Interactive Sketching for the Early Stages of User Interface Design

James A. Landay and Brad A. Myers

22 July 1994
CMU-CS-94-176

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also appears as Human-Computer Interaction Institute Technical Report
CMU-HCII-94-104

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 to 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 white-board. We are developing an interactive tool that allows designers to quickly sketch an interface using an electronic pad and stylus. Our tool preserves the important properties of paper: A rough drawing can be produced very *quickly* and the medium is very *flexible*. However, unlike a paper sketch this electronic sketch can easily be *exercised* and *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, the system can *transform* the sketch into a complete, finished interface in a specified look-and-feel. This transformation takes place with the guidance of the designer. By supporting the early design phases of the software life-cycle, our tool should both ease the prototyping of user interfaces and improve the speed with which a final interface can be created.

E-mail: landay@cs.cmu.edu
WWW Home Page: http://www.cs.cmu.edu:8001/Web/People/landay/home.html

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326.

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 an interface, they often sketch rough pictures of the screens. In fact, everyone who designs user interfaces seems to do this, whether they come from a graphic design background or not. Their initial goal is to work on the overall layout and structure of the components, rather than refine the detailed look and feel. Designers, who may also feel more comfortable sketching than using traditional palette-based interface construction tools, such as Apple's HyperCard, are able to use sketches for quickly trying out various interface ideas.

Additionally, research indicates that designers should *not* use current interactive tools in the early stages of development, since then colleagues focus too much on design details like color and alignment rather than on the major interface design issues, such as structure and behavior [29]. What designers need are computerized tools that allow the freedom to quickly sketch rough design ideas [28].

We are developing a prototype interactive tool called SILK 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 with a paper sketch, the designer or test subjects will be able to exercise the electronic sketch before it becomes a finished interface. At each stage of the process the interface can be tested by manipulating the interface with the mouse, keyboard, or stylus. Figure 1 shows what a simple sketched interface might look like. The interface features include a scrollbar, a window that will contain the data to scroll, several buttons at the bottom, a palette of tools, and a pulldown menu at the top.

In addition to providing the ability to implement rapidly-sketched user interface ideas, SILK will allow a designer to easily 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. Various changes made to a design over the course of a project can be reviewed, including viewing attached written annotations. Thus, unlike paper sketches, SILK sketches can evolve very easily without forcing the designer to always start over with a blank slate.

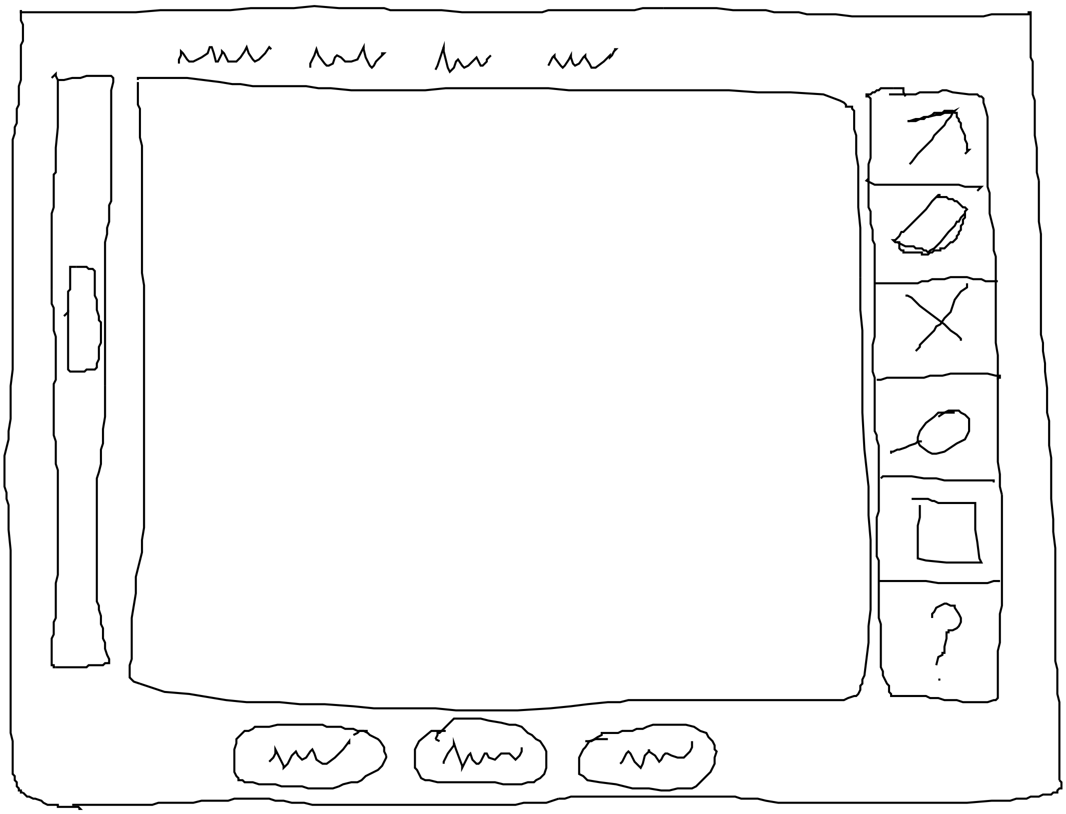

Figure 1: A sketched interface to a simple application created with SILK.

Unlike most existing tools, SILK will support the entire design cycle – from early creative design through prototyping and final implementation. The 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. SILK will be able to replace, to some extent, the use of tools like Apple's HyperCard and traditional user interface construction tools, such as the NeXT Interface Builder, in the design, construction, and testing of user interfaces (see Figure 2).

The rest of this paper describes and motivates SILK. The next section gives an overview of some of the problems with the current tools and techniques used to design user interfaces. We then discuss some of the advantages of electronic sketching for user interface design. To ensure that our system would work well for our intended users, we took an informal survey of professional user interface designers on the techniques they now use for interface design. We present those results next. The results of the survey, along with the advantages of electronic sketches, led to the user interface design for SILK, in the following section. The sketch recognition algorithm, described in the next to last section, was designed to handle the types of widgets designers typically draw. Finally, we summarize the related work and the current status of our system.

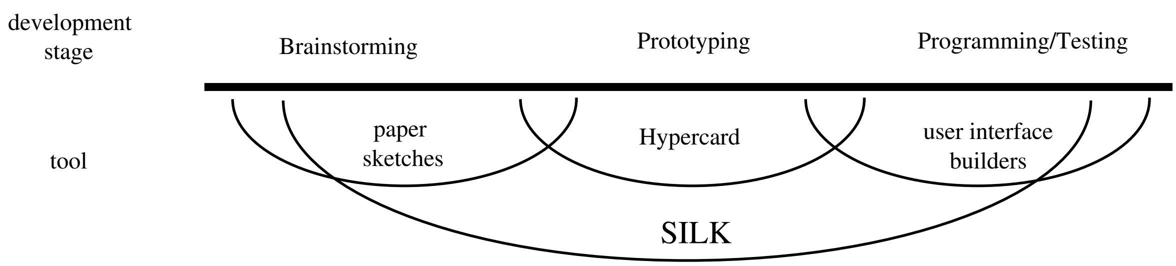

Figure 2: SILK can be used during all stages of user interface design, construction, and testing.

DRAWBACKS OF CURRENT DESIGN TECHNIQUES

User interface designers have become key members of software development groups. Designers often use sketching and other “low-fidelity techniques” [24] to generate early interface designs. *Low-fidelity techniques* involve creating mock-ups out of sketches, glue, scissors, and post-it notes. Designers use these mock-ups to quickly try out design ideas, later moving on to prototyping tools, user interface builders, or handing the design off to a programmer. HyperCard is the most commonly used *prototyping tool* used by interface designers. It allows non-programmers to write simple applications in a fraction of the time required by traditional programming techniques. *User interface builders,* the most common type of user interface construction tool, have become invaluable in the creation of both commercial and in-house computer applications. They allow the creation of much of the look of a user interface by simply dragging widgets from a palette and placing them at the desired location on the screen. This facilitates the creation of the widget-based parts of the application user interface with little low-level programming, allowing the engineering team to concentrate on the application-specific portions of the product. Unfortunately, prototyping tools, user interface builders, and current low-fidelity techniques, such as paper sketches, have several drawbacks when used in the early stages of interface design.

Interface Tools Constrain Design

Traditional user interface tools force designers to bridge a 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 speaks of drawing rough sketches of design ideas and solutions [2, 23], yet most user interface tools require the designer to specify much more of the design than a rough sketch may contain.

For example, the designer may decide that her interface requires a palette of tools, but is not yet sure what the individual tools really are. Using SILK, with a rough sketch known as a *thumbnail sketch*, she could easily draw the palette with some arbitrary squiggles to represent the tools (see Figure 1.) This is in contrast to commercial interface tools which require the designer to specify many unimportant details such as the size, color, 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 either abandon computerized tools until much later in the design process or to change her 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 the larger the number of iterations made during a project. It is very important to iterate more quickly in the early part of the design process because that is when radically different ideas are examined as the design team tries to explore the design space. This is another area in which current tools fail during the early design process. The desire to turn out new designs quickly is hampered by the requirement for detailed designs. For example, the interface sketched using SILK in Figure 1 could be created in just seventy seconds (sketched on paper it took fifty-three seconds), but to produce it with a traditional user interface builder took five-times longer (see Figure 3). In addition, the interface in Figure 3 does not even have real icons in its tool palette due to the time required to design or acquire them.

Apple's 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. The HyperCard "programming" metaphor is based on the sequencing of different *cards*. For example, to make a button change state the developer can have two cards that differ only in the button appearance. The developer then specifies that the second card in the sequence can replace the first on the screen when the mouse is clicked on the button. HyperCard shares many of the drawbacks of traditional user interface builders: It often requires designers to specify more details of the look 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, forcing the development team to reimplement the user interface with a different tool.

Director was designed primarily as a media integration tool. Its strengths are in combining video, animation, audio, pictures, and text. These areas, along with its powerful scripting language, Lingo, have made it the choice of multimedia designers. These strengths 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, its full-powered programming language is inappropriate for non-programmers. Finally, it lacks support for creating standard user interface widgets (*i.e.,* scrollbars, menus, and buttons) and specifying their behavior in a straightforward manner.

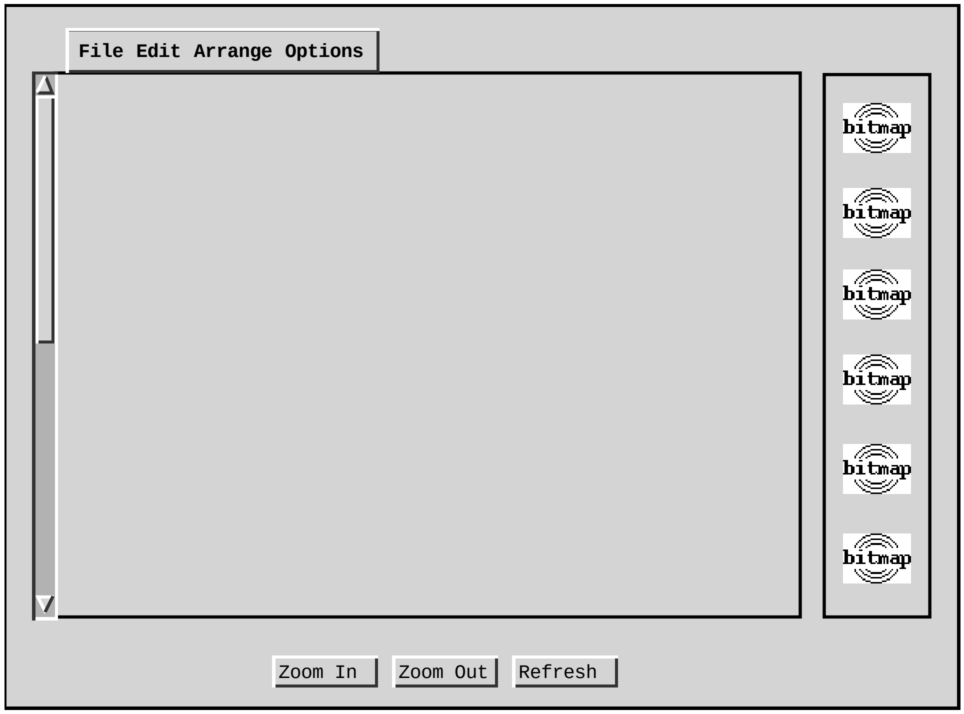

Figure 3: A finished version, created with a standard interface builder, of the sketched application from Figure 1.

Drawbacks of Sketching on Paper

Brainstorming is a process that generally moves quickly between radically different design ideas. Sketches allow a designer to quickly preserve thoughts and design details before they are forgotten. One of the disadvantages of making these sketches on paper is that it is hard to modify them as the design evolves. The designer must often redraw the common features that the current design retains. One way to avoid this repetition is the use of translucent layers [30, 7]. Another possible solution is the use of an erasable white-board. 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 can not go back and easily search these annotations in the future to find out why a particular design decision was made. Practicing designers have found that the annotations that go along with design sketches serve as a diary of the design process and are often more valuable to the client than the sketches themselves [2]. In addition, storing, organizing, searching, and reusing sketches may be difficult with paper designs.

One of the biggest drawbacks to using paper sketches is the lack of interaction possible between a user, which may be one of the designers at this stage, and the paper-based design. In order to actually see what the interaction might be like, a designer needs to "play computer" and manipulate several paper sketches or models in response to a user's verbalized actions. This technique is often used in low-fidelity prototyping [24]. Designers need tools that allow the freedom to quickly sketch rough design ideas, the ability to test the designs by interacting with them, and the flexibility to fill in the design details as these choices are made.

ADVANTAGES OF ELECTRONIC SKETCHING

Electronic sketches have most of the advantages described above for paper sketches: *e.g.*, they allow designers to quickly get ideas out of their head and realized in some tangible form. In addition, they do not require the designer to specify details that may not yet be known or important. Electronic sketches also bring with them all of the advantages normally associated with computer-based tools: ease of editing, storage, duplication, modification, and searchability. 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 our tool, 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. Despite their differences, an electronic stylus is similar enough to pencil and paper that most designers should be able to pick up our sketching interface with little training.

Anecdotal evidence that shows a sketchy looking interface may be much more useful in design reviews than a more finished looking interface is another compelling reason to explore the use of rough electronic sketches in user interface design. Wong found that rough 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 [29]. Designers working with other low-fidelity prototyping techniques offer similar recommendations [24]. 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 four practicing 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. Finally, we also had many designers send us sketches that they had made early in the design cycle of a user interface. In this way we can design our tool to support the types of elements designers currently sketch when making interface designs.

The designers we surveyed had an average of over six years experience designing user interfaces. They work for companies that focus on diverse areas: consulting, multimedia software development, and computer hardware manufacture. In addition, like our intended users, they all have an art or graphic design background. All of the designers surveyed use either HyperCard, Director, or Visual Basic during the prototyping stage of interface design. Some even used high-powered user interface builders.

All of the designers 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 felt that user interface tools, such as HyperCard, would waste their time. It was felt that a drawing and an explanation can be tested against management and users in order to obtain a commitment to build 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 felt that user interface builders were always too slow, "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 paper and pen "interface" was described as intuitive and natural.

The designers found 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 its lack of a widget set and that the designs had to be thrown away with no chance of reuse in the final product. The respondents also disliked the fact that every control and every system response had to be created from scratch when using Director. HyperCard was also cited for its lack of all necessary user interface components.

In contrast, the designers complemented 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 the tools, especially those with scripting languages, was considered a drawback. Also, the designers wanted the ability to draw arbitrary graphics in the tool and some tools did not allow this. In fact, most of the designers expressed an interest in being able to design custom controls.

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 the electronic versions 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 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 avoids many of the limitations of these approaches. SILK enables the designer to quickly move 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 the user 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 a sketched scrollbar can be dragged.

Next, the designer must specify the behavior of the interface elements in the sketch. For example, SILK knows how a button operates, but it can not know what action should occur when a user presses the button. Much of this can be inferred either by the type of the element or with by-demonstration techniques [4, 19], but some of it may need to be specified using dialog boxes or even a scripting language for very complex custom 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 graphic user interface, such as Motif or the Macintosh. The transformation process will be primarily automated, but will require some user guidance to finalize the details of the interface (*e.g.*, textual labels, the standard look-and-feel to use, colors, *etc.*) At this point, programmers can add callbacks 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 (though SILK would have retained the sketched palette icons from Figure 1.)

Feedback

Widgets that have been 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. Either of these types of feedback can be disabled to allow the designer to quickly sketch ideas without any interruptions.

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. 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 of which SILK has no knowledge, the designer can group the basic components in question and then give the system a name for the widget. This will allow the system to recognize this widget in the future, though SILK will know nothing about that widget's behavior until the designer demonstrates it.

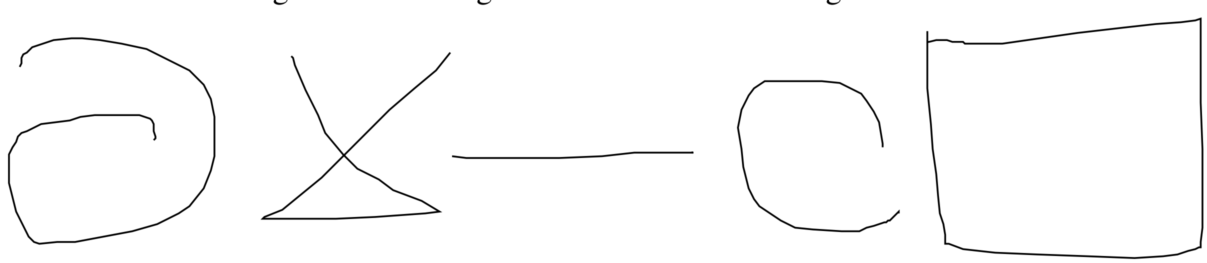

Figure 4: Gestures for cycling, deleting, moving, copying, and grouping.

Editing Sketches

One of the advantages of our interactive sketch system over paper sketches is the ability to quickly edit them. When the button on the side of the stylus is held down, strokes are interpreted 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 of the stylus.

SILK supports gestures for deleting, moving, copying, grouping basic components or widgets, and cycling among inferences. 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, such as undo.

Design History Support

One of the important features of SILK is its strong design history support. SILK allows the designer to easily annotate a design by either typing or sketching on an “annotation layer”. This layer can be easily displayed or hidden with the click of a button. In addition, the annotations that were made using the keyboard can be searched later using a simple search dialog box.

Another important history mechanism is the ability to save designs or portions of designs for later use or review. Multiple designs can be displayed at once in order to perform a side-by-side comparison of their features or to copy portions of one design for use in a new design. SILK can also display several designs in a miniaturized format in order to allow the designer to quickly search through previously saved designs visually rather than purely by name.

Specifying Behavior

SILK supports three major modes: sketch mode, run mode, and demonstrate mode. In sketch mode the pen strokes are used for adding new widgets to the interface or editing the ones that are currently there. 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 to show this selection 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. There needs to be a way to specify dynamic *behavior between widgets* in response to user actions. Sequencing between screens, usually in conjunction with hand drawn “kinetic storyboards”, is a behavior that has been shown to be a powerful tool for designers making concept sketches for early visualization [2]. For example, the designer may wish to specify that a dialog box appears when a button is pressed. Demonstrate mode is used to specify this dynamic behavior between widgets and the basic behavior of new widgets or application-specific objects drawn in sketch mode.

Currently our system is unable to deal directly with this dynamic behavior, but we are investigating several alternatives. Programming-by-demonstration (PBD) is a technique in which we specify a program in the same language as that of the interface. In sketch mode we specify the layout and structure of the interface as described above, while in demonstrate 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 interface behaviors in the Marquise system [20].

The primary problem with by-demonstration 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 later case, to give the user feedback about the system's inferences. In addition, 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 a programming background.

We may be able to solve this problem by combining by-example techniques with visual languages as in the Pursuit [15] 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].

Using a notation of marks that are made directly on the sketch is attractive to us 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 previously 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 dialogs that appear upon pressing the button.

Other techniques that may be used to give feedback in PBD systems include animation [14], questions and answers [16, 26], and auditory cues [13]. Another technique we can use to specify behavior is by selecting from a list of known behaviors and attaching it to a drawn element. 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 [17].

The visual language and by-example approaches allow a specification method that is similar to the way the interface is used, while the list approach is easy to use and may be quite successful for common behaviors. SILK will most likely support two of the techniques and we will then perform user testing to see which works best in the user interface specification domain. After the interface structure and behavior has been determined, we can generate the finished interface and a model of the interface that can be used by other tools to determine its usability [21].

RECOGNIZING WIDGETS

Allowing designers to sketch on the computer, rather than on paper, has many advantages that we have described above. Several of these advantages can not be realized without software support for understanding the user interface widgets embodied in the sketch. Having a system that recognizes the widgets being drawn gives the designer a tool that can be used for design, testing, and eventual production of a final application interface. SILK's recognition engine tries to identify individual user interface components as they are drawn, rather than after an entire sketch has been completed. Working within the limited domain of common 2-d user interfaces widgets (*e.g.*, scrollbars, buttons, pull-down menus, *etc.*) eases the recognition process. This is in contrast to the much harder problems faced by systems that try to perform generalized sketch recognition or beautification [22]. Our sketch recognition algorithm uses a rule system that contains basic knowledge of the structure and make-up of user interfaces to infer the interface widgets that are embodied in the sketch.

Recognizing Widget Components

The recognition engine uses Rubine's gesture recognition algorithm [25] to identify the basic components that make up a user interface widget. These basic components are then combined to make more complex widgets. For example, the scrollbar in Figure 1 was created by sketching the tall, thin rectangle and then the small rectangle in Figure 5 (though the order in which they were sketched does not matter). Of course when creating an actual scrollbar the small rectangle must be drawn inside of the larger rectangle. Each of the basic components of a widget are trained by example using the Agate gesture training tool [12], which is part of the Garnet User Interface Development Environment [18].

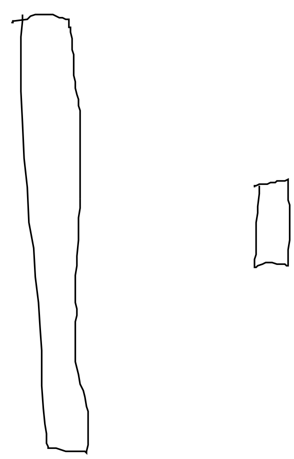

Figure 5: The two rectangular components that make up a sketched scrollbar.

This algorithm currently limits our system to single-stroke gestures for the basic components. This means that the designer drawing the scrollbar in Figure 5 must use a single stroke of the pen for each of the two rectangles that comprise the scrollbar. We intend to develop a better algorithm later that will allow us to use multiple-strokes to draw our basic components.

Rubine's algorithm uses statistical pattern recognition techniques in order to train classifiers and then recognize gestures. These techniques are used to create a classifier based on the features extracted from several examples. For instance, the examples shown in Figure 6 were used to train the gesture classifier to recognize the tall, skinny rectangle in Figure 5. 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.

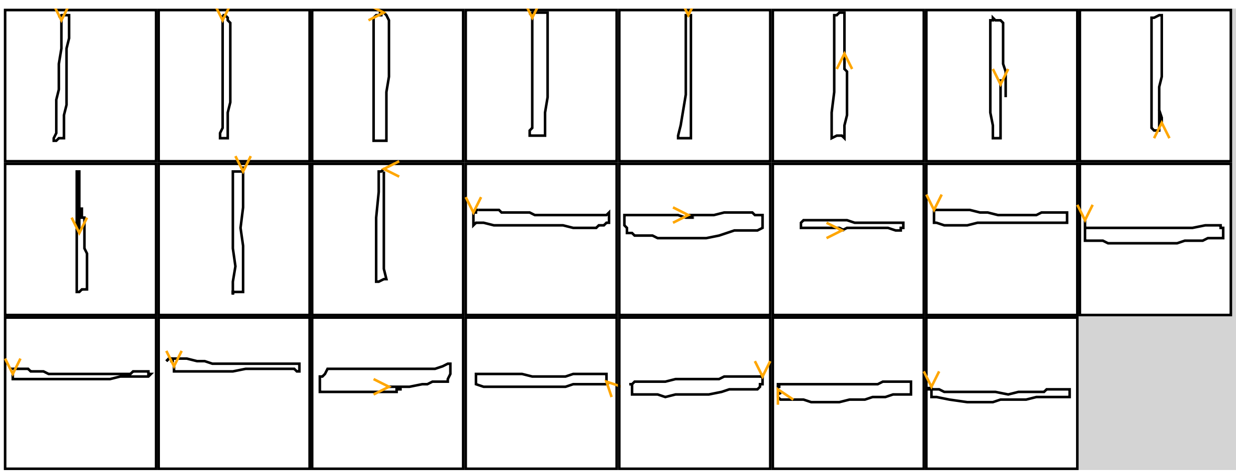

Figure 6: The examples used to train the classifier for recognizing a component of a scrollbar.

Composing Components

In order to recognize user 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 we pass it to an algorithm that looks for the following relationships:

- Does the new component *contain* or is it *contained by* another component.
- Is the new component *near* another component or widget.
- Is the new component in a *sequence* of components of the same type.

The first relationship is the most important for classifying most 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 that might have a box with some text next to it. The final relationship allows for groupings of related components that make up a widget (*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, it is assumed that there is not yet enough detail to recognize a 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. Similarly, using gestures to delete components of a widget can cause a new classification of the widget.

Each of the widgets that SILK recognizes have corresponding Garnet objects that use the Garnet interactor mechanism [17] to support interaction and feedback. When SILK identifies a widget, it attaches the sketched components that compose it to an instance of the object that implements the required interaction.

RELATED WORK

Wong's work in scanning in hand-drawn interfaces was the major impetus for starting our work in this area [29]. Wong's anecdotal evidence that colleagues give more useful feedback when evaluating interfaces with a sketchy look seems to be a commonly held belief in the design community. 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 formally show that Wong's anecdotal evidence is supported in practice.

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 the nice printouts produced by CAD tools before showing works in progress to their clients [27]. 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 drawings made with a CAD system 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 we are trying to recognize user interface elements rather than architectural elements. More importantly, our tool also allows the specification and testing of the *behavior* of the drawing, whereas the architectural drawing is fairly static.

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. It is implemented using Garnet [18]. The current implementation supports recognition and operation of several standard widgets, but does not allow the specification of behavior between the widgets (*i.e.*, the scrollbar can be scrolled but it can not yet be attached to the window containing data to scroll.) In addition, the history mechanisms for storing and annotating sketches have not yet been implemented. Finally, the system can not currently transform SILK's representation of the interface to a finished interface in a standard look-and-feel. This final portion of the system will be fairly easy to implement.

We are currently adding support for recognizing 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 the process of designing a formal study to compare the types of problems found when performing a heuristic evaluation on both sketchy and finished-looking interfaces.

CONCLUSIONS

We envision a future in which most if not all of the user interface code will be generated by user interface designers using tools like SILK rather than by having programmers write 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 user 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, the designer or test subjects will be able to interact with the sketch before it becomes a finished interface. We believe that, by supporting the entire interface design cycle with an interactive sketching tool, designers will be able to produce better quality interfaces in a shorter amount of time than they can currently.

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 Dale James for help with the writing of this paper.

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 (1994), 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,* British Computer Society, Sep 1992, pp. 137–153.

11. Kurlander, D. *Graphical Editing by Example*, Ph.D. dissertation, Columbia University, Department of Computer Science, 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. Lieberman, H. Mondrian: A Teachable Graphical Editor. In *Watch What I Do: Programming by Demonstration*. MIT Press, Cypher, A., Ch. 16, pp. 340–358, Cambridge, MA, 1993.

14. Maulsby, D.L., Witten, I.H., and Kittlitz, K.A. Metamouse: Specifying Graphical Procedures by Example. *Computer Graphics 23*, 3 (July 1989), 127–136, ACM SIGGRAPH '89 Conference Proceedings.

15. 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.

16. Myers, B.A. *Creating User Interfaces by Demonstration,* Academic Press, Boston (1988).

17. Myers, B.A. A New Model for Handling Input. *ACM Transactions on Information Systems 8*, 3 (July 1990), 289–320.

18. 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.

19. Myers, B.A. Demonstrational Interfaces: A Step Beyond Direct Manipulation. *IEEE Computer 25*, 8 (August 1992), 61–73.

20. 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.

21. P. Szekely, P. Luo, and Neches, R. Facilitating the Exploration of Interface Design Alternatives: The HUMANOID Model of Interface Design. In *CHI '92: Human Factors in Computing Systems,* Monterey, CA, May 1992, pp. 507–515.

22. 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.

23. Polya, G. *How to Solve It,* Princeton University Press, Princeton, New Jersey (1973).

24. Rettig, M. Prototyping for Tiny Fingers. *Communications of the ACM 37*, 4 (April 1994), 21–27.

25. Rubine, D. Specifying gestures by example. In *Computer Graphics,* Sederberg, T.W., ACM SIGGRAPH '91 Conference Proceedings, July 1991, pp. 329–337.

26. Singh, G., Kok, C.H., and Ngan, T.Y. Druid: A System for Demonstrating Rapid User Interface Development. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology,* Snowbird, UT, Oct 1990, pp. 167–177.

27. Strothotte, T., Raab, A., Preim, B., Schumann, J., and Forsey, D.A. How to render frames and influence people. In *Proceedings of Eurographics '94,* Oslo, Norway, To appear, 1994.

28. 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.

29. 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.

30. 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.