SATIN: A Toolkit for Informal Ink-based Applications

Jason I. Hong and James A. Landay

Group for User Interface Research, Computer Science Division

University of California, Berkeley

Berkeley, CA 94720-1776 USA

+1 510 643 7354

{jasonh, landay}@cs.berkeley.edu

ABSTRACT

Software support for making effective pen-based applications is currently rudimentary. To facilitate the creation of such applications, we have developed SATIN, a Java-based toolkit designed to support the creation of applications that leverage the informal nature of pens. This support includes a scenegraph for manipulating and rendering objects; support for zooming and rotating objects, switching between multiple views of an object, integration of pen input with interpreters, libraries for manipulating ink strokes, widgets optimized for pens, and compatibility with Java's Swing toolkit. SATIN includes a generalized architecture for handling pen input, consisting of recognizers, interpreters, and multi-interpreters. In this paper, we describe the functionality and architecture of SATIN, using two applications built with SATIN as examples.

Keywords

toolkits, pen, ink, informal, sketching, gesture, recognition, interpreter, recognizer, SATIN

INTRODUCTION

Sketching and writing are natural activities in many settings. Using pen and paper, a person can quickly write down ideas, as well as draw rough pictures and diagrams, deferring details until later. The informal nature of pens allows people to focus on their task without having to worry about precision.

However, although more and more computing devices are coming equipped with pens, there are few useful pen-based applications out there that take advantage of the fact that pens are good for sketching[1]. Most applications use pens only for selecting, tapping, and dragging. These applications simply treat the pen as another pointing device, ignoring its unique affordances.

Furthermore, the few compelling applications that do exist are built from scratch, despite the fact that many of them share the same kinds of functionality. This is because of the rudimentary software support for creating pen-based applications. Despite the fact that many new and useful pen-based interaction techniques have been developed, such as gesturing[1] and pie menus [5], these techniques have not yet been widely adopted because they are difficult and time-consuming to implement.

With respect to input and output for pens, we are at a stage similar to that of windowing toolkits in the early 1980s. Many example applications and many novel techniques exist, but there are no cohesive frameworks to support the creation of effective pen-based applications. As a first step towards such a framework, we have developed SATIN[2], a toolkit for supporting the creation of informal ink-based applications [15]. From a high-level perspective, there were three research goals for SATIN:

- Design a generalized software architecture for informal pen-based applications, focusing on how to handle sketching and gesturing in a reusable manner
- Develop an extensible toolkit that simplifies the creation of such informal pen-based apps
- Distribute this toolkit for general use by researchers

As a first step, we surveyed existing pen-based applications (both commercial and research) in order to determine what shared functionality would be most useful. Afterwards, we implemented the first iteration of the toolkit in Java, and built our first significant application with it, DENIM [26] (see Fig. 1). From the lessons learned, we developed the second iteration of SATIN, and built another application, SketchySPICE.

In this paper, we first outline functionality common in existing pen-based applications, and take a look at current software support for pen-based interfaces. We continue by describing the high-level and then detailed design of the SATIN toolkit. Specifically, we focus on a generalized architecture for handling pen input, consisting of three components: recognizers, interpreters, and multi-interpreters. We describe how pen input is handled in terms of the two applications, DENIM and SketchySPICE. We conclude with an evaluation of the toolkit, as well as our plans for future work and a discussion of lessons learned.

---

[1] By *sketching*, we mean the process of drawing roughly and quickly. We use the term *ink* for the strokes that appear. By *gesturing*, we mean a pen-drawn stroke that issues a command

[2] The SATIN project page and software download is at: `http://guir.berkeley.edu/projects/satin`
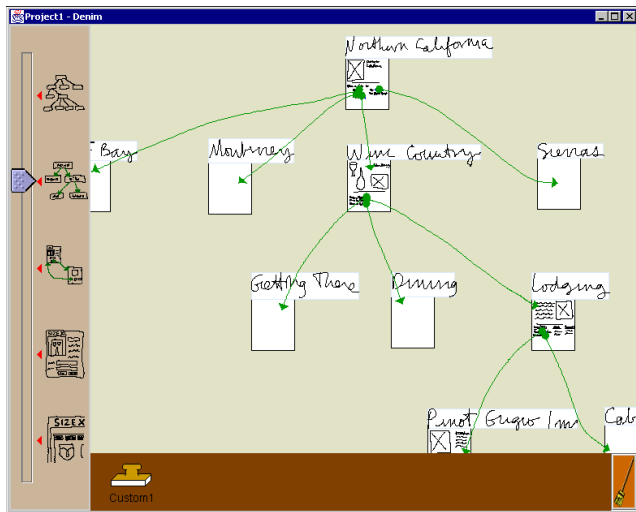
Figure 1 – A screenshot of DENIM, a sketch-based web site design tool created on top of SATIN

PEN APPLICATION SPACE

Recently, there have been many applications developed that use sketching and gesturing. We performed a survey of these applications, looking specifically for examples of informal ink-based interaction, ones that step away from rigid structure and precise computation, instead supporting ambiguity, creativity, and communication [15]. Many pen-based research systems have headed in the direction of informal interfaces in recent years, either by not processing the ink [11, 41, 43] or by processing the ink internally while displaying the unprocessed ink [14, 24, 32, 40].

The applications we examined include design tools [9, 12, 14, 20-22, 24, 43, 47]; whiteboard applications [1, 32, 33, 37]; annotation tools [41, 44-46]; note-taking applications [10, 11, 42]; and applications demonstrating new interaction techniques [19, 28, 40]. These applications share much functionality with each other, including:

- Pen input as ink
- Pen input as gestures
- Pen input for selecting and moving
- Interpreters that act on ink input
- Manipulation of other kinds of objects besides ink
- Grouping of objects
- Layering of objects
- Time indexing of ink input
- Transformation of ink to other cleaned-up objects
- Immediate and deferred processing of ink

Later, in the process of developing DENIM, our first application, we discovered we needed techniques for managing information, and turned to using zooming and semantic zooming, as demonstrated in Pad++ [3] and Jazz [4]. We decided that this functionality was useful enough to developers that it should be included in the toolkit.

EXISTING PEN FRAMEWORKS

In this section, we outline existing frameworks for developing pen-based applications, and describe where SATIN builds on their ideas.

Commercial Software Support for Pens

PalmOS [8] offers some very simple pen input processing. The default behavior is to process strokes and taps in the silk screen area as key events, with all other strokes passed on to the application for processing. PalmOS also provides some APIs for getting individual stroke points, enabling and disabling the Graffiti shorthand recognizer, and for getting the last known location of the pen.

Microsoft Windows for Pen Computing [29] provides minimal support for pens. Text entry areas were replaced either by handwriting edit controls (`hedit`) or by boxed edit controls (`bedit`), in which individual characters can be written. Simple gesture recognition was also supported. These extensions give the developer very little support for building informal ink-based applications.

In Windows CE [30], pen input is treated as a subset of mouse input. Applications can receive messages when the pen is moved, goes down, comes up, and is double-tapped. Windows CE also provides simple handwriting recognition.

NewtonOS [2] uses sheets of paper as its input metaphor. Users can write on these sheets without having to explicitly save. Furthermore, users can specify several ink modes in which strokes are processed as text, as shapes, or left unprocessed as raw ink. Recognition errors can be corrected by choosing from an n-best list. Gestures are also integrated into the system. Drawing a zig-zag shape over a word or shape, known as scrubbing, deletes that object. Holding down the pen for a second activates select mode. After select is enabled, the user can drag the pen and either highlight or circle the objects to select. Lastly, NewtonOS provides an extensive widget set for pens, designed to minimize the amount of end-user writing necessary.

Perhaps the most sophisticated commercial support for pens was in GO Corporation's PenPoint [6]. PenPoint is an operating system built from the ground up to support pens. Besides providing many of the services described above, such as gestures and pen widgets, PenPoint also has such features as live embedding of documents within documents, and extensive integration of gesture recognition and handwriting recognition.

There are two main differences between SATIN and the systems described above. First, all of the systems listed above are designed to build formal user interfaces, and are thus focused on handwriting recognition and form entry tasks. In contrast, SATIN is targeted towards the development of informal ink-based applications. The second difference is extensibility. Aside from handwriting recognition, the systems listed above provide minimal support for manipulating and processing ink. In contrast, one of our primary goals with SATIN was to give developers flexibility in how ink is processed and to make it simple to do so. For example, new gestures cannot be added in the systems described above.

Research Software Support for Pens

Simple ink and gesture support is provided in Artkit [16]. Artkit uses the notion of *sensitive regions*, invisible rectangles that can be placed on top of screen objects. The sensitive region intercepts stroke input, and processes the input in a recognition object, which possibly forwards a higher-level event to the screen object underneath.

Mankoff et al., extended the subArctic toolkit [17] to support inking, gesturing, and recognition, specifically for exploring techniques in resolving ambiguity [28].

Garnet [23, 34] and Amulet [36] also have support for gestures. A gesture interactor was added to these toolkits to support recognizing pen gestures using Rubine's algorithm [39]. The recognizer simply calls the registered callback procedure with the result as a parameter. No other pen and ink-based support is provided.

Flatland [18, 37] is a lightweight electronic whiteboard system that has much in common with SATIN. Flatland uses the notion of *segments* to divide up screen space, and uses *strokes* both as input and as output. Furthermore, *behaviors* can be dynamically plugged into segments, changing how stroke input is processed and displayed. This architecture is very similar to SATIN.

One clear difference between Flatland and SATIN is Flatland combines mechanism and policy in several cases, mixing *how* something is done with *when* it is done. For example, in Flatland, all strokes belong to a segment, and new segments are automatically created if a stroke is not drawn in an existing segment, whether or not an application designer wants a new segment. Our goal with SATIN was to focus on fine-grained mechanisms that can be used for a range of ink-based applications. Another difference is that Flatland only allows one application behavior to be active in a segment at any time. We introduce the notion of multi-interpreters to manage multiple interpreters.

Kramer's work in translucent patches and dynamic interpretations [21, 22] significantly influenced the design and implementation of SATIN. We use Kramer's notions of patches and dynamic interpretation, but again, our focus is at the toolkit level.

The chief characteristics that differentiate SATIN from all of the work above are flexibility and fine granularity. We are focused on developing an extensible toolkit. We provide a set of mechanisms for manipulating, handling, and interpreting strokes, as well as a library of simple manipulations on strokes, with which developers can build a variety of informal pen-based applications.

HIGH LEVEL DESIGN OF SATIN

SATIN is intended to support the development of 2D pen-based applications. We chose to support 2D instead of 3D since most of the applications surveyed utilize two dimensions only. The current implementation of SATIN does not support multiple users, as that introduces another level of complexity beyond the scope of this project.

SATIN is built in Java, using JDK1.3[3]. SATIN uses Java2D for rendering, and makes extensive use of the Java core classes as well as the Swing windowing toolkit [31].

Fig. 2 shows how a pen-based application would be built using SATIN, Swing, and Java. Roughly speaking, SATIN can be partitioned into twelve interrelated concepts (See Table 1). Each of these concepts is briefly summarized in the next section. Some of these concepts are very loosely coupled to one another, and can be used independently of the rest of the toolkit. In other words, a developer can use some portions of the SATIN toolkit without a complete buy-in of the entire system.
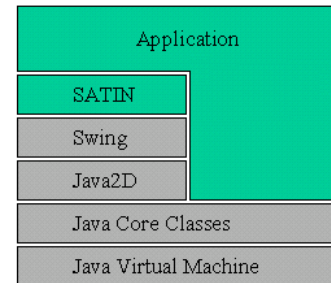

Figure 2 – This diagram shows the relationship between Java, Swing, SATIN, and pen-based applications.

Concept	Can use outside SATIN?	For pens only?
Scenegraph	No	No
Rendering	No	No
Views	No	No
Transitions	No	No
Strokes	Some portions	Yes
Events	No	Yes
Recognizers	Some portions	Yes
Interpreters	No	Yes
Clipboard	No	No
Notifications	Yes	No
Commands	Yes	No
Widgets	Yes	Yes

Table 1 – The twelve major components in SATIN. Some portions of SATIN have been designed to be independent of the rest of the system and can be used outside of SATIN.

Design Overview

We call objects that can be displayed and manipulated *graphical objects*. Like most 3D modeling systems (such as Java3D and OpenGL) we use the notion of a *scenegraph*, a tree-like data structure that holds graphical objects and groups of graphical objects. The simplest graphical object that the user can create is a *stroke*, which is automatically created in SATIN by the path drawn by a pen or mouse. Another primitive graphical object is a *patch*, an arbitrarily shaped region of space that can contain other graphical

---

[3] We began SATIN in JDK1.2, and transitioned to each early access version of the JDK as they were released.

objects. Patches *interpret* strokes either as gestures or as ink. Our notion of patches is derived from the work by Kramer [21, 22]. SATIN also provides a *sheet*, which is a Java Swing component as well as a graphical object. A Sheet serves as the root of a scenegraph, and is essentially a drawing canvas that can contain SATIN objects.

Graphical objects have x-, y-, and layer-coordinates. The x-axis and y-axis coordinates are Cartesian coordinates. The layer-coordinate is used to denote the relative position of one graphical object to another along the z-axis. That is, SATIN simply keeps track of which objects are on top of others, but does not store exact z-axis coordinates.

Graphical objects also have *styles*. Styles take many of the graphics concepts in Java, such as line style, color, and font, and translucency, abstracting them out into a single object. Styles are automatically applied by the rendering subsystem when rendering.

When *rendering*, SATIN uses the same damage-redraw cycle that is standard in windowing systems. The system never repaints a region unless it is marked damaged. If an area is damaged, then only the graphical objects in the damaged area are traversed. For common operations, such as translation and rotation, graphical objects automatically damage the region they are in. For application-specific operations, however, the developer may need to explicitly call the damage method.

SATIN also automatically changes the rendering quality depending on the current context. For example, when the user is drawing strokes, the damaged areas are rendered in low quality in order to speed up performance. However, when the stroke is completed, SATIN reverts to the highest quality rendering level.

Graphical objects have one or more *view* objects, which dictate how a graphical object is drawn. If a graphical object has more than one view, then it must also have a *MultiView*, an object that specifies the policy of which view objects are rendered and when. An example multi-view we have included is a Semantic Zoom Multi View, which uses the current zoom scale to choose the view to be displayed, as in Pad++ [3] and Jazz [4].

SATIN provides support for simple *transitions* on graphical objects, such as zooming and rotation. Given a graphical object and a transform, the system can automatically generate and render the intermediate steps, providing a smooth animation. The default transition type is Slow-In / Slow-Out [7, 25], a transition that spends the majority of time in the beginning and in the end of the animation.

There are also several classes for manipulating strokes. The *stroke assembler* aggregates user input into *strokes* and dispatches them as *events* to graphical objects. Each graphical object knows how to handle stroke events, and can choose how the stroke events are handled. This process is described in more detail in the Detailed Design section. There are also utility classes for manipulating strokes, such as splitting strokes, merging strokes, turning strokes into straight lines, and for simplifying strokes.

We use the term *recognizers* to mean subsystems used to classify ambiguous input, such as ink strokes. In SATIN, we have defined recognizers as objects that take some kind of ambiguous input and return a well-defined n-best list of classifications and probabilities ordered by probability. This definition allows us to plug in other stroke recognizers into the system. Examples of stroke recognizers include Rubine's recognizer [38, 39] and neural net recognizers. Currently, SATIN only contains the gdt [27] implementation of Rubine's recognizer. Recognizers may or may not retain state across classifications. However, recognizers do not take any kind of action based on the act of classification. Instead, this is left to interpreters.

Interpreters take action based on user-generated strokes. For example, one interpreter could take a stroke and transform it into a straight line. A different interpreter could issue a command if the stroke resembled a gesture in the system. Interpreters can use *recognizers* to classify strokes, but are not required to do so.

We distinguish between *gesture interpreters* and *ink interpreters*. A gesture interpreter tries to process a stroke as a command (e.g., cut), while an ink interpreter processes a stroke and displays the result as ink (e.g., straightens it out). We also make the distinction between *progressive-stroke interpreters* and *single-stroke interpreters*. A progressive-stroke interpreter tries to perform actions as a stroke is being drawn, while a single-stroke interpreter only takes action after a stroke is completed. SATIN currently does not support multi-stroke interpreters.

A graphical object can have one or more gesture interpreters, as well as one or more ink interpreters. Like views, a *MultiInterpreter* specifies the policy for which interpreters are used when more than one is present. Multi-interpreters are a new concept introduced in SATIN, and are discussed in the Detailed Design section.

The *clipboard* acts the same as in modern GUIs, supporting cut, copy, and paste for graphical objects.

Notifications are messages generated and sent internally within the system in order to maintain consistency. These messages are often used to maintain constraints between graphical objects or to notify objects that a graphical object has been deleted.

Commands are a common design pattern used for supporting macros, as well as undo and redo [13, 35]. Commands reify operations by encapsulating a transaction into an object that knows how to do, undo, and redo itself. SATIN's command subsystem extends the one provided in Java Swing (`javax.swing.undo`), by adding in the notion of executing a command (instead of simply undoing an operation). The command subsystem also has a notion of time, tracking when commands were executed, as well as allowing classes of commands to be enabled and disabled.

Application developers are not required to use the command subsystem in order to use SATIN. The Command subsystem can also be used outside of SATIN.

SATIN also provides some *widgets* optimized for pens. Currently, the only new widget we provide is a pie menu [5] that can be used as a normal Java Swing widget. The pie menu implements `javax.swing.MenuElement`, Swing's menu interface, and in many cases can be used in lieu of normal pop up menus with few changes to the code.

We also provide a Pen Pluggable Look and Feel (PenPLAF). The PenPLAF uses Java Swing's pluggable look and feel [31] to modify the standard file opener and slider widgets to make them easier to use for pens. The file opener was modified to accept single mouse clicks to open folders (instead of double clicks). The slider was modified to have a larger elevator, as well as the ability to have the slider value changed by tapping anywhere on the slider. The pie menu and the PenPLAF are not tied to SATIN, and can be used in Java applications outside of the toolkit.

Bridging the Gap between Java Swing and SATIN

We also provide some classes to help bridge the gap between SATIN and Java Swing (See Fig. 3). Currently, SATIN support for Swing consists of two classes. The first, `GObJComponent`[4], wraps up Swing widgets in a SATIN graphical object. Thus, Swing widgets can be displayed in SATIN, though full interaction (e.g., keyboard input), has not yet been completed. The second, `GObImage`, allows Java `Image` objects to be displayed in SATIN. This enables SATIN to be able to display any image file format that Java understands.

Conversely, SATIN can be used in Swing applications. As stated before, the Sheet is both the root of a scenegraph in SATIN and is a fully compatible Swing widget. A `JSatinComponent` is a Swing widget that wraps around a SATIN graphical object, letting SATIN graphical objects be displayed in Swing applications. Lastly, `SatinImageLib` provides some utilities for turning SATIN graphical objects into Java `Image` objects. This enables SATIN to be able to write out to any image file format that Java understands.

DETAILED DESIGN OF SATIN INK HANDLING

In this section, we describe strokes, recognizers, and interpreters in more detail, as well as how they interact with each other at runtime.

Strokes

In SATIN, strokes are simply a list of (x, y, t) tuples, where x is the x-coordinate, y is the y-coordinate, and t is the time the point was generated (since the Unix epoch).

SATIN also provides some utilities and interpreters for manipulating strokes, including splitting a stroke into

---

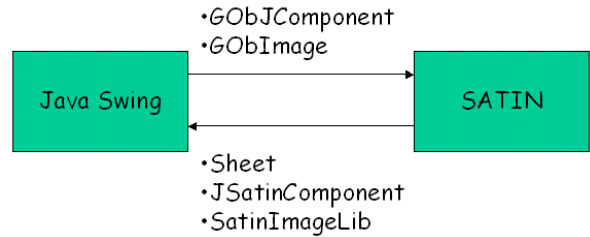[4] `JComponent` is the parent class of all Swing widgets.

Figure 3 – Classes bridging the gap between SATIN and Java Swing. Swing widgets can be displayed in SATIN, and SATIN graphical objects can be embedded in Swing applications.
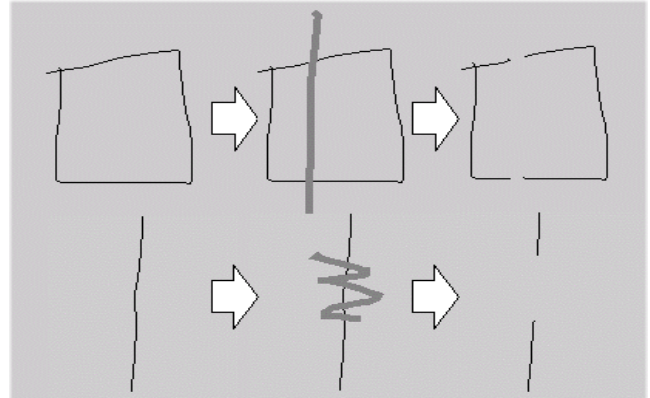

Figure 4 – Two example policies of splitting strokes. The thicker line is a gesture created by pressing the right button.
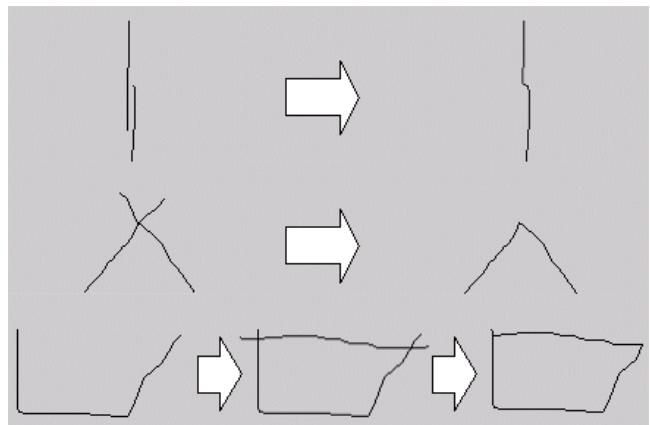

Figure 5 – At the top, two separate strokes near each other are combined into a single stroke. In the middle, two separate strokes that intersect near their endpoints are merged into a single stroke. At the bottom, two separate strokes that intersect near both of their endpoints are merged into a closed shape.
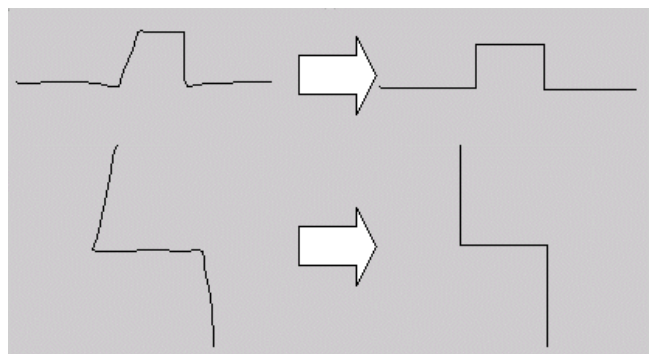

Figure 6 – Two examples of straightening strokes.

smaller substrokes, merging strokes together, straightening strokes into straight lines, and simplifying strokes.

Strokes can be split by specifying a rectangle in which all substrokes will be removed. Fig. 4 shows a sample interpreter that removes substrokes that lie in the bounding box of the gesture stroke.

Fig. 5 shows some examples of merging strokes. To see if two strokes can be merged, the algorithm first checks if the two strokes are near each other. If they are, then the algorithm checks if either extremity of one stroke is near an extremity of the other. If a successful match is made, then the two extremities are joined together in a new stroke, with short trailing ends discarded.

SATIN straightens strokes by changing strokes to lines that go up, down, left, or right (See Fig. 6). To straighten a stroke, we first examine each pair of adjacent points and classify each pair as going up, down, left, or right. For each subsequence of points that is going the same direction, we create a line that goes through the average value of that subsequence. After this is done, all of the lines created are joined together and returned as a new stroke.

Figure 7 – Two examples of stroke simplification. The algorithm generates a stroke similar to the original stroke, but has fewer points and can thus be rendered faster.

SATIN also provides utilities for simplifying strokes (See Fig. 7). This technique is automatically used to help speed up animated transitions. The following approach is used to simplify a stroke:

- For each point, calculate the absolute angle relative to the stroke's top-left corner using `atan2()`
- Calculate the angle delta between each adjacent pair of points
- Add the starting and ending point of the original stroke to the simplified stroke
- Go through the deltas and add each local minima to the simplified stroke

Once a stroke is simplified, it is cached in the system. On a sample set of fifty strokes, the number of points reduced ranged from 20% to 50%, averaging a 32% reduction. Using a battery of performance regression tests using 100 to 1000 strokes, the performance speedup[5] for animating simplified strokes ranged from 1.02 to 1.34, with an average speedup 1.11. Speedup improves somewhat linearly as the number of strokes is increased, as expected.

---

[5] Speedup $_{overall}$ = Execution time $_{old}$ / Execution time $_{new}$

Recognizers
In SATIN, a recognizer is a subsystem that classifies ambiguous input, which in our case are strokes. SATIN defines a standard interface for two types of recognizers: *progressive stroke* and *single stroke* recognizers. These definitions are not mutually exclusive, so a recognizer could be both a progressive and a single stroke recognizer. SATIN also defines a `Classification` object, which recognizers are defined to return when passed a stroke to classify. The classification is simply an n-best list of beliefs, ordered by probability. This definition for recognizers means that new recognizers can be plugged into the system simply by implementing the defined interface.

Interpreters
The class diagram in Fig. 8 illustrates the relationship between the classes used for interpretation, and shows some of the interpreters built in SATIN.

Besides processing strokes, interpreters are also stroke event filters, meaning they can specify what kinds of strokes they will accept. The simplest filter accepts or rejects strokes depending on which pen button was held when creating the stroke. Another kind of filter rejects strokes that are too long. In addition to filtering, individual interpreters can also be disabled, meaning that they will not process any strokes at all.

Some of the interpreters, on the right side of Fig. 8, have already been discussed (see above), or will be discussed with DENIM and SketchySPICE (next section). The more interesting part is the left portion of Fig. 8, which shows the multi-interpreters. Multi-interpreters are collections of interpreters combined with a policy that controls which interpreters are used and when they are used.

The default multi-interpreter is the Default Multi Interpreter, which simply calls all of the interpreters it contains, stopping when one of the interpreters says that it has successfully handled the stroke. The Multiplexed Multi Interpreter lets the developer specify one interpreter as active, which can be changed at runtime. The Semantic Zoom Multi Interpreter enables and disables interpreters depending on the current zoom level.

Runtime Handling of Strokes
Strokes are dispatched to graphical objects in a top-down manner: strokes are sent first to the parent before being re-dispatched to any of the parent's children. A stroke is re-dispatched to a child only if the child contains the stroke entirely (within a certain tolerance). By default, graphical objects handle strokes in a four-step process, as follows:

- Process the stroke with the gesture interpreters
- Re-dispatch the stroke to the appropriate children
- Process the stroke with the ink interpreters
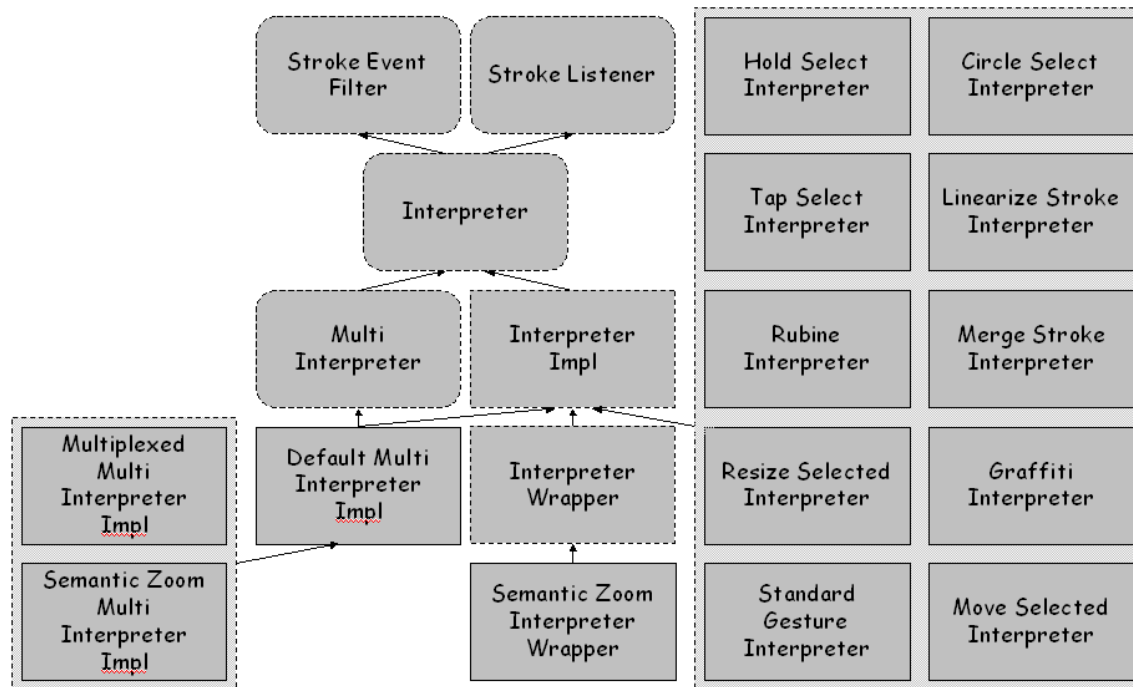- Handle the stroke in the graphical object

Figure 8 – Class diagram for Interpreters and Recognizers. Arrows point up towards parent classes. Rounded rectangles are interfaces; dashed square rectangles are abstract classes, and solid square rectangles are concrete classes.

At any point in this process, an interpreter or a graphical object can mark the stroke as being handled, which immediately stops the dispatching process. We give some examples of how strokes are handled in the DENIM and SketchySPICE sections below.

We chose this four-step approach as the default in order to separate handling of gestures from handling of ink. Processing gestures first lets gestures be global on the Sheet, or within a patch. This default approach can also be overridden in user code.

APPLICATIONS BUILT WITH SATIN

In this section, we describe two applications built using the SATIN toolkit, their high-level architectures, as well as how strokes are processed and interpreted in each.

First Application – DENIM

DENIM [26] is a web site design tool aimed at the early stages of information, navigation, and interaction design (See Figs. 1 and 9). An informal pen-based system [15], it allows designers to quickly sketch web pages, create links among them, and interact with them in a run mode. Zooming is used to integrate the different ways of viewing a web site, from site map to storyboard to individual page.

Although there are many gesture and ink interpreters in DENIM, from a user perspective, DENIM seems to use a minimal amount of recognition. Gestures are differentiated from ink by using the "right" pen button, while ink is created using the "left" button. This is the behavior we selected in DENIM, but can be modified in SATIN.

The scenegraph is comprised of five objects: the sheet, labels, panels, ink strokes, phrases, and arrows. The *sheet* is the root of the scenegraph. *Labels* are titles of web pages, for example "Lodging" and "Cabernet Lodge." Labels are

sticky, meaning that they are always displayed the same size, to ensure that they can always be read at the same size they were created. *Panels* are located beneath labels, and represent the content in a web page. *Ink strokes* are what are drawn in a panel. *Phrases* are collections of nearby strokes automatically aggregated together. *Arrows* connect ink and phrases from one page to another page.

Currently, DENIM only uses single stroke interpreters. All strokes are first passed through the Sheet's gesture interpreters, and then, if rejected by all of the gesture interpreters, are passed to the ink interpreters[6]. The gesture interpreters used in DENIM are all provided by SATIN, and include (in the order called):

- *hold select*, which processes a tap and hold to select shallowly if zoomed out (i.e. selects top-level scenegraph objects such as panels), or deeply if zoomed in (i.e. deeper level scenegraph objects, such as individual ink and phrases)
- *circle select*, which processes a circle-like gesture to select everything contained in the gesture (again shallowly or deeply depending on zoom level)
- *move*, in which all selected objects are moved the same distance the pen is moved
- *standard gesture*, which uses Rubine's recognizer [39] to recognize simple gestures like cut, copy, paste, undo, redo, and pan. Some gestures work shallowly if zoomed out, deeply if zoomed in.

---

[6] This is where the right and left button distinction is made. All gesture interpreters in DENIM only accept "right" button, and all ink interpreters only accept "left" button.
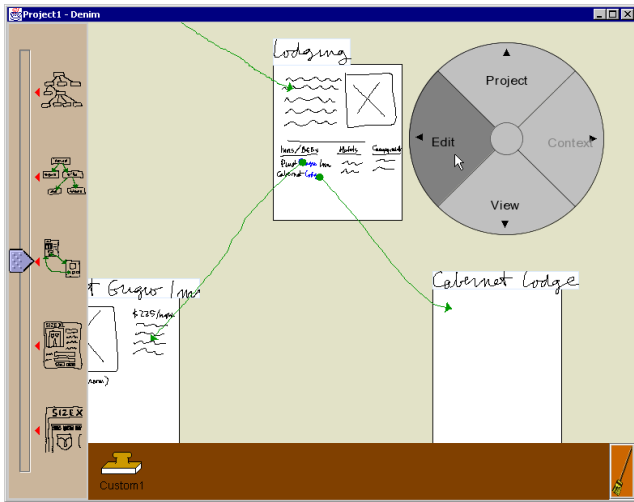
Figure 9 – A screenshot from DENIM, an application built on top of SATIN. This picture shows some ink, as well as the pie menu provided by SATIN. The Swing slider on the left is used to zoom in and out, and was modified by the PenPLAF to have a larger elevator, as well as the ability to have its value changed by taps anywhere on the slider.

If a stroke is not a gesture, then we check if the stroke should be re-dispatched to any of the Sheet's children, which in this case are labels and panels. A stroke is re-dispatched only if the label or panel bounds contain the stroke. If the stroke is re-dispatched to the label, then it is added to the label. If the stroke is re-dispatched to a panel, it is first processed by a *phrase* interpreter, which tries to group nearby ink strokes together in a single phrase object. Otherwise, it is just added to the panel as ink.

If the stroke is not re-dispatched, then the stroke is processed by the Sheet's ink interpreters. The ink interpreters are part of DENIM's code base, and include (in the order they are called):

- *arrow*, which processes lines drawn from one page to another, replacing the line by an arrow
- *label*, which processes ink that might be handwritten text, creating a new label & web page
- *panel*, which processes ink that resembles large rectangles, creating a new label and web page

If the stroke is not handled by any of the Sheet's ink interpreters, then it is just added as ink to the Sheet.

The pie menu is attached to the Sheet, and is activated by clicking the right button and not moving too far. We assigned this behavior so as not to interfere with gestures.

Second Application – SketchySPICE

SketchySPICE[7] is a simple circuit CAD tool intended as a demonstration of some features in SATIN (Figs. 10 and 11). Users can sketch AND, OR, and NOT gates, as well as wires connecting these gates. As proof-of-concept, AND and OR gates can be drawn in two separate strokes instead of just one, but this feature uses specific domain knowledge

---

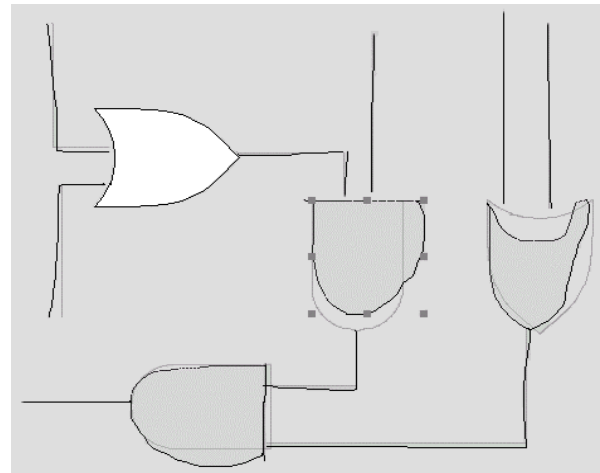[7] SPICE is a circuit CAD tool developed at UC Berkeley.
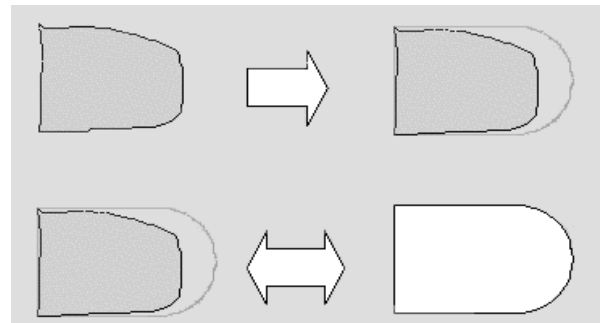

Figure 10 – A screenshot from SketchySPICE.

Figure 11 – SketchySPICE gives feedback by rendering the formal representation of the object translucently (top). An object can be displayed either in its original sketchy format, or in a cleaned-up format (bottom).

and is not part of SATIN. Once an object is recognized, SketchySPICE will take one of two actions, depending on the current mode. In *immediate* mode, recognized sketches are replaced immediately by a cleaned up version. In *deferred* mode, recognized objects are left sketchy, but feedback is provided to let users know that the object was recognized. This feedback consists of drawing the recognized object translucently behind the sketched object.

Individual gates can be selected and "cleaned up" to be displayed as formal looking gates, or can be "sketchified" and returned to their roughly drawn origins. In addition, the entire diagram can be cleaned up or sketchified.

The only new interpreter is the Gate interpreter. When a new stroke is added, the Gate interpreter looks at that stroke and the last stroke that was added. The two strokes are classified by Rubine's recognizer [39]. If the two separate classifications combined have a high probability of being a gate, then an AND Gate or an OR Gate object is added.

EVALUATION OF SATIN

SATIN has been in development for about two years, and is currently in its second iteration. There are about 20,000 source lines of code, and 13,000 comment lines of code, distributed in 2192 methods in 180 source code files. SATIN also uses debugging, collection, and string manipulation libraries developed by our research group, consisting of about 8000 source lines of code.

In contrast, DENIM, a fairly mature and large app, is only about 9000 source lines of code in 642 methods. The four interpreters in DENIM (arrow, label, panel and phrase) are only 1000 lines of code. Overall, it took three people three months to implement DENIM as described in [26].

SketchySPICE, a small proof-of-concept application, took about three days to implement. It is only 1000 lines of code in 32 methods. Half of the code is devoted to the pie menu, and 350 lines to the Gate interpreter.

	SATIN	DENIM	SketchySPICE
#source files	180	76	7
size of source files (kbytes)	1900	865	63
#methods	2192	642	63
#comments lines of code	13000	4500	400
#source lines of code	20000	9000	1000
#class files	220	131	32

Table 2 – Code size of SATIN and applications

Performance

We have used performance regression tests throughout the development of SATIN. The regression test suite is a repeated battery of operations, comprised of adding randomly generated graphical objects (always using the same seed value), zooming both in and out, and rotating. The regression tests were all run on the same computer, a Pentium II 300MHz running Windows NT 4.0 with a Matrox Millennium II AGP video card.

The overall performance speedup, from when the first regression test was run to when this paper was written, is 1.87. Approximately 54% of the speedup is due to code optimizations in SATIN, with the rest due to performance enhancements in the Java Virtual Machine. The two most significant gains came from polygon simplification and reduction of temporary objects generated.

FUTURE WORK

We are currently implementing a more extensive PenPLAF, which would make existing Java Swing applications more usable with pens. Besides eliminating the need for double-taps and making some widgets larger, we are also looking at integrating handwriting recognition and other interpreters with the existing Swing widgets.

Furthermore, we are working on making interpreters more sophisticated. For example, we are looking at mechanisms for adding in notions of time, to make it easy for developers to specify operations in which the pen must be held down for a period of time. We are also examining techniques to make it easier for developers to manage ambiguity. This ranges from implementing reusable, generic probabilistic data structures and algorithms, to interaction techniques, such as the mediators suggested by Mankoff [28].

SUMMARY

We introduced SATIN, a Java-based toolkit for developing informal pen-based user interfaces. By informal interfaces, we mean user interfaces that step away from the rigidity of traditional user interfaces, supporting instead the flexibility and ambiguity inherent in natural modes of communication. As a reusable toolkit, SATIN provides features common to many informal pen-based prototypes, including scenegraph support, zooming, multiple views, and stroke manipulation.

We have also described a generalized software architecture for informal pen-based applications that can handle sketching and gesturing in an extensible manner. This architecture consists of separating recognizers, which are components that classify strokes, from interpreters, which are components that process and manipulate strokes. Furthermore, multi-interpreters allow developers to specify policies of which interpreters are used and when they are used. Combined together, these features in the SATIN toolkit simplify application implementation.

With respect to input and output for pens, we are at a stage similar to that of windowing toolkits in the early 1980s. There are many bits and pieces here and there, but no cohesive frameworks to support the creation of effective informal pen-based applications. We hope that SATIN will be a significant step towards creating such a framework.

SATIN has been publicly released and can be found at: `http://guir.berkeley.edu/projects/satin`

ACKNOWLEDGEMENTS

REFERENCES

1. Abowd, G., et al. Investigating the Capture, Integration and Access Problem of Ubiquitous Computing in an Educational Setting. In Proceedings of *CHI '98*. Los Angeles, CA. pp. 440-447, April 18-23 1998.
2. Apple, Newton Toolkit User's Guide. 1996.
3. Bederson, B.B. and J.D. Hollan. Pad++: A Zooming Graphical Interface for Exploring Alternative Interface Physics. In Proceedings of *the ACM Symposium on User Interface Software and Technology: UIST '94*. Marina del Rey, CA. pp. 17-26, November 2–4 1994.
4. Bederson, B.B. and B. McAlister, *Jazz: An Extensible 2D+Zooming Graphics Toolkit in Java*. Tech Report HCIL-99-07, CS-TR-4015, UMIACS-TR-99-24, University of Maryland, Computer Science Dept, College Park, MD 1999.
5. Callahan, J., et al. An Empirical Comparison of Pie vs. Linear Menus. In Proceedings of *Human Factors in Computing Systems*. pp. 95-100 1988.
6. Carr, R. and D. Shafer, *The Power of PenPoint*: Addison-Wesley, 1991.
7. Chang, B. and D. Ungar. Animation: From Cartoons to the User Interface. In Proceedings of *UIST'93*. Atlanta, GA: ACM Press. pp. 45-55 1993.

8. Palm Computing., Developing Palm OS 2.0 Applications.

9. Damm, C.H., K.M. Hansen, and M. Thomsen. Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard. CHI Letters: Human Factors in Computing Systems, CHI '2000, 2000. 2(1): p. 518-525.

10. Davis, R.C. and J.A. Landay, Making sharing pervasive: Ubiquitous computing for shared note taking. *IBM Systems Journal*, 1999. **38**(4): p. 531-550.

11. Davis, R.C., *et al.* NotePals: Lightweight Note Sharing by the Group, for the Group. In Proceedings of *CHI '99*. Pittsburgh, PA. pp. 338-345, May 15-20 1999.

12. Forsberg, A., M. Dieterich, and R. Zeleznik. The Music Notepad. In Proceedings of *UIST98*. San Francisco: ACM Press 1998.

13. Gamma, E., et al, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

14. Gross, M.D. and E.Y. Do. Ambiguous Intentions: A Paper-like Interface for Creative Design. In Proceedings of *ACM Symposium on User Interface Software and Technology*. Seattle, WA. pp. 183-192, November 6–8 1996.

15. Hearst, M.A., M.D. Gross, J.A. Landay, and T.E. Stahovich. Sketching Intelligent Systems. *IEEE Intelligent Systems*, 1998. **13**(3): p. 10-19.

16. Henry, T.R., S.E. Hudson, and G.L. Newell. Integrating Gesture and Snapping into a User Interface Toolkit. In Proceedings of *UIST90*: ACM Press 1990.

17. Hudson, S.E. and I. Smith. Ultra-Lightweight Constraints. In Proceedings of *UIST96*: ACM Press 1996.

18. Igarashi, T., et al. An Architecture for Pen-based Interaction on Electronic Whiteboards. To Appear In Proceedings of *Advanced Visual Interfaces*. Palermo, Italy May 2000.

19. Igarashi, T., S. Matsuoka, S. Kawachiya, and H. Tanaka. Pegasus: A Drawing System for Rapid Geometric Design. In Proceedings of *CHI98*. Los Angeles: ACM Press. 1998.

20. Igarashi, T., S. Matsuoka, and H. Tanaka. Teddy: A Sketching Interface for 3D Freeform Design. In Proceedings of *ACM SIGGRAPH99*. Los Angeles: ACM Press. pp. 409-416 1999.

21. Kramer, A. Dynamic Interpretations in Translucent Patches. In Proceedings of *Advanced Visual Interfaces*. Gubbio, Italy, 1996.

22. Kramer, A. Translucent Patches – Dissolving Windows. In Proceedings of *ACM Symposium on User Interface Software and Technology*. Marina del Rey, CA. November 2–4 1994.

23. Landay, J.A. and B.A. Myers. Extending an Existing User Interface Toolkit to Support Gesture Recognition. In Proceedings of *INTERCHI '93*. Amsterdam, The Netherlands. pp. 91-92, April 24–29 1993.

24. Landay, J.A. and B.A. Myers. Interactive Sketching for the Early Stages of User Interface Design. In Proceedings of *CHI '95*. Denver, CO. pp. 43-50, May 7–11 1995.

25. Lassiter, J. Principles of Traditional Animation Applied to 3D Computer Animation. In Proceedings of *ACM SIGGRAPH '87*: ACM Press. pp. 35-44 1987.

26. Lin, J., M. Newman, J. Hong, and J. Landay. DENIM: Finding a Tighter Fit Between Tools and Practice for Web Site Design. CHI Letters: Human Factors in Computing Systems, CHI '2000, 2000. 2(1): p. 510-517.

27. Long, A.C., J.A. Landay, and L.A. Rowe. Implications For a Gesture Design Tool. Proceedings of *CHI '99*. Pittsburgh, PA. pp. 40-47, May 15-20 1999.

28. Mankoff, J., S.E. Hudson, and G.D. Abowd. Providing Integrated Toolkit-Level Support for Ambiguity in Recognition-Based Interfaces. CHI Letters: Human Factors in Computing Systems, CHI '2000, 2000. 2(1): p. 368-375.

29. Microsoft, *Microsoft Windows for Pen Computing - Programmer's Reference Version 1*: Microsoft Press, 1992.

30. Microsoft, MSDN Library: Windows CE Documentation. http://msdn.microsoft.com/library/default.asp

31. Sun Microsystems. Java Foundation Classes. http://java.sun.com/products/jfc

32. Moran, T.P., P. Chiu, and W. van Melle. Pen-Based Interaction Techniques For Organizing Material on an Electronic Whiteboard. In Proceedings of *UIST '97*. Banff, Alberta, Canada. pp. 45-54, October 14-17 1997.

33. Moran, T.P., P. Chiu, W. van Melle, and G. Kurtenbach. Implicit Structures for Pen-Based Systems Within a Freeform Interaction Paradigm. In Proceedings of *CHI'95*. Denver, CO. May 7–11 1995.

34. Myers, B. *et. al.*, Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer*, 1990. **23**(11): p. 289-320.

35. Myers, B. and D. Kosbie. Reusable Hierarchical Command Objects. In Proceedings of *CHI'96*. Vancouver, BC, Canada: ACM Press 1006.

36. Myers, B., *et al.*, The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering*, 1996. **23**(6): p. 347-365.

37. Mynatt, E.D., T. Igarashi, W.K. Edwards, and A. LaMarca. Flatland: New Dimensions in Office Whiteboards. In Proceedings of *CHI99*. Pittsburgh, PA: ACM Press 1999.

38. Rubine, D., *The Automatic Recognition of Gestures*, Unpublished Ph.D. Carnegie Mellon, Pittsburgh, PA, 1991.

39. Rubine, D., Specifying Gestures by Example. *Computer Graphics*, 1991. **25**(3): p. 329-337.

40. Saund, E. and T.P. Moran. A Perceptually-Supported Sketch Editor. In Proceedings of *the ACM Symposium on User Interface Software and Technology: UIST '94*. Marina del Rey, CA. pp. 175-184, November 2–4 1994.

41. Schilit, B.N., G. Golovchinsky, and M.N. Price. Beyond Paper: Supporting Active Reading with Free Form Digital Ink Annotations. In Proceedings of *CHI '98*. Los Angeles, CA. April 18-23 1998.

42. Truong, K.N., G.D. Abowd, and J.A. Brotherton. Personalizing the Capture of Public Experiences. In Proceedings of *UIST'99*. Asheville, NC 1999.

43. van de Kant, M., et al. PatchWork: A Software Tool for Early Design. In Extended Abstracts of *CHI '98*. Los Angeles, CA. pp. 221-222, April 18-23 1998.

44. Weber, K. and A. Poon. Marquee: A Tool for Real-Time Video Logging. In Proceedings of *CHI '94*. Boston, MA. pp. 58-64, April 24-28 1994.

45. Whittaker, S., P. Hyland, and M. Wiley. Filochat: Handwritten Notes Provide Access to Recorded Conversations. In Proceedings of *CHI '94*. Boston, MA. April 24-28 1994.

46. Wilcox, L.D., B.N. Schilit, and N.N. Sawhney. Dynomite: A Dynamically Organized Ink and Audio Notebook. In Proceedings of *CHI'97*. Atlanta, GA. pp. 186-193, March 22-27 1997.

47. Zeleznik, R.C., K.P. Herndon, and J.F. Hughes, SKETCH: An Interface for Sketching 3D Scenes. *Computer Graphics (Proceedings of SIGGRAPH '96)*, 1996.