Just Draw It! Programming by Sketching Storyboards

James A. Landay and Brad A. Myers

27 November 1995
CMU-CS-95-199

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also appears as Human-Computer Interaction Institute Technical Report
CMU-HCII-95-106

Abstract

Current interactive user interface construction tools make it hard for a user interface designer to illustrate the behavior of an interface. These tools focus on specifying widgets and making it easy to manipulate details such as colors, alignment, and fonts. They can show what the interface will look like, but make it hard to show what it will do, since they require programming or scripting in order to specify all but the most trivial interactions. For these reasons, 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 have developed an interactive tool called SILK that allows designers to quickly sketch an interface using an electronic pad and stylus. However, unlike a paper sketch, this electronic sketch is *interactive.* The designer can illustrate *behaviors* by sketching *storyboards*, which specify how the screen should change in response to end-user actions. This paper describes our storyboarding mechanism and provides design ideas for a production-level system.

E-mail: landay@cs.cmu.edu
WWW Home Page: http://www.cs.cmu.edu/~landay

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, pen-based computing, SILK, visual languages, programming-by-demonstration.

INTRODUCTION

When professional designers first start thinking about a visual interface, they often sketch rough pictures of the screen layouts. These screens are often tied together by simple storyboarding techniques: the designer illustrates sequences of system responses to end-user actions by annotating the sketches to indicate these relationships. Figure 1 illustrates a simple sketched storyboard. The storyboard illustrates that the rectangle in the drawing window should be rotated when the button at the bottom of the screen is clicked.

Sequencing between screens by using hand drawn storyboards is a technique that has been shown to be a powerful tool for designers making concept sketches for early visualization [2]. In fact, all but one of the 16 designers we surveyed [12] claim to use sketches or storyboards during the early stages of user interface design. Storyboards are a natural representation, they are easy to edit, and they can easily be used to simulate functionality without worrying about how to implement it. In addition, the success of HyperCard has demonstrated that a significant amount of behavior can be constructed by sequencing screens upon button presses.

We have developed an electronic sketching tool called SILK which allows designers to illustrate these interface behaviors while the interfaces are still in their rough early stages. Last year we reported on the basic widget sketching interface for individual screens [12]. This year we have added a powerful storyboarding mechanism which allows a designer to specify the transitions between screens. The main advantage of our tool over paper sketches is that it allows the storyboards to come alive and permits the designer or test subjects to exercise the interface in this early, sketchy state. Buttons and other widgets were active in our previous system (*i.e.*, they would give feedback when clicked), but they could not perform any *actions*. Our new storyboarding component allows a wide variety of behaviors to be illustrated by sequencing screens on mouse clicks.

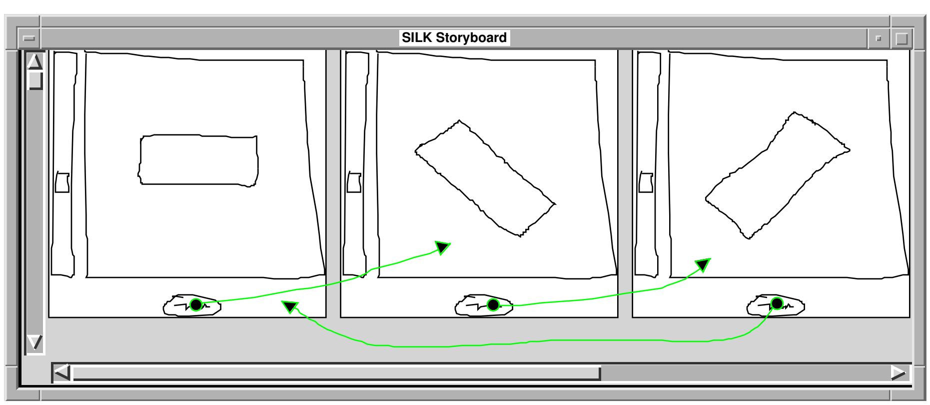

Figure 1: A storyboard that illustrates rotating a rectangle upon button presses.

SILK preserves the important properties of pencil and paper: a rough drawing can be produced *very quickly* and the medium is *very flexible*. SILK 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. At each stage of the process the interface can be tested by manipulating it with the mouse, keyboard, or stylus. Electronic sketches also have the advantages normally associated with computer-based tools: they are easy to edit, store, duplicate, modify, and search.

This paper describes how SILK can be used effectively by user interface designers to illustrate sequencing behaviors. The first section describes some of the problems associated with using current tools and techniques for specifying behavior. Next, we give an overview of SILK. In the third section, we describe the design and implementation of SILK's storyboarding mechanism. Next we describe some potential extensions to our system. Finally, we summarize the related work and the status of SILK to date.

DRAWBACKS OF EXISTING TOOLS

One of the major problems with existing user interface construction tools is the focus they place on the finished details of a user interface like color and alignment, rather than on the overall layout, structure, and interaction. This is significant during the early stages of development when it is important to quickly explore a wide range of designs. This focus problem is discussed in detail elsewhere [22, 1, 12].

When it comes to supporting interaction, existing tools fall short of the ideal. Most user interface builders, such as the NeXT Interface Builder [17] and Visual Basic, require the use of programming languages in order to specify any interaction beyond that of individual widgets. Design tools such as Director and HyperCard allow the sequencing of screens, and although they use direct-manipulation methods to specify these sequences, these methods lack the fluidity of paper-based storyboarding. For anything but the most simple sequences, these tools require the use of scripting languages.

Requiring the use of programming or scripting languages is not realistic for our application: the rapid prototyping of early user interface ideas by user interface designers. We have tried to design a system that allows the rapid illustration of a significant amount of interaction by sketching alone.

Due to the lack of good interactive tools, many designers use *low-fidelity prototyping* techniques [19]. These techniques involve creating mock-ups using sketches, scissors, glue, and post-it notes. One of the biggest drawbacks to using low-fidelity prototypes 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. Our system performs the screen transitions automatically in response to a user's actions. This allows more realistic testing of rough interface ideas. In addition, rather than being thrown out like paper prototypes, these electronic specifications may be used to automatically generate code to implement the transitions in the final system.

DESIGNING INTERFACES WITH SILK

Sketches are often used to explore the overall layout and structure of interface 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. Rather than just sketching many unrelated screens, designers often build up storyboards from these sketches. By numbering the screens, drawing arrows on them, and attaching annotations, a designer can describe the major transitions that occur between screens when a user manipulates the interface. A desire to iterate quickly leads designers to use paper-based sketches for this type of work.

Designers need tools that give them the freedom to sketch rough design ideas quickly [21], the capability to specify transitions between screens and behavior of interface elements, the ability to test the designs by interacting with them, and the flexibility to fill in the design details as choices are made. SILK was designed with these needs in mind. The rest of this section reviews the major components of SILK [12] and explains how it is used in practice.

Sketching Interfaces

SILK blends the advantages of both sketching and traditional user interface builders, yet it avoids many of the limitations of these approaches. Our goal was to make SILK's user interface as unobtrusive as pencil and paper. In addition to providing the ability to rapidly capture user interface ideas, SILK enables the designer to move quickly through several iterations of a design by using gestures to edit and redraw portions of the sketch. Changes and written annotations made to a design over the course of a project can also be captured and reviewed. Thus, unlike paper sketches, SILK sketches can evolve without forcing the designer to continually start over with a blank slate.

For individual screens, SILK tries to recognize user interface widgets and other interface elements *as they are drawn*. 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 and will be confined to the enclosing rectangle.

Specifying Behavior

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. For example, SILK knows how a button operates, but it cannot know what interface action should occur when a user presses the button. This is specified using the new storyboarding mechanism described in this paper. Storyboarding allows the specification of the *dynamic* behavior between widgets and the basic behavior of new widgets or application-specific objects, like a dialog box appearing when a button is pressed.

Our storyboarding technique uses a visual notation that is drawn on and between copies of the interface screens. Using a notation of marks that are made on the sketch is beneficial for several reasons. First, these sketchy marks are similar to the types of notations that one might make on a whiteboard or a piece of paper when designing an interface. For example, sequencing is expressed by drawing arrows from buttons to screens that appear when the button is pressed (see Figure 2). In addition, we can now use the same visual language for both the specification of the behavior and the static representation that can be later viewed and edited by the designer. We believe that this static representation is very natural and easy to use, unlike the hidden and textual representations used by other systems, such as HyperCard and Visual Basic.

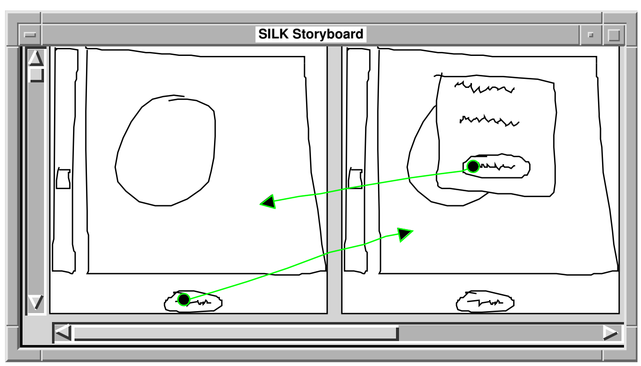

Figure 2: Make a dialog box appear when the button is pressed.

Run Mode

In addition to editing and creating new objects in sketch mode, SILK also supports run 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 buttons shown in Figure 1, the designer can switch to run mode and operate the buttons by selecting them with the stylus or mouse. The buttons will highlight when held down. With our new storyboarding component, the system will also make transitions to new screens when the buttons are pressed.

Transformation

When the designer is satisfied with the interface, SILK can replace the sketches with real widgets and graphical objects; these can take on the look-and-feel of a specified 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.

SILK STORYBOARDS

We have chosen a simple model for our storyboarding language. Our visual language has two types of objects, *screens* and *arrows*. Each screen is a sketch of an interface in a particular state. For example, Figure 1 illustrates three screens that differ in only the orientation of the rectangle in the drawing window. Arrows connect objects contained in one screen with a second screen. The arrow indicates that when the object in the first screen is manipulated (our current model is limited to mouse clicks), the system should display the second screen instead of the first. For example, the arrows in Figure 1 indicate that when the user clicks on the button at the bottom of the screen, the user should see the rectangle in its new (rotated) orientation.

Examples

SILK storyboards make it easy to illustrate several common interface behaviors. In this section we present several examples. In Figure 1 the designer has illustrated a repeating sequence of rectangle rotations. Each time the button is clicked, the rectangle in the drawing window rotates 60 degrees. This example also shows that the transitions can loop back to the screen they started on. An important point about this example is that it shows that a designer can illustrate a behavior (*i.e.*, rotation) that the underlying tools, SILK and the toolkit it uses, do not even support. The designer's knowledge and creativity allow the definition of behaviors that the underlying system does not support.

Figure 2 illustrates bringing up a dialog box on top of the sketched drawing window. This example is interesting since the designer is able make the dialog box opaque so that it hides any objects it may appear over. This technique can also be used for illustrating pull-down and pop-up menus. An alternative design we have considered is to have SILK recognize these types of widgets and set this property automatically.

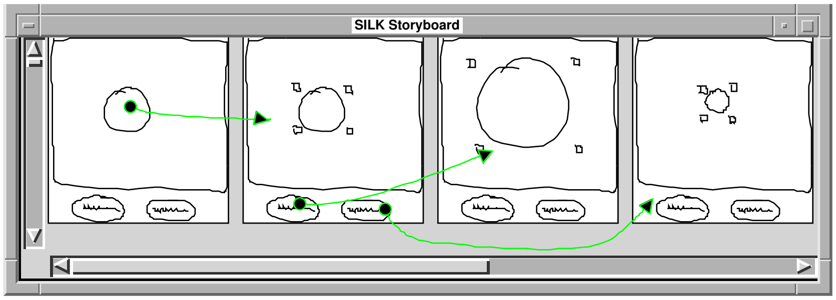

Figure 3: Scaling a circle with selection handles.

Figure 3 illustrates a sequence in which the user can select a circle by clicking on it. The circle can then either be doubled or shrunk in half by clicking on the buttons at the bottom of the screen. This example also lets the designer illustrate the feedback of selection handles.

Finally, Figure 4 illustrates how a designer could illustrate the operation of an arbitrary palette of tools. In this example the user is able to create any of three basic objects in a drawing window by first clicking on the object in the palette and then clicking in the drawing window.

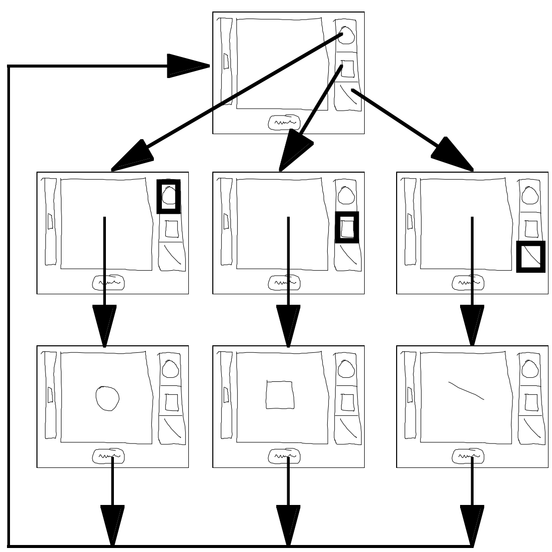

Figure 4: A partial screen tree for a simple drawing tool. Clicking on a palette item changes the state and then clicking on the background creates an object of the right type. The bottom button deletes the object.

Screen Trees

SILK's storyboarding model implies that a program can be thought of as a tree (see Figure 4). The nodes of the tree are the different states of the program (*i.e.*, screens) and the arcs out of each node represent the end-user actions that cause state changes. In order to fully specify a program, the designer would have to specify the entire tree. However, we do not believe this is a major drawback of our model, since storyboarding is generally used for illustrating important sequences in the interface, rather than for specifying an entire interface. For those that require more power, we propose several inferencing techniques in the next section which vastly reduce the amount of work needed to specify the screen tree.

The important sequences in an interface can be thought of as partial paths through the tree in Figure 4. For example, the middle path illustrates the sequence in which the user creates a rectangle and then deletes it by clicking on the delete button. By illustrating a few partial paths, the designer can specify enough of an interface for the design team to quickly consider several possible interactions.

Storyboard Construction

The system was designed to make specifying sequencing paths easy. The designer constructs storyboards by sketching screens in the SILK sketching window. These screens can be sketched with a pen-based graphics tablet or a mouse. Screens are then copied to the storyboard window. At this point, the original screen can be modified in the sketch window to show how its state might change. After this, the new screen is also copied to the storyboard window. Now, the designer can start drawing arrows in the storyboard window that indicate screen sequencing or more screens can be produced.

The arrows can be drawn from any widget, graphical object (*e.g.*, decorations), or the background to another screen. Thus, the designer can cause transitions to occur when the user clicks on any of these items. The arrows show an anchor point on the object they were drawn from and an arrowhead on the screen they are drawn to. Unlike the arrows in many visual dataflow languages [20] and CAD tools, our storyboarding arrows are free-form. This unconstrained control permits the designer to avoid some of the “rats-nest” problems associated with these other systems, where lines cross at 90 degree angles and are thus hard to follow.

Testing the Interaction

When the designer is ready to test the specified interaction, she can switch to run mode. At this point, the designer must specify which screen will be the initial screen to start the interaction. This is accomplished by selecting a screen in the storyboard window and copying it back to the sketch window. Now the designer or an end-user can start interacting with the sketch and it will make the proper transitions as defined by the visual program displayed in the storyboard window. Each time the user clicks on an object that has the source of an arrow attached to it, the system will replace the current screen with the screen attached to the head of the arrow. For example, the behavior illustrated by Figure 1 will show a progression of rectangle rotations when the user clicks on the button.

Algorithm Animation

In order to allow the designer to debug their storyboards, we have supplied some feedback mechanisms that are displayed while in run mode. First, the currently active screen (*i.e.,* the one being displayed in the sketch window), is always highlighted in the storyboard window. This allows the designer to know the current state of the system. Second, the object that caused the last transition to the current screen is highlighted along with the arrow leading to the current screen. A designer can use these mechanisms to help check that her visual program is working properly.

Implementation

As an outgrowth of our simple storyboarding model, the implementation is also quite straightforward. Every sketched widget and screen is given a unique ID. When a screen is copied from the sketch window to the storyboard window, the objects are copied along with their IDs.

When the system is put into run mode, a *screen transition table* is built by examining the arrows along with the objects and screens they connect. For each arrow in the storyboard the system creates a transition entry that contains the object's ID along with the origination and destination screen IDs. When the user clicks on an object in the sketch window, the system checks whether that object has a transition defined on it by looking it up by its ID and the ID of the current screen. If a transition is defined, the system copies the screen specified by the destination ID to the sketch window. Thus a program executes upon each input event by examining the transition table and copying the specified screen.

This implementation could be made more efficient (and exhibit less screen flicker) by checking for differences between the two screens and storing in the transition entry the necessary operators to effect the change. Instead of the outright replacement of objects that change, we could have the transition modify "interesting" object parameters, such as: visibility, left, top, width, height, and scale. Then when a transition fires, the system would only change parts of the screen rather than make an entirely new copy.

EXTENDING STORYBOARDS

Our current model of storyboard sequencing is sufficient for many applications, but falls short when the designer wishes to specify a complete interface. There are several ways we can extend our basic model to increase the power of this specification style.

Parallel Storyboards

Currently the system supports multiple paths through an interface screen tree (as seen in Figure 4), but only one of these paths can be followed at a time. An obvious extension of this is to implement the difference checking algorithm mentioned above and also allow multiple paths to be active at the *same* time. This implies that more than one state could be active, but only one in each independent path. This would allow the description of paths for manipulating different objects (for example the shapes in the palette of Figure 4) and allow those objects to be visible on the screen at the same time. For example, if we defined the two paths given in Figure 5, we would be able to produce the screen shown in Figure 6 by creating a rectangle followed by a circle or vice-versa. This allows the size of the specification to grow as the product of the number of objects and the number of supported operations on them. This is a significant improvement on our current model in which the specification grows exponentially.

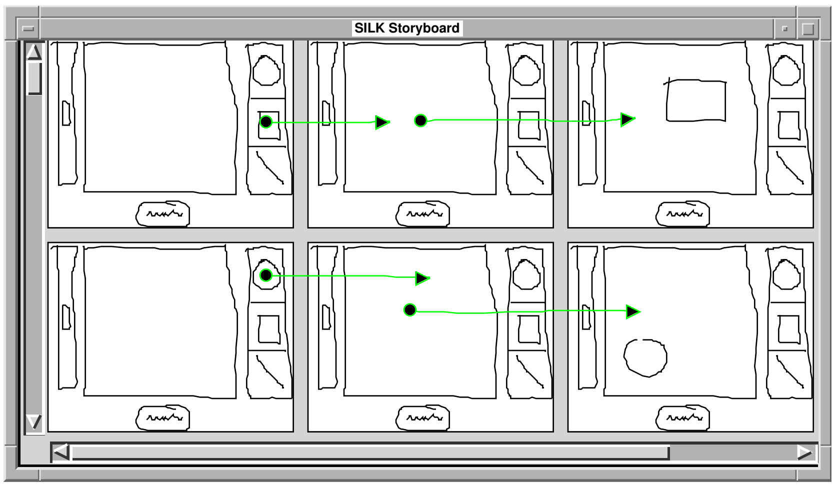

Figure 5: Parallel storyboard paths that allow the creation of circles and rectangles in any order.

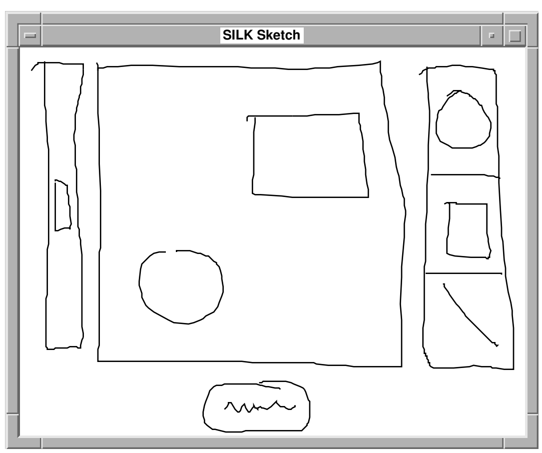

Figure 6: A drawing that could be created with the parallel storyboards defined in Figure 5.

Inferencing Techniques

Another way in which we can make storyboards more expressive is by applying inferencing techniques to them. Programming-by-demonstration (PBD) is a technique in which one specifies a program by directly operating the user interface [3, 15]. The system then tries to infer a program to implement the interaction. Our storyboarding system is similar to a PBD system. In the sketch window we specify the layout and structure of the interface, while in the storyboard window we demonstrate possible end-user actions and show how the layout and structure should change in response.

In our current system there is no inferencing involved. All actions and responses must be specified by the designer. Consider the rotation case illustrated in Figure 1. A PBD system could take the two rotation steps shown and try to infer the amount of rotation to apply on any subsequent button press. The PBD system could then indicate this inference by replacing the screens in question by one *compound* screen in which the inference is made explicit. Double clicking on the compound screen would display the original sequence as drawn. This would save both the designer's time and lots of valuable screen space.

Another way in which we can use inferencing is to allow the system to infer that operations applied to one type of object may also be applied to other objects. For example, in order to support scaling for both rectangles and circles the designer currently needs to specify two separate sequences operating on both types of objects. If the system could infer that scaling is simply the modification of a parameter of the selected object, then one example sequence would allow scaling on all objects. Again, this would save a considerable amount of designer time and storyboard space.

A critical problem with PBD techniques is the lack of a static representation that can be later edited. Marquise [16] and Smallstar [7] 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 believe that our visual notation can be extended to show the inferences that a PBD system might make.

It may also be useful to defer much of the inferencing until transformation time, thus preserving the fluidity of the sketching and brainstorming phases. At this point, PBD might also be used to help construct a call-back skeleton from the transitions.

Storyboard Space Saving Techniques

One of the major problems with any visual language is the large amount of screen space they require as compared to textual languages. Besides PBD, there are several other space-saving techniques we have considered to help overcome this problem.

The first major problem we have encountered is the "rats-nest" of arrows connecting screens. As an interface gets more and more complex, these arrows become hard to follow. This problem is solved in many CAD systems by performing automatic routing algorithms. This is especially useful after moving or deleting screens in the storyboard. We intend to allow multiple views of the storyboard window. For example, a designer should be able to specify that she wishes to only see arrows coming into a particular screen, out of a particular screen, or out of a particular object. Another view might only show screens that are *reachable* from the current screen or selected arrow. We believe that user-controlled routing and editing of free-form arrows, along with multiple views, can solve many arrow related problems.

The second major problem we have seen involves the size of the storyboarding panels. Currently when we copy a screen to the storyboard window, it is displayed at 50% of its original size. It may be useful to allow the designer to vary this parameter for more control over the space. It may also be nice to automatically vary the scale among different panels according to which part of the interface the designer is working on. This technique is similar to the use of fisheye views in visualization tools [5].

Another technique for saving on storyboarding space is to only show relevant changes to the screens, instead of the entire screens. In the rotation example (see Figure 1), the second storyboard screen might only show the rotated rectangle without the palette, window, or button. This is similar to some of the techniques used in Chimera [11] and systems based on graphical rewrite rules. Another way to conserve storyboard space is to try to compress multiple changes into a single screen. The Pursuit [13] system uses similar techniques. We could do that with our system if arrows were drawn not just to the next screen, but to the object that should be modified by the specified action. Thus, multiple arrows could arrive on distinct objects in the same screen to illustrates multiple state changes.

Additional Events, Animation, and State Matching

The only event the current system supports is clicking on widgets or graphical objects. We could specify more complete user interfaces by supporting dragging, drag and drop, double clicking, timer events, and typing. In addition, a null event arrow that would match on any user event that did not match on the current screen would make error handling easier. Allowing arrows to be annotated with event types could support some of these additional events. We envision supporting this by having a palette of arrow types in the storyboard window (*e.g.*, a mouse icon for single click, two mice for double click, a clock for timer events, *etc.*) The palette would allow the designer to select the current mode for the arrows drawn in the storyboard window. Arrows of different types would be distinguished by distinct colors and possibly line styles.

The timer event is especially interesting in that it would allow designers to quickly mock-up multimedia applications that have animation or video by specifying a few key frames that SILK would automatically transition between when the timer event occurred. Professional designers we have spoken with have requested this feature.

Another limitation of the current model is that it only supports checking that an event takes place on a particular object. It would be useful to have a transition that occurs conditionally on the object in question being in a particular state. For example, we may want a mouse click on a check box to only cause a transition if the box was previously unchecked.

RELATED WORK

We have drawn inspiration from several different systems in our design of SILK's storyboarding mechanism. HyperCard's in-place "card" transitions were a major influence on our transition scheme. Unlike HyperCard, SILK's transitions are visible and several can be viewed at once. Thus, our storyboards may be easier to understand and edit.

Marks or symbols layered on top of the interface are used for feedback indicating graphical constraints in Briar [6] and Rockit [9]. In Rockit, the marks kept the user informed of the current inference of the system. SILK differs in that the designer makes the marks, rather than the system.

Our storyboarding mechanism is based on specifying screen shots from before and after an end-user action. Chimera [11] and Pursuit [13] are both based on the before-and-after cartoon strip metaphor. SILK differs in that it allows the designer to specify what these actions are, rather than trying to infer this information from examples, as is done in Chimera and Pursuit. These systems are successful doing inferencing in their domains, graphical editing and graphical shell file operations, respectively.

SILK storyboards are similar to finite state transition diagrams. These diagrams have been used for UI specifications in the past, but have fallen into relative disuse because of problems with the exponential blow-up of the specification and an inability to handle the dynamic nature of direct manipulation user interfaces. Parallel storyboards try to attack the first problem in a way that is similar to Jacob's use of independent embedded state diagrams per object [8]. In addition, we do not believe designers will try to use SILK to specify an entire interface, but instead will concentrate on *simulating* key sequences.

Two other relevant end-user programming systems are Agentsheets [18] and KidSim [4]. Both of these systems use graphical rewrite rules to allow the creation of dynamic simulations. The rewrite rules specify a graphical pre-condition that must be met. When it is met by the state of the screen, the screen state is changed by the graphical action specified in the rewrite rule. These systems focus on animated simulations, whereas SILK concentrates on end-user actions and the changes to the UI state that should occur upon those actions.

Kramer's sketching system [10] is similar in its goal of supporting a very fluid free-form design system. Our systems differ in that SILK concentrates on user interfaces and allows the sketches to behave, whereas Kramer's system allows attaching many different "dynamic interpretations" to sketches, but supports only a limited set of actual interpreters.

Finally, Wong's work on scanning in hand-drawn interfaces was the major impetus for starting our work in this area [22]. Wong attached behaviors to her sketches with Director, whereas we give designers a *tool* that allows them to create both the *look* and *behavior* of these interfaces directly with the computer.

STATUS

SILK runs under Common Lisp on both UNIX workstations and the Apple Macintosh with a Wacom tablet attached. It is implemented using the Garnet User Interface Development Environment [14]. SILK supports the recognition and operation of several standard widgets and the transformation of the sketch to an interface with a Motif look-and-feel. The storyboarding mechanism supports the specification of screen trees, as described previously. SILK does not currently support parallel storyboards, inferencing, or any of the proposed screen space saving techniques. In addition, the system does not support any end-user actions other than clicking with the mouse. Thus, although the sketchy scrollbar in Figure 1 can be moved up and down, the designer cannot specify how much of the associated data should be scrolled.

We plan for design students to use SILK in a user interface design course to see how it performs in practice. In addition, we will be releasing the system for general use in early 1996[1]. In addition to collecting anecdotal comments from designers, we hope to follow a small set of designers intensively before and after using SILK to observe how their design methodology changes after using the system.

CONCLUSIONS

SILK storyboarding is a key step to a future in which much of a user interface will be illustrated, specified, and tested by a user interface designer. Through questionnaires and site visits we have found hand-drawn storyboards to be a common tool used to illustrate behavior. 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. Our interactive tool overcomes these problems by allowing designers to quickly sketch an interface using an electronic stylus. Unlike paper sketches, our electronic storyboards allow the designer or test subjects to interact with the sketch before it becomes a finalized interface. We believe that an interactive sketching tool will enable designers to produce better quality interfaces in a shorter amount of time than with current tools.

---

[1] For more information see the Web page of the first author.

ACKNOWLEDGMENTS

The authors would like to thank David Kosbie, Tom Moran, and Marc Ringuette for their helpful comments on this work. Finally, we would like to thank Richard McDaniel, Phoebe Sengers, and Bradley Vander Zanden for help with technical writing.

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. Cypher, A. *Watch What I Do: Programming by Demonstration,* MIT Press, Cambridge, MA (1993).
4. Cypher, A. and Smith, D.C. KidSim: End user programming of simulations. In *Proceedings of CHI '95: Human Factors in Computing Systems,* Denver, CO, 1995, pp. 27–34.
5. Furnas, G.W. Generalized fisheye views. In *Proceedings of ACM CHI '86 Conference on Human Factors in Computing Systems,* Boston, MA, 1986, pp. 16–23.
6. Gleicher, M. and Witkin, A. Drawing with constraints. *The Visual Computer 11*, 1 (1995), To appear.
7. Halbert, D.C. *Programming by Example*, Ph.D. dissertation, Computer Science Division, EECS Department, University of California, Berkeley, CA, 1984.
8. Jacob, R.J.K. A specification language for direct-manipulation user interfaces. *ACM Transactions on Graphics 5*, 4 (October 1986), 283–317.
9. 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, September 1992, pp. 137–153.
10. Kramer, A. Translucent patches – dissolving windows. In *Proceeings of UIST '94: Seventh Annual Symposium on User Interface Software and Technology,* Marina Del Rey, CA, 1994, pp. 121–130.
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. Interactive sketching for the early stages of user interface design. In *Proceedings of CHI '95: Human Factors in Computing Systems,* Denver, CO, May 1995, pp. 43–50.
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., 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.
15. Myers, B.A. Demonstrational Interfaces: A step beyond direct manipulation. *IEEE Computer 25*, 8 (August 1992), 61–73.
16. 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.

17. *NeXTStep and the NeXT Interface Builder*, NeXT, Inc., Redwood City, CA, 1991.

18. Repenning, A. *AgentSheets: A tool for building domain-oriented dynamic, visual environments*, Ph.D. dissertation, Dept. of Computer Science, University of Colorado at Boulder, 1993.

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

20. *Prograph*, TGS Systems, San Francisco, CA, 1992.

21. 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.

22. 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.