont[h[2]]):
                # double circle
                img = cv2.drawContours(img, cont, i, (0,255,0),-1)
            else:
                # outer contour circle, inner contour not circle
                img = cv2.drawContours(img, cont, i, (255, 0, 255), -1)
        else:
            # 1 child
            if hier[0,h[2]][0] == -1 and hier[0,h[2]][1] == -1:
                # find centroid of the contour and its child
                M = cv2.moments(c)
                cX = int(M["m10"] / M["m00"])
                cY = int(M["m01"] / M["m00"])
                M_int = cv2.moments(cont[h[2]])
                cX_int = int(M_int["m10"] / M_int["m00"])
                cY_int = int(M_int["m01"] / M_int["m00"])
                # if parent and child contours are concentric
                if abs(cX - cX_int) <= 2 and abs(cY - cY_int) <= 2:
                    img = cv2.drawContours(img, cont, i, (0, 255, 255), -1)
                else:
                    img = cv2.drawContours(img, cont, i, (255,0, 0),-1)
```

Figure 3: Implementation of the decision tree in Python.

Results

After applying the previously described logic to the image, the following output was produced, correctly identifying every object in the image.

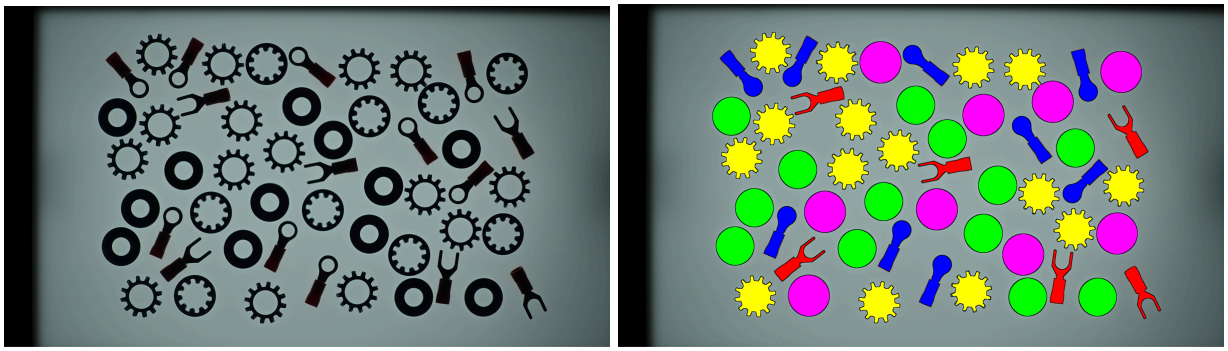

Figure 4: The original image (left) and the output with the objects identified by color (right). Each object in the original image was correctly identified by the method described above.

Problem #2: Shape Matching for Defect Detection

Methods

To identify defected spade terminals in the provided image, a template of a healthy terminal was selected and compared to all contours in the image using the cv.matchShapes() method. First, the image was binarized and cleaned up using erosion and dilation.

```
# Read image and prepare for contour detection
img = cv.imread("ps6-2/spade-terminal.png")
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
thr,dst = cv.threshold(gray, 60, 255, cv.THRESH_BINARY)

for i in range(1):
    dst = cv.erode(dst, None)
for i in range(1):
    dst = cv.dilate(dst, None)
```

Figure 5: Image preparation for contour detection.

Next, the contours and hierarchy were detected using the cv.findContours() method. By visual inspection, one of these contours was selected to be the "template contour" which will be the reference for comparison used to determine whether or not a spade terminal is defective.

```
# Find contours and select one to be the template for a non-defective terminal (done by visual inspection)
cont, hier = cv.findContours(dst, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)
template = cont[1]
```

Figure 6: Contour and hierarchy detection and template selection.

Each childless contour was compared to the template contour using cv.matchShape(). This method returns a "distance" which can be treated like a "similarity score". Higher scores mean the two contours are more dissimilar. To identify detective terminals, a threshold was set on the similarity score. After some tuning, a threshold value of 1.0 was found to produce accurate results.

```
# Loop through every contour
for i in range(len(cont)):
    c = cont[i]
    h = hier[0, i]
    # Check only contours with no children and who's parent is the image frame
    if h[2] == -1 and h[3] == 0:
        # Find the similarity with the template
        d = cv.matchShapes(c, template, cv.CONTOURS_MATCH_I2, 0)
        # If the similarity value is too high, highlight the terminal as defective
        if d > similarity_thresh:
            img = cv.drawContours(img, [c], -1, (0,0,255), -1)
```

Figure 7: Implementation of the template matching algorithm for detecting defective terminals.

Results

Using the algorithm described above, defective terminals were identified in the original image with a 100% success rate. If another image were to be provided, the threshold value may need to be retuned. Methods to improve robustness could include comparing contour areas or perimeters.

Figure 8: The original image (left) and the resulting image (right) after applying the defect detection algorithm via template matching. All defective terminals were correctly identified with no false positives or false negatives.