Touchscreen Interfaces for Visual Languages

Michael M. Hackett Philip T. Cox
mhackett@cs.dal.ca pcox@cs.dal.ca

Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia, Canada B3H 1W5

Abstract—Touchscreen tablets are all the rage at the moment; they are inexpensive, highly portable, and frequently house more processing power than the laptops of even a year or two past. They are proving to be wonderful devices for digesting content, but can they be used for "real work"? The lack of a physical keyboard may make them poorly suited for working with text, but the touchscreen removes a layer of indirection introduced by the mouse and allows for a more direct relationship—literally "hands on"—with the objects on the screen. Through the construction of a simple mock visual programming language (VPL) editor, this study compares two different styles of touchscreen interface and demonstrates the natural fit between touch input and visual programming. The addition of "multi-touch" also opens up intriguing possibilities for two-handed, immersive interfaces, with the potential for greater efficiencies than possible with the mouse's single point of interaction.

Index Terms—Computer interfaces, programming environments, user interfaces

1 INTRODUCTION

One advantage often cited by proponents of visual programming environments is the greater sense of *direct manipulation* of program code and data that these environments provide, through the immediate feedback received while working with some graphical representation of the code. However, when the manipulation is performed with a mouse or a trackpad, as is typical in existing systems, an unnatural barrier is placed between the user and the software environment. Additionally, having to frequently switch hand positions and "operating modes" between mouse and keyboard slows users down and breaks their sense of flow. This cuts into their productivity and may drive them back to their textual languages, where they can code and navigate without leaving their trusty keyboards.

In the last couple of years, large touchscreen devices have become inexpensive and readily available, and new user interface designs have been developed that are tailored to finger-based input. Through these touchscreen devices, users can interact more directly with their data, manipulating visual representations with the touch of a finger, and using gestures on the display surface to invoke operations on objects and the environment. However, there has so far been little public research or development of programming tools that actually run on and take advantage of these new platforms.[1] While the lack of a physical keyboard would seem to make these devices poorly suited for traditional text-based programming, the more direct control of on-screen objects that the touchscreen affords might make them ideal for visual programming languages (VPLs).

Additionally, there has recently been a good deal of very interesting research into bimanual (two-handed) interfaces, and a touchscreen capable of recognizing multiple simultaneous touches would seem to be an ideal platform on which to explore some of the ways bimanual input can be applied to visual programming. As programming is a very productivity-oriented task, users will not tolerate novelty for its own sake, especially if it slows them down. Any interface changes must have a positive effect on productivity in order for programmers to accept them over traditional methods.

This paper describes the design and implementation of a simple VPL code editor—codenamed "Flow"—for a touchscreen tablet device (an Apple iPad). The prototype was built in order to experiment with various user-interface options and to get feedback on these from potential users. To minimize the influence of language syntax on overall usability, the VPL is closely modeled on Prograph [1], [2], an existing and well known dataflow programming language. Also, due to time constraints, the program is only a facade of a true visual programming environment—you can edit the code (in a limited way), but not execute it. Videos demonstrating the Flow interface are available at http://web.cs.dal.ca/~pcox/visual/Multitouch/.

The prototype was shown to a small group of users for their feedback. These users were a mix of those with a great deal of previous experience with Prograph (in its desktop GUI form) and those with little or no previous VPL experience. Their feedback will be used to guide further development and experimentation, leading to more formal user studies in the future.

To provide a focus for the study, the following primary research questions were posed, with respect to the Flow

1. Since starting this project, the author has learned of one other touch-tablet VPL project, Catroid, which is an implementation of the Scratch VPL (http://scratch.mit.edu/) for Android tablets.

prototype:

1) Is this type of environment appealing to use?
2) Is the sense of direct manipulation greater than with a mouse as input?
3) Is programming in this manner more efficient than using a mouse and physical keyboard?

Initial feedback from experienced Prograph users has been largely enthusiastic. In particular, all users found the system very intuitive and responsive, and the expert users felt they would be able to work faster in such an environment than with the earlier mouse-and-keyboard-based tools (though, of course, this cannot be tested until the entire programming environment is complete).

A secondary question as to whether this sort of environment could raise the profile and attractiveness of visual programming cannot be answered with this small sample of users, but when the full environment is ready for testing, more extensive user studies will be undertaken.

2 Background and Related Work

Although there is little to draw upon with respect to visual programming on a touchscreen tablet, there is a good deal of touch-input and general user-interface work that informed the design of the Flow software described herein. Of particular interest and inspiration was research into bimanual interfaces, kinesthetic feedback, and touchscreen gesture design.

2.1 Two-Handed Input

Bimanual interfaces require or take advantage of both of the user's hands together, usually in an asymmetric fashion where the non-dominant (NP) hand defines the context for the work that the dominant (P) hand performs. Away from computers, we work like this all the time, when writing on paper, hammering a nail, chopping food, stirring a pot, and so on. In fact, it emerges as a very strong pattern within human tool use. Yet it is very much a rarity in computer interfaces.

This type of arrangement seems particularly well suited to multi-touch input devices, and it was decided early on to experiment with interface designs that made simultaneous use of both hands, and to compare the usability and efficiency of such an interface with one that followed a more traditional design, such as drag-and-drop, or persistent (a.k.a. sticky or latching) modes.

Yee [3] and Wu et al. [4] both suggest using the NP hand to establish and maintain modes while the P hand does work within that context (as suggested in Guiard's Kinematic Chain model of bimanual action [5]). The Flow prototype was developed with two different interfaces and in one of these, modes are selected by pressing and holding buttons on a toolbar on one side of the display (typically the side of the user's NP hand) while manipulating the target objects or tapping a target location with another finger, essentially requiring the use of both hands in most cases (at least for ease and efficiency).

2.2 Kinesthetic Feedback

Sellen et al. [6] give five dimensions along which feedback can be characterized—sensory modality of delivery; reactive vs. proactive; transient vs. sustained; demanding vs. avoidable; and user-maintained vs. system-maintained—and they provide details on two experiments that compared kinesthetic feedback with visual feedback and user-maintained feedback vs. system-maintained. Using a foot pedal as their kinesthetic feedback device, their experiments indicate that *actively maintained* kinesthetic feedback significantly reduces mode errors (that is, instances in which a user performs an action appropriate for a mode that the application is not currently in).

Translating these findings to the touchscreen, one of Flow's two interface designs features buttons that must be "held down" in order to engage the command, while the other depends on a continuous gesture that both establishes the mode and performs some action within its context. In both cases, the command mode is exited automatically when the user's fingers leave the screen. Both are examples of "kinesthetically held modes" [4], where the user must actively maintain some physical action in order to keep the mode active. Using this technique, users are much less likely to forget that a particular mode is engaged, and they can always count on the application having been reset to the default editing mode whenever their fingers have left the screen. "Latching" modes, where one tap sets a mode which stays in effect until cancelled or changed, have been avoided entirely. Not only do these tend to lead to more mode errors [6], [7], but users often struggle to figure out how to cancel the mode selection if they change their minds or realize they have made an error.

2.3 Touchscreen Gesture Design

Nielsen et al. [8] have defined a number of principles and guidelines for designing gestures with usability and ergonomics in mind. They also presented a procedure for first building a "gesture vocabulary" (the set of gestures in an interface) through user studies, then refining and testing the resulting gesture set. They stress the importance of keeping ergonomics in mind and warn against basing gesture design on the recognition capabilities of the hardware, as this may result in gestures that are illogical and "stressing or impossible to perform to some people".

Wobbrock et al. [9] performed a user study (using the methodology of Nielsen et al. [8]), using a tabletop surface, in which participants were asked to come up with touchscreen gestures for various operations, having been shown an animation of the intended effect. Users acted on a static display and so received no active feedback, although their gesture activity was recorded by the system. As above, the intent was to produce a gesture set that was not constrained by system designers' concerns about reliable recognition.

Frisch et al. [10] performed a similar study, specifically looking at diagram and graph editing on multi-touch tabletops. (Their study also looked at the use of a stylus and the combination of touch and stylus input, but the focus of the current study is solely on touch gestures. There are, however, special styli available for use with the iPad's capacitive touchscreen which, although not commonly used, could be employed in a future iteration.)

In both studies, there was a tendency for participants to stick to single-finger gestures that approximated actions from the desktop GUI paradigm (with which they were more familiar), such as tapping to select and then holding down to select from a context menu. When multiple fingers were employed, most users did not consider the number of fingers significant, often choosing based only on the size of the object to be touched. These results suggest that long-time computer users may actually have to "unlearn" some mousing habits in order to make better use of touch-based interfaces, although there is the possibility that, in some domains, users may actually prefer single-finger gestures. Nevertheless, there is plenty of empirical evidence to suggest that advanced users will over time become quite comfortable with multi-finger gestures and will appreciate the operational efficiencies these afford.

Finally, Mauney et al. [11] performed a user study with participants from 8 different countries to look at how cultural difference might affect users' intuition with respect to gestures. They found that there was strong agreement across cultures for gestures with a physical or metaphorical correspondence to the objects being manipulated ("direct manipulation gestures"), but fairly low agreement on gestures that were symbolic in nature. This suggests that symbolic gestures should be avoided (a point also made by Frisch et al. [12]), except perhaps as expert-level shortcuts.

2.4 Prograph

Prograph is an object-oriented visual programming language based largely on a dataflow execution paradigm, extended with some control-flow elements (e.g. conditionals and loops). The language has classes and methods, supports dynamic typing and execution, is inherently concurrent, and its IDE allows code and data to be inspected and changed "on the fly". For the purposes of this discussion, however, one need only know that Prograph methods consist of one more *cases*, and it is within these cases that all of the executable code in a Prograph program resides.

A case consists of *operations* of various types, identified by their unique shapes (see Fig. 1). Operations (commonly called *opers* for short, and for disambiguation from the more general sense of the word) have inputs and outputs, represented by *terminals* and *roots*, respectively; collectively, these are referred to as *nodes*. Terminals appear along the top edge of operation icons, roots along the bottom edge. *Datalinks* connect roots (outputs) to terminals (inputs) to indicate the flow of data from one operation to another. The resulting directed acyclic graph forms the executable code for a case. Fig. 2 presents an example of a Prograph case.

Operation	*Sample Call*	*Action*
Input		Copy value from terminal of calling operation
Output		Copy value to root of calling operation
Simple	Quicksort	Call user method *Quicksort*
Constant	256	Output constant 256 on root
Match	NULL ☒	Next case if value on terminal is not NULL
Persistent	Reviews	Output value of persistent *Reviews* in root
Instance	Index	Output new *Index* instance on root
Get Attribute	key	Output value of attribute *key* of input instance on right root
Set Attribute	review	For left input instance, set value of attribute *review* to right input, and output instance
Local	check	Call local user method *check*

Fig. 1. Prograph operation types. (Adapted from [1], p. 152.)

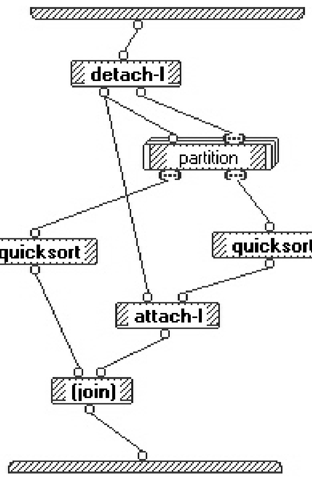

Fig. 2. Prograph code example. Opers are connected by datalinks over which data passes from roots (along the bottom edges of opers) to terminals (along the top edges). Cases are usually laid out such that data flows from top to bottom.

The current Flow prototype implements only a simple case editor, and provides only a subset of the operation types, sufficient for demonstration purposes. In Prograph, opers can also have various annotations, such as those for specifying looping or conditional behaviour, but these have not been included in the prototype.

3 Design Principles

3.1 Discoverability

Norman and Nielsen [13] have criticized today's commercial touch tablet interfaces, citing numerous examples of gestures that are not easily discoverable or guessable, and are learned only by reading about them "out-of-band"—in a forum, a blog, or even (gasp!) a manual. Discoverability was considered a high priority for Flow; this is reflected in the choice to present a toolbar with buttons for all available commands, rather than relying on a series of hidden (and likely arbitrary) gestures that need to be learned. For example, what gesture should be used to create a simple operation? How would it differ from one to create an instance operation? Each of the opers has a different shape, so perhaps the gesture could be based on that. But tracing the shape, even just enough to distinguish it from the other types, would be tedious and error prone. "Shortcut" gestures, intended for experienced users who might find them more efficient, could be added later as an alternative means of input, but should not be the sole means of executing a command.

On the other hand, moving operations by dragging them with a finger is intuitive and easily discoverable. It is not necessary to create a special button to enable this, and in any case, one would still have to know to drag the operation symbol while in this mode.

3.2 Minimizing Modes

Modes result in mode errors (see above); therefore one hopes that reducing (or eliminating) modes will result in fewer mode errors. Despite some claims of having "modeless" interfaces, this is rarely actually achievable or even desirable, since it would require a good deal of context to be repeatedly and tediously reestablished. Sellen et al. write, "what is actually meant by a modeless interface often refers to design in which contextual information is provided to minimize mode errors and in which modes can be easily entered and exited." [6, p. 143]

As mentioned earlier, it has been shown that user-maintained modes, using kinesthetic feedback, typically result in much lower rates of mode errors, and faster resumption of activity after an interruption. What Wu et al. [4] refer to as "kinesthetically held modes" (actively user-maintained, with kinesthetic feedback), Raskin calls "quasimodes" [7, p. 55]—a kind of mode still, but less problematic, perhaps. To illustrate the difference, compare how often you accidentally type upper case letters as a result of forgetting that you are holding down the Shift key on your keyboard (probably not very often) with how often this happens with your cell phone (which likely has a latching Shift key soft button) or as a result of accidentally hitting the Caps Lock key. Caps Lock creates a mode; the Shift key creates a quasimode.

The closest equivalent for a strictly touchscreen device is to require at least one touch be maintained for the entire duration of the quasimode. This provides the requisite kinesthetic feedback that a mode is active. In the prototype, two different means of achieving this have been attempted: In one, the user holds one finger (or thumb) down on a toolbar button that engages the mode, while one or more other fingers perform some manipulation in the main editor pane within the context of the quasimode. In the second, a single continuous gesture engages the mode and performs the manipulation, the mode ending when the touch(es) are lifted from the screen, signalling the end of the gesture. Dragging and dropping an object from a "parts bin" into the editor pane to create a new instance of that object type is an example of the latter, as is the act of moving an existing object within the editor.

3.3 Responsiveness

Although this is only an early prototype, it was believed that the experience had to be smooth and responsive in order for any user feedback to be meaningful. To give the user a real sense of direct manipulation, objects would have to react without any hesitation or sluggishness. (That this intuition was correct was borne out in the comments from users who were all very impressed and pleased with the responsiveness of the application.)

4 Prototype Application

4.1 Design

For the initial prototype of Flow, two separate interfaces were created, each employing a different user interaction style, for comparison. The visual layout for the two versions is essentially identical, featuring a toolbar along one side of the display and a large area for displaying the code being edited. (See Fig. 3.) The toolbar contains buttons for creating each of the language elements (for now, this includes operations, nodes, and datalinks) as well as various commands (only Delete at the moment). The toolbar can be placed along either the left or right edge of the screen (for right- or left-handed users, respectively), as selected through a user preference.

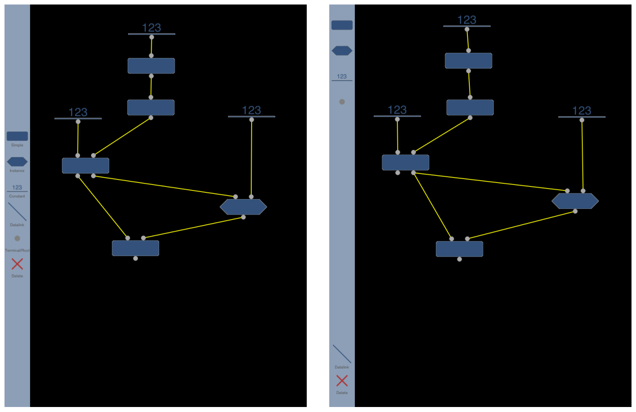

Fig. 3. Prototype interface: Quasimodal (left); Drag-and-drop (right).

In one version, the buttons control the current operating mode of the editor—adding an operation, adding a node, adding a datalink, or deleting any of the above—and a button must be "held down" (the touch must be maintained on the button) to engage the mode, providing continuous, user-maintained kinetic feedback. (Fig. 4.) Although this can be done (somewhat awkwardly) with one hand, the intent is for this interface to be used with two hands, the NP hand selecting the mode and the P hand tapping or dragging in the code pane. This will be referred to as the "quasimodal" version, using Raskin's term.

In the second version, all of the buttons for creating new objects (except for datalink creation, which was not completed in time for this report) are operated by dragging a finger from the button onto the code pane, which creates an object for the user to drag into place. (Fig. 5.) The delete command is still operated using the two-handed method, but that could be replaced or supplemented with a gesture (such as making a stroke through an object, or dragging it to a trash area along another edge of the screen). Unfortunately, this was not included in the version the users were given, so any conclusions must be tempered by the knowledge that the second interface was not as consistent as it should have been in order to make a fair comparison. (The visual separation of the toolbar buttons in this interface was done solely to provide a clue that the two sets of buttons were operated differently, but the placement of the buttons may not have been ideal.) This interface, despite the mixing of styles, will be referred to as the "drag-and-drop" (or DnD) version.

As one user put it, a fundamental difference between the two interfaces is the use of one hand versus two, although that is not the only point of differentiation, nor are these the only possible approaches in either of those two categories.

The drag-and-drop style should be familiar to most users from desktop GUIs, but the other mode may require a bit of explanation. It will initially be unfamiliar to most, but it was hoped that, once the operation is explained, it would be easy to understand and offer a viable alternative to the more customary designs.

In both versions, the editor operates in a default mode when no buttons are pressed. In this mode, objects can be moved simply by dragging them (Fig. 6)—in fact, several can be moved independently all at once, each with a different finger. In a full-fledged editor, there would probably be support for other actions that can be easily and naturally controlled through unique gestures. In the quasimodal version, objects cannot be moved while any of the buttons are engaged, as that action might be confused with a gesture related to the active quasimode. Take, for example, creating a datalink, which involves dragging one's finger across the screen from one node to another (while the Datalink quasimode is engaged). Since the point of contact would be on or near the edge of an oper icon, it would be hard to reliably distinguish this action from an attempt to move the oper, if it were not for the quasimode to provide context. Because a finger touch is not nearly as precise as a mouse or stylus, and some of the targets are necessarily small, the software must be forgiving and allow a good deal of leeway in hitting

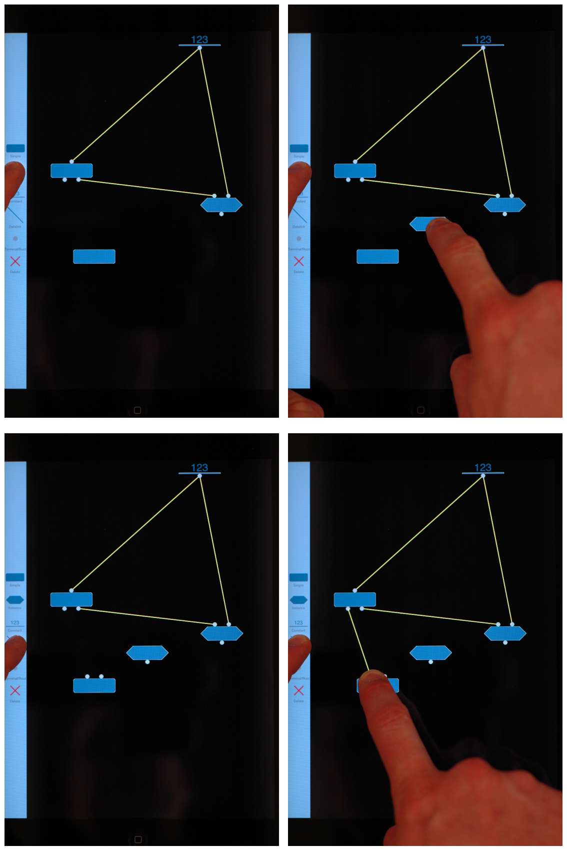
Fig. 4. Quasimodal operation: Creating a new instance operation involves actively maintaining a touch on the Instance operation button (top-left photo) and tapping with another finger to place the oper in the code pane (top-right). To create a new datalink between nodes, the Datalink button is held (bottom-left) while another finger draws a line from the start node to the end node (bottom-right).

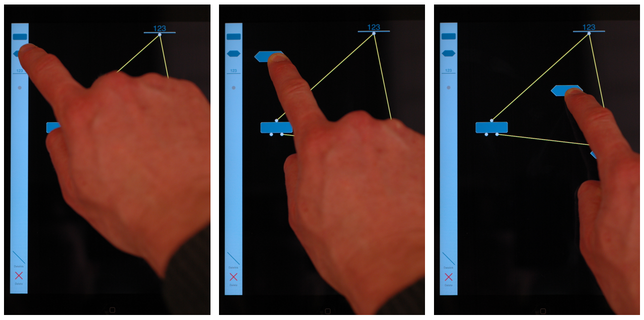
Fig. 5. Drag-and-drop operation: New operations are dragged from the toolbar onto the editor pane and placed wherever desired. (Sequence shown left-to-right.)

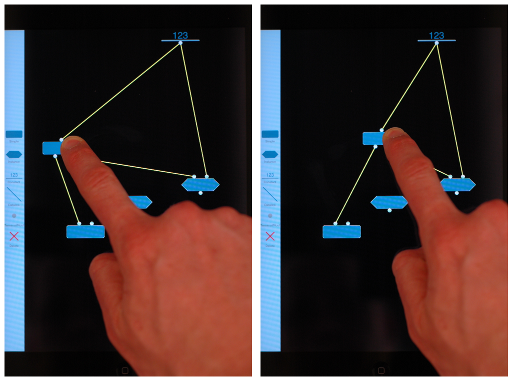
Fig. 6. Moving opers: In the default editing mode (and at any time in the drag-and-drop interface), opers can be moved simply by dragging them.

targets. If active targets overlap or are too close together, the recognizer might misjudge the intention, likely resulting in a frustrating user experience. By limiting the set of potential targets within a mode or quasimode, recognition accuracy can be much higher, which makes the interface seem more intelligent.

In drag-and-drop mode, creating new code objects by dragging them from the toolbar invokes a kind of quasimode as well, in as much as the creation mode is active only as long as the user maintains screen contact with the finger that started the drag gesture. However, unlike in the full quasimode interface, where engaging a toolbar button locks the entire interface into the corresponding quasimode, here the quasimode applies only to that one finger. Other fingers can continue to move existing objects in the editor, or begin additional creation actions. This feature is not inherent in the design; it was more an accident of some implementation choices. However, it seems more in the spirit of an immersive multi-touch interface to allow this, so it will probably remain. If some users find this confusing, it would be easy enough to add a preference to limit the interface to one mode or quasimode at a time.

Persistent modes, apart from the default mode that automatically becomes active when all fingers have been removed from the screen, have so far been completely avoided. There are still many features to be added to make the prototype into a usable code editor, but it is hoped that this design path can be maintained as more functionality is added.

4.2 Development Platform

The target hardware for the prototype is a first-generation Apple iPad. The iPad has a 9.75″ (247 mm diagonal) screen with a capacitive touchscreen overlay, which can simultaneously detect independent touches from all ten fingers.

The original plan was to develop the program for the WebKit [14] HTML 5 platform (with Apple's extensions), to allow it to be easily ported to other platforms that have the same environment available (including Android-based tablets and most desktop OSes). However, the available API frameworks were, at the time (January 2011), found to be too immature to use without significant further development, and the example programs built with these frameworks felt somewhat sluggish in operation. As a result, the initial prototype was developed as a "native app", using Objective-C and Apple's iOS SDK [15]. Apple has since reportedly doubled the speed of their WebKit JavaScript engine, and the frameworks will no doubt get better in time, so it may be possible to revisit the WebKit platform at a later date.

5 User Evaluation

While there were no plans for formal user testing within the scope of this phase of the project, a small sampling of users known to the author were given the software to try and were interviewed while using the software and afterwards. Some were expert Prograph users and so fully understood the model of the visual editor and the symbols used, while some others were non-programmers or novice Prograph users who, nonetheless, were able to give feedback on the responsiveness and naturalness of the interface options.

5.1 Interview (Semi-)Structure

Although these were to be informal interviews, a set of questions was prepared as a guide, to help ensure that all the key questions were asked of each interviewee. There was also a set procedure regarding how the software was presented to the user and what instruction was to be provided to each:

1) Configure the software into *quasimodal* mode, with the toolbar on the appropriate side for the user, then give the tablet to the user, initially with no instruction.
2) Ask the user to think out loud; tell the user that he or she is free to ask questions. Allow the user a few minutes to try the program; if he or she becomes stuck, offer advice or demonstrate the use of the buttons or other feature.
3) Give a full tutorial on two-handed use, if necessary, demonstrating and explaining that mode or context selection is done with the NP hand, manipulation with the P hand. Allow the user to try again.
4) Take the tablet, switch the interface mode to *drag-and-drop* mode, and pass it back to the user.
5) As before, allow the user to experiment without providing any instruction, but answer any questions.

After allowing the user sufficient time to play with the software, the following questions were asked:

1a) Did the interface feel natural to you? Was it easy to guess how things worked? Were there elements that required explanation?

- b) How do you think it could be made more natural or intuitive?
- 2a) Did you find that the touchscreen felt more or less direct than using a mouse, or about the same?
- b) How was the responsiveness to touches?
- 3a) How did you find the operation of the held-down buttons for setting the current mode or context? Once it was explained (if necessary), did it begin to feel natural?
- b) Was it at all awkward or fatiguing?
- c) Did you have any trouble finding or holding the buttons with your thumb or finger?
- 4a) Using both hands in this way, with the off-hand leading the dominant one, did it feel more efficient? Strange? Comfortable?
- b) Did it bring to mind any other two-handed tasks?
- 5) Overall, did you find this environment appealing to use?
- 6) Do you have any other comments or suggestions?

Due to the informality and semi-structured nature of the interviews, the questions were often answered out of order, sometimes as a result of the direction of the conversation during the test or the interview, and sometimes incidentally through comments made while using the program. Also, the interviewer observed the user while he or she used the program and made note of any apparent difficulties or lack thereof.

5.2 Summary of Results

Overall, there was a strong consensus that the design was intuitive and very responsive. All but one user (and all of the expert users) felt that the sense of direct manipulation was greater than with a mouse, and that the responsiveness of the interface to touches was an important part of that.

The expert users all initially assumed that drag-and-drop would be the way to operate the toolbar buttons, while some novice users first tried to tap on the buttons to latch them on. When asked, they generally attributed this to their previous computer experience and expected the tablet to be the same. However, after an explanation was given (often as little as suggesting that he or she try using two hands), all but one quickly mastered the quasimodal operation, and the majority said that they preferred this version to the DnD version (even when stating that he or she found it less “natural”). Most found it faster to create new objects with the two-handed interface than by dragging items from the toolbar. And the expert users found adding nodes by dragging them from the toolbar to be somewhat awkward (it “felt odd”, one said, perhaps because there was no analogue to it in the desktop version). No one had a problem adding nodes by tapping using the quasimode version.

(It should be noted that node dragging was an immature feature, added only the day before the user sessions, and it lacked even the limited visual feedback and snap-to assistance that datalink creation has. It is not really surprising that it was the least liked feature.)

Two users expressed a preference for the drag-and-drop interface, primarily because they preferred to operate the program with one hand. This seemed to be partly because the form factor of the tablet invites one to sit back and hold it in one hand, rather than placing it on a table so that both hands are free. One user suggested moving the quasimodal buttons up towards the top edge of the screen so that the unit could be gripped up higher (better balancing the tablet in the hand) while still leaving the thumb free to press buttons. It was also suggested that a landscape orientation for the editor (it is currently portrait-orientation only) would accommodate holding the tablet with both hands while operating the interface with the thumbs.

The size of the interface elements was generally considered very good. One user suggested adding borders around the toolbar buttons and adjusting the spacing between icon and text label to make the pairings more clear. However, no one asked for larger buttons or more space between, although some button misses were observed.

The snap-to-node feature of datalinks worked pretty well, but some users expected or expressed a preference for being able to see the point of connection above the tip of the dragging finger. This was actually tried for node placement, but the offset used was based on the author’s finger size and angle, and it turns out that there is a wide variance, even within this small sample, in how people hold their finger on the screen. Some keep the angle low, others practically vertical. Some expected the connection point to be at the point of finger contact, some wanted it above the end of the fingertip, some below it. This was generally not a problem for opers, whose shapes are larger than a finger tip, but it’s clear that more and/or different types of feedback are required for smaller shapes.

6 Analysis of Feedback

Going back to the original set of study questions, the feedback obtained so far seems to point to affirmative answers to at least the first two of these: Users *do* find the environment appealing to use, and the sense of direct manipulation *is* enhanced by the touchscreen interface. It is a bit early to begin judging coding efficiency, but the experienced Prograph users were very excited and are looking forward to putting that question to the test with a future, more fully realized version.

Given the slight novelty of the quasimodal interface, as compared with what is common in most desktop and mobile GUIs, it appears that users do need at least a small amount of instruction to figure out how to operate the program. Because mobile applications rarely come with any instruction manuals, users will expect to be able to use the program without reading any external documentation. Therefore, some sort of in-program assistance should be added to quickly show the user how to operate the controls. Testing so far does suggest, however, that once users learn the "trick", many do prefer the quasimodal interface, so it does seem worth asking them to put in a small amount of effort to learn the unfamiliar system.

Mobile games may offer some useful examples to follow, as it is common for games to have specialized rules and control systems that must be learned before or during play. Perhaps even something as simple as displaying a comics-style speech bubble, attached to the side bar, the first few times the editor is opened, and having it disappear as soon as the user successfully uses the buttons once or twice. Later, if the program detects actions that suggest the user is trying to operate the controls in a different manner (such as trying to drag from the buttons or tapping the button, instead of pressing and holding, to engage the mode), the program might pop the bubble back up as a reminder. Or, in the case that this is a new user who happens to be using someone else's tablet and who isn't familiar with the interface, providing the initial instruction again. (Many desktop applications have a "don't show this message again" box for instructional or warning messages, but these impose the preferences of the primary user of the system on any others who may also sometimes share the computer and who may not be at the same level of experience with the software.)

Another alternative would be to support multiple interaction styles simultaneously, as much as possible. For example, the gesture recognition system would likely have little difficulty distinguishing between a stationary "press-and-hold" touch on a toolbar button and a dragging action from the same button. Provided there is very little chance of the system behaving in an unexpected manner, this is a reasonable option. There is a danger, however, that most users will first try what they are familiar with—that is, drag-and-drop—and as a result may never discover that there is another interaction style available, one which they might actually prefer.

Node dragging as currently implemented needs work. It lacks sufficient visual feedback such as highlighting the destination oper when the node is placed on or near it, snapping the node to the edge of the oper, and perhaps animating the other nodes to their new positions. Feedback when connecting datalinks could also be improved; the action of snapping to a node when in range is effective, but it might also be helpful to draw a circle around the target node, wide enough to be seen around a fingertip, to provide confirmation that the obscured connection is being made. Alternatively, the end of the datalink could float at some distance from the touch, allowing the connection to be made with the fingertip covering the node. Some users find this awkward, however, and others would have to adjust the way they hold their fingers in order for this to be of any help.

The sessions also produced several excellent suggestions for additional editing features, including being able to grab datalinks along their lengths to reposition them, thereby reducing the cost of a placement error, which will help minimize user frustration.

7 Conclusions and Future Work

The goal for this project was to try some novel user interface designs for an existing visual programming language and get feedback on what seems to work and what doesn't. Although there is still much work to be done before the current prototype is ready for real use, the initial feedback, particularly from the experienced VPL users, is very encouraging. The sense of direct manipulation, a key feature of visual programming, is more effectively realized with a touchscreen than with a mouse. And the ability to use multiple fingers and both hands instead of just a single cursor opens up many possibilities for more immersive and more efficient programming interfaces. It may be, however, that not everyone is ready for such a paradigm shift, and as development progresses, it may be necessary to keep some more traditional alternative options in mind so as not to alienate these users. It may be wise to make use of progressive enhancement techniques to gradually introduce novel interface features, wherever it is possible for the two styles to co-exist.

Proper user testing needs to be done before declaring the use of kinesthetically-held modes (quasimodes) a success, but in our limited sample all found it easy to learn how to use the buttons (many require no further prompting than to "use two hands"), and despite the unfamiliarity, many of the testers quickly grew to like the interface and expressed a preference (if tentative, given the limited amount of exposure) for the quasimodal interface over the drag-and-drop style with which they were more familiar. The ones that did not prefer this style expressed a preference for being able to operate the tablet with one hand, in a more casual manner. At this early stage in its adoption, it may be that the tablet is not seen as a tool for serious work, but more something to sit back and browse with. It will be interesting to see if this attitude changes over time, as more productivity applications appear for tablet computers. It would also be interesting to bring the programming environment to a large table-top touch surface system, in order to experiment with how the additional screen real estate could be used, and to see if the change of context changes users view towards the "seriousness" of the system.

Future work will largely be focused on experimenting with designs for navigating higher-level constructs, such as cases, methods, classes, and libraries. There are a number of general approaches to consider, such as zooming, overview+detail, focus+context (all described in [16]), and/or a more traditional hierarchical system. And within those approaches, there are several ways to map them to a small screen and to take advantage of touch input. Again, particular attention will be paid to the option of bimanual input, with the hope that this can spur further investigation into the effectiveness of this mode of interaction.

Additionally, there was a third system originally intended to be included in this first prototype, one based on the Toolglass widgets system [17]. However, development of the first two interfaces took longer than anticipated and there was not enough time to complete the third for inclusion in this report. Toolglass was originally designed for a mouse-and-pen two-handed system, but it sounds like a perfect fit for a two-handed touchscreen system. The author is not aware of any attempts to implement Toolglass in this context (either in a VPL or in a touchscreen environment); it would be interesting to see what a touch-tablet implementation might look like and how users would react to it, as it is quite different from what most are familiar with.

Acknowledgment

The authors would like to thank the members of the Dalhousie University Visual Languages and Design Group for their valuable input during the development of the Flow prototype, and their former Pictorius colleagues who took the time to provide expert-user feedback on the prototype.

References

[1] P. T. Cox, F. R. Giles, and T. Pietrzykowski, "Prograph: a step towards liberating programming from textual conditioning," in *Proc. 1989 IEEE Workshop on Visual Languages*, 1989, pp. 150–156.

[2] Pictorius, *Prograph User Guide*, version 1.1 ed. Halifax, NS, Canada: Pictorius, Inc., 1997.

[3] K. Yee, "Two-handed interaction on a tablet display," in *CHI '04 extended abstracts on Human factors in computing systems*. New York, NY, USA: ACM, 2004, pp. 1493–1496.

[4] M. Wu, C. Shen, K. Ryall, C. Forlines, and R. Balakrishnan, "Gesture registration, relaxation, and reuse for multi-point direct-touch surfaces," in *Proc. 1st IEEE Int. Workshop on Horizontal Interactive Human-Computer Systems (TableTop 2006)*, 2006, pp. 185–192.

[5] Y. Guiard, "Asymmetric division of labor in human skilled bimanual action: The kinematic chain as a model," *Journal of Motor Behavior*, vol. 19, pp. 486–517, 1987.

[6] A. J. Sellen, G. P. Kurtenbach, and W. A. S. Buxton, "The prevention of mode errors through sensory feedback," *Human-Computer Interaction*, vol. 7, pp. 141–164, Jun. 1992.

[7] J. Raskin, *The Humane Interface: New directions for designing interactive systems*. Reading, MA, USA: Addison-Wesley, 2000.

[8] M. Nielsen, M. Störring, T. B. Moeslund, and E. Granum, "A procedure for developing intuitive and ergonomic gesture interfaces for HCI," in *Gesture-Based Communication in Human-Computer Interaction*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 2915, pp. 105–106.

[9] J. O. Wobbrock, M. R. Morris, and A. D. Wilson, "User-defined gestures for surface computing," in *Proc. 27th Int. Conf. Human Factors in Computing Systems (CHI '09)*. New York, NY, USA: ACM, 2009, pp. 1083–1092.

[10] M. Frisch, J. Heydekorn, and R. Dachselt, "Investigating multi-touch and pen gestures for diagram editing on interactive surfaces," in *Proc. ACM Int. Conf. Interactive Tabletops and Surfaces (ITS '09)*. New York, NY, USA: ACM, 2009, pp. 149–156.

[11] D. Mauney, J. Howarth, A. Wirtanen, and M. Capra, "Cultural similarities and differences in user-defined gestures for touch-screen user interfaces," in *Proc. 28th Int. Conf. Human Factors in Computing Systems (extended abstracts) (CHI EA '10)*. New York, NY, USA: ACM, 2010, pp. 4015–4020.

[12] M. Frisch, J. Heydekorn, and R. Dachselt, "Diagram editing on interactive displays using multi-touch and pen gestures," in *Diagrammatic Representation and Inference*, ser. Lecture Notes in Computer Science, A. Goel, M. Jamnik, and N. Narayanan, Eds. Springer Berlin / Heidelberg, 2010, vol. 6170, pp. 182–196.

[13] D. A. Norman and J. Nielsen, "Gestural interfaces: a step backward in usability," *interactions*, vol. 17, pp. 46–49, Sep. 2010.

[14] WebKit Open Source Project, "WebKit," http://webkit.org/, 2011.

[15] Apple Computer, "iOS Reference Library," http://developer.apple.com/library/ios/navigation/, 2011.

[16] A. Cockburn, A. Karlson, and B. B. Bederson, "A review of overview+detail, zooming, and focus+context interfaces," *ACM Computing Surveys (CSUR)*, vol. 41, p. 2:1–2:31, Jan. 2009.

[17] E. A. Bier, M. C. Stone, K. Pier, W. Buxton, and T. D. DeRose, "Toolglass and magic lenses: the see-through interface," in *Proc 20th Ann. Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH '93)*. New York, NY, USA: ACM, 1993, p. 73–80.

[18] D. Saffer, *Designing Gestural Interfaces: Touchscreens and Interactive Devices*. O'Reilly Media, Inc., 2008.