Integrating Droplet into Applab – Improving The Usability of a Blocks-Based Text Editor

David Anthony Bau
Phillips Exeter Academy
Exeter, New Hampshire 03833
Email: dbau@exeter.edu

***Abstract*—Droplet is a programming editor that allows dual-mode editing in blocks and text for any text program. This paper presents observations and improvements to Droplet based on integrating Droplet into Applab, Code.org's JavaScript sandbox learning environment. Droplet's unique interactions with both text and blocks create several unusual problems and opportunities for improvement.**

I. Introduction

This paper describes a series of usability improvements for the dual-mode block/ext editor, [1]. Droplet provides a visual editing mode similar to Scratch [2], Alice [3], and Blockly [4]. However, Droplet is unique because it works as a text editor and provides a block interface on top of parsed text code. In previous work, Droplet the following features were implemented in Droplet:

- Blocks based on parsed text, allowing lossless conversion between blocks and text.
- A palette of short prewritten code fragments, represented as blocks.
- An editor supporting drag-and-drop assembly and editing of the blocks in a program.

Because Droplet is a text editor, many features of other block languages were initially unimplemented. For example, Weintrop [5] found that a key benefit of block languages is that the two-dimensional surface allows bottom-up assembly of code. Neilsen's usability heuristics [6] include user control and freedom through undo commands, as well as the need for guidance when choosing socket values. Finally, since Droplet can edit any text program, with the potential for errors, Dropet needs good support for identifying and recovering from errors.

This paper describes solutions for these four usability issues, as worked out in the context of integrating Droplet's JavaScript mode into Code.org's [7] Applab environment, and compares Droplet's approach, necessitated by its core identity as a text editor, to those taken by Scratch and Blockly, which are primarily block editors.

II. Background

A key motivation for Droplet is to allow students to edit any program with blocks. Droplet is built as a block editor framework that supports multiple text languages. Droplet's layout algorithm is designed to allow students to see source code in blocks the same way they would in text code. For example, text is always placed in the same rows in blocks as it appeared in the text source. This allows Droplet to achieve a smooth animation between blocks and text.

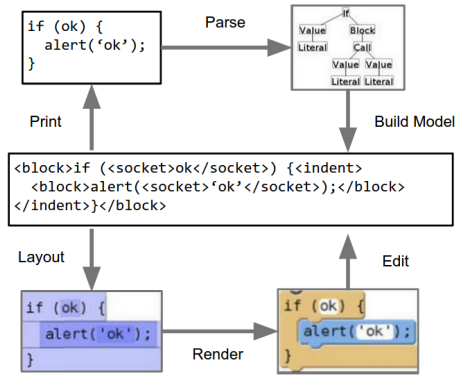

Fig. 1. Droplet's Lifecycle for a JavaScript Program

Figure 1 shows the lifecycle of a Droplet program in JavaScript. When the user opens a file, the language adapter for JavaScript runs the code through a standard JavaScript parser. It uses the resulting syntax tree to annotate the text stream to indicate the text ranges of blocks and sockets with tokens such as `blockStart` or `blockEnd`. The adapter also annotates information about color, shape, and droppability rules. Droplet then lays out and renders the resulting stream. When the user saves or runs the file, the markup is discarded to recover the original text.

III. Two-dimensional Editing

Two-dimensional editing surfaces, like Scratch supports, are beneficial to students, according to a study by Weintrop [5]. They allows students to try out different ways of performing the same task, and to compose programs in a "non-linear" way. Maloney et al. [8] refer to this as "tinkerability," and say that it supports "a bottom-up approach to writing scripts where small chunks of code are assembled and tested, then combined into larger units."

Both Scratch and Blockly support two-dimensional editing, but in different ways. Blockly runs all floating code in top-left to bottom-right order, while each Scratch block stack is associated with an event handler and runs whenever the attached event is fired. Scratch also runs a stack when it is double-clicked.

Two-dimensional editing is inconsistent with the linear nature of text programs, and our solution to this dilemma

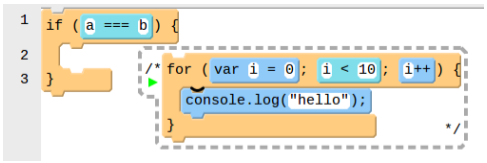

Fig. 2. An Example of Droplet's Floating Block Graphics

in Droplet is to allow the construction of "floating" blocks to the right of the main program, in the empty space in the editor. These are not executed when the program is run, and are surrounded by a dotted line and the comment symbol (fig. 2). Droplet displays a "play" button (fig. 2) that allows students to run individual stacks, but the stacks are not included in the main program.

In the future, Droplet may represent these blocks in the code by inserting them as comments. This would allow Droplet show an animation between the floating stacks in text mode and in block mode. Currently, floating blocks are lost whenever programs are converted to text or saved and loaded, since only the text code is saved.

IV. User Control and Freedom

Especially in untyped languages like JavaScript, it is easy to accidentally drop expression blocks into the wrong socket, so it is important for users to be able to recover from mistakes. Other block languages do not have this problem because sockets with information like variable names or long strings are not usually drop targets for other blocks. Scratch supports single-level undo, but neither Scratch nor Blockly supports a full undo stack. In contrast, the flexibility provided by Droplet needs to be balanced by robust support for recovery from mistakes. There are two interactions where recovery is helpful. One is when a block is removed from a socket, and the other is when the user wants to undo a previous action.

A. Remembering Old Socket Values

When a student accidentally drops a block in the wrong location, their natural reaction is to remove the block and continue dragging it to its intended location. To make editing smooth, and avoid requiring users to press the undo key, Droplet now restores old socket values to a socket whenever a block is removed from it (fig. 3). This occurs even after other alterations to the document, or after the socket has been moved.

Implementing this restoration requires a good locations model. Because Droplet frequently reparses blocks, attaching the remembered value data directly to the socket is not possible: the same socket instance may not exist after an editing operation has been done. Instead, Droplet maintains a map from socket locations to remembered values. However, the method by which locations should be serialized is subtle. The structure of a Droplet document can drastically change when Droplet reparses blocks, but the text of the Droplet document remains unchanged. This suggests that a locations should be based on text offsets.

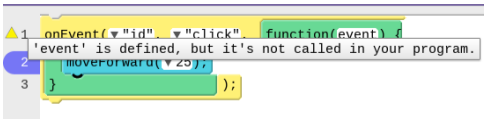

Fig. 4. An Example of Droplet Gutter Decorations in Applab

B. Full Support for Undo and Redo

The second natural action when a user makes a mistake is to use the undo command. Undo stacks are important to usability, and are included in Neilsen's widely-recognized user interface heuristics [6]. However, Droplet faced two obstacles in implementing full undo stacks for Droplet. First, because of Droplet's text-based affordances, Droplet had a large number of types of mutations, which were difficult to track and maintain. Second, Droplet did not have a way to unambiguously serialize the locations at which operations were happening.

Droplet mutations can be reduced to combinations of inserts and deletes. Locations, however, present a difficulty. A text-based location, like the remembered socket mechanism might use, is ambiguous when blocks or sockets are adjacent without intervening text. Because the undo stack would track reparses, the structure of the document when the location is retrieved is would be identical to that when it was serialized. This suggest that locations should be based on token offsets.

C. Resolving the Location Dilemma

To permit both socket value restoration and a full undo stack, Droplet uses two location models and converts between them as necessary. To assist this, Droplet has a third type of fundamental mutation: replace. A replace operation is used only for reparsing, and requires that the text content of the replaced section does not change. Droplet stores all locations as token-based offsets. When a replace operation occurs, Droplet converts any locations inside the replaced section that need to be persisted to text-based locations and converts back afterward. This allows Droplet to preserve socket locations across most reparses, but for the primary location model to be unambiguous.

V. Error Prevention and Recovery

A. Breakpoints and Line Annotations

Droplet allows users to work with arbitrary program text as blocks, which means that users can create runtime errors or code that deserves warnings. It is therefore important to support annotations and debugging tools. Line breakpoints and live annotations are a part of most major professional development environments, including Applab's text mode and Eclipse [9]. A study by Murphy [10] found that over 70% of Eclipse users use breakpoints. In 1986 Baecker [11] proposed "Metatext" or annotations as one of the five main principles of program visualization.

Applab had existing support for live errors and warnings and debugging breakpoints in text mode. Because Droplet blocks have a one-to-one relationship with text code, adding breakpoint and live line-annotation support to Droplet could easily take advantage of Applab's existing debugging infrastructure.

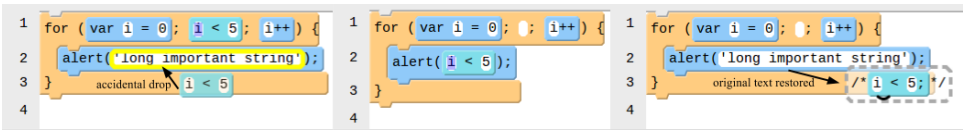

Fig. 3. An Example of Droplet Restoring Old Socket Values

Fig. 5. Droplet's New Behavior on Syntax Errors

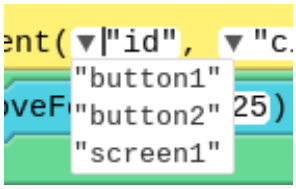

Fig. 6. An Example of Droplet Dropdowns in Applab

Droplet now supports breakpoints and annotations in the gutter the same way major text editors do (fig. 4). Droplet mimics Ace editor's API to allow Applab and other embedders to easily convert their existing debugging infrastructure from Ace editor to Droplet.

B. Handling Syntax Errors

Droplet allows users to type free-form text into sockets, which it will reparse on-the-fly and turn into blocks. This helps give students the experience of writing text without switching fully to text mode. However, it also means that users can create syntax errors by typing into sockets, unlike in other major block languages. Scratch and Blockly will only allow valid inputs in text areas. Droplet will now outline the violating input when a syntax error is created, and supports error annotations to help users identify the error (fig. 5).

VI. Recognition Rather than Recall: Dropdowns in a Text-Based Editor

Because all Droplet blocks are generated from text, Droplet did not have good support for dropdowns from sockets, which other major block languages do. Dropdowns, like autocomplete in text code, help students remember what parameters are valid, in accordinance with Neilsen's heuristic of recognition vs. recall [6].

Both Scratch and Blockly implement dropdowns for their text inputs. Both have special selectors for colors, allowing users to use a color picker or to "eyedrop" existing pixels on the screen.

Droplet added new configuration to allow the embedding application layer to specify dropdowns. Embedders may specify dropdowns by function name and argument position in JavaScript and CoffeeScript mode – for instance, in Figure 6, the "fd" function has a dropdown specified at argument 0. Dropdowns can be dynamically generated – in Figure 6, a list of element ids is generated using information taken from Applab's WYSIWYG HTML Design Mode.

VII. Acknowledgements

The author would like to thank Code.org for their support of this work, and Sarah Filman at Code.org and David Bau at Pencilcode for their advice.

References

[1] Bau, D. A. Droplet, A Blocks-Based Editor for Text Code. Journal of Computer Science in Colleges. 30, 6 (June 2015).

[2] Scratch. https://scratch.mit.edu/. Retrieved July 24th, 2015.

[3] Alice. http://www.alice.org. Retrieved July 24th, 2015.

[4] Blockly. https://blockly-games.appspot.com/. Retrieved July 24th, 2015.

[5] Weintrop, D. and Wilensky, U. To Block or Not To Block, That is the Question: Students' Perceptions of Block-based Programming. IDC '15 proceedings (June 2015).

[6] Nielsen, J. (1994). Heuristic evaluation. In Nielsen, J., and Mack, R.L. (Eds.), Usability Inspection Methods, John Wiley & Sons, New York, NY

[7] Code.org. http://code.org. Retrieved July 24th, 2015.

[8] Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. 2010. The scratch programming language and environment. ACM Trans. Comput. Educ. 10, 4, Article 16 (November 2010), 15 pages. DOI = 10.1145/1868358.1868363. http://doi.acm.org/10.1145/1868358.1868363.

[9] Mars Eclipse. http://eclipse.org. Retrieved July 24th, 2015.

[10] Murphy, G. Kersten, M. and Findlater, L. How Are Java Software Developers Using the Eclipse IDE? IEEE Software (July/August 2006) 72-82.

[11] Baecker, R. and Marcus, A. Design Principles for the Enhanced Presentation of Computer Program Source Text. CHI '86 proceedings (April 1986).