User-Directed Sketch Interpretation

by

Matthew J. Notowidigdo

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

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by . . . .
Robert C. Miller
Assistant Professor
Thesis Supervisor

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

User-Directed Sketch Interpretation

by

Matthew J. Notowidigdo

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

I present a novel approach to creating structured diagrams (such as flow charts and object diagrams) by combining an off-line sketch recognition system with the user interface of a traditional structured graphics editor. The system, called UDSI (user-directed sketch interpretation), aims to provide drawing freedom by allowing the user to sketch entirely off-line using a pure pen-and-paper interface. The results of the drawing can then be presented to UDSI, which recognizes shapes and lines and text areas that the user can then polish as desired. The system can infer multiple interpretations for a given sketch, to aid during the user's polishing stage. The UDSI program offers three novel features. First, it implements a greedy algorithm for determing alternative interpretations of the user's original pen drawing. Second, it introduces a user interface for selecting from these multiple candidate interpretations. Third, it implements a circle recognizer using a novel circle-detection algorithm and combines it with other hand-coded recognizers to provide a robust sketch recognition system.

Thesis Supervisor: Robert C. Miller
Title: Assistant Professor

Acknowledgments

I would first like to thank my parents for encouraging me to stay in school and complete this thesis and my degree program. They have made incredible sacrifices to give me the opportunity to study at MIT, and I want them to know that this thesis represents only one of many academic opportunities that I enjoyed while I was here.

I would also like to thank all of the many members of the MIT Computer Science and Artificial Intelligence Laboratory who have assisted me: Bryt Bradley and Adel Hanna for their constant availability and for having all the answers; Alisa Marshall, Vishy Venugopalan, Michael Bolin, Ryan Jazayeri, Brian Stube, Maya Dobuzhskaya, Phil Rha, and Min Wu for helpful comments and for evaluating early versions of the system; and John Guttag for being an excellent academic advisor, who steered me towards this enlightening path, helped me choose challenging courses, found me excellent research opportunities, and eventually gave me the opportunity to teach.

Lastly, I want to thank my remarkable thesis advisor, Rob Miller. You gave me constant support for my project, and even when progress was slow, your encouragement never ceased. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Structured diagrams (e.g. flow charts, module dependency diagrams) are commonly created using an editor such as Visio or XFig. These applications are powerful and expressive, but they are cumbersome to use, and they make it difficult to communicate drawing styles and shape sizes. Realizing the shortcomings of the user interfaces of these applications, recent research has focused on *sketch understanding* systems that recognize hand-drawn structured diagrams [3, 8]. These systems use stroke information collected on a tablet computer to recognize parts of the diagram as the user sketches them. While these systems are more natural to use than the menu- and mouse-driven editors, they have subtle deviations from an actual pen-and-paper interface. For example, adding an arrowhead to a line segment that was drawn much earlier may confuse the system since the recognition depends on temporal information. The user must delete the line segment and redraw it with the arrowhead.

UDSI (User-Directed Sketch Interpretation) is a new system for creating structured diagrams that is based on understanding hand-drawn sketches of structured diagrams. Unlike existing systems that require devices that can capture stroke information while the user sketches the diagram, UDSI uses scanned images of sketches. The user presents a scanned pen sketch of the diagram to UDSI and guides the application's interpretation of the sketch. The final result is a structured diagram that can be incorporated into technical documents or refined in an existing structured graphics editor. The user can therefore iteratively create the diagram using a pure

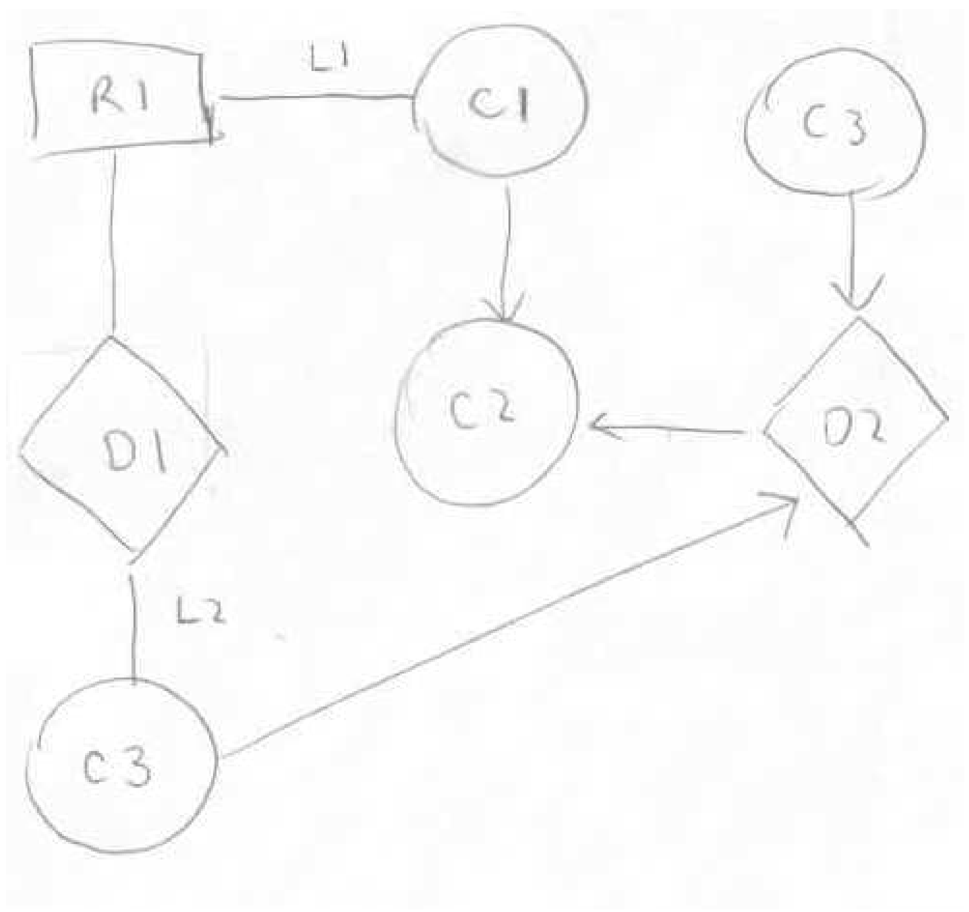


Figure 1-1: Original pencil sketch (scanned)

pen-and-paper interface until she is satisfied with the style, layout, and position of the components of the diagram.

The power of this system is that the user can use the pen-and-paper interface where it is natural and convenient (e.g. sizing shapes, placing arrows, and routing line segments) and then can use a mouse and keyboard where it is more appropriate (e.g. typing in text, selecting colors, choosing fonts).

The UDSI system combines standard algorithms from the machine vision literature for filtering, segmentation, and shape detection [5] with novel algorithms for extracting text regions and recognizing arrowheads. These algorithms produce (possibly conflicting) interpretations of the scanned sketch that are combined using an elimination algorithm to produce not only the best candidate interpretation, but also multiple alternative interpretations. These alternatives are presented to the user

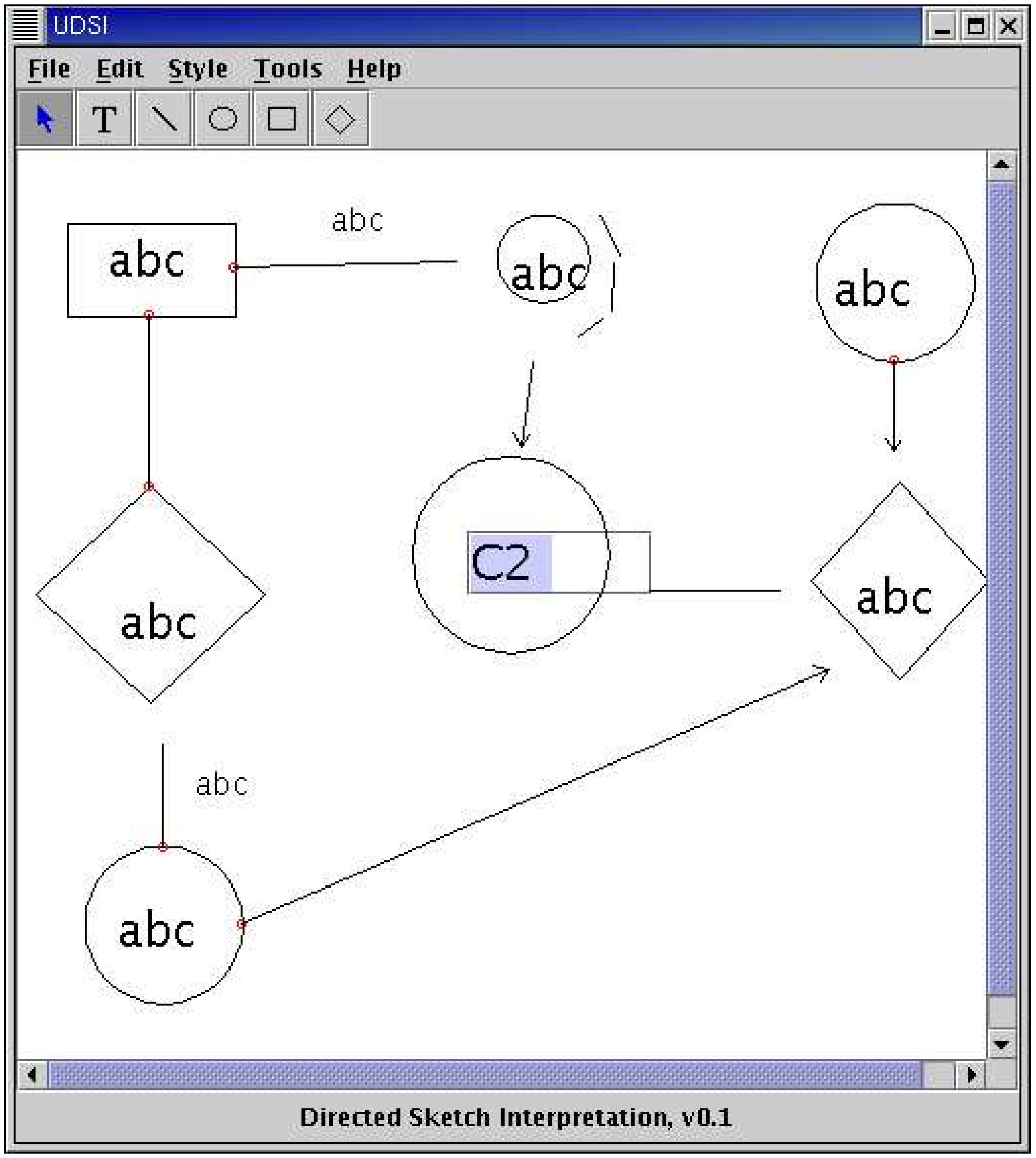


Figure 1-2: User editing a recognized sketch in UDSI

through a novel user interface that allows the user to effectively select alternative interpretations that the system has generated.

The motivation for this work is similar to other sketch recognition systems (see Related Work below): to provide the user with a more natural interface for creating structure diagrams. Many of the sketch recognition systems reviewed below are on-line systems that require an instrumented tablet PC to properly recognize the user's strokes. UDSI eliminates this restriction and allows the user to create sketches off-line.

The rationale is that though tablet computer represent an extraordinary opportunity, the adoption rates continue to lag, while adoption rates for low-cost image acquisition devices (scanners, digital cameras, and camera phones) continue to soar. Therefore, I believe it is still worth investigating off-line recognition, even though it might sacrifice some amount of accuracy and precision. It is worth noting that my recognition approach should be immediately usable on a tablet PC, since it is vision-based instead of stroke-based.

1.1 Thesis Contributions

This thesis makes the following contributions:

- It presents an off-line sketch recognition system that is capable of interpreting scanned pencil sketches of box-and-line diagrams. The recognition system incorporates standard algorithms from the machine vision literature with novel algorithms for circle recognition, shape filtering, and automated generation of alternative interpretations.

- It describes the design and implementation of a user interface for correcting recognition errors.

- It describes a method for testing the usability of the entire system and it analyzes results from a controlled experiment that quantitatively and qualitatively compares the system to existing structured graphics editors.

1.2 Thesis Overview

Chapter 2 discusses the substantial existing literature and related work. Chapter 3 details the motivation for this work, and how it fits into the existing literature. Chapter 4 discusses the design and implementation of the user interface. Chapter 5 discusses the recognition module, including the novel algorithms for recognition and filtering. Chapter 6 presents the alignment module. Chapter 7 analyzes the quantitative and qualitative results of the user study. Chapter 8 concludes and presents future work.

Chapter 2

Related Work

This section briefly reviews related work in sketch understanding and image editing. Previous research in sketch understanding has generally chosen one of two paths: (1) on-line interfaces that require an instrumented drawing surface that captures stroke information. or (2) off-line interfaces that allow the user to sketch using pen-and-paper.

2.1 On-line Interfaces

On-line sketch recognition systems have been developed for a wide range of diagrams. Smithies [15] and Zanibbi [17] created systems to interpret handwritten mathematical expressions. Smithies developed a system that used on-line feedback to guide the understanding of hand drawn mathematical expressions. The system had a user interface that allowed the user to make corrections when the system misinterpreted an expression. The user made these corrections using gesture recognition. For example, to tell the system to reinterpret an equals sign that had been misinterpreted as two minus signs, the user first switches to "regroup" mode and then the user draws a line through both lines in the equals sign. Analogous strokes can be used to regroup symbols that were mistakenly interpreted as separate expressions. The goal of this system was to allow LaTeX and Mathematica novices to easily input complex mathematical expressions. During time trials, novices still took a long time to enter simple mathematical expressions, and experienced LaTeX and Mathematica users were able to create the expressions in much less time than experienced users in the sketch-based system.

Lank, et. al [8] and Hammond [3] independently designed systems that recognize hand drawn UML diagrams. Lank used on-line stroke information to recognize UML "glyphs". The interface to this system was similar to the system described in the Smithies research paper. The user was able to indicate when the system would interpret the sketched diagram, and the user could also force the system to regroup or degroup expressions, and reinterpret them, if necessary. The UI allows the user to correct the system at each stage of understanding: acquisition, segmentation, and recognition.

Hammond also designed a system to recognize sketches of UML diagrams. The system was on-line, like the Lank system. The differences are that the Hammond system does not attempt to use character recognition, whereas the Lank system provided a correction mechanism to ask the user whether the interpreted glyph was character data or UML data. Hammond also uses different heuristics to determine which UML component was drawn. In addition to stroke information, collections of strokes ("gestures", they are called) are used to define UML component classes. A nice feature of the Hammond system is that it is non-modal: users can edit and draw "without having to give any explicitly advance notification." This presumably leads to a more natural interaction, since the user can focus on the UML design, and not giving directions to the application. Also, the system avoids a gesture recognition interface entirely.

Sezgin, et. al [14] have created a sketch understanding system that recognizes geometric primitives. Using stroke information, as well as shape and curvature information, they have created a three phase system (approximation, beautification, basic recognition) that transforms sketches into sharpened diagrams. Sezgin also mentions four key components for on-line sketch understanding interfaces: (1) it should be possible to draw arbitrary shapes with a single stroke, (2) the system should do automatic feature point detection, (3) the system should be non-modal, and (4) the system should feel natural to the user.

Landay developed a tool called SILK to allow users to sketch a user interface [7]. The sketch could then be converted into an interactive demo of the sketched GUI. The system used gestures (single strokes, in this case) to allow the user to indicate various actions (deleting widgets, ungrouping widgets, etc.)

Igarashi, et. al developed a "Flatland" user interface that supported stroke-based interaction on an augmented whiteboard [7]. The goal was to create an enhanced whiteboard that still had the same physical advantages (e.g. persistent strokes). The modal system supports several behaviors (to-do list, map drawing, 2D geometric drawing, calculator) that are selected by the user.

Jorge and Fonseca present a novel approach to recognizing geometric shapes interactively [6]. Using a digitizing tablet, they provide a recognition algorithm to recognize multi-stroke sketches of geometric shapes through the use of fuzzy logic and decision trees.

2.2 Off-line Sketch Understanding Systems

All of the sketch understanding systems discussed above are on-line systems that require that the user be present at the computer when she is sketching her design. There is also existing research on interpreting sketches that were created with pen-and-paper, which is closer to the UDSI system.

Valveny and Marti discuss a method for recognizing hand-drawn architectural symbols [16]. Their method operates on 300 dpi bitmaps and therefore works as an off-line system. Their recognition rates are around 85%, but they do not discuss how the user might correct an incorrect recognition. The method of deformable template matching, however, should extend beyond hand-drawn architectural symbols to other types of hand-drawn symbols (such as chemical structures or flow charts).

Ramel, et. al discuss a new strategy for recognizing handwritten chemical formulas [10]. They utilize a text localization module that extracts text zones for an external OCR system. They also use a global perception phase that incrementally extracts graphical entities. Their system achieves recognition rates of 95-97%, but as in Valveny and Marti, there is no discussion of how the user can correct recognition errors.

2.3 Sketch Editing

There has been recent research investigating novel image editing interfaces for informal documents such as scanned sketches or handwritten notes [11], [13]. This work has introduced novel user interface techniques for selecting image object material from a static bitmap. These tools attempt to fill the current void in computer editing tools that provide support for perceptual editing of image bitmaps. UDSI is similar to this line of research in that the input documents are informal, rough sketches, but UDSI differs in the functionality it provides. While ScanScribe allows users to edit the bitmap directly, UDSI provides an interface for the user to gradually transform the rough image bitmap into a structured diagram.

Chapter 3

Motivation

Consider the following scenario: a small team of engineers is conducting a design discussion in a meeting room. People take turns turns editing a high-level diagram (perhaps an abstract problem object model) on a dry-erase board. One of the members sketches the resulting diagram on her legal pad. She then returns to her office, where she re-creates the diagram in a structured graphics editor and then sends it to the team along with the meeting notes. The engineer's task is essentially taking a sketch of an abstract diagram and creating a digital version that can be edited in a structured graphics editor.

This task adequately summarizes the problem I am trying to solve. UDSI should allow the user to convert sketches of structured diagrams into recognized forms that can be refined in existing structured graphics editors. Motivation comes from two sources: (1) frustration with the cumbersome interfaces of existing structured graphics editors, and (2) belief that vision-based approaches to sketch recognition expands the set of recognizable diagrams. This chapter discusses and expands on this motivation.

3.1 Task Analysis

I conducted a task analysis[1] of structured diagram creation using a common existing structured graphics editor, Microsoft PowerPoint. I recruited several graduate

[1]For an excellent introduction to user and task analysis, see [2].

teaching assistants and observed them completing a task that asked them to create a module dependency diagram from a code sample. Graduate teaching assistants were chosen so that they would have no problem completing the task. As the users completed the assigned task, I observed some interesting behavior: users did not plan how the diagram would be laid out. Instead the users immediately started creating shapes in PowerPoint. As figure 3-1 shows, often users had to re-organize the layout of their diagram. Because these users did not user PowerPoint frequently enough to learn any of the special features that would help in this process (such as grouping and alignment), the users would manually move each shape around and around until they were satisfied. As discussed in the Evaluation section, users often produced cleaner diagrams if they were told to sketch the diagram beforehand. This suggests that users are overconfident (or not sufficiently risk-averse) and therefore they believe that immediately using PowerPoint before thinking is an optimal strategy [2]

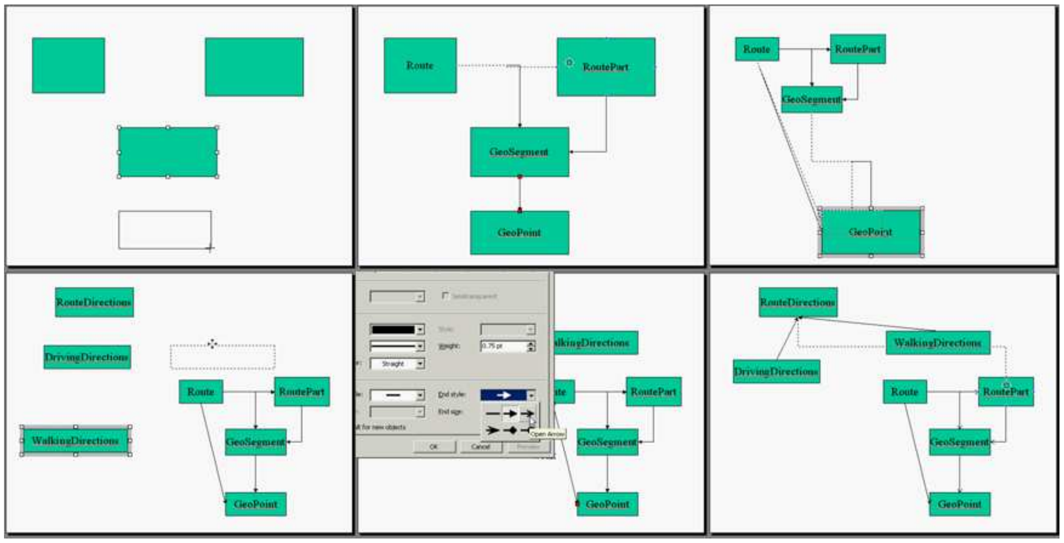


Figure 3-1: Screenshots from task analysis of user producing a diagram in PowerPoint. Note the considerable re-organization. Total task took 12 minutes to complete. These screenshots are taken at 0:59, 2:22, 4:53, 7:29, 9:08, and 10:52.

In addition to the constant moving and aligning of components, I observed many mode errors as the users tried to select and/or create elements. This mode error

[2]This myopic behavior reminds me of the notion of "bounded rationality" that is currently an active research area in psychology and economics. Like the utility-maximizer who doesn't know what's best for him, it is certainly difficult to design an interface for the user who doesn't know (and doesn't bother to think about) the best way to solve the problem.

has also been observed by [12]. Finally, the users seemed to find the user interface cumbersome to use. Because the subjects in the task analysis used structured graphics editors infrequently[3], they did not remember how to do many of the tasks needed to modify their diagram (e.g. change line endpoints or modify line styles to create dotted lines). Even when the users figured out how to execute their tasks, they were presented with dialogs like in Figure 3-2. This dialog slows the task completion time because users must scroll through all possible line endpoints to select the appropriate endpoint. There is no way for the user to immediately indicate the user's intended line endpoint.

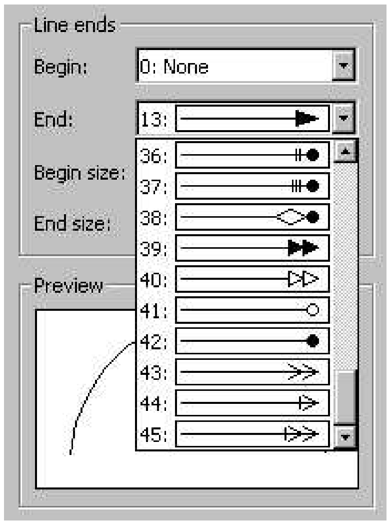


Figure 3-2: Editing line endpoints in Microsoft Visio.

By contrast, when I observed users completing this task (during the user study) by sketching the document with pen and paper, users were able to immediately indicate the intended line endpoint by simply drawing the type of line endpoint that they wanted. There were also no mode errors; users could effortlessly switch between creating shapes and lines. When users struggled, they found that their initial layout was unsatisfactory, and they had to delete and re-create the elements, either by erasing or starting over again.

Recall the introductory software engineering lesson of "Think Before You Code!".

[3]Presumably because these types of diagrams are generally "afterthoughts" and not maintained like they *should* be. Perhaps if these diagrams – and code documentation in general – were updated more regularly, the problem would look different.

Similar advice seems warranted for creating diagrams in structured graphics editors ("Think Before You Draw!"). In addition to the Task Analysis conducted, I also collected several homework assignments from students in the introductory software engineering laboratory course at MIT[4]. The homework assignments asked them to create module dependency diagrams, three examples of submissions are shown in figures 3-3, 3-4, and 3-5. Figure 3-3 was created in Microsoft Visio; figure 3-4 was created in Dia, a structured graphics editor for UNIX; and figure 3-5 is a scanned pencil sketch.

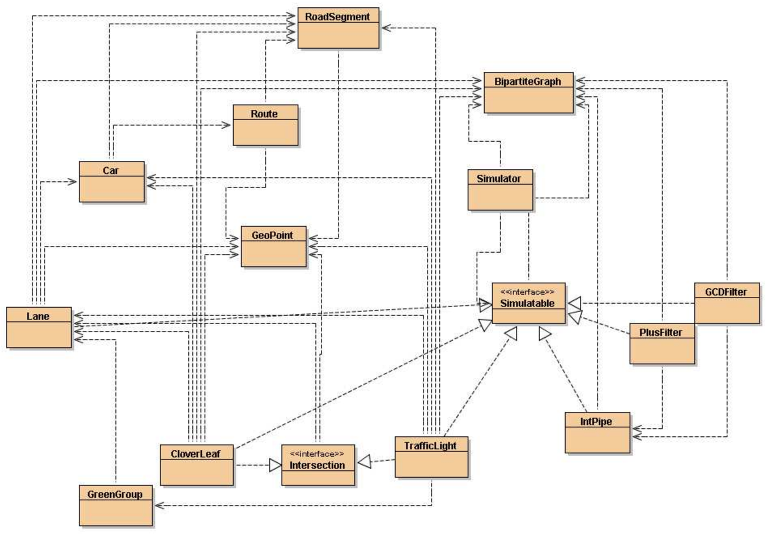


Figure 3-3: Module dependency diagram created in Microsoft PowerPoint. Note how the auto-routing of the connectors has ensured that lines do not needlessly cross, but the student has allowed that feature to substitute for the student's own thinking about the layout of the diagram.

These figures show that users seem to be more able to communicate their diagrams using pencil and paper than using existing structured graphics editors. In figure 3-3, the user has let Visio automatically route the connectors, instead of thinking about how to arrange the shapes so that the relationships between classes is clearest. In figure 3-4, the diagram is more clear, but again the user was either too lazy, or could not figure out how to route the connectors between shapes in order to clearly present the dependencies. By contrast, figure 3-5 clearly expresses the relationships between

[4]Many thanks to the students who agreed to let their work be used.

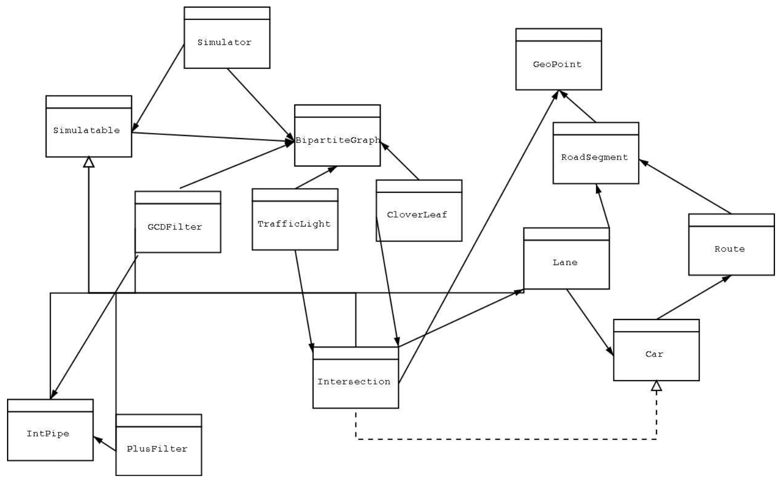


Figure 3-4: Module dependency diagram created in Dia. The user has used both automatically routed connectors, as well as basic lines to create connections (dependencies) between classes.

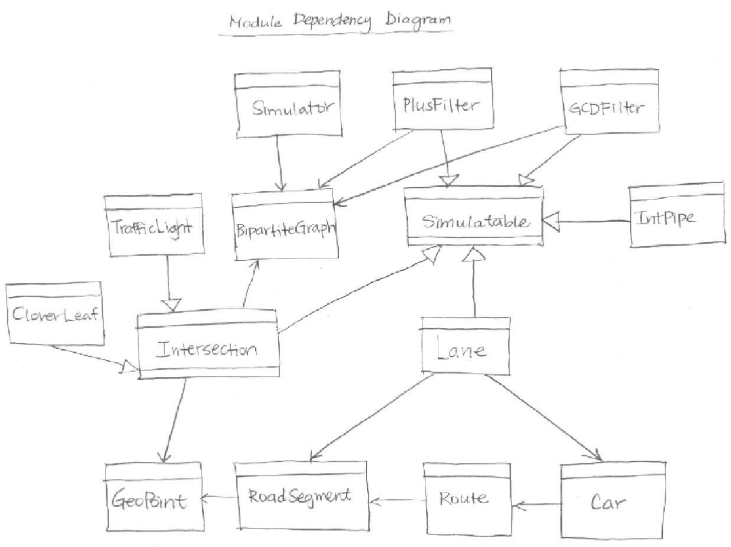


Figure 3-5: Module dependency diagram sketched by hand.

the classes in the diagram. This student likely created one or more drafts until the layout below was decided, but since pencil sketches are much easier to re-create than structured diagrams, such re-starting is not costly.

The dialogs in figure 3-2 (and others like it) and the students' figures represent a strong motivation to consider moving away from menu-, mouse-, and keyboard-driven user interfaces for structured graphics editing applications. If a user wants to add a certain endpoint to a line, the natural interface is not typing in a string describing the endpoint, or selecting the endpoint from a list of options, but simply *drawing* the endpoint that you wish to add on the end of the line. The argument that diagram creation is more naturally accomplished through sketch recognition interfaces is central to much of the research discussed in the previous chapter on sketch understanding systems. UDSI has the same motivation of providing the user with drawing freedom. I hope that the resulting interface is less cumbersome and more natural to the user.

Instead of building on the existing research of on-line sketch recognition systems, I have decided to focus on off-line recognition. As discussed in the Introduction, part of the motivation is that adoption rates of tablet computers continue to lag, while adoption rates for low-cost image acquisition devices (scanners, digital cameras, and camera phones) continue to soar. Also, as argued in the next section, off-line recognition offers interesting insight into how to approach the non-gesture-based and non-stroke-based recognition problem.

3.2 Towards a Vision-Based Approach

By restricting the system to off-line sketch recognition, the user should have complete drawing freedom during the sketching stage. The pencil-and-paper interface imposes no restrictions on the type of strokes the user can make and no restrictions on the order of the strokes. The off-line approach also allows the user to easily "overtrace" – to re-draw the shape over itself. Overtracing has proven difficult for many of the on-line recognition systems[5], but the system may handle overtraced images without difficulty (the gaussian filtering stage, discussed in the Recognition section, smoothes overtraced shapes so that the resulting edge points are almost identical.

Many on-line systems have tried to move towards recognition systems that are invariant to stroke order instead of requiring rectangles to be drawn in a fixed number of strokes in a fixed order (e.g. four strokes starting at the top and going clockwise). In Hammond's system for recognizing hand-drawn UML diagrams [3], for

[5]From conversation with Metin Sezgin.

example, rectangles can be drawn using one, two, three, or four strokes in arbitrary order. The only requirement is that all the strokes from the shape must be drawn sequentially; in other words, the user cannot draw two lines for the rectangle, then go and draw a circle, and then complete the drawing of the rectangle. This might seem to be a harmless assumption, but many times users will want to change aspects of components after they have already been drawn. One example could be filling in an open triangle arrowhead that has already been drawn much earlier. In fact, recognizing "filling" may be difficult because systems often interpret jagged strokes as "delete" gestures. With an off-line approach, the user is interacting with only a pencil-and-paper interface while sketching, so the user can fill the arrowhead at any time, using any stroke sequence. Figure 3-6 shows a sketch from the mozilla documentation page that illustrates several features that were discussed above (filled shapes, filled arrowheads).

Hopefully, UDSI can demonstrate that in restricted domains, off-line sketch recognition is robust enough to be practical. I believe this would show that a vision-based approach to sketch recognition is promising. By "vision-based", I mean using only information about the final image to be recognized, not information that might have been collected *while the image was being created*. Vision-based recognition may broaden the class of recognizable shapes (e.g. filled arrowheads), and it also has the potential to increase the usability of sketch recognition systems, both on-line and off-line.

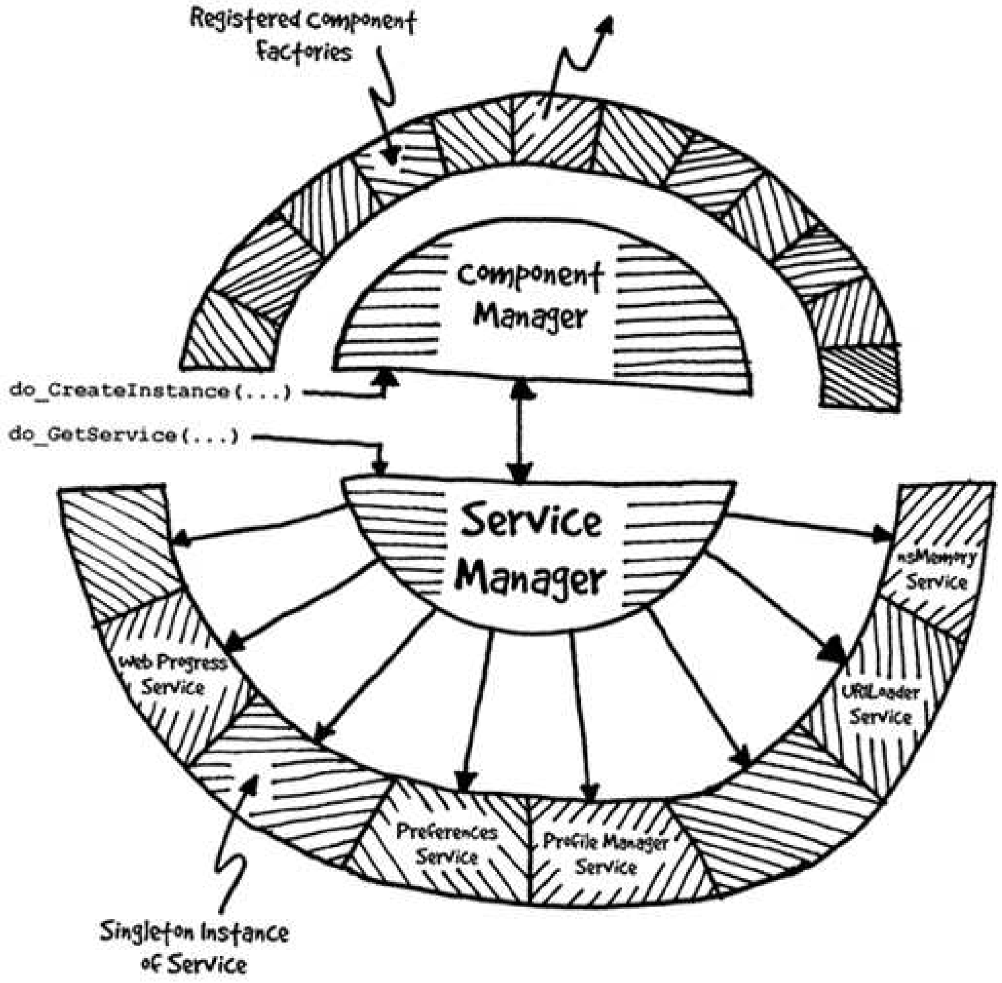


Figure 3-6: Sketch from mozilla documentation. This sketch would be difficult to create in an on-line recognition system because stroke-based recognition makes filled triangle arrowheads difficult. The filled areas would also be difficult strokes to categorize.

Chapter 4

User Interface

This chapter describes the novel graphical user interface that allows the user to correct recognition errors. The first section of this chapter discusses the general model of the task, and the second section discusses the design of the user interface.

4.1 UDSI Task Model

The task model when using UDSI is as follows: first, the user sketches a diagram using pen and paper. This sketch should be reasonably clean and should adequately represent the user's intended layout and positioning. If not, the user can erase and re-arrange the sketch or simply re-create a new sketch from scratch. Second, the user creates an image bitmap from the pen sketch. Currently, I use a scanner to scan and crop the pen sketch, but one could also use a digital camera or a camera phone, assuming adequate resolution. Finally, the image bitmap is presented to the UDSI application, which recognizes lines, shapes, and text regions. The user then corrects recognition errors within the application until the user is satisfied with the structured diagram.

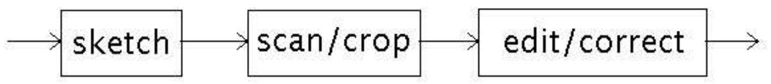


Figure 4-1: Task model of UDSI. Note that this diagram was produced using UDSI. The original image bitmap and initial interpretation are in Appendix B.

4.2 Design

The UDSI graphical user interface combines many features from standard graphics editors (move-to-back/front, drag-select, multiple-selection, connection points) with a mediation interface to select from alternative interpretations. The result is a GUI that looks and feels like a standard graphics editor (e.g. Visio, XFig) that has been slightly instrumented to take advantage of recognition information to help guide the user's editing process. The user interface was implemented entirely using Java Swing.

When the recognition module is accurate, the users should only need to edit text regions to contain appropriate text (see figure 1-2). When there are recognition errors, the users should consider the mediation interface that presents alternate interpretations (see figure 4-2). These choices can save the user time because they will present consistent alternatives. The alternate interpretation is shown in more detail in figure 4-3.

Using the work of Mankoff, et. al, [9] as a guide, I implemented a choice-based interface to resolve ambiguity. Because all of the recognition is done off-line, however, it is important to recognize shapes and discover alternate interpretations after the user has drawn all of the shapes in the diagram (as opposed to in real-time). To use the dimensions discussed in [9], I decided that the instantiation of the choice-based mediator would be simply selecting the shape that contains multiple interpretations. Describing to the user which recognized shapes have alternate interpretations is an open problem that is left for future work. One solution that is currently being implemented is an "orange bubble" around shape(s) that the system has multiple interpretations for; though this might be disorienting to the user, especially if the initial interpretation is correct, since then the user might have to take action to tell the system to ignore the alternatives (and remove the orange bubble). The current implementation simply pops up the alternatives any time it has alternates, and the user can just ignore them on the right-hand side of the screen.

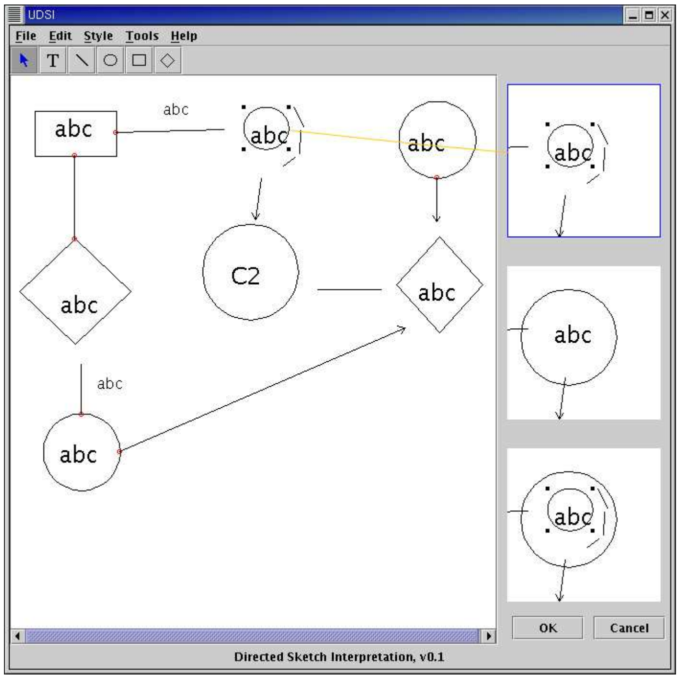


Figure 4-2: User selection of ambiguous shape triggering display of alternate interpretations.

4.2.1 Iterative Design and Prototyping

The user interface was developed by iterative design. I was unsure about what kind of user interface would allow the user to guide the interpretation of the scanned sketch, so I began the design process by creating and evaluating several low fidelity (paper) prototypes (see figure 4-4). Evaluating the lo-fi prototypes generated two important: first, like many existing structured graphics editors, the proposed interface suffers from serious mode errors. For example, when the user was in Create Rectangle mode and clicked to place a rectangle on the screen, the rectangle was created and the user then moved the 'mouse' and clicked to place another rectangle on the screen. However, the system had (by default) switched to Pointer mode after the first rectangle

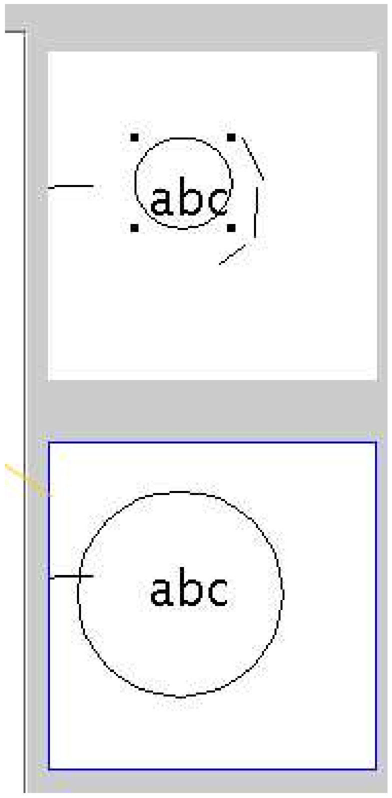


Figure 4-3: User selecting an alternate interpretation in UDSI. Note that the alternate interpretations are consistent: the lines segments that previously existed are now subsumed by the larger circle in the new interpretation.

had been created; therefore, the user experienced a mode error. This mode error required that the user explicitly switch back to Create Rectangle mode, increasing the amount of time it would take to create the desired set of rectangles. When I changed the prototype to instead remain in Create Rectangle mode after a rectangle had been placed, a different set of users made a different mode error. This time the users intended to click to select the recently created rectangle, but instead the system created a second rectangle in a similar position. As before, the user needed to explicitly change modes to recover. This mode error problem has also been noted by Lank & Saund [12].

Second, I learned that users were unwilling to test out the experimental features, specifically the choice-based mediator that was designed to assist the user in interpreting the scanned sketch. In the paper prototype there was a Reinterpret Shape mode where the user could click on a shape and be presented with a list of reinterpretations. I believed that users would prefer to choose other interpretations from a list of alternatives to having to delete an incorrect interpretation and create the correct shape. To use the language from Mankoff, et. al, [9], I believed users would prefer a choice-based mediator to a repetition-based mediator in a different modality. The rationale was that the repetition-based mediator was cumbersome: the user needed to delete the misrecognized shape, change mode to be able to create correct shape, and finally place new shape at correct location. Users offered two explanations for why they stuck to the repetition-based mediator: (1) they knew it would work, (2) they weren't sure how they would use the Reinterpret Shape mode once they were in it.

Assuming sufficiently accurate initial recognition, there should be no huge efficiency loss for users who only use Delete and Create operations to resolve interpretation errors. The Delete and Create model is also a simple one for users to understand; a user immediately grasps it when first encountering the application. I believe, however, that when the recognition is less than stellar, the user might be able to receive assistance from the stored alternative interpretations generated during the greedy elimination algorithm, particularly for a set of related misinterpretations (see figure 4-3).

The remaining prototypes attempted to refine a user interface that allowed the user to respond to presentations from the application of alternative interpretations. Several heuristic evaluators provided feedback on how to present this interface to the user. They concluded that it was difficult to understand the purpose of a labeled button or a mode that said Modify Interpretation or Reinterpret Shape. In the final design, therefore, I decided to make the presentation of alternative interpretations a more active part of the diagram editing process. Now when the user clicks on a shape that the system (based on heuristics) believes might have been incorrectly recognized, a non-modal dialog opens on the side of the application that allows the user to select from alternate interpretations. The user can simply ignore the window if either the original interpretation is correct or if the user prefers to execute the previously described Delete-Create loop instead.

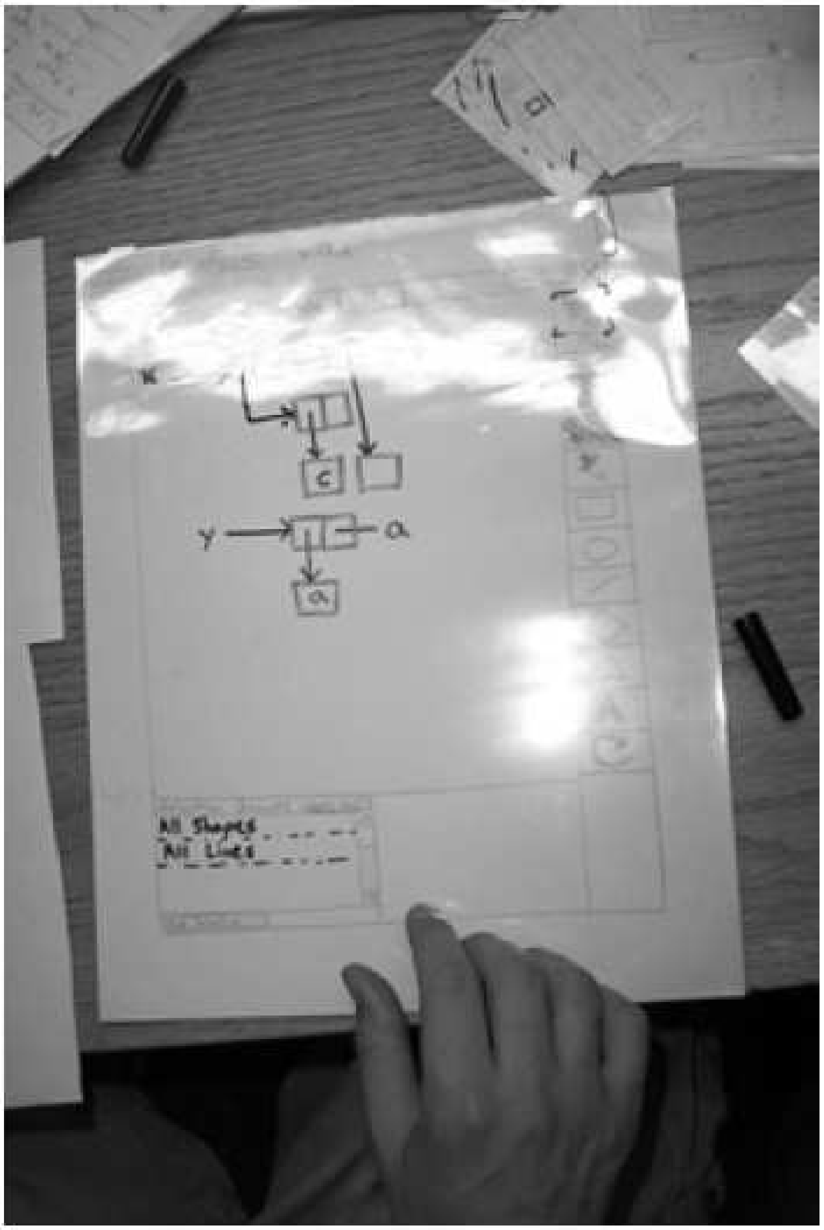


Figure 4-4: Paper prototype of UDSI. The interface used transparencies and dry-erase markers to easily indicate modes to users and also to allow the "computer" running the prototype to easily update the status of the interface.

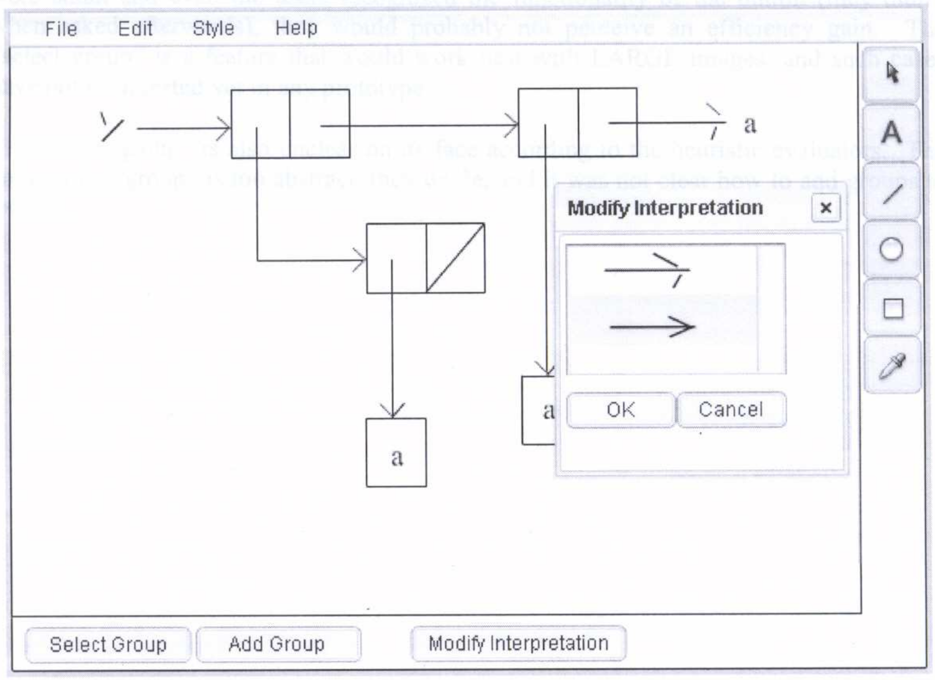


Figure 4-5: Computer prototype of UDSI written in Flash. This prototype was evaluated by heuristic evaluators. The interface, however, was not completely static. By using Flash's built-in support for easy mouse-over mapping, it was possible to evaluate different types of mouse-over feedback easily using this prototype.

Chapter 5

Recognition

This chapter describes the recognition module. The module was written entirely in Java; it extensively uses the Java image API (`java.awt.image.*`) and the Java geometry package (`java.awt.geom.*`).

5.1 Overview

The purpose of the recognition module is to take an image bitmap and convert it to a set of geometric primitives (lines, shapes) and text regions that are presented to the user in the GUI.

The sketch recognition proceeds sequentially through several steps: image smoothing, segmentation, text/shape recognition, and generation of alternatives. At the end of this process, the system contains an initial interpretation and a set of alternative interpretations. The initial interpretation is then aligned appropriately (see Alignment section below). The first two steps of the sketch recognition process are implemented using standard algorithms from the machine vision literature [5]. The image bitmap is passed through a Gaussian filter, and then the segmentation step is implemented using a Canny line detector and algorithms to split and merge line segments. The final step in the recognition process uses hand-coded recognizers for arrowheads, circles, rectangles, diamonds, and text regions. The remainder of this chapter discusses each of the steps of the recognition process in greater detail.

5.2 Segmentation

The first step in the recognition process is *segmentation*. The segmentation phase starts with the original scanned image and produces a set of segments, grouped by contour. All of the other recognizers in the system use the output of this phase to recognize higher-level structure (e.g. shapes, text, alignment). Early prototyping of UDSI convinced me that in order to achieve acceptable performance, algorithms needed to scale with the number of segments, as opposed to algorithms that scaled with the number of edge pixels in the image. This influenced the decision to implement shape recognizers that depend on segmentation results, instead of implementing algorithms that use edge pixels (e.g. template matching).

The segmentation process is accomplished through the following sequential steps: gaussian filtering, line detection, contour following, line splitting, and segment merging. These steps are discussed in more detail below.

5.2.1 Gaussian Filtering

Existing image acquisition devices (e.g. scanners, digital cameras) introduce some noise into the sampled image bitmap of the original sketch. Because of this, an image filter preprocesses the image bitmap before passing it to the core of the segmentation process. We chose to use a discrete gaussian filter because it is rotationally symmetric and therefore will not bias subsequent edge pixel detection in any particular direction (as discussed in [5]). I empirically determined the optimal filter size based on the sample of images collected from the pilot user study, and found that the optimal filter size is 3x3. This optimal filter size is dependent on the size of the image, and also the dpi at which the image was scanned. For all of the experiments, the sketches were scanned at 100 dpi, and the images were all drawn on 8.5x11 paper. The reason the optimal filter size is small is because the scanned images are already highly thresholded; therefore, not much smoothing is needed to aid the line detection. For comparison, figure 5-1 shows an image being passed through various discrete gaussian filters. The trade-off when choosing filter size is that the larger filter smoothes out more of the noise, but at the expense of image detail. The small filter chosen enables preservation of much of the image detail that is useful for later stages of recognition (e.g. arrows, text). The output of the Gaussian filter is an image bitmap representing a smoothed version of the original image. This image bitmap is presented to the Canny line detector to compute the line pixels of the image.

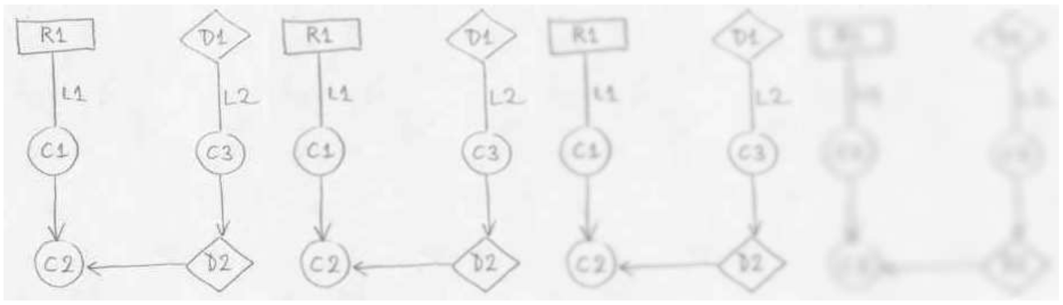


Figure 5-1: Resulting images after applying different discrete Gaussian filters. Original image is at left, followed by 3x3, 7x7, 15x15 Gaussian filters.

5.2.2 Canny Line Detector

The Canny line detection algorithm is a standard algorithm from the machine vision literature that takes, as input, an image bitmap and produces a set of pixels representing line points. At an abstract level, the algorithm is the following (see [5], section 5.7):

1. Compute the gradient magnitude and orientation using finite-difference approximations for the partial derivatives
2. Apply nonmaxima suppression to the gradient magnitude
3. Use double thresholding algorithm to detect and link line pixels

The second derivative should be used in step 1 when approximating partial derivatives in order to find *line* pixels. Using the first derivative will instead find *edge* pixels. To visualize why this change to step 1 results in line detection instead of edge detection, think of an edge in an image as a step function, and a line in an image as an impulse function – which is the derivative of a step function. Once the approximations for the second partial derivatives (in both the x and y directions) have been calculated, gradient magnitude $M[i, j]$ and orientation values $\theta[i, j]$ are easily derived (P_x is an array of first-difference values in the x-direction; P_{yy} is an array of second partial derivative approximations in the y direction):

$$M[i, j] = \sqrt{P_x[i, j]^2 + P_y[i, j]^2} \tag{5.1}$$

$$\theta[i, j] = arctan(P_{yy}[i, j], P_{xx}[i, j]) \tag{5.2}$$

A Sobel operator (see [5], section 5.2.2) is used to calculate the finite-difference approximations in step 1. I empirically determined it was best operator to use, after testing the Roberts, Sobel, and Prewitt operators. It is not immediately clear why this operator performs best, but I see two possible explanations: First, the Sobel and Prewitt operators avoid calculating finite-differences about an interpolated point, which increased segmentation accuracy for the sample set of images. Second, the Sobel operator places emphasis on pixels horizontally and vertically bordering the current pixel, which is where neighboring line pixels are often found, since the shapes and lines in the diagram often contain some horizontal and vertical alignment.

In Step 2 of the Canny line detection algorithm, nonmaxima suppression thins broad ranges of potential line pixels into “ridges” that are only one pixel wide. This helps the segmentation process only produce one line segment per sketched line. Before nonmaxima suppression, however, the array of orientation values $\theta[i, j]$ is bucketed into an array $L[i, j]$ according to figure 5-2. The sector array $L[i, j]$ is used by the nonmaxima suppression algorithm, as described below ($N[i, j]$ is the result array):

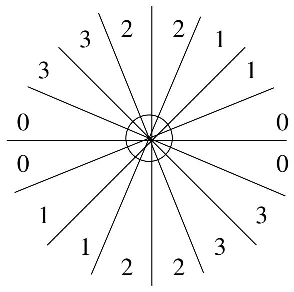


Figure 5-2: Converting gradient values to sectors.

```
NONMAXIMA_SUPPRESSION(i,j,M[,],L[,],N[,]) {
   //horizontal gradient
  if (L[i,j]=0 & M[i,j]>M[i+1,j] & M[i,j]>M[i-1,j])
   N[i,j]=M[i,j]
   //45-degree gradient
  else if (L[i,j]=1 & M[i,j]>M[i+1,j+1] & M[i,j]>M[i-1,j-1])
   N[i,j]=M[i,j]
   //vertical gradient
  else if (L[i,j]=2 & M[i,j]>M[i,j+1] & M[i,j]>M[i,j-1])
   N[i,j]=M[i,j]
   //135-degree gradient
  else if (L[i,j]=3 & M[i,j]>M[i-1,j+1] & M[i,j]>M[i+1,j-1])
   N[i,j]=M[i,j]
  else // suppress
   N[i,j]=0
}
```

The final step in the Canny line detector is the double thresholding algorithm that uses the resulting image array from the nonmaxima suppression, $N[i,j]$. This algorithm uses two threshold values, T_1 and T_2 (with $T_2 >= T_1$), chosen *a priori*, to determine the line pixels. The double thresholding algorithm is straight-forward: any $N[i,j]$ value below T_1 is set to zero; any $N[i,j]$ value above T_2 is immediately marked as a line pixel; a $N[i,j]$ value between the threshold values is marked as a line if-and-only-if there exists a marked line pixel in the 8-neighborhood of pixel (i,j). Figure 5-3 shows the Canny line detection on a sample image; the line pixels are shown in white. Note the performance around the arrowheads. Because of loss of image detail imposed by the Gaussian filter, the arrow recognizer (see below) uses the original image and not the Gaussian smoothed image. This gives slightly more detail which improves arrowhead detection.

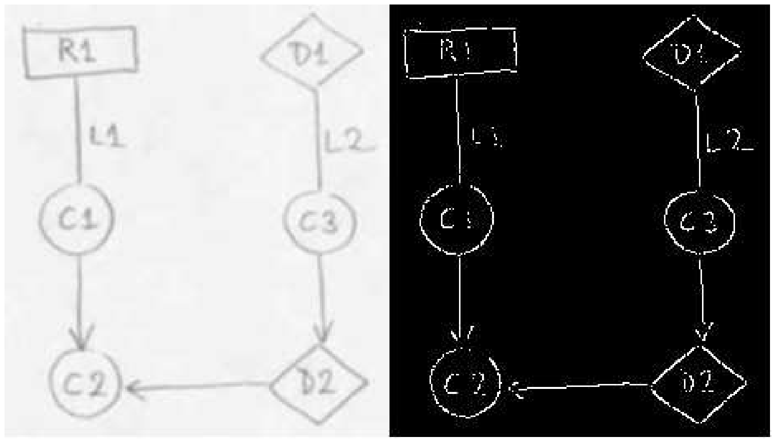


Figure 5-3: Result of Canny line detection. Line pixels are shown in white in image at right; at left, the input (Gaussian smoothed) image.

5.2.3 Contour Following

After the Canny line detector has finished, the array $N[i, j]$ contains the set of line pixels found in the image. The next step in the segmentation process is to aggregate these line pixels into contiguous sets, or *contours*. The contour following algorithm simply includes a line pixel as part of a contour if it is within the 8-neighborhood or 16-neighborhood[1] of another pixel in an existing contour. Line pixels are marked after they have been assigned to a contour; therefore, this algorithm scales linearly with the number of pixels in the image.

5.2.4 Line Splitting

Once the line pixels have been grouped into contours, the next step is to split each contour into a set of connected lines. This is accomplished using a *recursive subdivision* algorithm that creates polylines from contours. The algorithm requires a constant parameter, $d \in (0, 1)$, that is used to determine the maximum tolerated

[1]The 16-neighborhood is the band of pixels around the 8-neighborhood; this second band is included to improve performance of subsequent recognizers. Circle detection, in particular, benefits from accurate contour information.

deviation from the line fit. The algorithm begins by assuming that the contour is a single line segment between the two endpoints of the contour. The algorithm recursively splits the line a pixel in the contour set is beyond the maximum tolerated deviation. The polyline splitting algorithm is described graphically in figure 5-4.

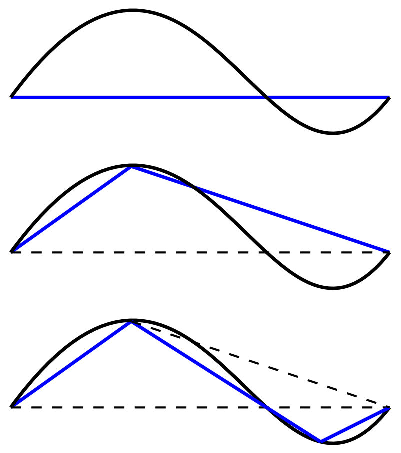


Figure 5-4: Description of line splitting algorithm. The contour is split into a set of line segments.

5.2.5 Noto's Segment Merging Algorithm

The final step in the segmentation process merges line segments together. After the line splitting step, the contours have been split into segments. Before passing this set of line segments to the recognizers, a segment merging algorithm is used to merge line segments that are aligned and close together, regardless of contour. This step is included because the Gaussian filter sometimes introduces holes in shapes, and this merging algorithm merges line segments straddling the hole so that the shape recognizers can be more accurate.

Unlike some of the merging algorithms discussed in the machine vision literature (see [5], section 6.4.2), this merging algorithm is not based on previously marked contours; in other words, in the algorithm, all other line segments are candidates for merging. Lines are merged if and only if they have an approximately equal angle (within a threshold, θ_m), and one line's endpoint is within a distance, d_m, of an endpoint of the other line.

This algorithm is inefficient, but seems to converge quickly in practice. It has been very helpful in creating elongated edges from lines that have been split from different contours, which increases the accuracy of the shape recognizers. The result of the line merging algorithm is shown in figure 5-5. Note how the edges of the rectangles and diamonds have been merged successfully.

Once the line merging algorithm terminates, the resulting set of line segments is preserved and used by the recognizers. The next section discusses each of the recognizers in detail.

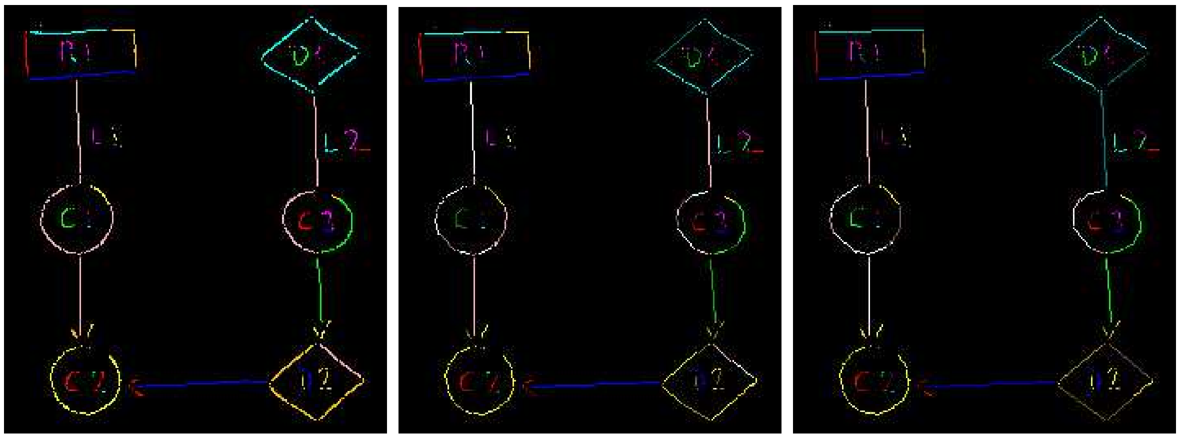


Figure 5-5: Resulting images of contour following, line splitting, and segment merging. At left, result of contour following step. Line pixels that are part of the same contour are colored the same. The center image shows the result after the polyline splitting algorithm. The right image shows the result after the merging algorithm. Note how the top line of the rectangle has been merged into a single segment.

5.3 Recognizers

After the segmentation step, the recognizers are executed to recognize shapes, text regions, and line endpoints (arrowheads). The recognizers are largely independent, but some of them require output of other recognizers (e.g. the rectangle and diamond recognizers require output of the corner recognizer). The recognizer dependencies are graphically presented in figure 5-6.

All of the recognizers are feature-based, as opposed to statistical. This means that the rectangle recognizer asks questions like “Are there corners parallel and per-

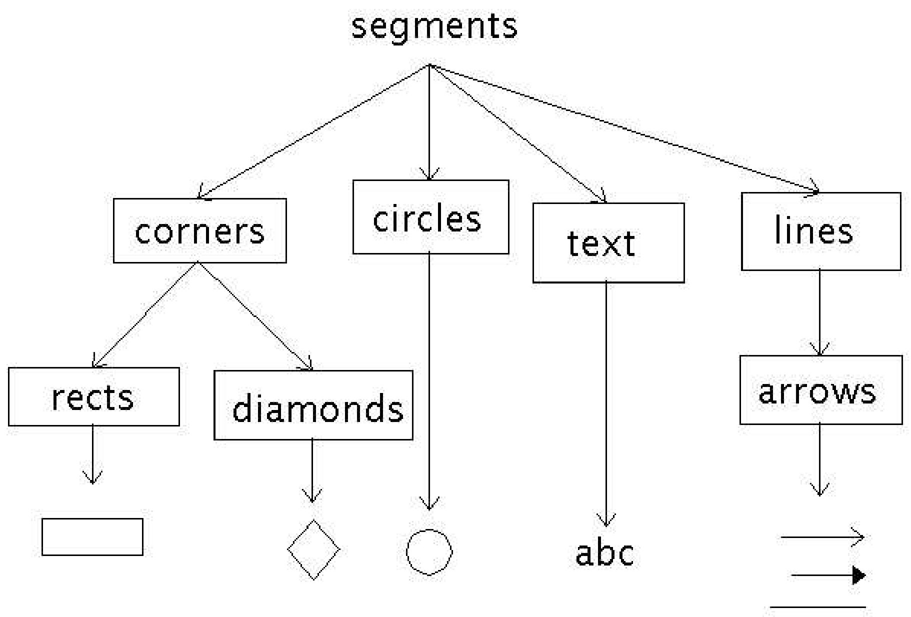


Figure 5-6: Overview of recognizer dependencies. Note that this diagram was produced using UDSI. The original image bitmap and initial interpretation are in Appendix B.

pendicular to me?" instead of questions like "How similar am I to this known shape, *s*?" The recognition design, however, should be modular enough so that statistical recognizers could be substituted for the hand-coded recognizers. The decision to use feature-based recognition was based largely on simplicity.

Currently, UDSI recognizes text regions, rectangles, diamonds, circles, and arrowheads. These recognizers are discussed in more detail in the following subsections.

5.3.1 Text Recognition

Dense regions of short line segments are assumed to be text regions. The system does not implement character recognition, but it does localize and recognize text regions that it presents to the user as editible text fields. The extraction of text regions proceeds using a simple merging algorithm. For each small line segment, the system scans for other similar segments that are within a small neighborhood of the given segment. If the two segments are close enough, then the system merges the two segments into a single text region. This algorithm continues until the system can no longer merge any recognized text regions. Finally, the algorithm drops text regions containing fewer segments than a fixed threshold. This threshold was determined empirically to be at least three segments per text region, but this value seems to be very sensitive to the text size and the resolution (dpi) of the input image. The runtime[2] for this algorithm is $O(n^3)$, where n is the initial number of eligible segments. This algorithm is summarized below:

```
allLines[] = //set of lines from segmentation step
lines[]    = //filter allLines with length < T_text
textRegions[] = // init with one text rectangle
                // for each line in lines
didMerge=true
while(didMerge) {
  didMerge=false
  for each t in textRegions
    for each t' in textRegions
      if (t != t' and t is close to t')
        t=merge(t,t')
        remove t' from textRegions
        didMerge=true
}
textRegions[] =
 //filter text regions of more than N segments
```

5.3.2 Corner Detection

Corner detection is accomplished by finding segments that end at approximately equal points and testing that the angle between them is approximately 90 degrees. Corners are then classified as “axis-aligned” or “non-axis-aligned”. A double-threshold is used so that a corner can be classified as both types if it is in between the threshold values

[2]There may be a tighter upper-bound, but as the Performance section (see below) describes, polynomial-time algorithms that operate in segment space are adequate.

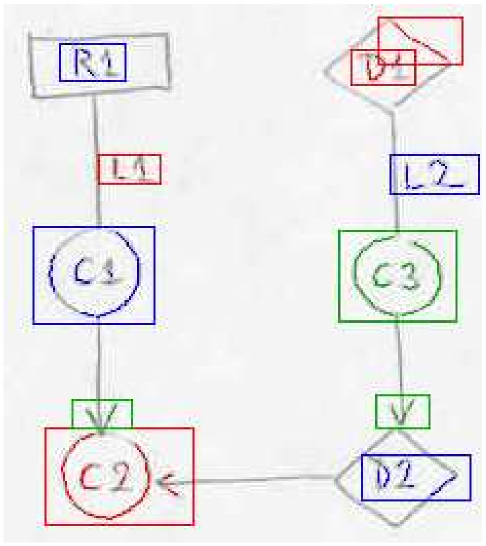


Figure 5-7: Recognition of text rectangles. Original image shown with recognized text rectangles (and constituent segments) drawn in.

(see Figure 5-8). This classification is useful so that the set of corners used to detect rectangles is disjoint with the set of corners used to detect diamonds. The runtime for this algorithm is $\theta(n^2)$, where n is the initial number of eligible segments.

5.3.3 Rectangle Recognition

Rectangles are detected using the following algorithm: if a corner is (approximately) horizontally aligned and (approximately) vertically aligned with two other corners, then a rectangle exists that spans the area of the three corners. This algorithm can generate false positives since it only requires three corners to produce a rectangle interpretation, but it generally produces all true positives. False positives are filtered out by the heuristics and/or greedy elimination algorithm described below. The approach to shape recognition, in general, follows the same approach: try to generate ALL true positives, even if it involves generating a fair amount of false positives as well. There are two reasons for doing this: (1) if the true positive is missed, then there is no way subsequent steps can ever achieve perfect recognition, and (2) users can more easily delete a false positive than create a (missed) true positive. The runtime for this algorithm is $\theta(n^3)$, where n is the number of recognized corners.

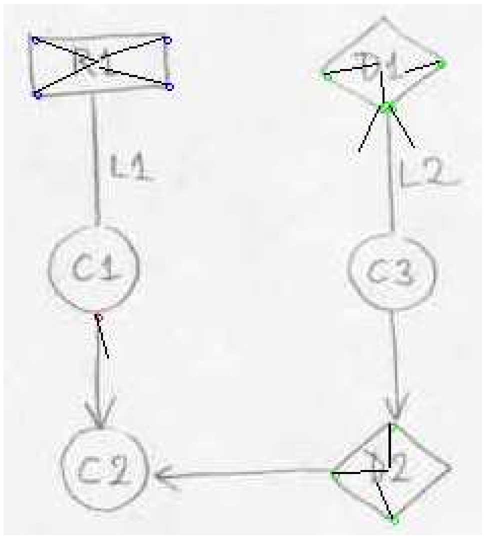


Figure 5-8: Results of corner detection. Corners that will only be used by the rectangle recognizer are shown in blue; corners that will only be used by the diamond recognizer are shown in green; corners that will be used by both recognizers are shown in red.

In order to improve the accuracy of the rectangle recognizer, two modifications to the simple algorithm described above were made. The first is described in figure 5-9. This figure shows two sets of corners. The set of (unfilled) red corners are aligned appropriately, but a rectangle is not recognized because the interpolated midpoint of each corner would not lie within the bounding box of the rectangle (one of the corners points outward). The set of (filled) green corners, on the other hand, will be used to construct a recognized rectangle. This rectangle, however, will not make it into the initial interpretation because a pre-filter is used. As shown in figure 5-10, when rectangles overlap, the smaller rectangle is chosen, and the larger, overlapping rectangle is discarded.

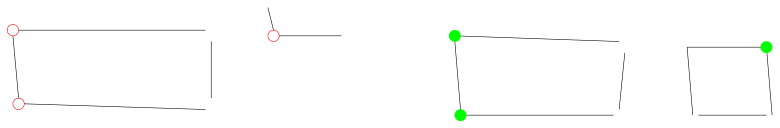


Figure 5-9: Description of rectangle recognition. The red corners constitute a rectangle that is not recognized because one of the corners is facing outwards. The green corners constitute a rectangle that is recognized.

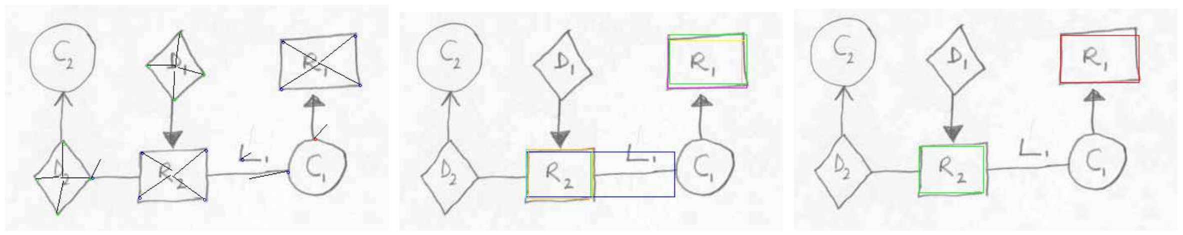


Figure 5-10: Results of rectangle detection. The images show the results of corner detection, initial rectangle recognition, and pre-filtering of overlapping rectangles, respectively.

5.3.4 Diamond Recognition

Diamond recognition is analogous to rectangle recognition: if a corner is (approximately) aligned with another corner along the y=x axis and is also (approximately) aligned with a corner along the y=-x axis, then a diamond exists that spans the area of the three corners. As with the rectangle recognizer, the diamond recognizer generates some false positives (though less frequently than the rectangle recognizer), but it generally generates all possible true positives, as well. Choosing among the false and true positives is deferred until the filtering stage. The runtime for this algorithm is $\theta(n^3)$, where n is the number of recognized corners.

5.3.5 Circle Recognition

Circle recognition is done by an augmented Hough transform algorithm. The major modification of the algorithm is that it uses segments (produced from segmentation) instead of edge pixels. The accounting step is borrowed from the Hough transform (to find center points and radius values), but instead of using gradient values, the algorithm interpolates along each segment's perpendicular bisector. This algorithm seems to work well in practice. To visualize what the algorithm is doing, imagine segmenting a circle, and then drawing infinitely long lines along each segment's perpendicular bisector. Areas of close intersection correspond to approximate circle center points, and the average distance from the center point to each of the segments is the approximate radius of the circle.

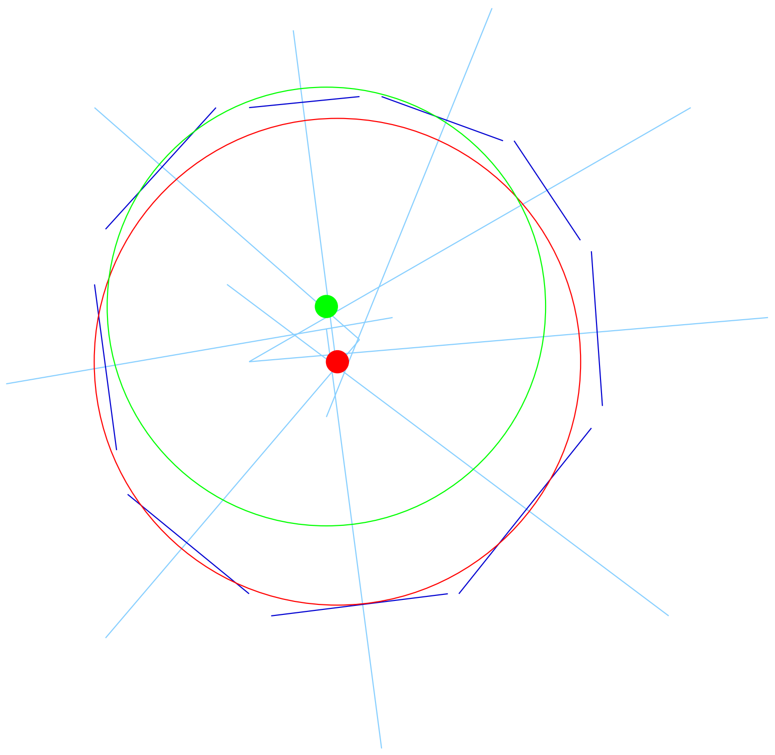


Figure 5-11: Description of augmented Hough transform. This algorithm recognizes circles by using perpendicular bisectors of constituent segments.

5.3.6 Arrow Classification

Arrow recognition is implemented using a modified version of the algorithm described by Hammond [3]. The algorithm is pruned for simplicity to recognize only line arrowheads and filled triangle arrowheads (but not open triangle arrowheads).

5.3.7 Performance

The algorithms described above are essentially search algorithms where the search space is the set of segments generated from the segmentation step. These search algorithms are not efficient, but we have found the inefficiency to be negligible in practice. The reason is that these algorithms search in segment space instead of pixel space, and there are orders of magnitude fewer segments than pixels in a sketched diagram. The time spent in the Gaussian filter and segmentation phases (which are algorithms that scale linearly with the number of pixels in the image) is approximately equal to the time spent in the remainder of the recognition module (where the algorithms scale quadratically or cubically with the number of features).

5.4 Global Interpretation

After the recognition phase, the system has generated a set of recognized shapes, segments, arrowheads, and text regions from the original image. The final step is to filter the set of recognized objects to create an initial interpretation and a set of alternative interpretations. Because all of the recognizers have a very low false negative rate (meaning that they rarely miss true positives), this step can be effective as purely a filtering process. There are two types of filtering that occur: (1) heuristic filtering that uses domain-specific information to eliminate global interpretations that do not satisfy domain constraints, and (2) greedy elimination to choose the best interpretation among competing local interpretations. These two types of filtering are discussed in greater detail below.

5.4.1 Heuristic Filters

Recognition rates improve with the use of *heuristic filters* – domain-specific filters that use domain-specific constraints to discard non-conforming interpretations. As an example, the domain of box-and-line diagrams does not allow overlapping shapes; therefore, if the shape recognizers independently recognize the same area to contain two different shapes, then a heuristic filter (representing the heuristic "shape areas cannot overlap") can choose the shape with higher confidence and relegate the other shape to an alternate interpretation. I have found three heuristics to be useful in increasing the recognition rates. These heuristics, however, have become less important as the shape recognizers (described above) have been tuned and the greedy elimination algorithm (described below). Nevertheless, the heuristic filters are kept in the system in order to maximize the probability of an adequate recognition.

The first heuristic filter implements the "shape areas cannot overlap" heuristic. Figure 5-12 graphically describes how this filter behaves. Shapes that overlap are not allowed in the domain; therefore, if shape areas overlap the system chooses the shape with a higher confidence value. Each of the recognizers, even though they are all feature-based, still assign a confidence value for each recognized shape. For some recognizers (e.g. Rectangle, Diamond), this value will be the same for all shapes; for others (e.g. Circle), the value will vary.

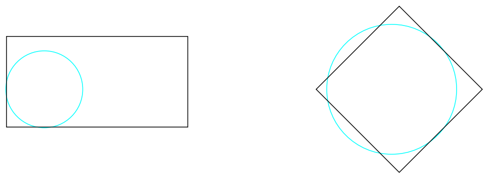


Figure 5-12: The "shape areas cannot overlap" heuristic filter. Here the circles are recognized with lower confidence values than the rectangle and diamond; therefore, they will not be included in the initial interpretation.

The second heuristic filter implements the "line segments cannot intersect shape areas" heuristic. This is similar to the previous filter, except that instead of comparing shape confidence value, it just looks for recognized line segments within recognized shape boundaries. Because of the way the greedy elimination algorithm behaves (see next section), and because of the imperfect results of the merging algorithm, this filter will "clean-up" so that no line segments are left around inside the boundaries of shapes. In figure 5-13, the cyan segments represent segments that will be thrown out by this heuristic filter. The dark blue segments will be thrown out during greedy elimination because they are part of the corners that make up the recognized diamond.

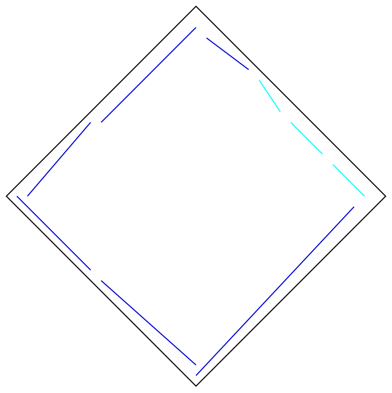


Figure 5-13: The "line segments cannot intersect shape areas" heuristic filter.

The final heuristic filter implements the "line segments cannot intersect text rectangles" heuristic. This heuristic ensures that mid-length segments that are inside text rectangles are not included. In pilot tests, users reported that they had trouble seeing these extra segments and that they were frustrated that they had to delete them.

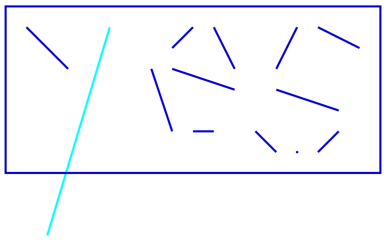


Figure 5-14: The "line segments cannot intersect text rectangles" heuristic filter.

These heuristic filters were simple to create, and they can be easily disabled (to allow for diagrams in a different domain). The fact that they are helpful at all means that the lower-level recognizers are not optimal, and that more work could be done to produce better filtering logic that does not rely on these hard-coded heuristics. For future work, developing perceptual filters using Gestalt principles might help for this kind of shape recognition.

Some of these filters are executed before the greedy elimination algorithm, while others are executed after. The before/after decision was made after looking at the results from the pilot study, and analyzing when the filters would be most effective. For example, the line segments cannot intersect shape areas heuristic filter is executed after greedy elimination because at that point the system is most confident about the set of shapes remaining. It would not make sense to execute that heuristic filter before greedy elimination because a shape that would otherwise be eliminated during greedy elimination would be used to incorrectly eliminate line segments.

5.4.2 Greedy Elimination Algorithm

The final stage of the sketch recognition module selects the best global interpretation and generates alternative interpretations. Each of the shape recognizers described above proceeds independently, and assigns a confidence value when it recognizes a shape, as well as marking each recognized shape with the constituent segments that were used to form the hypothesis. A straightforward greedy elimination algorithm is used that first sorts all interpreted shapes by confidence and then selects shapes as part of the best interpretation as long as a previously selected shape does not share any segments with this shape.

This algorithm can clearly be modified along several interesting dimensions. Instead of greedily eliminating a shape that shares any segments with a previously chosen shape, the algorithm can eliminate a shape only if all of its segments are shared with previously recognized shapes. There are also interesting possible steps to take if only some of the segments are not shared by previously chosen shapes. This analysis is left for future work.

The greedy elimination algorithm provides an effective filter among the false and true positives because, in general, the true positives are recognized with higher confidence levels. They are therefore selected first, and the false positives are eliminated because their segments overlap with previously chosen shapes.

5.5 Alignment Module

Once the recognition module has generated interpretations of the sketch, but before loading the initial interpretation into the graphical user interface, the application attempts to sensibly align the components. The purpose of this alignment module is to make lines and shapes that the user has *approximately* aligned in her original sketch *exactly* align in the initial interpretation. We developed heuristics for when lines close to shapes actually represent *connection points*, and also found thresholds for when to decide to make lines vertical/horizontal.

Because it is important to maintain connection point relationships while aligning lines and shapes, the alignment module must efficiently solve systems of linear constraints. The UDSI application uses the simplex solver in Cassowary [1]. The following are a sample set of constraints automatically generated (and then solved) to align a line segment connecting two shapes (the variables are described in Figure

5-15):

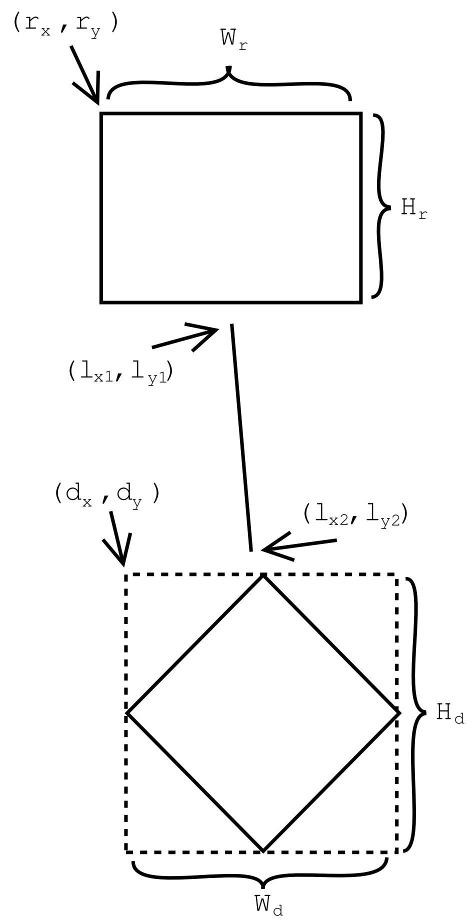


Figure 5-15: Visual description of alignment module.

$$(r_x + W_r/2, r_y + H_r) = (l_{x1}, l_{y1}) \tag{5.3}$$

$$(d_x + W_d/2, d_y) = (l_{x2}, l_{y2}) \tag{5.4}$$

$$l_{x1} = l_{x2} \tag{5.5}$$

$$l_{y2} - l_{y1} = L \tag{5.6}$$

The equations above represent the following constraints: (1) One end of the line must remain at the midpoint of the bottom edge of the rectangle, (2) the other end of the line must remain at the midpoint of the top of the diamond, (3) the line must be vertical, (4) the length of the line must be constant. These four constraints together, when solved, align the shapes and line in such a way that the vertical line connects the diamond to the rectangle at the specific connection points. Figure 5-16 shows a recognized sketch without automatic alignment of shapes and lines, while figure 5-17 shows the results of executing the alignment module. In general, when the system detects connection points, the automatic alignment seems to have a positive impact on the quality of the image. This is discussed more in chapter 6.

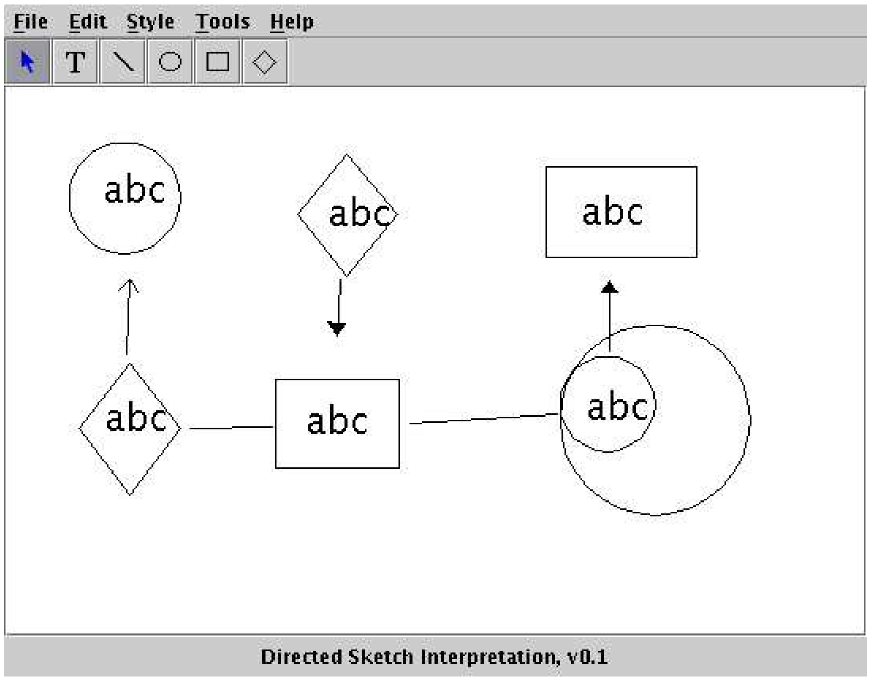


Figure 5-16: Recognized sketch without automatic alignment of shapes and lines.

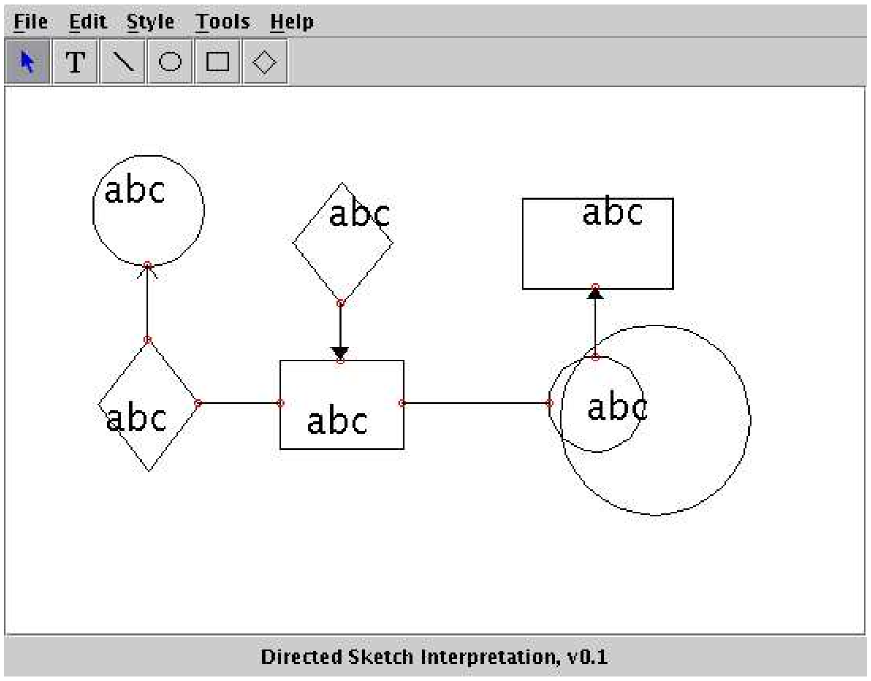


Figure 5-17: Recognized sketch with automatic alignment of shapes and lines.

Chapter 6

Evaluation

This section describes the results from two separate evaluations of the UDSI system. The first is a controlled experiment that attempted to compare and contrast the effectiveness of the system versus an existing structured graphics editor. The second evaluation estimated the accuracy of the recognizers of the system, using the corpus of images collected from various user studies.

6.1 User Study

I carried out a user study to evaluate the utility of UDSI and to compare it against an existing commercial application, Microsoft PowerPoint. I recruited users by advertising on campus, and users were compensated for their participation.

6.1.1 Usability Indicators

There are two main parameters of usability that I wished to measure. The first is simply the amount of time it would take users to create a diagram that was described to them. This time would be measured for both PowerPoint and UDSI, and the tasks were designed so it would be possible to use a within-subjects comparison of time taken. The second parameter is the "cleanliness" of the diagram that was produced by the user. This qualitative parameter should measure the quality of the layout, alignment, and spacing of the diagram. I created a controlled experiment that measured these two parameters for both interfaces (UDSI and PowerPoint).

6.1.2 Controlled Experiment Design

All users filled out a pre-test questionnaire (see Appendix A) that asked their computing experience and artistic background. The small sample contained users of diverse backgrounds and abilities (see User Profile section below).

After the pre-test questionnaire, the users completed three tasks. The first task was a warm-up task that allowed me to describe the syntax and semantics of the diagram language that would be used for the remainder of the study. This command language describes diagrams that the user was instructed to create. This language is as simple and domain-independent as possible so users would not spend time struggling with translating the textual description into a diagram. The warm-up exercise asked the user to sketch (with pen and paper) the diagram described by the following commands:

```
Circle C1
Rectangle R1
Diamond D1
C1 points to R1
R1 is connected to D1 with label L2
```

There are two commands to describe connections between shapes: "points to" means that an arrow should be drawn from the receiver to the operand, while "is connected to" means that a line segment should be drawn between the shapes.

After the user completed the warm-up task, the users completed the next two tasks in a random order. The two tasks both involved creating more complex diagrams (described in the same language discussed above) using either PowerPoint or UDSI. Users were told to work quickly and accurately because they would be timed, and also that they should make their diagrams as crisp and clean as possible. I suggested they imagine that their resulting diagram would be used in a business presentation slide. For the UDSI condition, the user was first told to sketch a clean version of the diagram using pen and paper (and to iteratively create sketches to make the pen and paper sketch as crisp and clean as possible). The user then handed in the paper, which I then scanned and cropped; finally, UDSI loaded the scanned image. The decision to scan and crop the image for the users was made to normalize the amount of time spent between the Sketch stage and Application stage, since the controlled experiment needed to compare the time spent using the two interfaces. When UDSI loaded, the user was then presented with an initial interpretation of her sketch, generated from the scanned image. The remainder of the task involved cleaning up this diagram: fixing recognition errors, aligning components, and editing text regions. Because the system only recognizes text regions and does no character recognition, users needed to edit each text region to contain the appropriate text.

I hypothesized that (1) the users would create cleaner diagrams (better alignment, layout, and spacing) using UDSI as compared to PowerPoint, and (2) that the users would create the diagrams in less time.

To test these hypotheses, I stored all diagrams produced in every task by every user. To test hypothesis (1), I recruited a set of judges to subjectively evaluate a sample of the diagrams produced along various qualitative dimensions. To test hypothesis (2), I simply measured the amount of time it took to complete each task. For the PowerPoint task, I measured the total time it took the user to complete the diagram. For the UDSI task, I measured the amount of time it took the user to create the pen-and-paper sketch, the time it took to scan the image, and the time it took the user to edit the recognized sketch in UDSI. I measured the times of the UDSI sub-tasks separately so I could remove the scan time from the time comparison.

6.1.3 User Profile

Users were recruited by advertising on campus[1]. The set of 22 users contained a mix of men and women, students and non-students, and people with a variety of artistic and technical backgrounds. One user did not have enough of a computing background

[1] `free-money@mit.edu` provided an excellent source of subjects.

parameter	sample set characteristics
gender	men=11,women=11
age	$\mu = 23.3, min = 19, max = 33$
PowerPoint experience	$none = 2, little = 5, some = 10, lots = 5$
Artistic training	$none = 13, little = 6, some = 2, lots = 0$

Table 6.1: Characteristics of users.

to complete either task; therefore there are only 21 users are included in the analysis. The user characteristics are summarized in the table in figure 6.1.

6.1.4 Subjective Variables

Regressing the subjective post-test responses from users on the amount of time it took to complete the task (dependent variable) on the actual amount of time it took to complete the task (independent variable) gives an R^2 of 0.25. This suggests that users were able to somewhat accurately perceive the amount of time it took to complete each task, and correctly perceived which task took longer. This suggests that users will notice when the user interface allows them to complete their task in less time.

6.1.5 Experimental Results

The results of the user study are given in figure 6.2. The time for the UDSI task is separated into three sub-tasks. *Sketch* is the time spent sketching the diagram with pen and paper (including erasing/iterating); *scan* is the time spent scanning/cropping; and *edit* is the time spent using the UDSI application (including load time and recognition time). The sum of all three times is also given, as is the time of only the first and third time, *sketch+edit*. All values in the table are given in seconds.

First, some overall comments regarding the first two columns: the results show that users, on average, spent less total time creating their diagrams using PowerPoint than using UDSI ($326.3 < 443.5$). Subtracting out the scan times, users still spent less time completing the PPT task ($326.3 < 368.3$). The sample variance of the UDSI times, however, is larger than the sample variance for PowerPoint users ($\{155.6, 160.5\} > 119.8$). In particular, the UDSI task times for 4 users are very large

task	All users		-4 outliers	
	μ	σ	μ	σ
PPT total	326.3	119.8	318.4	122.1
UDSI sketch	142.5	79.5	127.6	66.9
UDSI scan	75.2	27.5	79.5	28.5
UDSI edit	225.9	122.9	186.1	84.8
UDSI total	443.5	155.6	393.1	109.7
sketch + edit	368.3	160.5	313.7	107.4

Table 6.2: Results from user study. All times are in seconds.

relative to those users' PowerPoint task times.

All four of these users uncovered severe usability flaws in the UDSI interface that swamped their UDSI task times. These errors are discussed in more detail in the next section, and the results with these four users removed are also presented. The "-4 outlier" case, however, is not used for much of the statistical analysis; for one thing, it's unfair. There are two other users, after all, who saved large amounts of time ($>$ 70 s.) using USDI because they experienced serious mode errors using PowerPoint. As discussed in the end of this subsection and in the Future Work section, a future study might attempt to normalize for this by using the same structured graphics editor in both tasks. The "-4 outlier" case, however, shows UDSI task times slightly closer to the PPT times. Under both the "All users" sample and the restricted "-4 outliers" sample, a two-tailed t-test fails to reject the null hypothesis at the 1%,5%, and 10% confidence values (t-value for "all-sample" is -1.597, critical value for 5% is -2.086). I therefore conclude that there is no statistically significant difference in PPT times and UDSI task times.

It is important to note that 10 out of the 21 users saved time using UDSI. For these users, the system recognized the bulk of the shapes, arrows, and segments/connectors, with few errors. The users quickly corrected these errors and edited the text regions, and were generally satisfied with the automatic alignment done by the system. As mentioned above, however, the differences in execution time among these users is not statistically significant.

From analyzing the data, I noticed that users who indicated "none" or "little"

PowerPoint experience saved time using UDSI, in general. Results for this set of users with little or no PowerPoint experience, are given in figure 6.3.

	MS PPT	UDSI					(total) -
#	total	sketch	scan	edit	total	sk+ed	(sk+ed)
5	629	208	59	375	642	583	46
7	368	87	61	239	387	326	42
15	369	101	73	143	317	244	125
16	505	307	95	121	523	428	77
19	303	183	81	257	521	440	-137
22	404	212	82	130	424	342	62
μ	429.7					393.8	
σ	118.0					117.3	

Table 6.3: Descriptive results of users with little or no PowerPoint experience.

This table shows users saving, on average, more than 30 seconds using UDSI. Also, 5 out of the 6 users in this sample saved time using UDSI (positive numbers in the rightmost column correspond to time saved in the UDSI task). It is a small sample set, however, and no statistical conclusions can be drawn from it (a one-sided t-test is not significant at even the 10% level; the t-statistic is 0.977 and the 5% critical value is 2.57.

Taken together, these results suggest that one must reject hypothesis (1) that asserted that UDSI would save time for users. The second hypothesis was that users would produce higher-quality diagrams. We evaluated this by recruiting judges to evaluate the diagrams produced by the users. As with the user study, judges were recruited by advertising on-campus and compensating the judges for their time. The only requirement for participation was that the potential judge did not participate in the original user study. The judges evaluated the diagrams produced from the PPT and UDSI tasks along 4 qualitative dimensions: alignment, spacing, layout, and overall quality. The judges were not told what system was used to produce each diagram, and every effort was made to conceal consistent differences between images (for example, I took each PowerPoint diagram produced during the user study and matched the fonts with the default font used in UDSI). The results of the 3 judges are given below in figure 6.4.

dimension	PPT	UDSI	t-statistic	two-tailed $P(T <= t)$
alignment**	3.00	3.89	-2.204	0.042
spacing	3.78	3.89	-.306	0.726
layout	3.33	4.17	-1.34	0.197
overall quality*	3.22	3.89	-1.80	0.090

Table 6.4: Results from judging of user diagrams. All values are averages and are drawn from a 1-5 scale. *t-test significant at the 10% level **t-test significant at the 5% level

These values show that the judges preferred diagrams produced in UDSI along all qualitative dimensions. The "overall quality" measure is statistically significant at the 10% level, with only 18 observations (17 degrees of freedom). These results suggest that users create higher-quality diagrams in UDSI versus PowerPoint. The "alignment" measure is also statistically significant at the 5% level. Also, all t-statistics are negative, meaning that judges, in general, thought that the quality of the diagrams produced using UDSI was higher than diagrams produced using PowerPoint.

6.1.6 Analysis of Errors

For one user, there was a large usability flaw in the UDSI interface for editing text regions that caused this user to be unable to figure out how to edit the default text in text regions. The errors resulting from this serious usability flaw (which was quickly fixed for subsequent users) swamped the time spent correcting recognition errors and aligning components using UDSI. For another user, the initial interpretation was very poor (the UDSI system did not recognize most of the shapes in the diagram), and therefore the user spent a lot of time re-creating shapes that the system failed to detect. This was caused by a bug in the circle recognizer that did not generate true positives in a certain size range.

It is also interesting to note that I observed mode errors in both PowerPoint and UDSI relating to creating and selecting objects. In UDSI, I tried to reduce these mode errors by intelligently switching the mode based on some heuristics derived from observing users (e.g. if a user clicks to place a shape inside another shape, UDSI switches over to selection mode instead of placing an identical shape inside an existing shape). The changes, however, did not seem to significantly improve the frequency of mode errors.

6.1.7 Discussion

Though it doesn't appear that UDSI saves users' time, the comparison of the quality of diagrams produced using UDSI to those produced using PowerPoint revealed that the diagrams produced using UDSI were higher quality. A user's PowerPoint diagram is shown in Figure 6-1, and that same user's UDSI diagram is shown in Figure 6-3. In general, the UDSI diagrams were better aligned (because of the automated alignment), and they often had a clearer layout. Because some of the users iteratively sketched copies of the diagrams during the UDSI task, they were able to choose a better layout. Observing users laying out the diagram within PowerPoint, describing layout was very time-consuming, and users appeared to just stop and give up when their image no longer had intersecting segments and connectors.

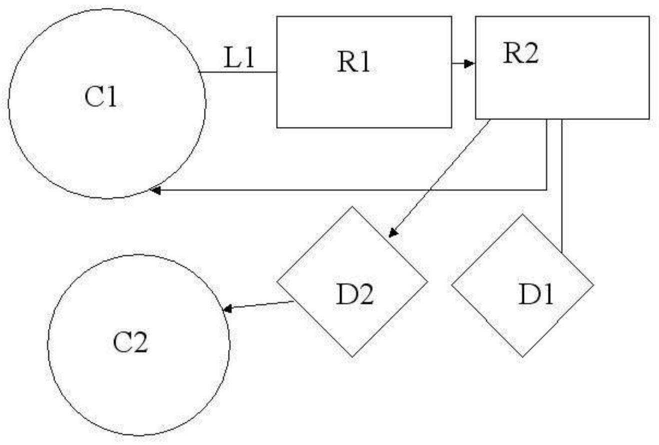


Figure 6-1: PowerPoint diagram from user 6.

Finally, note that only one user used the novel user interface for selecting alternative interpretations. I propose two explanations: (1) that users are more comfortable with the Delete and Create task, and that no amount of automatic assistance will lead them away from that plan, or (2) that users do not really notice or comprehend

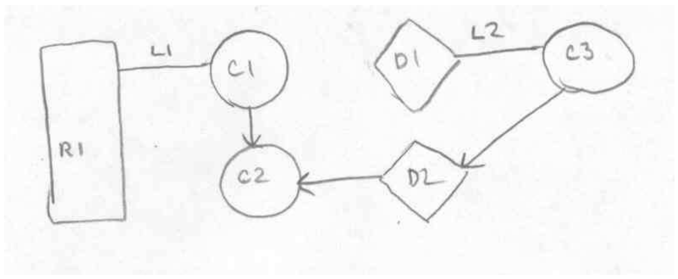


Figure 6-2: Pencil sketch from user 6.

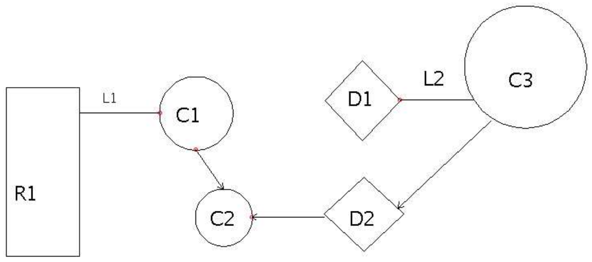


Figure 6-3: UDSI diagram from user 6.

the popped-up panels of alternative interpretations because these alternatives are presented outside of the user's locus of attention. Recognizing this, the user interface has been modified so that when a user clicks on a shape for which the system has multiple interpretations, a line is drawn from that shape to the panel representing the multiple interpretation. This was implemented using the Magpie toolkit [4].

6.2 Analysis of Recognizers

Using the sketches collected from pilot studies and the user study, I used the set of 25 image bitmaps to analyze the accuracy of the recognition module. This section discusses how I calculated the accuracy values and discusses the results.

6.2.1 Recognition Accuracy

For each image, I loaded the image bitmap into UDSI and counted the total number of shapes that were recognized by UDSI but did not appear in the original sketch, as well as the total number of shapes that appear in the original sketch but do not appear in the recognized structured diagram. Table 6.5 presents the false positive and false negative rates for shapes, lines, and text regions. The false positive rates f_p are calculated by dividing the number of false positives (shapes that appear in the recognized diagram but did not appear in the original sketch) by the total number of shapes that appear in the original sketch. The false negative rates f_n are calculated by dividing the total number of shapes that were not recognized by the system (but occurred in the original sketch) by the total number of shapes that appear in the original sketch. These values are calculated for each image in the sample, and the averages and standard deviations of these values are reported in table 6.5.

dimension	f_p		f_n	
	μ	σ	μ	σ
shapes	.102	.129	.073	.108
lines	.254	.309	.032	.095
text regions	.707	.273	.021	.050
overall	.358	.182	.043	.064

Table 6.5: Results of analysis of recognition module.

The chart in figure 6-4 displays the false positive and false negative rates for each image in the sample. This chart re-iterates that the false positive rates are much higher than the false negative rates. Also, this chart shows that the false negative rates are zero for several images. When the false negative rate is zero, this means that any time a shape, line, or text region appeared in the original sketch, it also appeared in the recognized diagram. This does not mean that recognition was perfect – indeed, recognition was never perfect because every recognized diagram contained false positives – but it does mean that filters can potentially filter the recognized diagram to produce a correct diagram.

6.2.2 Discussion

Both kinds of errors from the recognizers (false positives and false negatives) require effort from the user to correct the error. In the case of false positives, the user must

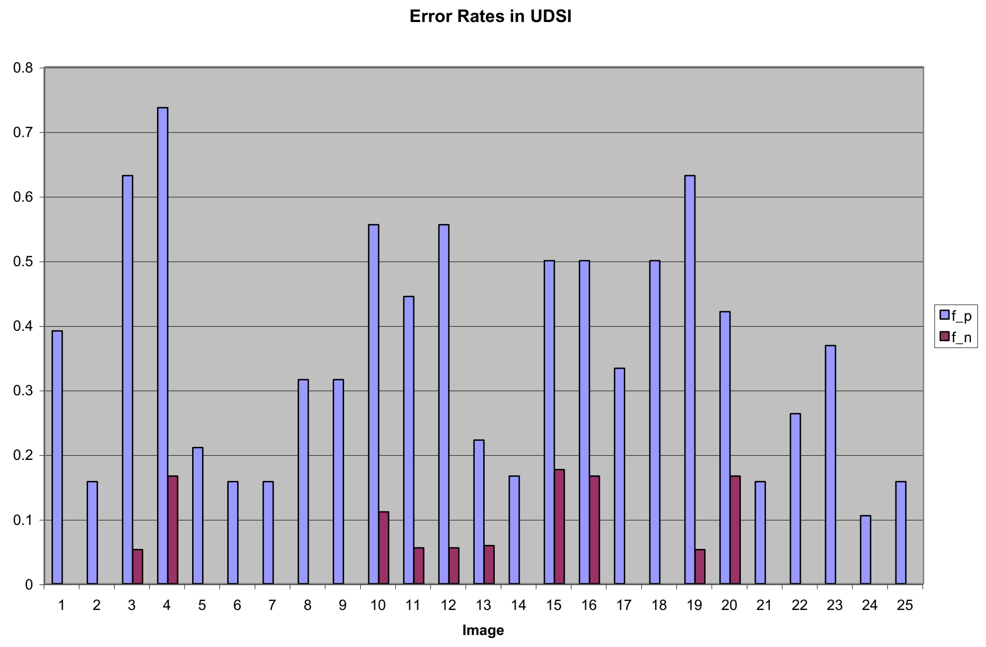


Figure 6-4: Accuracy of recognition module. The blue bars represent the false positive rate, f_p; the red bars represent the false negative rate, f_n.

delete the shape from the diagram. In the case of false negatives, the user must create the shape at its appropriate location. The total amount of effort that the user must exert to correct the recognized diagram is a weighted sum of the false positive correction rate and false negative correction rate. Future work can attempt to actually estimate the weights, but from observations made during the user study, I believe that false positives are easier to correct than false negatives.

The actual error rate averages presented above show that the false negative rates are, in general, much lower than the false positive rates. This means that the system rarely misses shapes that occur in the original sketch, but it generates a fair amount of false positives (that must be deleted by the user). Text recognition is particularly bad; for every text region correctly recognized, there are .707 text regions that are also recognized, but do not appear in the original diagram. This suggests that better recognition and better filtering are needed to improve the performance of the text recognition module. One possible improvement would be to include more domain-specific information (for example, a heuristic that text regions cannot intersect shapes and/or lines). During the user study, however, most users simply deleted these unnecessary text regions. Also, note that the text recognizer generates a large number of false positives, but it has the lowest rate of false negatives.

Overall, the false negative rate for all recognizers is quite low. The system was intentionally designed to have a low false negative rate because the task analysis concluded that it is easier for users to remove existing components than to create new components. Therefore, it was very important that the recognizers rarely miss true positives. Through the use of heuristic filters and the greedy elimination algorithm, the recognition module could filter out false positives before presenting the results to the user. It would not be possible given the recognition architecture, however, to produce a correct interpretation of all of the shapes in the original image if the system did not first generate all true positives.

One (simplified) way to look at these data is that the false negative values represent the quality of the recognizers while the false positive values represent the quality of the filters. Since the low-level recognizers will often be confused without any domain-specific information, one would only ask that they generate very few false negatives. Since the filters should have domain-specific information to help arrive at a reasonable global interpretation, one would expect them to filter out as many false positives as possible.

Chapter 7

Conclusion and Future Work

I presented a novel approach to creating structured diagrams by combining an off-line sketch recognition system with the user interface of a traditional structured graphics editor. The current system recognizes geometric shapes, lines, text regions, and arrowheads, and it provides a user interface to modify interpretations and edit components. For future work, there are several possible paths. The remainder of this section discusses these options in greater detail.

7.1 A Vision-Based On-Line Sketch Understanding System

It would be interesting to evaluate the effectiveness of UDSI's vision-based approach of recognition within an on-line system on an instrumented tablet computer. Since the system is not on-line, there is no need for image acquisition (no scanning/cropping time), and it will also allow UDSI to provide some real-time feedback if users begin to draw shapes that the system will be unable to recognize at all (a constant risk of the purely off-line approach). Bringing this feedback on-line will hopefully increase recognition rates even further.

UDSI should work immediately as an on-line system that just uses pixels created on an instrumented tablet computer as line pixels, and dispenses with all early-stage image processing (Gaussian filtering, Canny line detection). An on-line UDSI system would pass the stroke-created input image directly to the contour following step (see Recognition chapter) and proceed from there.

This approach would differ in some interesting ways from many existing on-line systems (see the Related Work section for a discussion of existing on-line systems). Since the recognition module is vision-based, the stroke ordering would be completely unimportant. In fact, the strokes themselves would not be important, only the resulting pixels created from the stroke. An on-line UDSI would also allow users to erase in the middle of a previously drawn stroke to create two distinct lines. All reviewed on-line systems use strokes as primitives, so that erasing a previously drawn stroke removes the entire stroke from the image.

7.2 Character Recognition

Several of the users in the user study wrote that they would be more inclined to use UDSI regularly if UDSI explicitly recognized sketched text, instead of merely recognizing text regions. Given sufficiently accurate character recognition, this would allow users to avoid having to type in character into text regions. Although I was not able to find suitable off-the-shelf character recognition systems to use, since a handful of users expressed a desire for UDSI to include it, it would be worth incorporating into UDSI.

7.3 Pre-segmentation Image Processing

The recognition results are sometimes sensitive to the image pre-processing (Gaussian filtering) that is done before the Canny line detection. It would be interesting to investigate if it is possible to determine an adequate filter, *a priori*, by scanning the original image. The ScanScribe [13] system uses a novel foreground separation algorithm that would likely be useful for this purpose, as well.

Also, I only investigated using a scanner for image acquisition. As shown in figure

7-1, digital cameras can also be used to acquire images of sketched diagrams. The thresholding characteristics of these images are quite different than scanned pen-and-paper sketches, and therefore would not be immediately usable within UDSI.

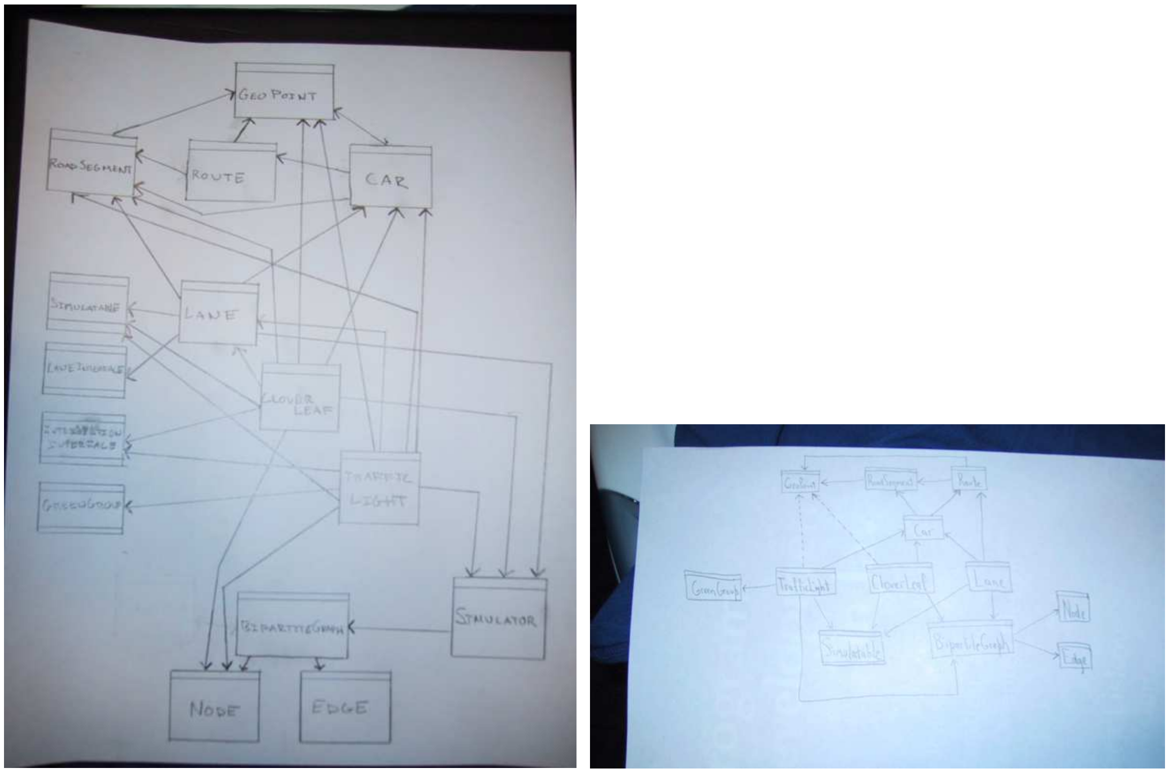


Figure 7-1: Images of sketched diagrams acquired using digital cameras.

7.4 Other Usability Experiments

One experiment that would be very insightful would attempt to measure the harm done by a PowerPoint diagram that was poorly created. This experiment would ask students to create a diagram in PowerPoint (much like the task in the user study presented in this thesis), but then it would ask them to change the diagram. The changes would involve moving shapes, connections, and modifying text. The hypotheses I have are the following: (1) that this process will take a long time, perhaps even close the amount of time it took to originally create the diagram, and (2) using connectors and grouping components in the initial creation step will speed the time of the second task.

7.5 Comparing Circle Recognizer to Hough Transform

UDSI recognizes circles using an augmented Hough transform algorithm. It would be interesting to quantitatively compare the performance of this augmented algorithm with the standard Hough transform algorithm. In the small sample set of images collected in the user study, the augmented algorithm has equal specificity, but a substantial increase in sensitivity (that is, much fewer false positives). This result should be verified in a larger experiment.

7.6 Extending UDSI to Different Diagram Domains

Lastly, to test the robustness of UDSI, it is important to consider other diagram domains besides simple box-and-line diagrams. UML diagrams, circuit diagrams, and simple chemical structures are potential domains to consider. To allow UDSI to recognize sketches of these domains, new recognizers and filters would need to be created and plugged into the recognition framework.

Appendix A

User Study Materials

This appendix contains the following materials used in the user study:

- The pre-test questionnaire
- The description of the diagram syntax and the set of tasks
- The post-test questionnaire

PRETEST QUESTIONNAIRE

1. How would you describe your level of experience with the following applications?

Program	**None** (never used it, or don't know)	**Little** (used once or twice)	**Some** (used enough to be comfortable)	**Lots** (use regularly)
Microsoft Visio				
Adobe Illustrator				
XFig				
Dia				
Corel Draw/Corel Flow				
Microsoft PowerPoint				
Microsoft Paint				

2. How would you describe your level of experience with the following features included in some drawing programs?

Feature	**None** (never used it, or don't know)	**Little** (used once or twice)	**Some** (used enough to be comfortable)	**Lots** (use regularly)
Multiple Selection				
Move-to-Front				
Connection Points				
Lasso Selection				
Snap To Grid				

3. What is your formal training or job experience in art, drafting, or other drawing on paper?

______ None

______ Little

______ Some

______ Lots

4. Your job is: ____

If you are a student, your major is: ______________________________

5. Is English your first language? ______ Yes ______ No

6. Your gender is: ______ Female ______ Male

7. Your age is: ______

Overview of Diagram Syntax

For this study, we will be using a simple syntax to describe the diagrams in text that we would ike you to create. The syntax is the following:

`Circle myLabel1` - draw a circle with the label, *myLabel1*, inside.
`Rectangle myLabel2` - draw a rectangle with the label, *myLabel2*, inside.
`Diamond myLabel3` - draw a diamond with the label, *myLabel3*, inside.

`myLabel1 is connected to myLabel2` - draw a line connecting the shapes labeled myLabel1 and myLabel2
`myLabel1 is connected to myLabel2 with label myLabel3` - draw a line connecting the shapes labeled myLabel1 and myLabel2 and label that line segment with myLabel3

`myLabel1 arrow points to myLabel2` - draw a line with an arrowhead from the shape with label myLabel1 to the shape with the label myLabel2
`myLabel1 filled points to myLabel2` - draw a line with a filled triangle arrowhead from the shape with label myLabel1 to the shape with the label myLabel2

KEY:

- connected to: ———
- arrow points to: ←———
- filled points to: ◀———

Tasks

Task 1: Warm-Up

Using the provided pencil and paper, draw a clean sketch of a diagram according to the following description. Feel free to sketch a draft quickly, and then sketch a cleaner draft. Let the investigator know when you have completed this task.

```
Circle C1
Rectangle R1
Diamond D1
C1 filled points to R1
R1 is connected to D1 with label L2
```

Task 2: Using Microsoft PowerPoint

Working quickly and accurately, create a diagram according to the following description within Microsoft PowerPoint. You should work to make the diagram crisp and clean. You should imagine this diagram is going to be used in a business presentation slide.

```
Circle C1
Rectangle R1
Rectangle R2
Diamond D1
Diamond D2
Circle C2
C1 filled points to R1
R2 is connected to C1 with label L1
R2 is connected to D2
D2 arrow points to C2
D1 filled points to R2
```

Task 3: Using UDSI

Working quickly and accurately, create a diagram according to the following description using UDSI. First, you should create a clean sketch of the diagram according to the description. You will then place your diagram on the scanner and press the leftmost button on the scanner. Your image should be loaded in UDSI, and you may then complete the task within the application. You should imagine this diagram is going to be used in a business presentation slide. (NOTE: when sketching on paper, try to use straight lines, but feel free to allow the lines to cross)

```
Rectangle R1
Circle C1
Circle C2
Diamond D1
Diamond D2
Rectangle R2
C1 filled points to R1
D1 filled points to R1
C1 is connected to C2 with label L1
D2 is connected to C2
D2 arrow points to R2
```

POSTTEST QUESTIONAIRE

1. Please rate the two interfaces for creating diagrams on **ease-of-use**:

Interface	**Very Hard**	**Somewhat Hard**	**Average**	**Somewhat Easy**	**Very Easy**
UDSI					
Commercial Application					

2. Please rate the two interfaces for **time taken** to complete the tasks:

Interface	**Very Long Time**	**Somewhat Long Time**	**Average Time**	**Somewhat Short Time**	**Very Short Time**
UDSI					
Commercial Application					

3. Please rate the two interfaces based on your overall experience and **satisfaction**:

Interface	**Very Unsatisfied**	**Somewhat Unsatisfied**	**Average Satisfaction**	**Somewhat Satisfied**	**Very Unsatisfied**
UDSI					
Commercial Application					

4. If available for your regular use, would you use the application for your own tasks?

Interface	**Very Unlikely**	**Somewhat Unlikely**	**Neutral**	**Somewhat Likely**	**Very Likely**
UDSI					
Commercial Application					

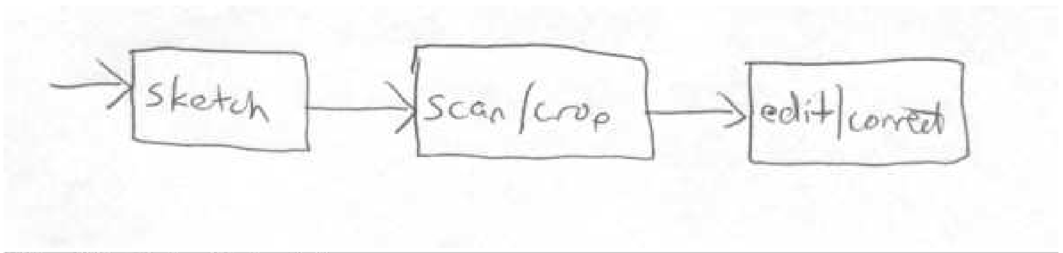
5. How much did you **trust** that the UDSI application would correctly interpret your sketch?

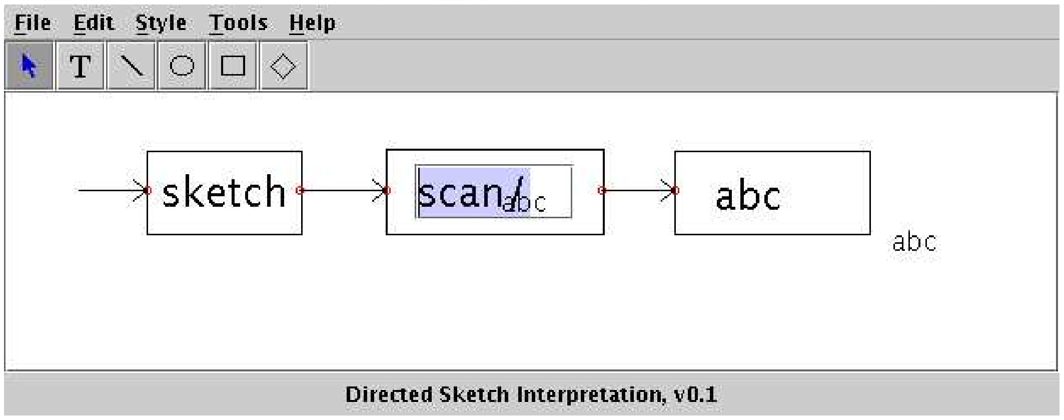
<u>Interface</u>	**Very Distrusting**	**Somewhat Distrusting**	**Neutral**	**Somewhat Trusting**	**Very Trusting**
UDSI					

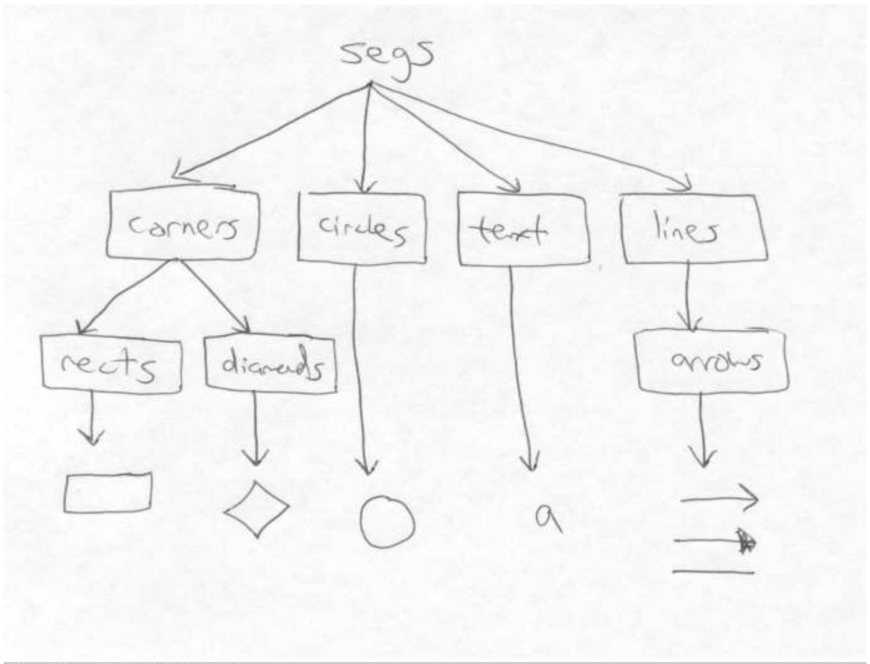
Appendix B

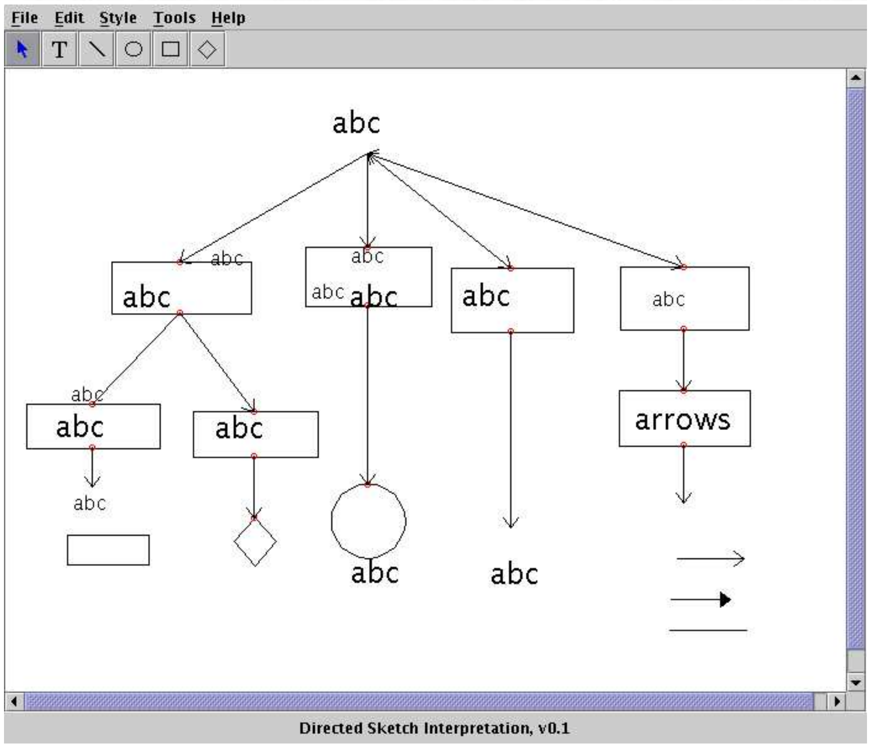
Sketches and Screenshots

Two diagrams in this thesis were produced using UDSI (figures 4-1 and 5-6). The original pencil sketches of these diagrams and the screenshots containing the initial interpretations are included here to give a sense of the performance of the UDSI system.


sketch
scan|crop
edit|correct


File
Edit
Style
Tools
Help
T
sketch
scan/abc
abc
abc
Directed Sketch Interpretation, v0.1


segs
corners
circles
text
lines
rects
diamonds
arrows
a


File
Edit
Style
Tools
Help
T
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
abc
arrows
abc
abc
abc
Directed Sketch Interpretation, v0.1

Bibliography

[1] G. Badros, A. Borning, and P. Stuckey. The Cassowary linear arithmetic constraint solving algorithm. *ACM Transactions on Computer Human Interaction*, December 2001.

[2] J. T. Hackos and J. C. Redish. *User and Task Analysis for Interface Design*. McGraw-Hill, 1998.

[3] T. Hammond. Nautral sketch recognition in UML class diagrams. Technical report, Proceedings of the MIT Student Oxygen Workshop, July 2001.

[4] D. F. Huynh, R. C. Miller, and D. R. Karger. Breaking the window hierarchy to visualize UI interconnections. *Submitted to UIST*, 2004.

[5] R. Jain, R. Kasturi, and B. G. Schunck. *Machine Vision*. McGraw-Hill, 1995.

[6] J. A. Jorge and M. J. Fonseca. A simple approach to recognize geometric shapes. *GREC, LNCS 1941*, 1999.

[7] J. Landay. Silk: Sketching interfaces like krazy. *Conference companion on Human factors in computing systems: common ground*, April 1996.

[8] E. Lank. Interactive system for recognizing hand drawn UML diagrams. *Proceedings for CASCON 2000*, 2000.

[9] J. Mankoff, G. D. Abowd, and S. E. Hudson. Oops: A toolkit supporting mediation techniques for resolving ambiguity in recognition-based interfaces. *Computers and Graphics (Elsevier)*, December 2000.

[10] J.-Y. Ramel. A structural representation adapted to handwritten symbol recognition. *GREC, LNCS 1941*, 1999.

[11] E. Saund, D. Fleet, D. Larner, and J. Mahoney. Perceptually-supported image editing of text and graphics. *Proceedings for ACM UIST*, 2003.

[12] E. Saund and E. Lank. Stylus input and editing without prior selection of mode. *Proceedings for ACM UIST*, 2003.

[13] E. Saund and T. Moran. A perceptually supported sketch editor. *Proceedings for ACM UIST*, 1994.

[14] T. M. Sezgin and R. Davis. Early processing in sketch understanding. *Proceedings of 2001 Perceptive User Interfaces Workshop*, 2001.

[15] S. Smithies. A handwriting-based equation editor. *Proceedings for Graphics Interface*, June 1999.

[16] E. Valveny and E. Marti. Deformable template matching within a bayesian framework for hand-written graphic symbol recognition. *GREC, LNCS 1941*, 1999.

[17] R. Zannibi. Aiding manipulation of handwritten mathematical expressions through style-preserving morphs. *Proceedings for Graphics Interface*, June 2001.