Transparency and Liveness in Visual Programming Environments for Novices

Steven L. Tanimoto
Dept. of Computer Science and Engineering
University of Washington
Seattle, WA, 98195, USA
tanimoto@cs.washington.edu

***Abstract*—Blocks-based programming environments offer an alternative program representation to textual source code that simplifies the considerations of syntax for novices. This position paper raises the question of whether additional affordances related to transparency of data, transparency of semantics, and liveness of execution can be consistently added to these environments to obtain some significant advantages for these novice programmers.**

Keywords—programming environment, learning to code, novice programming, blocks-based, transparency, liveness, visual programming, data factory, live coding.

I. Introduction

Blocks-based programming languages represent the computer program with a diagram that shows juxtaposed rectangular or prismoidal blocks in an assemblage [1, 4]. Legal program syntax is hinted and/or enforced through mechanisms such as jigsaw-puzzle-style interlocks, color coding, icon coding, and the shapes of nested containers. These clues and constraints provide pedagogical scaffolding that permits novice programmers to shoulder a reduced cognitive load when building and editing programs.

Additional affordances, especially transparency of data and semantics, and livenes, can further reduce the cognitive load. Blocks-based languages support these less consistently than they do program syntax. Better support for them could further facilitate programming by novices.

II. Transparency of Data and Semantics

Transparency of data in a programming environment refers to the visibility of program state. It traditionally has been minimal, and debugging tools have been needed to see any value that was not explicitly printed by the program or reflected in the graphical display as a result of extra coding for this purpose. Transparency of data is helpful in understanding what a program is doing. Another level of transparency can be called semantic, and it refers to the functionality that is or is intended to be implemented in a program. Affordances for data transparency are one means towards semantic transparency, because semantics deals with change in state. But other approaches include the use of suggestive icons for program constructs, and program animation.

The lack of transparency of data was long a challenge for novice programmers, and they were encouraged to sprinkle print statements throughout their code in order to find out what the values of their variables would be during execution. With the use of microworlds in novice programming environments such as Logo, Agentsheets, and many newer systems, the values of variable have often been directly reflected by the state of the microworld display -- the turtle's position, for example. Still, most systems hide variable values by default, which typically imposes another cognitive load on novices. The issue is, therefore, the question of how to lower that cognitive load by altering the environment. In my "Data Factory" system of 2003, I took transparency to an extreme: all values are always visible. This doesn't scale to large programs, but it gives a consistent affordance at small scales. Finding the right balance for various learning scenarios is the issue.

III. Liveness

Liveness of execution, as a characteristic of a programming environment, relates to the latency between editing operations (such as changing a programming construct in a program) and the visible change in the program's behavior as a result of the edit. An example of a novice programming environment that incorporates a form of liveness is Sonic Pi, developed by Sam Aaron for the Raspberry Pi and in use by many young, novice coders in the U.K. The ability of Sonic Pi to support live coding (programming of music synthesis in front of an audience) is a major selling point contributing to the environment's wide adoption. My position is that the best programming environments for novices will tend to provide multiple forms of scaffolding. Blocks-oriented environments do particularly well on scaffolding for syntax. However, they incorporate these other aspects with varying degrees of success.

Liveness is another feature that can potentially reduce the cognitive load of programming for novices. In traditional environments, there is a significant time delay between programming act (e.g., adding a statement, changing a constant's value) and seeing its effect on execution. The delay could be arbitrarily long -- so long that many edits might be made before seeing the effect of any one of them. This means that the set of possible causes for a bug, in the mind of the

978-1-4673-8367-7/15/$31.00 ©2015 IEEE

programmer, could be quite large. Lowering that latency not only caters to short attention spans but reduces the cognitive load of debugging -- typically lowering the set of seemingly plausible causes for the bug. Not only that, liveness helps the programmer to quickly understand the effect of a program change on execution. How can novice programming environments such as blocks-based ones be made more "live?" A variety of techniques are possible. Some of these were covered in my recent keynote at the International Conference on Live Coding held during July 2015 at Leeds, UK. Techniques include using implicit loops around the code in focus, secondary executions, predictive execution, and automatic checkpointing of execution to responsively support an evolving program semantics.

REFERENCES

[1] E. P. Glinert, "Towards "second generation" interactive, graphical programming environments," in 2nd IEEE Computer Society Workshop on Visual Languages, 1986, pp. 61–70.

[2] S. L. Tanimoto, "Programming in a Data Factory," Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'03), Oct 2003.

[3] S. L. Tanimoto, "A perspective on the evolution of live programming." Proc. LIVE 2013, workshop on Live Programming, San Francisco.

[4] F. Turbak, S. Sandu, O. Kotsopoulous, E. Erdman, E. Davis, and K. Chadha, "Blocks languages for creating tangible artifacts," in IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '12), Oct. 2012, pp. 137–144.