The Applications of Image Processing Techniques to Sign Language Recognition Through a Web Camera Interface

Byron Hood

January 23, 2008

Abstract

Sign language recognition is the first step in a long road towards natural language processing, or the ability for a computer to "understand" naturally spoken (or signed) language. Such an invention would drastically lessen the amount of time required for user-computer interaction, by as much as a factor of two. This project explores using image processing techniques such as edge detection, line detection, and line interpretation to identify sign language as it is performed, using frame-based input from an average web camera ("webcam"). When research and programming is complete, it is expected that the program will be able to identify the sign language gestures for alphanumeric characters with a high degree of accuracy, in real time.

1 Introduction

1.1 Purpose

The purpose of this project is to provide an interface for people who use sign language for everyday communication with others (referred to as “sign speakers”) to enter input into a computer using their native form of communication. Such input could theoretically increase their input speed twofold (see the explanation under Background) or possibly even more. In addition, such interpretation of sign language gestures and hand positions is a first step towards fulfilling a dream of human-computer interaction nearly as old as the machines themselves: natural language processing, or the concept of a computer “understanding” naturally spoken (or in this case, signed) language and performing actions based on the gestures or speech interpreted.

1.2 Scope

This project will require some research into sign language hand positions and gestures, but more specific and deeper research is necessary into “image parsing” techniques. These include edge detection, line-finding, and methods of line classification. Additionally, as the program will attempt to insert characters into the computer’s input system, this will require some heavy digging into either windowing system input code (so that the program can feed into X windowing system input) or operating system-level input (so that the program can effectively emulate a keyboard or other such input device). Further, considering that data concerning hand positions will be stored in XML files to be read and parsed during the running time, to ease the process of editing hand positions, and also to make reading the said data simple when starting up; yet the project will also need some basic research into XML parsing schemes and methods of storing the XML data in memory.

To limit the overall complexity of the program, the final application will recognize and interpret only alphanumeric characters. The further this program goes in terms of recognizing additional characters, the more hand positions necessary to differentiate all of these different symbols. If the program continues to expand, the hand positions, by necessity, will become closer to each other and distinctions between different characters are made smaller and more difficult to determine. This, in turn leads to a higher rate of error; error that increases exponentially as one adds symbols to be recognized. A more practical approach, therefore, is to designate a reasonable number of different characters, and then to extend this basic program later on as image processing techniques improve and the rate of error decreases.

2 Background

In today’s society, people with auditory and locutory disorders usually opt to commincate using sign language, a silent variation on the local language which uses body language: gestures, mouthing, and hand/finger positions, instead of spoken words. Through extensive practice and use (as normal people might gain extensive practice speaking their native language), many sign speakers are capable of “speaking” as fast as non-impaired people speak orally, about 200 words per minute in normal conversation[2].

If the average word is taken to be six letters long and one accounts for a slight speedup due to the short amount of time required to communicate a single letter, spelling a word out letter by letter will likely reduce speed by a factor of four for both sign and oral speakers. Nonetheless, this is a hefty 50 words per minute, and compares quite favorably to average typing speeds. According to Karat, et. al. the average computer user can type 33 words per minute while copying text, and this drops to a mere 19 when composing[3]. Therefore, an average computer user will spend from two to three seconds typing any given word, whilst a sign speaker (could he or she sign into a computer), would spend half of that time. Finally, the "QWERTY" keyboard was designed expressly for the purpose of *slowing* typists down (this is a throwback to the days of typewriters, to help prevent jams). Therefore, a person signing has a double advantage over a person typing: first of all, they are not inhibited by the popular keyboard, and secondly, they sign faster than the average person types.

While extensive and highly specific research has been done in the field of computer vision (as shown by the sheer number of books available on the subject), little has been devoted to the recognition of sign language, and only one study[7] has considered using a webcam-computer setup (most other modern research explores using a specialized glove to transmit data back to the computer). This is for a combination of reasons: first of all, processor power was formerly far too expensive and not powerful to process a multitude of images (with a high enough resolution to distinguish sophisticated shapes such as the human hand) in anything close to real time. Additionally, the keyboard has been—and remains—an effective, flexible, cheap, and easily extensible tool for computer input. Finally there is as of yet little demand for such a novel mechanism of input.

2.1 Methods & Concepts

2.1.1 Edge detection

"Edge detection" is the process of highlighting differences in pixel intensity and pixel color over an image. It follows that an "edge" in this context is a small area of an image where either pixel intensity or pixel color is changing rapidly. At the very beginning of the research associated with this project, I analyzed various methods of doing this, listed below.

Horizontal differencing This method, along with vertical differencing, is the fastest and the least computationally intensive. However, the speed comes at a cost: this method is highly inaccurate and only manages to find edges with any degree of accuracy when these edges are near to vertical, or, rather, when (part of) the image is changing rapidly from left to right. The equation for this method is

$$V = |P_t - P_b| \tag{1}$$

where V is the value of the computed pixel in a new image which contains the outline of the image being subjected to edge detection, P_t is the value of the pixel above the current pixel, and P_b is the value of the pixel below the current pixel. This equation is applied to every pixel in the image, not including the top and bottom rows. If the differences are in a vertical direction, then this method will not recognize them. I quickly discarded this method of edge detection because a very important part of the edges formed by the outline of a hand would be missed by this method.

Vertical differencing This method of edge detection is nearly identical to horizontal differencing, explained above, except that this method registers changes in a vertical perspective. In the same way that horizontal difference's great weakness is missing any changes which occur vertically, this method does not record any edges which occur horizontally. And just as horizontal differencing is inadequate for the purposes of detecting the edges in a hand, vertical differencing misses crucial horizontal differences. The equations are also very similar:

$$V = |P_l - P_r| \tag{2}$$

where V is again the value of the computed pixel in a new image which contains the outline of the image being subjected to edge detection, P_r is the value of the pixel to the right of the current pixel, and P_l is the value of the pixel to the left of the current pixel. I discarded this relatively quickly as well for the same reasons as I discarded horizontal differencing.

Robert's Cross Although strikingly simple, the Robert's Cross method delivers good results for edge detection. Although not superb, because it misses the finest details, it finds all necessary lines. Plus, it minimizes the amount of noise from the joints of fingers and the joints between the fingers and the palm. The equation for Robert's Cross, applied to every pixel, is:

$$R = \sqrt{(P_t - P_b)^2 + (P_r - P_l)^2} \tag{3}$$

Where V is the value and P_t, P_b, P_l, and P_r are the pixels above, below, to the left, and to the right, respectively, of the pixel on which the edge detection is being performed. I did not immediately discard this method as it performed reasonably well. My final decision was for this method because it is the best balance between detecting too little and too much, and also was not overly intensive on the processor.

Sobel's Operator This method performed even better than Robert's Cross in terms of finding the edges in an image. While Robert's Cross might find faint traces of small, non-distinct edges, Sobel's operator would highlight those strongly and find edge where the eye couldn't have found a difference. This precision was made possible by the following operations, assuming the pattern

$$\begin{matrix} a & b & c \\ d & e & f \\ g & h & i \end{matrix}$$

around pixel e. The general idea of the equations is that they account for all eight of the neighbors of each pixel, while Robert's Cross accounts for only four, and horizontal and vertical differencing two.

$$V = \sqrt{(c + 2f + i - a - 2d - g)^2 + (a + 2b + c - g - 2h - i)^2} \tag{4}$$

where V is the final value of the pixel. I eventually chose against this method, and elected to continue with Robert's Cross, for two reasons. First of all, this method is more computationally intensive, and requires more memory accesses. While over a single pixel the difference is negligible, the difference over 307,200 pixels (the number of pixels in a 640x480 image) is far greater. Secondly, this method highlights *too much* detail, bringing out parts of the hand that I would prefer not to be visible in an image of edges.

2.1.2 XML Parsing

In modern computing, one of the most popular forms of data storage outside of databases is in XML files. This is due to the regular and highly-structured nature of XML, which allows parsers to easily sift through the file(s) and extract data in very little time. For this project, I have chosen to use XML to store data because it is the perfect balance between ease of entry of data and ease of reading for the computer. My program includes a very simple built-in XML parser: it would be unwise to simply rely upon an XML parsing library, although these are effectively standard on all machines, numerous difference libraries exist and each machine may have a different library or version of the same library.

2.1.3 Computer input

Despite the very vital nature of this research, I have not yet dug into code or documentation regarding inserting a new device into the input stream. If I seem likely to succeed, however, I will look into this to make my application even more impressive.

2.2 Prior Research in this Field

As very little substantial work (outside of Kraiss and Zieren's research[7]) has been done in this field, I am pretty much a pioneer and I must decide what path to follow on my own with little outside guidance from previous products and plans. This adds a new element of interest for me: success means that my program is one of the first of its kind in the world. Although others have performed studies and have even programmed sign language recognition systems, they have all used some form of aid to recgonize the hand: they have used a mechanical glove, or a colored glove, or a very specialized background. I intend to write this with *no* such requirements.

3 Development

3.1 Requirements and Limitations

A part of this project is to provide a relatively portable interface for human-computer interaction with a webcam. Therefore, the requirements of this program are rather basic. All that is necessary is a webcam—the basis of the application—and the associated drivers, a computer with Linux installed, and finally Video 4 Linux. To compile from source, a C compiler is also necessary.

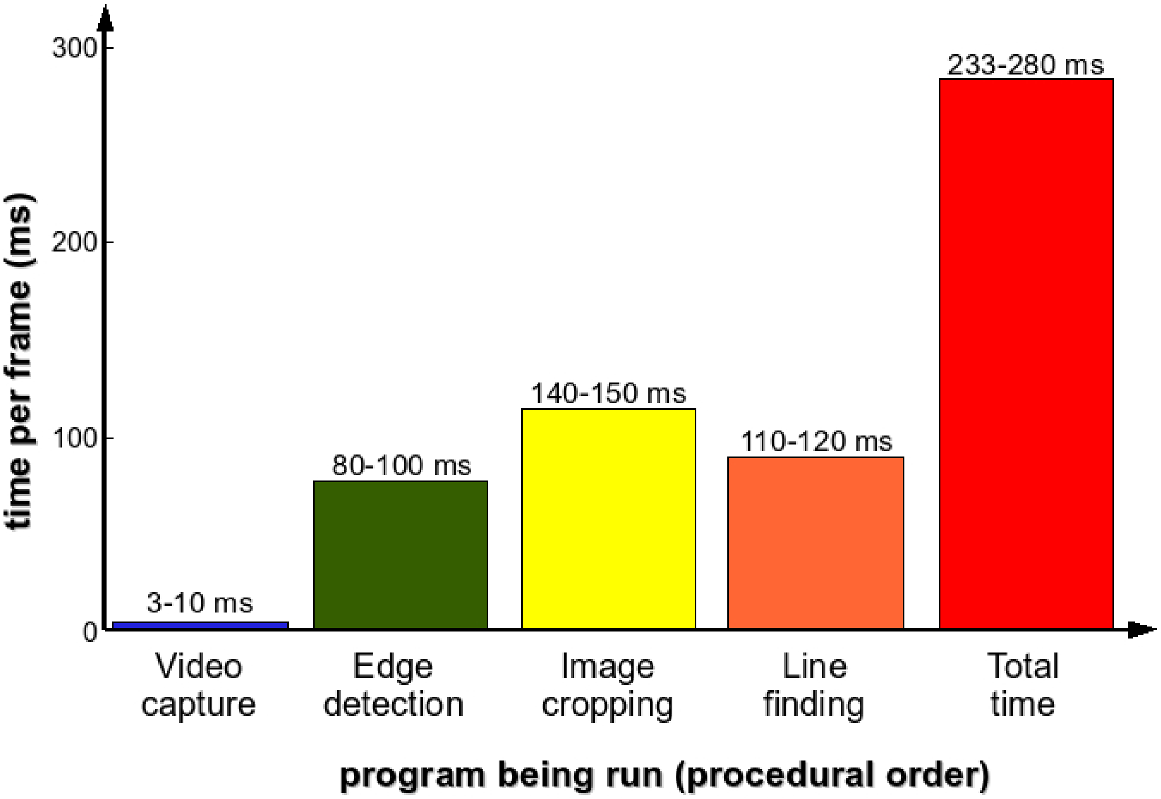
In terms of sign language recognition, the boundaries of this program will exist in terms of letters "understood" and accuracy. To simplify matters, the first program will only deal with alphanumeric characters, to provide a large enough distinction between letters to minimize some of the factors that might otherwise impede position recognition. In addition, the program will also have a limited quantity of time in which to analyze each frame, ranging from $\frac{1}{4}$ to $\frac{1}{2}$ of a second. The goal of acting in real time precludes any deep analysis of each image, and so therefore the program will inherently be somewhat inaccurate (although this may not be as much of a disadvantage as it seems; people make many mistakes at their keyboards as well).

3.2 Plan for Development

Originally, my plan was to program in the order of most testable programs first: first edge detection, then line-finding, then a line-interpreting AI, and finally an image-capture program to “grab” frames from a webcam. I soon found out, however, that this was impractical because the only good way to test a line-finding AI is to use a variety of images representing a variety of letters (otherwise I might just end up fine-tuning to a specific letter or image thereof and breaking compatibility with other letters). The best way to generate a multitude of realistic images is to capture them from a webcam; therefore my plan changed. Instead, I am working on perfecting a capture mechanism first, and then line finding and interpretation (I have already completed the edge detection phase). In brief, my plan is to program in the order of execution in the planned final product.

3.3 Testing and Analysis

The plan for testing my program(s) is rather straightforward: I will use a Python script to run each program several times and report the results and timing. Afterwards, I will inspect the image results from each portion (except the line-interpreting AI) to ensure that it is correct. In each circumstance, I will test ordinary conditions/images, boundary conditions/images, and images or conditions which should be discarded. For example, some very basic testing of four algorithms yielded these times:



This graph shows that the total processing time per frame is currently around 280ms, a little bit higher than the ceiling of 250ms per frame so that I can interpret sign language in real time at an acceptable pace. However, this calculation does no include line interpretation, so this figure of 280ms is subject to increase.

References

[1] Brown, C. M. (1988). Human-computer interface design guidelines. Norwood, NJ: Ablex Publishing

[2] Omoigui, N., He, L., Gupta A., Grudin, J. and Sanocki, E. (1999). "Time-compression: Systems concerns, usage, and benefits." *Chicago 1999 Conference Proceedings*, 136-143.

[3] Karat, C. M., et. al. "Patterns of entry and correction in large vocabulary continuous speech recognition systems." *Chicago 1999 Conference Proceedings*, 568-575.

[4] Foregger, Thomas. "Hough Transform." 06 Aug 2006. PlanetMath. 28 Sept 2007. <http://planetmath.org/encyclopedia/HoughTransform.html>

[5] <http://paginas.terra.com.br/informatica/gleicon/video4linux/videodog.html>

[6] "Typewriter." *Wikipedia*, <http://en.wikipedia.org/wiki/Typewriter>

[7] Zieren, Jörg and Kraiss, Karl-Freidrich. "Robust Person-Independent Visual Sign Language Recognition." <http://www.springerlink.com/content/u5bhmbkldruml981/>