Interactive Sketching for the Early Stages of User Interface Design

James A. Landay and Brad A. Myers
Computer Science Department
Carnegie Mellon University
5000 Forbes Ave.
Pittsburgh, PA 15213, USA
Tel: 1-412-268-3608
E-mail: landay@cs.cmu.edu
WWW Home Page: http://www.cs.cmu.edu:8001/Web/People/landay/home.html

ABSTRACT
Current interactive user interface construction tools are often more of a hindrance than a benefit during the early stages of user interface design. These tools take too much time to use and force designers to specify more of the design details than they wish at this early stage. Most interface designers, especially those who have a background in graphic design, prefer to sketch early interface ideas on paper or on a whiteboard. We are developing an interactive tool called SILK that allows designers to quickly sketch an interface using an electronic pad and stylus. SILK preserves the important properties of pencil and paper: a rough drawing can be produced *very quickly* and the medium is *very flexible*. However, unlike a paper sketch, this electronic sketch is *interactive* and can easily be *modified*. In addition, our system allows designers to examine, annotate, and edit a complete history of the design. When the designer is satisfied with this early prototype, SILK can *transform* the sketch into a complete, operational interface in a specified look-and-feel. This transformation is guided by the designer. By supporting the early phases of the interface design life cycle, our tool should both ease the development of user interface prototypes and reduce the time needed to create a final interface. This paper describes our prototype and provides design ideas for a production-level system.

KEYWORDS: User interfaces, design, sketching, gesture recognition, interaction techniques, programming-by-demonstration, pen-based computing, Garnet, SILK

INTRODUCTION

When professional designers first start thinking about a visual interface, they often sketch rough pictures of the screen layouts. In fact, everyone who designs user interfaces seems to do this, whether or not they come from a graphic design background. Their initial goal is to work on the overall layout and structure of the components, rather than to refine the detailed look-and-feel. Designers, who may also feel more comfortable sketching than using traditional palette-based interface construction tools, use sketches to quickly consider various interface ideas.

Additionally, research indicates that designers should *not* use current interactive tools in the early stages of development since this places too much focus on design details like color and alignment rather than on the major interface design issues, such as structure and behavior [23]. What designers need are computerized tools that allow them to sketch rough design ideas quickly [22].

We are developing an interactive tool called SILK, which stands for Sketching Interfaces Like Krazy, that allows designers to quickly sketch an interface using an electronic stylus. SILK then retains the "sketchy" look of the components. The system facilitates rapid prototyping of interface ideas through the use of common gestures in sketch creation and editing. Unlike a paper sketch, the electronic sketch allows the designer or test subjects to try out the sketch before it becomes a finished interface. At each stage of the process the interface can be tested by manipulating it with the mouse, keyboard, or stylus. Figure 1 illustrates a simple sketched interface. The interface has a scrollbar and a window for the scrolling data. It also has several buttons at the bottom, a palette of tools at the right, and four pulldown menus at the top.

Traditional user interface construction tools are often difficult to use and interfere with the designer's creativity. Our goal is to make SILK's user interface as unintrusive as pencil and paper. In addition to providing the ability to rapidly capture user interface ideas, SILK will allow a designer to edit the sketch using simple gestures. Furthermore, SILK's history mechanisms will allow designers to reuse portions of old designs and quickly bring up different versions of the same interface design for testing or comparison. Changes and written annotations made to a design over the course of a project can also be reviewed. Thus, unlike paper sketches, SILK sketches can evolve without forcing the designer to continually start over with a blank slate.

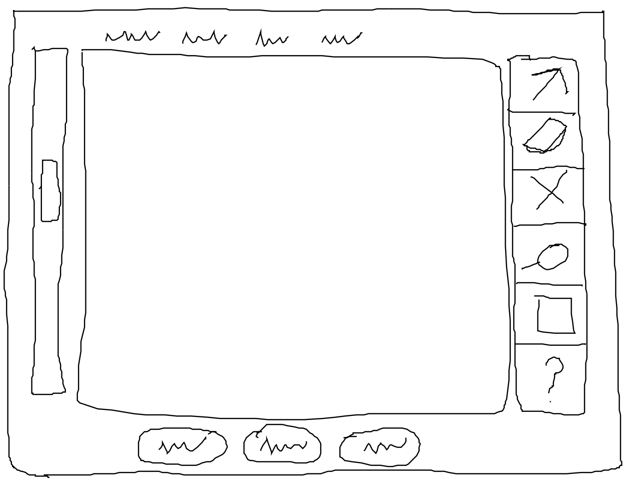

Figure 1: Sketched application interface created with SILK.

Unlike most existing tools, SILK will support the entire design cycle — from developing the initial creative design to developing the prototype, testing the prototype, and implementing the final interface. Our tool will provide the efficiency of sketching on paper with the ability to turn the sketches into real user interfaces for actual systems without re-implementation or programming. To some extent, SILK will be able to replace prototyping tools (*e.g.*, HyperCard, Director, and Visual Basic) and user interface builders (*e.g.*, the NeXT Interface Builder) for designing, constructing, and testing user interfaces (see Figure 2). At this time we have built a prototype of SILK that implements only a subset of the features described in this paper.

This paper describes how SILK functions and how it can be used effectively by user interface designers. The first section gives an overview of the problems associated with current tools and techniques. In the second section, we discuss the advantages of electronic sketching for user interface design. To ensure that the system would work well for its intended users, we took an informal survey of professional user interface designers to determine the techniques they now use for interface design. The results of the survey and a discussion of how these results were used in the design of SILK are presented in the next two sections. In the fifth section, we describe the sketch recognition algorithm. Finally, we summarize the related work and the status of SILK to date.

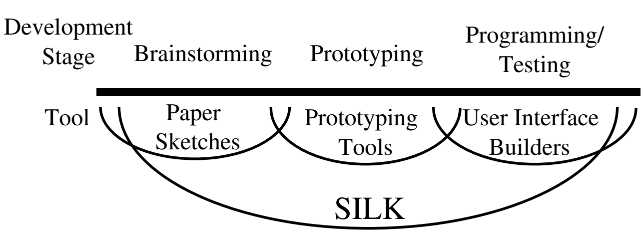

Figure 2: SILK can be used during all stages of user interface design, construction, and testing.

DRAWBACKS OF CURRENT DESIGN METHODS

User interface designers have become key members of software development groups. Designers often use sketching and other "low-fidelity techniques" [19] to generate early interface designs. *Low-fidelity techniques* involve creating mock-ups using sketches, scissors, glue, and post-it notes. Designers use mock-ups to quickly try out design ideas. Later they may use prototyping tools or user interface builders, or they may hand off the design to a programmer.

Prototyping tools allow non-programmers to write simple application mock-ups in a fraction of the time required using traditional programming techniques. *User interface builders,* the most common type of user interface construction tools, have become invaluable in the creation of both commercial and in-house computer applications. They allow the designer to create the look of a user interface by simply dragging widgets from a palette and positioning them on the screen. This facilitates the creation of the widget-based parts of the application user interface with little low-level programming, which allows the engineering team to concentrate on the application-specific portions of the product. Unfortunately, prototyping tools, user interface builders, and low-fidelity techniques have several drawbacks when used in the early stages of interface design.

Interface Tools Constrain Design

Traditional user interface tools force designers to bridge the gap between how they think about a design and the detailed specification they must create to allow the tool to reflect a specialization of that design. Much of the design and problem-solving literature discusses drawing rough sketches of design ideas and solutions [2], yet most user interface tools require the designer to specify more of the design than a rough sketch allows.

For example, the designer may decide that the interface requires a palette of tools, but she is not yet sure which tools to specify. Using SILK, a *thumbnail sketch* can easily be drawn with some rough illustrations to represent the tools (see Figure 1). This is in contrast to commercial interface tools that require the designer to specify unimportant details such as the size, color, finished icons, and location of the palette. This over-specification can be tedious and may also lead to a loss of spontaneity during the design process. Thus, the designer may be forced to abandon computerized tools until much later in the design process or forced to change design techniques in a way that is not conducive to early creative design.

One of the important lessons from the interface design literature is the value of iterative design; that is, creating a design prototype, evaluating the prototype, and then repeating the process several times [6]. Iterative design techniques seem to be more valuable as the number of iterations made during a project becomes larger. It is important to iterate quickly in the early part of the design process because that is when radically different ideas can and should be generated and examined. This is another area in which current tools fail during the early design process. The ability to turn out new designs quickly is hampered by the requirement for detailed designs. For example, in one test the interface sketched using SILK in Figure 1 could be created in just 70 seconds (sketched on paper it took 53 seconds), but to produce it with a traditional user interface builder (see Figure 3) took 329 seconds, which is nearly five times longer. In addition, the interface builder time does not include adding real icons to the tool palette due to the excessive time required to design or acquire them.

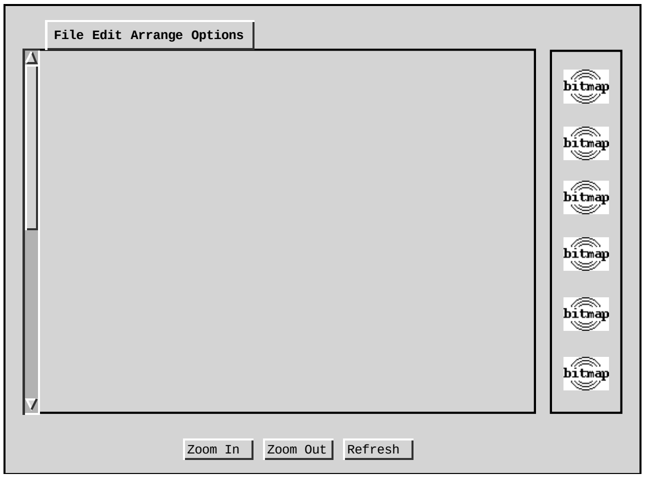

Figure 3: The sketched application from Figure 1 created with a traditional user interface builder.

HyperCard and Macromedia's Director are two of the most popular prototyping tools used by designers. Though useful in the prototyping stages, both tools come up short when used either in the early design stages or for producing production-quality interfaces. HyperCard's "programming" metaphor is based on the sequencing of different cards. HyperCard shares many of the drawbacks of traditional user interface builders: it requires designers to specify more design detail than is desired and often it must be extended with a programming language (HyperTalk) when the card metaphor is not powerful enough. In addition, HyperCard cannot be used for most commercial-quality applications due to its poor performance, which usually forces the development team to reimplement the user interface with a different tool.

Director was designed primarily as a media integration tool. Its strength is the ability to combine video, animation, audio, pictures, and text. This ability, along with its powerful scripting language, Lingo, has made it the choice of multimedia designers. These strengths, however, lead to its weaknesses when used as a general interface design tool. It is very hard to master the many intricate effects that Director allows. In addition, it lacks support for creating standard user interface widgets (*i.e.,* scrollbars, menus, and buttons) and specifying their behavior in a straightforward manner. Finally, its full-powered programming language is inappropriate for non-programmers. This is also the major drawback to using Visual Basic, which is becoming increasingly popular for interface prototyping due to its complete widget set and third-party support.

Drawbacks of Sketching on Paper

Brainstorming is a process that moves quickly between radically different design ideas. Sketches allow a designer to quickly preserve thoughts and design details before they are forgotten. The disadvantage of making these sketches on paper is that they are hard to modify as the design evolves. The designer must frequently redraw the common features that the design retains. One way to avoid this repetition is to use translucent layers [24, 7]. Another solution is to use an erasable whiteboard. Both of these approaches are clumsy at best. In order to be effective, translucent layers require forethought on the part of the designer in terms of commonality and layout of components. Whiteboards make it hard to scale and move compound objects, and they do not allow the designer to delete elements from a list easily. None of these solutions help with the next step when a manual translation to a computerized format is required, either with a user interface builder or by having programmers create an interface from a low-level toolkit. This translation may need to be repeated several times if the design changes.

Another problem with relying too heavily on paper sketches for user interface design is the lack of support for "design memory" [9]. The sketches may be annotated, but a designer cannot easily search these annotations in the future to find out why a particular design decision was made. Practicing designers have found that the annotations of design sketches serve as a diary of the design process, which are often more valuable to the client than the sketches themselves [2]. Sketches made on paper are also difficult to store, organize, search, and reuse.

One of the biggest drawbacks to using paper sketches is the lack of interaction possible between the paper-based design and a user, which may be one of the designers at this stage. In order to actually see what the interaction might be like, a designer needs to "play computer" and manipulate several sketches in response to a user's verbal or gestural actions. This technique is often used in low-fidelity prototyping [19].

Designers need tools that give them the freedom to sketch rough design ideas quickly, the ability to test the designs by interacting with them, and the flexibility to fill in the design details as choices are made.

ADVANTAGES OF ELECTRONIC SKETCHING

Electronic sketches have most of the advantages described above for paper sketches: they allow designers to quickly record design ideas in a tangible form. In addition, they do not require the designer to specify details that may not yet be known or important. Electronic sketches also have the advantages normally associated with computer-based tools: they are easy to edit, store, duplicate, modify, and search. Thus a computer-based tool can make the "design memory" embedded in the annotations even more valuable.

The other advantages of electronic sketching pertain to the background of user interface designers and the types of comments sketches tend to garner during design evaluations. A large number of user interface designers, and particularly the intended users of SILK, have a background in graphic design or art. These users have a strong sketching background and our survey (see the next section) shows they often prefer to sketch out user interface ideas. An electronic stylus is similar enough to pencil and paper that most designers should be able to effectively use our tool with little training.

Anecdotal evidence shows that a sketchy interface is much more useful in early design reviews than a more finished-looking interface. Wong [23] found that rough electronic sketches kept her team from talking about unimportant low-level details, while finished-looking interfaces caused them to talk more about the "look" rather than interaction issues. The belief that colleagues give more useful feedback when evaluating interfaces with a sketchy look is commonly held in the design community. Designers working with other low-fidelity prototyping techniques offer similar recommendations [19]. In the field of graphic design, Black's user study found that "the finished appearance of screen-produced drafts shifts attention from fundamental structural issues" [1].

SURVEY OF PROFESSIONAL DESIGNERS

In order to focus on how best to support user interface design, we surveyed sixteen professional designers to find out what tools and techniques they used in all stages of user interface design. We also asked them what they liked and disliked about paper sketching, and what they liked and disliked about electronic tools. We also asked the designers to send us sketches that they had made early in the design cycle of a user interface. Using this information, we designed SILK to support the types of elements designers currently sketch when designing interfaces.

The designers we surveyed have an average of over six years experience designing user interfaces. They work for companies from around the world that focus on areas such as desktop applications, multimedia software, telephony, and computer hardware manufacturing. In addition, like our intended users, they all have an art or graphic design background.

Almost all of the designers surveyed (94%) use sketches and storyboards during the early stages of user interface design. Some reported that they illustrate sequences of system responses and annotate the sketches as they are drawn. The designers said that user interface tools, such as HyperCard, would waste their time during this phase. They said that a drawing and an explanation could be presented to management and tested with users before building a prototype. One designer stated that in the early stages of design "iteration is critical and must happen as rapidly as possible — as much as two or three times a day." The designer said that user interface builders always slowed the design process, "especially when labels and menu item specifics are not critical." Most of the designers also cited their familiarity with paper as a graphic designer. The pencil and paper "interface" was described as intuitive and natural.

Almost all of the designers surveyed (88%) use either HyperCard, Director, or Visual Basic during the prototyping stage of interface design. Some also use high-powered user interface builders. The designers reported that Director was only useful for "movie-like" prototyping, *i.e.*, as a tool to illustrate the functionality of the user interface without the interaction. In addition, the designers disliked Director because it lacked a widget set, the designs could not be used again in the final product, and every control and system response had to be created from scratch. HyperCard was also cited for its lack of some necessary user interface components.

In contrast, the designers complimented the user interface builders on their complete widget sets and the fact that the designs could be used in the final product. The difficulty of learning to use these tools, especially those with scripting languages, was considered a drawback. Also, the designers wanted the ability to draw arbitrary graphics and some tools did not allow this. In fact, most of the designers expressed an interest in being able to design controls with custom looks. Twelve of the sixteen (75%) reported that 20% or more of their time was spent designing this type of widget.

The designers reacted favorably to a short description we gave them about SILK. Some were concerned that it was not really paper and that they might need to get accustomed to it. The designers felt our system would allow quick implementation of design ideas and it would also help bring the sketched and electronic versions of a design closer together. In addition, the designers were happy with the ability to quickly iterate on a design and to eventually use that design in the final product. All but two expressed a willingness to try such a system.

DESIGNING INTERFACES WITH SILK

SILK blends the advantages of both sketching and traditional user interface builders, yet it avoids many of the limitations of these approaches. SILK enables the designer to move quickly through several iterations of a design by using gestures to edit and redraw portions of the sketch. Our system tries to recognize user interface widgets and other interface elements *as they are drawn* by the designer. Although the recognition takes place as the sketch is made, it is unintrusive and users will only be made aware of the recognition results if they choose to exercise the widgets. As soon as a widget has been recognized, it can be exercised. For example, the "elevator" of the sketched scrollbar in Figure 1 can be dragged up and down.

Next, the designer must specify the behavior *among* the interface elements in the sketch. For example, SILK knows how a button operates, but it cannot know what interface action should occur when a user presses the button. Some of this can be inferred either by the type of the element or with by-demonstration techniques [4, 16], but much of it may need to be specified using a visual language we are designing or even a scripting language for very complex custom behaviors. Our prototype does not yet support the specification of behaviors.

When the designer is happy with the interface, SILK will replace the sketches with real widgets and graphical objects; these can take on a specified look-and-feel of a standard graphical user interface, such as Motif, Windows, or Macintosh. The transformation process is mostly automated, but it requires some guidance by the designer to finalize the details of the interface (*e.g.*, textual labels, colors, *etc.*) At this point, programmers can add callbacks and constraints that include the application-specific code to complete the application. Figure 3 illustrates what the finished version of the interface illustrated in Figure 1 *might* look like had it been transformed by SILK (although SILK would have retained the sketched palette icons from Figure 1.)

Feedback

Widgets recognized by the system appear on the screen in a different color to give the designer feedback about the inference process. In addition, the type of the widget last inferred is displayed in a status area. Although both of these mechanisms are unobtrusive, the feedback can be disabled to allow the designer to sketch ideas quickly without any distractions.

The designer can help the system make the proper inference when either the system has made the wrong choice, no choice, or the widget that was drawn is unknown to the system. In the first case the designer might use the "cycle" gesture (see Figure 4) to ask the system to try its next best choice. Alternatively, the designer can choose from a list of possible widget types. If SILK made no inference on the widget in question, the designer might use the grouping gesture to force the system to reconsider its inference and focus on the components that have been grouped together. Finally, if the designer draws a widget or graphical object that SILK does not recognize, the designer can group the relevant components and then specify a name for the widget. This will allow the system to recognize the widget in the future.

Editing Sketches

One of the advantages of interactive sketches over paper sketches is the ability to quickly edit them. When the user holds down the button on the side of the stylus, SILK interprets strokes as editing gestures instead of gestures for creating new objects. These gestures are sent to a different classifier than the one used for recognizing widget components. The power of gestures comes from the ability to specify, with a single mark, a set of objects, an operation, and other parameters [3]. For example, deleting or moving a section of the drawing is as simple as making a single stroke with the stylus.

SILK supports gestures for cycling among inferences, deleting, moving, copying, and grouping basic components or widgets. The grouping gesture acts as a "hint" in the search for sequence and nearness relationships. Examples of these gestures are illustrated in Figure 4. As we test our system with more interface designers we expect to add gestures for other common operations.

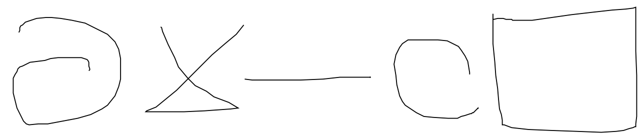

Figure 4: Gestures for cycling, deleting, moving, copying, and grouping.

Design History Support

One of the important features of SILK is its strong support for design history. A designer will be able to save designs or portions of designs for later use or review. Multiple designs can be displayed at the same time in order to perform a side-by-side comparison of their features or to copy portions of one design into a new design. SILK can also display several designs in a miniaturized format so that the designer can quickly search through previously saved designs visually rather than purely by name.

Another important history mechanism is SILK's support for annotations. The system will allow the designer to annotate a design by either typing or sketching on an *annotation layer*. This layer can be displayed or hidden with the click of a button in the SILK control panel. In addition, the annotations that were made using the keyboard can be searched later using a simple search dialog box. SILK will also support multiple layers, allowing different members of the design team to create personal annotations.

Specifying Behavior

In addition to editing and creating new objects in sketch mode, SILK also supports run mode and behavior mode. Run mode, which can be turned on from the SILK control panel, allows the designer to test the sketched interface. For example, as soon as SILK recognizes the scrollbar shown in Figure 1, the designer can switch to run mode and operate the scrollbar by dragging the "elevator" up and down. The buttons in Figure 1 can be selected with the stylus or mouse and they will highlight while the button is held down.

Easing the specification of the interface layout and structure solves much of the design problem, but a design is not complete until the behavior has also been specified. Unfortunately, the behavior of individual widgets is insufficient to test a working interface. Behavior mode will be used to specify the dynamic behavior between widgets and the basic behavior of new widgets or application-specific objects drawn in sketch mode.

We have identified two basic levels of behavior that the system must be able to handle. Sequencing between screens, usually in conjunction with hand drawn storyboards, is a behavior that has been shown to be a powerful tool for designers making concept sketches for early visualization [2]. The success of HyperCard has demonstrated that a significant amount of behavior can be constructed from sequencing screens upon button presses. For example, the designer may wish to specify that a dialog box appears when a button is pressed. Our survey also showed that designers often want to draw new widgets whose behavior is *analogous* to that of a known widget. For example, designers frequently need to draw a new icon and specify that it should act like a button. Finally, designers occasionally need to sketch a widget that has an entirely new behavior. Our experience is that this is not very common and thus we do not plan on supporting the definition of entirely new behaviors.

We are investigating several alternative ways to specify these behaviors. Programming-by-demonstration (PBD) is a technique in which one specifies a program by directly operating the user interface. In sketch mode we specify the layout and structure of the interface as described above, while in behavior mode we could demonstrate possible end-user actions and then specify how the layout and structure should change in response. A similar technique is used to describe new interface behaviors in the Marquise system [17].

A critical problem with PBD techniques is the lack of a static representation that can be later edited. Marquise and Smallstar [8] use a textual language (a formal programming language in the latter case) to give the user feedback about the system's inferences. In addition, scripts in these languages can then be edited by the user to change the "program". This solution is not acceptable considering that the intended users of SILK are user interface designers who generally do not have programming experience.

We may be able to solve this problem by combining PBD techniques with visual languages as in the Pursuit [13] and Chimera [11] systems. We are especially interested in using a visual notation that is made directly on the interface whose behavior is being described. Marks or symbols layered on top of the interface are used for feedback indicating graphical constraints in Briar [5] and Rockit [10]. In Rockit, the marks kept the user informed of the current inference of the system.

Using a notation of marks that are made directly on the sketch is beneficial for several reasons. One of the most important reasons is we can now use the same visual language for both the specification of the behavior and the editable representation indicating which behavior has been inferred or specified by the designer. In addition, these sketchy marks might be similar to the types of notations that one might make on a whiteboard or paper when designing an interface. For example, sequencing might be expressed by drawing arrows from buttons to windows or dialog boxes that appear when the button is pressed. Like the annotation layer described earlier, the layer that contains these behavioral marks can be turned on and off.

Another technique used to specify a behavior is to select it from a list of known behaviors and attach it to a drawn element. This seems well-suited for specifying analogous behaviors. This technique can be very limiting if the default behaviors do not include what a designer wishes to specify. We intend to survey many commercial applications and designers to see if there is a reasonably small number of required behaviors so that they can be presented in list form. The success of the Garnet Interactor model indicates that a small list may be sufficient [14].

The visual language and PBD approaches provide specification methods that are similar to the way the interface is used. The list approach, however, is easy to use and may be quite successful for common behaviors. We expect to combine these techniques and then conduct user testing to refine the interface.

RECOGNIZING WIDGETS

Allowing designers to sketch on the computer, rather than on paper, has many advantages as we have already described. Several of these advantages cannot be realized without software support for recognizing the interface widgets in the sketch. Having a system that recognizes the drawn widgets gives the designer a tool that can be used for designing, testing, and eventually producing a final application interface. SILK's recognition engine identifies individual user interface components as they are drawn, rather than after an entire sketch has been completed. This way the designer can test the interface at any point without waiting for the entire sketch to be recognized. Working within the limited domain of common 2-d interface widgets (*e.g.*, scrollbars, buttons, pulldown menus, *etc.*) facilitates the recognition process. This is in contrast to the much harder problems faced by systems that try to perform generalized sketch recognition or beautification [18]. Our sketch recognition algorithm uses a rule system that contains basic knowledge of the structure and make-up of user interfaces to infer which widgets are included in the sketch.

Recognizing Widget Components

The recognition engine uses Rubine's gesture recognition algorithm [20] to identify the basic components that make up an interface widget. These basic components are then combined to make more complex widgets. For example, the scrollbar in Figure 1 was created by sketching a tall, thin rectangle and then a small rectangle (though the order in which they were sketched does not matter). Each of the basic components of a widget are trained by example using the Agate gesture training tool [12].

The algorithm currently limits our system to single-stroke gestures for the basic components. This means that the designer drawing the scrollbar in Figure 1 must use a single stroke of the pen for each of the rectangles that comprise the scrollbar. We intend to develop a better algorithm so that the designer can use multiple-strokes to draw the basic components.

Rubine's algorithm uses statistical pattern recognition techniques to train classifiers and recognize gestures. These techniques are used to create a classifier based on the features extracted from several examples. In order to classify a given input gesture, the algorithm computes the distinguishing features for the gesture and returns the best match with the learned gesture classes.

Composing Components

In order to recognize interface widgets, our algorithm must combine the results from the classification of the single-stroke gestures that make up the basic components. As each component is sketched and classified it is passed to an algorithm that looks for the following relationships:

1) Does the new component *contain* or is it *contained by* another component?
2) Is the new component *near* another component?
3) Is the new component in a *sequence* of components of the same type?

The first relationship is the most important for classifying widgets. We have noticed that many of the common user interface widgets can be expressed by containment relationships between more basic components. For example, the scrollbar in Figure 1 is a tall, skinny rectangle that contains a smaller rectangle. The second relationship allows the algorithm to recognize widgets such as check boxes, which usually consist of a box with text next to it. The final relationship allows for groupings of related components that make up a set of widgets (*e.g.*, a set of radio buttons.)

After identifying the basic relationships between the new component and the other components in the sketch, the algorithm passes the new component and the identified relationships to a rule system that uses basic knowledge of the structure and make-up of user interfaces to infer which widget was intended. Each of the rules that matches the new component and relationships assigns a confidence value that indicates how close the match is. The algorithm then takes the match with the highest confidence value and assigns the component to a new aggregate object that represents a widget. If none of the rules match, the system assumes that there is not yet enough detail to recognize the widget.

Adding new components to the sketch can cause the system to revise previously made widget identifications. This will only occur if the new component causes the rule system to identify a different widget as more likely than its previous inference. Similarly, deleting components of a widget can cause a new classification of the rest of the sketch.

Each of the widgets that SILK recognizes has corresponding Garnet objects that use the Garnet Interactor mechanism [14] to support interaction and feedback. When SILK identifies a widget, it attaches the sketched components that compose it to an instance of an interactor object that implements the required interaction.

STATUS

We currently have a prototype of SILK running under Common Lisp on both UNIX workstations and an Apple Macintosh with a Wacom tablet attached. The prototype is implemented using Garnet [15]. The prototype supports recognition and operation of several standard widgets. In addition, the system can transform SILK's representation of the interface to an interface with a Motif look-and-feel. The system currently only recognizes a few ways of drawing each widget. Using the sketches sent to us by designers, we plan to extend the rule system to recognize more alternatives. SILK does not yet allow the specification of behavior between the widgets (*i.e.*, the sketchy scrollbar can be scrolled but it cannot yet be attached to the window containing data to scroll.) In addition, annotations are the only implemented history mechanism.

We are currently adding support to recognize more widgets and application-specific graphics. In addition, we are looking at ways to support multiple stroke recognizers. We plan to have design students use SILK in a user interface design course to see how it performs in practice. We have also begun designing a formal study to compare the types of problems found when performing an evaluation on both sketchy and finished-looking interfaces.

RELATED WORK

Wong's work on scanning in hand-drawn interfaces was the major impetus for starting our work in this area [23]. Our work differs in that we give designers a *tool* that allows them to create both the look and behavior of these interfaces directly with the computer. In addition, we will try to show that Wong's anecdotal evidence is supported in practice by comparing the types of comments made and the problems found when performing an interface evaluation on both sketchy and finished-looking interfaces.

Much of the work related to our system is found in the field of design tools for architects. For example, Strothotte reports that architects often sketch over printouts produced by CAD tools before showing works in progress to clients [21]. This seems to lend further evidence to the assertion that a different level of feedback is obtained from a sketchy drawing. In fact, Strothotte has produced a system that can render precise architectural drawings in a sketchy look.

Another important architectural tool allows architects to sketch their designs on an electronic pad similar to the one we are using [7]. Like SILK, this tool attempts to recognize the common graphic elements in the application domain — architectural drawings. Our tool differs in that it allows the specification and testing of the *behavior* of the drawing, whereas the architectural drawing is fairly static.

CONCLUSIONS

We envision a future in which most of the user interface code will be generated by user interface designers using tools like SILK rather than by programmers writing the code. We have designed our tool only after surveying the intended users of the system. These designers have reported that current user interface construction tools are a hindrance during the early stages of interface design; we have seen this both in our survey and in the literature. Our interactive tool will overcome these problems by allowing designers to quickly sketch an interface using an electronic stylus. Unlike a paper sketch, an electronic sketch will allow the designer or test subjects to interact with the sketch before it becomes a finalized interface. We believe that an interactive sketching tool that supports the entire interface design cycle will enable designers to produce better quality interfaces in a shorter amount of time than with current tools.

ACKNOWLEDGMENTS

The authors would like to thank Dan Boyarski, David Kosbie, and Francesmary Modugno for their helpful comments on this work. We would also like to thank the designers who responded to our survey. Finally, we would like to thank Stacey Ashlund, Elizabeth Dietz, and Dale James for help with technical writing.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, ARPA Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

REFERENCES

1. Black, A. Visible planning on paper and on screen: The impact of working medium on decision-making by novice graphic designers. *Behaviour & Information Technology 9*, 4 (1990), 283–296.

2. Boyarski, D. and Buchanan, R. Computers and communication design: Exploring the rhetoric of HCI. *Interactions 1*, 2 (April 1994), 24–35.

3. Buxton, W. There's more to interaction than meets the eye: Some issues in manual input. In *User Centered Systems Design: New Perspectives on Human-Computer Interaction,* Norman, D.A. and Draper, S.W., Lawrence Erlbaum Associates, Hillsdale, N.J., 1986, pp. 319–337.

4. Cypher, A. *Watch What I Do: Programming by Demonstration,* MIT Press, Cambridge, MA (1993).

5. Gleicher, M. and Witkin, A. Drawing with constraints. *The Visual Computer 11*, 1 (1995), To appear.

6. Gould, J.D. and Lewis, C. Designing for usability: Key principles and what designers think. *Communcations of the ACM 28*, 3 (March 1985), 300–311.

7. Gross, M.D. Recognizing and interpreting diagrams in design. In *Proceedings of the ACM Conference on Advanced Visual Interfaces '94,* Bari, Italy, June 1994.

8. Halbert, D.C. *Programming by Example*, Ph.D. dissertation, Computer Science Division, EECS Department, University of California, Berkeley, CA, 1984.

9. Herbsleb, J.D. and Kuwana, E. Preserving knowledge in design projects: What designers need to know. In *Proceedings of INTERCHI '93: Human Factors in Computing Systems,* Amsterdam, The Netherlands, April 1993, pp. 7–14.

10. Karsenty, S., Landay, J.A., and Weikart, C. Inferring graphical constraints with Rockit. In *HCI '92 Conference on People and Computers VII,* BritishComputer Society, September 1992, pp. 137–153.

11. Kurlander, D. *Graphical Editing by Example*, Ph.D. dissertation, Department of Computer Science, Columbia University, July 1993.

12. Landay, J.A. and Myers, B.A. Extending an existing user interface toolkit to support gesture recognition. In *Adjunct Proceedings of INTERCHI '93: Human Factors in Computing Systems,* Amsterdam, The Netherlands, April 1993, pp. 91–92.

13. Modugno, F. and Myers, B.A. Graphical representation and feedback in a PBD system. In *Watch What I Do: Programming by Demonstration*. MIT Press, Cypher, A., Ch. 20, pp. 415–422, Cambridge, MA, 1993.

14. Myers, B.A. A new model for handling input. *ACM Transactions on Information Systems 8*, 3 (July 1990), 289–320.

15. Myers, B.A., Giuse, D., Dannenberg, R.B., Vander Zanden, B., Kosbie, D., Pervin, E., Mickish, A., and Marchal, P. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer 23*, 11 (November 1990), 71–85.

16. Myers, B.A. Demonstrational Interfaces: A step beyond direct manipulation. *IEEE Computer 25*, 8 (August 1992), 61–73.

17. Myers, B.A., McDaniel, R.G., and Kosbie, D.S. Marquise: Creating complete user interfaces by demonstration. In *Proceedings of INTERCHI '93: Human Factors in Computing Systems,* Amsterdam, The Netherlands, April 1993, pp. 293–300.

18. Pavlidis, T. and Van Wyk, C.J. An automatic beautifier for drawings and illustrations. *Computer Graphics 19*, 3 (July 1985), 225–234, ACM SIGGRAPH '85 Conference Proceedings.

19. Rettig, M. Prototyping for tiny fingers. *Communications of the ACM 37*, 4 (April 1994), 21–27.

20. Rubine, D. Specifying gestures by example. *Computer Graphics 25*, 3 (July 1991), 329–337, ACM SIGGRAPH '91 Conference Proceedings.

21. Strothotte, T., Preim, B., Raab, A., Schumann, J., and Forsey, D.R. How to render frames and influence people. In *Proceedings of Eurographics '94,* Oslo, Norway, September 1994, pp. 455–466.

22. Wagner, A. Prototyping: A day in the life of an interface designer. In *The Art of Human-Computer Interface Design*. Addison-Wesley, Laurel, B., pp. 79–84, Reading, MA, 1990.

23. Wong, Y.Y. Rough and ready prototypes: Lessons from graphic design. In *Short Talks Proceedings of CHI '92: Human Factors in Computing Systems,* Monterey, CA, May 1992, pp. 83–84.

24. Wong, Y.Y. Layer Tool: Support for progressive design. In *Adjunct Proceedings of INTERCHI '93: Human Factors in Computing Systems,* Amsterdam, The Netherlands, April 1993, pp. 127–128.