Efficient Editing in a Tree-Oriented Projectional Editor

Tom Beckmann
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
tom.beckmann@student.hpi.de

ABSTRACT

In contrast to text editors, using projectional editors typically requires users to learn a new editing metaphor. In text-oriented projectional editors, rules may be used to give users the impression of working in a regular text editor. As such, the editing metaphor appears clear to the user, but if the implemented rules do not suffice, users may be confused why certain actions do not work. Block-oriented projectional editors, where the editing metaphors are typically clearer, often present challenges in editing efficiency, as code takes up more space and editing is perceived to be slower. We present a block-oriented projectional editor that tries to address the issues of editing efficiency and space usage. Through a reusable set of commands for tree modifications, a space-efficient AST visualization, and context-aware input we allow users to edit as fast or faster than in a text editor. We provide a first evaluation by comparing the number of keystrokes required to carry out common edit operations.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**.

KEYWORDS

Projectional Editing, Modal Editing, Editing Efficiency

ACM Reference Format:
Tom Beckmann. 2020. Efficient Editing in a Tree-Oriented Projectional Editor. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3397537.3398477

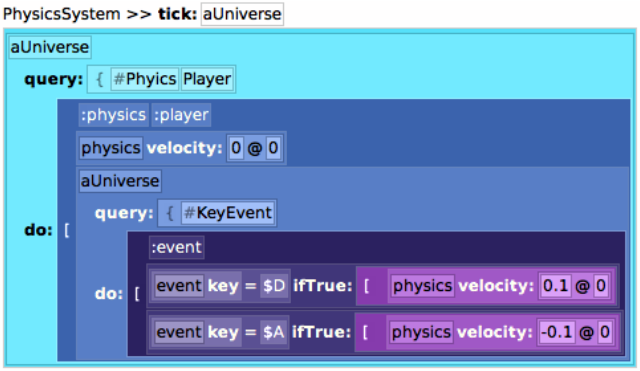

Figure 1: A screenshot of our editor.

1 INTRODUCTION

Working in projectional editors can provide an editing experience that is as fast or faster than working in text editors [1]. In contrast to text editors, the editing metaphors of projectional editors typically require robust design of a large set of editing operations to reach the expressiveness of a text editor [10]. We distinguish between text-oriented and tree-oriented projectional editors in this paper. A text-oriented editor uses similar layout and syntactical elements as a textual language, but constrains editing to maintain a valid abstract syntax tree (AST). While providing a familiar way of reading code, editing operations may as such not be possible in the way users would expect from the interface, unless special care is taken [9]. A tree-oriented projectional editor makes the structure of the AST apparent through user interface elements. While often perceived as inefficient, the editing metaphors may appear clearer to users when compared to a text-oriented projectional editor, as boundaries between elements are immediately apparent. We hypothesize that supporting users' understanding of the AST through presentation in a tree-oriented projectional editor can increase editing efficiency, as opposed to making projectional editors feel more like text editors. To test this hypothesis we present the design of a tree-oriented projectional Editor for Smalltalk [3]. The editor is available on GitHub (https://github.com/tom95/sandblocks).

2 RELATED WORK

Text-oriented projectional editors include mbeddr [8], which maps a dialect of C, or Lamdu [5], a functional, projectional language. For mbeddr and its host framework MPS specifically, a number of studies on its usability have been conducted [1, 9, 10].

Scratch is an example of a tree-oriented projectional editor [7]. Its design aims to be easy to understand for use in education, rather than edit efficiency. Dataflow visual programming languages, such as Blueprints in the Unreal Engine [2], are often domain-specific.

3 DESIGN

Our editor uses nested blocks to denote inclusion structures within the tree, see Figure 1. Colors become brighter to show deeper nesting and change to a new base color for each Smalltalk block. By

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3398477

laying out the blocks similar to text, we have similar space efficiency to textual languages, however as the nesting increases our rows get higher, such that inclusions remains clear.

Similar to the Vim editor, we use a command mode for traversing the tree and issuing commands, and allow entering a separate input mode for typing names and literals. A selection mode allows users to mark complex selections. The commands are designed to be reusable: depending on the selected node the same command may perform a semantically similar operations, such as inserting a statement in a block, or adding an element to an array. Commands mirror aspects of their functionality if modified via shift, reducing the number of distinct shortcuts to remember.

To allow for a natural editing flow, the input mode is context-aware: each node will recognize inputs that are not valid for its current context, but can be translated to a simple tree transformation. For example, typing "+" while inside a number will wrap the number in a binary expression, and typing another number while inside the binary message's selector will move the new number to the second operand.

4 EVALUATION

To arrive at a sufficient level of expressiveness, we define 25 unique, essential commands, and a total of 90 commands. This, as well as the mode-based design, will likely present a high entry barrier to new users of our editor. However, mode-based editing has been found to allow for faster editing for fixed editing operations [6], potentially balancing its downsides.

As a preliminary evaluation, we compared the number of keystrokes required in a text editor and our editor design to carry out a list of common editing operations [4], see Table 1. For the most common operations, we typically require the same number of keystrokes, plus entry to input mode. Operations that restructure the tree, such as reordering list elements or changing parentheses, however, can generally be performed faster in our editor.

5 CONCLUSION AND FUTURE WORK

The design of our editor will not perform better for simple edits in terms of number of keystrokes, but only for complex editing operations. However, rather than looking at the number of keystrokes, the total time spent performing edits may be a more representative figure, as it allows to include the overhead of switching to the mouse to perform selection and navigation. We plan to experiment with different designs to make the editor approachable and learnable, such that we can carry out controlled experiments to make stronger statements on the editor's performance.

Table 1: Edit operations and their prevalence [4] plus number of keystrokes required in a text editor and in our editor to carry out the operations. We adapted each example to Smalltalk and thus dropped prefix operations from the original list (which accounted for 1 % of edits). We used the string "subExp" where applicable as example input for comparisons and otherwise used the example used in parentheses in the operation column.

Operation	Freq.%	Text	Our	Δ
Create a name	25.37	6	7	−1
Create infix from left operand	9.60	1	1	0
Insert new statement	8.05	9	7	2
Create list	7.36	1	2	−1
Replace part of name	5.59	6	7	−1
Create complete literal	4.80	6	7	−1
Fix typo in name (change "E" to "e")	4.30	4	5	−1
Type keyword (true)	4.26	4	5	−1
Modify literal (change 4 to 7)	2.16	5	5	0
Name autocompletion (prefix sub)	2.15	4	5	−1
Insert element into list	2.07	8	7	1
Insert element into list via paste	2.07	3	1	2
Remove element	1.84	3	1	2
Comment out	1.80	2	1	1
Replace full name	1.72	7	7	0
Remove name	1.72	2	1	1
Split name	1.29	4	12	−8
Replace infix with left operand	1.20	3	1	2
Reactivate autocompletion	1.20	2	3	−1
Create annotation	1.11	3	1	2
Replace literal with expression	1.04	3	1	2
Type keyword into existing	0.96			0
Rename identifier	0.86	3	1	2
Create infix by splitting	0.75	4	12	−8
Remove entire infix	0.75	3	1	2
Create call	0.74	7	7	0
Move delimiter (include/exclude)	0.69	3	3	0
Wrap operand in infix (self subEx: 5)	0.60	14	14	0
Delete list	0.46	2	1	1
Flatten list	0.46	3	1	2
Create optional structure	0.42	n/a	n/a	0
Unwrap operand	0.30	3	1	2
Change infix to binary	0.30	2	2	0
Declare via refactor	0.24	2	2	0
Replace with param (self subExp: 5)	0.17	3	1	2
Create infix by paste	0.15	1	1	0
Replace infix with right operand	0.15	1	2	−1
Remove method call (self subExp)	0.12	3	1	2
Remove keyword call (a ifTrue: [5])	0.09	5	1	4
Replace expr via comment	0.09	3	4	−1
Sum		137	129	8
Sum weighted by frequency		4.44	4.73	−0.29

REFERENCES

[1] Thorsten Berger, Markus Völter, Hans Jensen, Taweesap Dangprasert, and Janet Siegmund. 2016. Efficiency of projectional editing: a controlled experiment. In *FSE 2016*. 763–774. https://doi.org/10.1145/2950290.2950315

[2] Epic Games. 2012. *Unreal Engine Blueprints*. https://web.archive.org/web/2019/https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html

[3] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, USA.

[4] Amy Ko, Htet Aung, and Brad Myers. 2005. Design requirements for more flexible structured editors from a study of programmers' text editing. 1557–1560. https://doi.org/10.1145/1056808.1056965

[5] Eyal Lotem and Yair Chuchem. 2016. *Lamdu*. https://web.archive.org/web/20191002233046/http://www.lamdu.org/

[6] Merle Poller and Susan Garter. 1984. The Effects of Modes on Text Editing by Experienced Editor Users. *Hum Factors* 26 (08 1984), 449–462. https://doi.org/10.1177/001872088402600408

[7] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and et al. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (Nov. 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[8] Tamás Szabó, Markus Voelter, Bernd Kolb, Daniel Ratiu, and Bernhard Schaetz. 2014. Mbeddr: Extensible Languages for Embedded Software Development. *Ada Lett.* 34, 3 (Oct. 2014), 13–16. https://doi.org/10.1145/2692956.2663186

[9] Markus Voelter, Tamás Szabó, Sascha Lisson, Bernd Kolb, Sebastian Erdweg, and Thorsten Berger. 2016. Efficient Development of Consistent Projectional Editors Using Grammar Cells. In *Proc. SLE* (Amsterdam, Netherlands) *(SLE 2016)*. ACM, New York, NY, USA, 28–40. https://doi.org/10.1145/2997364.2997365

[10] Markus Völter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *SLE 2014*. 41–61. https://doi.org/10.1007/978-3-319-11245-9_3