Blocks at Your Fingertips: Blurring the Line Between Blocks and Text in GP

Jens Mönig
jens@moenig.org
CDG

Yoshiki Ohshima
Yoshiki.Ohshima@acm.org
Communications Design Group (CDG), SAP Labs

John Maloney
jmaloney@media.mit.edu
CDG

***Abstract*—Visual blocks languages offer many advantages to the beginner or "casual" programmer. They eliminate syntax issues, allow the user to work with logical program chunks, provide affordances such as drop-down menus, and leverage the fact that recognition is easier than recall. However, as users gain experience and start creating larger programs, they encounter two inconvenient properties of pure blocks languages: blocks take up more screen real-estate than textual languages and dragging blocks from a palette is slower than typing.**

This paper describes three experiments in blurring the line between blocks and textual code in GP, a new blocks language for casual programmers currently under development.

I. Introduction

We are currently developing a new general purpose blocks programming language, code named "GP", aimed at "casual programmers" (teen to adult). We hope to welcome new programmers with a blocks-based authoring system that is as easy to use as Scratch and to support them as they grow in expertise. GP has been designed to allow the code for complete applications, including the GP programming environment itself, to be viewed, edited, and debugged as blocks. Thus, the budding programmer need not learn a new language or even switch from blocks to textual code as their abilities and ambitions grow.

II. Problems

A. The screen real estate problem

Blocks-based programming languages replace text with graphical objects representing programming language elements such statements, expressions, and control structures. These graphical program blocks typically have borders ornamented with notches and indentations to suggest how the blocks fit together. Blocks contain embedded labels, icons, editable text fields, and interactive widgets such as drop-down menus and color pickers. As a result, a block representing a single statement usually takes more space than its textual counterpart. The actual amount of extra space depends on the visual design of the blocks, the hardware platform, and target audience. For example, a blocks language that targets young children using touch-screens might use larger blocks than one aimed at adults using laptops.

Short block scripts can be quite readable. Unfortunately, even a small increase in statement size multiplies with the number of blocks in a stack and the number of stacks in a window. This expansion spreads blocks code over a larger area than its textual equivalent, making it harder to get an overview of the code at a glance, and increasing the burden of scrolling and navigation. This screen real estate problem was pointed out long ago by Peter Deutsch[1]. In addition, the colors, borders, and graphical elements of blocks can be visually distracting, making it harder to scan the textual labels on the blocks that carry most of the meaning.

B. The input problem

A blocks palette helps newcomers quickly discover what commands are available. (Some blocks systems, such as Scratch, allow blocks to be tested right in the palette, further facilitating discovery and understanding.) However, experienced programmers who use blocks languages often complain that, once they know what commands are available, assembling scripts by dragging blocks out of a palette is cumbersome and takes much longer than it would take to type the code. Searching for a block in the palette involves searching one or more categories, visually scanning the blocks in each category and possibly scrolling to find the desired block. In addition to taking time, this process interrupts the users flow of thought about the code.

The input problem gets worse as the number of blocks in the palette grows. The first version of Scratch had about 80 blocks. The current versions of Scratch 2.0 and Snap each come with about 140 blocks, and they can be extended with external modules and user-defined block libraries that add additional blocks. Since GP is aimed at a wider spectrum of applications, its palette already includes 250 blocks, and since GP is designed to be easily extended, that number will grow.

We seek to combine the benefits of blocks with the speed of reading and writing textual code. The rest of this paper describes three experiments that explore ways to address the screen real estate and input problems.

III. Experiments

A. Switching between blocks mode and text mode

Internally, GP code is represented as abstract syntax trees that can be rendered easily as either blocks or text. Thus, the first experiment (inspired in part by D. Anthony Bau's Droplet Editor for Pencil Code [3], which we saw in 2014) is to allow the user to switch between blocks and text modes in place

[1] wikipedia.org/wiki/Deutsch_limit

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

Fig. 1. The script for a gray-scale gradient in both blocks mode and text mode. The text mode version has been edited to remove the closing curly-bracket of the "for" loop, so it is no longer syntactically correct. The vertical line at the end of the text is the cursor.

Fig. 2. The same code viewed as normal blocks (top) and as "text blocks" (bottom). The downward arrowhead in the first line is a drop-down menu widget. The horizontal arrowheads can be used to expand blocks to show optional parameters.

(Figure 1). In text mode, one can type, delete, and position the cursor at the character level, as one would in any text editor. Clicking outside of the script parses the text and converts the script back into blocks—assuming it is syntactically correct.

Text mode addresses the screen real estate problem: text consumes less space than the blocks. With the current font choices, the vertical space required for the text is about half that of the blocks. Text mode also addresses the input problem: an expert can type code quickly without having to drag blocks out of the palette. Text mode also allows an expert to create blocks for GP functions that are not advertised in the palette, a "feature" that has sometimes saved us during demos.

However, we found that text mode has a number of disadvantages. First, it re-introduces the possibility of making syntax and spelling errors. Second, switching to text loses the interaction affordances we enjoy with blocks. For example, some blocks include drop-down menus. color picker widgets, or numeric input slots that can be "scrubbed" to adjust the input value; with text, one must type expressions or constants to generate the desired values.

Finally, editing code as pure text requires using the internal form of GP code, a LISP-like function call format that does not support inline keywords. Furthermore, one must use the internal function names. In the example, the internal function name for the statement in the loop body is "self_fillRect" rather than "fill rectangle x _ y _ w _ h _". Dropping inline keywords and exposing the internal function names is confusing enough in English, but it poses even bigger programs for translating GP blocks into other languages. A huge benefit of allowing block labels to be independent of the underlying function names is that one can render the same code into multiple spoken languages. Scratch supports over 60 languages, and scripts written in one language (say, Spanish) can viewed in another (say, Japanese). Scratch even flips blocks to support right-to-left languages like Hebrew and Arabic. We want the option of creating a Scratch-like translation system for GP.

B. Blocks that look like text

The second experiment is an attempt to combine the benefits of blocks with the compactness of textual code. The idea is to retain the graphical object structure of blocks code but change its appearance and layout by removing all borders and graphical ornaments except those needed to show structure, such as in nested subexpressions. This "text blocks" mode (currently controlled by a "blocks-text" slider in the UI that operates globally on all blocks and scripts) condenses the blocks into roughly the same screen real-estate as textual code and minimizes visual distractions, thus improving the readability of larger pieces of code. However, the blocks are still there and active. They can be dragged, dropped, duplicated and assembled in different ways. To make this clear, faint outlines of the block shapes appear as the mouse cursor hovers over them. The blocks also retain any interactive input widgets such as drop-down menus, color pickers, and, as seen in Figure 2, the arrowheads used to reveal optional block parameters.

With GPs current font and layout choices, a stack of blocks in "text blocks" mode takes up about half the vertical space as that same stack viewed as normal blocks. Surprisingly, text blocks code can also take less vertical space than its textual equivalent. When viewed in TextWrangler, a popular textual code editor, a 39-line GP "quicksort" method actually required 40% *more* vertical space than it did when viewed as text-blocks using the same font. The main reason for this is that TextWrangler uses a generous amount of space between lines, possibly to help programmers locate errors by line number, whereas GP's current line spacing is somewhat cramped. However, these are details; the key point is that code rendered as "text blocks" can be at least as dense as the equivalent textual code in a conventional code editor, and thus the Deutsch limit is not an issue.

Inspired by a demo of Etoys given by Alan Kay at the 2004 OOPSLA conference, we parameterized the transition between conventional blocks and text blocks. This allows us to provide a "blocks-text" slider so that the user can set the blocks appearance to any intermediate point along the blocks-text continuum, as Alan showed in his talk. It also allows the transition to be animated, as it is in PencilCode.

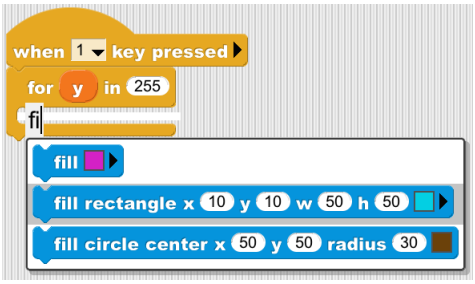

Fig. 3. Keyboard editing. A block is being inserted at the block editing cursor (white bar). The user has typed "fi" and the menu shows several possible matching blocks. The user can either type enough letters that the desired "fill rectangle" block is the only match or, as shown here, use the down arrow key to select the desired block. Once the desired block is selected the user can press the enter key to insert it.

C. Keyboard-based block editing

The third experiment explores ways to input and edit blocks code using only the keyboard. This effort was initially inspired by an interest in making GP accessible to users with visual or physical impairments, but we quickly realized that keyboard-based block editing addresses the input problem and thus benefits all GP users.

For this experiment, we added a movable "block editing cursor" to the scripts editor (Figure 3). The block editing cursor can be moved through all blocks in the scripts editor using the arrow and tab keys, and the block before the cursor can be deleted using the backspace key. Pressing a letter key allows the user to type the name of a block to be inserted at the cursor. [2] As they type, the system shows a short list of potentially matching blocks that is updated after every keystroke. Matches are determined by comparing the letters typed with the subset of blocks from the palette that would be syntactically correct at that input location. For example, when the cursor is in an input slot, only reporter blocks (expressions) are offered as possible matches. The enter key can be pressed to select and insert the top-hit in the match list, or the arrow keys can be used to select one of the other alternatives. This mechanism is similar to the auto-completion feature found in some textual code editors, although in this case the user must choose a valid block, whereas in a text editor the user can ignore the auto-completion suggestions and type something else.

Keyboard editing makes inputting blocks code much faster for experts. Features for experts can make a system less welcoming for beginners, but not in this case. A new GP user can easily ignore the keyboard editing features. They can explore the block palettes to discover what commands are available and can use drag-and-and drop to assemble blocks into scripts, just as they do in Scratch. However, keyboard editing may be useful even for a relative newcomer to GP, since it leverages the fact that recognition is easier than recall. The user need only remember (or guess) enough of a block name to make the desired block appear in the list of possible matches.

Keyboard editing supports translation to different spoken languages, since block matching is based on the (translated) block labels, not on the internal function names.

[2]Block matching was inspired by Snap's search-bar, originally prototyped by Kyle Hotchkiss (pull request #403 for Snap) and by Greenfoot3's frame editor by Michael Kölling, Neil C. C. Brown, and Amjad Altadmri [2].

IV. Reflection

These three experiments have provoked some reflections. The first experiment, converting between text and blocks seemed promising until we tried it. However, simply editing blocks code as text re-introduces the potential for syntax and spelling errors, loses the convenience of input widgets such as menus and color-pickers, and poses problems for block translation to other languages. The second experiment suggests that we can eliminate visual distraction and achieve the same compactness as textual code by changing the graphical appearance and layout. By retaining the underlying block structure, users still enjoy freedom from syntax and spelling errors, the benefits of structural editing, and the convenience of input widgets. The third experiment suggests that entering and editing blocks code can be done efficiently using only the keyboard. Furthermore, in contrast to the free-form text mode editing of the first experiment, keyboard-based blocks editing eliminates the potential for syntax and spelling errors and supports translation.

Of course, other projects have explored ideas similar to those discussed here, include StarLogo TNG [1], Greenfoot [2], and Pencil Code [3][4]. The Greenfoot paper includes an excellent discussion of other related work, including several structure-based code editors from the 1980's, Alice, and Touch Develop.

While blocks languages are a tremendous boon to beginners and casual programmers, experienced programmers often prefer text-based programming tools. While it is too soon to tell how well the techniques described in this paper—along with other techniques yet to be discovered—will serve programmers, it is our fond hope that experienced programmers may eventually find blocks programming environments more convenient and productive than the text-based tools they currently use.

References

[1] Corey McCaffrey. StarLogo TNG: The Convergence of Graphical Programming and Text Processing. Master's thesis, Massachusetts Institute of Technology, 2006.

[2] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. Frame-Based Editing: Easing the Transition from Blocks to Text-based Programming (to appear). In *The 10th Workshop in Primary and Secondary Computing Education (WiPSCE)*, 2015.

[3] D. Anthony Bau. Introducing the Droplet Editor, 2014. `https://youtu.be/PGDj1IzOtoo`.

[4] David Bau, D. Anthony Bau, Mathew Dawson, and C. Sydney Pickens. Pencil Code: Block Code for a Text World. In *ACM Interaction Design and Children (IDC)*, pages pp. 445–448, 2015.