NTNU
Norwegian University of
Science and Technology

Edge and line detection of complicated and blurred objects

Kari Haugsdal

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Ketil Bø, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Edge and line detection is a central process in many image-processing projects. This process can be quite problematic when the relevant object is complicated and or blurred.

The aim of this project is to develop and implement a sturdy algorithm for edge and line detection of images with complicated or blurred objects. Examples are pictures with lots of texture or blurriness.

It should also be possible for the program to detect and identify simple objects.

Implementation should be done using C++

Assignment given: 15. January 2010
Supervisor: Ketil Bø, IDI

Acknowledgments

I would like to acknowledge the support and help from my supervisor Ketil Bø. He was a most important help and drive force in guiding me in this project. I would also take the time to thank Bakkelandet choco bocco café for excellent Internet connection and a delicious lunch menu.

Summary

This report deals with edge and line detection in pictures with complicated and/or blurred objects. It explores the alternatives available, in edge detection, edge linking and object recognition. Choice of methods are the Canny edge detection and Local edge search processing combined with regional edge search processing in the form of polygon approximation.

Table of Contents

Chapter 1

Background

This chapter will explain line and edge detection in general. A presentation of the foundation for this master assignment, namely the pre-project and its results will be given. In the end we will look at the stat of art today, and also some examples of real life applications.

1.1 Line and edge detection

Line and edge detection is basically the process of finding all lines and edges of interest in an image.

To clarify, it's in its place to explain the difference between a line and an edge. The difference between lines and edges are simple. Edges are the border of an object in the picture, while lines are lines, which are something very thin in the picture, like for example a clear-cut horizon or a thin string I will now and then refer to lines and edges, only as lines, or only as edges, to avoid having to write the whole "lines and edges" sentence in every mentioning of them.

Here follows an example of line and edge detection:

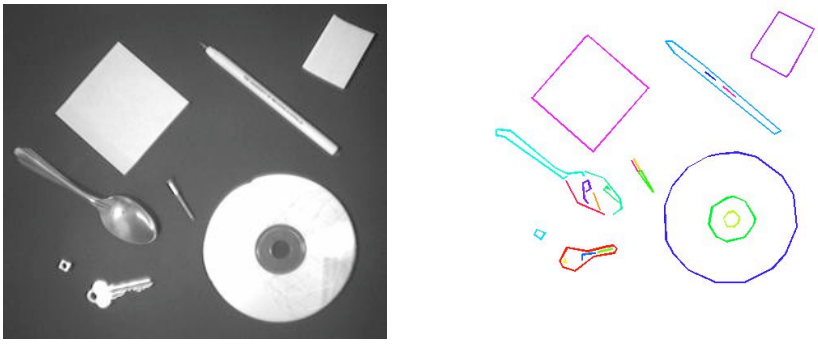


Figure 1 t.l figure 2 t.r [2]

The image being processed is a simple image consisting of clearly defined shapes. Its output edges and lines are all coloured a random colour and drawn with a pixel width of two. Even if it is a simple picture you can see that some irrelevant and incomplete lines and edges has made its way into the output.

As humans we have no problem recognising the spoon as an object and tracing its edge around. But a computer has difficulty understanding that the bright spot in the spoon is not an object, but part of the spoon. Also the spoon

blends into the background to such a degree that the computer has difficulty tracing the whole edge.

Many different methods for edge and line detection can be used and some programs might have better success than others. Even so, when a simple picture like the one above is tough to perform perfect edge and line detection on, then consider the difficulties encountered in more complex real life (not computer generated) pictures.

If you have some knowledge about the images being processed, or the objects you want to find in them, that is if you have some knowledge about your problem space at all, it is possible to tailor the algorithm towards your target. For example you know that all input images are gone be separate objects, lying on a table, or all input images are gone be portraits of peoples faces and you want to find their eyes. Then it is possible to achieve pretty good edge and line detection, followed by shape recognition. Even if the only thing you know is that all input images are gone be manmade scenes and objects, like city landscapes and furniture, and not nature images, you still have an advantage. The reason is that man made objects have more straight lines and perfect circles than pictures of nature.

The problem space of this project is defined in the problem description as “developing a sturdy algorithm for complicated and/or blurred objects”. This means in practice all kinds of pictures, difficult pictures as well as normal difficulty pictures. To make an algorithm like this, that works equally well on all kinds of pictures, without tailoring towards any kind of pre-knowledge about the images, is very difficult, and perfection is impossible. Computers are still fare from humans when it comes to vision.

1.1.1 Shape recognition

Shape recognition was mentioned and will be explained right away here in this section, as it is an important part, or partner to edge and line detection. The normal proceedings in edge and line detection, is first highlighting the lines and edges of the image using some sort of edge detecting algorithm. The resulting image is usually imperfect, with gaps in the lines and noise in the form of small edges, even wrong edges. Some image analysis is needed to correct and perfect the image.

You can go over and link similar edge points that is in close proximity, get rid of noise edges, analyse the shapes in the output and complete the edges based on this, or some combination these methods and so one.

Analysing the image gives you information about the shapes and lines found in the image. Is the edge a line, a straight line? Maybe the edge is part of the closed boarder of an object, and thus classifies as an actual edge? Is the object in the shape of a square or a circle or is it a more complex arbitrary shape (shapes having no simple analytical form)? Shape recognition should be able to find and classify all the lines in an image like this.

Showing you an example of simple shape recognition, here is the Hough transform in its simplest form. The “shape” recognised is prominent straight edges, and they are highlighted red and superimposed on the raw edge detection output of the original picture.

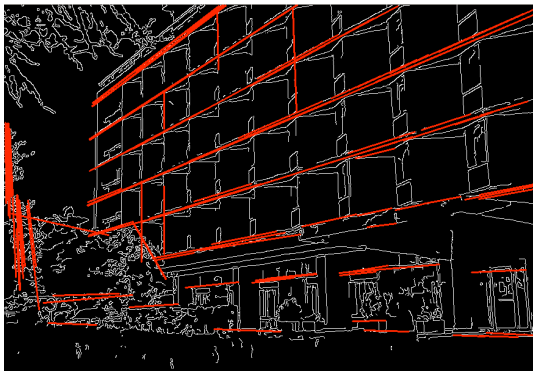



Figure 3 t.l figure 4 t.r [3]
I will refer to shape recognition as object detection/recognition as well. Even though objects are something more complex than shape. An object can have a certain shape, but a shape can’t have an object. The problem definition states that the resulting program should be able to detect and identify simple objects.

Also it is natural to think that shape detection is a form of feature detection. A line being straight or curved is a feature, and an edge being connected to itself in a closed border is also a feature. But the concept future detection is already used to refer to a class of methods that aim at computing abstractions of image information, and then make local decisions at every image point whether there is an image feature of an given type at that point or not [4]. Edge detection methods are feature detectors. To avoid confusion I will be consistent in my use of the word shape/object detection/recognition.

1.2 Experiences from the pre-project

The first part of edge and line detection, namely the feature detection part, was dealt with in the pre-project. Referring to the pre-project report this section will present the algorithm selected for the feature detection part. Basic image-processing methods and techniques that are important for both the pre-project and this report will also briefly be presented.

1.2.1 Basic image-processing methods and techniques

Edge and line detection is a form of image segmentation, something that can be done using either similarity or discontinuity. Thresholding, region growing, region spitting and merging are all examples of similarity methods and are called region-based segmentation. With discontinuity you separate the picture based on sudden changes in intensity, this is called edge-based

segmentation. You don't necessarily need to base yourself on only on of the two, and better results are often reached using a combination of the two.

In edge based segmentation you use derivative to detect sudden changes in intensity in a picture.

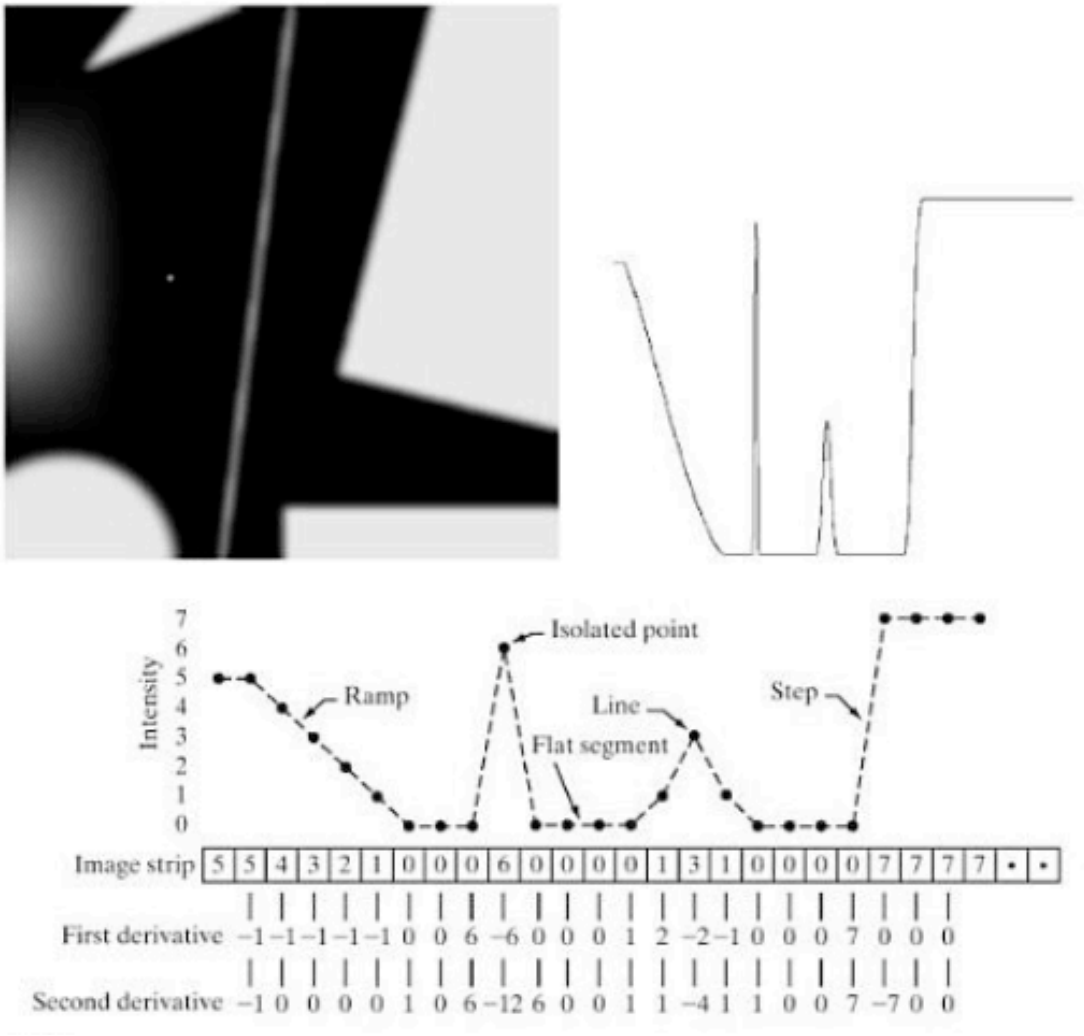


Figure 5 [5]

Through spatial filtering you can determine first and second derivative in a picture. One simple example is point detection. If we want to detect small points that stand out in a picture, convolve a Laplace mask with the picture.

1	1	1
1	−8	1
1	1	1

Figure 6, Laplace

Pass the filter over every pixel in the picture, computing the sum of products between filter and corresponding pixels at every point. This will give a strong response when the filter is centered on a point.

This method can be used for line detection as well, with for example horizontal and vertical lines as well as diagonal lines.

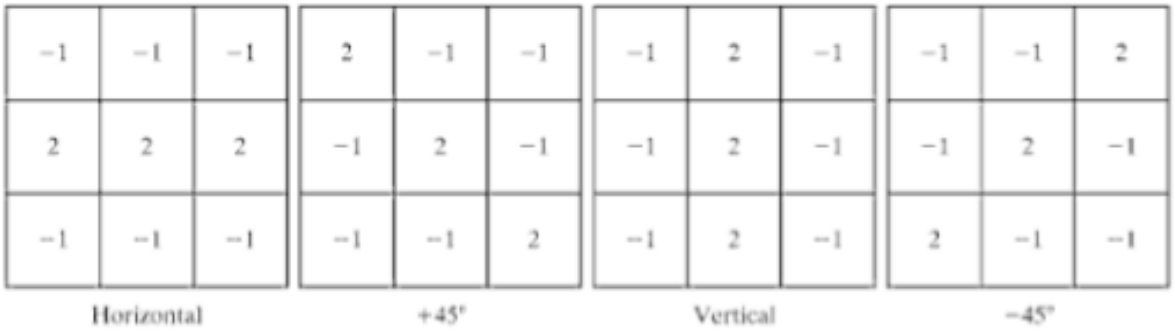


Figure 7

Edge detection is done in a similar way to point and line detection. You start with computing the gradient of every point in the picture to attain the strength and direction of that point. The gradient, which is a vector points in the direction of greatest rate of change and can be computed using simple 1x1 or 2x2 spatial masks

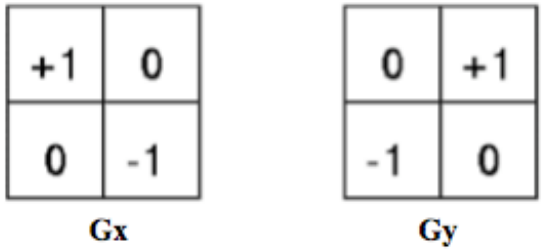


Figure 8, Roberts Cross

Example, if a point is along the edge of a dark are, then the gradient of this point will point in a 90-degree angle away from the edge towards the light

neighboring area. The magnitude, or length of the gradient vector gives us the value of this greatest rate of change and it can be computed using this equation.

$$\mathbf{G} = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}$$

Equation 1

The gradient angle is computed like this:

$$\mathbf{\Theta} = \arctan\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right)$$

Equation 2

Mask that are symmetric about the center point, the smallest being of size 3x3, take into account the nature of the data on the opposite sides, and thus carry more information about the direction of the edge.

-1	0	1
-1	0	1
-1	0	1

1	1	1
0	0	0
-1	-1	-1

Figure 9, Prewitt

-1	0	+1
-2	0	+2
-1	0	+1

+1	+2	+1
0	0	0
-1	-2	-1

Figure 10, Sobel

1.2.2 Canny edge detector

There are several methods for edge detection which all has the aim of identifying points in an image of which the image brightness changes sharply.

The Canny edge detector, or Canny operator was designed to be an optimal edge detector (according to particular criteria – there are other detectors around that also claim to be optimal with respect to slightly different criteria) [6]. To make an optimal edge detection algorithm is in principle impossible, and a highly subjective task. Canny Edge detection was developed by John F. Canny in an attempt to create an optimal edge detection algorithm. Although his work was done in the early days of computer vision, the Canny edge detector is still a state-of the-art edge detector, and it is hard to find an edge detector that performs significantly better than the Canny edge detector [7] [8].

With this in mind the canny operator was the choice of feature detector for my algorithm, and thus the base for this report.

The canny edge detector is a multi-stage process, and each step will be briefly explained here.

1. First step is like many other edge detection methods, noise reduction. You smooth the input image with a Gaussian filter.

2. Second step is computing the gradient magnitude and angle images, using Roberts, Prewitt or Sobel for example.

3. Third step is to apply nonmaxima suppression on the gradient magnitude picture, this because the gradient magnitude picture still contains many thick edges that we whish to make thinner. Nonmaxima suppression is done by using the gradient angle in every point to find the direction (horizontal, vertical and two diagonal directions) that describes the point best. If the value in this point is less than minimum two of its neighbors along its direction, then suppress the point (make it part of the background instead of the edge).

4. Step four is to use double thresholding and connectivity analysis to detect and link edges. Double thresholding or hysteresis thresholding make use of two thresholds, one low and one high. All points from above the highest threshold are adopted, but points between the higher and lower threshold is included, only if they connect to the strong points calculated from the high threshold. Deciding if points are connected can be done using for example 8-connectivity, or you can trace all points using the directional data of the gradient picture.

As you can see some choices are given as to how to implement the details of the algorithm, where the main adjustable parameters that affect the effectiveness and computation time the most, are the size of the Gaussian filter and choice of Thresholds [9]. Besides this the recipe is pretty rigid, and results between different implementations are often similar.

My implementation from the pre-project of the Canny edge detector comes attached to this report, and is the base for my project, and part of the finished product. Some bug fixing has been done since the completion of the pre-project and the detector performs its task quite well. Here is an example use of the finished program, in all its stages:

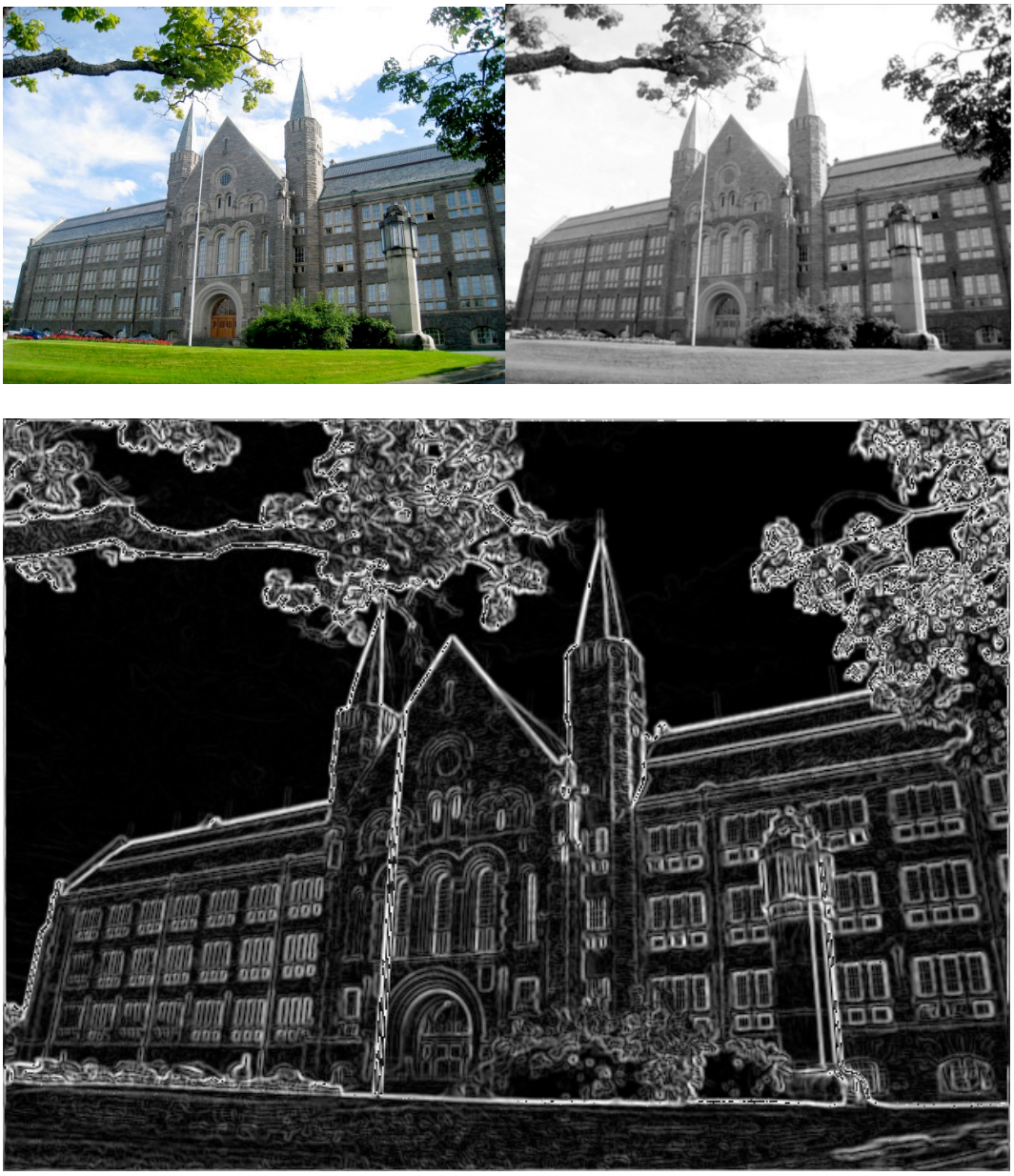


Figure 11 t.l, the NTNU main building, figure 12 t.r, greyscaled and smoothened, figure 13 bottom, gradient Image.

Figure 14 top, Nonmaxima suppression, figure 15 bottom, Hysteresis Thresholding.

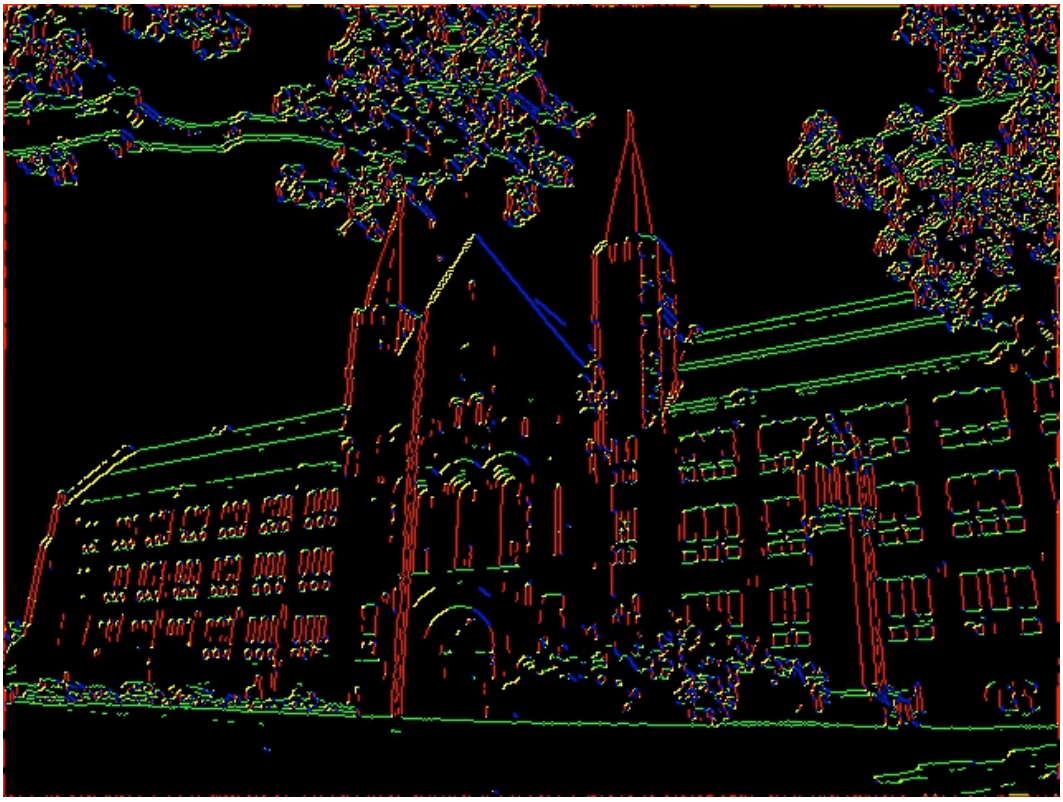


Figure 16, Angles colored

The first step as explained is gray scaling and smoothing of the picture. Second up comes computing the gradient, then its time to thin the edges down, and get rid of noise using nonmaxima suppression and hysteresis thresholding. The uneven surface of the building gives us a lot of noise, but as you can see most of this is taken care of with the hysteresis thresholding. This combats one of the problems mentioned in the problem description, namely images with lots of texture.

The last image is added to show the angles saved in the underlying data structure. Every pixel has a red, blue and green value, as well as an angle value. Coloring vertical pixels red, horizontal pixels green, 45-degree pixels yellow and 135-degree pixels blue, it is possible to visualize these angles.

1.3 State of the art

Today, edge detection is used in many real life applications of image processing and computer vision. Major application areas are medical image processing, shape and object recognition like for example industrial image processing, and so one. Real time video processing for edge detection are also in use, in the form of video surveillance, traffic management and so one. These operations typically require very high computation power.

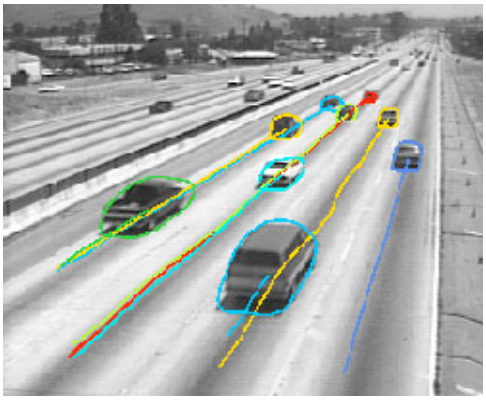
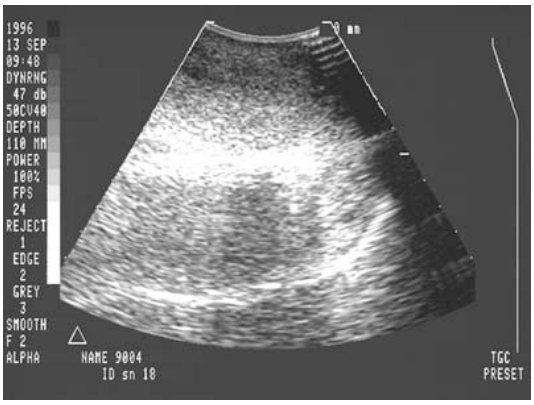


Figure 17 t.l, Traffic Surveillance, Figure 18 t.r Motion Stereo Parking Assistant [10]



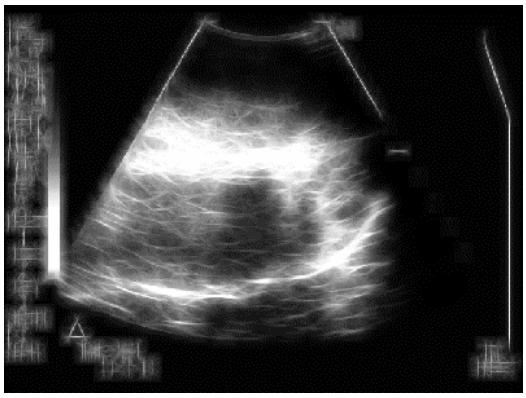


Figure 19 t.l, Ultrasound, figure 20 t.r, processed enhanced image showing lines more clearly.

Very good results can be achieved when we have knowledge about the problem space, as is the case in most of the real life applications in use today. Perfect edge detection is a subjective task, and we are fare from that goal. Also, the most popular methods in use today, like the Canny operator and Hough transform, were developed the early days of computer vision, and are still popular today. So despite good results in current methods, it is easy to see that the room for improvement is huge.

Chapter 2

Possible solutions

This chapter is all about reviewing all the different alternatives available when it comes to choice of method for this project. Only the big scale methods are examined, and choices available after choosing the main approach will be researched further in the next chapter along with the reasoning behind choosing it. We will start of lightly and examine different edge thinning algorithms before we move on to looking at probably the main point of this report, namely edge linking.

2.1 Edge-thinning algorithms

An alternative “fifth” step in the Canny edge detection algorithm is to apply an edge-thinning algorithm to the result. Despite applying nonmaxima suppression in the Canny edge detectors third step, there are still edges thicker than 1 pixel left, and for achieving effective edge linking, which is the next step after edge detection, it is advised to use 1-pixel thick edges as a foundation.

When it comes to the choice of an edge-thinning algorithm we have some alternatives. Here the two most common methods is thinning and skeletonizing. These are both methods for thinning, or eroding binary pictures into 1-pixel thick edges, but give significantly different output that is worth taking a look at.

Thinning and skeletonizing are both morphological operations. Morphology in biology deals with form and structure of animals. In the context of mathematical morphology, morphological operations are tools for extracting image components that are useful for describing region shape, such as boundaries and skeletons.

2.1.1 Skeletonization

The intuitive definition of a skeleton is based on something called “the prairie fire concept”. Visualize the image region to be skeletonized as a prairie of dry grass. Suppose a fire is lit along its border, all fire fronts will advance into the region at the same speed. The skeleton of the region is then the set of points reached by more than one fire front at the same time.

The skeleton of a region R with border B is thus defined as follows:

- For each point p in R, we find its closest neighbor in B.

- If p has more than one such neighbor, it I said to belong to the skeleton of R.

There are many types of skeletonization algorithms, all of which produce slightly different results. Some simple examples of skeletons follow here.

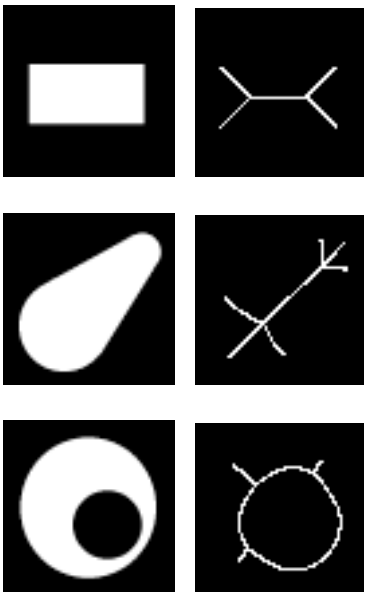


Figure 21 t.l, shapes, figure 22 t.r, the skeleton of the shapes:
[11]

2.1.2 Erosion

As a small footnote I would like to mention erosion, as it is a method easily confused with thinning. Also an morphological operation erosion erodes a picture by a variable sized structure element with the goal being eroding away noise like lines or salt and pepper noise, while still keeping the rest of the picture bigger than 1-pixel thick edges. This makes erosion unsuitable for our purpose, and we can disregard it as an option.

2.1.3 Thinning

Thinning is essentially a morphological operation that is used to remove selected foreground pixels from binary images, somewhat like erosion. Thinning can be used for several applications but is said to be particularly useful for skeletonization [13] which tells us that thinning, produces a sort of skeleton, and that these two different methods are also closely related.

The morphological operation thinning in itself is most commonly employed to tidy up the output of edge detectors [12], which describes exactly the use we are planning for something that bids well! Now, lets look at the method and it output, before reaching any conclusions.

The thinning algorithm work like all other morphological operations, by using a structuring element “the thinning operation is calculated by translating the

origin of the structuring element to each possible pixel position in the image, and at each such position comparing it with the underlying image pixels. If the foreground and background pixels in the structuring element *exactly match* foreground and background pixels in the image, then the image pixel underneath the origin of the structuring element is set to background (zero). Otherwise it is left unchanged. The operator is normally applied repeatedly until it causes no further changes to the image" [14].

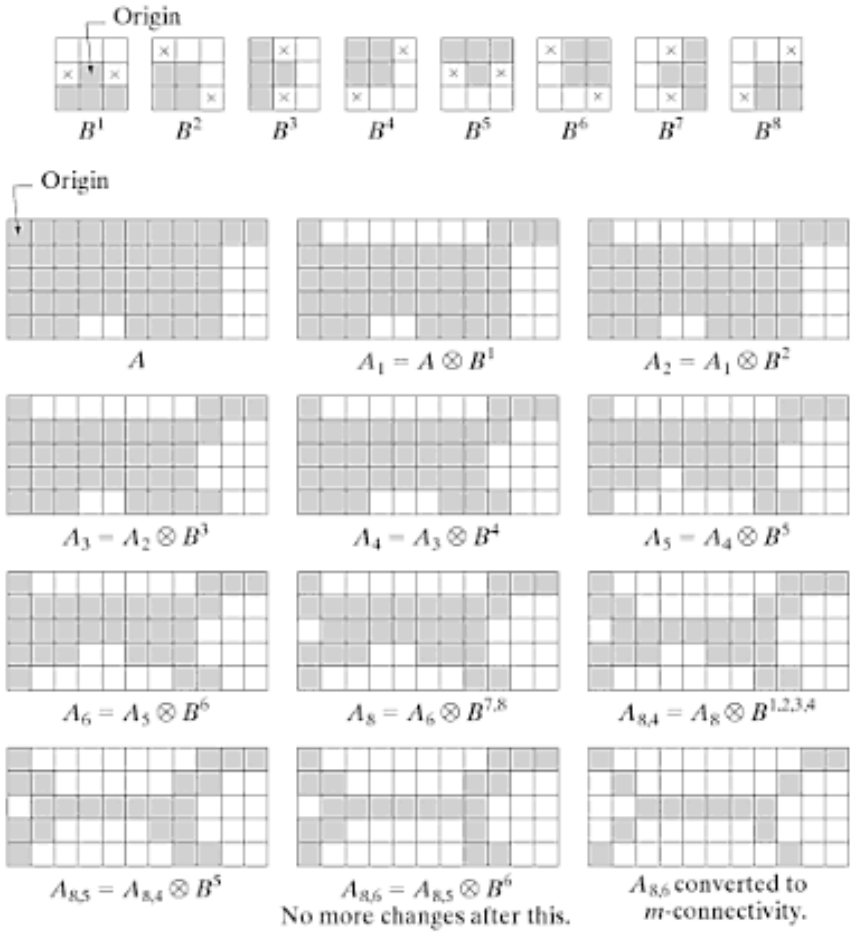


Figure 23 [15]

The structuring element can vary in appearance, and the choice of structuring element decides the application for the thinning operation. Also, some pruning or conversion to m-connectivity is also usually applied at the end to get rid off unwanted irregularities or spurts, and in the m-connectivity case to eliminate multiple paths.

2.2 Edge-linking algorithms

Edge detection should ideally yield sets of pixels lying only on edges. In practice however, these pixels seldom characterize edges completely because of noise and breaks. An example of this is simply the canny edge

output picture, presented earlier in this report. A linking algorithm therefore typically follows edge detection, and there exists three fundamental approaches to this, namely local, regional and global processing [16].

2.2.1 Local processing

Local processing is one of the most simple and intuitive approach to edge linking. You analyze the characteristics of pixels in a small neighbourhood about every point that has been declared an edge point by the presiding edge detection method.

In analyzing you look at the two principal properties used for establishing similarity of edge pixels, namely gradient magnitude and angle. Two pixels are similar if the difference in gradient magnitude and angle doesn't exceed a set magnitude and a set angle threshold.

Local edge linking methods usually start at some random edge point and consider points in the local neighbourhood of that point for similarity.
If the points satisfy the similarity constraint then the points are added to the current edge set. The neighbourhood based around the recently added edge points are then considered in turn and so on.

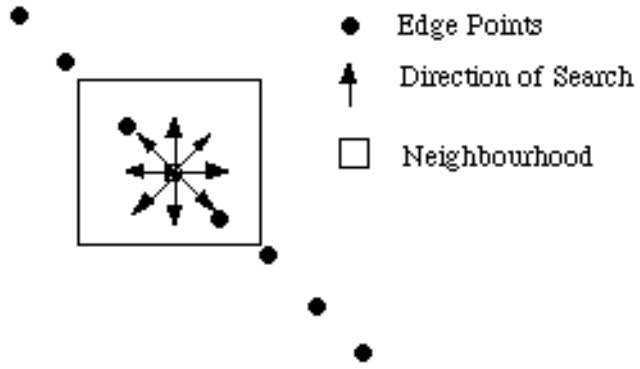


Figure 24 [17]

If no more points satisfying the constraint are found, we conclude that we are at the end of the edge and stop the process. Continuing, the method moves over to a new starting point, and the process is repeated until all edge points have been linked or at least considered for linking once. [18]
This is the main idea of local processing, but versions taking use of many different strategies have been adopted to control the search and selection process. Graph three search, dynamic programming and relaxation labeling techniques are some possibilities [20]

Gonzales chooses to describe a simple version in more detail, suitable for real time applications because of its computationally cheap implementation. It goes like this:

1. Compute the magnitude and angle arrays

2. Form a binary image whose value at any point is set to 1 if the point satisfies the magnitude and angle threshold.
3. Scan the rows of the binary image and fill all gaps in each row that do not exceed a specified length.
4. To detect gaps in any other direction, rotate the image by this angle and apply the scanning procedure again.

It is worth to notice that this application is best soothed for simple images, where our interest lies in horizontal land vertical lines. This is also the application found most frequently in practice. When our interest lies in numerous angle directions, it is more practical to combine step 3 and 4 into a single scanning procedure since image rotation is expensive.

2.2.2 regional processing

Often, the location of regions of interest in an image is known or can be determined. For example you know the end points of an edge and the points belonging to the edge, and you want to find the best way of fitting a straight line between the points to get a polygonal approximation.

A polygon is a closed figure made by joining line segments, where each line segment intersects exactly two others [21] The following are examples of polygons:

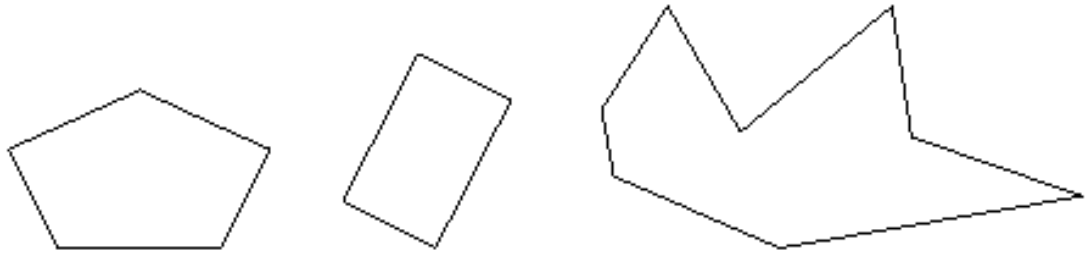


Figure 25 [19]

And these are examples of figures that are not polygons:

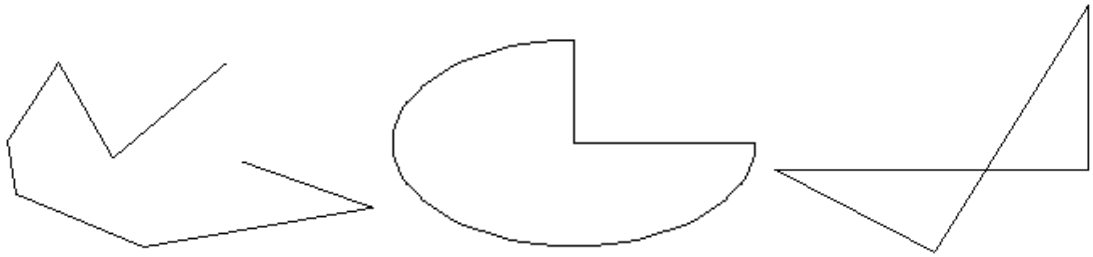


Figure 26 [22]

A real life example is an x-ray image of a human tooth [23]. You know the start and the end of the edge of the tooth in every case, and you also know that the edge should be complete and border the tooth. Using this information along with polygon approximation, a regional processing method, it should be relatively easy to construct the edge of the tooth.

At a first glance regional processing sounds like this might not be an alternative solution. The project problem space consists of all imaginable pictures and we don't have the luxury of that level of specific pre knowledge. Second thoughts give another picture. Even if we don't have any pre-knowledge, it doesn’t mean we can’t acquire the knowledge.

2.2.2.1 Polygon Approximation

Polygon approximation is attractive because they can capture the essential shape features of a region. This can come in handy if you are trying to achieve object recognition. One way of doing this is illustrated in Gonzales like this:

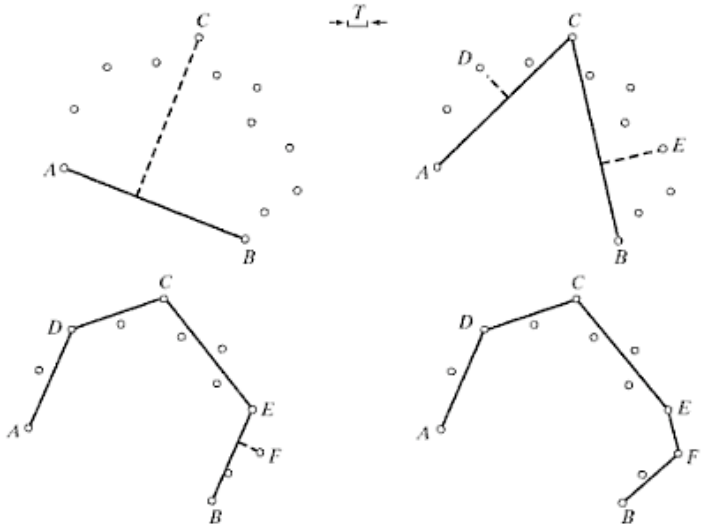


Figure 27 [24]

The points belong to an open curve, and we whish to approximate it. To do this you start by computing the parameters of a straight line passing through A and B. Then you compute the vertical distance from all other points in that curve to this line, and select the point that gives the maximum distance. If the distance exceeds a specified threshold, then the corresponding point is declared a vertex.

Here another version of polygon approximation, illustrated on a closed edge. The approximation is done while searching the edge, and knowing where the end pixel of the edge is not necessary.

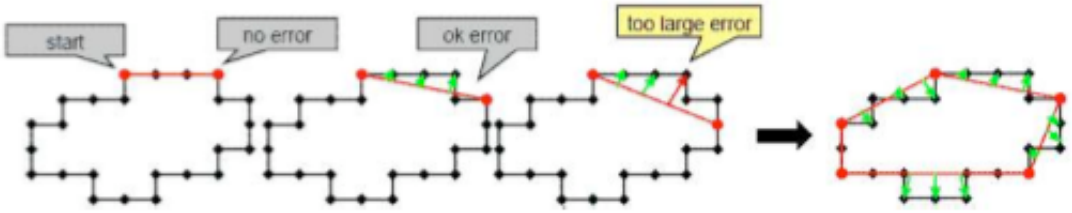


Figure 28 [25]

2.2.3 Global processing

The methods discussed so fare are applicable in situations where knowledge about pixels belonging to individual applications are at least partially available. For local processing this means that we have some idea of how big a neighbourhood we want for our implementation, for gap linking to be effective.

If gaps between pixels in an object are very large, local processing methods are not that effective. If all we have is an edge image and no knowledge about where objects of interest might be, then all pixels are candidates for linking and we need some way to consider all the edge points in the image at the same time.

Global edge linker does this, all the edge point of an image is considered at the same time and sets of edge points are sought according to some similarity constraints, or global property.

2.2.3.1 Hough transform

The most common and famous global processing method is the Hough transform. It is a technique that can be used to isolate features of a particular shape within an image. The classical Hough transform requires the desired feature to be specified in some parametric form, and is therefore most commonly used to detect regular curves, such as lines circles and ellipses.

A modification of the classical Hough transform enables the Hough transform to detect arbitrary objects as well, given a description of the shape true a model. This is called generalized Hough transform and is computationally more complex than the classical version, which is complex enough. Also, since we are considering all kinds of input pictures, and we have no knowledge of what kind of arbitrary shapes we are looking for I will leave off discussing generalized Hough transform any further.

The motivating idea behind the Hough transform for line detection is that each input coordinate point, indicates its contribution to the physical line, which gave rise to that image point [26] A simple example is fitting a set of discrete image points to a set of line segments, here illustrated with some possible solutions.

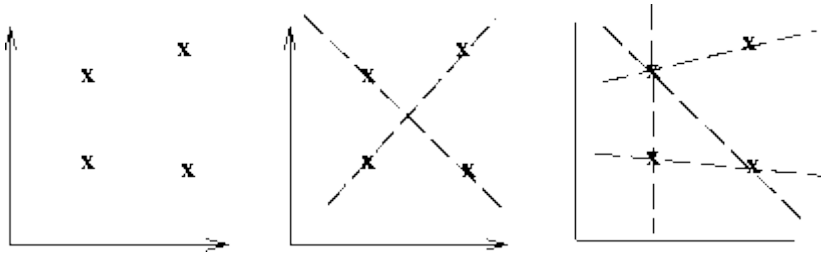


Figure 29 [26]

Further on, we know how to describe a line segment conveniently as the equation:

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

Equation 3

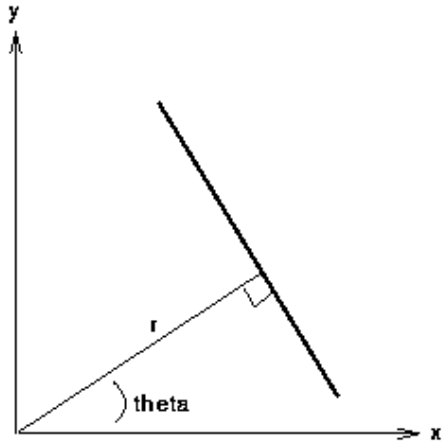


Figure 30 [27]

This makes r and theta the unknown variables we seek. If we plot the possible (r,theta) values defined by each (x,y) edge coordinate point, then we get a point-to-curve transformation. Viewing this Hough parameter space, the transform is implemented by quantizing the space into finite intervals of accumulator cells. Resulting peaks in the accumulator array represents strong evidence that a corresponding straight line exists in the image.

We can use the same procedure to detect other features with analytical description, like for example circles where the parametric description goes like this.

$$(x-a)^2 + (y-b)^2 = r^2$$

Equation 4

a and b are the coordinates of the center, and r is the radius.

To show you an case of practical use I will present a simple example.

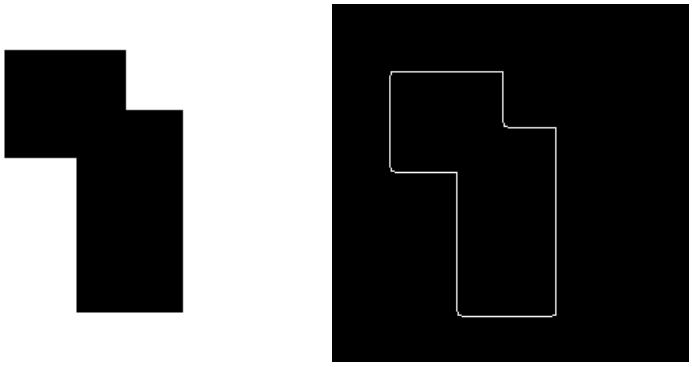


Figure 31 t.l Input 32, t.r picture after applying Canny edge detector [28]

The Hough transform gives us this accumulator array intensity image as output:

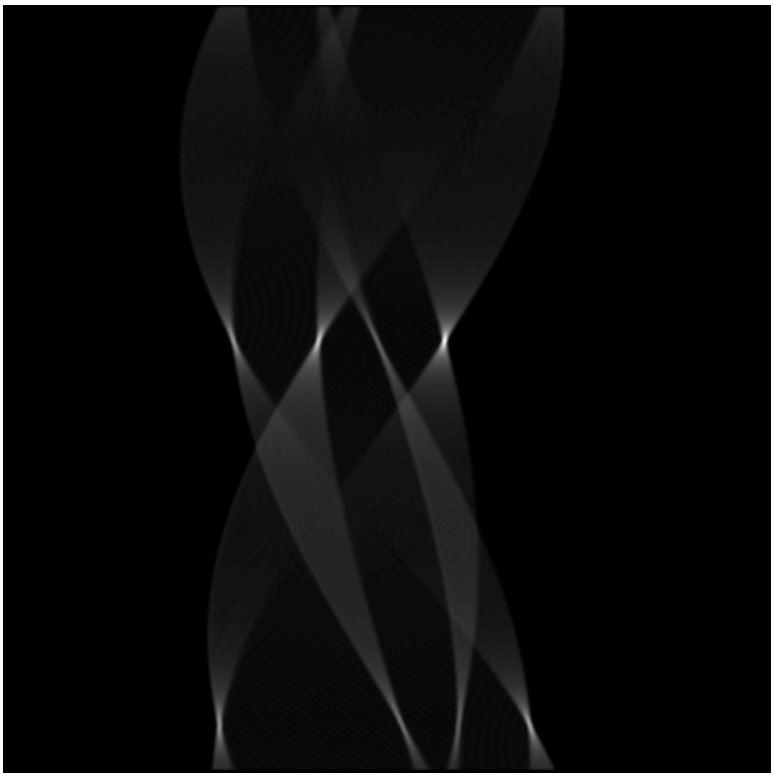


Figure 33

The eight separate straight lines segments are clearly visible as high intensity peaks.

Mapping back the Hough transform space into Cartesian space yields a set of line descriptions of the image. By overlapping these lines on the original image we can confirm the result.

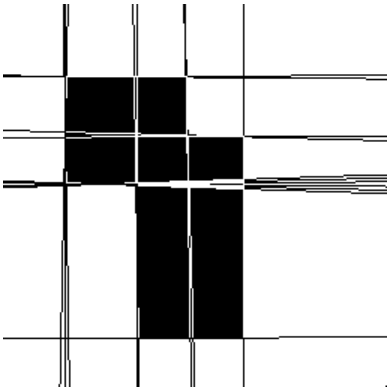


Figure 34

Note that the lines generated by the Hough transform are infinite in length, but that further image analysis, where you only include the portions of the lines that have some of the original edge pixels in them, will give us the actual lines of the image.

Looking at the good and bad aspects of this technique the most apparent arguments are the Hough transforms resistance to noise and large gaps, plus its ability to distinguish different shapes within a picture. The bad aspect is its computational complexity.

Chapter 3

Solution

In this chapter the choice of solution is described and the reason for choosing it is also given. We look upon the solution in more detail, and discuss new choices encountered concerning the implementation of the solution.

3.1 Edge-thinning algorithm

The first choice encountered one the path of making a sturdy edge and line detection algorithm based on canny edge detection is seemingly a small and simple one, namely the choice of edge-thinning algorithm.

As discussed in the alternative solutions chapter, after some literature study on the subject the candidates were thinning and skeletonization. Both skeletonization and thinning strives for a result consisting of 1-pixel thick edges, and looking at their output, they are ratter similar.

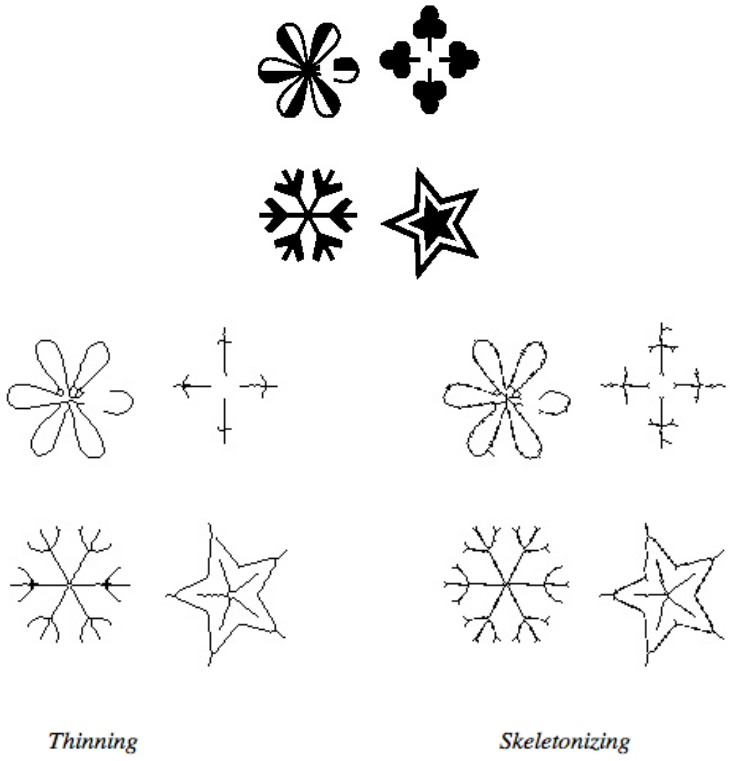


Figure 35 [29]

However skeletonization does seem to have a tiny bit more complex output than thinning, with a tendency to produce unwanted spurts and irregularities. Also, if you look at the aim of the two methods, you will find them to differ greatly.

Skeletonizing aims is to produce a skeletal remnant that preserves the shape and connectivity of the original region [30] It is consequently easy to see that the method is supposed to be used on objects and regions bigger than the output of an edge detection method.

This realisation, combined with observations from comparing the two methods, in addition to claims that thinning is the most common used morphological method for tidying up edge detector output [31], leads to the conclusion that thinning is the superior method.

3.1.1 m-connectivity

m-connectivity is the last step of the thinning algorithm, and is used to avoid multiple paths in the thinned edges. The last figure in the image thinning figure shows the result of such a conversion to m-connectivity. To achieve this you simply convolve the picture with another hit-or-miss transform.

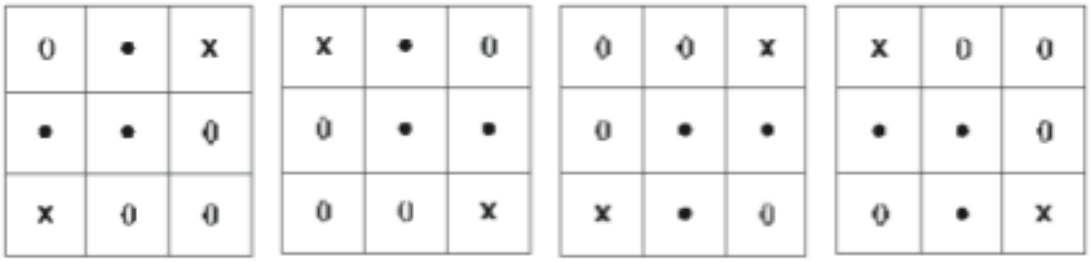


Figure 36 [1]

It is easy to see that no edges will be broken using this.

3.2 Edge-linking algorithm

Choosing between the different edge linking algorithms was not as easy as choosing an edge-thinning algorithm. Immediately Hough transform looks like the best candidate, for a number of reasons.

Looking at the problem definition the task is to develop a sturdy algorithm for edge and line detection for images with complicated and/or blurred objects. At the same time, it should be possible for the program to detect and identify simple objects.
As discussed in chapter 3, the Hough transform is both resilient to noise, and therefore sturdy. More so than local processing that relies on some knowledge of the picture to avoid big gaps. Or actually, you can make a relatively good

local processing method that doesn't relies on any information about the input picture, but you will get different performance depending on the form of the input, for example factors like distance between objects in a picture, overlap and so on.

While global edge linking methods are more complicated, they avoid merging of different contours into one object (which is a big problem with local search based processing) to a much larger degree than other processing will [32]

Taking into account the third demand for the program to be able to detect and identify simple objects, something Hough excels at, or ratter is tailored for, Hough seems the clear-cut best choice. While Hough transform is not perfect, it is still fare superior to the other alternatives, considering the problem definition.

Hough biggest weakness is its sophisticated calculation, something that has prevented effective real-time application of the Hough transform for a long time. (Recently a Kernel based Hough transform has been proposed to improve this). This suggests that a need for low computation cost applications of edge and line detection, plus object recognition might exist.

Also even though Hough is good at identifying clear features that can be specified in parametric form it is not good for identifying arbitrary objects. The problem definition states that the program should be able to identify simple shapes, but it also says that the input images should consist of complicated objects. This is slightly contradicting and can be interpreted as simple object recognition is expected possible, but that identifying complex objects, like for example a human, as well should be tried implemented.

To achieve this, maybe the best solution is a combination of the above methods? Global processing to identify the clear-cut shapes and lines of a picture combined with edge tracing and polygon approximation for the arbitrary objects, or blobs, as I will referee to them from now one.

Hough is not a good method for blob recognition, as the generalized Hough transform that deals with them require a model of the blob before it can identify it. Maybe implementing local processing first, as this is the most intuitive, and not as difficult method, is the best way to go?

In addition to all this local processing seemed intriguing to implement. In the pre-project I went for the clear-cut choice method, the canny edge detector, and implemented it step by step, with not much room for adaptation and experimenting. Going for the Hough transform would be pretty much the same thing. The simpler local processing combined with regional processing approach just seemed more appealing, with many choices for implementing the search and the preceding image analysis. It looked like, its minimal definition made room for experimentation and improvement.

The best choice would be to implement them all, starting with the simplest and building upon it within the reach of once ability and time limit. This along with

the blob argument, and my early interpretation of the problem definition resulted in me going with local processing as choice of solution. Despite Hough transform being the “correct” solution.

Chapter 4

Implementation

This chapter will clarify shortly the tools used in the implementation. It will also give an overview of the structure, main methods and principle behind the finished program. Actual experiments and results will be discussed in the next chapter.

4.1 Programming tools

The programming was done in C++, an excellent tool for image processing, as computation cost can become pretty high. The program was written on a Mac in the programming environment Xcode, version 3.0.

Tools used for the image loading and handling was SDL, Simple Directmedia Layer. This is a cross-platform multimedia library with OpenGL.

The drawing of the edges, after getting all the vertices in place was done using platform independent OpenGL methods.

The two are included like this

```
#include "SDL.h"
#include "SDL_opengl.h"
```

Some problem was encountered with the two using different definitions of RBG color.

For the SDL part of the program this definition had to be used

```
typedef struct {
unsigned char a, r, g, b;
} pixel;
```

While the openGL part only worked with this definition.

```
typedef struct {
  unsigned char b, g, r, a;
} pixel;
```

This made it easier to separate the two parts into two programs, and is the reason for doing so.

4.2 Program structure and main methods

Starting with the first part, namely the edge detection part, the canny edge detector is implemented using five self-explaining methods:

```
void ModifyImage(SDL_Surface *surface, SDL_Surface *copy, SDL_Surface *copy2){

GreyScaleHighestRGB(surface, copy);
GaussianSmoothing(surface, copy);
Gradient(surface, copy);
NonmaximaSuppression(surface, copy);
HysteresisThresholding(surface, copy, copy2);

//pre-project (some bug fixing done)
//----------------------------------------
```

Afterword comes the new additions.

```
//----------------------------------------
//master

Thinning(surface, copy);
MConnectivity(surface, copy);
MakeFrame(surface);
Linking(surface, copy);
GetRidOfSmallEdges(surface, copy, 20);
LinkGaps(surface, copy, 10);

//ColorEdges(surface, copy); //tool for viewing results easily
```

Thinning and m-connectivity has been discussed throughout, one small addition being a bug discovered in the choice of m-connectivity chosen. If you look at the original hit-or-miss filters contains 0 and X “don't care” values. Actually all of these should be “don't care” values, or it is easy to get multiple paths in the instances that the edge has contacts with other edges.

Make Frame is to get rid of noise in the frame when the algorithm interprets the frame as an edge in itself.

Linking, is the main part, and contains the local processing edge search. It works simply as explained earlier in this report by searching the picture for edge points that it uses for start points of an edge. It traces the edge as long as there are neighbors available, and end in an end point if not. The edge traced is deleted form the picture and next start point is searched out.

A record must be kept of the linked points, and a simple bookkeeping procedure for this is used by assign different intensity values to each set of linked edge pixels. Also each edge point has a value defining if it is an start, middle or end point. If the edge forms a closed formation, the start point is sett to be the SaE value,that is Start and End point.

After linking the picture is gone true and edges shorter than 20 pixels are deleted.

Then, the edge liking continues with gap linking. Every end and start point are searched in a 10 x 10 neighborhood, and candidates for linking are found here. The candidates are chosen based on four cases. First case is if the pixel is in the 3x3 neighborhood, then it is preferred over other pixels. Case 2 is if the pixels share the same angle value, hen this is prefer over other candidates. Case 3 is similar (but not the same) angels, and Case 4 is the closest one, if non of the above cases where achieved.

Moving over to the shape detection and edge drawing part of the program we have the method

```
void DrawEdges(SDL_Surface *image, SDL_Surface *copy, int pictureWidth, int pictureHight);
```

This extracts one and one edge, calls

```
Int info = ShapeRecognition(copy, x, y, questionErrorLimitPolygonAdaption, c1, c2, c3, cornerLimit, strictCornerLimit, smalestDegreeTriangel, squareSideLimit);
```

And asks what shape the edge has. It then draws the edge with a color according to its shape.

The different instances are:
0 purple - straight line
1 and 2 green – semi straight lines
3 yellow – triangle
4 red – square
5 blue – blob (closed shape, not a triangle or square)
6 green – everything else

The number here represents number of vertices encountered in the edge when it is drawn with a pretty “angular” polygon approximation.

After getting this information DrawEdges() starts drawing the actual edges using a much less “angular” polygon approximation.

This is the main structure of my implementation.

Chapter 5

Testing, Evaluation and Conclusion

This chapter evaluates and tests the resulting program.

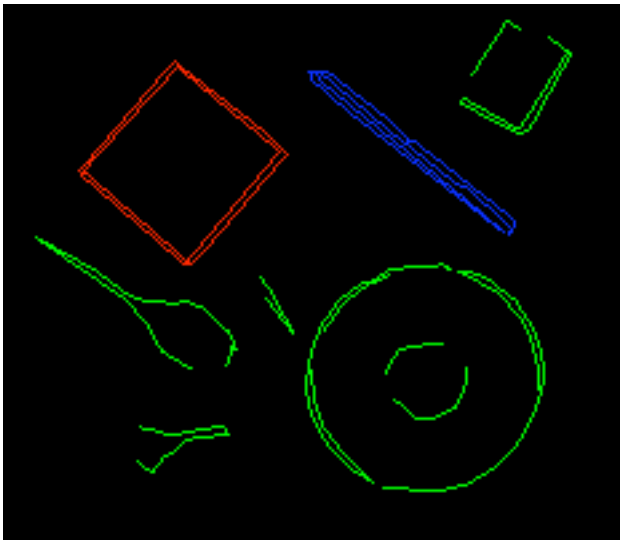
5.1 Output Edges and its classification

Looking at different input images an it's output I will now evaluate the performance of the edge detection implementation.

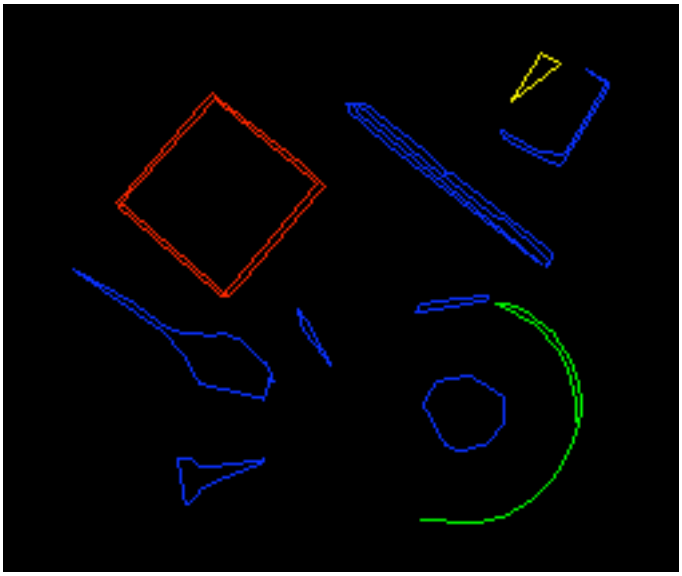
Taking a lock at the NTNU main building example again, you will see the progress made.

The different coolers are due to classification of the shape of the edge outputs. Most of them are unclassified, and green, while a few blue blobs has made its way into the output, along with straight, or semi straight lines collard purple.

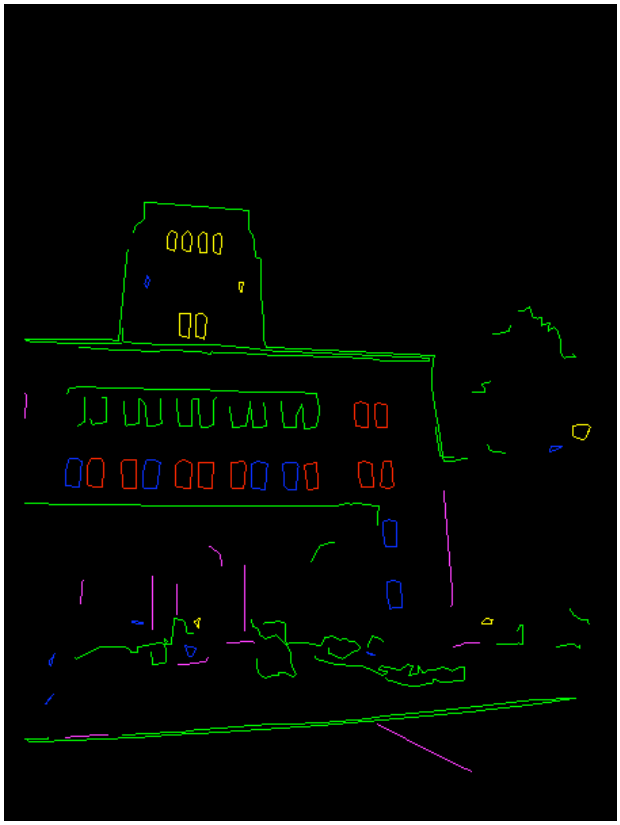
A better example to use is this simple picture.



Here shanging the distance for gap linking to a higher number gives better results.



And example of a problem encountered with my implementation, is false or wrong counting of edge vertices.



A you can see are half of the windows classified as blobs or triangles. I have been unable to identify the bug causing this problem.

At last one more example, of good performance from my implementation, in in its place.

As for the evaluation of my results I think they could have been much better if I had gone with Hough transform instead. Even so working with local and regional processing has been fun, and I have learned a lot about image processing in the process.

My conclusion would be that local processing is good for tracing all edges, not just geometrical once, but it is probably not the best approach for object detection.

References

[1](http://www.cis.hut.fi/Opinnot/T-61.5100/laskarit/dkkexer6_ans.pdf)
[2](http://www.csse.uwa.edu.au/~pk/research/matlabfns/LineSegments/example/)
[3](http://moscoso.org/pub/video/opencv/svn/opencvlibrary/trunk/opencv/doc/ref/opencvref_cv.htm)
[4](http://en.wikipedia.org/wiki/Feature_detection_(computer_vision)
[5] (Rafael C. Gonzalez and Richard E. Woods, Digital Image Processing, Third Edition, 2008, page 694)
[6](http://homepages.inf.ed.ac.uk/rbf/HIPR2/canny.htm)
[7](Rafael C. Gonzalez and Richard E. Woods, Digital Image Processing, Third Edition, 2008, page 719)
[8](http://en.wikipedia.org/wiki/Edge_detection#Approaches_to_edge_detection).
[9](http://en.wikipedia.org/wiki/Canny_edge_detector, Parameters)
[10] (http://www.mathworks.de/company/newsletters/digest/2008/nov/motion-stereo.html)
[11] (http://homepages.inf.ed.ac.uk/rbf/HIPR2/skeleton.htm)
[12](http://homepages.inf.ed.ac.uk/rbf/HIPR2/thin.htm)
[13](http://homepages.inf.ed.ac.uk/rbf/HIPR2/thin.htm)
[14](http://homepages.inf.ed.ac.uk/rbf/HIPR2/thin.htm)
[15] (Rafael C. Gonzalez and Richard E. Woods, Digital Image Processing, Third Edition, 2008, Chapter 9.5.5)
[16] (Rafael C. Gonzalez and Richard E. Woods, Digital Image Processing, Third Edition, 2008, Chapter 10.2.7)
[17](http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MARSHALL/node31.html)
[18](http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MARSHALL/node31.html)
[19] (http://www.mathleague.com/help/geometry/polygons.htm#polygon)
[20](http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/MARSHALL/node31.html)
[21] (http://www.mathleague.com/help/geometry/polygons.htm#polygon)
[22] (http://www.mathleague.com/help/geometry/polygons.htm#polygon)
[23](Rafael C. Gonzalez and Richard E. Woods, Digital Image Processing, Third Edition, 2008, Chapter 10.2.7)
[24] (Rafael C. Gonzalez and Richard E. Woods, Digital Image Processing, Third Edition, 2008, Chapter 10.2.7)
[25] (http://folk.ntnu.no/stormark/nambafa/depot/bildetek.pdf)
[26] (http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm)
[27] (http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm)
[28] (http://homepages.inf.ed.ac.uk/rbf/HIPR2/hough.htm)
[29] (http://www-scf.usc.edu/~zhangxia/my_projects.html)
[30](http://homepages.inf.ed.ac.uk/rbf/HIPR2/skeleton.htm)
[31] (http://homepages.inf.ed.ac.uk/rbf/HIPR2/thin.htm)
[32] (http://www.lana.lt/journal/19/Atkociunas.pdf)