PADCAM: A Portable, Human-Centric System for Handwriting Capture

by

Amay Nitin Champaneria

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002
June 2002

Author .....
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by .....
Larry Rudolph
Principle Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

PADCAM: A Portable, Human-Centric System for Handwriting Capture

by

Amay Nitin Champaneria

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we propose and demonstrate a perceptual interface for pen-based input that captures live video of handwriting and recovers the time-ordered sequence of strokes that were written. Our approach uses a novel combination of frame differencing, pen-tracking, and ink-detection to reconstruct the temporal information in the input video sequence. We present the design and implementation of PADCAM, a prototype handwriting capture system, along with preliminary results and useful applications. Specifications for extending PADCAM for use on handheld devices are also included, along with an extensive discussion of future directions for this research.

Thesis Supervisor: Larry Rudolph
Title: Principle Research Scientist

Acknowledgments

First and foremost, I would like to thank my advisor, Larry Rudolph, for providing the motivation, guidance, and resources for this work. His boundless creativity was infectious, or dare I say "pervasive," which made the environment conducive to groundbreaking research. He has done a great job of making this experience challenging yet rewarding.

To those who contributed directly to my work, I extend my gratitude for your help and my respect for your technical abilities. At the top of this list is Mario Munich, who not only pioneered this particular perceptual interface but also provided me with indispensable assistance in writing my system. I would like to thank Nick Matsakis for his robust and well-designed Natural Log system, and Mike Oltmans along with the entire Sketchpad team for their great work.

Many other faculty members at the MIT Laboratory for Computer Science and Artificial Intelligence Laboratory also contributed indirectly. I would like to thank Leonard McMillan for the valuable computer graphics experience I gained while working in his group. My interest and exposure to computer vision is largely due to Trevor Darrell's vision interface seminar. And the advice I received from Randall Davis allowed me to better understand the Sketchpad system and underlying vision.

I would like to recognize Jamey Hicks for his efforts to organize the handhelds.org project to port Linux to the iPAQ. His work, along with that of Andrew Christian, Gita Sukthankar, and Ken Steele, helped me realize the value of the open source community, and I am proud to have been a part of this development effort.

In addition, I would like to thank all of my officemates and project partners. In particular, Todd Amicon, Shalini Agarwal, Josh Jacobs, and Jorge Ortiz have made this experience fun and rewarding. In addition, I'd like to thank Jason Dettbarn, Orlando Leon, Brendan Kao, Leon Liu, Ishan Sachdev, Sonia Garg, Marc Bourget, and Craig Music.

In the context of Project Oxygen, I must acknowledge the Stratech team at Microsoft for exposing me to handheld and pervasive computing environments—Suze Woolf, Roland Fernandez, Ron Day, Iain Hackett, Wis Rinearson, and Craig Mundie.

I would like to thank Billy Corgan, James Iha, D'Arcy Wretzky, Jimmy Chamberlain,

and Jimi Hendrix, as well as Greyhound bus lines, without whom my 30-hour scenic tour of the Southeast United States would have been impossible.

Lastly and most fundamentally, I would especially like to thank my parents, Nitin and Smita Champaneria, and my sisters, Neha and Reshma Champaneria, without whose support I could not have made it this far.

This work is supported in part by our Oxygen industrial partners, Acer, Delta Electronics, Hewlett-Packard, Nippon Telegraph and Telephone, Nokia Research Center, and Philips Research, as well as by DARPA through the Office of Naval Research contract number N66001-99-2-891702.

Contents

Chapter 1

Introduction

Perceptual interfaces that allow natural human-computer interaction are an integral part of pervasive computing initiatives such as MIT's Project Oxygen. We propose and demonstrate a perceptual interface for pen-based input.

1.1 Human-Centric Computing

In part to address the lack of progress in human-computer interfaces, many prominent technology companies and academic institutions are initiating projects that strive to move the technology industry from computer-centric to *human-centric* computing. At MIT, this movement is embodied by Project Oxygen, a joint effort between the Laboratory for Computer Science and the Artificial Intelligence Laboratory at MIT. As Rudolph explains:

> The Oxygen vision is to bring an abundance of computation and communication within easy reach of humans through natural perceptual interfaces of speech and vision so computation blends into peoples' lives enabling them to easily do tasks they want to do—collaborate, access knowledge, automate routine tasks and their environment. In other words, *pervasive, human-centric computing.* [25]

Of interest in the context of this thesis is the goal of natural perceptual interfaces. Specifically, we aim to provide a new interface that is based on an old but indispensable means of recording information—handwriting on paper.

1.2 The "Paper-and-Pencil" Approach

The paper-and-pencil approach to recording information is both powerful and convenient. It accepts virtually all types and formats of notes, bounded only by the fact that the paper is two-dimensional. It provides an easy and natural way to delete notes and correct mistakes (i.e. using the eraser). And it is incredibly reliable—you never have to worry about the "system" crashing and wiping out all of your work.

Despite these strengths, the paper-and-pencil approach leaves much to be desired in the areas of information exchange and replay. After a notetaking session, one often has to type up the notes on a computer to send to others. Scanners and digital cameras help by allowing digital capture of the raw information as images. However, a more desirable solution would provide a meaningful interpretation of the raw information—the actual text that was written or sketch that was drawn rather than an image of it. This interpretation would be more useful and easier to store. It would also be helpful if one could replay the entire notetaking session, perhaps with the sound from the original session.

We aim to develop a perceptual interface for handwriting capture. Because of its basic strengths as an information capture medium, the paper-and-pencil model is at the core of our own approach. We augment this with a system that uses visual input to reconstruct the temporal information associated with the handwriting and uses this information to recognize what was written.

1.3 Outline

This thesis will discuss an interface for handwriting input that combines the ease-of-use of paper-and-pencil with the power of electronic information capture. Section 2 surveys the problem of handwriting capture, defines a useful taxonomy for discourse on the topic, examines some of the interesting challenges of the problem, and finally presents related research. After this thorough inspection of the problem, we dive into our solution, PADCAM.

The design and implementation of PADCAM is presented in detail in Section 3. The key components to this system are a sequence of pre-processing stages and temporal recovery

stages. Included in pre-processing is a novel page detection algorithm and a method for monitoring blocks in the page. We present two main approaches to temporal recovery—one based on frame differencing and the other based on pen tracking. Finally, we present the results of each stage and the system as a whole.

Section 4 provides a specification for extending the prototype system for use on hand-held devices, specifically the Compaq iPAQ. We discuss the challenges we faced in adapting PADCAM for the iPAQ and the specific design decisions we made in our implementation. Next, we submit experimental results and discuss some of the blocking issues that affected the performance. We conclude this section by suggesting ideas to improve our iPAQ implementation.

Next, Section 5 proposes some applications for PADCAM. Finally, we provide suggestions for improving PADCAM's performance, ideas for future work, and conclusions in Section 6.

Chapter 2

Problem Overview

At a high-level, our overall task is simply to capture pen-based input and extract the meaningful information associated with it. Though this seems like a straightforward description of our problem, it is far from complete and unambiguous. In this section, we will drill down on the details of the problem, first defining useful terms, then outlining some considerations that drove our design, and finally surveying the relevant body of research.

2.1 Terminology

In order to minimize ambiguity, it is necessary to specify more clearly the problem we aim to solve with our new system. Along the way, we will also define some of the terms that we will be using in our description of handwriting recognition tasks. First, it is important to note the distinction between *on-line* and *off-line* recognition systems. On-line systems capture handwriting as the user writes on a digitizer tablet or some other input device. Note that in this context the term on-line implies nothing about when or how fast the recognition occurs, only how the input is obtained. In contrast to on-line systems, off-line recognition accepts as input an already-written sample of handwriting, such as a scanned image of a page of handwriting. The difference is that on-line systems can use temporal information to disambiguate the input, while off-line systems have only the final image and are thus faced with a more challenging recognition task. Our handwriting capture system observes the page

as the user writes on it; thus, it is an on-line system.

Now we turn to the actual handwriting input. We will refer to the action of placing the pen tip on the page as a *pen-down* event. Similarly, when the pen tip is picked up from the page, it is a *pen-up* event. Our fundamental unit of handwriting will be the *point*, a time-stamped location on the page. A series of consecutive points occuring between a pen-down and the next pen-up event is called a *stroke*. A sequence of strokes comprise a *handwritten expression*.

2.2 Design Goals

2.2.1 Natural, Human-Centric Setup

Our basic intention is to design a handwriting capture system that provides a natural, human-centric interface for handwriting input. We wanted to allow the user to write with regular handwriting utensils, as opposed to a specialized stylus and surface required by commercial digitizer tablets. To satisfy this goal, we designed our interface for use with notes taken on any surface with practically any regular pen. While our prototype system was tested using a pen on paper, dry eraser markers on whiteboards, and even chalk on the sidewalk would be acceptable forms of input.

Another degree of freedom we provide the user is that of writing surface placement. Other vision-based handwriting interface systems dictate exactly where the writing surface must be located in relation to the camera [28]. Such restrictions require the user to adapt to the interface rather than the other way around, and thus provide only a marginal improvement over commercial digitizer tablets. We avoided such restrictions and instead allow the user to place the writing surface anywhere in the camera's view. This choice is made not only to provide the greatest convenience to the user but also to allow for the possibility of extending the system to support multiple unregistered cameras, which are becoming typical in pervasive computing environments.

2.2.2 Generality of Content

In the initial brainstorming of this project, we considered developing it as an all-in-one solution to the problem of mathematical expression recognition from notes taken on paper. Doing so would have tied our interface to a particular type of content—mathematical expressions—and in a sense would have violated the end-to-end design principle of system design [26], which states that extra features should be layered on top of core functionality rather than included therein. Thus, we decided against a limited all-in-one solution in favor of a general handwriting interface that accepts any type of content—printed text, cursive text, mathematical equations, and even drawings—and passes the task of recognition or meaningful interpretation to a higher-level application. We will see in Section 5 that this choice allows us to use our system as an input adaptor to existing recognition applications with little or no change to the application software.

2.3 Related Work

A survey of the existing body of work in this field will prove useful both for background and to provide a baseline for comparison for our new work. We begin by describing on-line systems that use video to recover the pen-based input, focusing primarily on the system on which PADCAM is based. After that, we present off-line approaches and some commercial products, highlighting the key ideas learned in each case.

2.3.1 Video-Based Recovery

The pioneering work in capturing pen-based input using computer vision comes from Mario Munich of Caltech. In his 1996 paper in ICPR, he introduced the problem of recovering pen-based input from video sequences and proposed a solution [22, 24]. His system consisted of a correlation-based pen-tip tracker, a recursive estimator for prediction of pen movement, and a classifier for pen-up and pen-down events. We incorporated Munich's tracker and recursive estimator into PADCAM and compared its performance to our own versions of these components.

Since his initial work, Munich has improved his system, adding a component to perform subpixel interpolation on the video images and thus achieving much higher spatial resolution [23, 21]. This modification made the interface sufficiently accurate for signature verification, which was the application that Munich had intended for it. Also, the speed of his system was improved, and enabling it to process frames in real-time at 60Hz. We aim to produce similar results and adapt the system for use on handheld devices.

We also gathered ideas from other similar systems. In designing their data tablet system for recognition of Japanese characters, Yamasaki and Hattori used a sequential differencing algorithm on which our masked difference approach is based [28]. Their system serves as a proof-of-concept, as it can only handle very slow input handwriting speed (0.8 cm/sec to 1.3 cm/sec) and it imposed strict requirements on the physical setup of the writing environment. As mentioned earlier, we plan to avoid such requirements in favor of a more natural, human-centric writing interface.

The idea of using differences between frames was extended by Bunke et al in their system for acquisition of cursive handwriting [4]. They outlined two challenges in the differential image approach: classification of differences to ignore pen and hand movements and occlusion of existing ink traces. To deal with these issues, several interesting ideas are proposed, including thresholding, dilation, and the use of an *aggregate image*. As described in Section 3.2.1, we build upon the idea of the aggregate image in our backward differencing algorithm.

2.3.2 Off-Line Recovery

Though our goal is to design an on-line video-based system, a survey of the recent off-line approaches may provide useful ideas that can be applied to our on-line task. After all, the on-line recovery task is essentially the off-line task with the benefit of temporal information, so any solution that can be applied to the latter should be applicable to the former.

The most comprehensive analysis of the off-line recovery task to date is presented by Doermann and Rosenfeld [12, 10, 11]. In formulating a system for document image understanding, they provide a complete taxonomy for classifying strokes and propose a platform for temporal recovery. While most other off-line systems rely solely upon continuation of the

ink trace for temporal recovery, Doermann and Rosenfeld point out several other heuristics in the document image that aid in the task. For instance, "hooking" is an effect that occurs in the ink trace when the writer anticipates the placement of the writing instrument for the next stroke before finishing the current stroke. Using this for temporal recovery, we can determine the direction of the next stroke by identifying hooks in the current stroke. Such clues can help us recover the temporal information when on-line approaches fail.

Several other systems attempt to recover temporal information to make handwriting recognition more accurate. Kato and Yasuhara present a method for finding the drawing order of single-stroke handwriting images using a graph-based tracing algorithm [16]. One interesting approach called "OrdRec" is proposed by Lallican et al. They attempt temporal recovery by constructing a graph of possible temporal orderings of the points in the ink trace and running an HMM-based recognition on the candidates to determine the most likely ordering [17]. We touch upon this idea of using a recognizer to select the most likely ordering in our search for the corrent page orientation, discussed in Section 3.1.2.

2.3.3 Commercial Products

In designing our system, it is useful to study both the academic literature and commercial products that are already available. The technology that is perhaps most similar in purpose to our perceptual interface is the Anoto digital pen and paper system. This system allows the user to write on special paper using a digital pen, which contains a tiny camera that registers the pen's movement across a grid surface on the paper [1]. Though this technology offers clear advantages over less natural digitizer systems and even digital whiteboard products [2], it still requires the user to adapt to a special pen and paper. Until a commercial pen-based input system allows the user to write using regular pens and writing surfaces, there is still room for human-centric research in this area.

Chapter 3

Handwriting Capture

The main component of PADCAM provides handwriting capture capabilities; from a video of handwriting it recovers the time-ordered strokes that were written. We achieve this task by sending the video frames through a pipeline of sequential stages of processing, outlined in Figure 3-1. At a high-level, these stages are roughly classified into pre-processing, temporal recovery, and stroke segmentation. In this stage, we describe in detail the computer vision algorithms we use in each stage.

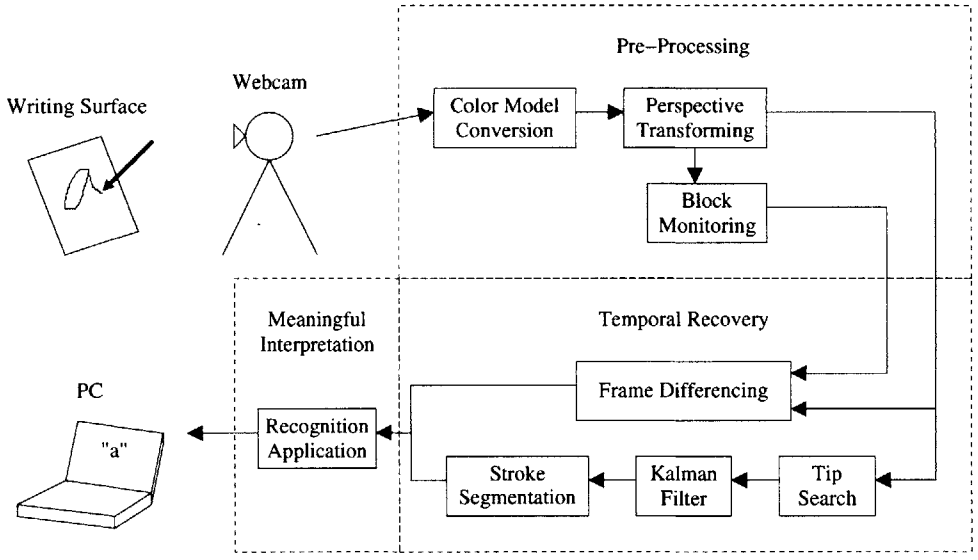

Figure 3-1: Block Diagram of PADCAM.

3.1 Pre-Processing

Before we begin recovery of the strokes, our pipeline requires initialization and other pre-processing.

3.1.1 Color Model Conversion

The first step in the pipeline is to get the data into a standard, useful format. In order to support a wide range of cameras, we use the video4linux interface to access the video. The problem is that different cameras deliver the video frames with different color models. Since most of our image processing is based on luminance, we convert the frame to a color model that separates luminance (brightness) from chrominance (color). We chose YUV because it is such a color model commonly used for video.

3.1.2 Page Detection

Because our interface allows the user to place the writing surface anywhere in the camera's view, the first step after color model conversion is to locate the surface. We make the reasonable assumption that the surface will be four-sided and roughly monochromatic. Note that monochromatism is not a strict limitation—in our tests, the system worked well with white and yellow pads with horizontal lines.

To detect the surface, we use a simple and intuitive algorithm. First, we perform Canny edge detection on the image which gives us a binary image with edge pixels colored white and all other pixels colored black [5]. In most cases, the contrast between the writing surface and the underlying desk is sufficient for the Canny edge detection to highlight the borders of the writing surface. However, due to noise from the camera and ambient lighting, the detected border often contains gaps. We fill the gaps by using a dilation filter, which "bleeds" the white parts of the image, as illustrated in Figure 3-2. Then we find lines in the resulting image using a Hough transform [27]. In most cases, the lines found coincide with the borders of the page.

Once we have lines bordering the writing surface, we need to find the corners in order to

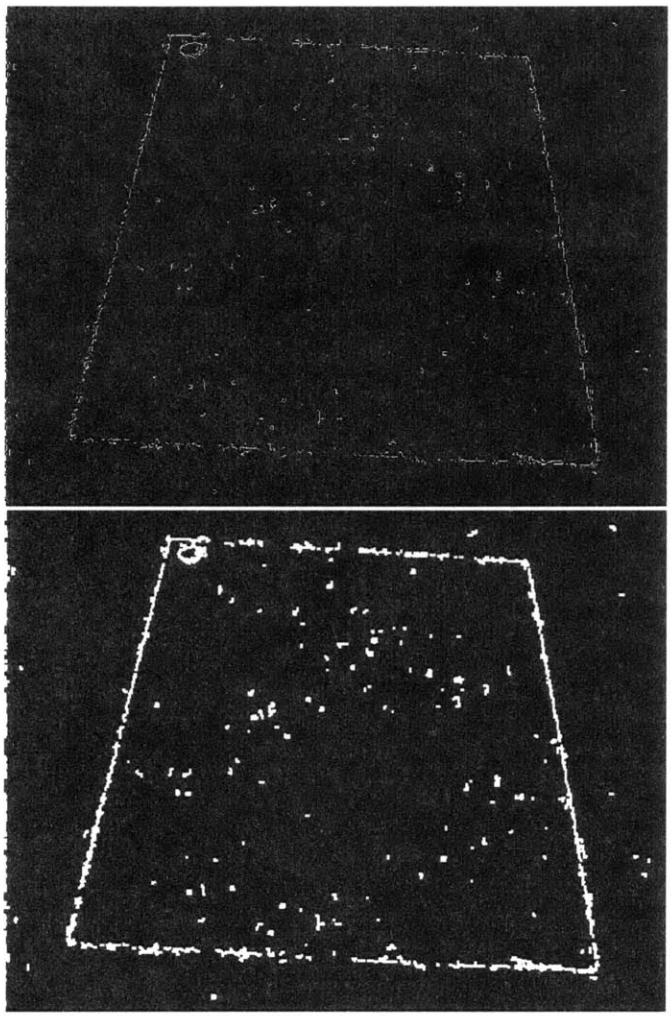

Figure 3-2: An Example of Dilation. (a) Sample Canny edge detection results. (b) The effect of dilation on the detected edges.

transform the image from coordinates in the image-plane to coordinates in the page-plane. First, we assume that the intersections between adjacent lines on the border occur near the corners of the page. We find these by calculating the intersections between lines for which the difference in slopes is above a certain threshold—which we call *high-angle intersections.* By limiting our search to high-angle intersections, we both reduce computation and eliminate non-corner intersections, such as those between almost-parallel lines.

The next step is to classify each high-angle intersection based on which corner it represents. To do this, a bounding box containing all high-angle intersections is computed and the center point of this box is chosen as the partition for classification. Each high-angle

intersection is then compared to the center point and classified as one of the four corners, and the centroid for each corner is computed. Note that at this point in the algorithm, we do not know the orientation of any of the corners (i.e. which is top-left, bottom-right, etc.). However, we do know their relation to each other—adjacent corners will remain adjacent and opposite corners will remain opposite in the correctly-oriented page. Because these relations are important, we store the corners in the order that they were classified: top-left, top-right, bottom-right, then bottom-left.

At this point, the only remaining task is to determine the orientation of the writing surface we have outlined. To avoid placing strict requirements on the orientation, we devised a simple way for the user to communicate this information to the system. Before page detection, the user draws a small red dot in the top-left corner of the writing surface. After acquiring the corner coordinates, the system searches for this mark in the neighborhood around each corner. Because we found the corners in clockwise order, we can easily deduce that the marked corner is the top-left; the next is top-right; the next is bottom-right, and the final corner is the bottom-left.

Note that finding the orientation explicitly is not always necessary. In many cases, the orientation can be deduced from the domain of the pen input. For instance, if the domain is handwriting, a general left-to-right, top-to-bottom pattern is expected in the input, so the corners can be assigned based on the spatial relation of sequential strokes. In fact, as a general solution the input can be run through the recognizer in all four orientations in parallel, and the actual orientation can be chosen based on which recognition produced the best results. Despite the elegance of these implicit orientation strategies, they require some knowledge of the input domain, which we did not want to assume. Because of this, we kept our explicit method of searching for the red dot.

Our procedure for detecting the page is explained in pseudocode in Algorithm 1. While page detection involves a great deal of computation, we expect to have to perform it only once at the beginning of the video sequence; thus, the overall performance of the system is not adversely affected by this step.

Algorithm 1 Detects the Page in the Current Video Frame

$edges \leftarrow CannyEdgeDetection(frame)$
$smoothEdges \leftarrow Dilate(edges)$
$lines \leftarrow HoughLineFinder(smoothEdges)$
$num \leftarrow 0$
for $i = 0$ to $lines.size$ **do**
  for $j = 0$ to i **do**
    if $|lines[i].angle - lines[j].angle| > threshold$ **then**
      $intersections[num] \leftarrow IntersectAtPoint(lines[i], lines[j])$
      $num \leftarrow num + 1$
    end if
  end for
end for
$bbox \leftarrow CalculateBoundingBox(intersections)$
$center \leftarrow Centroid(bbox)$
$cornerbuckets \leftarrow ClassifyClockwise(intersections, center)$
$redmax \leftarrow 0$
for $i = 0$ to 4 **do**
  $corners[i] \leftarrow Centroid(cornerbuckets[i])$
  $redvalue \leftarrow SearchForRed(corners[i])$
  if $redvalue > redmax$ **then**
    $redmax \leftarrow redvalue$
    $redcorner \leftarrow i$
  end if
end for
$topleft \leftarrow corners[redcorner]$
$topright \leftarrow corners[(redcorner + 1) \bmod 4]$
$bottomright \leftarrow corners[(redcorner + 2) \bmod 4]$
$bottomleft \leftarrow corners[(redcorner + 3) \bmod 4]$

Explanation of Subroutines:
$CannyEdgeDetection(image)$ - applies Canny edge detector on *image* and returns an image containing edges in white.
$Dilate(image)$ - returns a dilated version of *image*.
$HoughLineFinder(image)$ - applies a Hough transform on *image* and returns an array of infinite lines, specified by their offsets and angles.
$IntersectAtPoint(line1, line2)$ - returns the point at which *line*1 and *line*2 intersect.
$CalculateBoundingBox(points)$ - scans through the input array of points and returns a rectangle specifying the bounding box that contains all points.
$Centroid(box)$ - returns the point at the centroid of input rectangle or the input array of points.
$ClassifyClockwise(points, center)$ - classifies each point in *points* based on its relation to *center*, assigning points in a clockwise direction.
$SearchForRed(point)$ - calculates the amount of red in the pixels around *point*.

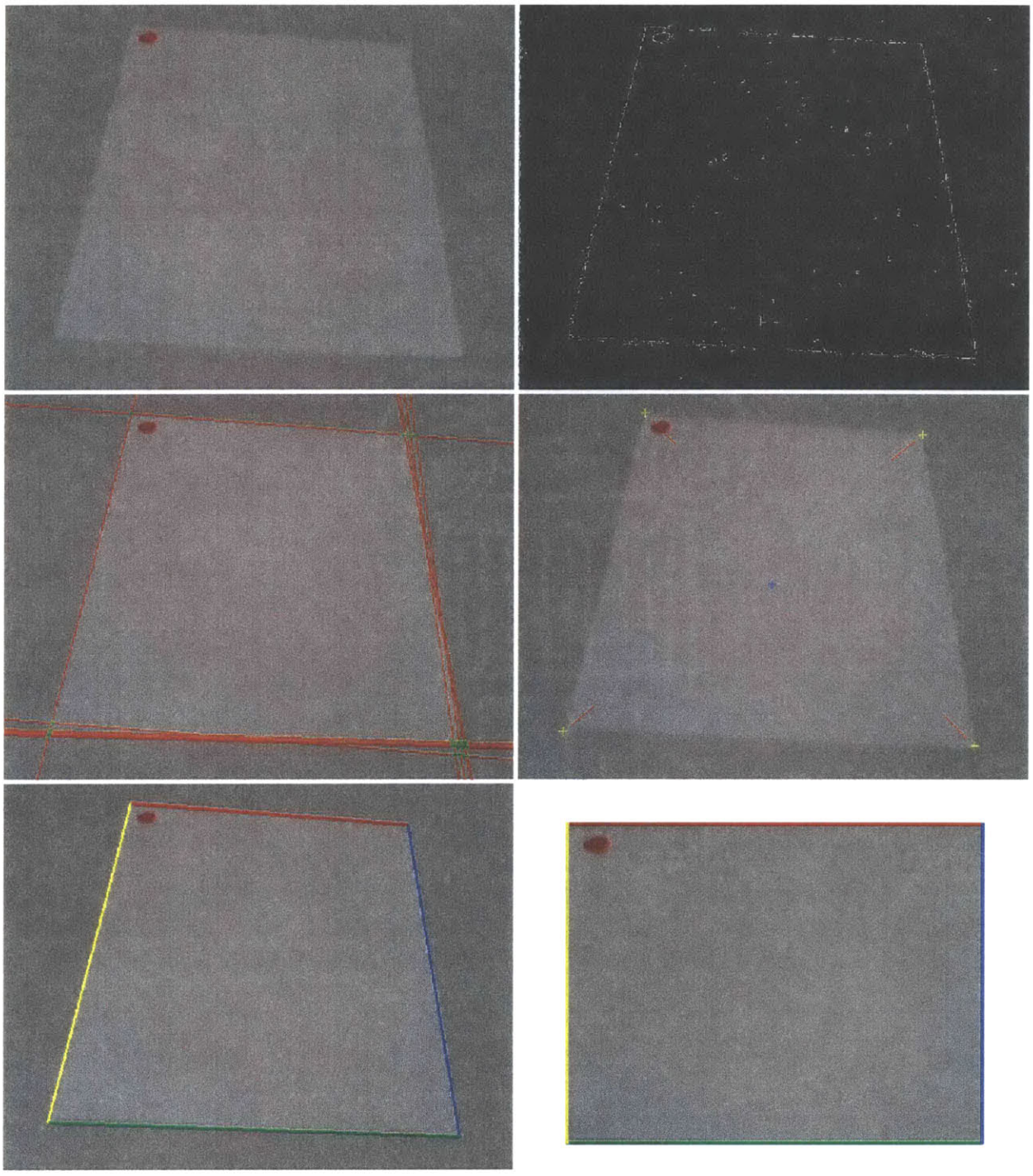

Figure 3-3: Page Detection. (a) Original video frame. (b) Results of Canny edge detection. (c) Lines found in Hough transform and the high-angle intersections in green. (d) Centroid in blue, corners in yellow, and red dot search space in red. (e) Detected borders of the page where red is top, blue is right, green is bottom, and yellow is the left. (f) Transformed page with the same border colors. Note: for the best results, view this figure in color.

3.1.3 Perspective Transforming

Once we know the corners and orientation of the writing surface, we can transform the video frame from coordinates in the image-plane to coordinates in the page-plane. We do this by first calculating a matrix that defines the necessary perspective transform based on the corners and applying this transform on the frame. This helps us by both reducing computation and simplifying the tracking because all further image processing is restricted to just the writing surface.

3.1.4 Background Recording

Our frame differencing algorithms attempt to determine the change in the appearance of the writing surface due to ink, which requires us to record what it looked like before the ink. Though it may seem sufficient to simply store an initial frame as the pre-ink image, we found that using a slightly more sophisticated approach yields more robust tracking. We model each pixel on the writing surface, or background, as a mixture of Gaussian random variables. To acquire our model of the background, we record a few seconds of frames and then compute the mean and standard deviation of each pixel. By saving this information, we can compare subsequent frames to the mean background frame and determine whether each pixel's deviation from its mean value significantly exceeds its standard deviation. This more sophisticated comparison compensates for noise in the image acquisition, such as from the camera or lighting conditions.

3.1.5 Block Monitoring

In addition to a model of the background, our frame differencing algorithms also require information on when writing began and ended in each of the areas, which we will refer to as *blocks*, on the writing surface. Rather than processing the whole video sequence from start to finish, these algorithms segment the writing into blocks and process each block independently. As we shall see later in Section 3.2.1, we need to provide initial and final frames for each block. We obtain these frames using an algorithm we call *block monitoring.*

The basic idea of block monitoring is to "watch" the page for significant changes and save snapshots at key times between changes. In each iteration, a *key frame* is saved, and all subsequent frames are compared to this frame to detect if anything has changed on the page. If so, this *change frame* is saved, and subsequent frames are compared against the change frame to determine if any large rectangular block that changed from the key to the change frame did not change since the change frame. If so, we assume that this block is stable, and we use the key and change frames as the initial and final frames for that block. If no stable block can be found, there must still be motion on the page, and we must wait until the motion moves to another area of the page before declaring the block stable.

From this description, it seems that our block monitoring algorithm requires a great deal of storage space in order to save key and change frames. One key aspect of this problem is that there is no penalty for waiting too long before declaring a block stable—the ink does not change once it is on the paper (assuming no smudges). We can take advantage of this fact by lowering the temporal resolution of our block monitoring algorithm. Instead of running this algorithm on each frame in the video sequence (or 30 times per second in a real-time video feed), we run it once every second. Even at this low temporal resolution, we can still accurately monitor the ink pattern and find blocks effectively.

3.1.6 Tip Acquisition

The tracking-based algorithms we used for temporal recovery must be initialized with a model of the pen tip. To satisfy this requirement we have two options:

1. Save a model of some predetermined pen and require that only that pen be used with the system.

2. Allow any pen to be used with the system and perform initialization to acquire a model of that pen's tip.

In keeping with our design goal of allowing a natural, human-centric setup, we chose the latter option, which allows the most freedom in terms of writing instruments. The cost of this option is that our system must be able to acquire the tip of any arbitrary writing

instrument. Fortunately, Munich had already developed a robust algorithm for doing this, so rather than reinventing the wheel, we used his approach, explained in Algorithm 2.

Algorithm 2 Acquires a Model of the Pen Tip (original algorithm from Munich [21])

1: Compute the difference between the current and previous frames within the rectangular box until enough pixels have a difference that exceeds a threshold, then proceed to step 2.
2: Compute the difference between the current and previous frames within the rectangular box until there is no motion detected in the box, then go to step 3.
3: Perform Canny edge detection within the box and fit a parabolic cylinder to the contrast surface in the neighborhood of each pixel.
4: Within the neighborhood, select a pixel only if it has sufficient contrast and the axis of its parabolic cylinder is close enough to the pixel's center.
5: Find the centroid of the selected pixels.
6: Classify each selected pixel into one of the four quadrants and take the mean orientation and count of the pixels in each quadrant.
7: Repeat steps 3-6 for several frames to compensate for noise in the frames, then proceed to step 8.
8: Find the mean position of the centroids from step 5.
9: Determine the most voted quadrant, which represents the edge of the pen tip that was detected most reliably. Compute the mean orientation for this quadrant.
10: Determine the second most voted quadrant, which is the second most reliably detected edge of the pen tip. Given the mean centroid position and the estimated orientation of one edge of the pen tip, the profile of the image is searched perpendicular to this orientation to find points with the highest contrast.
11: Calculate the pen tip's orientation as the mean of the orientations obtained in steps 9 and 10.
12: Get the profile of the image along a line that passes through the centroid obtained in step 6 with the orientation from step 9.
13: Find the position of the tip point and the finger in this profile by applying 1D edge detection on the image profile. Recompute the centroid position as the mean of the locations of the finger and the tip point.
14: Extract the template of the pen tip by copying the neighborhood around the centroid computed in step 11.

3.2 Temporal Recovery

We experimented with two different high-level approaches to recover the time-ordered points in a video sequence. The first was our own conception, based on the differences between consecutive frames in the video sequence. Within this broad approach, we tried two different

algorithms for finding the points. The second high-level approach was based on pen-tracking and was inspired by Munich's system. We tested an implementation of Munich's original Kalman tracker. The descriptions of each algorithm follow.

3.2.1 Frame Differencing

Our initial idea for reconstructing the time information in handwriting came by simply applying common sense to the problem. We assumed that while the user is writing on the page, the instantaneous change that we will see will consist of the movement of the user's hand and the current blob of ink being left on the page. We speculated that if can find a way to segment out the user's hand, then we can recover the ink blobs in the order they were left on the page, thus solving our problem. Now the challenge becomes finding an effective way to distinguish between movement of the user's hand (which we want to ignore) and the new ink blobs. Using the differences between consecutive frames to approximate instantaneous change, we developed two algorithms, each with its own method for classifying the differences, for reconstructing the order of the ink blobs.

Masked Differencing

Our initial solution based on frame differencing was straightforward but somewhat naive. First, we realized that the overall ink trace left on the page could be computed by simply finding the difference between the initial and final frames, assuming that neither frame contains the user's hand. By thresholding the overall ink trace, we could essentially create an *ink mask* with ones where ink was left and zeros elsewhere. Recall from above that the effectiveness of our frame differencing approach was contingent upon how well we can distinguish between movement of the hand and ink blobs. To make this classification, we decided to assume any motion occurring within our ink mask was due to ink blobs. So in this algorithm, the intersection between the ink mask and each frame-to-frame difference is computed, leaving only the ink blobs in the order that they were left. Algorithm 3 lists the procedure in pseudocode, and Figure 3-4 illustrates a simple example of the algorithm in action.

Algorithm 3 Recovers Points using Masked Difference between Frames.

$init \leftarrow frame$
$num \leftarrow 0$
repeat
  $savedframes[num] \leftarrow frame$
  $num \leftarrow num + 1$
until final frame signal received from block monitoring
$final \leftarrow frames[num - 1]$
$inktrace \leftarrow |final - init|$
$inkmask \leftarrow Threshold(inktrace, K, 0, 255)$
for $i = 1$ to num **do**
  $diff \leftarrow savedframes[i] - savedframes[i - 1]$
  $points[i] \leftarrow Centroid(inkmask\&diff)$
end for
return points

Explanation of Subroutines:
$Threshold(image, K, low, high)$ - compares each pixel in *image* to K and sets the corresponding pixel in the returned image to *low* if less than and *high* if greater.
$Centroid(image)$ - computes the mean position of the nonzero pixels in *image*.
Note: Arithmetic operations on frames are performed on all pixels in that frame.

Though our masked differencing algorithm seems as though it would work, our results show that it does not accurately recover the points. The main problem is with our assumption that any motion occurring within the ink mask was due to ink blobs; this does not hold. Though we can safely assume that motion outside of the ink mask is not due to ink, we cannot conversely assume that motion inside the ink mask is necessarily due to ink. The following scenario illustrates this fact. Suppose the while writing the numeral "7" a right-handed user draws the top line from left to right, then the diagonal line from top to bottom. While scribing the top line, the user's hand and pen obscure the pixels where the diagonal line will soon be, which looks like ink motion to the masked difference algorithm. Because of this, the algorithm will incorrectly determine that the first ink blobs occurred at the bottom of the numeral rather than at the top-left corner, as shown in Figure 3-5.

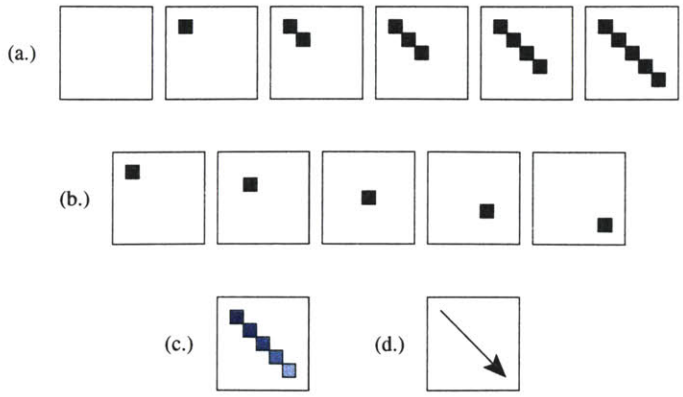

Figure 3-4: Example of Masked Differencing. (a) Simplified video sequence showing the sketching of a diagonal line. (b) Masked differences between frames with black indicating ink blobs. (c) Final sequence of points with more recent points in lighter blue. (d) Resulting path specified by an arrow.

Backward Differencing

After testing our masked differencing algorithm and learning from its flaws, we developed a new algorithm for recovering the temporal information in the pen input. In this approach, we model the change in a pixel's intensity and use this to determine if and when it becomes part of the ink trace. As before, the ink mask is obtained by thresholding the difference between the initial and final frames. To help determine the order of the points, we construct a specialized data structure called a *motion history image* (MHI). In general terms, a MHI is essentially an image with pixel intensity values corresponding to when the more recent motion occur in that pixel's position—so a bright area of the MHI indicates that there was recent activity in that area of the video sequence [8]. In the context of our task, the MHI is used to store a timestamp indicating the most recent significant change in a pixel's intensity. We call a pixel that undergoes such a change a *transition pixel.* To construct the MHI, the algorithm iterates backward through the sequence of frames and calculates the difference between each frame and the final frame. This difference image is then thresholded and all intensity values exceeding some constant K (transition pixels) are set to the timestamp of the current frame in the resulting *motion image.* Then the MHI is updated to reflect any new timestamped transition pixels. After iterating through all frames in the sequence, the ink blobs are found by iterating forward through the timestamps in the MHI and computing

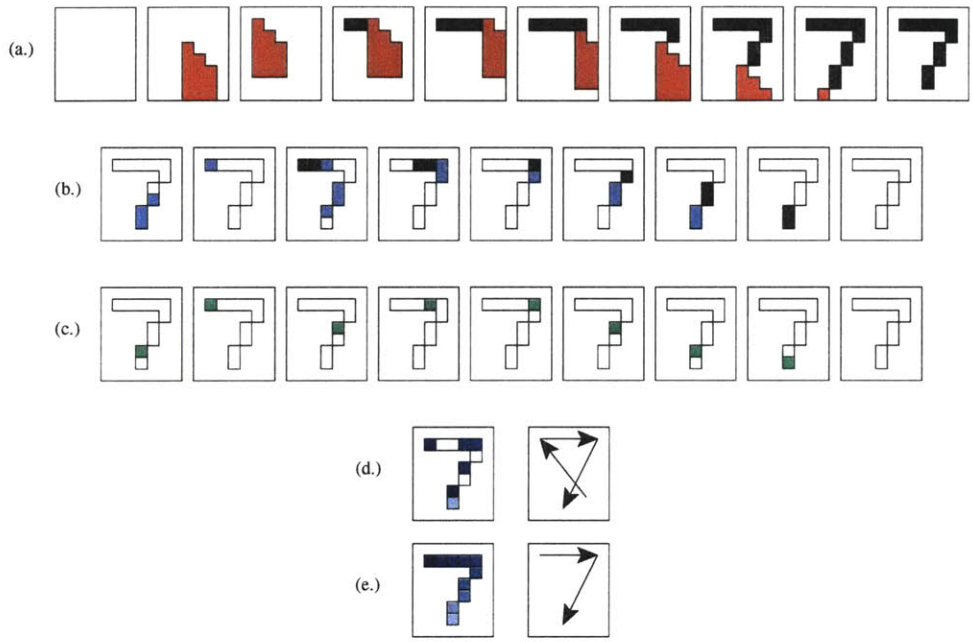

Figure 3-5: Problem with Masked Differencing. (a) Example video sequence showing the construction of the numeral "7" with non-ink pixels (such as from the pen or the hand) in red and ink pixels in black. (b) Masked differences between frames with blue indicating non-ink motion pixels and black indicating ink blobs. (c) Centroids of each masked difference. (d) Final sequence of pixels with more recent pixels in lighter blue and the corresponding path indicated by the arrows. (e) Actual sequence of pixels and corresponding path.

the centroid of the transition pixels for each timestamp. Algorithm 4 lists the pseudocode for this procedure.

Algorithm 4 Recovers Points using Backward Differencing.

$init \leftarrow frame$
$num \leftarrow 0$
repeat
  $savedframes[num] \leftarrow frame$
  $num \leftarrow num + 1$
until final frame signal received from block monitoring
$final \leftarrow frames[num - 1]$
$inktrace \leftarrow |final - init|$
$inkmask \leftarrow Threshold(inktrace, K, 0, 255)$
for each pixel p in mhi **do**
  $p \leftarrow -1$
end for
for $i = num - 1$ to 0 **do**
  $diff \leftarrow |savedframes[i] - final|$
  $motion \leftarrow Threshold(diff, K, 0, i)$
  $mhi \leftarrow Max(mhi, motion)$
end for
for $j = 0$ to num **do**
  $inkblob = Equals(mhi, j)$
  $points[j] = Centroid(inkmask \ \& \ inkblob)$
end for

Explanation of Subroutines:
$Threshold(image, K, low, high)$ - compares each pixel in *image* to K and sets the corresponding pixel in the returned image to *low* if less than and *high* if greater.
$Max(image1, image2)$ - returns an image with the maximum pixel values from image1 and image2.
$Centroid(image)$ - computes the mean position of the nonzero pixels in *image*.
$Equals(image, k)$ - compares each pixel in *image* to k and sets the corresponding pixel in the returned image to 1 if equal and 0 if not equal.
Note: Arithmetic operations on frames are performed on all pixels in that frame.

3.2.2 Pen Tracking

An alternative approach to detecting ink blobs would be to track the pen tip in the video sequence and use its position to approximate where ink blobs will be placed on the page. This approach can be broken down into four steps: pen tip acquisition, tip search, filtering,

and stroke segmentation. As described in Section 3.1.6, the first step is performed during initialization of the system. The remaining three steps are explained below.

Tip Search

The most basic goal in visual tracking is finding the tracked object in each frame in the video sequence. Since we obtained a model of the tracked object in our tip acquisition step, the remaining task is to search the video frames for this pen tip. We approach this problem as a signal detection task, where the input 2-D signal is the array of pixel intensities in the video frame and the desired signal to be detected is the array of pixel intensities in the acquired pen tip model. A straightforward solution is to apply a matched filter—a linear filter that looks like the signal to be detected—to the entire input signal. We employ this approach in our tip search by calculating the normalized correlation between the pen tip model and the neighborhood of the input frame centered on the *predicted position* of the pen tip. The next section will describe how this position is predicted. We assume that if the pen tip is contained within the searched neighborhood, it will be located at the point yielding the maximum normalized correlation, so this point is taken as the position of the pen tip. If the maximum normalized correlation is below some threshold, we claim that the pen tip is not currently in the search neighborhood, and we must expand the neighborhood until it is found again.

Kalman Filter

Our tip search method is based upon the notion of a search neighborhood around a predicted position of the pen tip. Following Munich's lead, we use a recursive estimation scheme called a Kalman Filter to provide these predictions [15]. The filter takes as input the observed historical pen trajectory and a model for the pen tip's motion and outputs a prediction for where the pen tip will be next. The model for the pen tip's motion is based on simple kinematics:

$$\mathbf{x}(t+1) = \mathbf{x}(t) + \mathbf{v}(t) + \frac{1}{2}\mathbf{a}(t) \tag{3.1}$$

$$\mathbf{v}(t+1) = \mathbf{v}(t) + \mathbf{a}(t) \tag{3.2}$$

$$\mathbf{a}(t+1) = \mathbf{a}(t) + \mathbf{n_a}(t) \tag{3.3}$$

$$\mathbf{y}(t) = \mathbf{x}(t) + \mathbf{n_y}(t) \tag{3.4}$$

where $\mathbf{x}(t)$, $\mathbf{v}(t)$, $\mathbf{a}(t)$ are the two-dimensional components of position, velocity, and acceleration of the pen tip, $\mathbf{n_a}(t)$ and $\mathbf{n_y}(t)$ are zero-mean Gaussian random variables, and $\mathbf{y}(t)$ is the predicted position of the pen tip given a noisy observation. The specifics of Munich's implementation, which serves as the basis for our tracking module, are covered in detail in reference [21].

Stroke Segmentation

One key distinction between our frame differencing algorithms and the pen tracking approach is that the latter does not provide pen-up and pen-down events. Instead it returns a complete time-ordered list of the pen's trajectory as though it were a single stroke. For applications such as cursive handwriting, this unsegmented output may be acceptable. However, we aim to handle arbitrary pen input which requires us to segment the pen tracker's output into strokes based upon when pen-up and pen-down events were detected. We considered approaches based on tracking and ink detection.

Tracking-Based Stroke Segmentation

While testing the tracking system, we found that it would often lose track of the pen when writing expressions with multiple strokes. We observed that very often the reason that tracking was lost was that the user picked up the pen and moved it quickly to another location to start another stroke. Usually the system would reacquire tracking shortly after the start of the next stroke, coinciding with the pen-down event. From these observations, we decided to interpret a loss of tracking as a pen-up event and reacquisition of tracking as a pen-down event. Though this yielded reasonable qualitative results as shown in Figure 3-6, not all of the pen-up and pen-down events would be detected using this simple approach, so we explored other methods of stroke segmentation.

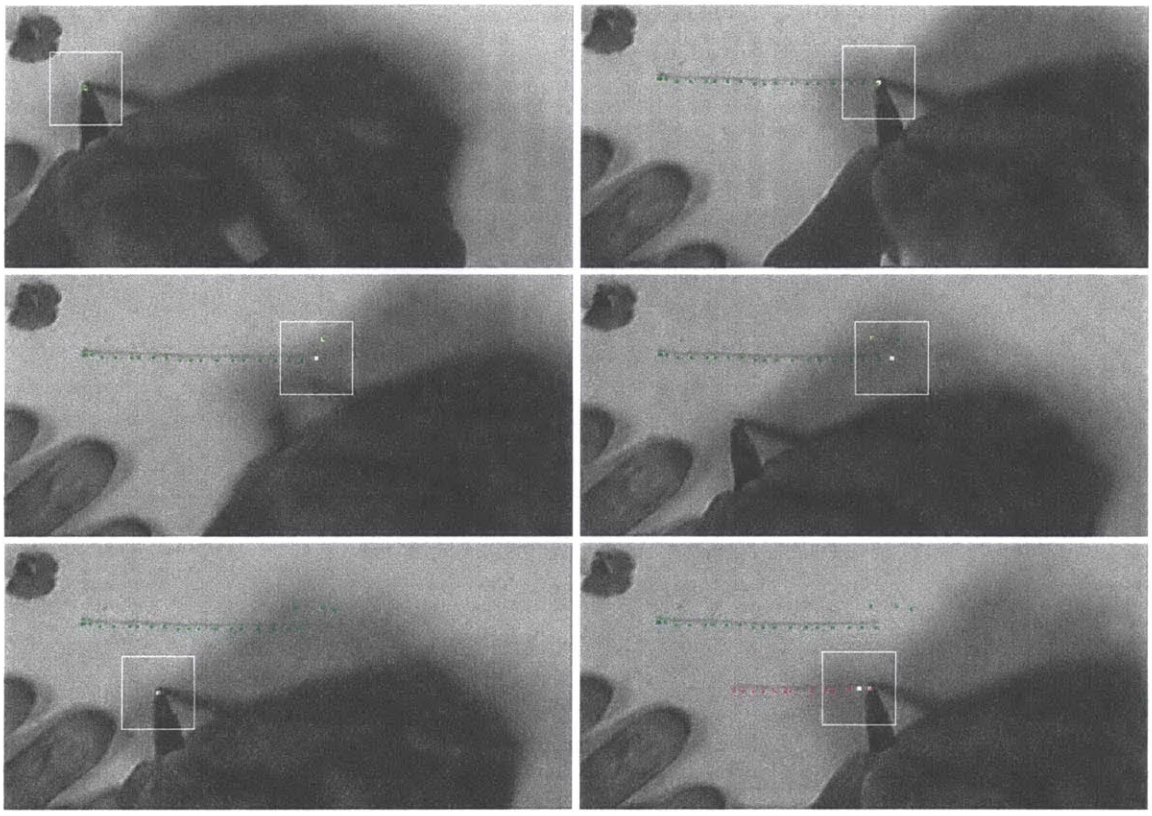

Figure 3-6: Demonstration of Tracking-Based Stroke Segmentation. (a) Tracking begins. (b) Tracking continues to the end of the stroke. (c) User picks up the pen tip and moves toward the beginning of the next stroke; tracking is lost. (d) User places the tip down on the page. (e) Tracking is reacquired and a new stroke is started. (f) Tracking resumes normally on the second stroke.

Ink-Based Stroke Segmentation

Perhaps the most direct way to detect pen-up and pen-down events is to check whether each tracked position of the pen tip resulted in an ink trace. Modeling our pen input as a finite state machine, we can easily determine whether the current state is either *ink trace* or *pen movement* based on the most recent pen-up or pen-down event, as illustrated in Figure 3-7. Because our tracker returns the location of the pen tip in the current frame, we cannot

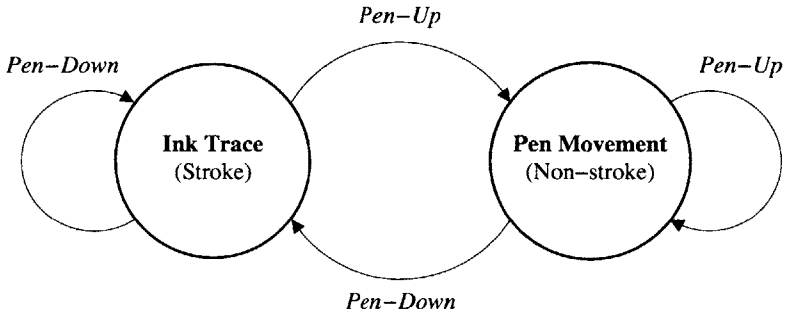

Figure 3-7: Pen Input as a Finite State Machine. The two states, ink trace and pen movement, are shown in bold. Transition events are shown in italics.

immediately test for the presence of ink at this location because it will be occupied by the tip. Thus, our ink-based stroke segmentation module waits a few frames for the pen tip to leave that location and then measures the observed intensity. Recall that in initialization we recorded a model of the writing surface with the mean and standard deviation of each pixel location's intensities. We use this information in stroke segmentation to compute the normalized difference between the observed intensity and the mean for the writing surface. If this difference exceeds a predefined threshold, it is assumed to be an ink trace. Otherwise, it is assumed to be pen movement.

3.3 Implementation and Setup

As described, the PADCAM system could be implemented on any hardware configuration with video capture capabilities. Our testing platform was an Intel Pentium III 1.13MHz PC with a commodity USB camera. Though we could have opted for a more expensive

high resolution camera, we wanted to prove that the system could work with commodity components, so we chose the Philips Vesta Pro Scan PCVC690K webcam. The physical setup of these components is depicted in Figure 3-8.

The PADCAM software is written in C for the Linux platform. Aside from its attractiveness as a development platform, the decision to implement on Linux was beneficial for two reasons. First, we could take advantage of the video4linux interface for video capture devices, which allowed us to experiment with many cameras without changes to the software. Second, the existence of a rapidly maturing handheld Linux distribution meant that our software would be easily adaptable for iPAQs and other handheld devices.

We made extensive use of Intel's OpenCV computer vision library in our image processing code. This package provided many useful vision routines and utility functions that simplified the coding. Also, the library was optimized for Intel's MMX and SSE technologies, which gave us better performance.

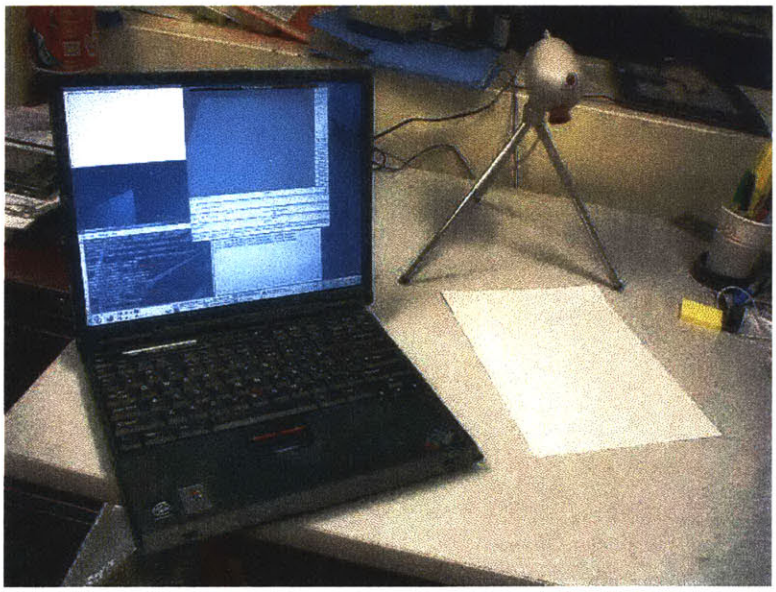

Figure 3-8: Physical Setup of PADCAM System

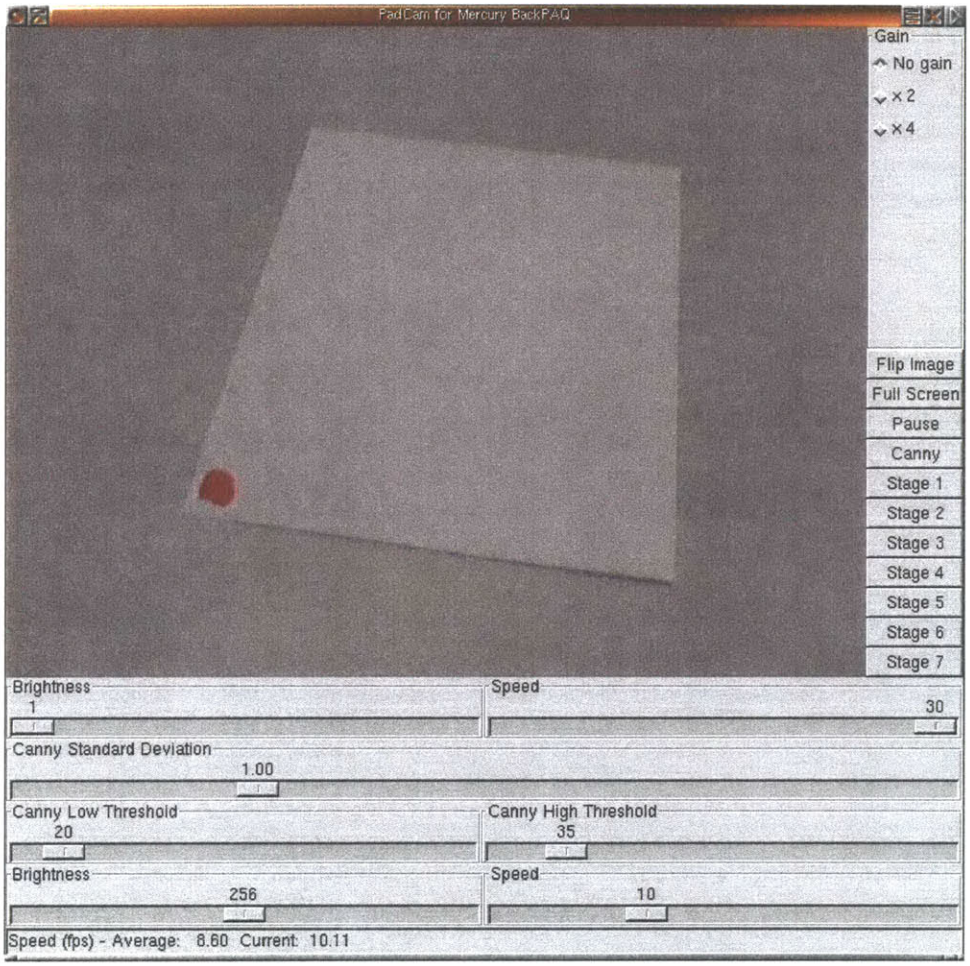

Figure 3-9: Screenshot of PADCAM Testing Application

3.4 Experimental Results

3.4.1 Page Detection

Our page detection algorithm performs well in most cases. We tested the accuracy of this component by hand-selecting the actual corners of the page and comparing these to the corners obtained from page detection. By calculating each detected corner's Euclidean distance from the respective hand-selected corner, we obtained an estimate of the average error in page detection. For analysis, the length of the diagonal across the page was also recorded—this gives us some indication of how large the page appeared in the video frames. Though we might expect that page detection accuracy would increase as this diagonal lengthens, the plot in Figure 3-10 shows that this is not the case. This result makes sense because although a larger page gives us a better picture of the page, it also means that our errors will be larger.

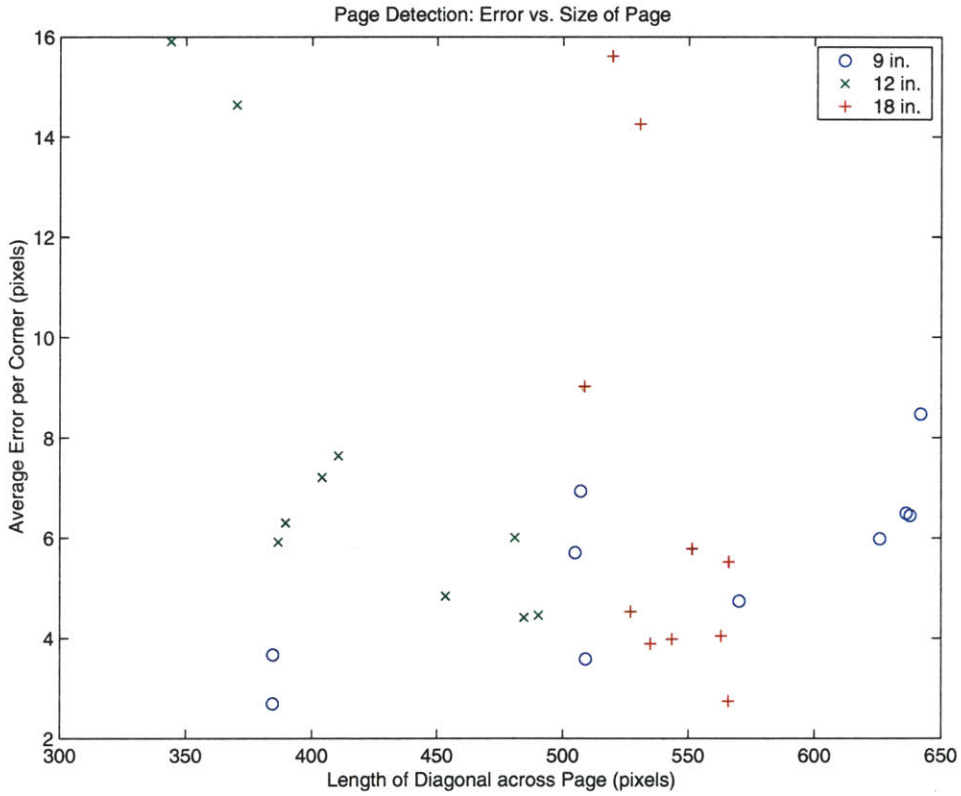

Figure 3-10: Plot of Page Detection Error versus the Size of the Page

Table 3.1: Average Error in Page Detection

Distance (inches)	Diagonal Length (pixels)	Average Error (pixels)
9	540.18	5.47
12	421.36	7.73
18	541.07	6.94

We speculated that as we placed the camera closer to the page, the page detection accuracy would increase. To test this, we varied the distance between the camera lens and the center of the page and ran 10 trials at each of the three distances. Our findings are listed in Table 3.1. As can be seen in the boxplot in Figure 3-11, there was very little correlation between distance and page detection error. However, we can also see that the variability of the error is high at the farthest distance we tested, 18 inches. We attribute this to the fact that when the camera is farther away from the page, the video frame has higher variability due to illumination changes and interfering objects, so the page detection accuracy will be more volatile in turn.

Though the page detection module's performance met our expectations, it was interesting to analyze the cases in which it failed. Figure 3-12 shows an example where too many interfering objects were placed in the field of view. Since our algorithm expects a somewhat controlled field of view containing only the page, it cannot handle situations with clutter. Another problem we encountered stemmed from camera distortion, as illustrated in Figure 3-13. The pictured page is so distorted that the page detector cannot fit a single line across each of the top and bottom edges, so instead it fits two lines and takes the average over these two. While this method will work reasonably, the correct approach would be to calibrate the camera in initialization and undistort the video frame. However, this would require additional setup, which would detract from the natural, human-centric feel of the system.

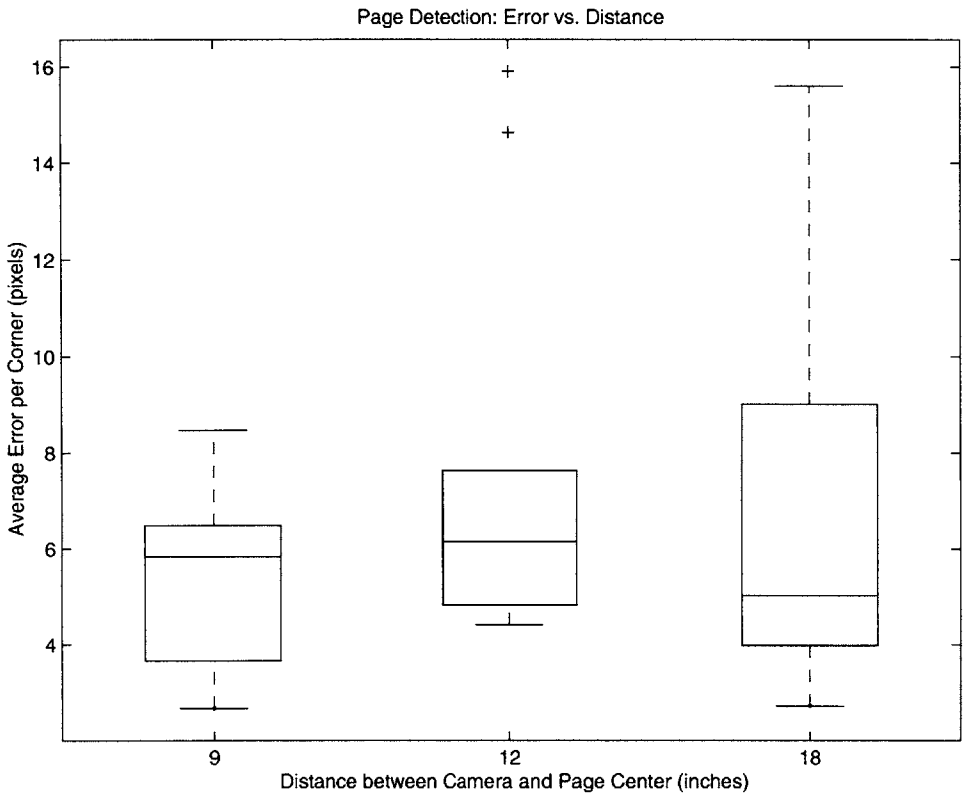

Figure 3-11: Boxplots of Page Detection Error for Tested Camera-to-Page Distances.

3.4.2 Temporal Recovery

Frame Differencing

As mentioned in Section 3.2.1, the masked differencing algorithm for temporal recovery was unsuccessful because it would recognize hand and pen movement as new ink blobs. Though we tried to solve this problem with backward differencing, in the end neither approach based on frame differencing yielded usable results.

Upon analysis of the backward differencing results, we found the problem to be that few of the pixels in the ink trace settle upon their final intensity value until the last few frames. As a result, the timestamp for each recovered point would usually be toward the end of the sequence, rather than when the point was actually added to the ink trace. This symptom seems to indicate that our algorithm is thrown off by transient shadows that are cast during writing.

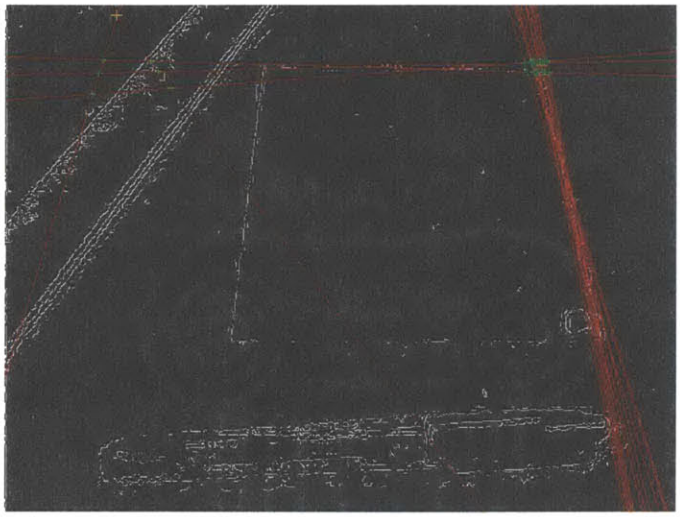

Figure 3-12: Example of a Cluttered Writing Surface. Our page detection algorithm will fail for this video frame because it contains too many objects that distract away from the actual page. The frame is shown after the Canny edge detection and Hough line detection steps to show the effects of the interfering edges. In the top-left corner of the video frame is a ruler and toward the bottom is a pen.

Pen Tracking

Fortunately, the tracking-based approach to temporal recovery performed well, as illustrated in Figure 3-14. We evaluated the accuracy of the tip search and Kalman filter separately from the stroke segmentation module, as the two components solve different problems and thus require different testing criteria. Then we measured the speed of the overall pen tracking system.

To determine the accuracy of the tip search and Kalman filter modules, we compared the their output of time-ordered points to reference points obtained using a Wacom digitizer tablet. This device has a resolution of 2540 lpi and accuracy of $\pm$ 0.01 inches, so the reference points we obtained from it were quite accurate. Much like our page detection tests, we varied the distance between the camera and the center of the writing surface. Our measure for accuracy was the average distance that each tracked point was from its corresponding reference point. We found that the tracked points were often offset from the reference points by a certain bias (usually only 4 or 5 pixels), indicating error in the tip acquisition process. In efforts to separate this error from the actual pen tracking error, we corrected the tracked

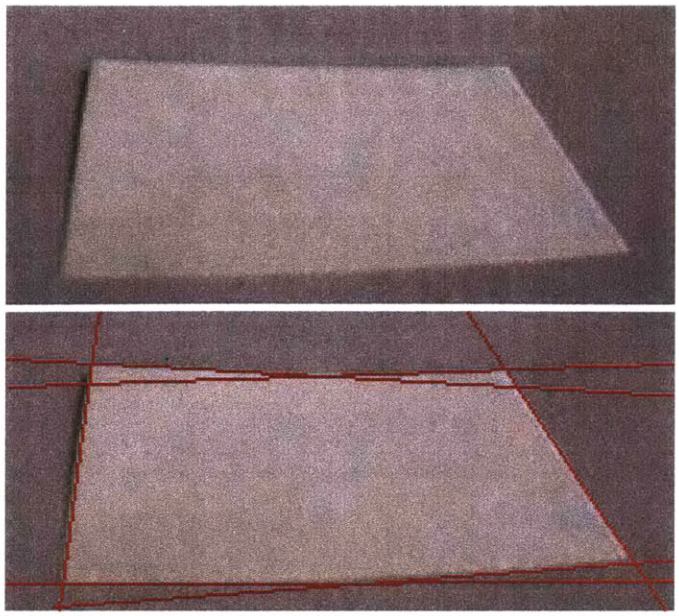

Figure 3-13: Example of a Distorted Page. Note how the bottom of the page is curved due to camera distortion. While our algorithm will detect most of the page, it is likely that the bottom corners will be inaccurate because we cannot fit one straight line across the bottom of the page. We approximate the bottom of the page by fitting two straight lines as shown in red in the lower panel and averaging the intersection points.

points for the bias. In our test sequences at each distance, we obtained the results shown in Table 3.2.

We measured the accuracy of both the tracking-based and the ink-based stroke segmentation algorithms. Recall that the stroke segmentation task basically involves classifying each tracked point as either ink trace or pen movement. We obtained reference values for our accuracy tests by hand-classifying each point. The results for 10 trials at different distances are shown in Table 3.3. Clearly, the tracking-based algorithm outperforms the ink-based

Table 3.2: Average Error in Pen Tracking

Distance (inches)	Average Unbiased Error (pixels)	Bias in x (pixels)	Bias in y (pixels)
9	6.23	3.4	-1.2
12	4.53	5.7	0.3
18	6.82	-3.9	6.0

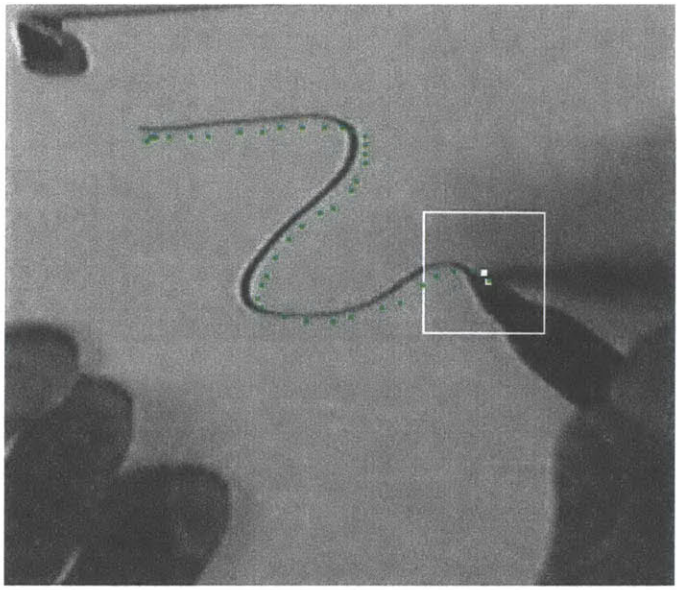

Figure 3-14: Visual Results of Pen Tracking. Recovered points are colored in green and the white box encloses the current neighborhood for the maximum correlation tip search.

Table 3.3: Accuracy of Stroke Segmentation Algorithms

Distance (inches)	Tracking-Based Algorithm % correctly classified	Ink-Based Algorithm % correctly classified
9	87%	48%
12	79%	24%
18	84%	22%

approach considerably. As with most of our other results, there does not seem to be a correlation between camera distance and accuracy, though it would take more trials to confirm this.

3.4.3 Cumulative Performance

We evaluated the speed of PADCAM as one cumulative unit and profiled it to determine which stages were taking the most time. Because our results from the frame differencing algorithms did not meet our expectations, we left out these stages, as well as block monitoring (which provided data necessary only to the frame differencing algorithms). Obtaining accurate timing data was challenging because we want to avoid the computational cost of displaying

Table 3.4: Processing Times for Stages in the System

Stage	Processing Time (ms)	Percentage of Total
Frame Capture	47.0	62.5%
Color Model Conversion	24.8	33.0%
Perspective Transforming	1.0	1.3%
Tip Search	2.4	3.2%
Kalman Filter	0.0	0.0%
Stroke Segmentation	0.0	0.0%
Total	75.2	100%

the GUI, but certain aspects of the tracking require some user interaction. For instance, positioning the pen for tip acquisition is difficult when the user cannot see the live frame for feedback. We tried saving a video sequence to disk and playing it back in a GUI-free session, but this solution simply replaces the GUI overhead with file system operations and JPEG decompression computations. Our eventual solution was to disable certain costly features of the GUI when they were not needed. For instance, after the tip was acquired, the application stopped displaying the video frames and processed them silently instead.

In our testing, we used the Philips webcam at 640x480 resolution and 15 frames per second. The overall system processed frames at 13.3 frames per second—not far below the limit of the camera. As shown in Table 3.4 and Figure 3-15, most of the time was spent capturing the video frames and converting the color models. The actual temporal recovery takes only 2.4 ms per frame or 3.2% of the overall processing time. Because our overall framerate is so close to the camera limit, it seems that video capture is the limiting step in our system. In future versions, we would consider using a camera with a higher framerate and perhaps switching from USB, which has a 1.5MB/s data transfer rate, to FireWire, which currently allows rates of up to 50MB/s.

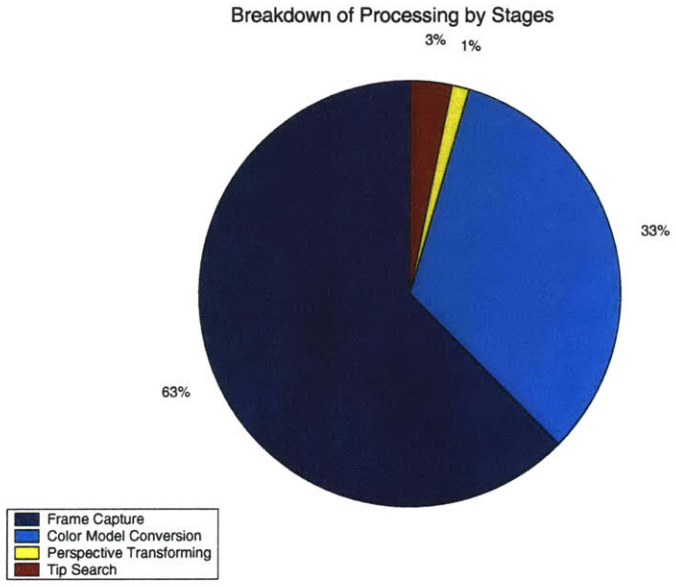

Figure 3-15: Breakdown of Processing by Stages.

Chapter 4

Specification for PADCAM on Handheld Devices

In efforts to keep the system as convenient and human-centric as possible, we attempted to implement it on a properly outfitted handheld device. This section describes our experience with this task and presents what we learned.

4.1 Motivation

Perceptual interfaces for pen-and-paper input, such as the system we described in the previous chapter, make great strides in human-centric computing. However, in keeping with the goals of Project Oxygen, we also aim to make this technology *pervasive*. We would like to implement the system on portable hardware, thus allowing it to go literally anywhere a pad and pencil can go. Though our prototype system could easily be run on a laptop with a camera, such a solution is unsatisfactory—a handheld device would provide much more flexibility and convenience. A handheld PADCAM system is desirable for two specific uses in the Oxygen view of the world—as a portable device and as a device embedded in the environment.

4.1.1 Portable PADCAM

The benefits of a portable interface for pen-based input are easy to see. If PADCAM were as easy to carry around as pen-and-paper it would be advantageous to always use PADCAM when taking notes. That way you could have PADCAM meaningfully interpret your notes while retaining a hardcopy for reliability.

It's also important to note that the hardware needed to support PADCAM is becoming available in all form factors as an industry trend. Recent advancements in CMOS imaging technology are making CMOS cameras cheap, low-power, and small enough to embed into cellular phone handsets. It's projected that by 2004, camera-enabled handsets will comprise 20% to 50% of the global mobile phone market [29]. Clearly, this creates an opportunity for us to provide PADCAM as a potential "killer app" for these new camera-enabled phones.

4.1.2 PADCAM in the Environment

Intelligent spaces are an essential component of the Oxygen initiative. These environments are embedded with cameras, microphones, displays, speakers, wireless networks and other devices called *E21s* in Oxygen terminology. With this in mind, we envisioned that we could extend PADCAM for use on a handheld to be used as another E21.

There are many advantages to using handhelds instead of full desktop or laptop PCs to embed PADCAM into the environment. Perhaps the most obvious are the practical advantages of space, heat, and power—we can place an iPAQ practically anywhere in the room without having to worry as much about heat and power consumption. Also, the fact that these devices are relatively small and light means we can move them around without much effort.

4.2 Special Considerations

Having decided to adapt PADCAM for use on handheld devices, we must consider some of the challenges we will face in doing so. We cannot simply recompile our code for an iPAQ and be done—handheld devices are designed with distinctly different specifications than regular PCs, so we must redesign our system taking these into account.

4.2.1 Computational Limits

One major concern in developing any software for handheld platforms is adapting to the reduced computational capacity and resources. Currently, a top-of-the-line handheld device typically boasts a 206 MHz processor and 64 megabytes of RAM. For comparison, a top-of-the-line desktop PC features a 2.4 GHz processor with 512 MB of RAM. A further concern is that handheld processors typically omit the floating-point unit, so any floating-point operations must be performed in a slower emulation mode. This stark contrast in processing capability and memory size must be taken into account when writing code for handhelds.

Given that our computational capacity is limited, we must be efficient with our code. However, certain stages of our image processing pipeline are both necessary and computationally expensive. We deal with this problem by realizing that the handheld, as an E21, is connected to an Oxygen N21 network, and thus we have the ability to offload certain computations to desktop PCs in the environment. The next challenge is to determine which stages to offload, which we will discuss in Section 4.3.3.

4.2.2 Power Consumption

Though we seem to have a solution for the computational limits of the handheld device, we must consider the consequences of offloading computation. Most notably, power consumption will be an issue, especially if we are using a wireless radio connection to transmit the data. A typical Cisco Aironet wireless card uses 1.7 watts (W) when transmitting, 1.2W when receiving, and 1.1W when idle. This amounts to only a few hours of battery life in typical situations, so we must take power usage into account when determining which data to transmit when offloading computation.

4.2.3 Software Concerns

Though not as limiting as the reduced computation and power concerns, software-related issues pose a problem as well. Many of the computer vision and image processing libraries that are available for the x86 architectures have yet to be built and released for handheld architectures. For instance, Intel's OpenCV computer vision library, which we used extensively

in the PC version of PADCAM, is not available for the StrongARM. Instead, we used Intel's Integrated Performance Primitives library (IPP), but this was not nearly as comprehensive a set of image processing routines.

4.3 Implementation and Setup

As a proof-of-concept, we tested components of PADCAM on a properly outfitted Compaq iPAQ. In this section, we outline the specifications for the hardware, the choices we made for software, and the physical setup of the system.

4.3.1 Hardware Specifications

We used a Compaq iPAQ 3760, equipped with a 206 MHz StrongARM processor, 32 MB of flash ROM, and 64 MB of DRAM. To provide video capture capabilities, we used a BackPAQ expansion pack. This research prototype, developed at Compaq's Cambridge Research Labs (CRL), features a 640x480 CMOS camera, 2 PC card sockets, 32 MB of additional flash storage, an a 2-axis accelerometer sensor. The test setup also included a 256 MB CompactFlash card for additional storage and an 802.11b card for wireless networking.

4.3.2 Operating System

Though the iPAQ ships with Microsoft Windows CE, we found this operating system unsuitable for our development needs. The Familiar Linux distribution, part of the handhelds.org project overseen by Compaq CRL, offers a much more attractive platform for developing video-enabled software for handheld devices. In addition, the open source nature of the underlying code allows us to view and change any aspects of the operating system, which would involve more non-technical barriers using a closed source OS. Another deciding factor in our choice of operating systems was the availability of drivers for the hardware we intended to use. Because the BackPAQ was a recent research prototype, its components only had driver support in the handhelds.org Linux distributions but not in Windows CE.

4.3.3 Adaptation of PADCAM Software for Handhelds

As mentioned earlier, handheld devices have different specifications that PCs, and we redesigned our system taking these into account. Because the most limiting factor is the reduced computational power, our design decisions were mainly driven by the goal of avoiding expensive computation on the iPAQ. However, realizing that network transmissions require both time and power, we also aimed to minimize the network bandwidth required by our system.

Our adaptation of PADCAM to the iPAQ splits the software application into RCAMD, a lightweight component called that runs on the iPAQ, and RPADCAM, the more substantial component that runs on a PC. We call the resulting system RCAMD-RPADCAM (RRP). After experimenting with a few different partitions of computation between the iPAQ and the PC, we decided to have RCAMD capture the video frames, JPEG compress them, and send them over a TCP socket to RPADCAM. RPADCAM then decompresses the frames and performs all of the original PADCAM processing. By compressing the frames, we were able to cut the network usage by a factor of 30 compared to the bandwidth it would have taken to transfer uncompressed 24-bit color 640x480 frames. Also, by keeping the bulk of the software on the PC, we avoided the implementation issues involved in switching architectures (such as being forced to abandon Intel's x86-based OpenCV library).

4.3.4 Physical Setup

Because of the portability of the iPAQ, the physical setup of RRP is remarkably flexible. Figure 4-1 illustrates just one of many possible ways to place the iPAQ in the system. The video frames resulting from a given setup can either be previewed on the iPAQ, as shown in Figure 4-2, or on the PC with the RPADCAM software.

4.4 Experimental Results and Analysis

We tested our software on the iPAQ in several different configurations. We were mainly concerned with the speed of the resulting system, since the accuracy should not change sig-

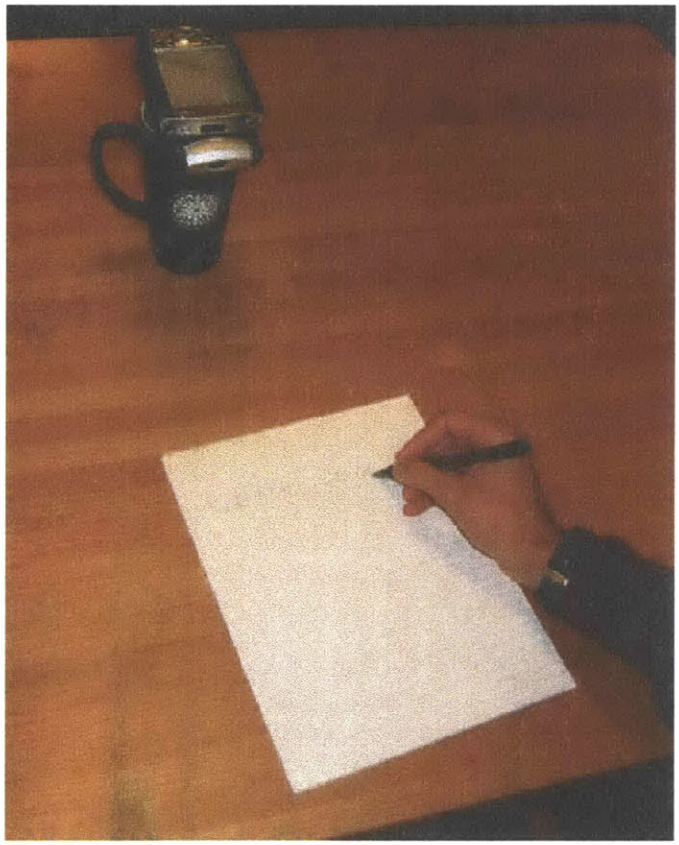

Figure 4-1: RCAMD-RPADCAM on iPAQ Physical Setup

nificantly if we keep the original PADCAM stages. The main parameter we varied was the partition of work between the iPAQ and the PC, but we also experimented with different camera resolutions. The first test was of raw video capture capabilities, listed in Table 4.1. These numbers represent the highest framerates we could achieve by simply grabbing the video frames and not doing any processing except for the JPEG compression and decompression in the case of RCAMD-RPADCAM. Thus, this test is mainly to establish the hardware limits, and we can assume that these results are the upper bounds on framerates we can expect in our final system. Right away we can see that at 640x480 the iPAQ's performance is unusable.

With this baseline established, we can explore what happens when processing is added. We ported the frame difference stage over to the iPAQ for comparison, and our results are listed in Table 4.2 and Figure 4-3. There are two significant points to observe:

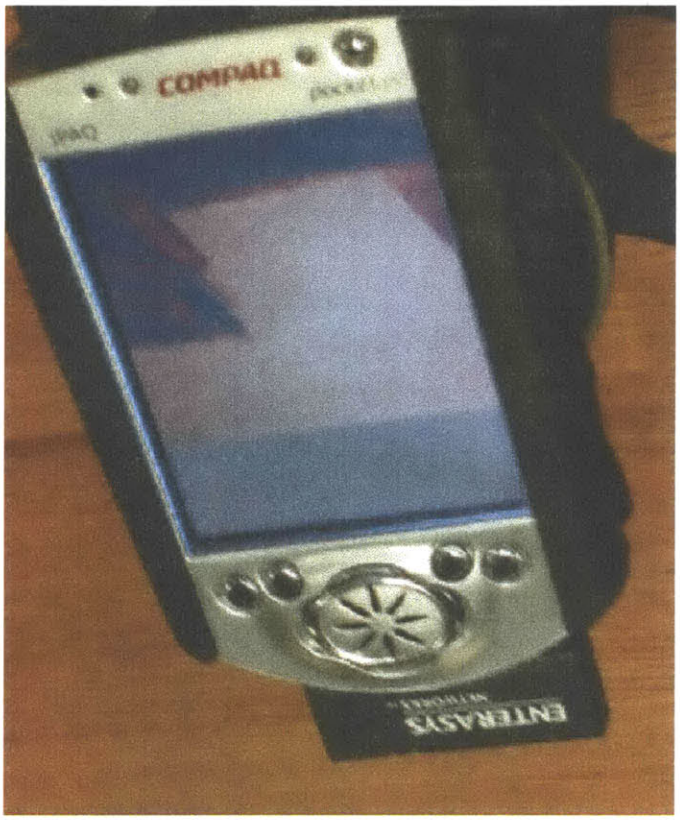

Figure 4-2: RCAMD-RPADCAM Preview of Video Frame on an iPAQ

- Frame Capture on the iPAQ is a nontrivial stage, and capturing high resolution frames is prohibitively costly.
- JPEG Compression is by far the most computationally expensive stage for iPAQs.

Note that frame differencing is probably the simplest of the PADCAM stages, so we can only expect the framerates to decrease as we add more complex stages to the iPAQ implementation. This discouraging fact prompts us figure out why it is that performance is so

Table 4.1: Framerates for Video Capture for Various System Setups (in frames per second)

System	160x120	320x240	640x480	Camera Limit
PADCAM on PC	14.5	14.5	14.5	15.0
PADCAM on iPAQ	30.0	21.2	5.8	30.0
RCAMD-RPADCAM	15.7	6.2	1.2	30.0

Table 4.2: Processing Times for each Stage in Various System Setups

System	Resolution	Processing Step	Time (ms)	Framerate (fps)
PADCAM on PC	160x120	Frame Capture	65	
		Frame Difference	1	
		Miscellaneous	0	
		Total	**66**	**15**
	320x240	Frame Capture	54	
		Frame Difference	8	
		Miscellaneous	3	
		Total	**65**	**15**
	640x480	Frame Capture	47	
		Frame Difference	15	
		Miscellaneous	2	
		Total	**64**	**15**
PADCAM on iPAQ	160x120	Frame Capture	23	
		Frame Difference	12	
		Miscellaneous	5	
		Total	**40**	**23**
	320x240	Frame Capture	32	
		Frame Difference	25	
		Miscellaneous	4	
		Total	**61**	**16**
	640x480	Frame Capture	135	
		Frame Difference	65	
		Miscellaneous	11	
		Total	**211**	**5**
RCAMD-RPADCAM	160x120	Frame Capture	8	
		Frame Difference	10	
		JPEG Compression	35	
		Transmit Frame	2	
		Miscellaneous	2	
		Total	**57**	**17**
	320x240	Frame Capture	31	
		Frame Difference	30	
		JPEG Compression	94	
		Transmit Frame	4	
		Miscellaneous	2	
		Total	**161**	**6**
	640x480	Frame Capture	139	
		Frame Difference	95	
		JPEG Compression	433	
		Transmit Frame	63	
		Miscellaneous	12	
		Total	**742**	**1.3**

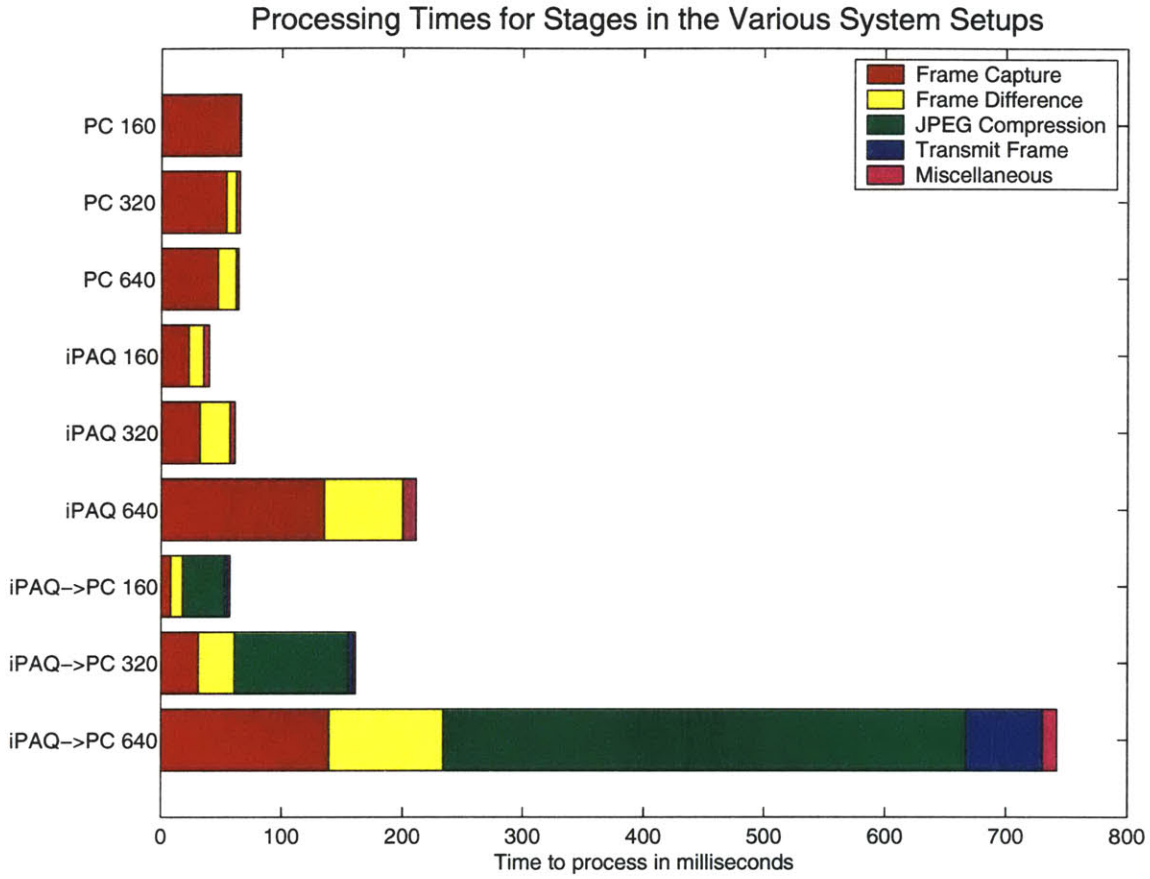

Figure 4-3: Processing Times for each Stage in Various Setups

slow.

Though we expected reduced computation on the iPAQ, we cannot blame the whole performance drop on the slower processor. A second look at the frame capture speeds reveals much of the problem: frame capture basically entails simply transferring data from the imaging device to the main memory, so why does the iPAQ take so long to capture a frame? The answer lies further in the iPAQ's hardware specifications: the expansion bus connecting the BackPAQ to the main processing unit is 16-bits wide and runs at only 8 MHz. The theoretical bandwidth limit for this bus is 16 MB/s. For comparison, a typical PC has a PCI bus that is 32-bits wide and runs at 33 MHz, yielding a theoretical bandwidth of 132 MB/s. Note that in practice, data transfer rates over these buses are a fraction of their theoretical limits; in the iPAQ, we were able to get at most 5.5 MB/s across the bus (640x480 24-bit pixels at 6 fps). Transferring 640x480 video frames with 24-bit color at 30 frames per second would require 27.7 MB/s. Clearly, a major cause of our performance drop is bus saturation.

4.4.1 Floating-Point Results

We also tested the iPAQ's floating-point performance. As we mentioned earlier, the StrongARM processor contains no floating-point unit, so all floating-point operations (FLOPS) must be done in emulation mode. To quantify the performance hit we should expect for FLOPS, we rewrote the frame difference stage using floating-point arithmetic and ran it for the 3 target resolutions, recording both the average time spent in the stage and the overall framerate. The results are shown in Table 4.3. In the extreme case, the iPAQ spent 4.3 *seconds* in the floating-point stage for a single frame. Our task of recovering time-ordered stroke information requires high temporal resolution, thus we cannot afford to spend 4.3 seconds or even 260 milliseconds in any one stage. Clearly, we must avoid FLOPS on the iPAQ at all costs, either by converting to fixed-point operations or offloading the FLOPS to PCs, which have dedicated hardware for such tasks.

Table 4.3: Costs of Floating-Point Operations on PC and iPAQ Systems

System	Resolution	Time (ms)	Framerate (fps)
PADCAM on PC	160x120	0.8	14.6
	320x240	4	14.5
	640x480	14	13.5
PADCAM on iPAQ	160x120	260	3.54
	320x240	1010	0.94
	640x480	4300	0.22

4.5 Discussion

Unfortunately, we found that our software was too computationally demanding for the iPAQ's system architecture. We found that frame capture was a limiting factor in the system, thus trying to sidestep the slower StrongARM processor in favor of offloading computation turned out to be ineffective. We speculate that the problem is rooted in the low bandwidth of the expansion bus, which severely limits all communications in and out of the iPAQ. It would be interesting to attempt the experiments again on a handheld device with more bandwidth. From our results, we observed about practical data transfer rates of about a third of the theoretical limit, so if we extrapolate this further, we would need a bus with a bandwidth limit of 90 MB/s to adequately handle our target capture specification of 640x480 with 24-bit color at 30 frames per second. This ideal bandwidth would be attainable in a handheld device with a 32-bit bus running at 25 MHz. Rather than waiting for such handhelds to become available, we suggest alternate software approaches for circumventing the bus bandwidth problem.

4.6 Suggestions for Future Work

Having revealed the main obstacle preventing our handheld implementation from running at reasonable framerates, we now propose some solutions to the bus bandwidth problem. This is followed by some general suggestions for future work that would be useful for handheld devices.

4.6.1 Solutions to the Bus Bandwidth Limitations

Subregion Capture

One conceptually simple but technically challenging way to solve the bus bandwidth problem is to capture only a subregion of the frames. Currently, the driver and firmware for the BackPAQ camera only provides an API for capturing the entire video frame. If instead we could request only the subregion of the frame that we are interested in, we could dramatically reduce our bus bandwidth requirements. Implementing this solution would entail writing VHDL code to support this feature and reprogramming the BackPAQ's FPGA to reflect the change.

Image Processing in the FPGA

As introduced above, the BackPAQ contains an FPGA that controls its various peripherals. Viewing this FPGA as another source of computing power located very close to the camera, we may be able to offload some of the image processing to it. Doing so might be even faster than performing the computation in the CPU because the bus connecting the FPGA to the camera may have higher bandwidth than the expansion bus. Also, depending on which stages can be implemented in the FPGA, it is possible that the data we need from the BackPAQ may be much smaller than the full 640x480 frame—in the extreme case, it could just be a timestamped point, in which case the expansion bus bandwidth would no longer be a limiting factor. This solution is somewhat speculative, but it may be worth exploring in future research.

4.6.2 Suggestions for New Features

In our experiments with PADCAM on handhelds, we came across features that would be useful in a practical implementation.

Motion Stabilization

One of the most annoying aspects of the iPAQ implementation was the tendency of the iPAQ to move during handwriting capture. These slight movements would result in large shifts in the video frame, which PADCAM did not handle well because it does not continually adapt to its environment. Because of this, we propose to add a motion stabilization feature to the system which would make it robust to movements.

We devised two approaches to implement this feature. The easiest way would be to have page detection run periodically instead of just once in initialization. This way, any significant change in the appearance of the page in the full video frame would result in new corners and accordingly a new perspective transform. A more powerful approach would involve calibrating and registering the camera and maintaining a 3-D model of the page. If there is movement, the system can use 3 or more reference points to update its 3-D model and calculate the change in the perspective transform. This "feature" is powerful enough to warrant its own dedicated system, as it would basically be capable of acquiring 3-D models with minimal setup.

Out-Of-Order Execution

We noticed early on that certain stages in our pipeline could be relocated to allow more efficient partitioning of computation between the iPAQ and the PC. For instance, the perspective transform could be computed at any time as long as it occurs before the timestamped points are returned (or sent to the recognition application). We can take advantage of this flexibility by moving computationally intensive stages, such as those involving floating-point operations, to the PC while keeping simple fixed-point operations on the iPAQ.

4.6.3 Low-Bandwidth Remote Processing

Realizing that frame-to-frame differences often contain much less data than full frames, we propose another tactic to improve efficiency on RCAMD-RPADCAM. Instead of sending each frame across the network to a PC for processing, our tactic entails sending a sparse data structure that stores frame-to-frame differences. By only transferring the changed pixels

as opposed to every single pixel, this approach could reduce the time spent in network communications, freeing up time for more computation.

Chapter 5

Applications

By itself, PADCAM does not offer much of an advantage over the paper and pencil approach to taking notes. As we discussed in Section 2.2.2, we opted against building a recognition system into the system to allow more general content. We found that there is already an abundance of applications that provide meaningful interpretation of mouse input or other stroke information. We can take advantage of this by sending the stroke information obtained by our system to such higher-level applications for meaningful interpretation. Essentially, we can use PADCAM as an input adaptor between a natural interface (writing on paper) and a useful form of data (stroke information) to be used in existing recognition applications. A description of just a few of the possible recognition applications follows.

5.1 Natural Log

Our baseline application was Natural Log (NL), a mathematical expression recognition system. Designed and implemented in Java by Nicholas Matsakis, NL receives input in the form of mouse events and converts these into strokes before performing recognition [20]. Recognition consists of three high-level steps: isolated symbol classification, expression partitioning, and parsing. Though the details of each of these steps are beyond the scope of this thesis, the recognition results in three meaningful interpretations of the expression: an image of the typeset expression, a TEXstring, and a MathML expression. Figure 5-1 shows a screenshot

of the demonstration program.

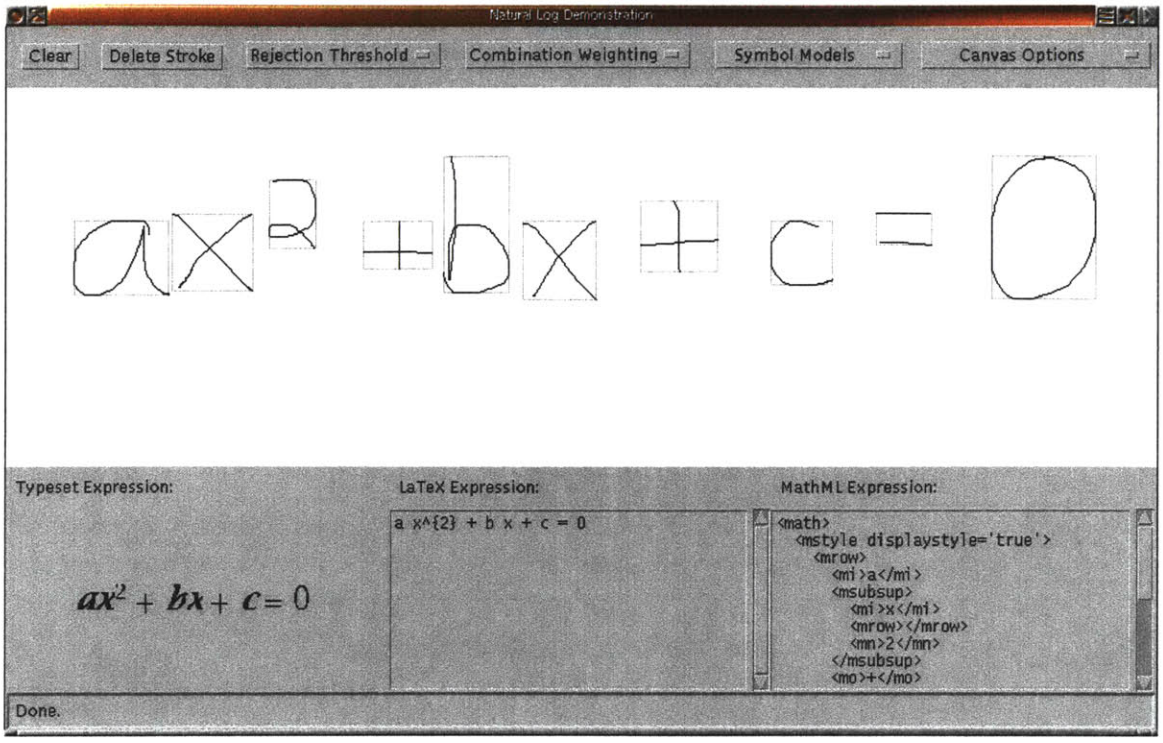

Figure 5-1: The Natural Log Demonstration Program. As the user writes symbols in the white canvas, the system recognizes the mathematical expression and displays an image of the typeset expression, a string specifying how to format it in TEX, and a MathML expression.

Adapting NL to accept input from PADCAM was simple and straightforward. Rather than feeding fake mouse events into NL, we decided to send the stroke data at a lower level of abstraction. Minor modifications to the NL Java application allowed it to accept strokes as time-ordered points over a TCP socket connection. In the resulting combined system, PADCAM sends a stroke as soon as it is finished to NL, which recognizes the mathematical symbol represented. As more strokes are sent, NL updates the resulting mathematical expression accordingly.

The NL-PADCAM combination provides an easy way to capture mathematical expressions quickly and naturally. Simple expressions were recognized correctly. The combined system would benefit from better ink detection on the PADCAM side—often strokes were not properly segmented before being sent to NL, which led to recognition errors. Despite these issues, our initial implementation of NL-PADCAM yielded encouraging results.

5.2 Sketchpad

Another suite of applications aimed at interpreting meaning from stroke information comes from the Sketchpad project at the MIT AI Laboratory. One of the main goals of this project is to develop perceptual interfaces that allow natural human input such as sketches. To demonstrate this idea, Davis and his research group have developed software applications to recognize mechanical drawings and software architecture sketches [9].

5.2.1 ASSIST

The initial prototype demonstrating the Sketchpad team's vision is ASSIST, or A Shrewd Sketch Interpretation and Simulation Tool. This tool allows an engineer to sketch a mechanical system using a pen-based input device, such as a digitizer, and then allows for interaction with this design [3]. For instance, from a drawing of a simple car on a ramp, ASSIST can simulate the motion in the system based on kinematics, as illustrated in Figure 5-2.

5.2.2 Tahuti

Another application that brings sketch-based input and recognition closer to reality is Tahuti. This sketch recognition environment interprets pen-based input of Unified Modeling Language (UML) diagrams, a commonly used method for describing the architecture of software applications [14].

Both ASSIST and Tahuti expect stroke information as input, and thus would make good targets PADCAM's output.

5.3 Handwriting Recognition

Perhaps the most straightforward application to consider for PADCAM is for handwriting recognition. While some efforts have been made to design handwriting recognition systems specifically for video-based interfaces [13], it is both convenient and desirable to use existing pen-based input recognition systems. Many such systems have been demonstrated to work

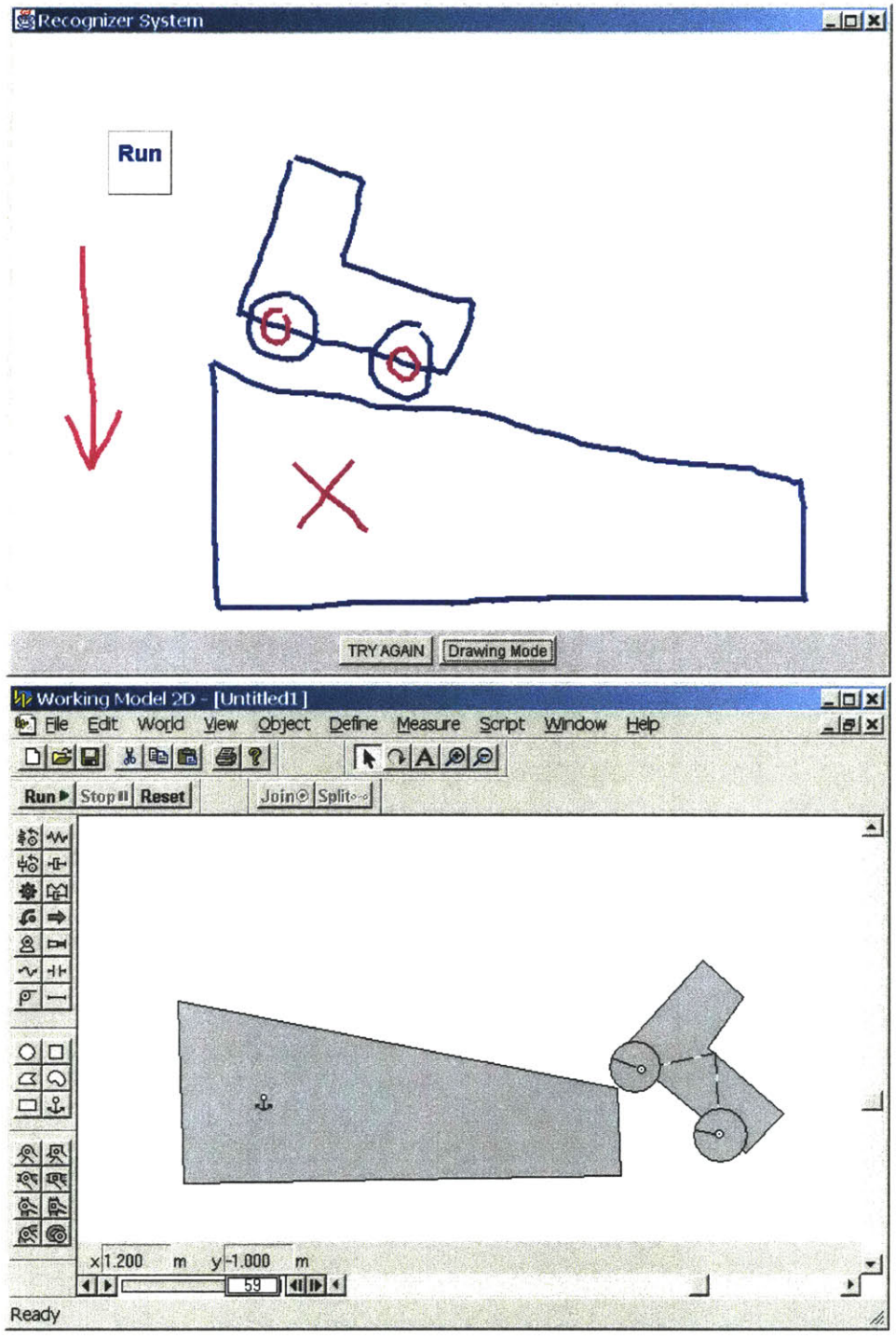

Figure 5-2: An Example Use of ASSIST. The user sketches a car, a ramp, and certain other symbols that indicate mechanical properties. ASSIST recognizes the sketched objects and performs a simulation of the kinematics in this mechanical system.

well [7, 6, 19, 18], and it makes sense to leverage these existing software applications to make PADCAM more valuable.

In our exposure to handheld devices, we became familiar with xstroke, an open source version of the popular Graffiti text input system used on PalmPilots. This program recognizes handwriting gestures based on each stroke's trajectory through a virtual 3x3 grid spanning the stroke's bounding box. We propose that this software would be easy to adapt for use with PADCAM, thus providing a quick and robust solution to character recognition.

Chapter 6

Discussion

Having proposed and evaluated our system for handwriting capture, we now take a moment to recommend future directions to take this research and then pause to draw some conclusions.

6.1 Recommendations for Future Work

In addition to the suggestions we made in Section 4.6 for improving the handheld implementation, we have recognized many ideas that would extend the original PADCAM system. Our future work is clearly divided into two distinct families of ideas: performance improvements and new features.

6.1.1 Performance Improvements

Switch from USB to FireWire

We would advise a transition from USB to FireWire cameras, because USB is too limiting to allow 640x480 frames at 30 fps. This would require nontrivial changes, since our implementation is based on the video4linux interface rather than one of the existing FireWire interfaces. While this change would allow us to double our capture framerate from 15 to 30 fps, it would mean that the processing would have even less time in between frames to get done.

Fix Backward Differencing

As discussed in Section 3.4.2, neither of our frame differencing techniques yielded usable results. We speculate that this was due to transient shadows that affect the ink trace, but it would be interesting to test this claim and propose a solution.

6.1.2 New Features

The most exciting ideas fall in the category of new features that add to the utility of the system.

Subpixel Resolution

Though camera jitter ruined video sequences every now and then, the annoying effect led us to propose a solution that can be generalized to solve other related problems. As described earlier, there are two clear solutions: one involving periodic page detection and the other involving 3-D modeling. We will leave the details of the former to Section 4.6.2, but the latter is interesting because it allows so much more than simple motion stabilization. Not only could we acquire 3-D models with this setup, but also we could make intelligent use of camera jitter to improve resolution to subpixel levels. This also opens the door for possibilities such as an environment with multiple unregistered cameras with which one can mosaic a large sketch surface or whiteboard.

Whiteboards

In the future, we plan to generalize PADCAM for use with whiteboards. This should only require minor modifications to the video capture module. This feature would be useful in meeting situations where the minutes of the meeting are to be recorded and made available to all participants.

Multimodal Uses

Also, adding a speech recognition module to provide context to the handwriting recognition module may increase accuracy. Such an architecture would take advantage of multimodal

input to provide better guess as to what is being written. In fact, the information exchange between the handwriting and speech recognition modules could be in either direction (or both), depending on which recognition task requires the most accuracy.

6.2 Conclusions

We proposed and demonstrated a perceptual interface for pen-based input that captures live video of handwriting and recovers the time-ordered sequence of strokes that were written. By experimenting with frame differencing, pen tracking, and ink detection, we were able to construct a working prototype with good results. We also motivated and presented an implementation of PADCAM for handheld devices, focusing on the challenges and the future work to be done on this front. Finally we discussed applications and future directions for this research. It is clear that this system is a useful interface, and our work has spawned an abundance of ideas that will help develop this prototype system into a practical method of pen-based input.

Bibliography

[1] http://www.anoto.com.

[2] http://www.e-beam.com.

[3] C. Alvarado. A natural sketching environment: Bringing the computer into early stages of mechanical design. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2000.

[4] H. Bunke, T. Von Siebenthal, T. Yamasaki, and M. Schenkel. Online handwriting data acquisition using a video camera. *Proceedings of the Fifth International Conference on Document Analysis and Recognition*, pages 573–576, 1999.

[5] J. Canny. A computational approach to edge detection. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986.

[6] K. Chan and D. Yeung. Elastic structural matching for on-line handwritten alphanumeric character recognition, 1998.

[7] S. Connell and A. Jain. Template-based online character recognition.

[8] J.W. Davis and A.F. Bobick. The recognition of human movement using temporal templates. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 23(3), 2001.

[9] R. Davis. Position statement and overview: Sketch recognition at mit. *AAAI Spring Symposium on Sketch Recognition*, 2002.

[10] D.S. Doermann. *Document Image Understanding: Integrating Recovery and Interpretation.* PhD thesis, University of Maryland, College Park, 1993.

[11] D.S. Doermann and A. Rosenfeld. Recovery of temporal information from static images of handwriting. *Proceedings of IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 162–168, 1992.

[12] D.S. Doermann and A. Rosenfeld. Recovery of temporal information from static images of handwriting. *International Journal of Computer Vision*, pages 143–164, 1995.

[13] G.A. Fink, M. Wienecke, and G. Sagerer. Video-based on-line handwriting recognition. *Proceedings of the Sixth International Conference on Document Analysis and Recognition*, pages 226–230, 2001.

[14] T. Hammond and R. Davis. Tahuti: A geometrical sketch recognition system for uml class diagrams. *AAAI Spring Symposium on Sketch Understanding*, 2002.

[15] R.E. Kalman. A new approach to linear filtering and prediction problems. *Trans. of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[16] Y. Kato and M. Yasuhara. Recovery of drawing order from single-stroke handwriting images. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 22(9):938–949, 2000.

[17] P.M. Lallican, C. Viard-Gaudin, and S. Knerr. From off-line to on-line handwriting recognition. *Proceedings of the Seventh International Workshop on Frontiers in Handwriting Recognition*, pages 303–312, 2000.

[18] X. Li, R. Plamondon, and M. Parizeau. Model-based on-line handwritten digit recognition, 1998.

[19] Xiaolin Li and Dit-Yan Yeung. On-line handwritten alphanumeric character recognition using feature sequences. In *ICSC*, pages 197–204, 1995.

[20] N. E. Matsakis. Recognition of handwritten mathematical expressions. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1999.

[21] M.E. Munich. *Visual Input for Pen-Based Computers.* PhD thesis, California Institute of Technology, 2000.

[22] M.E. Munich and P. Perona. Visual input for pen-based computers. *IEEE Proceedings of the 13th International Conference on Pattern Recognition*, 3:33–37, 1996.

[23] M.E. Munich and P. Perona. Visual input for pen based computers. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 24(3):313–328, 2002.

[24] M.E. Munich and P. Perona. Apparatus and method for tracking handwriting from visual input. *US Patent 6,044,165*, filed 6/15/1995, granted 3/28/2000.

[25] Larry Rudolph. Project Oxygen: Pervasive, Human-Centric Computing - An Initial Experience. In Klaus R. Dittrich, Andreas Geppert, and Moira C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 2068 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2001.

[26] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[27] Emanuele Trucco and Alessandro Verri. *Introductory Techniques for 3-D Computer Vision.* Prentice Hall, Inc., 1998.

[28] T. Yamasaki and T. Hattori. A new data tablet system for handwriting characters and drawing based on the image processing. *IEEE International Conference on Systems, Man, and Cybernetics*, 1:428–431, 1996.

[29] Junko Yoshida. Vendors race to pack cameras into cell phones. *EETimes*, 8 September 2000.