Will Freeman
EE 384
4/29/2021

Final MATLAB Project: Key Decoder

Introduction

In this project, I was able to create a MATLAB signal processing algorithm to decode a key from a photo. All keys have what is called a bitting code: a sequence of numbers representing cut depths that can be used to recreate the key. I use MATLAB's image processing features as well as my own signal processing to identify the key, cut depths, and decode them to a bitting code, which is printed in text on top the picture. The particular key this is designed for is a Kwikset KW1, the most popular home key in North America. This can be extended very easily to work with other key types.

Theory

This project uses many concepts covered in this class, as well as some not covered in the class. The concepts covered in class used in this project are reading and writing images, edge detection, noise reduction using filters, and color space conversions. Edge detection is the process of identifying changes in pixel intensity, usually associated with what we see as "edges". There are many edge detection algorithms, but for this project, I chose Canny edge detection since it produced the most detailed results. Noise reduction is the process of removing high frequency changes in pixel intensity. If applied in a large magnitude, it is considered blurring. Noise reduction is usually one of the first steps in image processing because images from cameras contain more detail than practical for identifying objects. We use a bilateral filter to blur the image so that the background can be easily ignored. Bilateral filters have the effect of blurring while preserving prominent edges, which is exactly what we want for this use case. Last, color space conversion is the conversion of one color space to another. In this case, our input image is an RGB image, but we do not care about color. To simplify our computation, we convert it to grayscale, so that only a single dimension of intensity is represented.

Some concepts used in the project that were not covered in this class are morphological transformations and Hough transforms. Morphological transformations are simple image processing operations on binary images using kernels. The two most basic morphological operations are erosion and dilation. Erosion erodes the boundaries of a binary image, keeping pixels with a 1 value only if all pixels within the kernel are 1. Dilation is the opposite, where all pixels within the kernel are set to 1 if at least 1 pixel has value of 1. The next two popular operations are opening and closing, which are combinations of erosion and dilation. Opening is erosion followed by dilation, which has the effect of removing stray pixels. Closing is dilation

followed by erosion, which has the effect of filling holes and connecting slightly disconnected components within a binary image. Since edge detection sometimes leaves gaps, a morphological close operation is used in this project to fill those gaps. Finally, a Hough transform is an algorithm which extracts line segments from a group of pixels which resemble a line. Since the key blade consists of a top edge, bottom edge, and a vertical line near the shoulder of the key, a Hough transform is used to detect the bounds of the key blade to assist in decoding.

Simulation

Important Constraints

- background should be relatively uniform
- camera should be normal to plane of key surface
- not many shadows present
- key should be oriented with cuts facing up and tip facing left

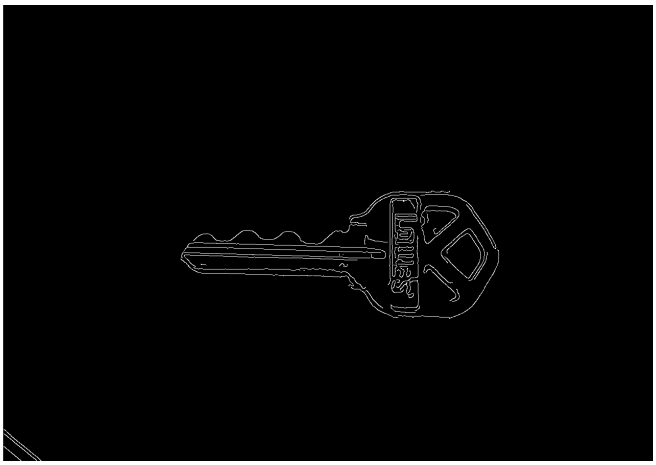
1. **read** image

2. **resize** image
    a. resize image so that the operations involving fixed-size kernels will have consistent effects regardless of image size

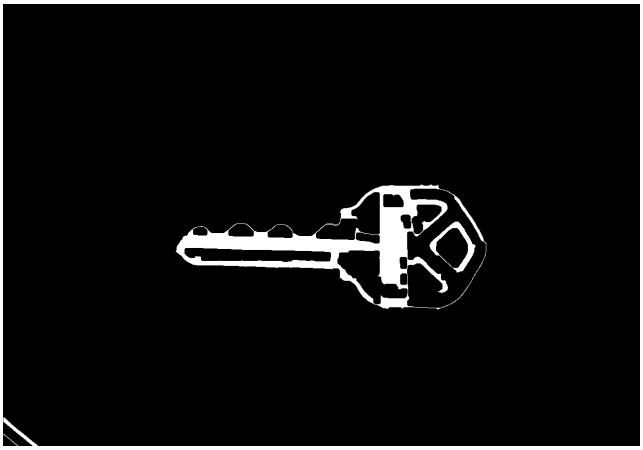
3. convert to **grayscale**
    a. we do not use color for decoding the key, since a key can be any color

4. use **bilateral** filter to smooth image while maintaining edges
    a. the bilateral filter helps prepare the image for edge detection by reducing false edges, such as a background with wood grain

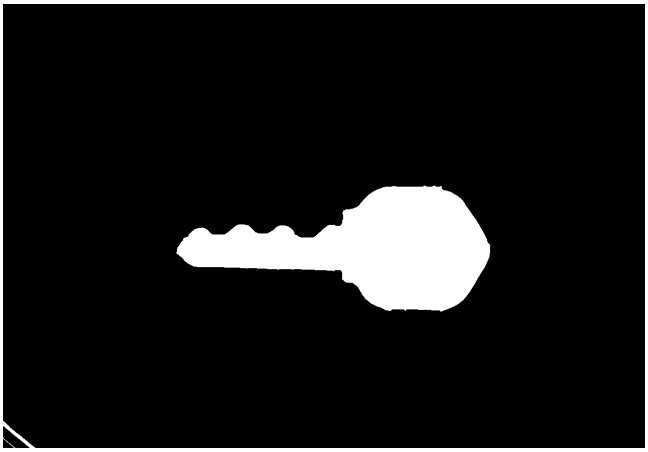
5. **Canny** edge detection
    a. Canny edge detection is one of the most effective edge detection methods and yielded the most detailed edges in testing
    b. Canny edge detection was a good candidate for key decoding because as many edges as possible need to be detected in order to accurately find the difference between the key cut and the bottom of the key blade



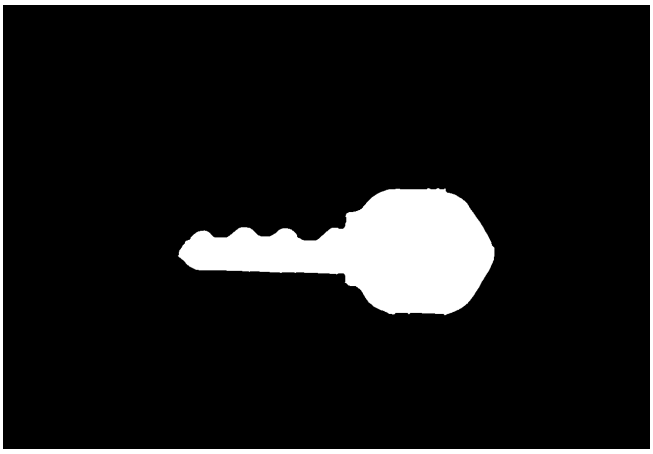
6. morphological close operation
    a. edge detection occasionally leaves gaps where the lighting is not perfect
    b. a close operation fills these gaps and results in a fully-closed shape



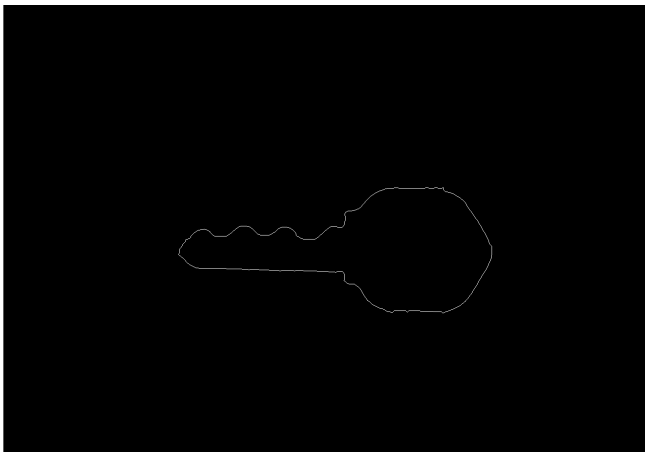
7. fill all contours
    a. edge detection can find other objects in the image, so filling all contours allows us to filter for the largest shape



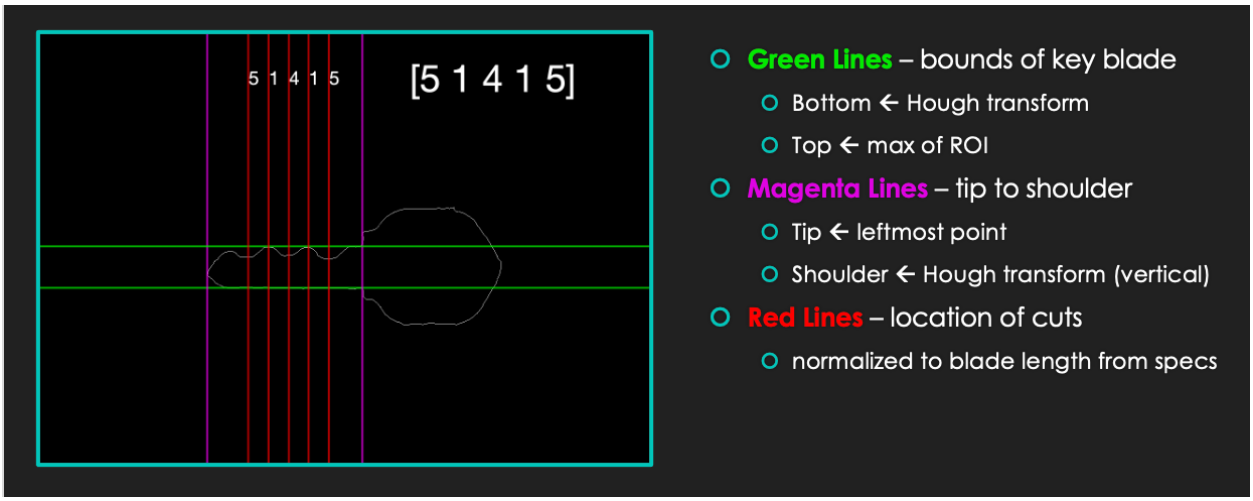
8. remove all but the largest connected component
    a. the key should be the largest object in the picture, so removing everything smaller than the largest object should leave only the key remaining in the resulting binary image



9. find outer contour
    a. finding the contour of the key allows us to easily identify the boundaries of the key
    b. knowing the boundaries of the key allows us to calculate the difference between the cut and bottom of the blade



The following figure represents the rest of the steps, color coded.

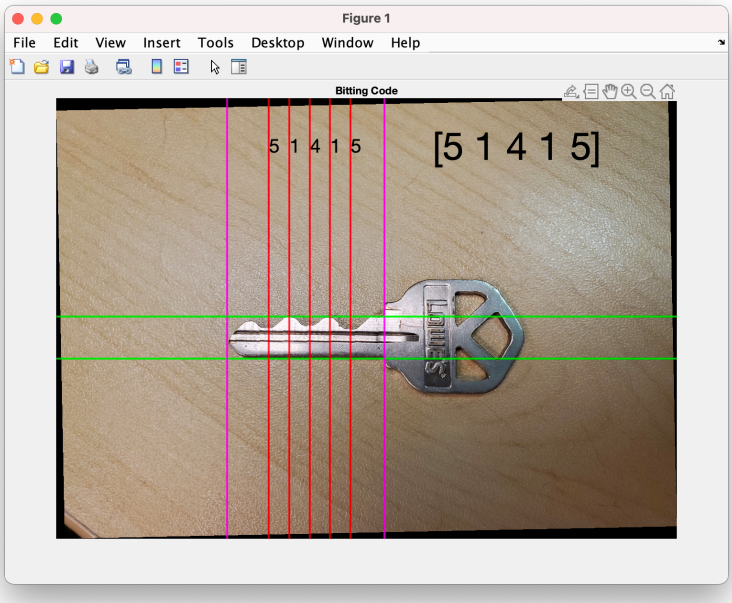


10. use Hough transform to find bottom edge of blade
    a. a horizontally-constrained Hough transform will identify the lower edge of the blade as its longest line segment
    b. the top edge will be ignored since a cut key is not flat on top
11. rotate image until Hough line is horizontal
    a. the key may not be perfectly straight, so rotation is necessary for the next steps where the cut y-position is subtracted from where the top edge should be
12. use Hough transform to find shoulder
    a. the shoulder is one of the bounds of the key blade

    b. we will use the bounds to determine where the cuts should be on the key
13. mark minimum X value as tip of key
    a. the tip of the key, if oriented as specified in the constraints for using this script, will be the far leftmost pixel
    b. this value serves another boundary for the key blade
14. create region of interest for key blade
    a. a region of interest (ROI) for the key blade allows easier calculations of the cut depth without paying attention to the decorative parts of the key
15. find the mode of the height of each key cut
    a. cuts are not perfect, but they are usually close to perfect
    b. using the mode rather than the mean (of a range of y values around where the cuts should be) ensures that even when the width of the cuts is inconsistent that it will not skew the results
16. compare the normalized cut depth with the normalized cut specs to obtain bitting code
    a. the KW1 cut specs, as shown in the powerpoint are normalized to a percentage of 1
    b. likewise, the cut depths calculate using image processing are normalized to a percentage of 1
    c. the value of the image cut depth is compared to each of the cut depths, producing an error (deviation) for each value
    d. the index with the lowest error is considered the official bitting value for that position
17. write bitting code over image
    a. bitting code is interpreted as the bitting value calculated at each position, read from shoulder to tip
    b. bitting code is overlaid on image

Result

The following figure shows a successfully decoded KW1 key, verified with the original bitting code. The color codes are the same as the previous figure.



Appendix

All code will be submitted as source files along with this report, but it will also be included in this section of the document. There is one main script with 2 user-defined functions.

decoder.m (main)

```
close all;
clear;
clc;

% imgFile = 'kw1.jpg';
% imgFile = 'kw1-blank.jpg';
imgFile = 'kw1-house.jpg';

% resize to familiar size (for use with my kernels)
mat = imread(imgFile);
[w, h] = size(mat);
mat = imresize(mat, [895 1280]);

% convert to grayscale
gray = rgb2gray(mat);
```

```
% smooth out noise while preserving edges
smoothed = imbilatfilt(gray, 0.5*diff(getrangefromclass(gray).^2), 2);

% find edges
edges = edge(smoothed, 'Canny', [0.05 0.20]);

% figure; imshowpair(smoothed, edges, 'montage');

% fill gaps with morphological close (dilation followed by erosion)
se = strel('disk', 7);
closed = imclose(edges, se);

% fill contours
filled = imfill(closed, 'holes');

% keep only largest object
keyonly = bwareafilt(filled, 1);

% find outer contour for line identification
outerLoop = bwperim(keyonly);

% find bottom of blade using hough
[H, T, R] = hough(outerLoop);
P = houghpeaks(H, 1, 'Threshold', ceil(0.1 * max(H(:))));
lines = houghlines(outerLoop, T, R, P, 'FillGap', 200, 'MinLength', 50);
bottomOfBladePreRotation = lines(1);

% rotate key to be straight
rotated = straightenToLine(outerLoop, bottomOfBladePreRotation);

% display image
figure; imshow(straightenToLine(mat, bottomOfBladePreRotation));
title('Bitting Code');
% figure; imshow(rotated); title('Bitting Code');

% find new bottom of blade
[H, T, R] = hough(rotated);
P = houghpeaks(H, 1, 'Threshold', ceil(0.1 * max(H(:))));
lines = houghlines(rotated, T, R, P, 'FillGap', 200, 'MinLength', 50);
bb = lines(1);

bottomBladeY = (bb.point1(2) + bb.point2(2)) / 2;
yline(bottomBladeY, 'LineWidth', 2, 'Color', 'green');



% find blade X limits

verticalTheta = -3:3; % theta values from -10 to +10

% hough transform to identify vertical lines
[H, T, R] = hough(rotated, 'Theta', verticalTheta);
P = houghpeaks(H, 4, 'Threshold', ceil(0.1 * max(H(:))), 'Theta',
verticalTheta);
lines = houghlines(rotated, T, R, P, 'FillGap', 500, 'MinLength', 10);
```

```
% leftmost line is the shoulder, find its midpoint
shoulderX = w;
for p = 1:length(lines)
    x = [lines(p).point1(1), lines(p).point2(1)];
    if (x(1) < shoulderX)
        shoulderX = x(1);
    end
    if (x(2) < shoulderX)
        shoulderX = x(1);
    end
end

xline(shoulderX, 'LineWidth', 2, 'Color', 'magenta');

% leftmost point is tip
[r,c] = find(rotated);
tipX = min(c);
xline(tipX, 'LineWidth', 2, 'Color', 'magenta');




% create ROI for top of blade detection
roi = rotated(1:bottomBladeY, tipX:shoulderX-10); % -10 for clearance
[r,c] = find(roi);
topOfBladeY = min(r);

yline(topOfBladeY, 'LineWidth', 2, 'Color', 'green');


bladeBounds = [tipX shoulderX; bottomBladeY topOfBladeY];
[bitting, pinPositions] = decodeKW1(rotated, bladeBounds);



% draw bitting on key
for p = 1:length(pinPositions)
    xline(pinPositions(p), 'LineWidth', 2, 'Color', 'red');
    text(pinPositions(p), 100, int2str(bitting(p)), 'fontsize', 20, 'color',
'white');
end

text(shoulderX + 100, 100, mat2str(bitting), 'fontsize', 40, 'color',
'white');
```

decodeKW1.m

```
function [bitting, pinPositions] = decodeKW1(mat, bladeBounds)

% find normalized pin positions
pinPositions = zeros(1,5);
for p = 1:length(pinPositions)
    pinPositions(p) = round(bladeBounds(1,2) - ((0.247 + 0.15*(p-1)) *
(bladeBounds(1,2) - bladeBounds(1,1)) / 1.15));
end

% find mode y value for each pin (-5:+5)
cutDepths = zeros(1,5);
for p = 1:length(pinPositions)
    x = pinPositions(p);
    roi = mat(bladeBounds(2,2):bladeBounds(2,1)-5, x-5:x+5);
    [r,c] = find(roi);
    cutDepths(p) = mode(r);
end

% bitting specs
normalizedBittingSpecs = 1 - ([.329, .306, .283, .260, .237, .214, .191] /
.335);

% normalized cut depths
normalizedCutDepths = cutDepths / (bladeBounds(2,1) - bladeBounds(2,2));

% calculate bitting code with smallest error
bitting = zeros(1,5);
for p = 1:length(pinPositions)
    error = abs(normalizedBittingSpecs - normalizedCutDepths(p));
    [m, i] = min(error);
    bitting(p) = i;
end


end
```

straightenToLine.m

```
function rotated = straightenToLine(mat, line)
    rotationDeg = rad2deg(atan2(line.point2(2) - line.point1(2),
line.point2(1) - line.point1(1)));
    rotated = imrotate(mat, rotationDeg);
end
```