Position Paper: Lack of Keyboard Support Cripples Block-Based Programming

Neil C. C. Brown
School of Computing
University of Kent
Canterbury, UK
nccb@kent.ac.uk

Michael Kölling
School of Computing
University of Kent
Canterbury, UK
mik@kent.ac.uk

Amjad Altadmri
School of Computing
University of Kent
Canterbury, UK
aa803@kent.ac.uk

***Abstract*—Block-based programming is very popular with beginners, but it has failed to gain traction among intermediate and expert programmers. The mouse-centric interfaces typically found in block-based programming environments make edit interactions (especially in large programs) tedious and awkward. We propose that adding keyboard support is a key step to extending the applicability of block-based programming ideas and would allow their use by intermediate and expert programmers, extending some of their benefits to new user groups. We describe an implementation of this idea, 'frame-based programming', which leads to a number of benefits in error avoidance and edit efficiency.**

I. Introduction

Block-based programming has gained significant popularity in the last ten years. The success of systems such as Scratch, Snap, StarLogo TNG, Blockly, App Inventor, Alice and many more demonstrate the great interest in this programming style, especially for young age groups (pupils between eight and twelve years old). This success, however, has been restricted to these young learners and is not mirrored in older age groups. It is interesting to consider why this is. After all, many of the advantages that block-based programming offers – easier manipulation, freedom from syntax errors, reduced memorisation of syntax and commands – would be of benefit to all programmers.

The reluctance of more experienced or ambitious programmers to adopt this programming style is rooted in a number of fundamental limitations of block-based programming systems, most of all those affecting the ease of manipulation and systematic organisation of the program. In this paper, we argue that the lack of keyboard support for program manipulation in block-based programming systems causes many of the weaknesses that prevent scaling to more proficient use. We describe how adding support for keyboard controlled manipulation can remove many of the hurdles and make some of the advantages of block-based systems available to more proficient programmers.

Our design of this novel interaction method, called frame-based programming, not only improves on block-based systems, but may also lead to increased efficiency for expert programmers compared to working with text-based systems.

To appear in Blocks and Beyond 2015

II. Advantages of Block-Based Programming

Block-based programming introduces several significant advantages over text-based programming which have the potential to provide benefits not only for young learners, but for all programmers. The pre-determined, uneditable structure of the blocks prevents syntax errors. In text-based programming, keywords of the language as well as syntactic structures, such as brackets, parentheses and punctuation, have to be recalled and can be mistyped or mismatched. Slips and small syntax errors distract and slow down developers and require significant mental effort for novice and intermediate programmers. In block-based environments, these errors are impossible to make, and no mental or manual effort has to be expended in memorising, recalling or typing these structures.

Some type errors are also prevented: many blocks mismatched in type or semantics cannot be snapped together, avoiding potential errors that otherwise would have to be detected and fixed. For example, an expression block cannot be inserted where a statement block is expected. While the severity of these errors decreases with increasing proficiency of the programmer, it is obvious that preventing these errors entirely is preferable to allowing such mistakes.

Blocks make it easier and quicker to create and manipulate entire syntactic constructs. Adding a fully-formed if-statement or a loop using a single gesture is faster than typing the whole construct as text. A statement can easily be dragged as a single entity and dropped into a new, valid location. In a text-based environment, the statement must first be selected, either using a combination of keyboard shortcuts or by selecting with the mouse, and then dropped at a carefully chosen location. Both of these parts are more tedious and error prone than with blocks: When selecting, it is easy to miss part of a compound statement, or to accidentally include or omit trailing line break characters. When dropping the selection, it may be placed into syntactically invalid locations. Blocks have no such issues, and deletion is similarly faster and less error-prone.

Block-based editors also remove a lot of lower-level tedium or trivial style decisions. There are no issues regarding depth of indentation, whether to use tabs or spaces, or the correct placement of curly brackets. Many other similar style issues become inapplicable.

Fig. 1. Complex expressions in Scratch are written by dragging many individual blocks together. This expression is composed of eight blocks.

Many of these advantages would benefit proficient programmers as well as beginners. However, block systems as a whole do not scale sufficiently to proficient programmers' style of work and size of programs. For anyone but beginners, the limitations of block-based systems outweigh their advantages.

III. Limitations of Block-Based Programming

One of the main limitations in block-based programming is the speed of entry and manipulation. Although it is *easy* to drag blocks, it can also be laborious and time-consuming. Performing a relatively small calculation such as the hypotenuse of a triangle (e.g. for the distance between two objects), $\sqrt{x \times x + y \times y}$ involves assembling eight blocks (as shown in Figure 1). Each block requires a sequence of gestures: finding the appropriate palette, selecting and dragging of the prototype, to dropping at the target location. In a text-based editor, the equivalent code entry requires 13 keypresses, representing significantly less interaction effort. Text-based approaches to formulas have been shown to be more usable than purely block-based approaches [1].

The lack of expressive flexibility in what is being dragged also causes slower manipulation. There is no easy way to select, for example, two or more adjacent blocks in the body of a control structure and drag them elsewhere. The contents must be unpicked by individual drag gestures, or a complex set of drags to detach and subsequently re-attach the trailing blocks which the user did not wish to manipulate.

The effort required to create or edit a program is directly related to its size. Larger programs require more entry and manipulation, and slowly the balance tips: text-based programming becomes the easier medium to write and maintain large programs. This is the key aspect which limits the use of block-based programming by intermediate and professional programmers: the ease is outweighed by the lack of speed. Here, ease refers to the low cognitive load and motor skills required to plan and execute the operation, while speed is how long the operation takes. For example, entering the expression $(1+2)\times(1+2)$ into a calculator is easy, but for those proficient in arithmetic it is faster to calculate the answer mentally.

Block-based programming also tends to have poor support for code navigation. While the free placement of program code segments (e.g. event handlers) in many block-based environments is very flexible, it often leads to disorganisation. Navigation of code written by other people – an activity rarely performed by novices, but frequently by experts – is not well supported. Key navigation of larger systems, such as jumping (both ways) between the usage and definition of an entity, is expected in professional programming environments but usually unsupported in block systems.

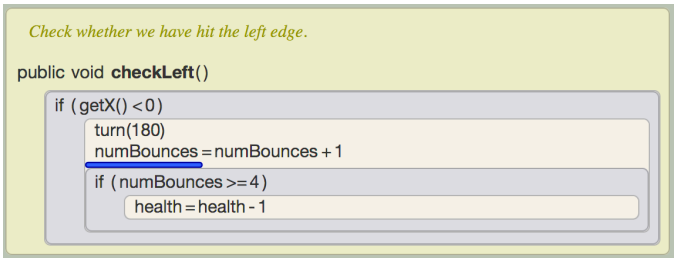

Fig. 2. The frame cursor is a thin horizontal blue line which occupies a small vertical space between frames, in the same way that a text cursor occupies a small horizontal space between characters.

IV. Frame-Based Programming

Frame-based programming is our own design of source code manipulation, combining aspects of blocks and text-based programming. One of its significant features is a combination of block-like entities – frames – with support for keyboard entry and manipulation. Keyboard control encompasses two elements: statement-level key support via a *frame cursor*, and expression-level support via *slots*.

A. Frame Cursor

Entering a new frame (akin to a statement-level block) requires two choices from the user: which frame to add, and where in the program code to add it. The former is straightforward, with different frames bound to different keys. The latter is achieved using a frame cursor. Just as textual entry on a computer involves a text cursor (or caret) indicating where the typed characters will be inserted, frame entry involves a frame cursor indicating where a new frame will be added.

The frame cursor occupies a small vertical space between frames (see Figure 2) and can be moved using the cursor keys. There are further shortcuts and modifiers: for example, pressing ctrl-up/down moves the cursor only at the current scope level, skipping the body of compound statements or entire methods. The frame cursor also allows selection: The user can use the shift modifier to select a contiguous block of frames, which can then be moved via mouse-dragging – or cut, copied, or pasted using standard keyboard shortcuts. Other keys perform logical actions: backspace deletes the current selection if there is one, or otherwise deletes the previous frame. The cursor can also be placed through clicking the mouse, and dragging can be used to create a selection.

There is only ever one cursor on screen: either a text cursor in a slot (see below) or a frame cursor. Because of this distinction, we can use single-key shortcuts for inserting frames when the frame cursor is selected. Pressing '`i`' inserts an if statement without the need for further modifiers. This makes entry of frames fast and convenient. To add a while-true loop, the user only needs to press five keys: `w t r u e`. The initial '`w`' creates a while loop and focuses the slot for the condition, and then '`true`' is typed into the slot.

B. Slots

In block-based programming, expressions are represented by blocks, with all the manipulation disadvantages discussed above. In frame-based programming, we allow expressions (e.g. conditions in loops or the right-hand side of assignments) to be entered textually with the keyboard.

Textual entry speeds up the creation of expressions, but it also re-introduces the possibility of syntax errors. The text is, however, not entirely free-form; paired symbols, such as brackets and quotes, are treated as a single entity. When the user enters an opening bracket, the closing bracket is automatically created at the same time. These brackets are paired forever: deleting one also deletes the other, and they can never nest incorrectly. Thus, some errors that may occur in traditional text editors are prevented.

Other tools typically provided in text-based environments are also provided in the frame editor: code completion, automatic corrections for mis-spelt variable names, real-time error annotations, etc. This allows the frame editor to support professional workflows and program sizes.

C. Navigation Improvements

Frame-based programming uses a structured presentation and layout for code much more similar to text than to the arrangement in block-based systems. All code is laid out vertically in classes, rather than freely placed on a larger canvas for each class. Specific segments in the code (such as field declarations, constructors, and methods) have a specified order and location. Navigation between elements (such as moving focus to the declaration of a variable, or showing the locations of uses of a variable) is supported via menu commands and keyboard shortcuts. The view pane of a frame-based class automatically scrolls to keep the frame cursor in view. Thus, the keyboard can be used to navigate the source code, by moving the frame cursor up and down the class.

V. Frames vs. Structured editing

Frame based programming is a specific variation of structure editing, an idea decades old, with a period of specific popularity and interest in the late 1980s and early 1990s. All structure editors have in common the principle that edit operations are performed on the underlying syntactical *structure*, not the textual representation on screen, and the goal of avoiding entry of many syntactically invalid programs.

Relevant examples of structure editors include GNOME [2], which used menus for entering low-level content (similar to our slots) with entry restricted to previously declared values, and Boxer [3], which used "boxes" instead of pure text – a construct that shares some aspects with our frames.

Existing structure editors managed to prevent many errors, but typically locked users into fixed workflows. Syntactically or semantically incorrect code could not be entered, even temporarily, preventing various methods of development and legitimate editing styles. This overly restrictive nature of many of the systems made them unpopular and led to failure to gain traction in the programming community.

Task	Scratch	Alice	NetBeans	Frames
Insertion	4.9	6.6	5.1	**1.6**
Modification	5.6	7.1	5.5	**5.0**
Deletion	5.4	2.6	7.8	**2.4**
Movement	5.5	**3.1**	6.0	4.8
Replacement	9.8	8.9	5.1	**2.3**

TABLE I
MEAN TIMES IN SECONDS (1 D.P.) FOR PROGRAM MANIPULATION TASK TYPES [4]. LOWER IS BETTER, BEST IN EACH ROW IS BOLDED.

In our frame editor, we do not prevent entry of many incorrect code segments, choosing instead to passively indicate erroneous code (via a red underline) without blocking the programmer from additional manipulations. Thus, we hope to provide better support while maintaining flexibility.

VI. Initial Results

An initial study evaluating the effectiveness of frame-based programming, using an earlier prototype of our editor [4], provides some first insights into its potential. The study compares cognitive models of different program modifications (insertion, modification, deletion, movement and replacement) in a prototype frame-based editor with various other systems, including Scratch, Alice, and NetBeans. Cognitive modelling computes a measure of task time by recording and analysing keystroke level interactions (such as key presses and mouse clicks) as well as "mental" operations (such as eye movement and reading time). The study was performed using CogTool, a software system that automates the recording and analysis of interaction sessions. The prototype frame-based editor was found to be the fastest in four out of five categories. Relevant results are reproduced here in Table I.

VII. Conclusion

The benefits of block-based programming have not yet been transferred to intermediate or professional programmers because the mouse-centric user interfaces make working with large programs too difficult. By combining aspects of block-based programming with keyboard support (along with several other innovations beyond the scope of this brief paper), frame-based programming removes the obstacles and extends those benefits to more proficient programmers.

References

[1] R. Koitz and W. Slany, "Empirical comparison of visual to hybrid formula manipulation in educational programming languages for teenagers," in *PLATEAU '14*. ACM, 2014, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/2688204.2688209

[2] P. Miller, J. Pane, G. Meter, and S. Vorthmann, "Evolution of novice programming environments: The structure editors of carnegie mellon university," *Interactive Learning Environments*, vol. 4, no. 2, pp. 140–158, 1994. [Online]. Available: http://dx.doi.org/10.1080/1049482940040202

[3] A. A. diSessa, "Twenty reasons why you should use Boxer (instead of Logo)," in *Learning & Exploring with Logo: Proceedings of the Sixth European Logo Conference*, M. Turcsnyi-Szab, Ed., 1997, pp. 7–27.

[4] F. McKay and M. Kölling, "Predictive modelling for HCI problems in novice program editors," in *BCS-HCI '13*. BCS, 2013, pp. 35:1–35:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2578048.2578092