Senior Project

John Leung

Fall 1997

The Book Reading Machine

I. PROJECT OVERVIEW:

The purpose of this senior project was to provide a low cost reading device for people with reading or visual disabilities. Such devices use an image capturing system to capture the images of the text and convert them into image file types that an optical character recognition (OCR) system can recognize. The OCR system then produces a text file and sends it to a speech synthesizer to translate it into spoken words. Such devices already exist in the market, but all of the ones I have found use flatbed scanners as the image-capturing device. The use of flatbed scanners as the image-capturing device has two main drawbacks. The first is that flatbed scanners must be use indoors, making the book reader just a book reader, without any other functions. If they were to use handheld scanners or even digital cameras on the other hand, such a device will not be limited to just reading books; it could be used outdoors as well. The use of handheld scanners is still limited in their usage -- users must be able to "touch" the source the text. For example, it would be very hard or even impossible for the users to read street signs. The use of digital cameras is much more versatile. The visually impaired can now take this book reader with them to anyway, speaking out text such as street signs, flyers, or even menu items in restaurants. This means that the visually impaired and blind persons can now access the vast and rich textual information that they have been kept out from.

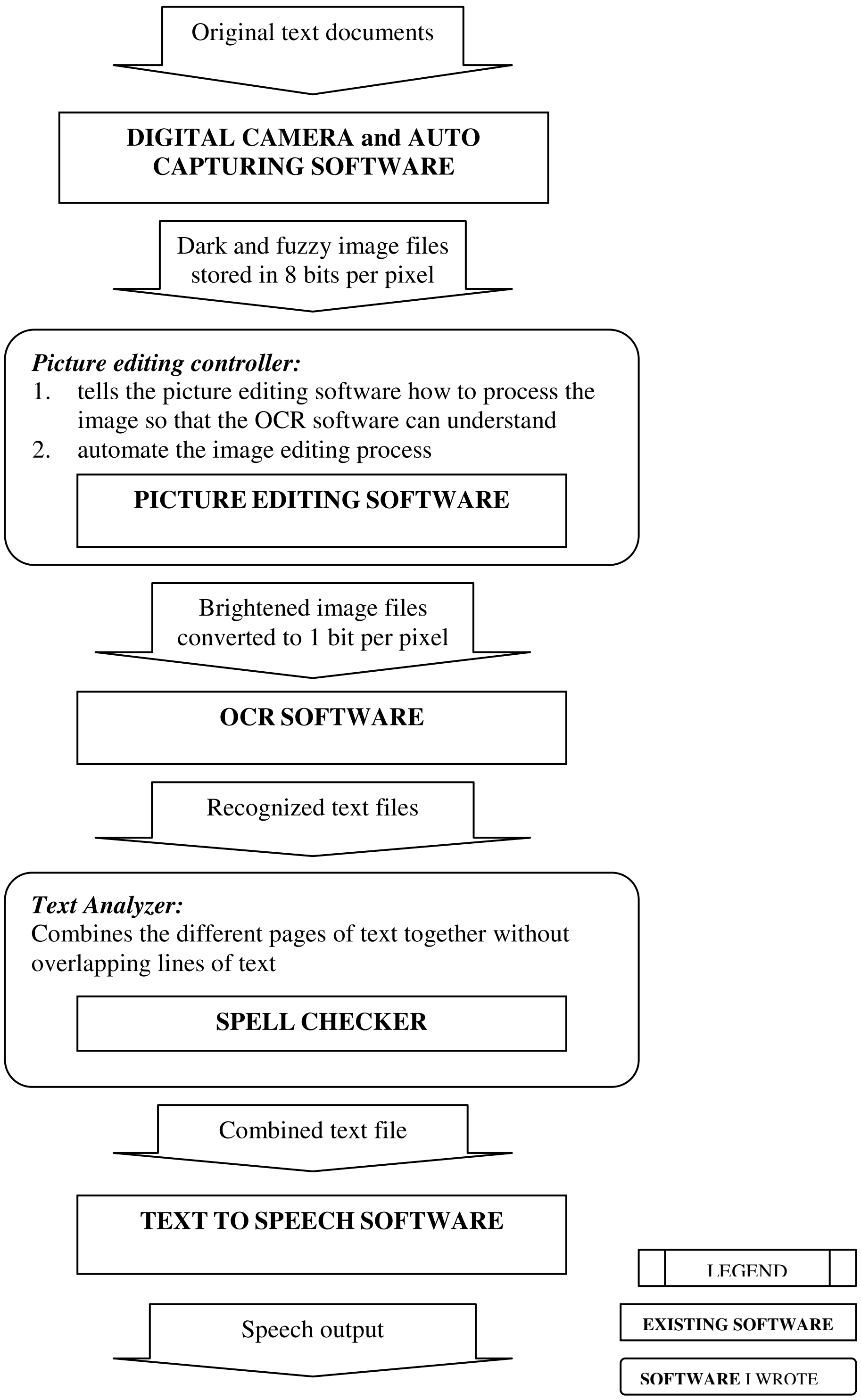
The second drawback for the use of flatbed scanners is that having the visually impaired person use a scanner is too difficult. The whole process includes the following tasks: feed the book onto the scanner, close the cover, scan, wait, open the cover, flip the page, lay it back down again, and continue this cycle all over again for every page. What I want to do is to design and build a system that can use a digital camera instead of a flatbed scanner. Thus, all the user has to do is turn the pages just like a normal person would, and the digital camera will auto-capture the images without much intervention from the user.

However, as it turns out, my book reader requires more user intervention then the scanner type. The main reason is that since most low cost digital cameras can only capture a small picture area when focusing on something as fine as text, different portions of the text must be captured and then combined together to form the original text. In doing so, either the user must manually move the different portions of the text to the image-capturing area, or there must be some form of mechanical movement device that will receive input from the program and drive the camera to the next position to take the next picture. Such a mechanical device will not be too expensive, but controlling it may be a difficult task. If, however, I found out that this is not the case, I will try to include the mechanical device to make this book reading machine truly usable even by the blind. But for now, this book reader is just targeted towards the reading disabled since it requires the user to manually move the page of text. But even with or without the mechanical device, since the output of the digital camera is no where near the quality of a scanner, lines of text must be captured at least twice in order to have a choice to choose the better recognized one. This is especially true for text taken near the top and bottom of the image area because for most low cost digital cameras, the output of those areas are not as clear as the center of the image. Therefore, there needs to be a good way to combine the different text images into one continuous page – meaning that the text output must not have repeated lines from areas that have been overlapped in the images. This is the main focus of this project since the method to “smartly” combine the different text files is necessary as long as low cost digital cameras do not have the image capturing size and quality of flatbed scanners.

II. THE PROCESS:

There are five main components (or subsystems) that are involved in the book reader.

Below is a graphical view of the system:


Original text documents
DIGITAL CAMERA and AUTO CAPTURING SOFTWARE
Dark and fuzzy image files stored in 8 bits per pixel
Picture editing controller:
1. tells the picture editing software how to process the image so that the OCR software can understand
2. automate the image editing process
PICTURE EDITING SOFTWARE
Brightened image files converted to 1 bit per pixel
OCR SOFTWARE
Recognized text files
Text Analyzer:
Combines the different pages of text together without overlapping lines of text
SPELL CHECKER
Combined text file
TEXT TO SPEECH SOFTWARE
Speech output
LEGEND
EXISTING SOFTWARE
SOFTWARE I WROTE

Because most of the subsystems are already built for the IBM-PC, the overall system is designed to run on the IBM-PC. More specifically, it is best to be run on 75Mhz Pentium or above since performance of both the image processing and optical character recognition are boosted up substantially on Pentium machines. Users can make this a portable device by using Pentium laptop computers. In fact, I built the entire project on an AMD 586 - 133Mhz (Pentium 80Mhz equivalent) laptop computer running Windows 95. This device can really be considered as a true handheld text reader if used with the Toshiba Libretto 50CT (a 75Mhz Pentium notebook computer that is the size of a VCR tape, currently selling at $1280.00, a price not much different than other laptop computers of similar speeds).

Of all the various subsystems that are involved in this project, I am only concentrating on the control of one of those and the full design and implementation (besides the spell checker component) of another. The reason for not trying to do all or any of the other parts is rather simple. First, companies have spent millions of dollars and years on those programs; it would almost be impossible for me to think of a better way to do them given the amount of time and budget that I have. Also, communication with the other Windows programs has always been a knowledge that I wanted to acquire. Furthermore, I do not know of any programs that have already been written to combine different text files into one “smartly,” that is to join the text files into one by evaluating their similarity and avoiding overlapping of lines.

The two subsystems that I wrote were done using Visual Basic 5.0. It has many great features. First, the language itself is based on Basic, providing me with the ease of use and gave me quick results. In fact, I had never used neither Basic nor Visual Basic before and was able to understand and begin programming simple tasks after learning it for just a few hours. Second, it has easy access to Win32API function calls that I needed in order to communicate with the other Window programs involved. Third, it has a very nice development and debugging environment; it lets me test the codes immediately without waiting for it to compile first every time. This was very critical since I almost needed to retest my code for every WIN32API function call that I added. It also has many functions to work with strings, which is what most of the text analysis part of the reader is about. One thing that it does lack is good bit manipulation functions. For example, it doesn't even have bit shift or rotation function call that almost all other languages have. All of the codes I wrote followed good coding, variable naming, and commenting styles.

III. THE SUBSYSTEMS:

1. The Camera and Its Software Package:

I have chose to use the Connectix Color QuickCam digital camera as the image-capturing source for many reasons. The most important is that it is the most affordable (besides the black & white non-adjustable focus version) digital camera around – just one hundred dollars. Not only does this enable me to spend the extra money for the other components of the project, it also means that if this ever does become an actual product, the consumers will not need to spend as much as well. The second most important reason is that it has an auto-capturing function built into the software; most, if not all, of the other digital cameras do not have this ability. The other benefits include small in physical size (2.5" in diameter), ability to save images in both *.bmp* and *.tif* formats, auto lighting adjustment, and adjustable focus (down to 2").

After the user sets the auto-capturing rate, it will start capturing images periodically at that rate. All the user has to do is to move the page of text to the next position to be recognized. The software automatically stores the images in a filename and location the user sets. The filenames are incremented as well (i.e. 1st file would be page0.tif, the 2nd page1.tif, and so on). Because the lowest resolution of QuickCam is 8 bits per pixel (256 colors of gray), and most OCR programs require the images to be store in 1 bit per pixel (each pixel must be on or off, true black or white), the outputs from QuickCam must be fed into an image editing software in order for the OCR program to be able to recognize.

2. The Picture Editing Software and Controller:

Not only does the picture editing software needs to decrease the resolution of the picture; it also needs to brighten it as well. The outputs from QuickCam can auto adjust the lighting of the image, but in doing so, it sets the overall picture into mostly grayish dots (values between 0 and 255). When the picture editor decrease the resolution of the image, it calculates that if that 16-bit value of individual pixels should decease to a single bit of 1 or 0, or black or white. Because most of the dots will turn into black, the image must first be brightened, before decreasing the color depth.

Paint Shop Pro 4.0 is one of the few picture editing software that can do both of the tasks with ease. However, it does not have a batch conversion function built-in to do so; the only batch conversion it has is to convert between one file format to another (i.e. from *.bmp* to *.jpg*). Thus, either I need to have direct access to the function calls that Paint Shop Pro uses so that I can call it from within my program, or I would need to write a batch conversion program that tells Paint Shop Pro what to do. Otherwise, it would be too tedious a task for the user to do all the conversion for every image. The former solution is the more elegant and robust; however, the company that developed Paint Shop Pro will most likely charge me a substantial amount of money in order to obtain the information. Thus, I was left with the latter solution.

Obtaining easy access to the control of other Windows programs requires the software to be originally built with OLE or DDE. Paint Shop Pro does not; thus, the only other solution is to simulate mouse and keyboard events in order to trick Paint Shop Pro to do the conversions. Even though less than half of the code for this project is for doing this, it took more than half of the time to learn how. Sending keystrokes to the program is quite easy; all that needs to be done is to call the function *AppActivate("Title bar caption of program to be activated")* to the set the input focus to that application, and call *Sendkeys("keys to be sent")* to simulate user keyboard inputs. Both of these functions are built-in in Visual Basic. However, after sending in the keystrokes, the control program must wait until the application finishes processing the image before sending in the next input keys. This is the trickiest part of all. If I were to set a preset amount of time for it to wait before sending in the next keystrokes, either that time period needs to be very long, or it will not be guaranteed to work every time because application performance depends greatly on the current CPU usage by other running applications. Even if no other applications are running, it will still fail if running on other computer with lower CPU speeds or less memory. To correct this problem, the controller must wait for specific messages that are passed between the application and Windows when a task is done in order to know exactly when to send the next keystrokes.

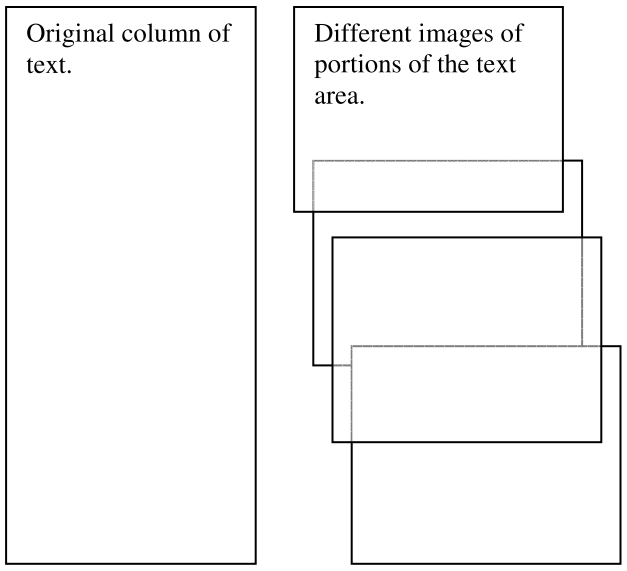
In order to know the messages that are being pass, I need to use the WIN32API function calls. WIN32API (Windows 32 bit Application Programming Interface) is the native language that Windows uses to communicate with the different applications. Visual Basic provides easy access to these function calls. All that needs to be done before using them is to write a declare statement that tells Visual Basic what kind of parameter it requires and the location that the function is store in. The use of WIN32API to get and send messages also requires a unique ID (called handle) that's dynamically assigned to the running applications every time they are started. After obtaining this handle, the controller can then know when the application has finished with its current task and can then send in the next.

3. The OCR Software:

The outputs from the image editing software are now ready to be read by the OCR (Optical Character Recognition) program. I have chose to use the Cuneiform 3.1 since it is the fastest if not also the most accurate of all the OCR programs out in the market. It can recognize a page of text image in less than seven seconds running on my AMD 586 – 133Mhz laptop computer, which is at least twice as fast as the other OCR's. It also has batch recognition built in so that I don't even have to write a control unit for it, saving me much valuable time I needed to work on the other components. The *tiger.dll* is their recognition engine. Information about their recognition techniques and a trial version of their Cuneiform OCR can be obtained at *http://www.ocr.com.*

4. The Text Analyzer:

After the OCR program finished recognizing the text image files, it outputs the recognized text into ASCII text files. It is now ready to be analyzed and combined into one coherent text file. The image capturing area of a regular document (font sizes 8 to 14) is about 3" wide and 2.5" high. Since recognizing any document that is larger than that size requires comparing both horizontal and vertical overlapping areas, it is best to solve one problem first, before attempting the other. Thus, I will only be concentrating in recognizing newspaper and magazine columns for now, eliminating the need to compare vertical overlapping areas, leaving that for future upgrades. Below is a graphical representation of the process of capturing and evaluating a column of text:



Since the program do not know ahead of time how many lines the user will overlap, it can not just find and replace a constant number line between the two files. Exact match of those lines also can not be used because the output of the camera for the overlapping areas will not be identical in both images. The first solution that I thought of is to count the individual characters that are used in the overlapping lines, then store that count for each character in a variable. For example, “this is a test,” will give: “a”Count = 1, “h”Count = 1, “s”Count = 3, and so on. Since the least memory space definable for a variable in Visual Basic 5.0 is a 16-bit integer, memory space need to hold the count of each character within each line (average of 15 –20 different characters) will be enormous. If, on the other hand, I were to make a 32-bit variable and make each bit represent a specific character, it will not be a good indicator of how similar the sentences are because most them will have the same twenty or so bits on. Also, for either solution, the position (or order) in which the characters appear in becomes insignificant, which should be at least somewhat important in similarity compares. Thus, I knew that I have to find better solutions.

Luckily, I was able to find a better solution, called the Harrison’s signature method. The Harrison’s signature method compares the similarity of strings by computing a “signature” for each string and compares the differences in the signature. The signature is computed as follows: (i.e. “a small line”)

i) Pair up all adjacent characters. -- i.e. a_ _s sm ma al ll l_ _l li in ne

ii) Assign each character any value. – i.e. a = 1, b = 2 … , z = 26, _ = 27

iii) Hash each pair of the string into a value between 1 and B, where B is the number of bits to be used for the signature. i.e. the hashing function can be: (((1st char)*(no. of different chars) + (2nd char)) mod B), where no. of different chars = 27 (ignore cases and count space as a char), and using 16 bits (B = 16) to store the signature gives:

$s = 19$, $m = 13$, $sm = ((19 * 27 + 13) \bmod 16) = 14$

Thus, the 14th bit of the signature will be set (along with all other hashed pairs), giving the bit pattern for "a small line" 1100011110001111.

(0th bit starts on the left)

The Harrison's signature method, however, does not cover that case that a pair of characters may appear twice, but that does not really matter for our purposes. The most critical problem is that if the length of the string is much longer than the number of bits used in the signature, then the result of this method becomes unreliable because different pairs of characters can easily set the same bit on. Since newspaper and magazine columns normally contain around thirty characters, using a 32-bit variable for the signature will decrease the chances of that happening.

The steps involved in applying Harrison's signature method are as follows. First, the mapping of characters to some form of numbering systems is done already; all I have to do is use the ASCII value of the character. The second step is to apply the formula to every line of text in both files. This is necessary because the program will not know ahead of time how many lines will the user recapture in the second image. Every line's signature will now be compared with all the signatures in the other file. The method for comparing the signatures will just be an XOR operation. The result will be the similarity between the two lines (zero being the most similar). Since it is possible for two lines to be slightly different yet able to yield an identical signature, zero must be reserved for identical match of the original string (Visual Basic has an exact string match function). Thus, we need to add a one to all similarity results of the signatures. After the similarity test, the program will know which lines of text are the most similar. For newspaper and magazine columns, similarity of four or below means that those two lines are suppose to be the same. Where as similarity of seven or above means that the two lines are most likely to be different. For similarity of five and six, the program will have to compare the adjacent lines to see if those lines are also similar to adjacent lines in the other file. If not, they are also likely to be different as well.

After the program had found out which lines are similar, it is time to delete one of them. The method for doing so is quite simple, spell checks and other criteria checks can be run on those lines; the worst line of the two will be deleted. The other criteria's include the check for three or more consecutive letters and the existence of too many special characters (such as "~" that the OCR program outputs when it can not recognize a character). More detailed methods, techniques, and special cases used for both the signature and line comparisons can be found in the program code included in the back of this report.

5. The Text To Speech Program:

The single file of combined text is now ready to be spoken. Many of the other book readers on the market use either the hardware version (PC card) or the software version of the DECTalk from Digital Equipment Corporation. For this project, I am using the software demo version. It accepts ASCII text files and phonetically synthesizes them into speech through the PC speakers. What this means is that the text does not have to be legal English words. For example, the program will try to pronouns "dsgetjk" with its knowledge of the English pronunciation rules, exceptions, and the context in which the word appears. This is especially useful since the program will not have to keep a huge dictionary of all the English words and all the different ways of saying the same word. Also, it will not be outdated by our ever-changing English language. Finally, it will be able to pronounce any foreign names such as "Dr. Ryujin" fairly accurately.

IV. ROADBLOCKS:

The amount of time I have spent in this project is at least two to three times what I expected. There were many reasons. First, since I wanted this system to be truly portable and easy to use by the visually impaired, I want the design to be mostly hardware. This would have been the ideal way since everything could be packed into one small stand-alone product. Whereas if I were to do it in software, the smallest it can get would be the size of a laptop that the user uses. However, after long searches for hardware solutions, I had to give it the idea since the components were either not available in hardware form, or cost a bundle.

Second, not only do I have to search all over the web for trial and demo versions of each component, I also had to try each of them out to see which one was best. Even after picking out the better component, optimizing them also required quite some time. For example, it took me many hours just to find out which one of the dozens of picture adjustments in Paint Shop Pro worked best for the images.

Third came the lighting problems. If the light coming from one side of the room was brighter than the other side, the image will not appear as well. At first I thought putting a few tiny light bulbs around the camera will do the job, but then there were dark and bright spots. Adding a few more bulbs gave smaller but more dark and bright spots. Then I switched to use tiny fluorescent light tubes, thinking that the lights would be more evenly distributed. It did not turn out that way; their brightness is concentrated in the center, and they were much bulkier than the bulbs. I then went back to light bulbs, solving the problem by putting a white filter (made out of those 2-gallon water container plastic) to fade the light concentration.

Next was the camera. I used the Connectix Black & White QuickCam at first because it was inexpensive (only $50.00), and thinking that I did not need color. I later found out that other factors were very important as well. For example, the black & white version can only capture an image size of 320 by 240 pixels, and the focus is non-adjustable. After trying to make it work right for a few days, I had to give up and got the color version that has an image size of 640 by 480 pixels and adjustable focus.

Finally, it was the programming. First, I had to decide which language to use. Thinking that I could also benefit from learning object-oriented design at the same time, I tried out Visual C++ and Java for a few weeks and decided that I had to let that idea go. Java was not the right tool, and Visual C++ required too much learning time. I finally gave Visual Basic a try, and was glad I did! Next is trying to figure out how WIN32API works. After many hours of frustration, I finally got the hang of it. Nevertheless, it was still hard; things don't always seem to work out, as they should. Finally, trying to think of a clever way to combine text was the most difficult. I had to go look for a real solution. Even after I found the solution, applying it in my application was not an easy task.

V. CONCLUSION:

Even after such hard work and time spent, I was very glad to have done this project. I gained much knowledge from the entire experience. However, this project could be viewed both as a success and a failure. I was able to carry out and finish what I have set out to do -- build a book reading machine with the use of a digital camera. However, one of the main purpose of the project still have not been met – providing an easy means for the visually impaired person to get text information. The book reading machine as-is is much harder to use than the scanner counter type. In fact, it is impossible for a blind person to operate it. However, I do not think that adding the mechanical drive to it will be such a difficult task, and adding it will certainly make it much easier to use.

The other main problem with this book reader was that because of the low quality output from the QuickCam, the text output from the OCR contained many misspelled words. Images from other cameras such as the Kodak PC 500 produced much better results, but it was both an expensive camera and it did not have the auto-capture function. The only solution to this is to wait for a better quality, auto-capture, and inexpensive digital camera to be out on the market.

Other features I will most likely add include, making the whole process fully automatic, adding algorithms to recognize and combine all types of text (instead of just columns), and replacing words within the duplicate lines instead of just taking one line or the other. Furthermore, if I were to build a robot one-day, I will certainly include this book reader in it so that it can truly communicate with humans.

VI. BIBLIOGRAPHY

Peter D. Smith, “An introduction to text processing,” The MIT Press pp.56-57, 1990