Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming

Sorin Lerner
University of California, San Diego
lerner@cs.ucsd.edu

ABSTRACT

Live programming is a regime in which the programming environment provides continual feedback, most often in the form of runtime values. In this paper, we present Projection Boxes, a novel visualization technique for displaying runtime values of programs. The key idea behind projection boxes is to start with a full semantics of the program, and then use *projections* to pick a subset of the semantics to display. By varying the projection used, projection boxes can encode both previously known visualization techniques, and also new ones. As such, projection boxes provide an expressive and configurable framework for displaying runtime information. Through a user study we demonstrate that (1) users find projection boxes and their configurability useful (2) users are not distracted by the always-on visualization (3) a key driving force behind the need for a configurable visualization for live programming lies with the wide variation in programmer preferences.

Author Keywords

Live programming; Programming environment; Program visualization; Debugging.

CCS Concepts

•**Human-centered computing** → **Graphical user interfaces;**

INTRODUCTION

Live programming is a coding regime in which immediate feedback is provided to the programmer each time the program is modified. One line of the research in this space focuses on the performing arts, where artists use specially designed live programming environments to create audio/visual pieces, sometimes in live performances.

In this paper, however, our focus will be on live programming environments for *general purpose programming languages*. In this setting, live programming environments provide a way to visualize the runtime values of the program each time the program is modified. Examples of such systems include Bret Victor's visualizations [19, 20], Omnicode [8], Alfie [1], Seymour [10, 9], Hazel [15] and the Babylonian editor [16].

CHI'20, April 25–30, 2020, Honolulu, HI, USA

ISBN 978-1-4503-6708-0/20/04…$15.00

DOI: https://doi.org/10.1145/3313831.3376494

One challenge in live programming environments for general purpose languages is *information overload*. Indeed, displaying updated runtime values at virtually each and every change can be intrusive, overwhelming and/or distracting, which could ultimately offset some of the benefits of the visualization.

In the context of this challenge, we present a new visualization technique called *projection boxes* for displaying values of programs. The key idea of projection boxes is to start with the full semantics of the program, and then *project* this full semantics into a subset of values that are displayed. While this idea is simple, it has significant ramifications, which can be summarized into two main points.

First, the expressiveness of projection boxes is quite surprising given their simplicity. By using different kinds of projections, we can achieve many different representations, including previously known ones, and also new ones. As such, projection boxes not only provide a framework for understanding and connecting several live visualization techniques, but it also inspired us to discover new ones.

Second, projection boxes provide a significant amount of versatility for the programmer, and as such provide a powerful tool for dealing with information overload. Indeed, different projections display varying amounts of information, from no-values-ever to all-values-all-the-time, and every point in between. This spectrum allows each programmer to pick different visualizations at different points in a programming task, thus allowing personalized on-the-fly (i.e.: while coding) control in dealing with the information overload problem. Since different programmers have different preferences (based on a variety of factors, including their background, expertise, familiarity with the language, coding style, etc.), we found that this kind of personalized on-the-fly configurability is essential in preventing programmers from being overwhelmed or distracted by always-on visualizations. A key observation of this paper is that differences in human preferences should be an important driver for designing always-on visualizations.

In summary, our contributions are as follows:

- We present a novel always-on visualization technique called *projection boxes*. We show how projection boxes are versatile enough to encode many different visualizations, including previously known ones, and also new ones.
- We present VERSABOX, an implementation of projection boxes for the Python programming language. VERSABOX provides an interface for programmers to quickly customize projection boxes on-the-fly (i.e.: while coding), thus allow-

ing programmers to harness the versatility of projection boxes for managing information overload.

- We present the results of a user study with VERSABOX showing that: (1) programmers find projection boxes useful (2) programmers find that projection boxes are *not* intrusive or distracting (3) different programmers pick different visualizations at different points.

RELATED WORK

There is a long line of research on live programming environments for general purpose programming, including: the seminal work of Hancock [5] that introduces many of the important concepts in live programming; essays categorizing the different kinds of liveness [18, 20]; live programming environments for various languages, for example Python [4, 8], Java [3], Javascript [16, 1], Lisp [2] and ML-like languages [15]; studies demonstrating the benefits of live programming [21, 11]; work on how to run/instrument programs for live programming [15, 17]; and work on live editing the output of a program through direct manipulation [6, 13, 7].

Broadly speaking, our work distinguishes itself in two primary ways: (1) our work presents a visualization framework that is flexible enough to encode both previously known visualization paradigms, and also new ones (2) our visualization framework supports quick non-intrusive on-the-fly reconfigurability, which we show to be an effective way to handle information overload in always-on visualizations. We now discuss in more detail the most closely related work.

Perhaps most closely related to our work is (what we will call) the Victor Visualization [19], introduced by Bret Victor in 2012, and later refined in various systems, like Alfie [1] and Seymour [9, 10]. The Victor Visualization shows, next to each assignment, a row of all the values produced by that assignment over time. If there is more than one value to display (for example because the statement is inside a loop), then the values from the same loop iteration are vertically aligned. We will show in this paper that our Projection Boxes can be configured to re-create the Victor Visualization, but with additional benefits (which will be described later in the paper). Most importantly, our Projection Boxes are general enough that they can implement several visualizations other than the Victor Visualization, and allow the programmer to quickly switch between them on-the-fly, while coding.

Another closely related system is Omnicode [8], a novice-oriented environment that makes the entire history of program execution available to the programmer. Omnicode supports (1) scatterplot visualizations (2) a full heap-as-a-graph visualization when focusing on a particular execution step and (3) swiping over the code to filter data based on the selected statements. Although our implementation does not support scatterplots or heap graphs, at a conceptual level projection boxes could be flexible enough to encode these. The strength of the Omnicode work lies in pushing the idea of "displaying all values all the time" to the extreme. In the context of Omnicode, the novelty of our work is in providing an expressive on-the-fly reconfigurable framework, which essentially gives a highly tunable slider from "no information at all" to "displaying all values all the time".

Our work is also related to the recently published Babylonian editor [16, 17]. This editor supports probes on variables, and sliders on loops to focus on particular loop iterations. Our Projection Boxes can encode some of the Babylonian visualizations (like single variable probes and focusing on particular loop iterations), but not others (like seeing a live view of the canvas for drawing programs). There are also paradigms we support that the Babylonian editor does not, like non-local probes, and full program state probes (described later).

The YinYang system [14] is a live environment with (1) probes displaying one value at a time and (2) a live trace pane on the right. Our work supports several additional paradigms, including probes that show the values of all iterations at once and probes that show more than one variable at time.

Another closely related work is the Theseus editor [12], which provides live information about method call-counts and an interactive pane to visualize the calling structure. Theseus does not however provide program state information at each line, the way projection boxes do.

PROJECTION BOXES

The core contribution of this paper is a versatile always-on visualization technique for runtime values called *projection boxes*. The key idea of projection boxes is to start with a full semantics of the program, and then *project* this full semantics into a subset of values that are displayed. While this idea is simple, its expressiveness is surprising. By using different kinds of projections, we can achieve different representations, including previously known ones, and also new ones. The programmer has the flexibility to pick different visualizations at different points in a programming task, and can even use different visualizations for different parts of the program.

Full Semantics

The starting point of our visualization is the full semantics of the program. This semantics is meant to capture the most detailed view of the execution of the program. We use a well-known complete semantics, the *state collecting semantics* of a program: at each line in the program, we compute the set of all program states that can occur at that line. For simplicity we will consider each program state to be a mapping from variable names to values. In reality, the program state also has the notion of a heap, but we assume for now that variables pointing into the heap will be mapped to a string representation of the heap data structure (for example a variable pointing to a list will be mapped to a string representation of that list).

Basic Box

Figure 1 shows the basic projection box visualization. There is one projection box for each line in the program. Each box is a table of values, with each column being a variable name, and each row being a runtime state. The "#" column shows iteration counts for loops. The boxes "float" to the right of the program, and are connected with a straight line to the place in the code that they are displaying values for. This ability of boxes to float to the side means that there is no need for an

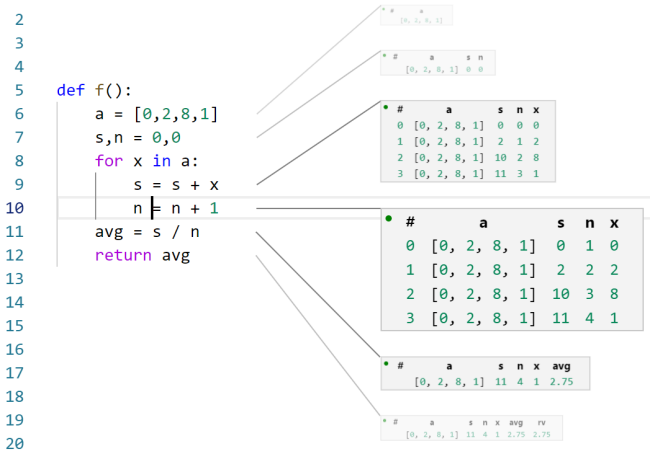

Figure 1: Basic Projection Boxes

extra pane to display values. This leaves program formatting completely unchanged, leading to a less intrusive visualization. Finally, the little green dot at the top left of each box indicates that the box is up-to-date. If the program changes in a way that it cannot be run (because of syntax or run-time errors), this dot becomes orange, and eventually red, to indicate that the information is out-of-date.

If a particular statement is not executed in a given loop iteration (typically because of a branch), we insert empty rows for those iterations. In preliminary experiments, we found that without this feature it was hard to understand how values flowed through branches.

Information Overload

The biggest drawback of projection boxes as described so far is that they display a lot of data, which can lead to information overload. To alleviate this problem, we start with three visual techniques that focus attention at the current cursor position. These techniques are shown in Figure 1, where line 10 is the current cursor position. While these three techniques alleviate the problem, we will shortly see that they will not be the ultimate answer. First, we align the box at the current cursor position with the cursor, and push all other boxes away, above and below the cursor. Second, we use a fish-eye effect that shrinks boxes that are farther away from the cursor: the farther a box is from the cursor, the more it shrinks. Third, we use a transparency effect: the farther a box is from the cursor, the more transparent it is. The result of these three techniques is that, as the programmer moves the cursor around, the size of the boxes change fluidly so that the box at the cursor remains fully visible as a focal point, and other boxes further away attract less attention, while still remaining visible.

Projections

Even with the above three techniques, the amount of data can still be overwhelming. To address this problem, we introduce a much more potent mechanism to address visual clutter: *projections*. A projection in our context can do two things:

1. *A projection can pick a subset of the values to display*. For example, a projection could choose to only display boxes for certain program locations and not others; or within a given box, a projection could choose to only display some variables and not others; or at an even finer granularity, a projection might choose to display only some of the values for a given variable (for example to focus on a particular loop iteration).

2. *A projection can switch the display order between column order and row order*. The boxes we've seen so far (Figure 1) display each variable in a column (column order). Alternatively, one can transpose the table and show one variable on each row (row order, which we will see later). Switching between the two orders can be beneficial for space management. For example, if there are many loop iterations, but few variables, then column order will lead to very tall skinny boxes, and it will be hard to display many of these on top of each other, and so in this case row order might lead to better space management.

Versatility of Projection Boxes

Although the idea of projections is simple, the expressiveness and generality it entails is surprising.

Indeed, projection boxes are able to encode many traditional visualization techniques from live programming, including: (1) Single variable probe at the point where a variable is read or written (2) Probe of the return value (3) Filtering to show only certain loop iterations (4) Filtering based on a particular input (5) Filtering to show the runtime values associated with only a particular input and (6) The Victor visualization from 2012 [19].

In addition, projection boxes enable several forms of visualizations that have not been explicitly supported previously, including: (1) Single variable probes at locations other than a read or a write (2) Multi-variable probes that display multiple variables at the same program location (3) Full table of all values at each program point and (4) A probe that follows the cursor and displays the runtime values at the cursor position.

To illustrate the power of projection boxes, we show how to encode the *Victor visualization*, a visualization introduced in a 2012 talk by Bret Victor [19], and also implemented in Seymour [10, 9] and Alfie [1]. Figure 2(left) shows a screenshot from the 2012 talk by Bret Victor.

At first sight, it might seem that the Victor visualization is quite different from our projection boxes. However, quite surprisingly, projection boxes are general enough to implement the Victor visualization. More specifically, to implement the Victor visualization we use the following projection:

- *Row vs Column*: Display each variable as a row
- *Values to Display*: Variable(s) modified at each line

Figure 2(right) shows the resulting visualization using projection boxes. To see the boxes, keep in mind that each projection box has a little green dot at the top left. Note that compared to Figure 1 the projection boxes in Figure 2(right) do not have a border, don't have space between them, and have bars to separate columns. These are simple minor visual adjustments.

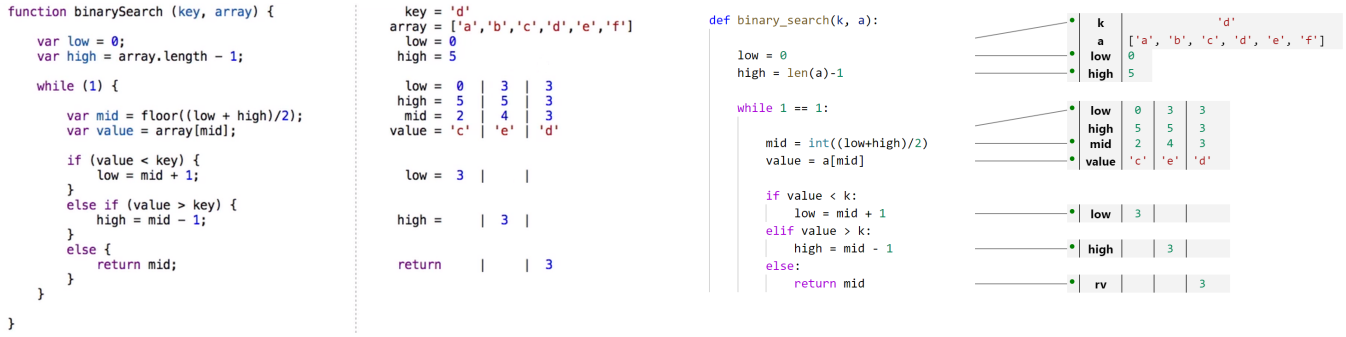

Figure 2: Victor Visualization. Left: original from 2012 talk. Right: using our projection boxes

Not only can projection boxes encode the Victor visualization, but the version based on projection boxes has two advantages over the traditional Victor visualization: (1) whereas the Victor visualization adds empty code lines to make sure values are aligned with the statement that produced them, projection boxes, by virtue of "floating", leave code spacing unchanged (2) whereas seeing the relation between two variables in the Victor visualization would require looking at lines in the code that can be arbitrarily far away from each other, projection boxes can be configured to show multiple variables at the same code line, one above the other, thus making it easier to see relationships between variables at a given program location.

FORMATIVE STUDY

We have seen that projections offer a powerful mechanism for customizing the visualization of runtime values. However, we have not yet addressed the issue of how users select/define projections. To understand the design space, we conducted a formative study in which 4 subjects used and provided feedback on a preliminary version of VERSABOX, our implementation of projection boxes. This preliminary version displayed a full table of values at each line, and had menu controls to add/remove projection boxes and variables in each box.

Formative Study and the Need to Customize Projections

In our formative study, we found that *differences in human preferences* led to a particularly strong need for different projections. Indeed, some programmers like to check their code after each line is written, whereas others like to write many lines before pausing to check their code; some programmers like to write code linearly, whereas others jump around when writing code. All of these differences affect the kind and amount of feedback that programmers want.

Furthermore, although live programming tries to blur the line between coding and debugging, we still observed a fluctuation between a more coding-centric mode, and a more debugging-centric mode. This fluctuation further accentuated the need for different levels of visualization. Programmers generally wanted less information when in a code-centric mode, and more information when in a debugging-centric mode.

Finally, we found that a programmer's experience with the programming language had a strong influence on the amount of immediate feedback they want. Programmers who were familiar with a language generally wanted less information, since they did not need to check every single step.

Formative Study and the UI for Customizing Projections

Our formative study also provided several insights on what UI to use for customizing projections. First, programmers unanimously stated that using the mouse to adjust projections was a major hurdle, and that keyboard shortcuts would have led them to adjust projections more often while writing code. Second, a recurrent feedback was that a simple set of presets for view modes would be really useful, so that programmers new to projection boxes do not get overwhelmed with the customizability options. Finally, we found that one programmer, while finding the visualization useful, wanted to turn it off some of the time. This pointed us what we will call a "stealth mode" where all boxes are hidden (projected out).

IMPLEMENTATION OF PROJECTION BOXES

Following the feedback from our formative study, we implemented projection boxes in a tool called VERSABOX. VERSABOX is built on top of Visual Studio Code editor, and works for the Python language. Each time the programmer makes a change to the code, VERSABOX runs the each function on unit tests to collect all the required data to display in the projection boxes.

There are many systems challenges in making this kind of "run-always" approach practical, including efficiently running code, supporting I/O, and supporting infinite-running programs. While these systems challenges are not fully solved yet, in this paper we explicitly *do not address these*. Instead our contribution is on the human-computer interaction research question of how projection boxes work and what they enable.

Using the insights of our formative study, VERSABOX offers 4 preset *view modes*, which the user can switch between with *keyboard shortcuts* (the first two views below use column order):

- *Full View*: Shows all boxes and all variables, as in Figure 1.
- *Summary View*: Shows box at cursor and return statement.
- *Row View*: Victor visualization, as shown in Figure 2(right).
- *Stealth View*: Shows no boxes.

Questions	avg
Q1: *VersaBox helped me write correct code*	4.7
Q2: *VersaBox was easy to use*	4.8
Q3: *Visualization boxes in VersaBox distracted me*	1.2
Q4: *Configuring displayed variables was useful*	4.4
Q5: *My favorite view:* [*Full* | *Summary* | *Row* | *Stealth*]	–
Q6: *Different views are best suited for different tasks*	3.8
Q7: *I would use VersaBox again*	4.6

Figure 3: Questions in survey along with average scores. All questions except Q5 are on a 5 point Likert scale with 1 being "Disagree" and 5 being "Agree". Q5 is a multiple choice question, whose answer distribution was (out of 10 subjects): Full (3), Summary (1), Row (6), Stealth (0).

Programmers can also configure which variables are displayed in projection boxes. A keyboard shortcut brings up a one-line textbox that hovers over the current line. This textbox accepts commands in the following syntax: `add|del|keep [@all] VarRegExp`. For example, "`del a`" would remove variable "`a`" from the box at the current line, and "`del@all a`" would remove "`a`" from all boxes. Wildcards and regular expressions can be used in variable names. The "`keep`" command keeps only the variables that match the given expression, so that "`keep` E" is equivalent to "`del *`" followed by "`add E`".

EXPERIMENTAL SETUP

One of the goals of projection boxes is to manage information overload, which is one of the biggest challenges with always-on visualizations, especially ones that are meant to be used *while* coding (not just debugging). As such, we want to understand if projection boxes provide a good balance between providing useful information, but without being distracting. This leads us to three main questions:

- **R1: Utility.** Do users find projection boxes helpful?
- **R2: Distraction.** How distracting are projection boxes?
- **R3: Customizability.** Do users find the customizability afforded by projection boxes useful? Why or why not? And if so, how?

To answer these questions, we ran a lab study with 10 programmers (2 women, 8 men), whose experience ranged from medium to expert. Programming experience ranged from 3 to 9 years. Python experience ranged from 1 to 5 years. We recorded the sessions (1.5 to 2.25 hours) to analyze them later.

We first taught subjects how to use the tool (15 minutes). We then asked subjects to program 6 tasks: (1) compute difference between *max* and *min* of a list (2) compute difference between *mean* and *median* of a list (3) compute *mean* of the first quartile of a list (4) functionally insert in a sorted list perserving sortedness (5) destructively insert in a sorted list perserving sortedness (6) parse a string containing a set of records.

We gave the subjects a handful of test cases and asked them to continue until their code worked on all test cases. After the study, we asked users to fill out a survey, whose questions are show in Figure 3. Finally we conducted post-study interviews.

EXPERIMENTAL RESULTS

R1: Utility

Users overwhelmingly found VERSABOX helpful (Q1 avg of 4.7/5) and easy to use (Q2 avg of 4.8/5). We observed that subjects found VERSABOX useful in four ways: (1) Finding and fixing mistakes right when they are introduced (2) Guiding code writing, where a common workflow would be to look at the current variables to guide how the next statement should be written (3) Testing the entire code after it is written, to check not only that it produces the correct result, but also that the control flow works as expected (4) Discovering and clarifying how certain operations work in Python, such as `sort`, `sorted`, `insert` `split`, `strip`, `stripl` and `stripr`.

The fourth usage mode above is especially interesting. Indeed, by trying out statements and looking at the result in projection boxes, programmers can use VERSABOX as an nonintrusive interpreter right in the editor, without having to switch to another tool. This interpreter is particularly useful because it runs commands in the context of the code surrounding the statement being written. This interpreter is also *not* a special case: it just falls out naturally from the visualization.

The above four uses all come together for various users at different frequencies. We often observed this during the study, and S8 mentioned it in particular during the interview: he mentioned that the granularity at which he would look at the boxes would depend on how confident he was about the code he is writing. We believe that the ability of VERSABOX to support this mode of operation is one of the key factors that make it useful:

> *Takeaway 1: If the level of distraction is minimized, programmers can benefit from always-on visualizations at different granularities, taking in the information on-demand, only when they find it useful.*

R2: Level of distraction

Subjects found that projection boxes did not distract from programming (avg of 1.2/5 to Q3 "The visualization boxes in VersaBox distracted me", where 1 is "Disagree"). By observing subjects, we noticed that one of the keys to achieving this level of non-intrusiveness is that subjects were able to configure the visualization on-the-fly, as needed, to match the level of detail they wanted:

> *Takeaway 2: On-the-fly configurability helps reduce distraction by enabling the programmer to configure the amount of information, and as such also mitigates information overload in always-on code visualizations*

R3: Customizability

Customizing Variables

Subjects generally found that customizing the set of displayed variables was useful (Q4 avg 4.4/5). By observing subjects, we found that subjects either removed variables when there was too much information in the Full View (for which the default is that each box shows all variables), or added variables when there was not enough information in the Row View (for which the default is that each box displays just the modified variable).

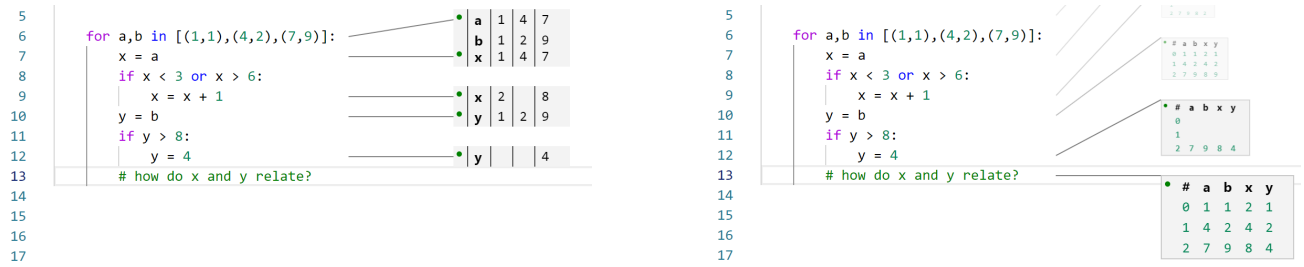

Figure 4: The relation between `x` and `y` on line 13 is very hard to see in Row View (left), but easy to see in Full View (right). Conversely, seeing which values are generated at each line is easy to see in Row View (left), but hard to see in Full View (right).

Customizing View Modes

The answers to Q5, "Preferred view mode", show that no one view was preferred: Row View was picked by 6 subjects, Full View by 3, Summary View by 1, and Stealth View by none. This already shows how view customization is useful.

The answers to Q6 further reinforce this point. Recall that Q6 is "Different views are best suited for different tasks", and that the scale is 1 to 5, where 1 means "Disagree", and 5 means "Agree". 6 out of the 10 subjects rated this statement 4 or 5, which we will categorize as generally agreeing. The remaining 4 subjects rated this statement 2 or 3, which we will categorize as generally disagreeing. Interestingly, the 4 disagreeing subjects (who essentially thought just one view is good enough for all tasks) did not all have the same preferred view: two preferred the Full View, one preferred the Summary View, and one preferred the Row View. This highlights even more the need for customizability: even among those who prefer to use just one view, they don't all prefer the same view.

All of the above differences in view preferences ultimately stem from an inevitable tradeoff between Row View (i.e.: the Victor visualization) and Full View. Row View makes it easier to focus on the changes happening at a given line, whereas the Full View makes it easier to understand the full program state and relationships between variables at a given line. For example, consider the code in Figure 4, shown in Row View (left) and Full View (right). In Row View, it is extremely difficult to see the relation between `x` and `y` on line 13 inside the loop (in fact, `x` is two times `y`). In contrast, in the Full View, this is immediately visible. We observed that subjects coped with this tradeoff in three ways: (1) primarily using Row View, and adding variables to see relations between variables (2) primarily using Full View, and removing variables to more clearly see changes at a particular statement (3) switching on-the-fly between Row View and Full View.

Takeaways about Customizability

All of the above results leads us to the following takeaway:

> *Takeaway 3: Different programmers have significantly different preferences for always-on code visualizations. Consequently, customizability based on programmer preferences should be a key driving force behind the design of always-on code visualizations.*

Another phenomenon we observed concerns mouse usage:

> *Takeaway 4: When configuring always-on visualizations on-the-fly (in the middle of coding), programmers shy away from using the mouse, even for those programmers who use the mouse otherwise in their coding style*

We believe this is partly because, while in the middle of *writing* code (i.e.: typing characters), using a mouse is cognitively disruptive. If we want a visualization to be customizable on-the-fly, at a fine-level of granularity, our experience shows that keyboard shortcuts are important.

LIMITATIONS AND FUTURE WORK

There are several directions for future work that would improve projection boxes and VERSABOX. One limitation is that our lab study was a small-scale experiment. Doing larger studies in the wild would give us a broader understanding of the tradeoffs and help make our conclusions stronger.

VERSABOX has currently been used on unit test inputs, which are relatively small. Visualizing larger data sets will require incorporating various additional mechanisms, such as previewing only part of the data, or using plots to see aggregate information. These ideas fit nicely in the framework of projection boxes, but care will need to be taken to build an interface for configuring all of these new forms of visualizations.

Code execution in VERSABOX can also be improved. I/O and reproducibility are interesting challenges that will come up, for example, if the code being written is a server. Still, it's important to realize that even in a server, there are many self-contained functions that do not perform I/O and just process data, functions which VERSABOX can already handle.

Another future work direction is adapting these ideas to other programming paradigms, for example functional or constraint-based. The idea of projection boxes could still apply, but additional challenges might arise, for example where to place boxes, or how to visualize the state of a constraint solver.

CONCLUSION

We presented an always-on reconfigurable visualization framework called projection boxes. Projection boxes allow programmers to quickly reconfigure the visualization on-the-fly, in the middle of coding. Through a user study, we showed that there is no single configuration that works best for all programmers or for all tasks, which motivates the need for highly customizable visualizations like projection boxes.

ACKNOWLEDGEMENTS

We would like to thank Nada Amin for suggesting that we think of our work through the lens of projections, Nishil Macwan for helping with some of the early exploratory work with Visual Studio Code, and the anonymous reviewers for their helpful comments and suggestions.

REFERENCES

[1] 2019. Alfie. https://alfie.prodo.ai/. (2019). Accessed: 2019-09-01.

[2] 2019. LightTable. http://lighttable.com/. (2019). Accessed: 2019-09-01.

[3] Benjamin Biegel, Benedikt Lesch, and Stephan Diehl. 2015. Live object exploration: Observing and manipulating behavior and state of Java objects. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 581–585. DOI: http://dx.doi.org/10.1109/ICSM.2015.7332518

[4] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 579–584. DOI: http://dx.doi.org/10.1145/2445196.2445368

[5] Christopher Michael Hancock. 2003. *Real-time Programming and the Big Ideas of Computational Literacy*. Ph.D. Dissertation. Cambridge, MA, USA. AAI0805688.

[6] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology (UIST '16)*. ACM, New York, NY, USA, 379–390. DOI: http://dx.doi.org/10.1145/2984511.2984575

[7] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual Symposium on User Interface Software and Technology (UIST '19)*. ACM, New York, NY, USA.

[8] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. ACM, New York, NY, USA, 737–745. DOI: http://dx.doi.org/10.1145/3126594.3126632

[9] Saketh Kasibatla and Alessandro Warth. 2017. Seymour: Live Programming for the Classroom. In *International Workshop on Live Programming Worskhop (LIVE 2017)*.

[10] Saketh Ram Kasibatla. 2018. *Seymour: A Live Programming Environment for the Classroom*. Master's thesis. University of California, Los Angeles.

[11] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan O. Borchers. 2014. How live coding affects developers' coding behavior. *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2014), 5–8.

[12] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing Misconceptions About Code with Always-on Programming Visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2481–2490. DOI: http://dx.doi.org/10.1145/2556288.2557409

[13] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 127 (Oct. 2018), 28 pages. DOI: http://dx.doi.org/10.1145/3276497

[14] Sean McDirmid. 2013. Usable Live Programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 53–62. DOI: http://dx.doi.org/10.1145/2509578.2509585

[15] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. DOI: http://dx.doi.org/10.1145/3290327

[16] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (2019).

[17] Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2019. Implementing Babylonian/S by Putting Examples Into Contexts: Tracing Instrumentation for Example-based Live Programming As a Use Case for Context-oriented Programming. In *Proceedings of the Workshop on Context-oriented Programming (COP '19)*. ACM, New York, NY, USA, 17–23. DOI: http://dx.doi.org/10.1145/3340671.3343358

[18] S. L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. DOI: http://dx.doi.org/10.1109/LIVE.2013.6617346

[19] Bret Victor. 2012a. Inventing on Principle. (2012). https://vimeo.com/36579366#t=18m05s

[20] Bret Victor. 2012b. Learnable Programming. (2012). http://worrydream.com/LearnableProgramming/

[21] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. 1997. Does Continuous Visual Feedback Aid Debugging in Direct-manipulation Programming Systems?. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '97)*. ACM, New York, NY, USA, 258–265. DOI: http://dx.doi.org/10.1145/258549.258721