Mobile phones as programming platforms

[Position Paper]

Georg Essl
Electrical Engineering & Computer Science and Music
University of Michigan
2260 Hayward St., Ann Arbor MI 48109
gessl@eecs.umich.edu

ABSTRACT

Programming is virtually equated with text entry. Programming languages are defined by their syntax and semantics that ultimately lead to functionality when translated into system level operations. This text-entry focus is a tacit assumption. It rarely is argued that one ought to expand or alter this notion of programming. We argue that the emergence of mobile smart phones as full-fledged computational devices may be challenging this assumption. The reason why text entry was viable in the past simply has to do with the fact that the keyboard has been the dominant input modality to computers. On many mobile smart phones, keyboard input is either reduced or unpleasant. Hence we argue it is worthwhile to reconsider the notion of programming when considering mobile devices as primary programming platforms and that we need to envision programming paradigms that are better aligned with the input capabilities of the device, hence suggest a replacement of text-based programming with other approaches. This forces a renewed exploration of what we mean by syntax, semantics and their relation to input and representation.

Keywords

mobile phone programming, visual programming, data-stream programming, user interface design

1. INTRODUCTION

What happens when we consider pocket-sized mobile devices our primary programming platforms? How will the available input capability shape how we think about what programming is? Does it make sense to try to translate existing programming paradigms literally over to the mobile platform or should we be ready to embrace a new way of looking at programming that embraces the input modality as part of what programming ought to be? These questions are at the core of this paper.

It is probably fair to say that whenever the word programming is uttered or written down, the immediate and tacit assumption is that this refers to writing lines of text, perhaps in a context-free grammar with additional semantic information sufficient to "compile" or otherwise transform it into system-level instructions that then run on the computational hardware itself. That is not to say that other paradigms have not been considered. Visual programming languages have a storied history as well [8]. Yet text-based programming is still the dominant type of programming language representation today.

Here we would like to argue that we might have a case for a need to consider alternative paradigms due to the establishment of mobile devices as primary computational platforms. Certainly the reason for the dominance of text-based programming languages is simply that text based input (and associated human readability) is one of the earliest forms of interacting with computers [11]. Certainly the keyboard to this day is one of the most important interfaces for a programmer to enter their creations. Visual programming emerged both out of a relation of alternative, data-flow driven hardware architecture ideas [8] and the recognition that alternative representations can be beneficial to the programmer as well. The mouse and the advent of color displays certainly helped the viability of this type of programming paradigm. Finally there is an emerging area of so-called programming tools. These are systems that help complete a program but the process of using them is not typically associated directly with programming itself. An example of such a tool would be for example an interface builder or a GUI engine (e.g. [10]). In our view this conceptual separation of tool use and programming is an indication of the dominance of text-entry being viewed as programming while other types of input is not. Hence we might suggest that the use of tools that contribute to completing a program should in a broader sense also be called programming. In fact interface building could be seen as a form of non-data-flow visual programming.

Overall the constraint of the interface has not played an important role in the discussion how to best support programming. The interface was established (keyboard and mouse) and if discussion related to the input it had to do with improving the bandwidth or support assuming these types of input (e.g. auto-completion [9]). We would argue that the transition to the mobile device should change how we look at programming. Devices no longer have a full alpha-numeric keyboard. The keyboard input ranges from reduced key-

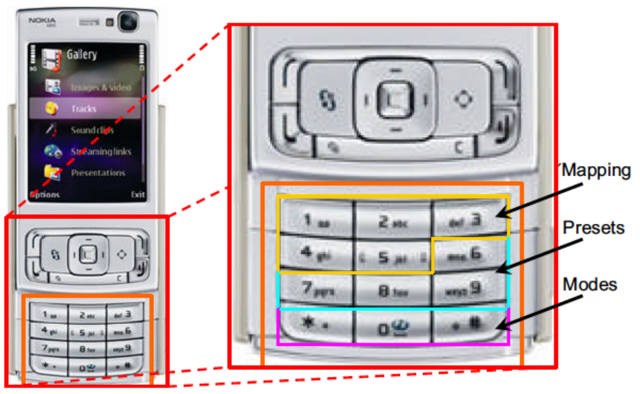

Figure 1: Organisation of program input keys in SpeedDial.

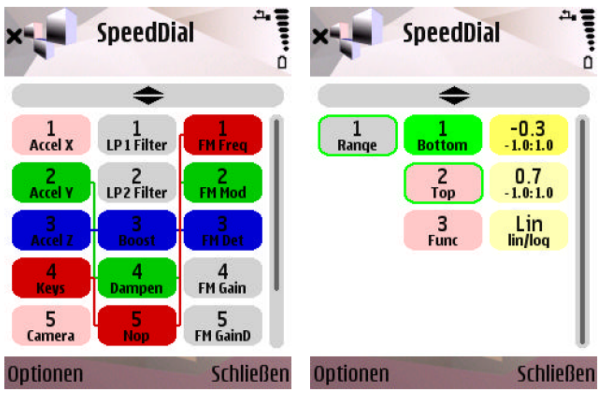

Figure 2: Data-flow and parameter editing interfaces of SpeedDial. Visual elements are numbers to indicate how key strokes address them.

blocks to 12-key numeric blocks to simulated keyboards on single- or multi-touch devices. The form factor of mobile devices encroaches on the space that would be needed to design a comfortable standard keyboard. In fact ways to support text entry is an important line of work in mobile HCI [2]. Furthermore the dominance of the mouse as a 2-dimensional planar input device is broken in the mobile realm. A range of technologies, from joysticks, cursor keys, trackballs, stylus or finger-based single or multi-touch input are being explored as alternatives.

All these new input technologies bring new difficulties. Joystick-type inputs are well known to not compete well with mouse-like input since the seminal Fitts' law study by Card, English and Burr [3]. And stylus or touch input pose other problems, the most widely recognized being the problem of occlusion. Because the input is in the same area as the interface, fingers and palm hide good portions of the screen. Furthermore the finger may occlude the exact part of the screen that is to be targeted (a problem that Baudisch coined the "fat finger problem" [1]).

However it seems that the two prevailing modes of input for a programmer are changing and hence it is worthwhile to reconsider programming paradigms with the new input modalities in mind. We believe that the field is wide open and much remains to be discovered. In the remaining of the paper we discuss two early examples of data-flow type visual interfaces designed for specific instances of input.

The first example will emphasize data-flow, while the second will also address interface building as programming. Both these programs come from the area of building interactions on mobile devices just using the mobile device itself, without the need of external hardware. Hence they are example of the philosophy that mobile devices themselves can be used as primary computational and programming platforms.

2. PROGRAMMING ON A 12-KEY NUMBER PAD

How might one program a mobile phone with a standard 12-key number block without connecting or otherwise utilizing an external programming platform? The most direct line of attack would undoubtedly be to simply type out programs in a standard text based language using multi-tapping [6]. Clearly that is a significant reduction of bandwidth from a standard keyboard, but it is in principle doable. This would also suggest that one displays standard text based programs on the display of the device. This too is doable but the amount of the program that is visible would be limited by the screen space and programming language syntax is often not optimized to make the best use of limited screen-space.

An alternative approach would be to ask: How can functionality be quickly established given the input capabilities of the device. It is obviously not very desirable to rely on multi-tapping routinely, hence it seems sensible to seek for an input space that is roughly of the size of the key block. Then we have to define how the input space relates to visible content (the program) on the mobile screen. Text input has a direct correspondence between typed text and visual outcome and ideally we want to retain this directness.

There are numerous ways to retain this directness. One particular choice is offered in SpeedDial [4], which is a somewhat unfortunately named music data flow programming environment for Symbian-based mobile smart phones such as the N95. In SpeedDial functional entities are numbered by a grid position on the screen. The functional block hence can be directly addressed by simply hitting the labeled key on the key pad and as the visual grid never exceeds the number of options on the keypad one can select, connect and disconnect functional units without the need of multi-tapping or other mechanisms to expand the input space. In order to give control to a possible large space of functional entities to be used in a data flow, SpeedDial offers a selection mode in which one can select which functional blocks to use in a current flow from a possibly much larger set. Finally in order to manage a trade-off between legibility and space management two keys are used for scrolling and a further key is used for selection.

Conceptually what SpeedDial's approach does, compared to text-based programming, is the removal of selection from a finite set of programmatic building blocks via typing. Instead these programmatic blocks are available as visible units and will be opportunistically be addressable by keys when

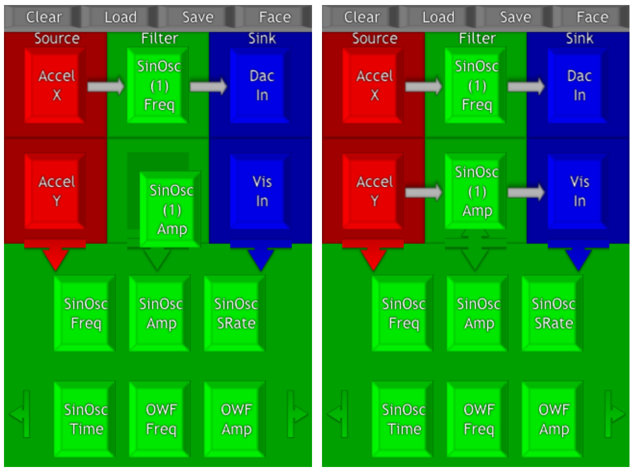

Figure 3: Multi-touch "programming" in urMus. Placing functional unit in grid positions connects them with their horizontal neighbors.

they are visible. Hence while the direct correspondence of input to specific functionality is broken, the correspondence of visible block to input is retained and the overall bandwidth needed to enter and establish functionality is reduced to selecting and connecting blocks.

Relations between functional blocks are in part determined by the position of elements on the screen. Blocks at the left of the screen have different semantics than blocks in the center and left respectively. Hence space is used to contribute semantic meaning in a program and choice which helps manage the input bandwidth, because some semantic meaning can be implied.

3. PROGRAMMING ON A MULTI-TOUCH MOBILE DEVICE

Multi-touch mobile devices such as the iPhone have a different input paradigm. They do offer a reduced keyboard with mode selection to expand the options. However typing with haptic feedback is known to be more difficult than typing on devices with clear haptic separation of keys, even if they are smaller. In general text input on the iPhone is not particularly easy and is helped by an auto-completion algorithm to counter unintentional slips in typing. While autocompletion works particularly well on a fixed vocabulary such as the syntax of a programming language it still is much more band-limited than text based program entry on a mobile phone.

Hence again one can prototype alternative solutions that more directly use the input characteristics of the device. urMus is a meta-environment designed to help explore these questions and we are yet in early stages of understanding what would constitute desirable input paradigms for programming and data-flow mapping [5]. But early prototypes already hint at the possibilities that can be considered when attempting to turn mobile multi-touch phones into primary programming platforms.

In order to honor the finger based input, urMus uses finger-tip sized icons are functional blocks in a data flow. It is noteworthy that this replaces the correspondence between key input and screen position. Hence in urMus one automatically always has correspondence between visible elements on the screen and the input. These blocks are again organized by their broad semantic meaning such as if they correspond to sensory input, functional processing or actuator output. But rather than entering connections through key strokes, connections are now established if two blocks become horizontal neighbors on a grid. Hence again we use programming by position, but in this case in a stronger sense. The position of two blocks on screen alone already establishes if they are connected and hence require no further input. This again helps reduce input bandwidth and trades it against semantic information on the screen.

urMus offers numerous mechanisms to manage screen space. One can for example expand the programming grid horizontally and vertically and then use touch swiping gestures to scroll on the grid. Furthermore the selection area for functional blocks can be expanded hence allowing a trade-off of space for selection of functional units against visible space of the program.

There are programming language concepts that support the input of data-flow in urMus. One key concept is a type system which carries semantic information of the data flow and normalizes interconnectivity. This achieves that the programmer no longer has to specify the semantics information at changes in connectivity of the data-flow and hence avoid any need for numeric input or semantically induced type casting.

Finally urMus offers an on-device layouting system that is implemented through an API using the embedded Lua language [7]. What is the crucial insight about this system is, however, that the Lua layer is in principle not necessary to allow "programming" of user interface layouts. These can be designed purely graphically. Regions in the layout can be created using multi-touch interactions and can be associated with properties and event handling using interface elements such as drop-downs. Hence interface design can bypass traditional programming paradigms while still having program-like outcomes. For example one can achieve equivalent positioning of UI elements by either using a text-program representation such as `region:SetPosition(x,y)` or one can simply use multi-touch interactions to move the region to said position and store the position after a suitable interaction (double-tap, mode-selection via drop-down etc). As the direct layouting is more convenient on mobile devices than the text entry it suggest itself as a better representation of programming.

4. DISCUSSION

Neither of the specific examples discussed in the previous two sections is a definite solution. In either case there are clear drawbacks that relate to the trade-offs chosen. Neither of these proposed solutions is good at giving a large overview of a complex data flow. Scrolling is required. Hence there clearly is much to be researched to overcome the counter-acting pressures of limited screen space and getting a good overview.

At the same time the added constraint forces a more detailed look at what programming means and how to best structure both its input and visual representation. There are numerous ideas encoded in the examples discussed above that address this. Both use graphical representations. These are both more compact in selection and input and allow compression of visual real estate. Both use position as a means of structuring program flow. This is not unusual. keywords in a program text are also spatially organized. However, the choice of position in space is much more deliberate in strongly tied to not only the ability to read and aesthetics, but also relates to how input is provided and understandable by the programmer and how semantics is established.

Strategies for space management are crucial. Multi-touch devices are much better equipped to provide easy and convenient navigation than key based devices. It is hard to predict at this point if the hardware of mobile devices will become more uniform. Currently there is an array of screen sizes, touch input methods, and keyboard layouts. Given the sever constraints on will likely have to consider separate solution for each configuration to optimize the overall programming experience.

Key aspects of our position are hence:

- Consider any input as a possible programming activity. Input is programming as long as it established some functional relation.
- Consider any output as a possible representation of a program. As long as the output conveys to the programmer the functional meaning, syntax and semantics of the program it is considered a proper representation.
- Multiple representations may concur and should be supported. A program may be conventional text based, visual data-flow or any other representation.
- Semantics of language features such as API and data types emerge from the requirements of the interaction.

Hence the goal of turning mobile phones into primary programming platforms requires progress in understanding the relation of the interface hardware capabilities to program representations, type systems and APIs. Bandwidth for both entering and representing become crucial constraints that are directly imposed by the hardware capabilities of the mobile device itself.

5. CONCLUSIONS

We argued in this paper that one can consider mobile smart phones as primary programming devices, that is as devices which can be programmed without the need of external hardware. However, this suggests reconsidering how we think about programming to bring the effort of working with the input capability and the visual representation in better alignment with the intrinsic strengths and weaknesses of mobile phones. Text based input requires substantial input bandwidth and is fairly poor at offering ways to utilize screen space in supportive ways. Hence we explored numerous visual programming ideas for 12-key phones as well as multi-touch phones to illustrate how one might start to think about offering programming on these devices.

If this idea is successful one can hope that if one is stranded on a lonely island with just a mobile smart phone, a power generator (and perhaps generously with the complete works of Shakespeare) one might still be able to write programs to solve or support new problems that arise from the new-found island life!

6. ACKNOWLEDGMENTS

This paper was written while the author was hosted by Limin Jia and Lujo Bauer, both of Carnegie Mellon University. The hospitality and patience that allowed this paper to happen is much appreciated! Also many thanks to the anonymous reviewers for insightful and constructive feedback.

7. REFERENCES

[1] P. Baudisch and G. Chu. Back-of-device interaction allows creating very small touch devices. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 1923–1932, New York, NY, USA, 2009. ACM.

[2] L. Butts and A. Cockburn. An evaluation of mobile phone text input methods. *Aust. Comput. Sci. Commun.*, 24(4):55–59, 2002.

[3] S. K. Card, W. K. English, and B. J. Burr. Evaluation of mouse, rate-controlled isometric joystick, step keys, and text keys for text selection on a CRT. *Ergonomics*, 21:601–613, 1978.

[4] G. Essl. SpeedDial: Rapid and On-The-Fly Mapping of Mobile Phone Instruments. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Pittsburgh, June 4-6 2009.

[5] G. Essl. UrMus – An Environment for Mobile Instrument Design and Performance. In *Proceedings of the International Computer Music Conference (ICMC) (forthcoming)*, June 1-5 2010.

[6] J. Gong, B. Haggerty, and P. Tarasewich. An enhanced multitap text entry method with predictive next-letter highlighting. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 1399–1402, New York, NY, USA, 2005. ACM.

[7] R. Ierusalimschy. *Programming in Lua, Second Edition*. Lua.Org, 2006.

[8] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[9] A. Nandi and H. V. Jagadish. Effective phrase prediction. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 219–230. VLDB Endowment, 2007.

[10] L. Pere and M. Koniorczyk. A universal fast graphical user interface building tool for arbitrary interpreters. *Journal of Visual Languages and Computing*, 16:231–244, 2005.

[11] K. Zuse. Über den allgemeinen Plankalkül als Mittel zur Formulierung schematisch-kombinativer Aufgaben. *Arch. Math.*, 1:441–449, 1948/49.