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

Byron Hood

June 11, 2008

Abstract

Sign language recognition can be considered the first step in a long road towards natural language processing, the ability for a computer to "understand" language as humans express it from day to day. Such an innovation would drastically reduce the time consumed by a user in communicating to his or her computer. Research, in fact, suggests that the average user would communicate a message six times faster. This project explores how image processing techniques such as edge detection, line detection, and line interpretation can be applied to identify a particular variant of natural language—sign language—as it is performed. In addition, to make the results more publicly usable, the program will use input frames from a web camera ("webcam"), technology that is nearly universally available. When complete, it is expected that the program will identify the sign language hand positions and gestures for alphanumeric characters in real time, while maintaining a high level of accuracy.

1 Introduction

1.1 Purpose

The resulting program is intended to provide an interface for people who use sign language for everyday communication (referred to as "sign speakers"). It will read images from a web camera, in which a sign speaker is using his or her native form of communication. As explained in the Background section below, this will increase their input speed dramatically. In addition, computer interpretation of sign language is an important first step in the direction of fulfilling an age-old vision of interaction: natural language processing.

1.2 Scope

This project required some research into sign language hand positions and gestures, and also more specific and deeper research 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 would have required 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), had the project gotten that far. Further, considering that data concerning hand positions is 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 also needed 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 may opt to communicate using sign language, a language and an associated 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, a line detection method had to be chosen; various methods of doing line detection are analyzed 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, this project pioneers sign language recognition using webcams without gloves; there exists little outside guidance to help decide what path to follow. This adds a new element of interest: success means that this project 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 recognize 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 (preferably GCC) 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

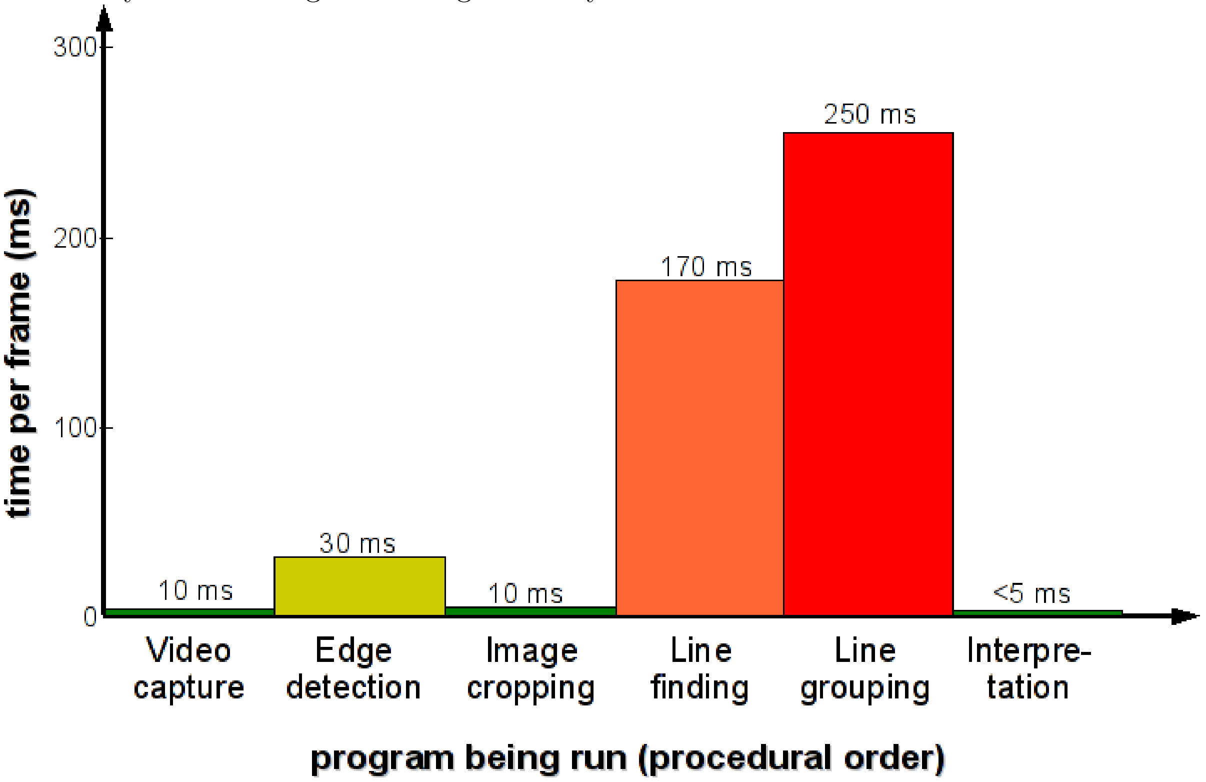
Originally, my plan was to program in the order of most testable programs first: first edge detection, then line-finding, then a line-interpreting AI, and finally an image-capture program to "grab" frames from a webcam. I soon found out, however, that this was impractical because the only good way to 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, the work plan has been more focused on finishing something within the timeframe of this year. As camera interaction might be the least important part of this program, considering that it would not impair the operation of the rest of the application, I have stopped working on that in favor of such subprojects as a mutable list structure, an image IO library, and line finding/interpretation. In this manner, I have completed nearly all of the project except of the camera interaction.

In fact, device control is often considered to be one of the most time-consuming and tedious tasks to program. It requires precise, almost perfect control code, or the results will not work. This, in turn requires many hours of patient debugging and testing to iron out bugs and issues, figure out why the code does not work, and so on. The rest of the application, by contrast, is far easier to code and therefore must faster to finish.

If the application with the exception of hand-camera interaction is finished, there will not really be an issue with functionality, because the basic function of the program is present and other options exist to make the webcam portion work properly.

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 460ms, 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 460ms is subject to increase.

The testing environment for this program has been on two machines. The first is a standard Gentoo Linux x86 installation with the GNU C Library (glibc). The second is another standard Gentoo Linux installation, but for x86_64 (64-bit processor), again with the GNU C Libary installed. Both computers used GCC $\geq$4.2.3 to compile the source code.

4 Results

As the project has not yet been pursued to completion, no complete set of test data and test results is available at the time of writing. Intermediate results, however, obtained by testing different portions of the program independently, are available. The results for each part of the program as described in the architecture section are discussed, whether or not that particular piece was completely implemented and tested.

4.1 Image capture

Capturing images from a standard web camera is one of the more tricky aspects of this project. In addition, it is the part which relies on the greatest number of non-standard features, principally a web camera abstraction layer called "Video 4 Linux." This layer, however, is only available on Linux systems, as the name suggests, and even then, it is not compiled into the Linux kernel by default. The unfortunate necessity for this compatibility layer is due to the fact that it would otherwise be necessary to code some level of support for every video camera model available into the main application.

The program to capture images from a generic webcam (thanks to Video for Linux) is not complete, however, despite the boost from an additional layer of abstraction. Although the `ioctl()` (device control function) calls return normally, no valid image data is actually captured, and whatever initial values were set prior to the capture are recorded in the image.

Despite the failure of the program to adequately capture images, certain timing data are nonetheless available and should be mostly accurate. Considering that all calls return successfully, including the capture `ioctl()`, the timing data collected ought to be within 20% of the timing data, were image capture functional at this point. The overall trend over several testing runs indicates that setting up the webcam device and properly configuring both the program and the device requires approximately 1.5 seconds, but after this, each frame is captured in 0.005 – 0.010 seconds. Since the goal to establish detection in real-time requires at least 4 frames per second to be analyzed, this fits well within the time budget.

4.2 Edge detection

Counting from September 2007, edge detection is the oldest and most-often revised part of the application. At the moment it is in its fifth major rewrite, each time being optimized or made to fit better within the architecture of the main program. The first time edge detect was written, it used the Robert's Cross algorithm described above, and chewed up nearly 0.3 seconds at runtime. Even excluding the time required to read and write 1.6MB of image data to and from disk, the inefficient first algorithm needed almost 0.08 seconds to detect the edges in an 800x703 image.

After subsequent modifications and modifications, the current version employs an algorithm independent methods in other words, the edge detection algorithm in use is not hard-coded into the looping mechanisms to iterate over the pixels and detect edges in an arbitrary image. Moreover, the newest version uses only 0.02-0.03 seconds, fitting much better into the strict time restriction imposed by four frames per second.

4.3 Image cropping

The concept of cropping images to emphasize important features while removing less important ones is not new. It does, however, present a new, simple method of optimizing image processing. Due to the fact that image processing often requires multiple iterations over thousands or even millions of pixels of data, cutting out even a small number of pixels can induce a large speed up in a larger application. Thus, this application uses a very simple cropping algorithm which removes featureless rows and columns from the image.

This piece of the overall application has also been revised multiple times. Originally non-functional in C, it was transferred to Python, using the Psyco optimization library. But after being rewritten in C, it is highly functional and very quick. Typically, this reduces the image size by about 25% in both vertical and horizontal directions, for and overall reduction in area of nearly 50%. The optimization obtained is on the order of millions of floating-point operations (FLOPS). Further, it is well worth the cost—considering a well-optimized program might push through several million FLOPS per second—because those millions of additional operations may end up costing an addition half or even a whole second per iteration.

4.4 Line finding

While the edge detection part of the program may be the oldest and most revised, this part has seen the most little adjustments. Since its inception in the second quarter, it has only been overhauled twice, but the intricate details of line recognition have been changed hundreds upon hundreds of times to obtain the maximal optimization and accuracy possible. Currently, the line detection program detects approximately 1500 lines in a 650x630 image (this is also dependent on the content of the image after edge detection). It does this in approximately 0.170 seconds; while this is above what is expected for real-time, it is about as fast as possible while staying within the constraints of line finding. Perhaps this could be accelerated by increasing the granularity—the number of radians between the angles at which lines are searched for—but doing so would have a very negative impact on the accuracy and completeness of the detection mechanism.

4.5 Line grouping (chaining)

By far the most time-consuming part of the operation of sign language recognition is grouping lines together. Unlike other parts of the application, which start out in polynomial time and which can often be reduced from, say, $O(n^3)$ to $O(n^2 \times log(n)$ or even better, line chaining is necessary $O(n!)$. Once a line is added to a group, all previous lines must be re-compared with the group because they may have become eligible based on the group's new slope and x- and y-positions. Due to the unavoidable factorial nature of line grouping, it requires nearly 0.250 seconds—the entire time allotment for each iteration—to run on its own.

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/>