

**ACQUIRING CURVES
FROM HAND DRAWN LINE ART**

**FALL SEMESTER 2002
INDEPENDENT RESEARCH**

**BY
WILKIE KIEFER**

**ADVISOR
PROFESSOR ADAM FINKELSTEIN**

ABSTRACT

This paper presents a method by which one can acquire accurate curve data from a pixel based image of line art. The process mixes a few key insights and image processing techniques with some modifications to the Canny Edge Detector algorithm[2] to provide promising results.

INTRODUCTION

Translating a pixel based image to a data based system of curves is not a simple task. Whereas the reverse transition is relatively simple, many problems arise when attempting to represent pixels as mathematical curves. However, the ability to make such a conversion is extremely useful and provides numerous useful applications in various fields such as animation, art, and design.

In animation, for instance, this conversion could allow for quick and easy generation of multiple frames from a single character or cartoon image. After acquiring the curves from a hand drawn image, a computer could alter the curves of the hand, leg, face, or any other part of the character and derive a new image or movement from the original drawing. This method prevents the continuous redrawing of images and simplifies current animation methods for cartoon movement.

A second application of curve acquisition is the restoration of old deteriorated images. In 1914, Winsor McCay presented the world with one of the first major animations, *Gertie*. This series of over 4,000 frames features a lovable dinosaur doing various tasks and tricks for the audience[7]. Unfortunately over time, many of these images have become dirty, faded and lost. Curve acquisition from these hand drawn frames can aid to restore and clean up many of the dirty frames as well as help to fill in missing sequences.

One key insight into this procedure is that the curves, in essence, are given in the image. Acquisition becomes a matter of precisely determining where the mathematical curves lie within that context so that they best represent the lines and curves in the image. The problem is that the so-called lines and curves in the original image come in an infinite variety of sizes, shapes, shades and textures. This paper will first examine the general approach and the specific method of the researched curve acquisition method. Next, the results of the method will be presented and discussed. Since there is no previous or definite method for pixel to curve translation, the goal of this paper will be to discuss and explain the reasoning behind various decisions made throughout

the research and implementation of the given method.

PREVIOUS WORK

Very little research has been conducted with regard to this specific problem. There exist many various edge detecting techniques in numerous graphic applications, however, little work has been done using edge detection as a method to create curve data. However, this has its advantages and disadvantages. While on one hand, there is a lot of freedom and room for research and development, on the other hand there is little information regarding other methods. Thus deriving an accurate and unique method without having previous research to examine makes development difficult.

APPROACH

The basic approach used in my final implementation is comprised of three main stages. The first stage runs a noise reduction filter over the original image. This noise reduction step is crucial in restoration applications and also very important in general. Since the hand drawn images will most likely be scanned, noise reduction removes dirt, hair, and other scanning errors from the image before attempting to determine curves. However, the main goal of the research is to find an appropriate method for translating pixels to mathematical curves. Thus, this step remains relatively simple and can easily be upgraded to more advanced noise reduction algorithms.

The second stage in the proposed method is a specialized adaptation of a commonly used edge detection algorithm, the Canny Edge Detector[2]. In the Canny Edge Detector, edges are defined as places in the picture where there is an abrupt change in color. In other words, edges are the outlines of the shapes in the picture. The output of the Canny Edge Detector is a white image with black lines one pixel thick representing the edges determined by the algorithm. This edge detection algorithm uses the first derivative of the image to compute a magnitude and direction for each pixel. Gradient fields representing the image are computed using these magnitudes and directions and from these gradients, the Canny Edge Detector determines which pixels are part of an edge and which are not.

The Canny Edge Detector works well for pictures and images, but for hand drawn line art the output is not helpful. Since the detector finds only edges, the output results in parallel double

lines everywhere there was a hand drawn line in the original image (see Fig. 10). To deduce curves from these double lines is difficult and ineffective. What in fact I needed was just a one pixel thick curve everywhere there was a line in the original image.

Thus, the best approach is to use the second derivative of the original image to create the gradient images. The second derivative provides strong gradients where the lines actually lie rather than at their edges, producing exactly the desired result. However, when moving from first to second derivatives, the computation of magnitudes and directions for all the pixels becomes more complex. Despite the more complicated computational aspects, adapting the Canny Edge Detector to use second derivatives tricks the algorithm into finding edges exactly on the lines of the original image.

The third and final stage for my approach is to examine the edge image output of the Canny Edge Detector and determine each individual curve. A recursive function is used to trace along each edge and store the data points in a curve structure. Thus, along with the edge image, the output of this step is a system of curve data structures representing all the curves in the image. The curve data structure is adaptable to store additional information such as texture or thickness if needed.

METHODOLOGY

There are various ways to implement a noise reduction filter. In this implementation, after the original bitmap image is converted to grayscale form, a 5 by 5 Gaussian blur matrix is run over the entire image. In the resulting blurred image, all the pixels below a certain threshold are replaced with pure white while all the pixels above that threshold return to their original value. This noise reduction technique is good for removing specks of dust and little imperfections in the image. Lone dark pixels surrounded by white ones tend to get filtered out. The advantage of this noise reduction technique is that it is easy to implement and tends to removed the speckled type noise caused by the scanning of an image. Since the noise reduction step is implemented independently of the remaining steps, it is simple to replace with a more advanced noise reduction step if needed.

The Canny Edge Detector is comprised of four main steps: blurring the image with a Gaussian filter, computing the derivative, magnitude, and direction for each pixel, non-maximum suppression, and hysteresis. First, the Gaussian filter blurs the image just enough to get smooth

gradient images in the second step. In that second step, a simple first derivative filter is used to calculate direction and magnitude for each pixel. The magnitude image and the directional image are then fed into the third and fourth steps. In non-maximum suppression, points that are not local maxima are suppressed. The output of step three is then examined in hysteresis. Points above a certain high threshold at this stage are considered to be on an edge. Hysteresis then traces these edge pixels and any points adjacent to an edge pixel that is above a certain low threshold but below the previous high threshold is thought to be a continuation of the edge and is therefore included.

The special adaptation of the Canny Edge Detector used for my approach differed only in the second step, the calculation of the magnitude and directional images. So rather than implement a Canny Edge Detector from scratch, I used the source code to a preexisting implementation of the edge detector written by Michael T. Heath and adapted it to fit my approach[5].

Basically for this step, I had to implement the calculation of the second derivatives and use the result to calculate the magnitude and directional images. This implementation is more complex than the original Canny Edge Detector because when using second derivatives, one must calculate four partial derivatives rather than two. Figure 1 shows the first derivative filter used in the original edge detector and Figure 2 shows the second derivative filters used in my implementation.



$$\begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} dy^t \\ dx^t \end{bmatrix} = \begin{bmatrix} 1 & 0 & -1 \end{bmatrix}$$


Figure 1.

$$\begin{bmatrix} dxx \\ dxy \end{bmatrix} = \begin{bmatrix} dyy^t \\ dxy^t \end{bmatrix} = \begin{bmatrix} 1 & -2 & 1 \end{bmatrix}$$


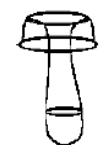
$$\begin{bmatrix} dxy \\ dyx \end{bmatrix} = \begin{bmatrix} dxy^t \\ dxx^t \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$


Figure 2.

The graphical representation of these derivative filters (shown to the right of each figure) can help visualize why the second derivative, rather than the first, accentuates lines in the derivative images.

Second derivatives are commonly represented by a Hessian matrix. This symmetric 2 by 2 matrix contains each partial derivative; dxx and dyy lie on the diagonal while dxy and dyx , which are always the same in this case, lie on the remaining reverse diagonal. The magnitudes of the second derivative are the eigenvalues of the Hessian matrix. Because the Hessian matrix is 2 by 2, it has exactly two eigenvalues, or magnitudes. The direction of each magnitude is represented by a 2 by 1 eigenvector associated with each individual eigenvalue[1].

The implementation of these calculations are as follows: The derivative filters are run over the noise reduced image. All four partial derivatives for each point are stored in a structure containing four lists, one list for each partial derivative. Since each point has its own unique number, the array implementation of storing partial derivatives is very efficient for retrieving data.

After many hours of work, attempting to implement various methods of computing the eigenvalues and eigenvectors of a 2 by 2 matrix, I found the GSL library[3][4]. This library contained very accurate and concise ways of calculating eigenvalues and eigenvectors, as well as very useful matrix and vector structures. Thus, in my final implementation, I used this library for the eigen calculations.

The disadvantages of my previous eigen calculation code was that it was not very precise and difficult to write. The GSL library provided for a very accurate alternative. However, one possible drawback is that math libraries such as GSL are optimized for much larger matrices. Depending on the GSL implementation, it is most likely inefficient to call library functions for such a small Hessian matrix. However, the advantages of accuracy and ease far outweigh the drawbacks in performance.

Nevertheless, at this point in my implementation, two eigenvalues and two eigenvectors are accurately calculated for each point. However, the final steps of non-maximum suppression and hysteresis require exactly one magnitude and one direction for each point. Thus, deciding which eigenvalue and eigenvector to use becomes an issue. Fortunately, in all the cases tested, there was exactly one negative eigenvalue and one positive eigenvalue. Choosing the positive eigenvalue and its associated eigenvector provided the required results. At points that are nowhere near an edge, both eigenvalues are zero and the question of which one to choose does not matter.

Thus, the chosen eigenvalue and its associated eigenvector for each point is stored in a structure containing three arrays, one eigenvalue array and two eigenvector arrays. This data is then fed into the remaining two steps of the original Canny Edge Detector process, non-maximum suppression and hysteresis. The final output of the Canny Edge Detector is a black and white image with lines one pixel thick everywhere the algorithm determines a line to be.

The third and final stage of my curve acquisition implementation is to actually acquire curves from the output image of the Canny Edge Detector. Here a recursive algorithm, set within a loop to examine every pixel, was used to trace along curves, adding a point at a time to the

curve data structure. The algorithm is summarized as follows:

```
follow_curves( )
{
    for ( every adjacent pixel )
    {
        if ( pixel is an edge pixel )
            if ( adjacent_count < 2 )
                add pixel to current curve, make pixel white
                follow_curves( new pixel )
            else
                make new curve
                add pixel to new curve, make pixel white
                follow_curves( new pixel )
            adjacent_count++
    }
}
```

Figure 3.

The `adjacent_count` in the above algorithm is important for preventing forks in curves. Since most likely the first point found on a curve will be a middle point, having two adjacent pixels allows for the capture of the whole curve. If only one adjacent pixel were allowed a single line would be divided into two curves, both leaving from the same middle point. In the algorithm, if a point has three pixel trails leaving from it, that point is an intersection, and curves by definition are not allowed to contain forks. I implemented this so that the third trail encountered is decided to be the start of a new curve structure, although this was not the only option. Another implementation I considered would ask the user for clarification in such a situation. However, this forking situation happens extremely rarely due to the process of edge detection. The canny edge detector does not consider intersections to be part of an edge. Therefore, any time two lines intersect, the resultant image is actually missing the few pixels that would form the actual intersection. This attribute of the edge detection is beneficial for curve acquisition because it alleviates many sources of confusion. However, the drawback is that an obvious curve to a human spectator that happens to cross over another line will sometimes be split into two curves due to the intersection phenomenon. However, there are still some rare cases where a curve will fork into two directions. After weighing the options, I determined that for efficiency and user ease, the situation was rare enough that the computer should decide what to do in these situations.

After the recursive algorithm completes, all the curves are stored in a bank containing the total number of curves. Each curve structure contains the number of points in that curve, as well as a list of the actual points. There is room in the structure for additional data to be stored. The

final output of my implementation produces the edge image and a text file containing all the curve information.

RESULTS

The goals of testing and revision were to obtain accurate results and produce valid curve structures. To be more precise, accurate results were those that did not contain extraneous curves, were not missing curves, and split the hand drawn lines into the minimum number of curves possible.

Initial results before testing and revision were discouraging. The resulting curves were broken up into an unreasonable amount of smaller curves and much of the detail of the original images was totally lost. In addition, the curves tended to appear in triples. The main curve representing a line would be accompanied by many fragments of two parallel side curves. After reworking the derivative and eigen calculations, results began to look promising. Many of the earlier problems subsided with more accurate calculations.

Three main variables affect the output of the program: the radius of the gaussian blur, the edge low threshold and the edge high threshold. In order to produce the best results, these three factors had to be just right. It turns out that a very small radius of the gaussian blur produces the most accurate results. Since the lines in hand drawn art tend to be fairly obvious, a large gaussian blur is not needed. J. F. Canny stated that ration of the high to low threshold in his implementation was roughly two or three to one. After extensive testing of values between 0 and 1, it turns out that a low threshold of .50 and a high threshold of .80 produces the best results. The ratio is a little lower than Canny suggests, but since our adaptation is quite different from the original Canny Edge Detector, this is no surprise.

The testing comprised of keeping all but one variable constant and trying various combinations. The objective measure of success used was just a visual examination of the resulting edge images. Pixel by pixel and curve by curve, I compared results to both other results and to the original image. The thresholds of .5 and .8 provided images closest to my guidelines for accurate results. Below is an original image alongside the resultant image.

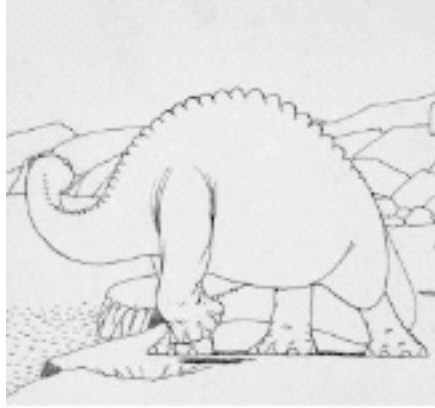


Fig. 4. Original[7]



Fig. 5. Resulting Curves

As I approached my final implementation, I began to notice that certain curve angles were not remaining contiguous. So I decided to run some performance tests on my method. First I created an image consisting of nothing but lines of varying angles to see how my method performed.

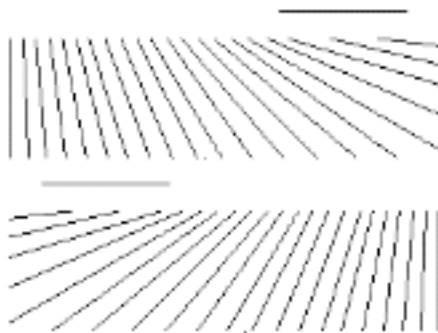


Fig. 6. Original

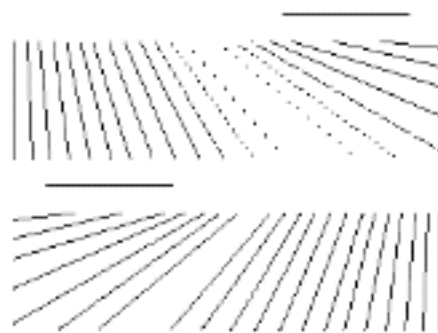


Fig. 7. Resulting Curves

As illustrated above, lines with angles of around -45 degrees have the worst performance, they do not appear at all. Also a line of 225 degrees disappears in the lower row. For a second test, I wanted to see how line width affects the output of the program. Below is a sample test of line width where the lines vary from one pixel wide to ten pixels wide:

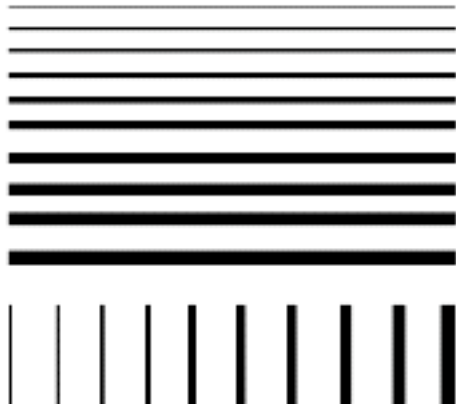


Fig. 8. Original

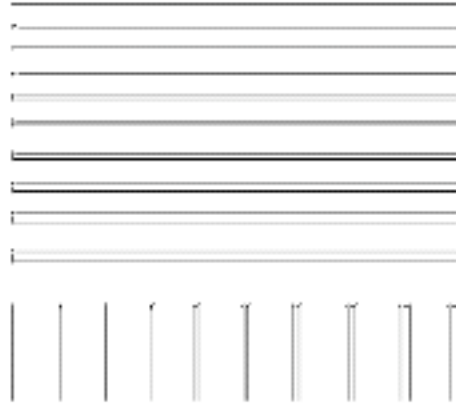


Fig. 9. Resulting Curves

This test shows that the method performs accurately when the original line has a width of 1 to 4 pixels. When line width increases over five pixels, double curves begin to develop.

On last interesting result of adding the `adjacent_count` variable to the recursive function described in Figure 3 was a severe reduction in the number of curves produced. An image that previously had 341 resulting curves was reduced to 193 curves. An image with 64 original curves lowered to just 25 curves. The output edge image was no different, the only change was that the curve acquisition step was more accurate and split curves into piece much less than the previous method.

DISCUSSION

Overall, the approach and method used for curve acquisition looks extremely promising. The images my final method produces are immensely better than the early results. Take for instance the dinosaur image of Figure 4 run through Canny Edge Detector before any adaptation (Figure 10). Such initial results would have been useless for any application. However, the current results are more accurate and only leave out bits and pieces of original lines. There is ample room for enhancements and improvements to my method. Each of the three main stages is independent, so drastic improvements to one stage would not affect the other stages.

There is still follow-up work that can be done with this method. The first and foremost priority would be to solve the problem of certain line angles disappearing. Such a solution might not be obvious but would almost perfect this method of curve acquisition. Additional work could definitely be done on the noise reduction stage, especially for restoration purposes. One good

idea would be to iterate between noise reduction and curve acquisition. Better knowledge of where curves lie would aid in noise reduction and better noise reduction would aid in better determining where curves lie. Thus, an iterative process could be useful to filter out noise in the original image. One last follow-up that I believe would be helpful would be to add on a basic user interface to aid the curve acquisition. For example, if the user, when presented with the final result, could remove curves by clicking on them, he or she would have more control over the acquisition and noise reduction.

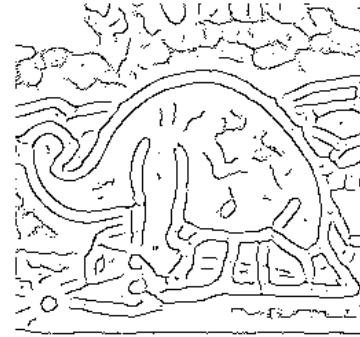


Fig. 10. Result of Figure 4 run through unmodified Canny Edge Detector

This research project was a fantastic experience for me to submerge myself in the world of computer graphics for the first time. Never having programmed or worked with computer graphics on anything deeper than the user level, I learned an extraordinary amount about the general principles of two dimensional graphics. After spending a considerable amount of time learning sufficient background information about graphics, I still feel that I came a long way with my research. I believe the method described above is very promising and has bountiful applications.

CONCLUSION

In summary, the method described in this paper allows for the translation from pixel based hand drawn line art to a system of curve structures by means of three main stages: noise reduction, modified Canny Edge Detection and curve acquisition. The results are accurate and the approach is quite promising for various applications. However, this method is by no means perfected. Many enhancements could be made and there still remains one unsolved issue of certain angled curves disappearing. Nevertheless, the method presented in this paper is a fully functioning and promising way to acquire curve data from hand drawn line art.

REFERENCES

- [1]Bretscher, Otto. *Linear Algebra with Applications*.
- [2]Canny, John F. A computational approach to edge detection. *IEEE Trans. Pattern Analysis and Machine Intelligence*, v.8 n.6, p. 679-698, Nov.1986.
- [3]Galassi, Mark, et. al. *GNU Scientific Library: Reference Manual*. Ed. 1.3.
[http://sources.redhat.com/gsl/ref/gsl-ref_toc.html]. Dec. 2002.
- [4]Galassi, Mark, et. al. *GSL Library source code* retrieved from the world wide web
[<ftp://ftp.gnu.org/gnu/gsl/gsl-1.3.tar.gz>]
- [5]Heath, Michael T. *Canny Edge Detector source code* retrieved from the world wide web
[ftp://figment.csee.usf.edu/pub/Edge_Comparison/source_code/canny.src]
- [6]Trucco, Emanuele and Alessandro Verri. *Introductory Techniques for 3-D Computer Vision*. Upper Saddle River: Prentice-Hall, Inc., 1998.
- [7] [<http://www.gertie.org>]. Dec. 2002.