Developer Paradigms and User Interaction in Sketch-Based Systems

Shane Zamora
University of California, Santa Barbara
char42@cs.ucsb.edu

Jeff Browne
University of California, Santa Barbara
jbrowne@cs.ucsb.edu

André Sayre
University of California, Santa Barbara
asayre@cs.ucsb.edu

Tim Sherwood
University of California, Santa Barbara
sherwood@cs.ucsb.edu

Abstract

Sketch-based systems must overcome some serious obstacles if they are ever going to see widespread adoption. In this paper we discuss several of the most important obstacles our group encountered in trying to develop new applications in this space, including the ability to incrementally and modularly develop rich applications, the graceful handling of errors, and the challenges that exist with the current generation of available hardware and system support. In particular we describe some of the ergonomic issues tablets currently face, the troubles we had in managing the many different types of errors possible in sketch-based systems, and the engineering issues that motivated us to explore alternative software architectures to more readily encourage code reuse and collaborative development.

Keywords

Sketch recognition, user interfaces

ACM Classification Keywords

D.2.2 Design Tools and Techniques, H.5.2 User Interfaces

Introduction

In recent years many research projects have been developed in the field of sketch recognition interface design. However, because of the work involved with their initial development, the effort required to prepare a recognition interface for public release, and a lack of precedent on what is expected from commercial sketch-based applications, many of these projects have remained research prototypes. In order for the number of quality sketch applications to expand, there must be a sizeable development community driving innovation. A key factor in such a community's growth is the capability of building on top of previous work, one that is missing from current generation sketch applications.

Additionally, a strong user-base is essential to motivating more and better applications. Unfortunately, current users are reluctant to adopt sketch-based applications due to their unfamiliarity with the pen as a primary input device. The problem is exacerbated due to the fact that current desktop environments are poorly suited for pen interaction. As long as pen-based interfaces are viewed as a secondary mode of interaction with a computer, the user base for sketch applications will remain relatively small.

We argue in this paper that a ”killer app” for sketch recognition must deliver a lower barrier of entry for both users and developers. First, we propose a standard development framework to encourage programmers to build on each others' work in a logical manner. Next, we discuss the failings of tablet hardware from a user perspective, and finally propose a specially tailored desktop environment for sketch-based interaction.

Current Developer Support

Currently, developing sketch applications involves reimplementing large amounts of code for each project. Because so many applications are constrained to a very specific domain (e.g. math equation, or physics diagram recognition), these functions are often tailored uniquely to their domain. Even if a function is conceptually shared between two applications, the code that is ”reused” must be largely rewritten. On top of this, programmers must also define their own error semantics, such that exceptions can be handled cleanly and consistently.

We argue that, in order to spur developers to write ”killer apps,” there must be a development framework to solidify error semantics and facilitate code reuse. Quality applications will implement these end goals anyway, but an application agnostic framework would encourage less technical developers to implement their ideas.

Code Reuse

A glaring weakness in current sketch applications is their tendency to be ”one-off” programs, finely tuned to support only a very constrained application domain. Thus, even if a separate application requires much of the same functionality as another, code cannot generally be shared unmodified (if at all). This inelegant code copying is rife with opportunities for introducing bugs, but moreover, novice developers will give up on working with sketch instead of investing the time to understand all low-level details of sketch recognition.

Instead of tailoring each standalone program to a single application, we propose that developers will be greatly served by a framework that separates recognition tasks into interacting ”apps,” which could be composed to provide more complex recognition, while the framework would provide the most common app functionality, such as raw stroke input. Developers could build apps without any knowledge of their lower-level implementation details and could instead focus on their higher-level strengths. A community of clever programmers could build on each others' work for common tasks, and like domain-specific code libraries, sketch apps could be optimized behind the scenes by their maintainers. For example, one group could maintain high quality circle and line apps, which another group composes into a generic graph recognizer. The graph recognizer could then be extended by a new directed-graph recognizer, which a Markov model or a finite state machine app could further build upon for their recognition tasks.

Modularization along these lines greatly reduces the learning curve for producing sketch recognition applications. For example, if there already exists effective graph topology recognizers and character recognizers, it is a simple matter of composing these two apps into a finite state machine recognizer and simulator. With a lower barrier to entry, clever ideas for killer apps can originate in developers unfamiliar with advanced solutions to the difficulties in low-level recognition.

Defining Error Semantics

In order to standardize application development in the sketch recognition domain, it is necessary to classify different types of errors in order to predictably correct them. In this section, we define several classes of errors that are meaningfully distinct: system errors, recognition errors, and errors according to application semantics.

We define system errors to be those that arise from implementation problems that leave the sketch system in an inconsistent state where strokes and their interpretation may differ depending on the observer. For example, with multiple simultaneous apps running, one of them may remove a stroke from the board, and without some facility for notifying other apps, the system will become inconsistent.

System errors can be difficult to address, since the underlying debugging system is just another observer. However, since these errors are generally detrimental to the entire system, they should be diagnosed and fixed as soon as they are introduced. Also, because they are infrequent, debugging via examining the entire system state may be a reasonable approach.

Recognition errors occur within a consistent sketch application, but involve misclassifying the user's intended stroke. For example, a system may recognize a stroke intended to be the letter ”q” as the letter ”g”, but this would not be a system error as the stroke is seen as a ”g” from every perspective. Debugging recognition errors during app development can be verbose, but if a user is interacting with the system, too much information can be just as detrimental as too little.

Correcting recognition errors could be done automatically, but this requires some notion of the application's domain in order to accurately guess the user's intent. A good development framework should provide facilities to correct users' strokes to their best fit, but in a general use sketch system where many apps interact with equal authority, a recognition error to one app may be acceptable to another app. For example, a dashed-line recognizer may see a bit of Morse code as messy, and cleaning up the strokes would be incorrect. In such a system with an unbounded number of domains, the best we can do is to carefully filter notifying the user about recognition errors.

Given that a system is consistent, and strokes are all recognized as intended, there still exists the possibility that an application may disallow some nonsensical inputs. For example, a circuit diagram app needs to degrade gracefully when a user draws more than one input to a NOT gate. How this happens depends entirely on the application's semantics, so the ultimate corrective action should be entirely up to the app developer. The development framework would provide this by allowing the app to remove the stroke from the board, correct the stroke, or display some information to the user (e.g. highlighting, textual output). However, this must be done through an interface provided by the framework, so that other apps can maintain a consistent view of the board.

The issues with error semantics raised in this section may not capture all errors that arise with every sketch application, but regardless of the specific errors, it is important that the developer has a standard way to handle exceptions. Standardization is key to an understandable system, since the error handling mechanism provided by the framework must work well across distinct apps that are simultaneously interacting with the user.

User Adoption

User studies in multimodal interfaces consistently show users as reluctant to adopt unfamiliar modalities, preferring familiar input methods whenever available, such as the mouse and keyboard. While the stylus is not considered as a primary input method it enjoys a unique position in the market of alternative input devices where tablet PCs are an established category of laptop computers. Once the "killer application" for sketch-based systems is found, there is a realistic potential for an explosion in market penetration for devices with integrated tablets.

However, any strong building must be built on a solid foundation. The ultimate objective of any such killer application will be to motivate users to adopt the pen as a primary input device. In order to accomplish this goal, critical usability issues of tablet-based computing need to be addressed regarding the hardware itself and its primary software package: the operating system.

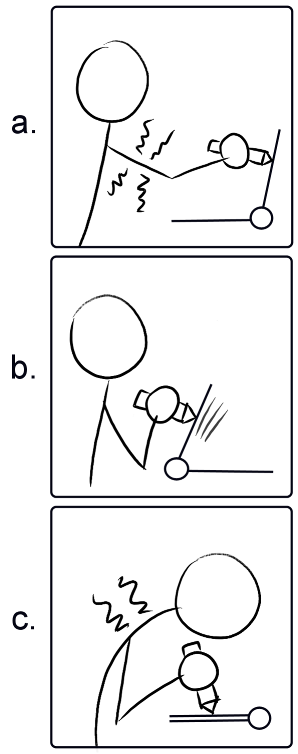

Figure 1: Ergonomic issues involved with pen-based interaction with a tablet PC. When sketching in the laptop configuration (a) the angle and distance to the screen, as well as trying to avoid hitting keys on the keyboard, can cause strain. In the easel configuration (b) the pressure of the pen can wobble the screen and interrupt interaction. In the slate configuration (c) poor viewing angles can require the user to hover over the screen. Additionally, the thickness of tablet impedes the user from resting their forearm on the desk.

Hardware Usability

The most suitable form factor for our killer application is the "convertible" style tablet PC (henceforth referred to as a tablet PC). These tablets act as standard laptop computers with a tablet integrated into a reversible display that can collapse and "convert" the computer into a slate form factor. By offering a superset of hardware functionality to a standard laptop (leveraging an established form factor as to not alienate users), and providing direct pen input by integrating the tablet into the LCD screen, the tablet PC is an ideal platform for sketch-based applications.

Other form factors of tablets are not as suitable for our killer application. Large desktop tablet displays are used by professionals who rely heavily on pen-based interfaces and do not need to be motivated to adopt them. With the exception of researchers looking for hardware to test their sketch recognition systems, standalone tablets such as the Wacom Bamboo are purchased almost exclusively by art enthusiasts for use in programs such as Photoshop and are not ideal for pen-based interaction due to their indirect nature. Users are likely to be unwilling to purchase such a tablet to adopt any killer application for a desktop or laptop without integrated tablet hardware.

However, tablet PCs do have significant ergonomic nuances to their operation, illustrated in figure 1. When a laptop is used at a comfortable distance for typing with the screen at an upright angle, reaching forward to sketch for any period of time can be fatiguing (1a). With the screen reversed and accessed as an easel, this fatigue is reduced at the cost of access to the keyboard. However, pen strokes push against the screen and can cause wobble when the pressure overcomes the resistance of the screen's hinge (1b). Using the tablet as a slate overcomes the wobble but introduces other issues – the thickness of the tablet PC can be an issue on desks built at heights for writing on paper, the angle of the screen can exacerbate glare from overhead lighting, and poor viewing angles reduce visual fidelity unless the user hovers directly over the screen (1c). Finally, the hand and lower arm of the user can obstruct the screen during sketching and impact usability.

Desktop Usability

Most tablet PCs run Microsoft Windows, an operating system heavily optimized for interaction with a mouse and keyboard. When a consumer is deciding between a tablet PC and regular laptop, the question is ultimately "Do I want to add a stylus to Windows?" and whether a pen can enrich the user's interaction experience.

Unfortunately, even simple tasks such as double clicking and hierarchical menu navigation can quickly become awkward with a pen in WIMP-based GUIs. The primary tasks expected of a modern PC are web browsing, media playback, text editing, and file manipulation – all of which are not significantly aided (or are even hindered) with a pen. Web browsing involves typing URLs, scrolling through content, and clicking small targets to link to other pages. Text editing involves typing content and selecting precise locations in a document. Media playback can involve browsing large stores of digital content and metadata, and file manipulation involves extensive browsing and double-clicking. Each of these tasks is awkward with a pen in different ways. Finally, Windows moves significant amounts of functionality into right-click menus, which can be difficult to invoke even with the barrel button located on the stylus.

Sketch-based applications running on Windows cannot break free of their habitat. Windows favors mouse and keyboard interaction, and the pen is foreign. The omnipresence of the taskbar and it's inaccessibly small widgets is a constant reminder of this. So long as the user needs to rely on the mouse and keyboard for the majority of their interaction, users will only perform (and consider) sketch in small, constrained environments.

Our Solution

We propose that a desktop environment optimized for tablet PCs in the easel or slate configuration built around the stylus as it's primary input device is the killer application for sketch-based systems. In order not to alienate consumers unwilling to purchase a computer with an unfamiliar primary interface, such an environment could establish itself as a full screen application bundled with tablet PCs and later function as a core OS desktop environment once established by users. This desktop environment should focus on providing a solid pen-based interface for web-browsing, media playback, text editing, and file manipulation. Advances in hardware should be introduced in order to alleviate usability issues that would hinder the reception and realization of such a desktop environment - a less reflective screen with better viewing angles, a thinner form factor, and a sturdier hinge to reduce wobble from interaction in the easel configuration. With users comfortable with the stylus as a primary input device, this environment could serve as a platform for further research and adoption of sketch-based systems.

Conclusion

Numerous factors raise the bar of entry and obstruct the adoption of sketch-based systems. In this paper we proposed that in order to be considered the ”killer app”, a system will need to address these factors and lower this bar for both users and developers alike. We offer that a standardized development framework would aid programmers in completing more polished applications, and a desktop environment geared towards the stylus as a primary input device will deliver users for these applications. Such a platform could spark the further adoption of sketch recognition interfaces and motivate the creation of sketch's ”killer app”.

Authors' Background

Jeff Browne and André Sayre currently work with Dr. Tim Sherwood at UC Santa Barbara on a general purpose framework for developing sketch applications in Python, furthering the research of Dr. Ryan Dixon [1]. Shane Zamora developed CircuitBoard [2], a sketch-based application for digital logic circuit design and analysis, and is currently working with Dr. Sherwood on a general purpose skeletal figure recognition system. Dr. Sherwood has led courses and seminars on the subject of sketch-based systems at UC Santa Barbara, and presented a paper with Dr. Dixon on “Whiteboards that Compute” at IISWC 2008, an overview of the HCI challenges involved with pen-based interfaces and a look at the current state of research projects that have approached these problems.

References

[1] R. Dixon and T. Sherwood. Whiteboards that compute: A workload analysis. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 69–78, Sept. 2008.

[2] S. W. Zamora and E. A. Eyjólfsdóttir. Circuitboard: Sketch-based circuit design and analysis. In *IUI Workshop on Sketch Recognition, 2009*, Feb. 2009.