EUROGRAPHICS Symposium on Sketch-Based Interfaces and Modeling (2012)
L. B. Kara and K. Singh (Editors)

Mobile Vision-Based Sketch Recognition with SPARK

Jeffrey Browne and Timothy Sherwood

{jbrowne, sherwood}@cs.ucsb.edu
University of California, Santa Barbara

Abstract

The sketch community has, over the past years, developed a powerful arsenal of recognition capabilities and interaction methods. Unfortunately, many people who could benefit from these systems lack pen capture hardware and are stuck drawing diagrams on traditional surfaces like paper or whiteboards.
In this paper we explore bringing the benefits of sketch capture and recognition to traditional surfaces through a common smart-phone with the Sketch Practically Anywhere Recognition Kit *(SPARK), a framework for building mobile, image-based sketch recognition applications. Naturally, there are several challenges that come with recognizing hand-drawn diagrams from a single image. Image processing techniques are needed to isolate marks from the background surface due to variations in lighting and surface wear. Further, since static images contain no notion of how the original diagram was drawn, we employ bitmap thinning and stroke tracing to transform the ink into the abstraction of strokes commonly used by modern sketch recognition algorithms. Since the timing data between points in each stroke are not present, recognition must remain robust to variability in both perceived drawing speed and even coarse ordering between points. We have evaluated Rubine's recognizer in an effort to quantify the impact of timing information on recognition, and our results show that accuracy can remain consistent in spite of artificially traced stroke data. As evidence of our techniques, we have implemented a mobile app in SPARK that captures images of Turing machine diagrams drawn on paper, a whiteboard, or even a chalkboard, and through sketch recognition techniques, allows users to simulate the recognized Turing machine on their phones.*

Categories and Subject Descriptors (according to ACM CCS): I.7.5 [Document and Text Processing]: Document Capture—Graphics Recognition and Interpretation I.3.3 [Computer Graphics]: Picture/Image Generation—Digitizing and Scanning

1. Introduction

While sketch research has demonstrated the impressive potential of pen based interfaces across a wide array of application domains, devices appropriate for deep sketch interaction are still quite rare. Touch systems are getting closer to the required functionality, but the size of the displays and the large impressions required by capacitive touch sensors make it difficult to create large or intricate diagrams. While it is possible to create a true digital whiteboard experience through a combination of pen capture systems like LiveBoard or eBeam, such set-ups are still not ubiquitous because they require space, money, and calibration time. The reality is that most sketching today still happens on traditional whiteboards, paper, and even chalkboards.

Rather than requiring users to move their work to devices appropriate for pen capture, the goal of this work is to examine bringing the power of sketch recognition to existing drawing surfaces by use of the modern smart-phone. The high quality of images provided by the cameras on even entry-level devices means that diagrams drawn on whiteboards and paper can be precisely captured after they are drawn, in theory allowing users to interact with aspects of their drawings (or models derived from them) directly on their phones. While this capture-then-recognize model certainly restricts the set of interactions that are possible, it does cover some interesting and important cases – for example the users who wish to operate on systems represented by a complex graphical notation such as a chemical formula or a mathematical expression.

In this work, we present the *Sketch Practically Anywhere Recognition Kit* (SPARK), a framework for developing mobile, image-based sketch recognition apps, as well as our ex-

DOI: 10.2312/SBM/SBM12/087-096

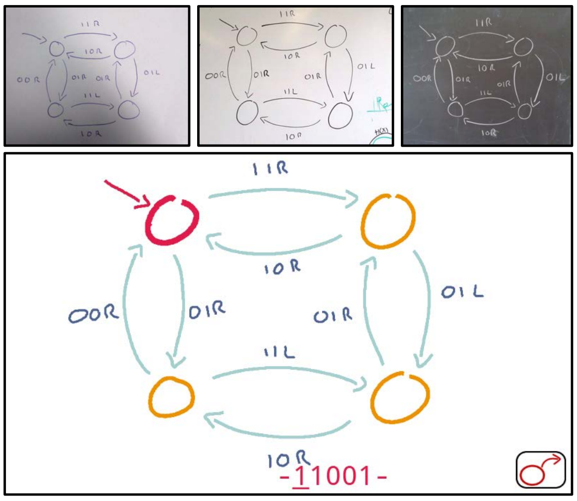


Figure 1: Apps within SPARK use sketch-recognition techniques on strokes extracted from photographs, enabling users to simulate Turing machine diagrams drawn on surfaces such as whiteboards, paper, and even chalkboards.

periences developing a Turing machine simulator within the framework (figure 1). The result is a full system that allows users to capture images of Turing machine diagrams drawn on paper, whiteboards, or even chalkboards, and to simulate them step-by-step on a mobile device. The system extracts stroke information from the captured ink in order to employ known sketch recognition techniques to this new domain.

Of course there are several challenges to overcome in undertaking such an endeavor. Despite the ever-improving capabilities of modern smart-phone cameras, there still exists noise from digitizing the image, as well as the issues of shadow, reflection, skew, and clutter. Even after the drawing area is isolated and normalized, current diagram recognition methods typically require sketch information presented using a model of separate strokes, each made up of a list of timestamped coordinate points. However, static image data contains no temporal information to determine how and when strokes are drawn, so we can only partly satisfy the standard sketch abstraction. We necessarily lose even coarse stroke timing information, so whatever sketch recognition techniques we use must not be overly sensitive to metrics like drawing speed or point ordering.

To explore these issues in an end-to-end way, we have developed the Turing machine app within SPARK as an example of the general architecture for this new kind of sketch recognition application. Through novel modifications to known image processing techniques, our framework can reliably capture hand-drawn marks and convert them to a form more readily handled by sketch processing systems. We describe the problems encountered when attempting to apply standard techniques, such as thinning, to the specific problem of stroke extraction and how, by making use of information about stroke thickness and common intersection patterns, we can overcome these limitations. We further show that timing information is not as critical to Rubine's gesture classifier for basic symbol recognition as may have been previously assumed, and we evaluate its accuracy under the new constraints presented by image-based sketch recognition. We find that, at least for simple, single-stroke symbols, recognition accuracy is minimally affected by the arbitrary ordering of points within- and between strokes. Finally we describe the overall software architecture of the app and how we bring both recognition and interaction to traditionally passive drawings.

The rest of the paper is structured as follows. In Section 2, we discuss previous work in capturing, tracing, and recognizing hand-drawn ink. In Section 3 we explain the usage and overall architecture of SPARK. We provide detailed explanation of our stroke extraction method in Section 4, while we describe the diagram recognition procedure and evaluate the performance of Rubine's classifier in Section 5. In Section 6 we explore the general consequences of adapting common sketch-based interfaces to image-based stroke capture methods and finally conclude with Section 7.

2. Related Work

At a high level, related projects that augment real ink with computation can be broken down according to their varying levels of recognition. Applications like Tableau [OL10] or He and Zhang's whiteboard capture system [HLZ02] have as a common goal cleaning up images of whiteboards or other handwritten content in order to archive, for example, meeting notes or other board work. These systems must employ image processing techniques such as board localization (finding the board in a cluttered image), adaptive thresholding for reflection and shadow filtering, skew adjustment, and alignment between multiple photos to produce their desired output. Zhang's whiteboard scanner, for example, stitches together a single panoramic image of an entire board's contents from multiple images, despite variations in angle, shadowing, and board location in each image [ZH07]. In general, systems in this class clean up board images so that they can be easily stored and shared in digital form, but they do not typically perform any recognition of the drawing region's contents.

More similar to our aim of mobile sketch recogntion from images of diagrams is work by Farrugia et al. [FBC*04]. Users of their system draw a specification on paper that describes a 3D "rotational component," which is then recognized from a cameraphone photograph, and the user is presented with a rendered version of the component. While the general interaction method of photographing real ink is shared between our system and theirs, the authors only discuss paper as an input surface, where we aim to recognize on a wider variety of surfaces with different background and ink characteristics. Further, while their image processing steps follow a similar binarization/skeletonization preprocessing step, they do not convert the thinned bitmaps to strokes, which our framework requires to leverage stroke-based sketch recognition algorithms. Instead, their Generalized Randomized Hough Transforms (GRHT) symbol recognizer can operate over raw bitmap structures, and as such they omit the conversion step.

Most commonly, applications that produce cleaned up versions of photographs and perform content recognition target the problem of optical character recognition (OCR) over typeset text. Specifically in the mobile sphere, OCRdroid [ZJK*] and Google Goggles [Goo11], allow users to extract text from printed documents and labels using the camera on a mobile phone. When operating on typeset text, OCR algorithms can leverage regularity in the characters that they recognize, and recognition is largely an issue of adequately compensating for skew, rotation, and other variation introduced by the scanning process before extracting vision-based features to classifying individual characters [Eik93]. Typeset diagram recognition, such as the ReVision [SKC*11] system, can also exploit precision inherent in automatically generated charts, but recognition techniques will vary with the type of chart that is targeted.

Recognizing handwriting from image data presents unique challenges not present in typeset text recognition, though systems such as that used by the US postal service to parse envelope addresses have shown great practical success [PS00]. Separate projects from Vajda [VPF09], Liwicki [LB09], and Schenk [SLR08] have addressed handwriting recognition from images of whiteboards. This domain presents different challenges than that of smaller-scale, paper-sized handwriting recognition, since assumptions about consistent slant and baselines no longer hold when sentences can span an arm's length or more [WFS03]. Different from printed characters, handwriting (especially from cursive) must often be segmented into individual characters using subtle algorithms before recognition can even take place. Further, since characters can vary in the thickness with which they are drawn, straight bitmap comparison can be unreliable, and hand drawn ink must first be converted to a common form before analysis. Commonly this is done through the use of bitmap thinning (skeletonization) algorithms, which take as input a solid region of pixels from a bitmap, and produce a single-pixel wide approximation of the medial axis for that shape [LLS92]. Depending on the thinning methodology, certain errors can be introduced into the final result. Work by Kato [KY99] showed that, in some applications, misaligned intersections can be corrected by utilizing the average width of a stroke in a global analysis phase, and clustering branch pixels in the identified "S-line" regions. Our method exploits a similar notion of stroke thickness to filter spurious edges and to correct misaligned intersections, but since the ink captured on whiteboards and chalkboards can vary in thickness even within a stroke, we compute thickness on a per-point basis to guide our correction process, rather than for each stroke as a whole.

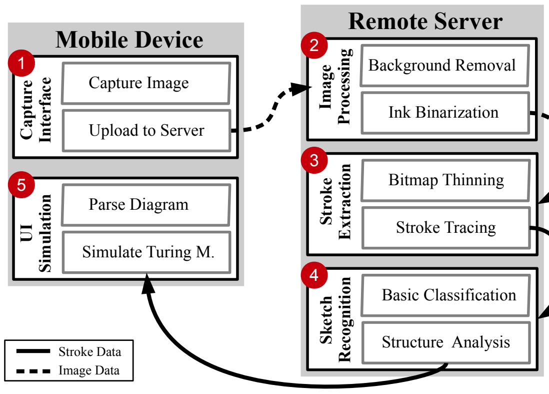


Figure 2: The system architecture for our Turing machine app built within the SPARK framework. Photos taken on the mobile phone (1) are processed by the remote server, which isolates ink in the original image (2), extracts stroke information from the ink (3), and performs recognition on the generated strokes (4). The final semantic meaning of the diagram is sent back to the phone, where a user can simulate the Turing machine (5).

While not always necessary, specific applications of image-based handwriting recognition require some notion of temporal data within pen strokes. For instance, forensic signature verification often requires a directed path through the ink points to produce worthwhile results. In these techniques, generating plausible path information is a matter of traversing an adjacency graph for the ink pixels, and previous works have explored driving traversal using probabilistic techniques, such as hidden Markov models [NdPH05], as well as static rules derived from assumptions about the number of strokes and their intersections [KY99]. These systems employ advanced path tracing techniques because they must be accurate for a small amount of ink data in a very specific context. In order to recognize a wider set of diagram domains, we wish to leverage general sketch recognition algorithms, which tend to represent diagrams as a sequence of strokes. Like a signature, the notion of a stroke requires path information, but without domain-specific assumptions about our multi-stroke, hand-drawn diagrams, our method provides only a coarse approximation of stroke timing information. We show, however, that our current tracing technique does not greatly impact final recognition accuracy.

3. System Description

Driving our work in ubiquitous surface, image-based sketch recognition is a Turing machine simulator built with the SPARK framework that enables people to simulate Turing machines by drawing a diagram and taking its picture with a

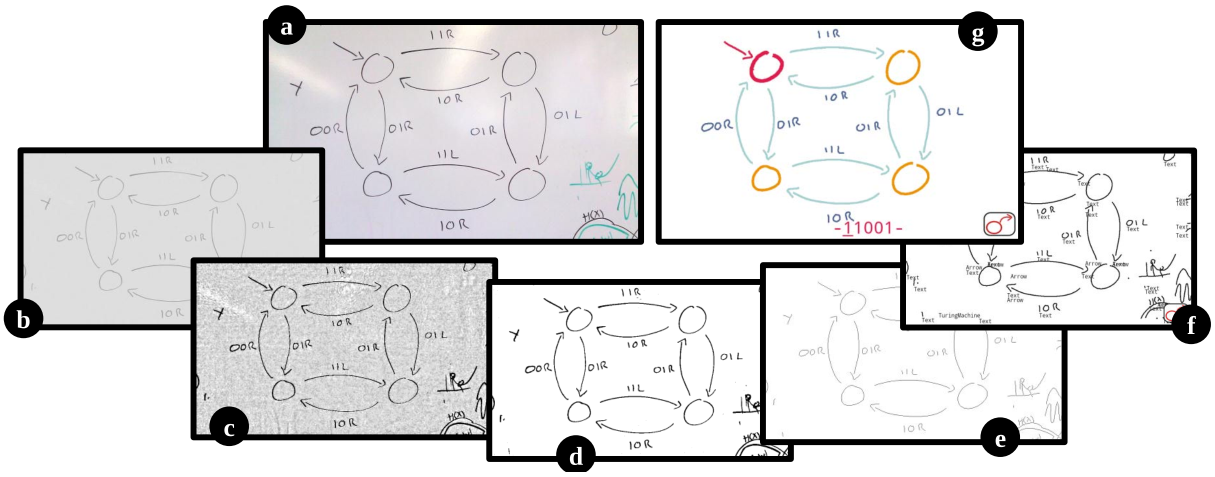


Figure 3: The overall process of recognizing sketched ink from an image of a whiteboard. The raw image (a) is processed to remove the background shadows and reflections (b), and is contrast boosted (c) before binarization (d). Strokes are thinned and traced (e) and then submitted for basic glyph recognition (f), and finally assembled into a Turing machine and displayed on the phone(g)

mobile phone. Students or researchers in computation theory can easily draw Turing machines, but traditional methods of reasoning step-by-step and manually tracking changes to the I/O tape are tedious book keeping tasks well-suited to automated simulation. A person using the app draws a Turing machine on a whiteboard, piece of paper or chalkboard using normal pen, marker, or chalk implements. Once the user reaches a point where she wishes to analyze her work, she photographs the relevant diagram portion, and a picture is sent to a recognition server. In a matter of seconds, the server returns a Turing machine recognized from the photo, which is displayed in "Simulation" mode on the device.

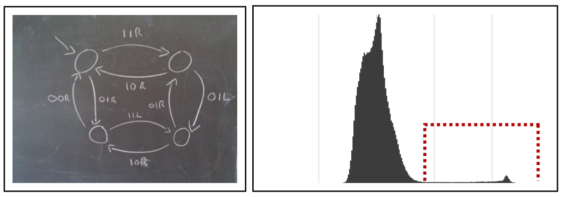
As can be seen in figure 1, when a recognized diagram is loaded into the simulation view, the initial state is "active" (colored red), and the user can begin stepping through the operation of the Turing machine straight away, with the I/O tape displayed below the diagram updating upon each press of the step button, and the states gaining or losing highlight depending on their active status. Through a menu system, she can define a different tape string to be placed on the tape for simulation, or save the recognized Turing machine for later examination. Upon the Turing machine halting (given no valid out edge from the current state), the colors are dulled and stepping the simulation has no effect.

Our app recognizes Turing machines as represented by state machine-like diagrams (shown in figure 1). In these diagrams, the machine's logic is represented as a set of states with transition edges denoted by triplets (R, W, δ), where $R \in A = \{“0”, “1”, “\text{-}”\}$ is the character on the tape currently under the read/write head, $W \in A$ is the character to write in the space when taking the transition, and $\delta \in \{“L”, “R”, “\text{-}”\}$ (short for "left", "right" and "stay put") is the direction the read/write head moves upon taking the transition.

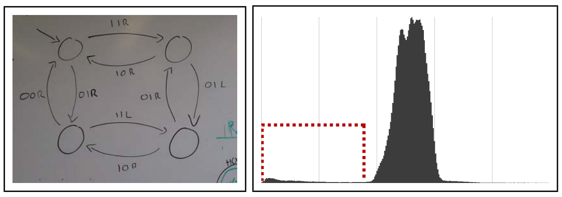
Since Turing machine diagrams do not require a tight interaction loop for useful simulation (it is reasonable to expect that users will draw for a while before invoking recognition), they serve as a good model for a class of sketch applications that we believe are both amenable to the draw-then-interact paradigm as well as provide useful functionality. Further, recognizing them brings application requirements common to modern sketch recognition interfaces. Interpreting these diagrams involves accurately classifying several different alphabet symbols (letters and symbols that label the transition edges), which users can be expected to draw with relatively consistent structure. At the same time, recognition must also handle structures that cannot be statically matched according to their overall shape, namely arrows whose tail strokes can trace any path between nodes. Finally, a diagram's meaning is derived from the structure of recognized primitives. For example, a directed graph consists of nodes with arrows that "point towards" them.

3.1. Architecture Overview

SPARK apps are implemented as applications on the Android mobile platform that communicate with a back end server written in Python 2.7 with OpenCV extensions. To simulate a Turing machine, the user will draw a diagram on a whiteboard, chalkboard or piece of paper, and then start the app on her phone. The initial interface is a live preview display of the camera's view and a camera button that, when pressed, invokes the device to capture an image and upload it to the remote recognition server. As shown in figures 2 and 3,



(a)



(b)

Figure 4: The long tails in the value histogram associated with the light (4a) versus dark (4b) ink better distinguish surfaces than the median value.

when the server receives an image from the phone, it proceeds along three major steps: Image processing, Stroke extraction, and Sketch recognition.

Image processing performs background subtraction, contrast boosting, and thresholding on the image to generate a black and white image of the ink. Stroke extraction then operates on the black ink areas by first thinning them down to single-pixel wide skeletons and then tracing paths over the unordered points to form strokes. Next, the collection of strokes extracted from the image passes to our recognition framework, where they are first matched against a low-level symbol alphabet using a Rubine classifier before being assembled into a description of a Turing machine. Finally, the strokes, along with a semantic description of the board's diagram structure, are sent to the phone, where the Turing machine is displayed, and simulated by the user.

4. Extracting Strokes from an Image

When a user takes a photograph of her diagram, we cannot expect uniform lighting conditions across the medium. Shadows, reflections, and smudges from unclean boards can turn what should be an evenly colored surface like a whiteboard or chalkboard into an image where static threshold values for ink and background separation will not suffice. Once a user's photograph is received by the server, the framework performs several image processing steps before attempting to extract stroke data that will be recognized by a specific app.

4.1. Image Normalization

After an initial step of converting the image to grayscale, we must first determine the medium on which the diagram is drawn from the value histogram of the image, whether dark ink on a light background, like on paper or a whiteboard, or the opposite, as in the case of a chalkboard. While the median pixel value of the image is not a reliable metric for separating blackboards from whiteboards (neither is distinctly black or white, but rather more of a gray), our system exploits the tail of the histogram, where values correspond to the ink as shown in figure 4. If there are more pixels significantly darker than the median than those that are lighter, we assume the ink is dark, and vice versa for ink that is lighter than the background. In the latter case, we invert the image, so our algorithms can proceed assuming dark ink.

Since surface wear or uneven lighting from reflection and shadow introduce background variation, our algorithm performs a modification of Niblack's adaptive thresholding [Nib86] that also enhances image contrast. To produce a signature image of the background, we perform a median filtering with increasingly large kernel sizes until there is only one local maximum in its value histogram, corresponding to the board's median value (meaning all ink is removed from the image). The background signature image is then subtracted from the grayscale image. A side effect of compensating for the background value is that ink pixels stand out less dramatically than before, so we then enhance image contrast in order to better separate ink from the board, before finally binarizing the image with an emperically chosen threshold value. Though contrast boosting can increase the potential for noise artifacts, in practice we have found the extra step to be worthwhile; as further discussed in Section 5, our recognition logic filters out extra strokes better than it compensates for missing strokes, as can occur if ink blends into the background. However, further evaluation is necessary to quantify the impact of noise for general sketch recognition applications.

4.2. Bitmap Thinning

After image normalization, user ink can be reliably separated from the background of a photo, but it is represented as regions of solid black against a white background, with potentially dozens of pixels corresponding to each "point" in the logical stroke that they represent. As a first step in transforming the bitmap of ink regions into strokes, we must determine the logical coordinates of each point in a path. The system accomplishes this by thinning the broad pixel regions into single-pixel wide lines that approximately follow the center axis of each stroke.

We utilize established thinning techniques [NS84, LLS92], modified to take into account the local stroke thickness at each point for improved results. Pixels that provide only redundant connections between black regions can be removed without globally disconnecting any neighboring pixels, and iterative scans of the image with a 3x3 window will find redundant pixels by setting a maximum and minimum

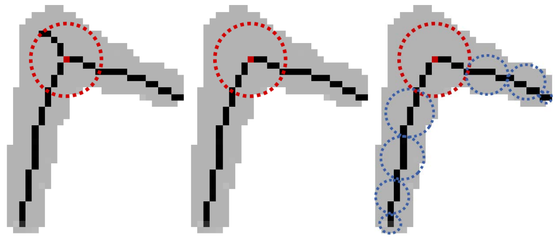


Figure 5: An illustration of the spurious edge artifacts from basic thinning. The true stroke can be iteratively merged by assuming intersections are in the stroke and iteratively connecting points not already covered by the thickness of the current true stroke.

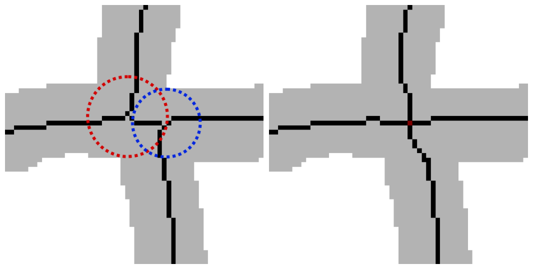


Figure 6: An illustration of the split intersections due to thinning. If the regions defined by stroke thickness surrounding two intersections overlap, they are collapsed into one.

threshold on the center pixel's "crossing number" (the count of transitions from black to white while circling it).

While this technique is common for turning thick bitmaps into narrower skeletons, iteratively making decisions local to 9-pixel regions has a tendency to magnify early errors. Shown in figure 5, a "spurious edge" error occurs when a pixel mistakenly forms what looks like a legitimate endpoint (one with only a single neighboring dark pixel) early in the thinning process. The pixel is then retained for subsequent thinning iterations, and the error compounds since other pixels must be retained to avoid disconnecting the initial point from the rest of the thinned stroke.

Spurious edges can be eliminated if their entire length falls within the thickness of another point on the stroke, which we examine while building the adjacency graph. As shown in figure 5c, our graph building algorithm first identifies intersection points as part of the final graph of correct points. Then, it iteratively merges any point that falls outside of the total thickness region covered by the current graph (along with the points that needed to connect it to a node in the graph) until all thinned points are covered by the cumulative thickness region.

Intersections in a drawing can also be corrupted by the traditional thinning process. Thinning is a form of intelligent erosion, where regions of a bitmap are evenly shrunk from all directions until the bitmap is a single pixel wide, passing ideally through the center of mass of the pixels. This works well for the length of a single lines, but when two lines meet, the centroid is often a poor descriptor of the intended intersection point. As illustrated in figure 6, intersections that should share a single point can be split into multiple segments in the final result.

To address this shortcoming, our modified algorithm examines a circular region with a radius of the stroke's thickness around every intersection in the adjacency graph, where the thinning results are reexamined. If multiple intersections overlap according to their thickness regions, then the algorithm creates a point with their average coordinates to use as a single replacement point.

4.3. Stroke Tracing

At the conclusion of the thinning process, strokes are represented by an adjacency graph with spurious edges removed, and more accurate intersection points. An adjacency graph can have arbitrary connections, though, and we wish to produce ordered lists of points that form strokes, so we must group points into associated strokes, and impose an ordering between them.

Since points in a stroke have at most two neighbors (points before and after in the path), we must decide the direction any stroke takes at an intersection. Our tracing algorithm begins by finding all intersection points within the graph (pixels with more than two neighbors), and links together pairs of points whose joining maximizes smoothness, computed as the median curvature over multiple point resolution scales.

Our tracing algorithm then repeatedly selects some pixel in the graph that does not yet belong to a stroke, and traverses the graph until it finds an end point (a pixel with a single neighbor), or it covers the entire graph (meaning the figure is cyclical). This endpoint (or the initial seed point) is chosen as the starting point of a new stroke, from which we can traverse the graph, extending the stroke with each pixel we encounter until another endpoint (or the initial seed point) is reached. Tracing is complete when all points in the graph are part of a stroke, at which point the entire collection is passed to the recognition phase.

5. Recognition over Extracted Strokes

The process of extracting strokes from an image of ink introduces unique challenges if we are to leverage sketch recognition techniques to understand the meaning of a diagram.

Feature	**Domain(s)**
Cosine from first to last point	Shape, Graph
Sine from first to last point	Graph
Dist. from 1st to last point	Shape, Graph, Class
Dist. from 1st to last / bounding box size	Shape,
Minimum drawing speed	Class

Table 1: Classification features determined to be timing-dependent. These features were disabled when generating results for R^- and R_t^-

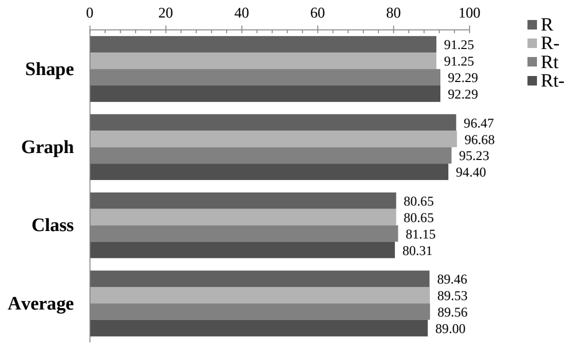


Figure 7: Accuracy of the classifier when training and classifying on traced strokes as a factor of accuracy when using actual stroke data as captured, with all features enabled.

In this section, we examine possible implications for adapting known recognition techniques given two differences between strokes resulting from this process and those captured directly from a digitizing pen: the lack of timing information and potential for extraneous "noise" strokes.

5.1. Impact of Stroke Timing on Recognition

If the thinning process works perfectly, strokes extracted from the image have the same geometric properties as if they were captured from a digitizing pen and tablet, but by the nature of generating strokes from a single image, even coarse-grain timing characteristics (for example, which points are "first" and "last") are necessarily lost. Stroke-based sketch recognition algorithms rely on this timing information to varying degrees, and it serves to quantify the impact of artificially imposed timing on recognition accuracy, so a system designer can decide whether or not key algorithms should be re-engineered to work independent of timing data.

Following the work of Blagojevic et al. [BCP10], we implemented a Rubine classifier [Rub91] that uses domain-specific sets of features over three domains of user drawings: basic shapes (Shape), directed graphs (Graph), and object-oriented class specification diagrams (Class). While the stroke data sets were provided by the authors, we implemented our own classifier and features (due to time constraints feature 4.1, "The result of the shape/text divider," was omitted). To establish a baseline accuracy of our classifier implementation, we first tested the Rubine recognizer with the complete sets of diagram-specific features on the original data sets (R in figure 7).

To evaluate the effects of tracing on Rubine's recognizer, we measured the classification accuracy over two sets of stroke data: the original data sets, as well as a "traced" version of the same strokes. To generate the traced data sets, we first converted each original stroke into a pixel-adjacency graph by linking together sequential points and inserting new points to link together self-intersections. We then ran our stroke tracing algorithm on this graph to generate a transformed stroke, with the same label as the original stroke. If multiple strokes were traced from a graph, we used the longest one as the best single-stroke approximation. The accuracy results of running the baseline recognizer on each data set on these transformed strokes are shown as R_t.

As shown by Blagojevic, more features are not always better, since poorly chosen features can adversely impact recognition accuracy. A possible consequence of image-based stroke extraction is that the timing information generated from the thinning and tracing processes will hurt classification accuracy; features that work well with real timing data could confuse the overall classification by overweighing inconsistent stroke characteristics. If this is the case, then since poor features cannot be entirely ignored in Rubine's classifier, disabling the offending features to exclude them from the training/classification process would be necessary to improve the accuracy of classifying over strokes extracted from an image.

To gauge the impact of timing-based features in general and when dealing with traced strokes, we identified five features of the original set that depend on timing characteristics of a stroke (table 1). We then tested training/classification accuracy with these features disabled on the original stroke data (R^-), as well as on the traced strokes (R_t^-).

As can be seen in figure 7, the impact of using traced approximations of strokes rather than the original strokes (R_t) was minimal; classification accuracy remained relatively stable despite the potentially significant transformation. We believe that the high accuracy results from the tendency for our tracing algorithm to produce strokes very similar to the original, which is likely aided by the fact that the original stroke data sets are composed of single-stroke symbols with limited self-intersections and little over-tracing.

While tracing more complicated symbols could produce strokes that are significantly different from the original, our symbol alphabet ("1", "0", "-", "R", "L") fits the simple, single-stroke model, and we have found classification to be reliable throughout development. Small increases in accuracy are likely due to some classification features benefitting from normalization side-effects of the tracing process.

Though tracing has little effect on recognition accuracy given the full feature set, our data show that removing features has the potential to worsen accuracy (R^- vs. R and R_t^- vs. R_t). With the original stroke data, R^- is not substantially worse that R, meaning that the timing-based features were not key to discriminating between symbol classes. However, disabling timing features when recognizing traced strokes seems to impact recognition, with R_t^- producing slightly worse results than R_t. We suspect that the tracing process expresses some useful geometric property as timing data, and ignoring it produces worse results.

5.2. Low-level Recognition

Recognition for apps within SPARK begins when, one at a time, the strokes are submitted to custom recognition library. In this library, strokes are added to a central *Board*, where various *Observers*, which implement recognition sub-tasks, are alerted to board recognition changes in an event-based manner. Low-level observers register to be alerted when new strokes are added to the board, so that they can directly examine the input for recognized patterns. Specifically in our Turing machine app, Rubine's symbol classifier is implemented as a low-level observer, as is logic for recognizing closed shapes, and arrowheads with their associated tails. When a low-level observer recognizes its specified pattern, it builds a semantic description of the symbol in the form of an *Annotation*, which is then tagged onto the associated set of strokes within the central board.

5.3. Impact of Extraction Noise on Recognition

Traditional writing surfaces commonly contain extraneous marks due to regular use; whiteboards and chalkboards are especially likely to have marks surrounding the relevant diagram space, whether from insufficient erasure or from unrelated, simultaneous board work. Beyond the ambiguity of separating relevant from irrelevant board work, the image normalization stage discussed in Section 4 boosts the contrast between ink and the background board before performing a threshold, which can result in noise artifacts eventually propagating to the recognition logic as extraneous strokes.

In order to remain robust to both sources of noise, sketch recognition logic that operates on strokes extracted from images of general surfaces should account for extraneous marks, for example through domain-specific filtering.

5.4. Turing Machine Recognition

Our application relies on the reliable structure of Turing machine diagrams to filter out ill-formatted text, as well as incomplete graphs. While there is still the possibility of false positives causing error in the final diagram's meaning, the chances are significantly reduced since, for example, text labels must have the correct number and type of characters.

Once individual strokes (or pairs of strokes for arrows) are annotated as basic structures, higher-level observers, which have registered with the board to be alerted to these annotations, examine the basic annotations for further meaning and can tag additional annotations of their own. For Turing machine recognition, our system implements three higher-level observers. First, a Text observer listens for individual letters as annotated by the Rubine recognizer, and, assuming they are of a similar scale and proximity and share a common baseline, assembles them into longer strings. Second, a Directed Graph observer listens for closed shape anda arrow annotations and assembles them into larger graph annotations by first considering arrow direction and proximity to nodes. Finally, a Turing machine observer collects text strings with length three as labels and directed graphs as a state machine, which are assembled into a Turing machine if they meet label and topological structure requirements, for example having one edge with no leading node ensures a valid start state is specified. These Turing machine annotations represent the semantic meaning recognized from the input strokes.

After all of the strokes extracted from the image along with their annotations have propagated through the recognition process, the state of the central board (strokes and annotations) is sent in XML format to the mobile phone. From here, a user can interact with her diagram using a mobile interface, such as the one discussed in Section 3.

6. Discussion

Our Turing machine simulator app serves as a proof-of-concept application within SPARK, combining sketch recognition and image-based stroke extraction, and as such there are many open questions regarding opportunities to increase the general quality of the system by improving stroke extraction, recognition accuracy, and the overall user interface.

In this work, we leverage the stroke-thickness locally to correct errors introduced during bitmap thinning, but there is still a gap between automated results and those that an expert would choose by hand. We suspect that ink drawn by people in a real world context may be too complicated to reliably thin with pixel decisions that only consider extremely local regions of pixels. For example, a more holistic view of the board could disconnect "weakly connected" pixels in the adjacency graph and split ink that belongs to overlapping but logically separate symbols. Beyond thinning, such a holistic perspective could also serve stroke tracing through evaluating multiple combinations of the strokes that arise from an image and choosing the best one according to some global metric which reasonably approximates human drawing patterns. Specifically, it should account for over-traced intersections better than simple smoothness metrics.

While we hope that the automated techniques will someday trace strokes as accurately as a human observer, timing information available to recognition algorithms will always be somewhat arbitrary when using strokes extracted from a single image. We have explored the possible impact of this information degradation on one popular style of stroke classification algorithm, Rubine's recognizer, but it is worth exploring similar impact on other methods. For example, the dynamic Bayesian networks built up by SketchREAD [AD04] seem like they could be robust to arbitrary variations in stroke ordering, but complicated machine learning techniques often react in surprising ways to abnormal data.

When making use of recognition, an important consideration for any sketch application is the rectification of inevitable (but hopefully infrequent) errors. Sketch is inherently ambiguous as an input mode, and without user guidance, some drawings would be recognized incorrectly, even by human observers. With our current system, users are able to see the inner workings of the recognition logic when they touch the screen to see the semantic labels that have been applied to their strokes, but future versions of the mobile interface will support user-guided rectification of recognition errors. We foresee a mode where users will select strokes using touch, on which they will impose the desired interpretation by "suggesting" a new annotation. Currently, the only means for correction are modifying the original drawing and re-capturing an image.

Users of SPARK apps do not need to plan ahead to incorporate recognition into their workflow, since they can capture their work after-the-fact from any whiteboard or paper, once the diagram is recognized, the current system does not support further modification of the diagram once it is captured. However, as pen + touch devices grow in consumer availability, the interaction benefits afforded by touch will come hand-in-hand with the precise drawing abilities of a stylus, and extending our framework to support user modification of captured ink will be a natural next step.

One thing that seems fundamental to this type of interface is that, once the drawing is captured, any changes that come about on the device will not be reflected in the original copy. However, since the diagram now exists as digital strokes, users are able to transition their work to a traditional sketch-enabled device, like a tablet PC. Blending the two workflows leverages the benefits of the impromptu recognition afforded by our mobile, image-based framework, while also gaining the interaction benefits of traditional sketch applications, such as eager recognition and immediate visual feedback.

7. Conclusion

Performing image-based stroke extraction from a mobile interface offers several unique advantages over traditional pen capture hardware. The devices themselves are ubiquitous, and by extracting strokes from images, we can add intelligence to a variety of common surfaces using sketch recognition methods.

Extracting strokes from images of drawn data is not trivial, however, and several considerations must be taken into account to provide results that are reasonable to a user. Raw photos of drawn ink must be robust to variations in lighting, and surface properties through image normalization. The ink that is separated from the board after normalization can be transformed into a single pixel wide adjacency graph using local values of stroke width. Tracing strokes from a single image necessarily strips accurate timing data, yet the results from our evaluation of Rubine's recognizer suggest that a general class of sketch recognition applications can still produce accurate results, despite the loss of information.

Our implementation of a Turing machine simulator in SPARK is a proof of concept system that ties together image normalization with contrast boosting for ink isolation, bitmap thinning and stroke tracing for stroke extraction, and timing-robust sketch recognition techniques for semantic understanding into a ubiquitous-surface Turing machine simulator.

Acknowledgments

The authors would like to thank Steven Boyd for implementation help, and Dr. Plimmer's sketch research group at the University of Auckland for providing the labeled stroke data.

References

[AD04] ALVARADO C., DAVIS R.: SketchREAD: a multi-domain sketch recognition engine. *Symposium on User Interface Software and Technology* (2004), 23. URL: http://portal.acm.org/citation.cfm?doid=1029632.1029637. 9

[BCP10] BLAGOJEVIC R., CHANG S., PLIMMER B.: The power of automatic feature selection: rubine on steroids. In *Joint Session of the Seventh Sketch-Based Interfaces and Modeling Workshop and Eighth Symposium on Non-Photorealistic Animation and Rendering* (2010), Eurographics Association, pp. 79–86. URL: http://dl.acm.org/citation.cfm?id=1923377. 7

[Eik93] EIKVIL L.: OCR-optical character recognition. *Pattern Recognition Letters 9*, 2 (1993), 274–288. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.25.3684. 3

[FBC*04] FARRUGIA P., BORG J., CAMILLERI K., SPITERI C., BARTOLO A.: A cameraphone-based approach for the generation of 3D models from paper sketches. *Sketch-Based Interfaces and Modeling* (2004). URL: http://smartsketches.inesc.pt/sbm04/papers/04.pdf. 2

[Goo11] GOOGLE: Google goggles. http://www.google.com/mobile/goggles, 2011. 3

[HLZ02] HE L.-W., LIU Z., ZHANG Z.: Why take notes? Use the whiteboard capture system. *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings. (ICASSP '03).* (2002), V–776–9. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1200086, doi:10.1109/ICASSP.2003.1200086. 2

[KY99] Kato Y., Yasuhara M.: Recovery of drawing order from scanned images of multi-stroke handwriting. *Proceedings of the Fifth International Conference on Document Analysis and Recognition. ICDAR '99 (Cat. No.PR00318)*, c (1999), 261–264. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=791774, doi:10.1109/ICDAR.1999.791774. 3

[LB09] Liwicki M., Bunke H.: Feature Selection for HMM and BLSTM Based Handwriting Recognition of Whiteboard Notes. *International Journal of Pattern Recognition and Artificial Intelligence 23*, 5 (2009), 907–923. 3

[LLS92] Lam L., Lee S., Suen C.: Thinning methodologies-a comprehensive survey. *IEEE Transactions on pattern analysis and machine intelligence* (1992), 869–885. URL: http://www.computer.org/portal/web/csdl/doi/10.1109/34.161346. 3, 5

[NdPH05] Nel E.-M., du Preez J. A., Herbst B. M.: Estimating the pen trajectories of static signatures using hidden Markov models. *IEEE transactions on pattern analysis and machine intelligence 27*, 11 (Nov. 2005), 1733–46. URL: http://www.ncbi.nlm.nih.gov/pubmed/16285373, doi:10.1109/TPAMI.2005.221. 3

[Nib86] Niblack W.: *An introduction to digital image processing*. Prentice-Hall International, 1986. URL: http://books.google.com/books?id=XOxRAAAAMAAJ. 5

[NS84] Naccache N. J., Shinghal R.: An investigation into the skeletonization approach of hilditch. *Pattern Recognition 17*, 3 (Jan. 1984), 279–284. URL: http://linkinghub.elsevier.com/retrieve/pii/0031320384900773, doi:10.1016/0031-3203(84)90077-3. 5

[OL10] Oliveira D. M., Lins R. D.: Generalizing Tableau to Any Color of Teaching Boards. *2010 20th International Conference on Pattern Recognition* (Aug. 2010), 2411–2414. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5595738, doi:10.1109/ICPR.2010.590. 2

[PS00] Plamondon R., Srihari S.: Online and off-line handwriting recognition: a comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence 22*, 1 (2000), 63–84. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=824821, doi:10.1109/34.824821. 3

[Rub91] Rubine D.: Specifying gestures by example. *ACM SIGGRAPH Computer Graphics 25*, 4 (1991), 329. URL: http://portal.acm.org/citation.cfm?id=122753. 7

[SKC*11] Savva M., Kong N., Chhajta A., Fei-fei L., Agrawala M., Heer J.: ReVision : Automated Classification , Analysis and Redesign of Chart Images. *Processing* (2011). 3

[SLR08] Schenk J., Lenz J., Rigoll G.: On-Line Recognition of Handwritten Whiteboard Notes : A Novel Approach for Script Line Identification And Normalization. *System* (2008), 6–9. 3

[VPF09] Vajda S., Plötz T., Fink G.: Layout analysis for camera-based whiteboard notes. *Journal of Universal Computer Science 15*, 18 (2009), 3307–3324. URL: http://jucs.org/jucs_15_18/layout_analysis_for_camera/jucs_15_18_3307_3324_vajda.pdf. 3

[WFS03] Wienecke M., Fink G., Sagerer G.: Towards automatic video-based whiteboard reading. *Seventh International Conference on Document Analysis and Recognition, 2003. Proceedings.* (2003), 87–91. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1227633, doi:10.1109/ICDAR.2003.1227633. 3

[ZH07] Zhang Z., He L.: Whiteboard scanning and image enhancement. In *Digital Signal Processing*, vol. 17. Mar. 2007, pp. 414–432. URL: http://linkinghub.elsevier.com/retrieve/pii/S1051200406000595, doi:10.1016/j.dsp.2006.05.006. 2

[ZJK*] Zhang M., Joshi A., Kadmawala R., Dantu K., Poduri S., Sukhatme G. S.: OCRdroid : A Framework to Digitize Text on Smart Phones. *Science*. 3