Liquid: Library for Interactive User Interface Development

Georg Freitag[1], Dietrich Kammer[2], Michael Tränkner[1], Markus Wacker[1], Rainer Groh[2]

Fakultät Informatik und Mathematik, Hochschule für Technik und Wirtschaft Dresden[1]
Fakultät Informatik, Technische Universität Dresden[2]

Abstract

Developing multi-touch applications is a great challenge for developers: they have to adopt a novel interaction paradigm, but lack suitable tools and reliable design guidelines. The goal of *Liquid* consists in introducing the concept of visual programming to multi-touch technology, implementing a promising approach to overcome the difficulties in developing applications for multi-touch devices. As a novel feature, Liquid allows the development of multi-touch applications with the help of the technology itself. This contribution illustrates the application of visual programming in the multi-touch context, presents related work, and explains the workflow of Liquid with the help of an instructive example.

1 Introduction

In recent years, hardware with multi-touch interfaces has become readily available to both, researchers and consumers. In addition to direct and natural interaction (Shneiderman & Plaisant, 2010), this technology enables multi-user and multi-input scenarios. Due to the numerous fields of application, multi-touch devices are nowadays used e.g. in mobile phones, tablet computers, and interactive installations. Software programming and the design of user interfaces (UI) are already complex and challenging tasks. With the emergence of multi-touch technology, these tasks have become even more intricate (Hoste, 2010). It is noticeable that particularly for multi-touch applications, there is a lack of visual editors and toolkits, which meet the additional requirements or unify the process of designing and programming in one application.

Due to the variety of different devices, guidelines, and application scenarios, there is currently no universal design principle that combines the diverse aspects of multi-touch (Norman & Nielsen, 2011). As a result, developers have to choose self-developed or proprietary criteria to evaluate their designs and interfaces. This additional work is a

complex, time consuming, and iterative process, which includes programming, user studies, and evaluation.

This paper proposes an approach to optimize the iterative development process by means of a library for interactive user-interface development (Liquid). Liquid is a multi-touch application itself, thus unifying the process of development and developed applications on one and the same medium. Liquid combines a multi-touch framework with an editor and a runtime, which executes the developed applications, where the editor is based on a visual programming approach. Its advantages are explained in Section 2. Section 3 covers related work in the field of visual programming and is followed by a requirements analysis of Liquid and its design in sections 4 and 5, respectively. Conclusions and future work are discussed in Section 6.

2 Visual Programming

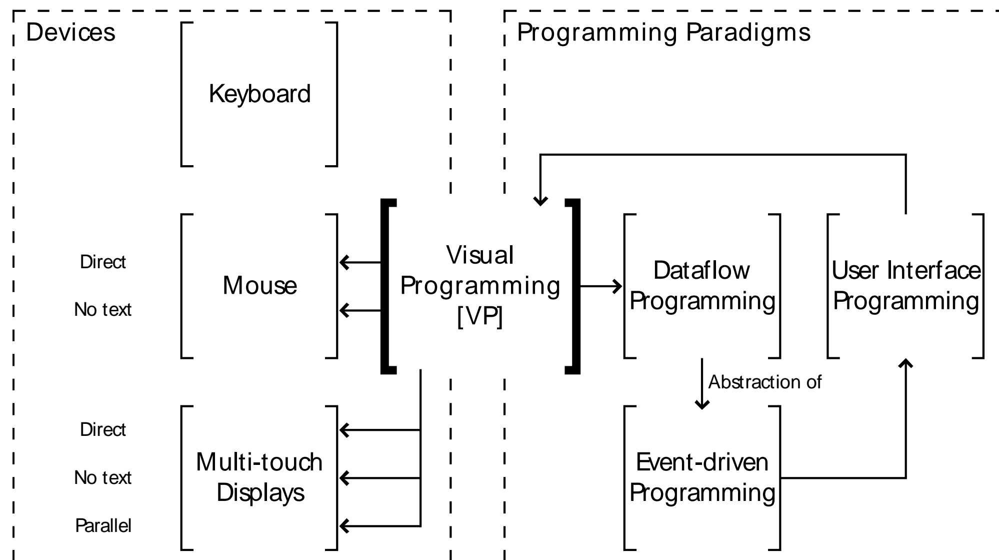

Figure 1 - Visual Programming depends on programming paradigms and devices

This section introduces visual programming (VP) and explains the basic principle behind its application in the case of multi-touch software development (cf. Figure 1). VP is beneficial to get an overview of the architecture and the available elements of an application. It visualizes dependencies between discrete objects in a spatial layout, which activates visual and spatial memory. This leads to an arguably improved performance in remembering structures and relationships (Kirsh, 1995). Moreover, visual programming permits easy and constant refinement of structures. Due to its immediate and abstract nature, it lends itself to

social and multi-user programming. In addition, it is easy to learn, even for “non-programmers”, e.g. designers and architects (Green & Petre, 1996).

The key elements of VP are preselected components, which ensure functionality and consistency. VP is commonly used for data-flow programming, which allows focusing on data while programming. This is supported by a pipe system architecture that permits data exchange by simply connecting input and output ports. VP facilitates the creation of structures and combinations between the contained elements; however, definition of complex application logic and algorithms is hardly feasible with this programming paradigm. Well-known visual programming environments are LabView (LabView, 2011), MAX/MSP (MAX/MSP, 2011) and VVVV (vvvv, 2011).We suggest that the application of VP in the context of multi-touch development is especially effective based on the following reasoning. Event-driven programming is about the interaction of components using separated or permanent events that are passed between them. Due to the similarity to dataflow programming, we suggest that events can be interpreted as dataflow. Consequently, event-driven programming should be feasible with VP. But since UI programming is event-driven by nature, UI programming for multi-touch displays should be realizable by VP as well (cf. Figure 1). Another argument for the application of VP to multi-touch displays is that they are mainly based on direct interaction and little text input. Moreover, parallel input is more feasible with the graphical approach of VP than with traditional code editors. Thus, we propose that VP can be performed even more effectively with multi-touch displays compared to traditional input devices such as a mouse. In addition, the strengths of the keyboard with regard to text input are less relevant for VP. As a result of these considerations, multi-touch actually fits VP best and the keyboard is the least suitable device for VP (cf. Figure 1).

3 Related Work

As mentioned above, there is a lack of visual tools for multi-touch based software development. Nevertheless, visual programming languages are already used in diverse research projects and editors. This section presents representative examples of these projects and specifies similarities and differences with regard to Liquid.

3.1 Squidy

Squidy is an interaction library for immediate development of so called *natural user interfaces* (König et al., 2009). The library allows combination of a variety of input and output devices via a pipe system. The modification or separation of data in the data-flow is achieved by filters and self-defined objects. Any component of Squidy is a separate node with predefined ports, which encapsulate the logic and algorithm. Furthermore, Squidy uses a zoomable user interface to hide and structure detailed information of components and pipes. Another notable feature is live programming, which allows the adaptation of components during runtime.

3.2 MaggLite

Another VPL Toolkit for user interface prototyping is MaggLite (Huot et al., 2004). MaggLite considers hardware as well as design aspects. Therefore, different views of a UI, which is under construction, are distingiushed. Users visualize the interface by drawing it in the main view of the MaggLite application. At this point, no functionality is defined. The second view is the scene graph of the application, in which the user structures the drawn interface components into a hierarchy. The last view contains an interaction graph (cf. Figure 2). It allows combination of input devices in the form of components as well as modification of the data-flow using meta-objects. The combination of these three views enables an incremental way of application development.

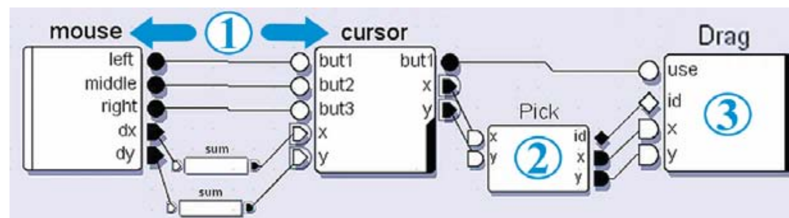

Figure 2 – MaggLite interaction graph combines hardware input with a cursor object

3.3 Comparison

One main difference of these approaches is its area of application. While Squidy focuses on hardware and software architecture, MaggLite offers an entire prototyping toolkit for design, structure, and behavior. Hence, MaggLite is based on the same idea as Liquid, realizing the entire development process within one program, but focuses on other aspects apart from interaction design. Moreover, Squidy is similar to Liquid with respect to the implementation of a live programming approach, which allows direct testing and adaptation during runtime. Each of the presented projects distinguishes between its input devices to consider and implement different kinds of input. In contrast, Liquid combines different input to offer a unified interaction style.

Relevant similarities of the presented projects compared to Liquid are the combination of VPL and components as well as a focus on a simple pipe system. To ensure a flexible and adaptable VPL, standard ports and defined data sets are used. Another similarity is the visualization of additional information of components and pipes, either by a separate view (MaggLite), a zoomable user interface (Squidy) or an overlay (Liquid).

4 Requirements and Design Decisions

There are a lot of requirements for an editor that is used to create multi-touch based applications using a visual approach. Furthermore, Liquid focuses on the development of multi-touch applications on multi-touch enabled devices. This multi-touch approach leads to additional requirements for the VPL-based editor, which are discussed in this section.

4.1 Multi-Touch

Multi-touch as the primary input of an application results in a variety of requirements for software development. Contrary to conventional applications and operation systems, a multi-touch application enables parallel processing of interactions and leads to novel interaction techniques. Such applications allow users to interact with both hands simultaneously. Due to the use of fingers or pens as input devices, the size of interactive elements has to be adapted (Sears & Shneiderman, 1991). Moreover, input areas as well as gestures should be well-defined to ensure the usability of such an application. Furthermore, suitable forms of feed-forward and feedback ensure that a user understands the interaction with elements of the user interface (Widgor, et al., 2009).

However, direct input has drawbacks such as the lack of tactile feedback, the challenge of text input without keyboard (Buxton et al., 1985), or the occlusion of content by direct use of hands (Vogel et al., 2009). Direct input is associated with a lack of established feedback mechanisms, like the different shapes of the cursor icon or tooltips when hovering over a user interface element. These challenges and disadvantages have to be considered when designing an editor and its corresponding VPL.

4.2 Visual Programming Language

The requirements of a VPL depend on its application scenario. If the development of algorithms is the primary use case, a VPL has to offer a variety of control flow elements (e.g. statements, loops, and function calls). In contrast, the development of UIs focuses on the combination of discrete components. These components offer predefined input and output ports to connect the data-flow with an algorithm, which is encapsulated in a component. To control the flow of data, the user connects ports with the help of a pipe system. While a direct combination of visual (UI) components is preferred, filtering and manipulation of the dataflow is sometimes required. This can be achieved using external manipulators which are invisible to the user at runtime.

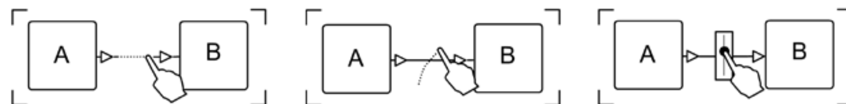

Figure 3 - Basic interaction between components: connect, disconnect, and manipulate

In conclusion, the VPL has to support three basic interactions: The connection of two ports by a pipe, the disconnection of two ports, and the manipulation of existing pipes (cf. Figure 3). To ensure the data exchange between connected ports, a type system has to be used. Because of the multi-touch requirements, the design of the ports as well as the design of the interaction has to be adapted. On the one hand, comprehensible and explicit gestures for each of the three basic interactions have to be identified. On the other hand, the size of ports has to be designed carefully. Moreover, feed-forward mechanisms need to be implemented, which visualize interaction using gestures. An approach, which is exploited by Liquid, is the use of the icon based *gesturecons* (Gesturecons, 2011). They contain subsets of icons or small images as a graphical abstraction of gestures to hint at interaction possibilities.

4.3 Editor

Liquid introduces the immediate creation and modification of multi-touch based applications. To this end, the editor provides live programming, which allows development and testing at the same time. As a consequence, a touch input has to be differentiated between editor (programming) and application (testing). To solve this issue, the editor can be implemented as an overlay or as a separated part of the interface. By choosing the overlay approach, the user interacts either with the editor or the application. In contrast, separation of the editor allows testing and programming at the same time, but requires more screen real estate (cf. Figure 4). Additionally to the distinction of input, the editor visualizes supplemental editing information of each UI element (e.g. ports and gestures).

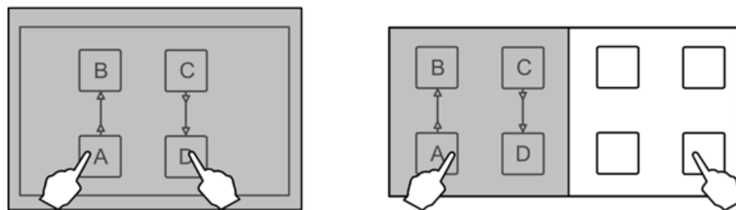

Figure 4 – Separation of editing and testing mode: Left: the editor overlays the application (implemented by Liquid); Right: the editor uses a part of the entire interface

Further requirements of the editor are the creation and adaptation of components as well as the extension of components by ports and gestures. The definition of input and output ports is required for every VPL. Ports define data access to the component, whereas gestures encapsulate recognition algorithms. Gestures and ports can be connected via the pipe system. Fundamental requirements of the multi-touch framework, which is integrated into the editor, are described by (Kammer et al., 2010a). For instance, platform and hardware independence and a suitable event system to process multi-touch input are necessary. Liquid relies on the flash runtime to establish platform and hardware independence. The library, which already contains a variety of predefined components, ports, and gestures, requires an extensible and

adaptable design to allow the straightforward implementation of further elements. For this purpose, predefined interfaces and factories are implemented in the underlying framework.

5 Workflow and Interaction

In this chapter, the typical workflow of Liquid is explained by means of a simple example: the implementation of a touch controlled scrollbar. In summary, we illustrate the visual programming steps where the vertical position of one object is manipulated by another. Figure 5 (top left) shows the Liquid application with an editor as overlay. Three buttons are displayed on the left side: *Components*, *Ports*, and *Gestures* (from top to bottom). When pressing and holding the buttons, the program switches to a separate interaction mode. Due to the dependencies between user interaction and mode, Jef Raskin established the term *quasimode* (Raksin, 2000). Pressing a button is a support task that needs no visual attention and is performed by the non-dominant hand (Owen et al., 2005). In our example, the left hand is expected to be non-dominant. Hence, the buttons are oriented to the left side. In contrast to the non-dominant hand, the dominant hand is responsible for complex and precise interaction (e.g. selection and manipulation of objects). The use of modes and its execution with the non-dominant hand leads to a well structured application and efficient interaction design (Ruiz et al., 2008).

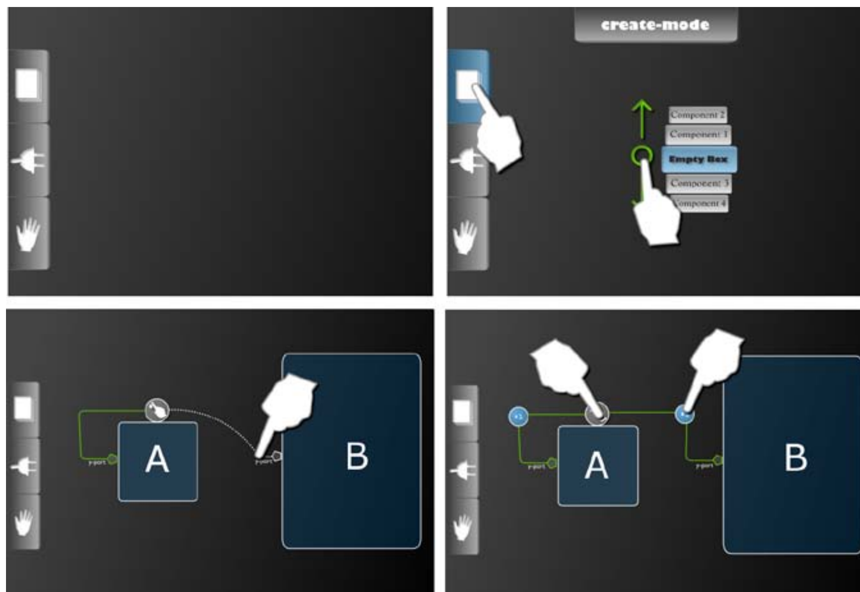

Figure 5-Top left: the three main buttons of Liquid: Components, Ports, and Gestures; Top right: while switching the mode with the non-dominant hand, the dominant hand opens an interactive menu and selects a new component; Bottom left: drawing a pipe between the drag-gesture and the entry-port; Bottom right: redefine the manipulation value of a pipe.

Without pressing a button, both hands can be used to interact with already created components. The selected instance of a component can be moved with one finger, or rotated and scaled with two fingers. To create a new instance, the non-dominant hand presses and holds the topmost button to switch into the quasimode for component creation. The dominant hand taps on an empty area of the interface to open an interactive menu, which contains all available components (Figure 5, top right). After a menu item is selected, an instance will be created precisely at the last position of the dominant hand. In Figure 5 (top right) a component with no predefined ports was chosen (component A). The element is extended with ports and gestures by pressing the button for port (second) or gesture (third) with the non-dominant hand while the dominant hand presses and holds the element. Again, an interactive menu opens and presents the available ports and gestures. In this example, the empty element becomes the vertical scrollbar. Such a scrollbar needs a port to manipulate its vertical position as well as a gesture to drag it vertically. In Figure 5 (bottom left) the appropriate y-port and the vertical drag-gesture of component A were added and connected to each other.

A second component is needed whose position can be manipulated (component B). We again create an empty object, adapt its size, and add the y-port. However, a separate drag-gesture is not required. To manipulate the position of both elements by only dragging the first one, the drag gesture and both ports must be connected. This is achieved by drawing a line between gesture and ports as shown in Figure 5 (bottom left). As a result, the vertical movement of component A is the same as component B. To create a scrollbar, where the movement direction is inverted, the pipe between the gesture and the y-port of component B has to be adjusted. For this purpose, the gesture is selected and the manipulation values of the pipes become visible. By dragging the value of the pipe between gesture and component B, the manipulation is changed to `-1`. This is the inverse of the former dragging value (Figure 5, bottom right). Now, both objects move contrary to each other.

6 Conclusion and Future Work

In this work, we introduced the development environment Liquid. Its aim is to immensely simplify the creation, modification and testing of multi-touch based UI applications. Liquid provides a task-specific interface for simultaneous testing and programming of applications by incorporating a viable VPL. This approach offers the potential to develop practical applications more directly and effectively. Moreover, VP appeals to non-experts in programming, which promises to include professionals from other domains into the development process (Kaindl, 2010). In contrast to the variety of multi-touch frameworks, which focus only on the programming part, Liquid aims towards a unification of framework, editor, and runtime in one tool on one device. Hence, programming, designing, and testing are merged smoothly into one another. As described in Section 5, an easily drawn and adapted network of pipes is instantly transformed into a multi-touch application. Such software benefits from the use of multi-touch devices during the development process, because related interaction principles and gestures are used by the programmer as well. Due

to the simplified iterative development process, generated feedback of an evaluation can be immediately taken into consideration. Another aspect is the extension of predefined ports and gestures. Since ports are an external representation of an internal algorithm, an immediate customization is a considerable challenge. In contrast to ports, automated gesture recognition is a conceivable task, especially for *offline gestures,* as defined by (Kammer et al., 2010b). In this paper Kammer et al. present an approach to formalize gestures, which will be implemented in Liquid in the future. Faster and more efficient development of multi-touch UIs by using a visual programming approach compared to a textual approach has to be proven. Hence, a setup of a user study is required, which measures the task time and error rate.

7 References

Buxton, W., Hill, R., & Rowley, P. (1985). Issues and techniques in touch-sensitive tablet input. *Computer graphics and interactive techniques* (pp. 215-224). San Francisco, USA: ACM.

Gesturecons. (2011). Retrieved March 20, 2011, from http://gesturecons.com/

Graham, N., Morton, C., & Urnes, T. (1996). ClockWorks : Visual Programming of Component-Based Software Architectures. *Journal of Visual Languages & Computing* (Volume 7, Issue 2), 175-196.

Green, T., & Petre, M. (1996). Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *Journal of Visual Languages & Computing* (Volume 7, Issue 2), 131-174.

Hoste, L. (2010). Software engineering abstractions for the multi-touch revolution. *International Conference on Software Engineering* (pp. 509-510). Cape Town, South Africa: ACM.

Huot, S., Dumas, C., Dragicevic, P., Fekete, J., & Hégron, G. (2004). The MaggLite post-WIMP toolkit: draw it, connect it and run it. *User interface software and technology* (pp. 257-266). Santa Fe, NM, USA: ACM.

Kammer, D., Freitag, G., Keck, M., & Wacker, M. (2010a). Taxonomy and Overview of Multi-touch Frameworks: Architecture, Scope and Features. *Workshop on Engineering Patterns for Multitouch Interfaces.* Berlin: EICS.

Kammer, D., Taranko, S., Wojdziak, J., Keck, M., & Groh, R. (2010b). Towards a Formalization of Multi-touch Gestures. *ACM Interactive Tabletops and Surfaces 2010* (pp. 49-58). Saarbrücken: ACM.

Khaindl, G. (2010). Towards a flexible software framework for multi-touch. *A workshop of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems.* Berlin, Germany: EICS.

Kirsh, D. (1995). The intelligent use of space. *Artificial Intelligence* (Volume 73, Issue 1-2), 31-68.

König, W., Rädle, R., & Reiterer, H. (2009). Squidy: a zoomable design environment for natural user interfaces. *Proceedings of the 27th international conference extended abstracts on Human factors in computing systems* (pp. 4561-4566). Boston, MA, USA: ACM.

LabView. (2011). *The LabVIEW Environment*. Retrieved March 20, 2011, from http://www.ni.com/labview/

MAX/MSP. (2011). *Cycling 74 - Tools for Media*. Retrieved March 20, 2011, from http://cycling74.com/

Norman, D. A., & Nielsen, J. (2011). *Gestural Interfaces: A Step Backwards In Usability*. Retrieved March 20, 2011, from http://www.jnd.org/dn.mss/gestural_interfaces_a_step_backwards_in_usability_6.html

Owen, R., Kurtenbach, G., Fitzmaurice, G., Baudel, T., & Buxton, B. (2005). When it gets more difficult, use both hands: exploring bimanual curve manipulation. *Graphics Interface* (pp. 17-24). Victoria, British Columbia: Canadian HC Communications Society.

Raksin, J. (2000). *The Human Interface.* USA: Addison Wesley.

Ruiz, J., Bunt, A., & Lank, E. (2008). *A model of non-preferred hand mode switching.* Windsor, Ontario, Canada: Canadian Information Processing Society.

Sears, A., & Shneiderman, B. (1991). High precision touchscreens: design strategies and comparisons with a mouse. *Int. J. Man-Mach. Stud.* (Volume 34, Issue 4), 593-613.

Shneiderman, B., & Plaisant, C. (2010). *Designing the User Interface.* Upper Saddle River: Pearson Addison-Wesley.

Vogel, D., Cudmore, M., Casiez, G., Balakrishnan, R., & Kelih, L. (2009). Hand Occlusion with Tablet-sized Direct Pen Input. *Human factors in computing systems* (pp. 557-566). Boston, USA: ACM.

vvvv. (2011). *vvvv - a multipurpose toolkit*. Retrieved March 20, 2011, from vvvv.org

Widgor, D., Williams, S., Cronin, M., Levy, R., White, K., Mazeev, M., et al. (2009). Ripples: utilizing per-contact visualizations to improve user interaction with touch displays. Victoria, BC, Canada: ACM.