The Promise of Live Programming

Sean McDirmid
Microsoft Research Asia
smcdirm@microsoft.com

Abstract

Live programming aims to create a more fluid feedback loop between the programmer and programmed. Unfortunately, it is not very clear what this feedback does and how it is useful: does it just lead to better debugging, or to something revolutionary? To answer this question, we clarify the experiential design challenges that must be overcome before live programming can emerge as a serious topic.

Live Programming

Live programming has been characterized as eliminating the edit-compile-test loop so code can be edited while it is executing. Realizing that experience is mostly a technical challenge: how can compilation and re-execution overhead be reduced so programmers can get timely feedback about how their edits effect program execution? Although automatic re-execution is a huge problem, it is not insurmountable; e.g. omniscient debuggers [17] can trace all computations with brute force, which can then "only" need to be incrementally updated according to edits. This is not an impossible problem and as Bracha points out, is anyways avoided if pieces of code can be executed/tested separately [1]. The biggest challenge instead is live programming's usefulness: how can its feedback significantly improve programmer performance?

Live programming feedback must be presented in a form that allows for quick reaction. Edwards' example-centric programming [3] projects example context into a code editor, where it is then available for quick introspection while writing code. Previous work in [15] shows how execution values can be inlined into the code editor as meta-text, and printf traces enable programmers not only to see a customized overview of how their program executes, but also navigate to various code and execution contexts associated with the line. DejaVu [10] shows how execution can be presented across a timeline (Figure 2); it is also possible

Figure 1. A fictional programming experience in Iron Man 2008.

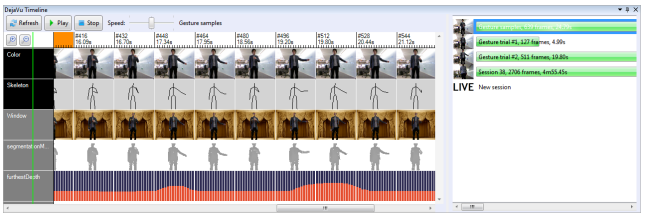

Figure 2. A timeline in Deja Vu.

to graph or even "strobe" the values so programmers can clearly see how they change over time [25].

Perhaps live programming is just super debugging with accessible execution context where programming otherwise stays the same. The experience, however, would not be very fluid: discrete key strokes applied to a text buffer mean the feedback is often not useful. At worst, feedback on an incomplete model can limit our thinking about a problem, distracting rather than helping us [27]. The live programming story is too incomplete at this point.

To envision a possible future for live programming, consider the fictional programming experience from Iron Man the movie (2008) shown in Figure 1. Most of the interesting problem solving in programming currently occurs in programmer heads augmented by notebooks and whiteboards, so movies must envision bringing that experience into an environment where it can be shared with the audience. The resulting interfaces are rich in affordance and fluid feedback, making them exciting to watch, but are of course complete fantasy. Could we "for real" bring the interesting parts of

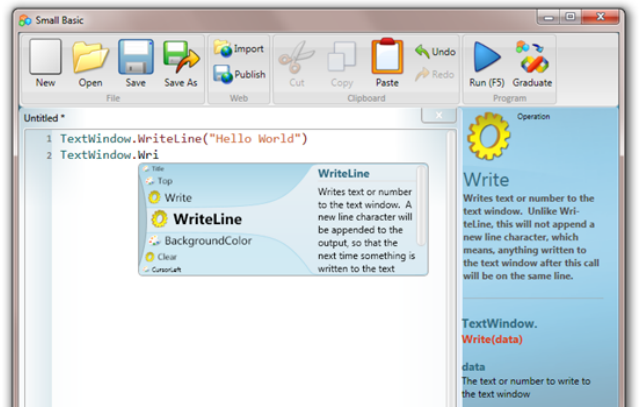

Figure 3. Intellisense (code completion) in Small Basic.

programming out of the programmer's head and into the environment where the computer can then better assist the programmer in a fluid experience rich in feedback and affordance? This position paper explores that opportunity.

The Water Hose and Scrubbing

Chris Hancock [7] points out that live programming should go beyond "continuous feedback" to be more like a **water hose**. The waterer *never stops aiming* at their target; they just move the stream of water in an ever evolving aim until the target is hit, while an archer loses much of their aiming context after every shot arrow. With a water hose, a programmer makes small changes to code while observing small changes to program output that get closer or farther away from their goals. The water hose is then analogous to a continuous function that defines an interpreter where small changes in code input lead to small changes in execution output.

As an example of a water hose, code elements can be edited with **scrubbing** via knob/slider-like widgets with live feedback on how the changing value affects program behavior [24]. A small mouse movement causes a small change in value and a small change in program output; e.g. scrubbing a position causes a shape to move slightly on each slight motion. Live programming demos often include scrubbing because of this fluid feedback.

Unfortunately, scrubbing only works for abstractions that exist in a continuum, such as values like numbers and points, or finite sets of interchangeable terms like literal colors. Scrubbing also cannot aid in writing, rather than modifying, code and, anyways, it cannot assist with the abstract reasoning that burdens programmers the most when writing code. This latter limitation is an intrinsic to the water hose: the decision on whether to use an abstraction or not has a large effect on the program's behavior. It is just not realistic that a programmer could "scrub" a program into existence.

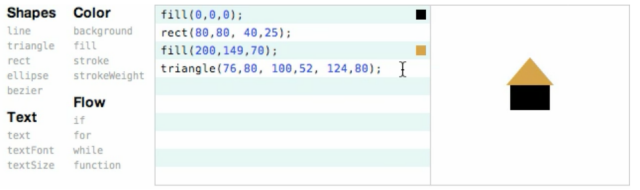

Figure 4. Dump parts on the floor.

Create by Reacting and Abstracting

Our abstract reasoning capabilities is our capacity to reason about and solve novel problems by identifying patterns and relationships that underpin these problems [2]. Computers currently have little capacity for abstract reasoning; e.g. machine learning is great at generalizing over lots of data, but any abstractions used in the learning process must be baked in. The limitation is similar to that of the water hose: there is just no feedback to smoothly iterate on to determine if code is "better" or not as abstractions are added/removed.

Computer-assisted abstract reasoning is much more viable. With the principle that "multiple choice" is easier than "fill in the blank," many programming environments today augment reasoning with *code completion*, which helps programmers recall what abstractions they might want to use without memorizing them [14] (e.g. see Figure 3). Live programming further enhances code completion by using runtime context in the selection process as well as by providing feedback about what each choice will do. Bret Victor explores many live code completion concepts as **create by reacting** in [25], described using a painter's analogy:

> An essential aspect of a painter's canvas and a musical instrument is the immediacy with which the artist gets something there to react to. A canvas or sketchbook serves as an "external imagination," where an artist can grow an idea from birth to maturity by continuously reacting to what's in front of him.

Victor then lists *get something on the screen as soon as possible* and *dump the parts bucket onto the floor* as two principles that move in this direction. For the former, the effect of a code completion selection can be realized as soon as it is selected, or even when it is just navigated to, depending on the API, creating immediate feedback. If the API has arguments, reasonable defaults can be used that can then be tweaked through scrubbing. Dumping the parts on the floor means relevant APIs can be listed even before code completion menus are activated (Figure 4), giving programmers "raw material to spark new ideas." As Smith puts it [21]:

> Pablo Picasso said "the most awful thing for a painter is the white canvas." One of his most memorable paintings, "The Studio at Cannes," is of his own studio. Its walls and ceiling are covered with paintings. Priceless works of art are jumbled together everywhere. But right in the middle of the room stands an easel holding a blank canvas. This is a

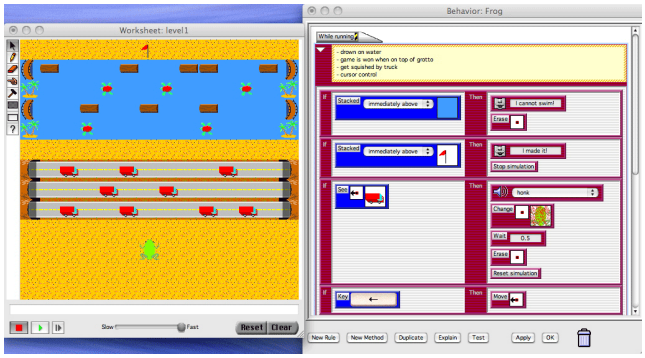

Figure 5. Conversational programming in [18].

> fitting image of the problem facing all creative people: how to get started. A blank coding pad is as much a barrier to programming creativity as a blank canvas is to a painter.

Repenning explores similar concepts as *conversational programming*, which harnesses computing power so programmers proactively knows how their code affects program execution as they explore various chosen contexts (Figure 5) [18]. For example, by dragging a car around the programmer can move it to different parts of the city to explore different scenarios. The programmer interacts with the world by changing it in order to get information back about the behavior of an agent–the programmer can drag a car to a different traffic light and discover that that car would turn in the wrong direction. Useful code then flows out of the world's configuration; e.g. if a frog is placed on top of water, then one of the choices shown in the code menu can be an expression that captures that relationship as a condition.

Given complex program goals, live code completion can be limited in effectiveness. First, many abstractions have parameters that must be filled in before they have meaningful execution, where reasonable default values do not always exist. Second, API calls often fit into an architecture that the programmer must prepare for ahead of time, or must be used in combinations that leave the code un-executable for long periods of time. Lastly, it can be difficult to manifest the example execution context needed to inform code completion approaches or suggest abstraction opportunities–some might even be impossible to describe without abstraction!

As another way forward, Bret Victor demonstrates in [25] **create by abstracting** how the programmer can interactively abstract a concrete code example it by combining constants with the same values into variables that can then be lifted into procedure arguments. Constants that do not match exactly can still be combined into variables, with the remainder of the value being accounted for in an expression; e.g. a variable "x" with the value of 100 can be used for the constant 50 as "$x + 50$;" if that was not intended, it can be scrubbed to another expression; e.g. "$x \div 2$." The computer helps out by making refactoring for abstraction easy as well as computing equivalent expressions when values do not match exactly. Still, too much burden is placed on the programmer in identifying abstraction opportunities; e.g. how do they know that two 100 values should be the same, how do they make the leap to relate 100 and 50, and anyways what if the values are related in complex ways?

The example in [25] involves shapes that can be drastically improved on with direct manipulation [19]: the programmer can draw shapes on the screen and activate "guides" that can codify as constraints the nearly coincidental relationships in the example (e.g. two edges of two shapes almost aligned). Direct manipulation makes the relationships obvious: the 100 values can be combined because they represent height of shapes that should be the same, and $\div 2$ was the right way to relate 100 and 50 because the latter was a radius of a circle and we just want the circle's "height" to be the same even if that isn't a direct value. The computer then assists programmers by highlighting abstraction opportunities. Unfortunately, it is not clear if similar assistance can be applied more broadly beyond 2D graphics.

Direct Manipulation

Live programming is naturally related to *direct manipulation* [19] as characterized by:

- Continuously represented objects of interest;
- Physical actions instead of complex syntax; and
- Rapid, reversible, and incremental actions on objects of interest with immediate feedback.

Continuous representation of objects is part of a steady frame of feedback where rapid, incremental, and feedback-rich actions resembles the water hose. Given a 2D scene programmed in code, directly combining constants into variables (as in [25] "create by abstracting") is not obvious as numbers lack meaning in the abstract. However, moving and resizing shapes directly in the 2D scene can be augmented with alignment guides that perform a similar constant combining that is much more obvious since it is direct.

For programming, direct manipulation often means code and execution are combined, which is otherwise not necessary for live programming. Morphic [12] is a UI toolkit based on direct manipulation of morphs that also supports liveness; however, no abstraction is supported as changes to morphs are not reflected in code. Attempts have been made to support direct manipulation with abstraction. In Edwards' Subtext, for example, code is simply literal computation that is then copied around as needed [4] (Figure 6). Programming in Subtext is inspired by "WYSIWYG" programming in spreadsheets, where actual values are computed immediately while expressions still refer to abstract "cell" locations. In Elliott's *tangible functional programming*, Eros programs parts are represented as "tangible values" that directly represent functional computation but also have the ability to be composed directly [6] (Figure 7). In both systems, directly manipulated objects reify code and abstract behavior.

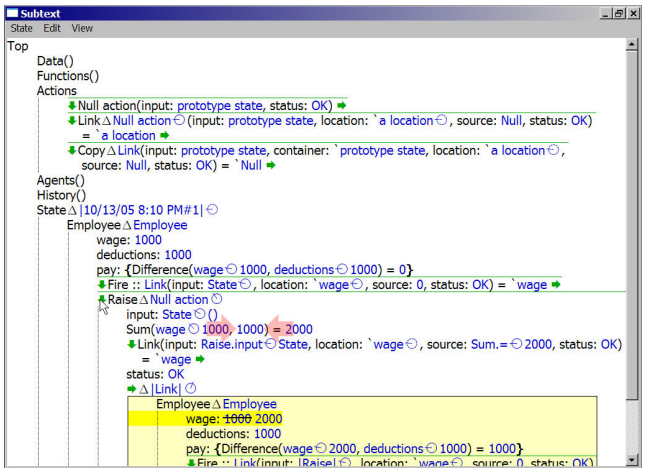

Figure 6. Programming in Subtext.

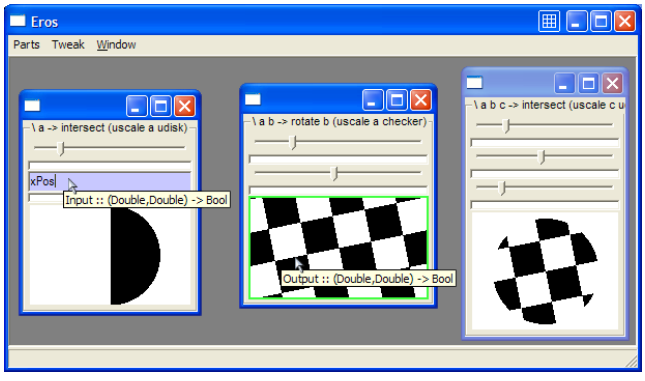

Figure 7. Programming in Eros.

Coupled with code, directly reified objects can provide context for many live programming features. For conversational programming [18], the world of objects can be selected and manipulated as part of a conversation to materialize abstract code. More generically, live programming is often explored and demoed in the context of 2D graphics, i.e. **the 2D box**, given that immediate feedback on such objects is easy to correlate with code.

Direct manipulation has many drawbacks, including:

- Although some abstractions can be re-inserted into the code as other objects (e.g. alignment guides), in general, most lack straightforward reification into displayable manipulatable objects (e.g. anything not visual).
- Structures that are deeply repetitive and intricate are difficult to build directly; e.g. consider 2D shapes, which at first glance are very amenable to direct manipulation, but when intricate patterns are desired with many shapes organized in complex topologies, abstract code becomes much more usable.

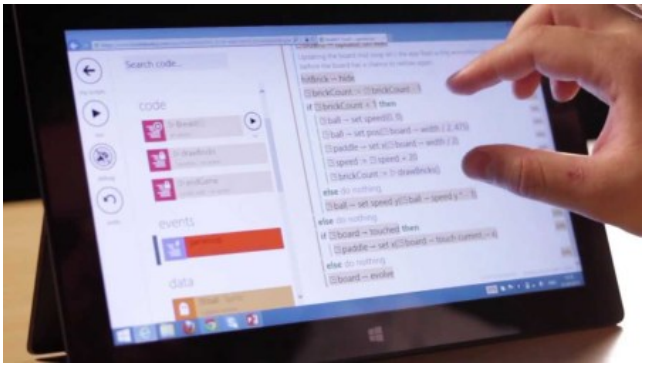

Figure 8. Programming in TouchDevelop [23].

- Our brains are more flexible and capable than any direct model the computer can currently manifest, so our thinking can often be more easily represented as abstract code.

On the last point, Hutchins et al. comment [8]:

> A more fundamental problem with direct manipulation interfaces arises from the fact that much of the appeal and power of this form of interface comes from its ability to directly support the way we normally think about a domain. A direct manipulation interface amplifies our knowledge of the domain and allows us to think in the familiar terms of the application domain rather than in those of the medium of computation. But if we restrict ourselves to only building an interface that allows us to do things we can already do and to think in ways we already think, we will miss the most exciting potential of new technology: to provide new ways to think of and to interact with a domain.

Another similar problem is simply that the "literal" metaphors that are chosen to make computation accessible can act against many useful "magical" behaviors; Smith discusses this using buttons in an Alternate Reality Kit (ARK) based on physical metaphors as an example [22]:

> Buttons have the message they send stamped on the surface - if the device does not understand the button's message, the button will fall right through the object. If the button's message is meaningful, it will stick to the surface of the object. An invisible connection is established automatically, and the button is immediately functional. Furthermore, buttons can be created that cause non-physical effects such as doubling an object's size and mass, or causing the object to vanish. Features like these are called magical because they enable the user to do powerful things that are outside of the possibilities of the metaphor.

In similar ways, general-purpose abstract code allows us to think freely where our programs are limited only by our imagination and our ability to reason abstractly over the many facets of a system.

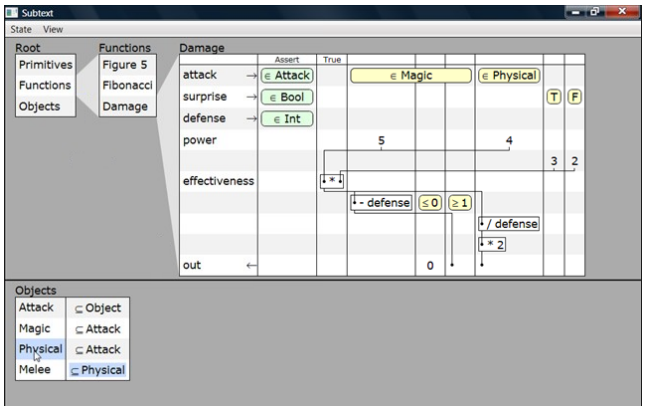

Figure 9. Subtext schematic tables.

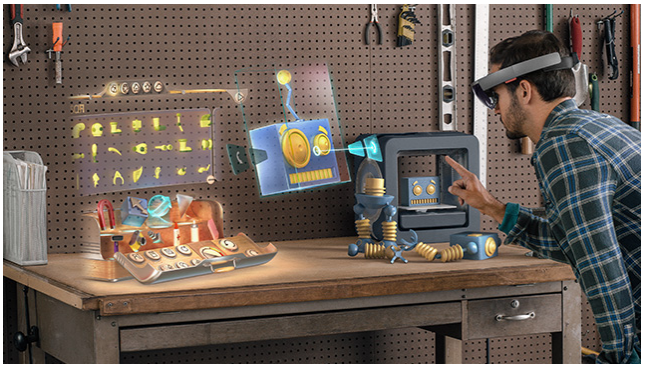

Figure 10. Hololens and creating.

Beyond Keyboards and Text

One of the requirements of direct manipulation are "rapid, reversible, and incremental" actions on objects being manipulated, which forms the core of live programming's water hose. Actions can be made quickly (rapid), unmade quickly if feedback about the action indicates they are wrong (reversible), and many small actions should compound together for greater effects (incremental). A value being scrubbed can be changed quickly, can be set back to its original value quickly, and large changes are simply made with a long continuous motion. Likewise, when two shapes come in alignment with each other, an alignment guide can be shown that the user can then "snap on" to keep the shapes in alignment as they move; but they can also "snap off" the guide if the behavior is no longer desired; while multiple guides can be added to provide for more complex behavior. Beyond direct manipulation, similar kinds of actions can be supported in abstract graphical code; e.g. VVVV [16] is based on data flow blocks (patches) that are wired together by simple actions, and whose wires can be cut quickly when not needed. Likewise, Scratch [11] is based on procedural blocks whose arguments are "snapped in" from other blocks or new ones via simple drag and drop actions.

As these rapid actions are either atomic or continuous, which do not benefit from keyboard capabilities, and can even be hindered by it–manipulating a slider with arrow keys is very annoying! Even in other cases such as code completion, as the computer helps out more, the keyboard becomes more of a hindrance to productivity. Consider programming driven by ubiquitous code completion: the mouse or touch (Figure 8) can be more efficient in making list selections [13, 14]. What the keyboard is good at, typing out abstractions in the support of free form unaugmented thinking, becomes less important as the programming environment becomes more live and direct.

As input moves away from the keyboard, it makes sense to explore graphical notation (if not direct) that provides more affordance. Conversational programming [18] depends on a graphically rich environment to surface conversation topics customized for the program being written. Likewise, Subtext supports schematic tables in [5] that provide for the direct manipulation of conditional logic (Figure 9). Schematic tables demonstrate how graphical notations can become very worthwhile when combined with live programming experiences. Finally, even the futuristic holographic interfaces of the movies are becoming feasible (e.g. Hololens in Figure 10), providing even more possibilities for future live programming environments.

Feedback Sometimes Considered Harmful

Feedback can be distracting when it does not help the programmer solve the problem at hand. Types are the classic example of potentially distracting feedback in that they require programmers to think ahead (adding them in a certain order to suppress compiler type errors), are necessarily conservative, and can introduce "noise" into the source code. As another example, structured program editors likewise interfere with the programmer's flow of activity, locking them into a series of correct edits, while visual languages often lead to literal spaghetti code that prevents realistic scaling to large problems. Live feedback has analogous problems: the program must be written in a certain order to keep it "live," live feedback often hides problem with generality, and rendering the value of a large data structure can easily overwhelm the programmer with unnecessary details.

Reifying the "wrong" model in the computer can actually make many tasks much more difficult or even impossible to do. Direct manipulation limits how programmers can solve a problem, or even if they can approach it at all, while abstract source code liberates the programmer allowing them to use their own mental models, abstractions, and reasoning. Overall, this leads to a situation where augmented programming tools are seen as beginner crutches, whose own program reasoning skills are undeveloped, and therefore assistance is seen as empowering rather than restrictive. Even as a beginner aide, the feedback does not necessarily train new programmers to perform without it; i.e. they cannot graduate to "real programming." For live programming to succeed, it must augment programming performance without being dis-

tracting or otherwise restricting what the programmer can do, and it must be adequate for both beginners and experts.

Ways Forward

This position paper concludes by discussing ways on how live programming can move forward and realize its potential.

Program with Examples

Programming by example promises to free the programmer from writing code at all by using numerous examples to synthesize a general solution automatically. Unfortunately synthesis is quite limited, and anyways numerous examples are often unavailable. However, one (or a few) example can provide a programmer with enough context to guide the programmer into abstracting the example into a general solution. Such an approach was first explored in Pygmalion [20, 21], the first iconic language with a specific focus on creativity. In Pygmalion, the screen images always contain concrete examples of the program's data, which eliminates an entire class of errors due to abstraction. This is a good basis for live programming, which should provide capabilities to form examples as well as progressively and interactively generalize the example into abstract code.

Water Hose

Although the water hose–continuous small change in code leading to continuous small change in output–is a nice ideal for live programming, it cannot be realized in practice. Still, a live programming experience should provide continuous aiming capabilities where they make sense in the form of scrubbing or similar mechanisms; e.g. to form examples or make choices on abstraction.

Create by Reacting

"Multiple choice" allows programmers to react to possibilities rather than go through the mental effort of imagining what could be from a blank canvas. Features that realize this, like code completion, are hardly unique to live programming, but can be boosted by execution context as well as being oriented to programming with examples; e.g. by showing what procedures can compute a value already known in the example from a known argument. By providing likely abstractions in a palette ready to use, the environment can also inspire programmers with what to do next, further reducing programming's mental burden.

Be Conversational

A live example can be manipulated to indicate the programmer's intent in generalizing it. For example, two diagram boxes in an example can be aligned manually by the programmer, indicating that continuous alignment might be a generalization to make in the code. By aligning the boxes, the computer can provide an affordance of permanent alignment to the programmer (e.g. as a guideline), giving them the choice to activate that abstraction.

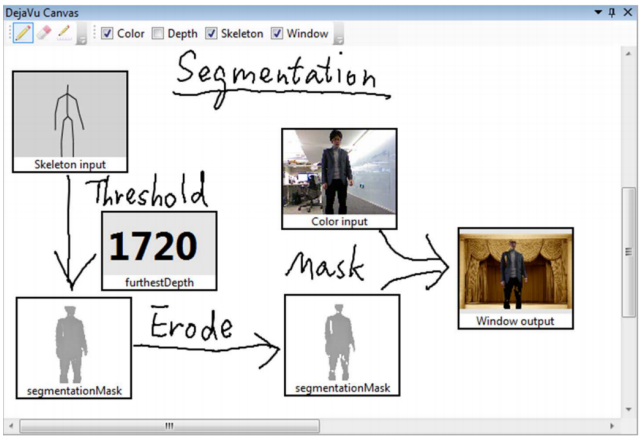

Figure 11. Canvas in DejaVu.

Basically, an example can be full of spatial coincidences that can be recognized and, at the programmer's discretion, converted into generalizations. This can work even for non-visual domains as long as they are somehow projected graphically. Consider a rendered "hello world" string rendered in an example with an arch that connects the 'h' to the space following the first 'o.' The meaning of that arc could simply be nothing, or it could indicate the range of text "hello", and even more specifically, could indicate a range of text that stops at spaces, or just includes letters.

Live Canvas

Because abstract code has a very high ceiling and is still quite usable even if unaugmented by live feedback, it is difficult to replace. Instead, just like the debugger co-exists with the code editor, abstract code should co-exist with live execution feedback. Likewise, for feedback to not be distracting, it should be under the programmer's control, showing only what the programmer wants to include in what is basically a "reified mental model." Besides exploring timelines, DejaVu [10] also explored the use of a canvas to organize live feedback according to programmer preference (Figure 11); this idea is also explored for image processing in VisionSketch [9].

Expanding on DejaVu's canvas, a more complete canvas for live programming can also be subject to manipulation, allowing code to be added to the program via techniques such as bi-directional projectional editing [26]. Such a view must generally include abstraction as well as live values, though how abstraction is represented can be more direct; e.g. through lines or compasses (as in Subtext [4]) that represent propagated values rather than through abstract names. More to the point, it should be possible to create abstractions in this view, especially through generalizing over live execution values.

References

[1] G. Bracha. Debug mode is the only mode. *Room 101*, 2012. URL http://gbracha.blogspot.com/2012/11/debug-mode-is-only-mode.html.

[2] R. B. Cattell. *Abilities: Their structure, growth, and action*. Houghton Mifflin, 1971.

[3] J. Edwards. Example centric programming. In *Proc. of OOPSLA*, pages 84–91, Dec. 2004.

[4] J. Edwards. Subtext: uncovering the simplicity of programming. In *Proc. of OOPSLA Onward!*, pages 505–518, 2005.

[5] J. Edwards. No ifs, ands, or buts: Uncovering the simplicity of conditionals. In *Proc. of OOPSLA*, pages 639–658, 2007.

[6] C. M. Elliott. Tangible functional programming. In *Proc. of ICFP*, pages 59–70, 2007.

[7] C. M. Hancock. *Real-time programming and the big ideas of computational literacy*. PhD thesis, MIT, 2003.

[8] E. L. Hutchins, J. D. Hollan, and D. A. Norman. Direct manipulation interfaces. *HCI*, 1(4):311–338, Dec. 1985.

[9] J. Kato and T. Igarashi. Visionsketch: Integrated support for example-centric programming of image processing applications. In *Proc. of GIC*, pages 115–122, 2014.

[10] J. Kato, S. McDirmid, and X. Cao. Dejavu: Integrated support for developing interactive camera-based programs. In *Proc. of UIST*, pages 189–196, 2012.

[11] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch programming language and environment. *TOCE*, 10(4):16, 2010.

[12] J. H. Maloney and R. B. Smith. Directness and liveness in the Morphic user interface construction environment. In *Proc. of UIST*, pages 21–28, nov 1995.

[13] S. McDirmid. Coding at the speed of touch. In *Proc. of SPLASH Onward*, pages 61–76, October 2011.

[14] S. McDirmid. Escaping the maze of twisty classes. In *Proc. of SPLASH Onward!*, pages 127–138, Oct. 2012.

[15] S. McDirmid. Usable live programming. In *Proc. of SPLASH Onward!*, Oct. 2013.

[16] Meso group. VVVV - a multipurpose toolkit, 2009. URL http://www.vvvv.org.

[17] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proc. of OOPSLA*, pages 535–552, 2007.

[18] A. Repenning. Conversational programming: Exploring interactive program analysis. In *Proc. of Onward!*, pages 63–74, 2013.

[19] B. Shneiderman. Direct manipulation. a step beyond programming languages. *IEEE Transactions on Computers*, 16(8):57–69, August 1983.

[20] D. C. Smith. *Pygmalion: A Creative Programming Environment.* PhD thesis, Stanford University, 1975.

[21] D. C. Smith. Pygmalion: An executable electronic blackboard. In A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors, *Watch What I Do*, pages 19–48. MIT Press, 1993.

[22] R. B. Smith. Experiences with the Alternate Reality Kit: an example of the tension between literalism and magic. *SIGCHI*, pages 61–67, May 1986.

[23] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *Proc. of SPLASH Onward!*, pages 49–60, 2011.

[24] B. Victor. Scrubbing calculator. *Worry Dream*, 2011. URL http://worrydream.com/ScrubbingCalculator.

[25] B. Victor. Learnable programming: designing a programming system for understanding programs. *Worry Dream*, 2012. URL http://worrydream.com/LearnableProgramming/.

[26] M. Völter, J. Siegmund, T. Berger, and B. Kolb. Towards user-friendly projectional editors. In *Proc. of SLE*, pages 41–61, 2014.

[27] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *Proc. of CHI*, pages 258–265, 1997.