Direct Manipulation Programming Systems (A Brief Introduction)

RAVI CHUGH, University of Chicago

Programming languages and direct manipulation user interfaces are distinct approaches for creating digital objects that combine visual and textual elements, such as documents, graphics, web applications, games, and data visualizations. At one end of the spectrum, programming languages provide experts an array of abstraction mechanisms to generate complex output, but the reliance on text-based representations (*i.e.* source code) limits the pace of productivity. At the other end, direct manipulation interfaces cater to a wide range of users with intuitive and interactive graphical tools, but the lack of abstraction capabilities results in tedious, repetitive work and, worse, limits users to features that are provided.

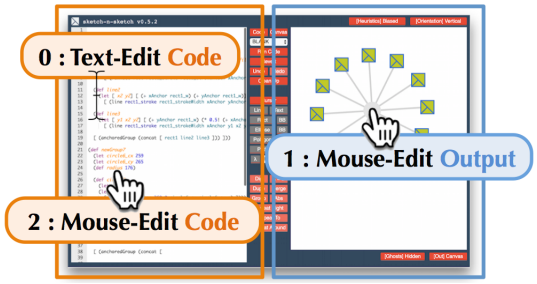
We propose a vision for *direct manipulation programming systems* that augment general-purpose, text-based programming languages with powerful new direct manipulation capabilities, a goal closely related to *live programming* (McDirmid 2013). As a first step, we are developing Sketch-n-Sketch, an IDE for creating programs that compute Scalable Vector Graphics output. Sketch-n-Sketch is a direct manipulation programming system equipped with (1) new direct manipulation techniques for manipulating the output of a program, and (2) new direct manipulation techniques for manipulating the source code of a program. Our goal can be summarized with the motto: "Programming with Less Keyboard, More Mouse."

1 DIRECT MANIPULATION OF OUTPUT

The iterative Edit-Run-View workflow of text-based programming is tedious. Oftentimes it would be preferable to directly change the output of the existing program and have the system synthesize updates to the program to match the changes.

In Sketch-n-Sketch, we have designed tools for interacting with the output of a program, which trigger automatic synthesis of updates to the program (Chugh et al. 2016; Hempel and Chugh 2016). These tools cater to four phases of development common to programming vector graphics. (1) In the *Draw* phase, as the user adds shapes to the canvas as they would in a direct manipulation editor, Sketch-n-Sketch inserts appropriate starter code for these new shapes in the program. (2) In the *Relate* phase, the user selects attributes in the output and instructs the system to relate them in some way — for example, making position or color values equal, or distributing points equally — and Sketch-n-Sketch finds a way to encode that relationship while fitting within the structure of the program. (3) In the *Abstract* phase, the user selects shapes to group, and Sketch-n-Sketch rearranges the corresponding code into a single abstraction parameterized over design parameters, so that the design can easily be duplicated (via function call). (4) In the *Tweak* phase, the user directly modifies position and color attributes, and Sketch-n-Sketch infers small updates to constant literals in the program in real-time to synchronize the program with the changes.

Currently, we are extending Sketch-n-Sketch with the ability to synthesize new relationships that the user may wish to incorporate into the program. We are also designing ways to interact with the user when there are multiple valid ways to update the program. For example, we use dialog boxes to display multiple potential program updates, showing a preview of the updated code and output when the user hovers over each choice.



2 DIRECT MANIPULATION OF CODE

When synthesizing program updates, Sketch-n-Sketch attempts to keep the program readable so that the user may text-edit the program if and when needed. Nevertheless, there are tedious and error-prone text-editing tasks that retard productivity. Automated tools help with certain refactoring tasks, but often not for several kinds of fine-grained, *micro-level refactorings* (Ko et al. 2005; Ko and Myers 2006) that are common to the coding process.

We are developing a structure-aware code editor augmented with *drag-and-drop refactoring* (Lee et al. 2013) support for several common tasks, such as: Reorder Expressions; Combine or Split Definitions; Introduce Local Definition; Merge Expressions and Introduce Lambda; and Add or Remove Function Parameter. When the user triggers a transformation via direct manipulation, the editor reasons about the scope and dependencies of the program to determine whether the transformation is valid. When there are multiple options, the user can inspect the source code diff for each option before choosing the desired transformed program.

Together, direct manipulation of code and output provide interactive tools to semi-automate common programming tasks, so that the user may instead spend keystrokes on tasks that require more human insight.

ACKNOWLEDGMENTS

Thanks to Brian Hempel, Jacob Albers, Grace Lu, Justin Lubin, and Mitch Spradlin for their collaboration on this project.

REFERENCES

Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Conference on Programming Language Design and Implementation (PLDI)*.

Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Symposium on User Interface Software and Technology (UIST)*.

Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. 2005. Design Requirements for More Flexible Structured Editors from a Study of Programmers' Text Editing. In *Human Factors in Computing Systems (CHI)*.

Andrew J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Human Factors in Computing Systems (CHI)*.

Yun Young Lee, Nicholas Chen, and Ralph E. Johnson. 2013. Drag-and-Drop Refactoring: Intuitive and Efficient Program Transformation. In *International Conference on Software Engineering (ICSE)*.

Sean McDirmid. 2013. Usable Live Programming. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*.