Playing with Recognizers: A Call for an Extensible Editor

Richard C. Davis
Singapore Management University
School of Information Systems
80 Stamford Road
Singapore 178902
rcdavis@smu.edu.sg

CHI 2010, April 10–15, 2010, Atlanta, Georgia, USA.
ACM 978-1-60558-930-5/10/04.

Abstract
The sketch recognition interface community has not produced a "killer" application, because access to sketch recognition technology has been too restricted. If recognition technologies were more freely available for experimentation, powerful new applications would evolve. This paper proposes a rough architecture for an extensible graphical editor that facilitates collaboration between recognition technology developers, user interface designers, and early adopters of sketch recognition interfaces. Only by serving all three communities will we reach the critical mass necessary for killer applications to emerge.

Keywords
Sketch recognition interfaces, extensible editors, file formats, research collaboration

ACM Classification Keywords
H.5.2 [Information Interfaces and Presentation]: User Interfaces—Graphical user interfaces, Interaction styles.

General Terms
Design, Human Factors

Introduction

The power of sketching as a medium for creative thought and the potential of sketch recognition technology has continued to inspire new research into sketch recognition interfaces. The pioneers behind these systems freely admit, however, that the technology has not matured to the point of producing a "killer" application. What is preventing sketch recognition interfaces from reaching maturity? This paper assumes that finding a killer app is simply a matter of time. Recognition technology has improved, and modern pen computing devices are more powerful than ever. Even if the time is ripe for breakthrough applications, however, they will only be found through experimentation by recognition technology developers, designers, and users. This paper proposes one model for such collaboration: an extensible graphical editor.

Consider the evolution of the desktop computer as a creative tool. Applications such as e-mail, word processing, spreadsheets, and graphical editors, which are now ubiquitous, evolved through a synthesis of more primitive tools with similar capabilities. This synthesis came about to serve the needs of students and researchers who were early adopters of these technologies. By sharing techniques and using each other's programs to do real work, the community was able to discover applications with commercial potential and produce convincing demonstrations. For the next generation of recognition-based interfaces to evolve, our community needs a similar collaborative environment. Since no such environment currently exists, we must endeavor to create it ourselves.

The open source community gives us a good model for such collaboration. Software components are freely shared among members of the community, allowing everyone to experiment with and build upon each other's work. Over time, robust applications suitable for everyday use begin to evolve. If sketch recognition developers would share working code and support other application builders, we would come a long way toward the collaborative environment we need. Still, recognition techniques are so complex that adapting them to new applications takes more resources than the average student or researcher can spare.

Even if sharing sketch recognition techniques became easy, however, designing user interfaces to take advantage of them would still be extremely difficult. Each technique has its own strengths and weaknesses, and finding a way to take advantage of it could require experimentation by many people. However, there is currently no way to do this without considerable time and programming skill. The platforms and tools for building applications are complex, varied, and constantly changing. Furthermore, few conventions have emerged for even basic interactions like text entry and menus, making it hard to accommodate all users.

An extensible graphical editor would address these problems. It would provide a stable data format and customizable editing environment on multiple platforms that would encourage early adopters to use it for real work. It would provide a stable plug-in architecture that would encourage developers to create interchangeable recognition technology components. Finally, it would support designers by providing facilities for recording macros and for modifying the editor's interface without programming. This would encourage the kind of collaboration we need and finally lead us to the breakthrough applications we have been searching for.

Here, I propose a rough architecture for an extensible editor, which borrows ideas from Emacs and the Piccolo interface framework [1]. It is my hope that this paper will spur discussion in our community and lead to deeper collaboration in the future. I begin with a brief look at related work on extensible frameworks in the sketch recognition and other communities. I then address three major aspects of this editor: the data format, basic interaction methods, and an extensibility mechanism. Following this, I briefly discuss support for multiple platforms and licensing issues. I close with conclusions and an author biography.

Related Work

Unix pipes and scripts are one example of an extensible framework that enabled collaboration. Since text editors were ubiquitous in Unix, text files served as a stable format for users' data. Developers created powerful components like grep, sed, and awk, which users could string together in arbitrary chains to manipulate their data. Some early adopters mastered these tools and lashed them together in scripts. Over time, powerful text processing applications began to emerge. Pipes and scripts are a powerful pattern for collaboration, but they do not facilitate interactive applications like sketch recognition interfaces.

Extensible editors provide a pattern for collaboration that does facilitate interactive applications. These editors give users basic tools for editing documents and can be customized to suit different working styles. Through an extension mechanism, these editors also give users access to powerful, experimental tools written by other developers. Emacs, for example, has extensions for code editing (written in Emacs Lisp) that may have been the breeding ground for modern integrated development environments. In addition, some editors allow people with no programming expertise to extend its capabilities with recorded sequences of editor commands (macros).

Vmacs [8] is an extensible graphical editor that follows the spirit of Emacs. Developers could extend vmacs by defining productions, which are graphical patterns that produce other patterns or trigger interactive experiences. This editor was never distributed widely enough to give users a stable work environment, and its extensibility mechanism would have been awkward for sketch recognition technology developers. It also lacked a macro recording facility for designers.

CogSketch [6] is a freely available sketch editor that is being advertised as a teaching aid (and also as a research data collection tool). Teachers can extend CogSketch by designing worksheets, in which students draw pictures of concepts and receive automated feedback. Parts of this editor can also be hidden for data collection experiments. Developers can automate CogSketch by sending it messages through a socket interface. These extension mechanisms are an excellent start, but designers need ways to create entirely new forms of interaction. Developers, likewise, need ways to add new recognition technologies.

There are also several frameworks that provide tools and architectures for developing sketch recognition interfaces. SATIN [7] and starPad [2] both provide pen-based interface components and architectures for connecting to recognizers. InkKit provides a configurable recognition engine and interface tools that allow certain classes of diagramming applications to be developed easily [10]. These frameworks do not give

users a stable work environment, nor do they provide tools for designers, but they do provide considerable support for developers.

Though several systems come close, no existing editor satisfies all the requirements I mentioned previously:

- A stable data format
- Availability on many platforms
- Customizations supporting multiple interface styles
- A stable extension architecture
- Editor modification without programming
- Macro definition

The remainder of this paper outlines an architecture for such an editor and discusses the challenges involved in creating it.

Data Format

To guarantee a stable working environment for users, the first step is to define a stable format for users' data. A "stable" format is one that will continue to be useful indefinitely, because viewers and editors in this format are freely available and will continue to be available on important platforms. Without this, users will be reluctant to use an application for everyday work. A killer application cannot emerge without this, because users cannot reliably evaluate a tool outside the context of their everyday work.

A tree of graphical objects is a straightforward and common way to represent graphical data, but which data types should be supported? Obviously, it is desirable to have an ink stroke data type that includes any information given by pen input hardware (e.g., position, pressure, time, and tilt). However, ink data is often transformed into other types of data or used to annotate other types of data. The types available will determine the range of applications that the format will support. Following is a list of possible data types.

- *Polylines*: This could be a simple list of straight and curved line segments, but a more powerful structure would keep track of common vertices (e.g., a vertex-edge-face list).
- *Rectangles and circles:* These primitives can be represented with polylines, but having special types for them could make some manipulations easier.
- *Text*: Portable fonts would be desirable to keep text looking the same across platforms, but refusing to use platform-specific fonts may be too restricting.
- *Images:* Also common in many applications.
- *Animation:* Motions applied to the above data types could become a common type of information. The types of key frames supported could be extensible.
- *Sound:* Commonly accompanies animation.
- *Video:* Just as interesting as animation and sound, but harder to support.
- *3D graphics:* This is also hard to support, but sketching of 3D models is a particularly exciting application area. To avoid unnecessary complications, 3D objects should almost certainly be supported as a separate type of tree (if at all).
- *Camera:* These optional objects would define views onto the data.

The list of available data types is only one consideration for a data format. The remainder of this section lists other desirable qualities.

- *Multi-page files:* of the same or differing sizes.
- *Compact files:* for efficient storage and download.

- *Fast scanning and loading*
- *Easy programming:* of parsing and output routines.
- *Data extensions:* Some editor extensions may need to store their own data. This could be done analogously to XML, giving nodes new attributes and tags that are ignored by other modules. Data extensions could be used to store intermediate recognition results, and could even allow the data type to represent more complex scene graphs (e.g., *multi-trees* allow a node to have multiple parents). If a particular extension becomes common, then it may be incorporated into the "official" format.
- *Human readability:* desirable if support for a particular extension becomes unavailable.
- *Existing support:* Choosing a format that already has a base of applications could have significant advantages. SWF is a reasonable candidate, but access to this format is controlled by Adobe. Microsoft claims that the new PowerPoint format (.pptx) is open and it may be extensible. SVG also has moderate acceptance, and InkML has generated some modest activity.

Editor Overview

Once a data format has been established, the next step is to create an editor that is fast, robust, configurable, and works on popular pen computing devices. The most important element of this editor will be a *canvas* for manipulating graphical data. Multiple canvases may be visible at any given time, containing different data files or different views onto the same data.

The extensibility of this editor will be visible to users through commands and interaction modes. A *command* will be an action executed by a user on all or part of a canvas. *Interaction modes* will determine the user's experience when interacting with a canvas. Some modes may add controls or canvases to the periphery of the main canvas or launch dialogs. Users will be able to choose commands and interaction modes by name or by selecting them from a list of available extensions.

As an additional stability guarantee for users, this editor should provide some way to view and edit any data extensions. If users do not have the software extensions necessary for interpreting the data extensions, the editor should help them to find the software with a package manager. If the software is unavailable, then allowing users to view or edit data extensions would give them some hope of recovery.

By providing access to a variety of commands and interaction modes, this editor already provides considerable support for designers. Further support would be provided by allowing sequences of commands to be recorded as macros. Also, special interaction modes could allow designers to configure the editor or modify the appearance and behavior of standard controls. This would allow both designers and users to tailor the behavior of the editor to their tastes or working environments. Over time, even more designer support could be added by developers through the extension mechanism described in the next section.

Extending the Editor

The editor would support developers of sketch recognition interfaces by handling common operations, such as file loading, rendering, simple editing, and undo. To work with the editor, developers would create plug-ins that implement one of the following interfaces:

- *Data visitors:* define the behavior of commands by visiting all or part of the graphical object tree.

These may execute once or repeatedly. Repeating visitors may add editor controls, but these would be removed when the visitor stops running. To keep the editor running smoothly, repeating visitors would be separated into two groups: those that run at interactive speeds (ten milliseconds) and those that take longer. This general interface is suitable for building a range of extensions, such as animated feedback, links to other applications, or segmentation, recognition, and beautification engines.

- *Interactors:* define interaction modes by receiving and responding to input events. These may use other interactors and add controls to the editor. Interactors may do much of their work through data visitors, and these visitors may likewise require certain interactors to be active.
- *Shells:* define the basic editor interface, allowing configuration for specific work environments.
- *Control styles:* define the behavior of a basic set of controls and interactions, allowing the editor to be tailored to a user's taste or the needs of particular hardware. Simple controls would include buttons, checkboxes, radio buttons, menus (anchored or pop-up), textboxes, listboxes, treeviews, sliders, and tooltips. Additional controls would manage input and output behavior common to graphical editors, such as text input, selection, context menu activation, file loading and saving, color choosing, and viewing help.

These four types of extensions would give designers and users access to a wide range of innovations. Furthermore, the separation of concerns in this architecture liberates developers. Recognition technology developers, for example, do not need to worry about users' preferred interaction styles. Interface developers, on the other hand, can create applications without committing themselves to one recognition technology. The set of available controls is small, which would seem to limit innovation. However, new controls and dialog boxes would be built on top of canvases and interactors, allowing unbounded extension (at the loss of some control style independence).

Extensions would have access to a set of core data types, including collections and dates. They would also have interfaces for manipulating the data model, getting input, formatting output, creating command objects (for undo, redo, and macro recording), and logging data. It may also be possible to provide some access to network and threading routines.

Choosing a Platform

Choosing a set of hardware platforms for this extensible editor is both important for attracting a user base and extremely difficult due to the complex caveats associated with each platform. Following is a brief list of candidate platforms:

- *Windows and Tablet PC:* Tablets are the best hardware for many recognition based interfaces.
- *Macintosh:* used by many students, educators, and creative professionals.
- *iPhone and iPad:* this popular platform is being used for ink applications despite the absence of a stylus. The iPhone has a prohibitively small form factor, but the new iPad is much larger.
- *Android and Pocket PC:* Google's and Microsoft's platforms for hand-held computers.

- *Linux:* less frequently used for pen computing, but interesting because it is free.

The choice of platforms also affects the languages that can be used to build extensions and to build the editor itself. Following is a short list of possible languages:

- *C and C++:* difficult to work with, but can be ported to almost any platform.
- *C# and Java:* easier to work with than C or C++ and supported on many platforms.
- *Python:* a popular scripting language that is easily ported to new platforms.
- *Ruby and Lisp:* more flexible than Python, but slower and with less support.
- *JavaScript and ActionScript:* related languages that are popular for web development but have limited support outside web browsers.

Scripting languages are better for simple extensions, as these languages are easy to work with. However, compiled languages are more appropriate for extensions that require high performance (e.g., recognizers). It may be possible to get the best of both worlds by allowing extensions to be written in both types of languages.

Platform and language choices will also be influenced by available software tools and APIs. Following is a brief list of software frameworks that might be used to develop an extensible editor.

- *Microsoft WPF:* This can be used only on Windows operating systems, but it includes many helpful features for managing and rendering ink data. Programs can be built with C# or with a variety of other languages such as Python and Ruby. The starPad system [2] was built with WPF and could provide the foundation for an extensible editor.
- *Microsoft Silverlight:* This is similar to WPF, but programs can also run on Macintosh computers and possibly Linux (eventually). Programs must be delivered initially through a web browser, but can be configured to run without the browser. Unfortunately, Silverlight's support for ink data is significantly less than WPF's.
- *Adobe Flex:* These programs run on all desktop platforms but few mobile platforms. Programs are written with Adobe's ActionScript language.
- *Apple Cocoa:* These programs run on the Macintosh, iPhone, and IPad. Interface components must be written in Objective C, and support for other languages is somewhat limited.
- *GTK:* This is an open-source toolkit for Windows, Macintosh, and Linux computers. It was originally designed for C, but other language options are becoming available.
- *Unity:* This is a commercial game development toolkit for many platforms, including Windows, Macintosh, and the iPhone [11]. Programming can be done in C#, JavaScript, and Python.

A good development strategy for this editor would balance developers' need to keep costs low with users' need for a robust and responsive interface that runs on many platforms. One approach would be to develop the editor first with Microsoft WPF, allowing extensions to be written in Python. Over time, core functionality could be moved into C++, dynamic linking to C++ extensions could be supported, and the editor could be ported to new platforms as needed.

License

If the extensible editor described here were developed, it would presumably be owned by the university or consortium responsible for its development. The license

under which the editor is made available will have far reaching implications for it acceptance and further development. Since the purpose of the editor is to promote development of sketch recognition interfaces, it should be made available for free, but under what license?

A strong open-source license that forces all extensions to be shared with the community (e.g., GPL) seems to be most in line with the goals of this editor. If there is to be any commercial development, however, it will be necessary to use a weaker license that permits proprietary extensions (e.g., LGPL) or a permissive license that permits modification of the editor (e.g., BSD). This paper has assumed, however, that a stronger license is needed to challenge the status quo and spur innovation.

Conclusions

I have outlined an architecture for an extensible editor that would create a stable playground for experimentation with sketch recognition technology. This playground would facilitate collaboration between recognition technology developers, interface designers, and early adopters of new sketch recognition applications. I have argued that such an editor could enable our community to discover a new generation of killer applications.

I offer this proposal to the sketch recognition community to urge deeper collaboration. I do not expect to accomplish anything by defining yet another *de jure* standard, as such standards tend to be ignored. However, if a motivated group of HCI researchers could find a way to collaborate with a motivated group of sketch recognition researchers, their work could bring about a vibrant community that changes the world.

Author Biography

I am an Assistant Professor of Information Systems at Singapore Management University, and I have been doing research and development in HCI and pen computing for over 10 years. My work shows my frustration with sketch recognition systems more than my expertise with them. My first system, NotePals [4], included note taking applications for PalmPilots and CrossPads that recognized simple gestures as well as offline handwriting recognition in a web-based repository. Fascinated with sketch recognition technology, but unable to deploy it effectively, my later work used it less. JotMail [12] linked voice mail with ink notes but provided no sketch recognition. I helped bring handwriting recognition to Mimio [9], but this was an expensive add-on that was never fully integrated with the system. SketchWizard [5] was originally intended to facilitate experimentation with sketch recognizers, but this feature was never completed due to cost. Finally, my K-Sketch animation sketching system [3] avoids the use of sketch recognition altogether. I am still eager to use recognition technology to make new creative tools, but recognition components seem continually out of reach.

References

[1] Bederson BB, Grosjean J, Meyer J. Toolkit design for interactive structured graphics. *IEEE Transactions on Software Engineering,* 30(8):535-46, 2004.

[2] Brown University, *starPad SDK*. http://graphics.cs.brown.edu/research/pcc/starpad.html

[3] Davis RC, Colwell B, Landay JA. K-Sketch: a "kinetic" sketch pad for novice animators. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp 413-22. Florence, Italy, April 2008.

[4] Davis RC, Landay JA, Chen V, Huang J, Lee RB, Li FC, Lin J, Morrey CB, Schleimer B, Price MN, Schilit BN. NotePals: lightweight note sharing by the group, for the group. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp 338-45. Pittsburgh, PA, May 1999.

[5] Davis RC, Saponas TS, Shilman M, Landay JA. SketchWizard: Wizard of Oz prototyping of pen-based user interfaces. *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp 119-28. Newport, RI 2007.

[6] Forbus KD. CogSketch Tutorial. *Proceedings of The Annual Meeting of the Cognitive Science Society (CogSci)* Amsterdam, Netherlands, July 2009.

[7] Hong JI, Landay JA. SATIN: a toolkit for informal ink-based applications. *Proceedings of the ACM Symposium on User Interface Software and Technology*, pp 63-72. San Diego, CA, August 2000.

[8] Lakin F, Wambaugh J, Leifer L, Cannon D, Sivard C. The Electronic Design Notebook: Performing Medium and Processing Medium. *The Visual Computer,* 5:214-26, 1989.

[9] Newell Rubbermaid, *mimio electronic whiteboard tool*. http://www.mimio.com

[10] Plimmer B, Freeman I. A Toolkit Approach to Sketched Diagram Recognition. *Proceedings of the British Computer Society Conference on Human-Computer Interaction*, pp 205-13. Lancaster, UK, September 2007.

[11] Unity Technologies, *Unity game development platform*. http://unity3d.com

[12] Whittaker S, Davis R, Hirschberg J, Muller U. Jotmail: a voicemail interface that enables you to see what was said. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp 89-96. The Hague, The Netherlands, April 2000.