Universiteit Leiden
Opleiding Wiskunde en Informatica

A Generalized Hough Transform

for the Recognition of Cars in Images

Name: R.L. Marsdon

Date: July 2, 2015

1st supervisor: Dr. A.J. Knobbe (Informatica)

2nd supervisor: Dr. E.A. Verbitskiy (Wiskunde)

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

The Hough transform is a technique often used in the field of computer vision, for the recognition of geometric shapes. Although the simplest form allows for the recognition of lines in images, the algorithm can be generalized to recognize ellipses, for instance, or even more complex figures.

For this thesis we have implemented the Hough transform for the recognition of cars in images. The boundaries between the tires and wheel caps of these cars form two ellipses that can be easily recognized by an edge detection algorithm. Using the generalized Hough transform (GHT) we try to recognize cars by these ellipses.

The biggest challenge of this problem concerns the complexity of the algorithm; the GHT uses a parameter space in which the number of parameters determine the dimensionality of this space. Therefore the space required by the algorithm is exponential as a function of the number of parameters. Not only does this affect the space complexity; the time needed to execute this algorithm is also dependent on the size of the parameter space, which makes it all the more important to find ways to reduce the size of the parameter space.

Contents

1 Introduction

1.1 History

In the history of computer science and mathematics there have often been discoveries or inventions that were originally for one specific purpose but have later grown to be useful in larger areas of work. For example, graph theory was originally invented for small problems, such as the seven bridges of Königsberg, in which a walk over some islands connected by bridges was sought, such that every bridge was crossed exactly once. Nowadays graph theory is used on large scale, for instance for networks and travel directions.

The Hough transform is another example of such an invention. In 1959 Paul V.C. Hough published a paper [1] introducing the Hough transform. This algorithm would be used for recognizing complex patterns in pictures at a subatomic level. At the time it would usually take hours to find these patterns by an expert in the field, so it was a good thing that a computer algorithm could now take on these tasks.

In the past decade a field that has been of greater importance, with the increase of computer devices with high quality cameras and the upgrowth of social media, among other things, is the field of Computer Vision. More attention is given to object recognition and the Hough Transform is one method that is used in this area of work, not only for its original purpose - the recognition of lines - but for the recognition of any arbitrary figure.

1.2 Hough Transform on Cars

For this thesis we have implemented the Hough transform for ellipses to recognize cars in images. Since the boundaries between the tires and wheel caps of a car form ellipses, which can easily be defined with four parameters, we can try to find these ellipses in the image. After finding a set of ellipses, we try to find whether two ellipses are in such proportions to each other, that they can belong to a car.

This thesis focuses more on the theoretical aspect of this process; how the algorithm is constructed, what problems we encounter and how to solve them (if we can solve them), how complex the process is in time and space complexity, etc. However, there are of course applications of this algorithm. The counting of cars that pass a certain road using a camera perpendicular to this road is one example.

The implementation of the algorithm is done in C++, using the OpenCV library. This library contains many useful methods for loading images and processing them. The Hough transform for lines is one of these methods. We have used this method as a starting point for the Hough transform for ellipses. Testing is done on some images of cars provided by my supervisor Arno Knobbe, and some taken by myself.

2 Hough Transform for Lines

2.1 The Algorithm

So how exactly does the Hough transform work? To explain the concept of the algorithm we will first consider its simplest form, that deals with lines. To find these lines, we will first choose how to parametrize these lines. One way to do this is with a slope-intercept parametrization, but a better way was proposed in 1972 by R. O. Duda and P. E. Hart [2], that uses an angle-radius parametrization.

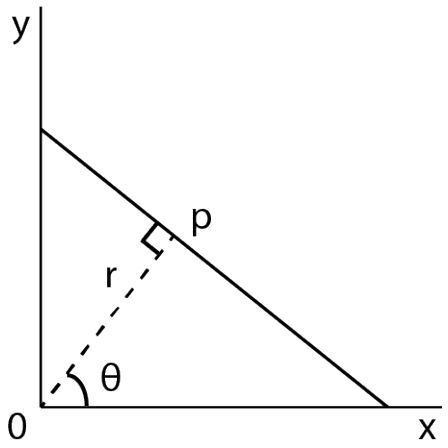


Figure 1: The angle-radius parametrization of a line

In the angle-radius parametrization, we use the fact that a line can be uniquely defined by the closest point p on that line to the origin $(0, 0)$. Indeed, as seen in Figure 1, the line would run perpendicular to the vector from the origin to p, as we could easily show that any other point on the line is of greater distance, with the triangle inequality. So by using the polar coordinates of p, i.e. the radius r or euclidian distance to p and the angle θ that is made between the vector from the origin to p and the positive x-axis, we can parametrize every line uniquely.

The Hough transform makes a transformation from the image, reduced to a binary image using edge detection, to a valued parameter space, called an accumulator after the manner in which it is valued. A parameter space is defined as the set of all possible combinations of parameters, so in our case every pair (r, θ) with $r \in \mathbb{R} \cap \left[-\sqrt{w^2 + h^2}, \sqrt{w^2 + h^2}\right]$ where w and h are the width and height of the image and $\theta \in \mathbb{R} \cap [0, 180]$. Note that the domains of r and θ are confined because in our case all possible lines cross the image and therefore the closest point is no farther than the outermost point (from the origin) in the image. First we value this parameter space using Algorithm 1.

Algorithm 1: Valuing the parameter space

Initialization: I the binary image with $I[x, y] \in \{0, 1\}$ for all x, y.
A the accumulator with $A[r, \theta] = 0$ for all r, θ.

foreach x, y **do**
  if $I[x, y] == 1$ **then**
    $A[r, \theta] = A[r, \theta] + 1$ for all lines (r, θ) that intersect the point (x, y).
  end
end

In this algorithm we look at every pixel of the image individually. If it is 'on', we consider every line in the parameter space that would cross this pixel if it were in the image. For one point this set of lines corresponds with a sinusoid in our parameter space, as seen in Figure 2, so for one line in the image we have a collection of sinusoids that all contain the pair in the parameter space corresponding with our line. Thus the intersection point in Figure 2 describes exactly the line in our image.

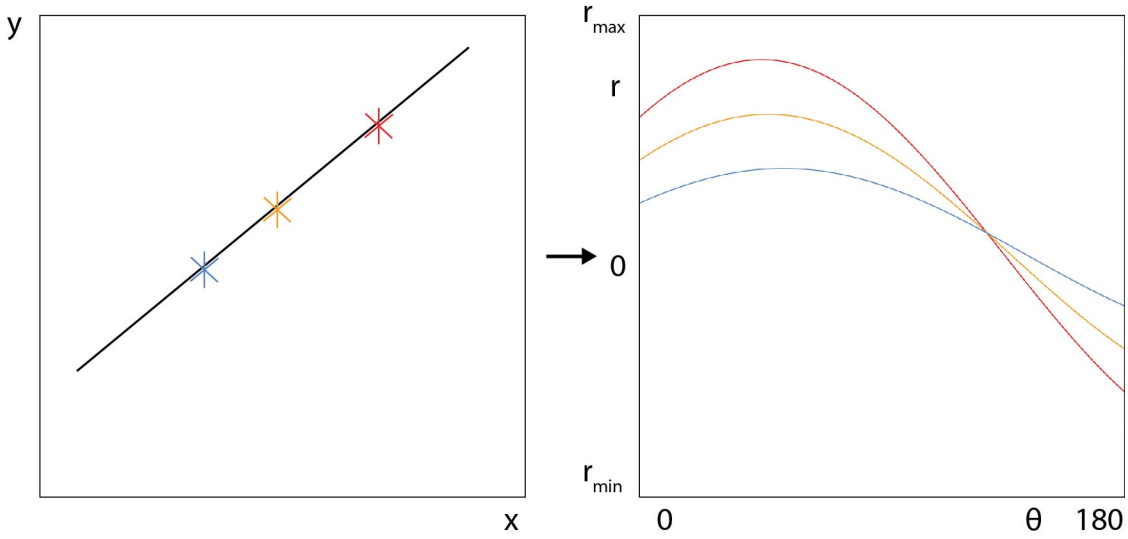


Figure 2: Transformation to the parameter space

2.2 The Algorithm in Practice

However, an algorithm like this is not practicable for a computer, since the sinusoids in Figure 2 are continuous and therefore consist of an infinite amount of points. To resolve this, we make a discretization of the parameter space: we divide the parameter space into equal-sized cells, creating a two-dimensional array. In this step it becomes clear why the angle-radius parametrization is a more convenient parametrization than the more traditional slope-intercept parametrization ($y = ax + b$) of a line; in this parametrization the domain of the slope a is not confined and therefore the parameter space is not easily divided into cells.

In the discretized parameter space, which we call an accumulator, we initialize each cell at a value of zero. Again, we consider for every pixel whether it is on, but instead of increasing the value of all possible lines that intersect this point, we increase the value of each cell that contains at least one of such a line. In this way, in a finite amount of time, there is an

accumulation of points in any cell containing a line that is actually in the image, hence the name accumulator. By looking at the maxima of the accumulator, or the cells with a value bigger than a certain threshold, we can roughly find the parametrization of the lines present in the image.

To find the possible lines that would intersect a certain point (x, y) we can loop over all the possible values of θ with a discretization interval of one degree, for example, and then find the corresponding value $r(\theta, x, y)$. We can find a formula for this r by expressing the conventional slope-intercept notation of a line $y = ax + b$ in polar coordinates. To accomplish this, we can find with simple geometry that

$$a = -\frac{\cos\theta}{\sin\theta}$$

and also that

$$b = \frac{r}{\sin\theta},$$

so substituting this in the slope-intercept notation gives us

$$y = -\frac{\cos\theta}{\sin\theta}x + \frac{r}{\sin\theta}$$

and finally

$$r = x\cos\theta + y\sin\theta.$$

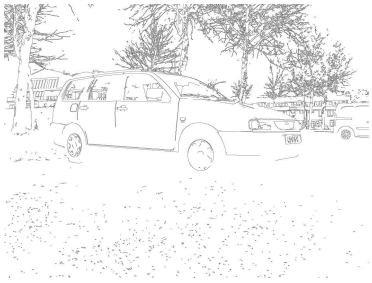
2.3 Results

In Figure 3 the results of the Hough transform for lines are shown. Two of the provided images of cars have been used as input, shown in Figure 3a and Figure 3d. To get a binary image we have used the Canny edge detector, for which OpenCV has provided a simple method, that gives us the images shown in Figure 3b and Figure 3e. Finally, we have run the Hough transform on these binary images and drawn the resulting lines on the images with another OpenCV method, resulting in the images shown in Figure 3c and Figure 3f.

We do get some good results: the line that is formed by the bottom-side of the car window in Figure 3c is found by the algorithm. Furthermore, the lines formed by the top of the car and the black side bumper are found in Figure 3f. However, in Figure 3f also a lot of non-existing lines are found, caused by the false evidence in the top of the image - the large amount of pixels formed by the tree branches.

Moreover, not all lines in the image are found. For example, the bottom of both cars are not seen in the resulting images. By decreasing the threshold above which a line candidate is accepted as existent in the image, we can perhaps find more of these lines, but we will also find more incorrect lines caused by false evidence. Since this project is about recognizing cars and not lines in images, we have not tried to improve the accuracy of finding lines, but in the following chapter we will discuss ways to do so, and have applied these to the GHT.

(a) Source image 1



(b) Canny edge detection

(c) Final result

(d) Source image 2

(e) Canny edge detection

(f) Final result

Figure 3: Hough transform for lines

3 GHT Algorithm

The outline of the GHT is more or less the same as that of the Hough transform for lines. We need to find a parametrization for the given figure - in our case ellipses. Then, for each 'on' pixel we increase the value of all figures in the parameter space that contain that pixel. Figures with a value above a certain accepting threshold are regarded as present in the image. In our case, there is one more step and this is the pairing of ellipses; since we need to find cars we are not looking for a single ellipse, but instead a pair of ellipses, which need to be at such a position in the image that, together, they can belong to a car.

The aspect that complicates the construction of the GHT algorithm is the fact that, with an increasing amount of parameters, the size of the parameter space grows exponentially. Because of this, we need to take great care in the manner in which we parametrize the ellipses, and try to find ways to make the computation faster also. More about the complexity of the algorithm can be found in chapter 4.

3.1 Ellipses

A natural way to parametrize a non-rotated ellipse is by using the position of its centre and its height and width. This gives us a total of four parameters. We have made some logical assumptions to reduce the amount and range of parameters.

Firstly, the pictures are taken from the side of the car, with the camera roughly parallel to the ground. Because of this the ellipses are indeed non-rotated, and we do not need more than four parameters. Secondly, the pictures are taken at or slightly above wheel height - say a

height of half a meter above the ground. From this it follows that all wheels will be at roughly the centre of the image or beneath it. Disregarding the top 40 percent of the image gives us a great advantage in both time and space complexity. Also the height of the ellipses will be about the same size as the width, or slightly bigger if the wheels of the car, or the car itself is not completely parallel to the camera.

For easier computing, we define a constant b as the base width of an ellipse - in other words, the smallest width we consider. The first parameter can then have a range from 0 to the largest width possible minus b, such that the width of the ellipse is equal to this parameter plus b. We define the second parameter as the number that we add to the base width and first parameter to get the height, after subtracting a small number c. We subtract this last number so that this parameter can also range from 0 to a certain constant, since the height can also be slightly smaller than the width of the ellipse.

While it is possible - and originally our plan - to parametrize a pair of ellipses with fewer than eight parameters, by also taking into regard that they belong to a car and therefore have some interdependencies, this still gives such a large increase in complexity, that we regard the pairing of ellipses subsequently as the better solution. We will discuss this complexity in greater detail in chapter 4.

3.2 Filling the Accumulator

In Algorithm 2 the generalized algorithm is shown that we have constructed for filling the accumulator for a given 'on' pixel with array indexes i, j. The `UPDATEACCUMULATOR[k]` procedure is a recursive algorithm that works for any figure with any amount of parameters n, while `CALCULATEREST`$[k, F]$ can be adjusted per figure and is now targeted at ellipses. By calling `UPDATEACCUMULATOR[0]` for every 'on' pixel i, j we can fill the entire accumulator.

In this algorithm we look at every possible combination of the first $n - m$ parameters, that is part of the parametrization of a figure that contains the given pixel. This is the recursive part of the algorithm, and is done in the if-clause of the algorithm. If we would apply this algorithm to lines, m would be equal to 1 and we would only look at all possible values of one parameter, namely θ. For ellipses, we take $m = 2$ and look at all possible combinations of the width and height of an ellipse.

From these combinations and the array indexes of the pixel, we calculate the remaining parameters with `CALCULATEREST`$[k, F]$, and for every complete set of parameters we increase the value in the parameter space by one. For lines this would be just one parameter value, calculated with the formula in chapter 2.2. For ellipses the other parameters are calculated in Algorithm 2: for every angle θ (with a certain discretization) that the vector from the center of the ellipse to the pixel makes we calculate the coordinates of this center with simple geometry rules. For the graphically minded, looping through these values for θ would be like running over the path of the found ellipse, regarding the point on the ellipse as the found pixel and calculating the associated ellipse center.

To improve the running time of the algorithm it is, of course, possible to take a bigger discretization step, instead of only one degree. In this case we just need to notice that there is less evidence for a present ellipse, so we need to lower the accepting threshold.

Algorithm 2: Filling the accumulator recursively for an 'on' pixel $\{i, j\}$

Initialization: `index`$(p_1, ..., p_n)$ - a function returning the accumulator index for a combination of parameters $p_1, ..., p_n$.

I the binary image with $I[x, y] \in \{0, 1\}$ for all x, y.

A the accumulator with $A[k] = 0$ for all index $k =$ `index`$(p_1, ..., p_n)$.

i, j the array indexes of a given 'on' pixel.

n the number of parameters.

m the amount of parameters that are calculated based on the other parameters, and the coordinates of the pixel. In our case, $m = 2$.

N the number of values the parameters can take on with $N[k]$ the number of possible values the k-th parameter can take on.

F the figure that runs through the pixel with $F[k]$ the value of the k-th parameter (to be filled).

b the base width.

c a small constant, as in chapter 3.1.

Procedure *UPDATEACCUMULATOR[k]*:

```
if k < n − m then
    for p = 0, 1, 2, ..., N[k] − 1 do
        F[k] = p.
        UPDATEACCUMULATOR[k + 1].
    end
else
    CALCULATEREST[k, F].
end
```

Procedure *CALCULATEREST[k, F]*:

```
for θ = 0, 1, 2, ..., 359 do
    F[k] = j + (b + F[0]) · cos(θ).
    F[k + 1] = i + (b + F[0] + F[1] − c) · sin(θ).

    A[index(F[0], ..., F[n])] = A[index(F[0], ..., F[n])] + 1.
end
```

3.3 Finding Potential Figures

After filling the accumulator we take those figures with an accumulator value above the accepting threshold, and sort them from highest to lowest value. After this we take the first n figures from this list as present in the image.

For one figure in the image it is possible that multiple figures are found that are very close together and share a large amount of the same evidence. For this a commonly used solution is to look at the local maxima, so that adjacent figures with a slightly lower accumulator value are disregarded. However, after implementing this method we noticed there were still ellipses found that were not directly adjacent, but still near to a higher valued ellipse.

We solved this problem by taking into consideration the entire neighborhood of a found ellipse

- which we define as a fixed box around the ellipse - after sorting them by value. After the sort we loop over the found figures and accept a figure if there is not already one accepted in its neighborhood. Since the list of figures with high accumulator value is sorted there will not be an ellipse in the neighborhood with a higher value than an ellipse should it be accepted. This solution has two advantages: the evident advantage is that there are no two ellipses found that belong to the same wheel in the image, but there is also no need to look at the local maxima in the image, which is far more time-consuming than this new solution; while we need to do multiple operation for each figure in the accumulator to find the local maxima, this new solution considers only the figures with a high accumulator value. The loop over the accumulator is one of the most time-consuming parts of the Hough Transform, so after implementing this feature we found a large improvement in the running time of the algorithm overall.

3.4 Gradient Direction

The final addition to the algorithm is use of the gradient direction of an 'on' pixel, as first described for lines in 1976 by Frank O'Gorman and MB Clowes. [3] In OpenCV we can use the Sobel operator on the original image to find an approximation of the gradient for every pixel. With this gradient we can compute the direction of a pixel. For a point on the border of an ellipse that is the direction of its tangent, so the direction towards the ellipse center is this value plus or minus 90 degrees. While in Algorithm 2 we looped over all possible values of this angle θ with a certain discretization, the gradient direction makes it possible to calculate an approximation of θ, which makes the algorithm more precise - ellipses that contain the same pixel but with another angle do not have an increased value in the accumulator - and also significantly faster.

Although the gradient direction is still not completely reliable in our case, nor is the algorithm overall, the final results are reasonable, as seen in Figure 4. The gradient direction perhaps does not improve the preciseness to great extent; in our project the biggest advantage is the improvement in time.

3.5 Results

We have run the final program on 18 images with several different combinations of parameters, and found that, with the resources we have, the program gives fair results on images of about 900x600 pixels using a width base b of 30, the constant c at 5 pixels and N defined as $N[1] = \text{height} \cdot 0.05$, $N[2] = \text{height} \cdot 0.025$, $N[3] = \text{width}$, $N[4] = \text{height} \cdot 0.6$. For θ we consider a margin of 40 degrees around the approximation derived from the gradient direction, and a discretization step of 2 degrees. An ellipse is accepted if its accumulator value is above 20.

Four of the results are shown in Figure 4. Although the program gives accurate results on some images, like in Figure 4b and 4d, not every wheel can be detected by the program. For example in Figure 4a only one of the wheels was found; the other wheel probably lies just out of the range of regarded ellipses. On some images, like in Figure 4c, some ellipses were found that were not actually wheels but part of the surroundings.

(a) Image 1

(b) Image 2

(c) Image 3

(d) Image 4

Figure 4: GHT for ellipses: results

3.6 Pairing up Ellipses

After finding potential figures there is just one more step to take: that is to find an ellipse pair that can belong to a car. By taking into consideration the position of the ellipses in relation to each other, the program can have a bigger success rate; if two ellipses are at illogical places in the image we can directly conclude that the car does not belong to a car.

In perspective we know that horizontal parallel lines have a vanishing point on the horizon. Since we take pictures horizontally the line that we can make by connecting the top of both wheels, and another line by connecting the bottom of both wheels, are horizontal and parallel to each other. Therefore they have a vanishing point roughly at the (vertical) middle of the picture.

We can calculate the y-coordinate of this vanishing point as follows. We define t_1 and t_2 as the y-coordinate of the top of the two wheels, and b_1 and b_2 as the y-coordinate of the bottom. We define f as the top line going through the top of both wheels, and g as the bottom line, going through the bottom of both wheels. Since translating and scaling both lines horizontally does not change the y-coordinate of the vanishing point - i.e. the intersection of f and g - we can redefine the x-coordinate of the first wheel as 0 and the x-coordinate of the second wheel as 1.

We then have

$$f(x) = (t_2 - t_1)x + t_1$$

and

$$g(x) = (b_2 - b_1)x + b_1.$$

So this gives us:

$$\begin{aligned} f(x_v) &= g(x_v) \\ (t_2 - t_1)x_v + t_1 &= (b_2 - b_1)x_v + b_1 \\ x_v &= \frac{b_1 - t_1}{t_2 - t_1 - b_2 + b_1} \end{aligned}$$

And finally the y-coordinate of the vanishing point:

$$y_v = f(x_v) = \frac{(t_2 - t_1) \cdot (b_1 - t_1)}{t_2 - t_1 - b_2 + b_1} + t_1$$

If this y_v lies in an interval surrounding the centre of the image, we will assume that the ellipses form a car. However, there are two exceptions we consider first: if f and g are parallel lines there is no vanishing point. We then assume that there is no car, unless the wheels are also at the same height in the image, which is in fact the other exception; due to the camera being not perfectly parallel to the floor, or other small errors, it is possible that the vanishing point lies far off the centre, for wheels that are at very similar heights and also have similar heights. In this case we assume that there is in fact a car present in the image.

4 Complexity

4.1 Space Complexity

One of the things that make the GHT difficult to implement efficiently, is the large amount of memory needed. This amount is mostly dependent on the accumulator; since this accumulator contains every combination of parameters, the size of it is the product of the amount of these parameters.

The values that are given to a combination in the accumulator in our program do not get very large, so we assign 2 bytes per element. This gives us a maximum accumulator value of $2^{2 \cdot 8} = 65,536$, which is easily enough for our problem. Therefore the space that the accumulator takes is $2 \cdot \prod_{i=1}^{4} N[i]$ bytes. As an example, we will take an image of 900x600 pixels. Then the size of the accumulator is:

$$2 \cdot \prod_{i=1}^{4} N[i] = 2 \cdot 0.05 \cdot 0.025 \cdot 0.6 \cdot 900 \cdot 600^3 \approx 470 \text{ MB}$$

To sort the accumulator we need another buffer of the same size, so the accumulator needs about a gigabyte of space. It is clear that the original idea to parametrize a pair of ellipses is problematic: with only one more parameter the space needed becomes larger than what is available on a conventional computer.

4.2 Time Complexity

The running time is also mostly dependent on accumulator size. For our images we found that filling the accumulator takes around 3 minutes, while the part taking longest is finding the figures that have high accumulator values - in our case this takes around 10 minutes. To find the figures with high accumulator values we take a look at all parameter combinations, so we consider a total of $\prod_{i=1}^{4} N[i]$ figures.

To fill the accumulator we consider every combination of width and height of an ellipse for every 'on' pixel - which amounts to $N[1] \cdot N[2]$ combinations - but the other two parameters are computed using the coordinates of the 'on' pixel and in our case 40 values for θ. We need to do this for every 'on' pixel so in the worst case we increase $40 \cdot \prod_{i=1}^{4} N[i]$ values in the parameter space. However, in reality this amount lies significantly lower due to a smaller quantity of 'on' pixels. It is also interesting to mention that the time complexity of this operation is dependent on the image, in contrast to finding the figures with high accumulator values, which always has the same amount of basic operations.

5 Conclusion

The Hough Transform is a well-functioning algorithm for the recognition of figures and also for cars it is a reasonable choice, although still not giving perfect results - as still seen regularly in the field of computer vision. An issue of the GHT is that it has a large time and space complexity. Although we managed to reduce the time complexity significantly using the gradient direction of a given pixel, it still takes multiple minutes to run the program. The space complexity is also a large part of the problem; this makes it impossible to find wheels of a large range of sizes, without a big amount of available memory.

References

[1] Paul V. C. Hough, *Machine Analysis of Bubble Chamber Pictures*, 1959

[2] R. O. Duda and P. E. Hart, *Use of the Hough Transformation to Detect Lines and Curves in Pictures*, 1972

[3] Frank O'Gorman and MB Clowes, *Finding Picture Edges Through Collinearity of Feature Points*, 1976