Mario Levya

Joseph Ntaimo

Lacthu Vu

Etch-a-Sketch

Introduction

The goal of our project was to model an Etch-a-Sketch drawing in MATLAB by taking in an image and returning the drawing that would be created by the toy. We initially considered the possibility of programming all the way from an image to generating motor controls to draw the image on an actual Etch-a-Sketch, but decided that a better use of our time would be to focus on determining the optimal method for tracing the edges of the image to imitate an Etch-a-Sketch toy.

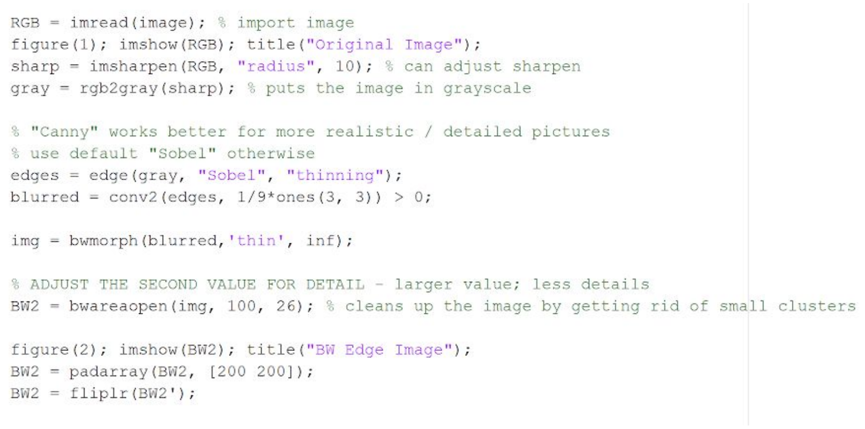
We considered two possible methods:

1. **Rasterizing:** "Shading" in the image by drawing lines back and forth, altering the distance between the lines to simulate different shades.
2. **Tracing:** Locating the edges in the image and attempting to trace them in a single continuous line in the likeness of the original image.

Rasterizing seemed the most simple at first, since we would not have to worry about controlling the direction the line goes beyond left and right, but we decided to trace the image instead in hopes of producing a better quality picture faster.

Method

I. Preprocessing Image:



```
RGB = imread(image); % import image
figure(1); imshow(RGB); title("Original Image");
sharp = imsharpen(RGB, "radius", 10); % can adjust sharpen
gray = rgb2gray(sharp); % puts the image in grayscale

% "Canny" works better for more realistic / detailed pictures
% use default "Sobel" otherwise
edges = edge(gray, "Sobel", "thinning");
blurred = conv2(edges, 1/9*ones(3, 3)) > 0;

img = bwmorph(blurred,'thin', inf);

% ADJUST THE SECOND VALUE FOR DETAIL - larger value; less details
BW2 = bwareaopen(img, 100, 26); % cleans up the image by getting rid of small clusters

figure(2); imshow(BW2); title("BW Edge Image");
BW2 = padarray(BW2, [200 200]);
BW2 = fliplr(BW2');
```

To preprocess the image so that we could then turn it into edges that we could plot, we used the following methods from MATLAB's Image Processing Toolkit:

A. Import the image using imread().
B. Sharpen image using imsharpen() to increase the distinction between colors before detecting edges. This allows the algorithm to be more sensitive to differences in colors.
C. Change the image to grayscale using rgb2gray().
D. Detect the edges using a Sobel or Canny matrix in the built in edges function. We found that Canny edge detection worked best for more complex or detailed figures, as it would filter out details so only more significant edges remained.
E. Blur the image with a 3 x 3 matrix of 1/9 on each pixel using conv2().
F. Remove small clusters using bwareaopen(). This would further eliminate unnecessary details by eliminating clusters that were smaller than a certain number of pixels in a certain number of directions.

G. Pad the image with whitespace zeros using padarray() to make sure that the algorithm does not exceed the bounds of the image and throw an error.

H. Flip the transpose of the image using fliplr() because the line generation algorithm does not maintain the original orientation of the image.

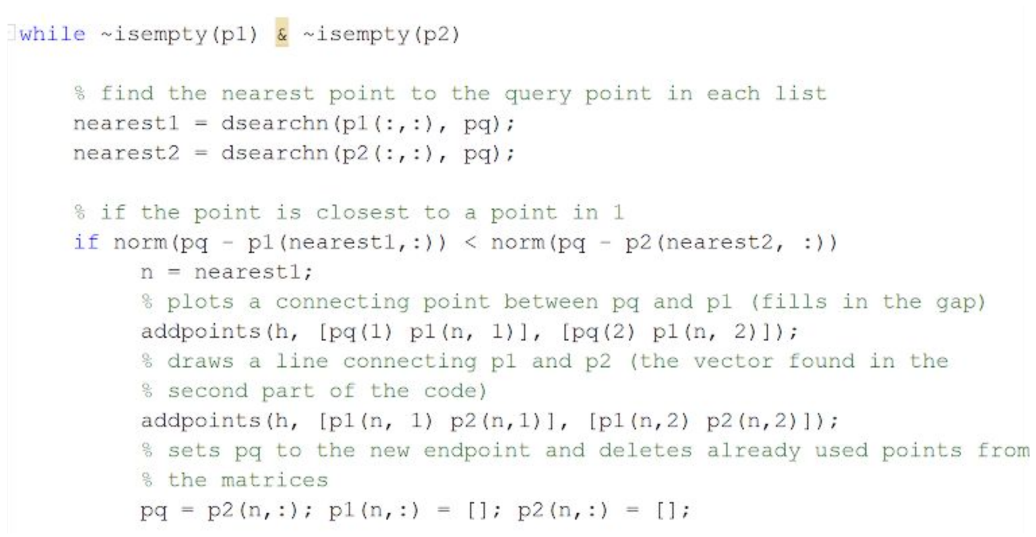
II. Line Generation Algorithm:

Given the preprocessed image, consisting of 1's and 0's our goal was to determine the most optimal set of lines which when connected via the Tracing Algorithm (discussed in part III). To do this, we decided that finding the set of longest lines would help us trace effectively. Therefore, in order to find a long line, we iterated through all of the pixels of the image, finding black pixels (or 1's in the image binary matrix). Once we find a black pixel, we would determine how long a line in 16 different directions could be made. That is, it searches outward to find continuous sets of black pixels. Since we are only interested in longer lines, we also included a threshold of how many black pixels had to be found to see whether a sufficiently long line could be made beginning at the pixel of interest. The 16 directions included the cardinal and intercardinal directions, and 8 auxiliary directions. The auxiliary directions were offset from the original 8, and seemed to optimize line generation, and including more directions would theoretically increase accuracy.

If a sufficiently long line is found, then the pixels are erased (converted to a 0, or white pixel) from the image, to prevent the pixels from the line from being found again. Then we proceed, to create more line segments from the ending location of the first line segment. This process helps create long continuous lines. If there are no more line segments that can be added to a chain, the algorithm begins again, starting with an unused black pixel.

The output of the algorithm is an (*n* by 4) matrix, where *n* corresponds to the number of lines generated and four integers corresponding to the x, y coordinates for both points defining a line, that is x_1, y_1, x_2, y_2.

III. Tracing Algorithm



```
while ~isempty(p1) & ~isempty(p2)

    % find the nearest point to the query point in each list
    nearest1 = dsearchn(p1(:,:), pq);
    nearest2 = dsearchn(p2(:,:), pq);

    % if the point is closest to a point in 1
    if norm(pq - p1(nearest1,:)) < norm(pq - p2(nearest2, :))
        n = nearest1;
        % plots a connecting point between pq and p1 (fills in the gap)
        addpoints(h, [pq(1) p1(n, 1)], [pq(2) p1(n, 2)]);
        % draws a line connecting p1 and p2 (the vector found in the
        % second part of the code)
        addpoints(h, [p1(n, 1) p2(n,1)], [p1(n,2) p2(n,2)]);
        % sets pq to the new endpoint and deletes already used points from
        % the matrices
        pq = p2(n,:); p1(n,:) = []; p2(n,:) = [];
```

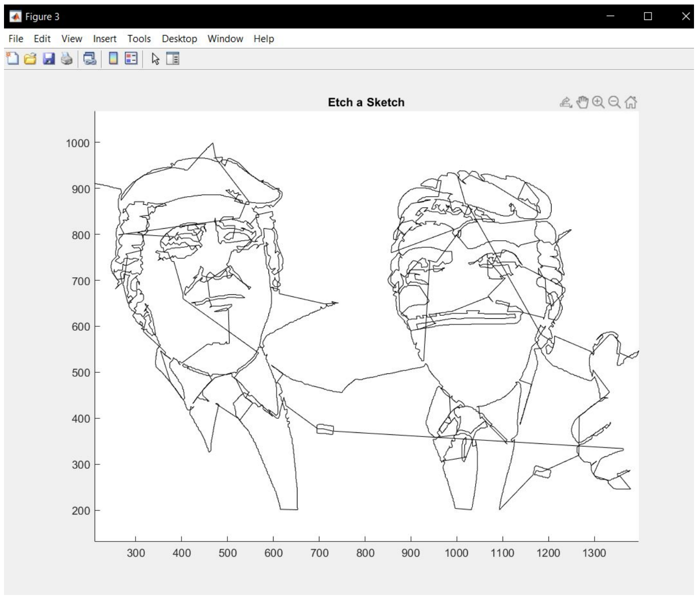
At the end of the second part of our method, we had a series of prominent edges drawn out, which we would have to connect together in order to get one line, similar to the tracing method done by the toy. Knowing the vectors from the previous part of our method, we would be able to draw our image using the *animatedline* function of MATLAB, which would make our plot seem like it was drawn in real time on an Etch-a-Sketch.

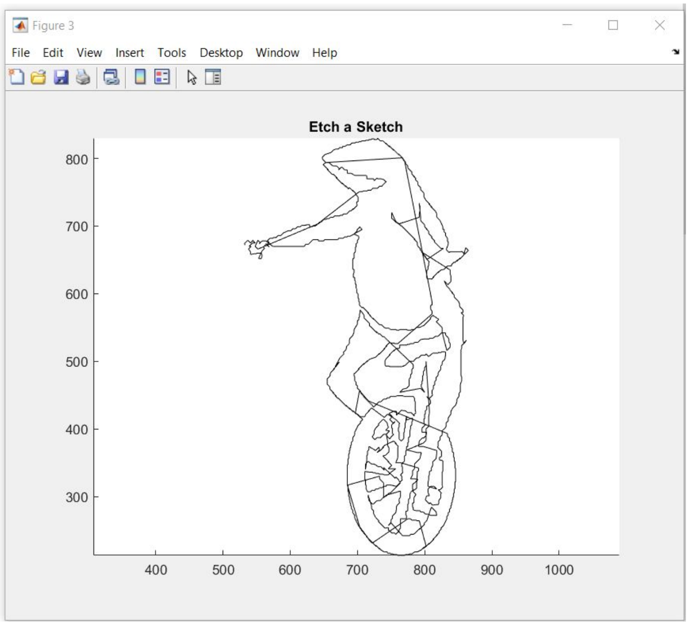
From the vectors *vec* in part II, we split these vectors into endpoint coordinates *p1* and *p2*. From here, we started with the first query point *pq = p1(1,:)*, which was arbitrarily chosen, and found the nearest point to this point from both lists. If a point from *p1* was closer to the query point, we would connect it to the nearby point, then trace out the vector associated with *p1* and its corresponding *p2* and remove the endpoints of that vector from both lists. Otherwise, we would do this with the point from *p2*, connecting the query point to this nearest point in *p2* and then drawing out the vector. We would continue finding closest points until both lists *p1* and *p2* were exhausted, at which point all of the individual vectors would have been connected together into one image.

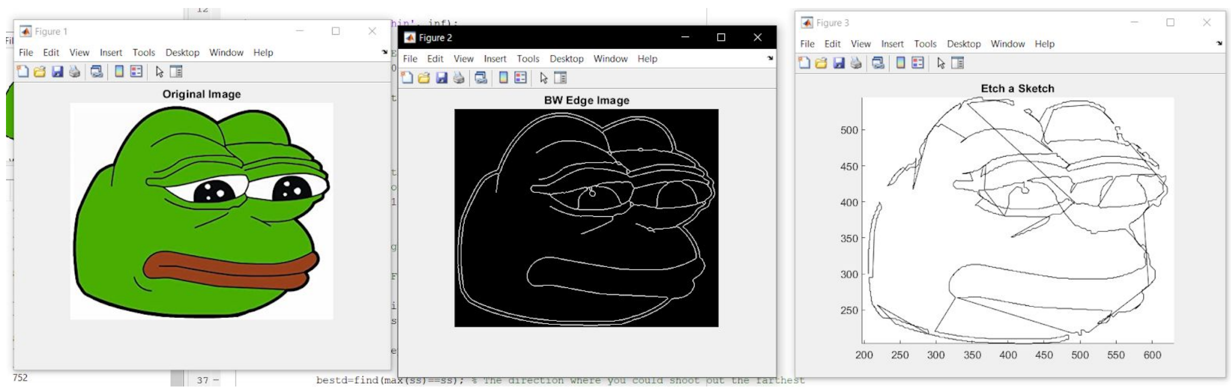
Considerations

The edge detection methods that we used in our preprocessing had a significant effect on how the resulting Etch-a-Sketch plots looked like; while using Canny detection worked better for more realistic or detailed images (such as people's faces), simpler images such as pixel art could be traced out much better using the Sobel method. Our line generating algorithm was limited by the number of directions that it could trace, which parallels the function of an Etch-a-Sketch, but more refined images could be made by allowing our algorithm to trace out more lines with different slopes. Lastly, the tracing algorithm at the end connected nearby vectors, which, while providing one connected line, often ran into the error of giving longer lines that connected across the image. This could possibly be improved by a better selection of an initial query point, which would minimize the total distance that the Etch-a-Sketch traced (resulting in much greater computation cost) or by implementing a method that could trace backwards across already used endpoints to find an even closer nearby point. Along with attempting to create this image on an actual Etch-a-Sketch with stepper motors, these limitations could be considered in trying to improve our MATLAB script.

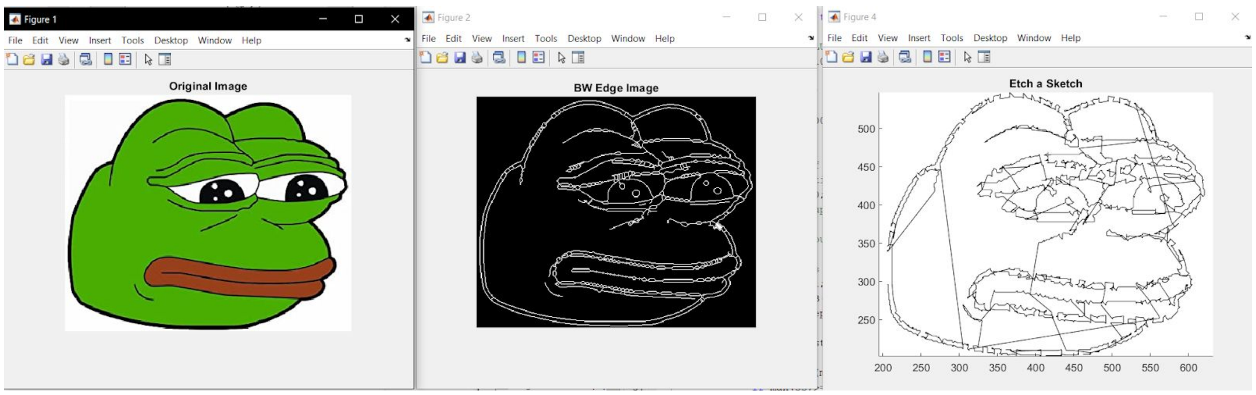
Images



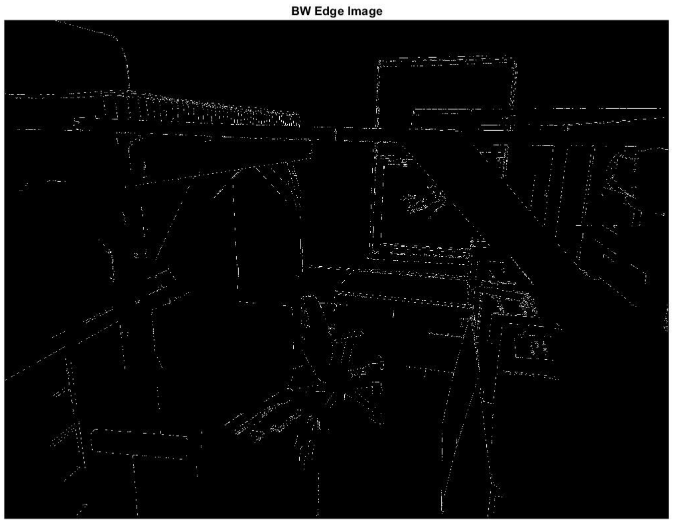


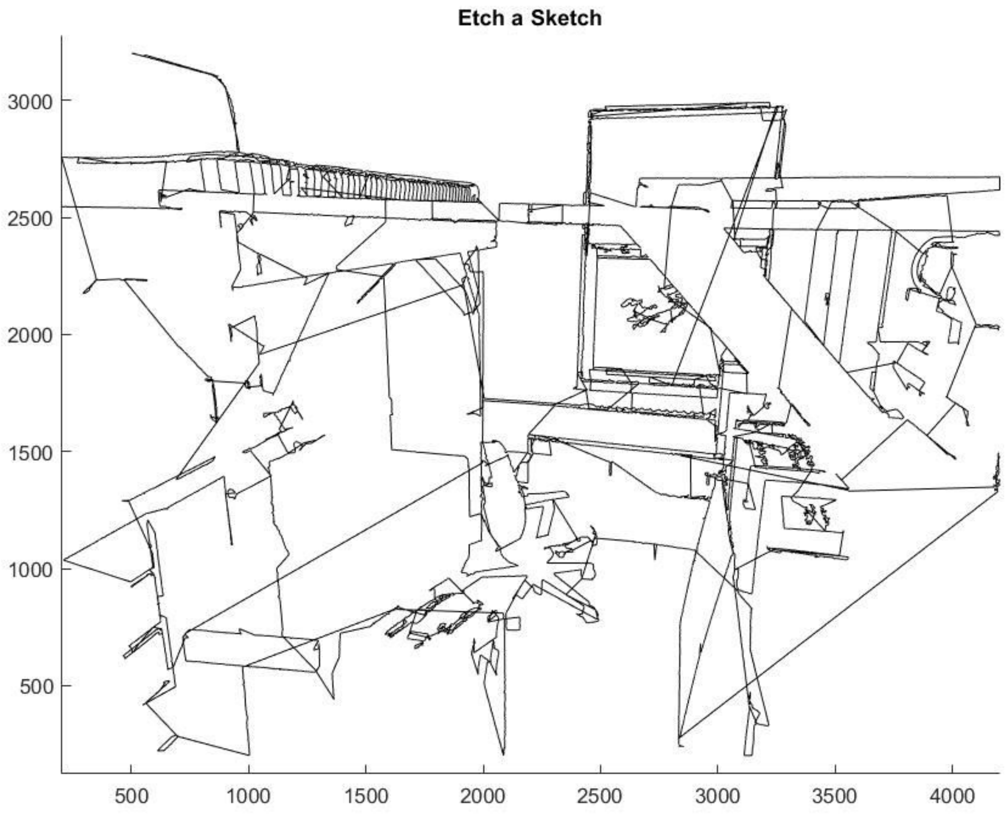


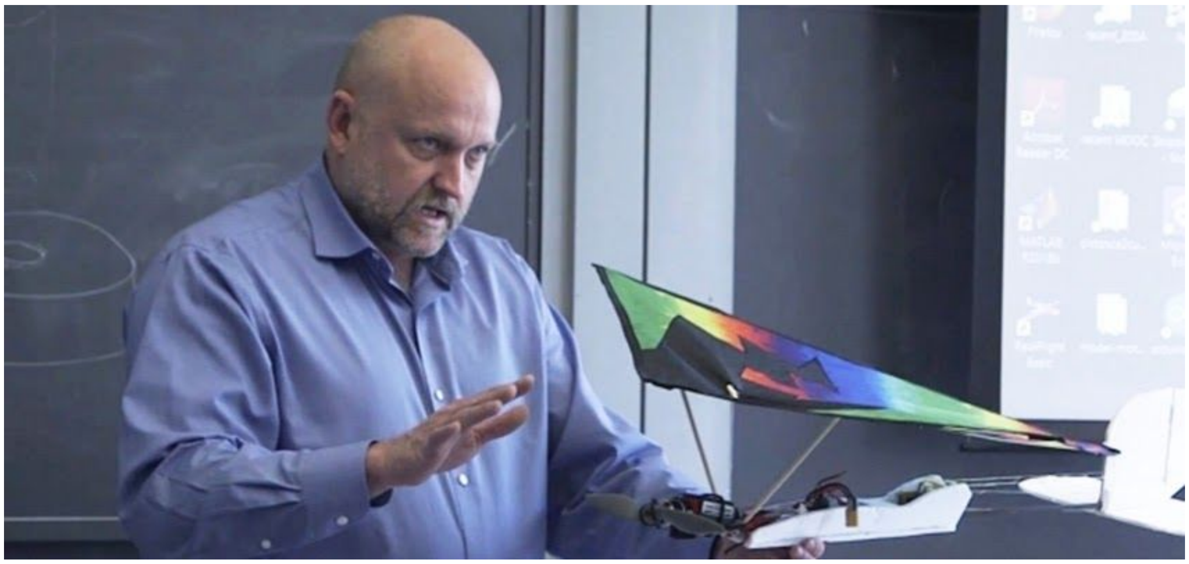
```
bestd=find(max(SS)==SS); % The direction where you could shoot out the farthest
```



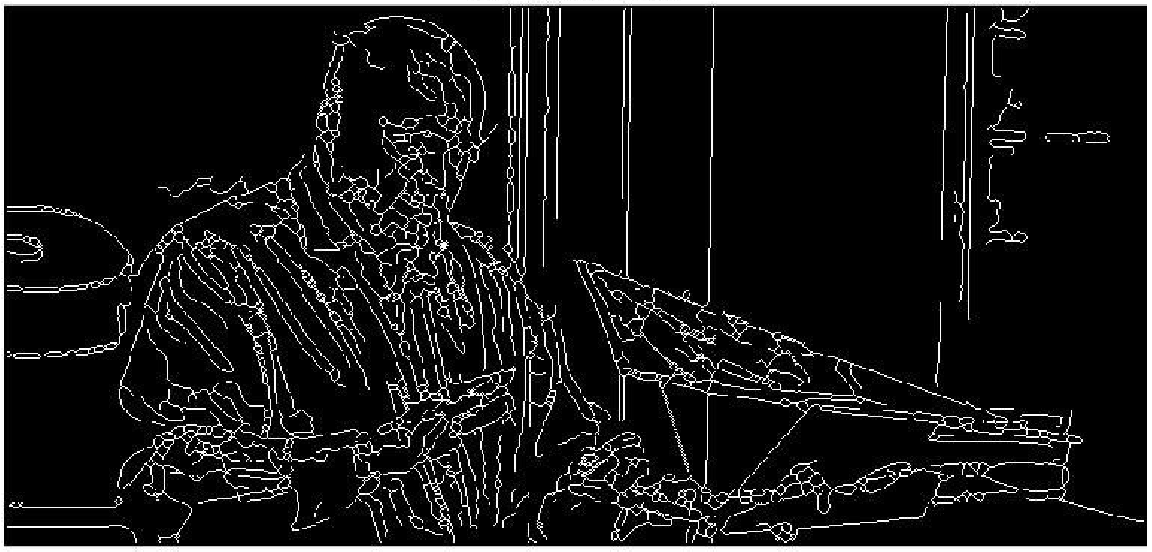
U-HAUL
100% BUY-BACK GUARANTEE


BW Edge Image

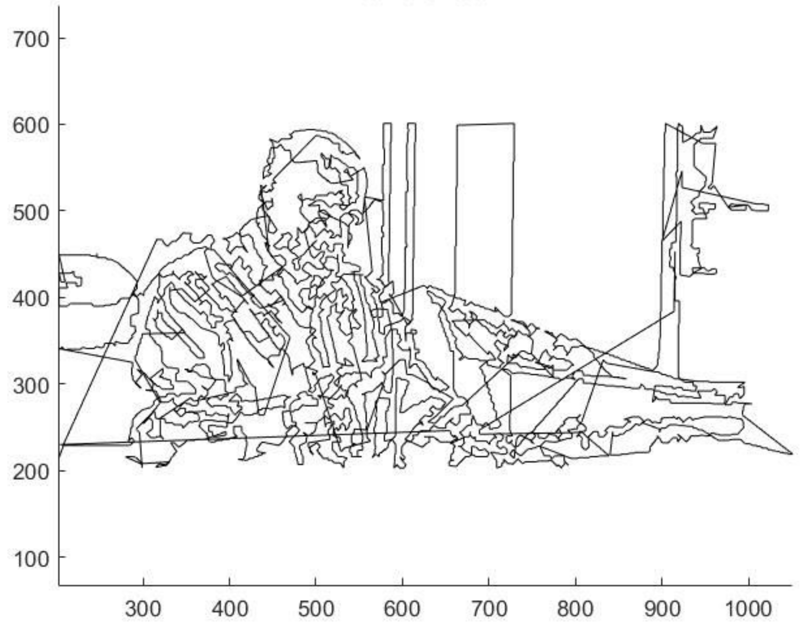

Etch a Sketch
3000
2500
2000
1500
1000
500
500
1000
1500
2000
2500
3000
3500
4000

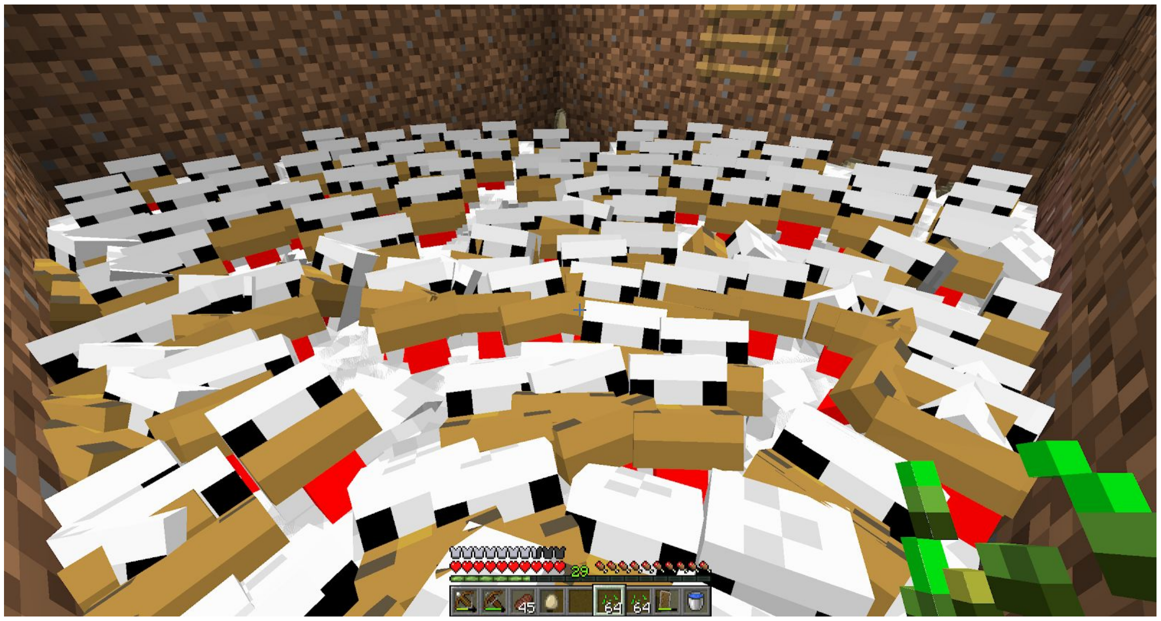


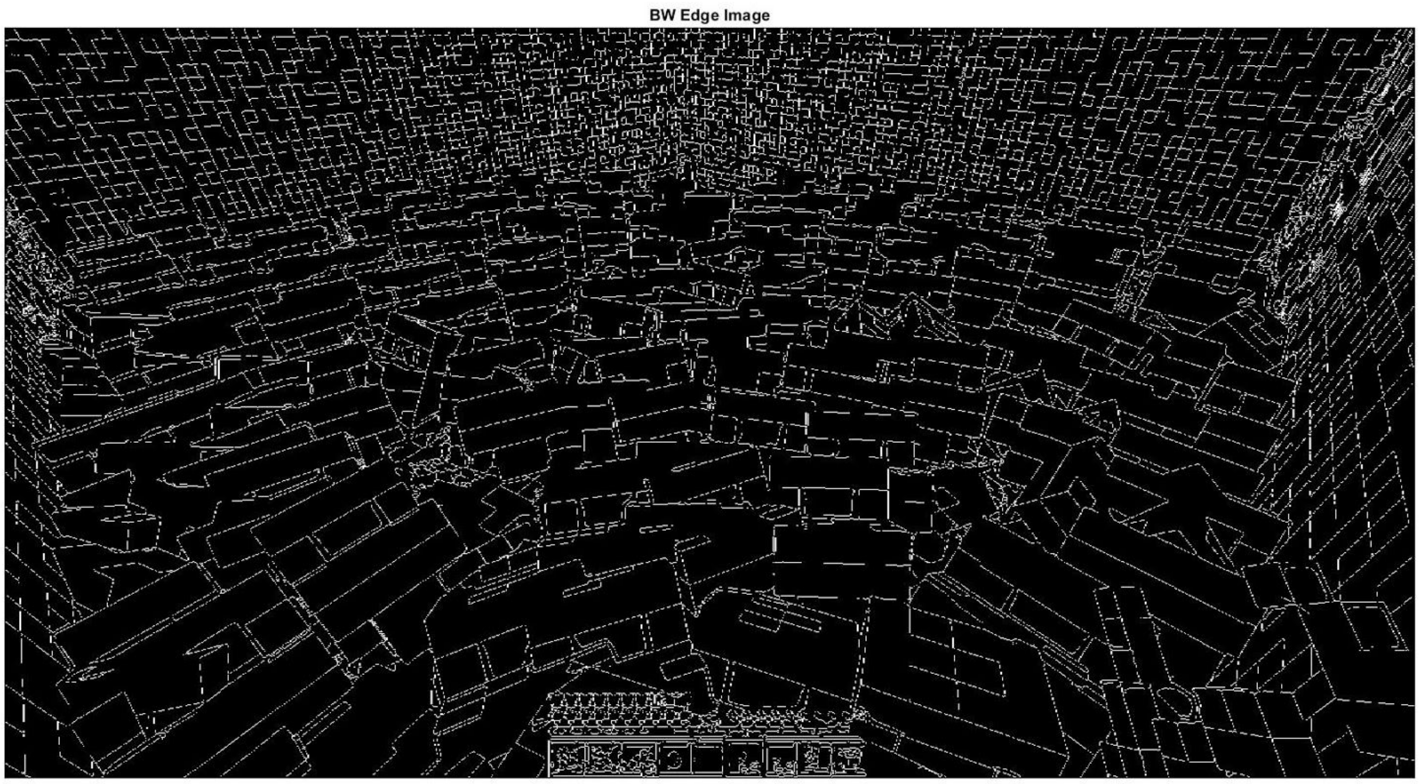
BW Edge Image

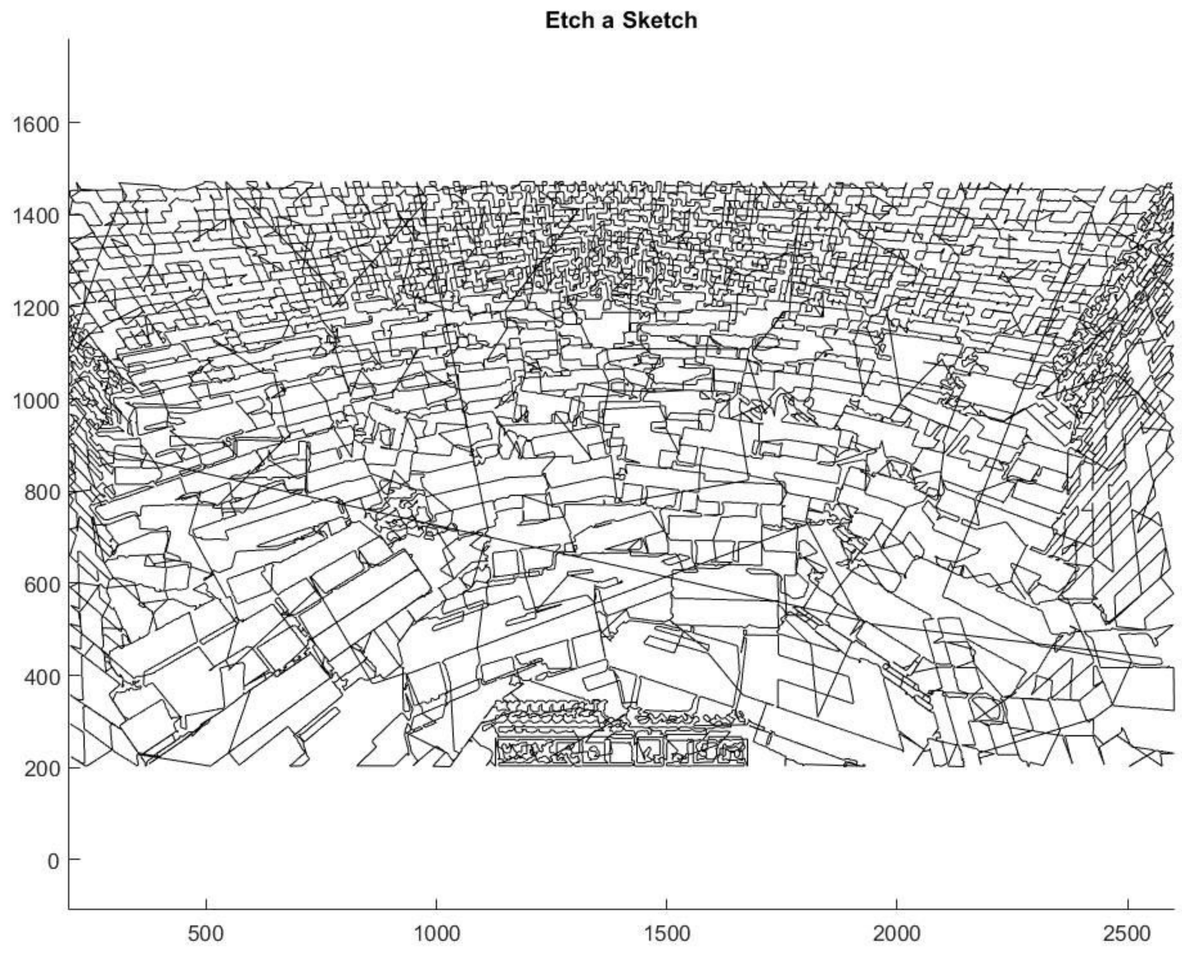


Etch a Sketch

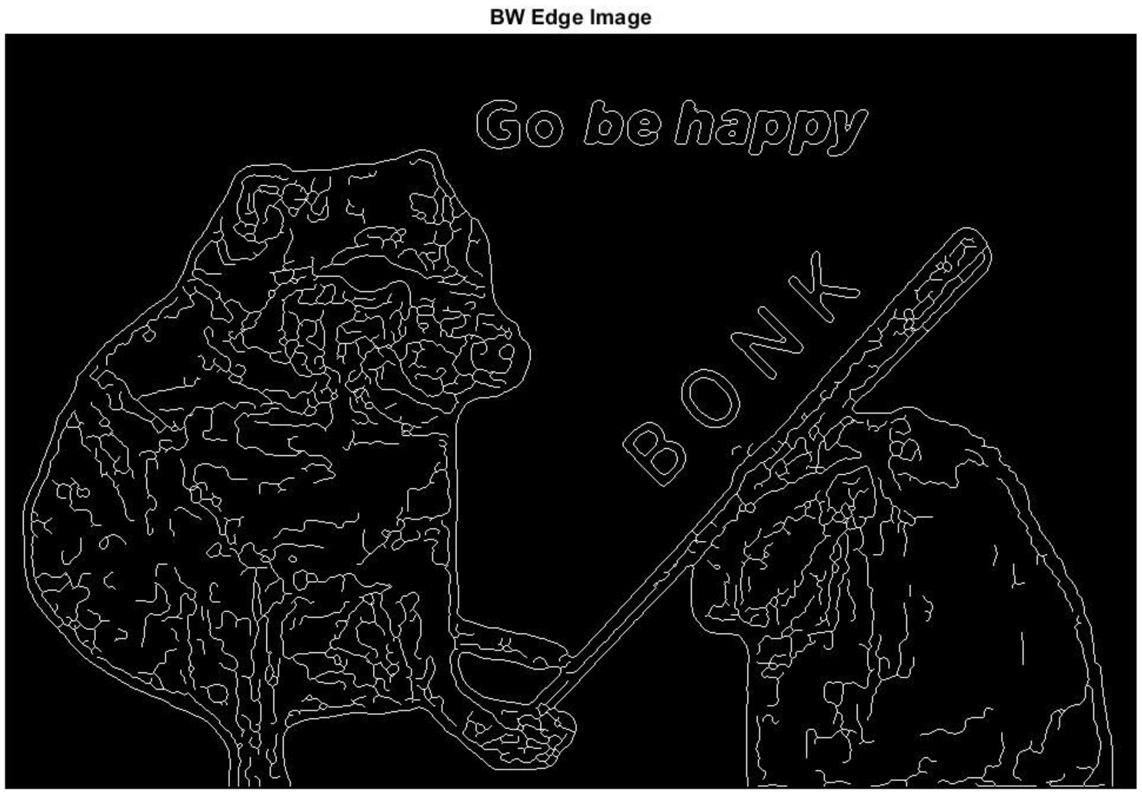


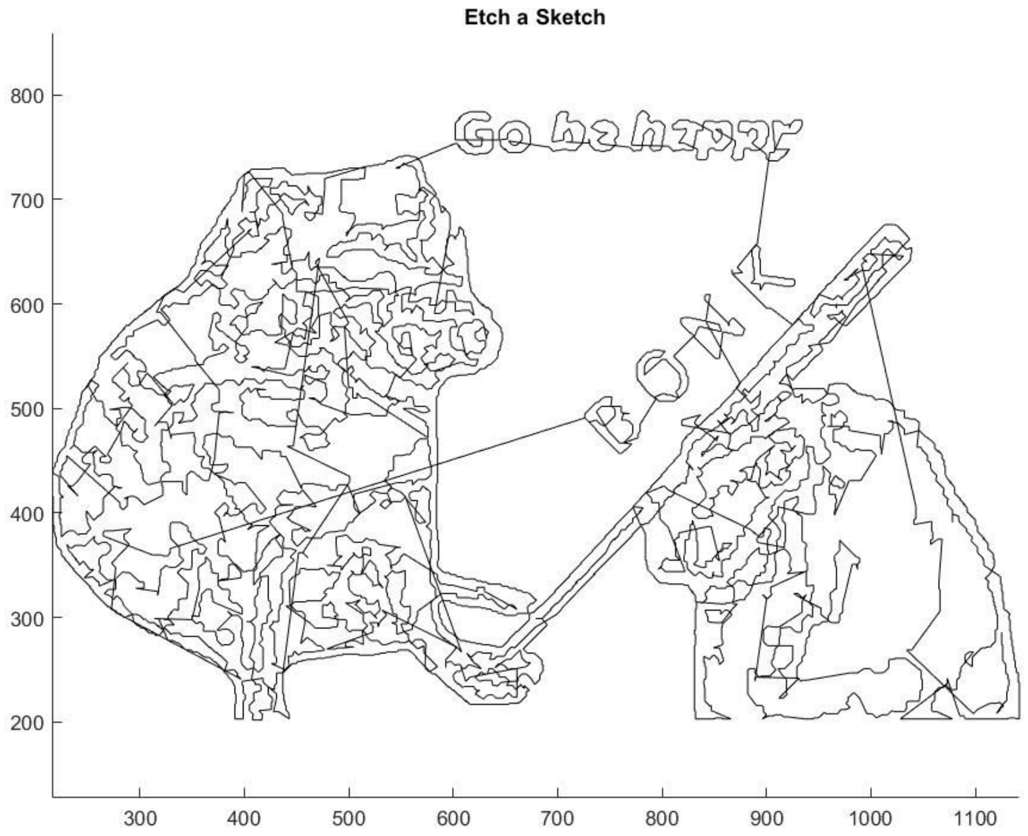

29
45
64
64


BW Edge Image


Etch a Sketch
1600
1400
1200
1000
800
600
400
200
0
500
1000
1500
2000
2500

Go be happy
BONK


BW Edge Image
Go be happy
BONK


Etch a Sketch
Go be happy
BONK
800
700
600
500
400
300
200
300
400
500
600
700
800
900
1000
1100