Design Considerations to Increase Block-based Language Accessibility for Blind Programmers Via Blockly

Stephanie Ludi
Department of Computer Science & Engineering
University of North Texas
Denton, TX, USA
Stephanie.ludi@unt.edu

Mary Spencer
Department of Information Science and Technology
Rochester Institute of Technology
Rochester, NY, USA
mc3630@rit.edu

Abstract *Block-based programming languages are a popular means to introduce novices, specifically children, to programming and computational thinking concepts. They are tools to broaden participation in computing. At the same time, block-based languages and environments are an obstacle in broadening participation for many users with disabilities. In particular, block-based programming environments are not accessible to users who are visually impaired. This lack of access impacts students who are participating in computing outreach, in the classroom, or in informal settings that foster interest in computing. This paper will discuss accessibility design issues in block-based programming environments, with a focus on the programming workflow. Using Google's Blockly as a model, an accessible programming workflow is presented that works alongside the conventional mouse-driven workflow typical of block-based programming. The project presented is still in progress.*

Keywords *accessibility; block-based languages; visually impaired*

1. Introduction

Block-based systems have gained prominence in recent years as a means of introducing novices to programming. The Blockly framework was developed by Neil Fraser's team at Google [1]. Blockly (sometimes with modifications) is often used as a framework for other block-based languages and activities (such as App Inventor, Gameblox, Code.org Hour of Code activities, as well as a newly announced version of Scratch [2]).

In some cases, such as MIT's Scratch, online communities exist and the tools are integrated into pre-college computer science curricula (e.g. Exploring Computer Science, CSPrinciples) [3]. As these systems have become components of curricula, after-school camps, and outreach, the lack of accessibility for many students with disabilities creates an obstacle for participation in these activities that are devised to increase participation in computing. In particular, users with visual impairments are generally not able to use block-based tools unless they have enough vision to view the screen comfortably. So one motivation for our work is to make popular block programming environments and activities more accessible to the visually impaired.

The other motivation is that, as with novice programmers who are sighted, the learning curve of dealing with the syntax of text-based code persists with novice programmers who are visually impaired. For example, many pre-college students are not familiar with curly braces and their location on the keyboard. The benefit of focusing on programming concepts over syntax is relevant to for the visually impaired as well as for sighted novice programmers. By extension, the ability to create and modify programs may be easier with the structural features of blocks compared to text, if the tools are designed appropriately.

Visually impaired individuals may have some sight or have little to no vision. Assistive technology needs vary according to the degree of sight a person possesses. People who can read magnified text may use screen magnification software to view the contents of the screen by a specified magnification factor, adjust foreground and background colors, or increase the size of the cursor. These users typically use the mouse alongside the keyboard. Individuals who are blind do not use the mouse. Instead, navigation relies on the keyboard (often through keyboard shortcuts). Screen readers (and for some, refreshable Braille displays) are the means to access information that is displayed on the screen. When a website or program is designed correctly, the screen reader will read the content, including menus and other navigational elements. In the case of typical block-based programming environments, the screen reader reads no content or navigation elements.

Google has recently undertaken work on an accessible version of Blockly [4]. Their approach focuses on screen reader compatibility, which is the foundation for access for blind users. Google's Accessible Blockly demo re-imagines the UI, so that text is used over blocks, as shown in Figure 1.

Our approach to redesigning Blockly's user interface focuses on preserving the current drag-and-drop user interface for creating block-based programs, while adding additional features to increase access to visually impaired users (or others who cannot use a mouse). Our approach includes the addition of a keyboard interface, screen reader compatibility, appearance customization, and related features to increase access to Blockly-based systems. Once complete, the authors envision that our additions could be applied to any tool that uses the block-based Blockly framework.

This paper also presents accessibility issues with the design of block-based languages. Users with visual impairments, including blindness, are the focus of this paper. Our redesign of the Blockly framework, specifically features to facilitate the

DOI reference number: 10.18293/VLSS2017-013

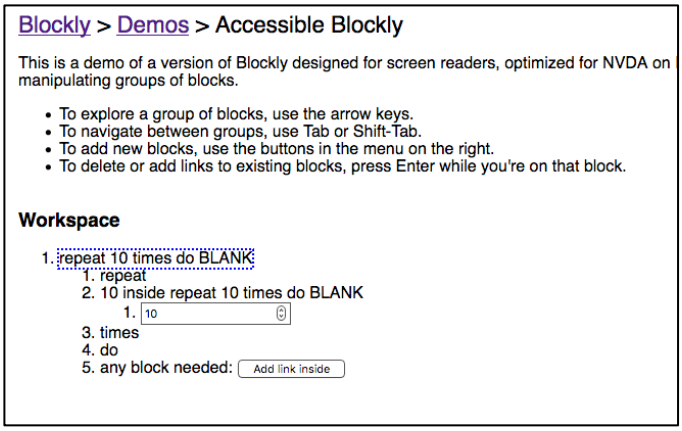

Figure 1. Screenshot of Accessible Blockly demo depicting the **`repeat`** block as text.

programming workflow and access to the block-based programs created in Blockly, are discussed. Our work is ongoing and has not had formal evaluation with users. However, the work seeks to make block-based programming accessible to the blind and other users with visual impairments, and can help other developers. Furthermore, the goal is to add the functionality on top of the Blockly framework via files that can be added to any Blockly-based project, thus making accessibility a seamless option for developers.

2. Research Questions

Using the Blockly platform, we are re-engineering the system to enable use by users who are visually impaired. As such, our preliminary research questions are:

- How can Blockly be made accessible to the visually impaired, while at the same time remain usable to sighted users?
- What features will both visually impaired and sighted users appreciate?

3. Block-Based Programming Environment Features that affect Accessibility

Each block-based project team makes design decisions for their project that in some cases have unforseen consequences. Like any software project, our team prioritizes features based on a variety of needs that are considered. Examples of these design decisions are described in the following subsections.

3.1. Technology and Platform Choices

The technologies used to develop block-based systems can have a large impact on accessibility of said systems to the disabled. The impact can be significantly positive or negative.

Scratch 2.0 uses Actionscript/Flash. A positive implication is that Scratch runs in the browser, making installation seamless and thus easy to access for many people using various operating systems. At the same time, the technology selection makes accessiblity impossible. In 2010, Adobe announced accessibility suport for Flash/Flex in terms of the Accessible Rich Internet Applications (ARIA) specification [5]. While ARIA is supported in terms of roles and states for HTML, Flash objects and Actionscript do not support ARIA roles and states, so presenting and interacting with a system is not possible (additional details are provided in Section 4.1). Another issue for screen reader users, who rely on keyboard shortcuts, is that Flash can override those keyboard shortcuts.

The Lego Mindstorm block-based robotics-programming environment is traditionally a desktop application that uses LabView as the underlying technology. While the software runs on the Windows and Mac operating systems, the software is not compatible with screen readers. As a result, users who are blind will not hear anything.

On the positive side, Blockly is developed in CSS, SVG, and JavaScript. As such, Blockly also runs on the web browser. However, Blockly does not have the same accesibility issues, because it can leverage ARIA specification from the W3C's Web Accessibility Initiative. Developers need to adhere to the ARIA specification since compatibility does not happen automatically for any web application. By following ARIA, a screen reader can read the structure of the web page, the labels on buttons and menus, as well as the graphical objects (e.g. blocks). Widgets, such as sliders and tree items, can be described. Support also includes keyboard navigation, as well as clearly articulated properties for drag-and-drop, widget states, or areas of a page that can be updated over time or based on an event. [6]

3.2. Mouse-centric Input

Many block-based systems rely on mouse input as the primary means of accessing features, including selecting blocks and adding them to programs, selecting attributes, and running the created program. One cannot use the keyboard to locate, select, and place a block onto the workspace in Scratch or Blockly as they were originally designed to rely on the mouse for such interaction.

Many users with visual impairments rely on the keyboard as the primary input device. As such, keyboard-focused commands and shortcuts are key to making interaction possible. Systems must be designed in order to utilize the keyboard as input in terms of menu navigation, programming, and accessing various panes in the programmng environment (e.g. changing focus to access specific information). In addition, the keys used to access features and information needs to be consistent with said standards and not conflict with keyboard shortcuts used by screen readers. In order to provide appropriate access to both sighted and visually impaired students, designing the system to allow for interaction via the mouse or keyboard is needed. This aspect is the foundation for much of the work in our project.

Wagner and Gray [7] discuss the use of a Vocal User Interface for Scratch. While a voice-based user interface can be used by users who are blind, their work focused on users with physical disabilities where using the mouse is not feasible or comfortable for long periods of time. The use of voice is another option for some users, though the audio presentation of

DOI reference number: 10.18293/VLSS2017-006

the system and blocks are the focus for this paper. Nonetheless, their work shows that there is not a simple one size fits all for accessibility.

3.3. Feedback

User feedback in block-based programming environments are often visual in nature (e.g. a visual change on the workspace, pop-up messages, dialog boxes) without an audio feedback mechanism. Examples include a successful compilation or incompatible blocks that the user tries to connect together. Providing associated audio-based feedback can take various approaches depending on the nature of the feedback.

Students who use screen readers need to have all content, including errors and any status messages, provided audibly. The audio capabilities of many block-based systems are limited, whether it be the ability for a robot to play a tone or recorded sound or for a Microsoft Kodu game to play a sound effect when an event occurs. The audio capabilities, including the ability for a form of audio description when an animation is played or dialog are displayed, is needed. Tapping into the location attributes or dialog text to enable compatibility with a screen reader is possible in many technologies (e.g., JavaScript, Java, C++, C#).

Audio-based feedback can be in the form of speech or sound. As a pane or dialog box gets focus, the error or status text can be read (assuming it is programmed to enable a screen reader to access the text). Other feedback may be in the form of sounds (e.g. an audio icon or earcon) that correspond to a specified meaning. An example of an audio icon is the sound of crumpling paper when a file is moved to the trashcan [8]. An example of an earcon is a tone or chord that is abstract in terms of the sound itself, but it is given meaning according to the association, such as a deep sound may correspond to an unsuccessful download of the program to a robot [8]. We are leveraging related work that has been conducted earlier to assess the use of audio cues in programming for programmers, though the study was conducted in a traditional, text-based programming environment [9]. Our work uses a working prototype of Blockly as the environment, enabling the developers to get early user feedback that helped direct the work described here.

3.4. Block-based Programs Created by the User

Each block-based program is designed for a particular programming environment. Scratch programs can be in the form of animations or games. Lego Mindstorms NXT-G programs allow a robot to move and interact with its environment. Programming environments based on the Blockly framework enable users to create programs in a variety of languages. Some Blockly-based tools produce JavaScript or Dart, where other tools are used to program robots.

4. Redesigning Blockly for Accessibility

Our work in modifying Blockly to provide access to users who are visually impaired is ongoing. The following sections provide an overview of system modifications. Our team uses only the technologies that Blockly uses, with one exception: the addition of a single JavaScript library to provide specific audio feedback.

The overarching work focuses on program creation and program navigation. The key goals of program preparation are:

- Allowing the user to change the highlight color for the current block or the connection point of a block in the workspace to improve discernability. This is in contrast to the findings in Fraser [10], where a borderless look prevails.
- Allowing the user to change the color of the workspace in order to minimize visual discomfort and improve readability.
- Enabling the blocks to be read on the workspace and in the toolbox (the menu where blocks are chosen).
- Visually linking blocks with any associated comments, where the user can jump from the block to the comment and back as desired.
- Providing a unique identifier with each block to enable visual and audio-based understanding of blocks.

While program creation is critical, one tends to program incrementally in terms of adding features or fixing defects. As such, the need to enable visually impaired users to be able to navigate their code is critical. Our team designed the following features to facilitate the navigation of code:

- Each block as well as each block part (e.g. mutator or inner block) can be read as a single block or in the program as a whole, depending on user need, as described in Section 4.1.
- The keyboard can be used to navigate between blocks and within a block (e.g. mutators), as described in Section 4.2.
- Each block (including vertical blocks) is given a unique identifier to provide a visual and auditory structure for the program. The identifier is presented in the tree view, as described Section 4.2.
- Audio cues are used in order to reinforce the level of nesting. A comparative study is underway.

The following subsections outline our efforts to increase access to Blockly in these areas.

4.1. Block Content Presentation by the Screen Reader

Blockly had no screen reader support initially. Our team added screen reader support. The text read by the screen reader is generated by our modified Blockly function that converts a

given block into a string. The original Blockly function returns a string containing both the given block and all child blocks combined. This meant entire blocks of code were read in one shot, providing too much information to the user. The function was altered to return only the selected block and any child blocks that would translate into a single line of code, for instance, the multiple blocks needed to create an **`if`** statement. Figure 2 shows a sample program that contains an **`if`** statement.

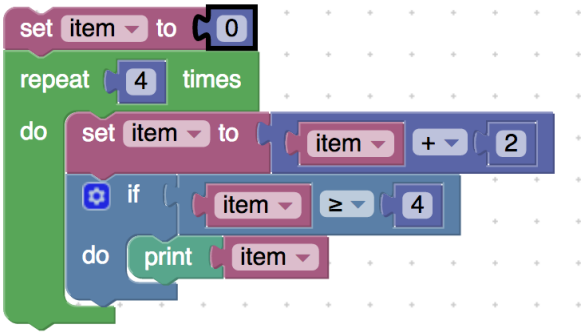

Figure 2. A sample program created in Blockly.

In the sample program in Figure 2, there is an **`if`** block inside of a **`repeat`** block. The screen reader reads the **`if`** statement as *if item greater than or equal to 4.*

The original Blockly function also replaced empty connections with question marks (e.g., create a list with items ?, ?, ?). As symbols, question marks do not translate well verbally nor do they indicate the location of the connection in relation to other connections. To resolve this, we replaced the question marks with the letters 'A' through 'Z' in order. This allows for 26 empty connections per block and the default blocks require 4 at most.

Referring back to the **`if`** block from Figure 2, the removal of *item* and *4* would result in the screen reader portraying the **`if`** block as *if block A is greater than or equal to B*. This is an improvement to the screen reader saying *if ? is greater than or equal to ?*, as stating *question mark* multiple times in quick succession may be jarring to the listener. Further testing will assess how best to provide information without overloading short-term memory.

Our team added screen reader support by dynamically adding Web Accessibility Initiative – Accessible Rich Internet Application (WAI-ARIA) attributes to the page. WAI-ARIA is the World Wide Web Consortium (W3C) standard for increased accessibility [6].

Three particular WAI-ARIA attributes are used to make Blockly accessible: aria-live, aria-label and role. These attributes are applied to a hidden div on the page that is updated with the necessary string to be read aloud. In order to have cross-browser support, the attributes are added both statically when the program loads (Chrome, Firefox, Edge/Internet Explorer) and dynamically each time the div is updated (Safari). The dynamic update each time the div is updated in Safari is a workaround at the time, but as web browsers evolve the approaches may become more streamlined.

The aria-live region attribute is placed on an element to inform the screen reader when the content of that element is updated. This region has 2 potential values: polite and assertive. A live region with a value of polite will only read the updated content after the current screen reader buffer is emptied. This ensures that the user does not lose their place mid-page. A live region with a value of assertive will interrupt the current screen reader buffer to immediately read the updated information. For Blockly, the assertive value is used so that the screen reader can immediately respond to user input when a block is selected or placed on the workspace.

While the aria-live region technique reads the content of the div on most browsers, Safari requires the div to also have an aria-label attribute. An aria-label defines what is read when an element is selected. Examples include reading the content of a block that is selected when browsing in the block menu or reading the name of each block category in the block menu (e.g. Logic, Math). Usually a div element would not need an aria-label. However, Safari only responds when the aria-label changes in the live region, not when the inner text of the div changes.

4.2. Keyboard Support for Block Menu and Block Navigation

Blockly is designed to use a mouse as the primary form of input. The only parts of the interface that are keyboard accessible are the non-Blockly-specific ones that were created using Google Closure (i.e., the outer toolbox menu). Users who are blind use only the keyboard to navigate the web, meaning all mouse events in Blockly need to be replaced by our team. This includes selecting, adding, connecting, and editing blocks as well as interacting with the mutator and context menus.

Screen readers usually have their own hotkeys for web navigation. The hotkeys are also not necessarily consistent across screen readers. This is a problem for applications that require the use of keyboard shortcuts. To mitigate conflicts, we selected a set of keys to facilitate navigation. Selecting and navigating between blocks in Blockly, for instance, are mapped to the W (up), A (left/back), S (right/next) and D (down) keys because the arrow keys are already used navigate through the HTML content of the page. These keys were chosen because they are commonly used for directional navigation in games. The WASD keyboard convention also allows the user to use their left hand for navigation and their right hand for selecting items with the enter key.

In addition to navigating among blocks and menus, it is also necessary to add comments to blocks and go back and forth between comments and blocks. As part of our redesign of the user interface, the user has a box on the side of their screen to view comments in a tree view (Microsoft Excel does this as well). The screen reader reads the comments as desired. The tree view is updated automatically, and the structure will match that of the program using the hierarchy of identifiers associated

with each block. In addition, a line will connect the current block with the comment line in the tree view.

The hierarchy was created using an n-ary search tree that cycles through a block and all of its children. Each block is given an alphanumeric label to help give the user a sense of location while navigating. The outermost blocks are lettered (A), while each inner block is given a number (A1), and each additional nested level is given a decimal (A1.1). In Figure 2, the outer **`set`** block is A, the **`repeat`** block is B, and the two blocks nested with the **`repeat`** block are the inner **`set`** block (B1) and the **`if`** block (B2). By extension, the **`add`** block that is connected of the **`repeat`** block is noted as B1.1.

4.3. Connecting and Editing Blocks on the Workspace

In order to use the keyboard to add blocks to the workspace, an edit mode was provided. This mode is needed to avoid conflicts with other Blockly features. If a user presses the E key when a block is selected, the WASD navigation keys transition from selecting different blocks to selecting any of the connections or fields on an individual block. Each time the user switches connections, the screen reader announces the new connection. Pressing the Enter key on one of these connections or fields allows the user to attach another block (setting an insertion point) or edit the field. If a block is added to the workspace unconnected, it will automatically be attached to the last outermost block. If a block cannot be attached to another block, then it will be unconnected and need to be moved or otherwise managed (e.g. deleted if undesired). As noted previously, users insert and move blocks by setting an insertion point if the desired location is different from the current location. The authors are currently assessing how to best accomplish this critical feature, including how to clearly articulate the location of the insertion point.

In Blockly, some blocks have mutator menus that allow the user to drag additional inputs or statements onto a block. For example, an **`if`** block can turn into an **`else if`** block. The mutator menu is a pop-up dialogue where users can drag and drop blocks onto the existing block to alter it. This mouse-based menu was revised in two ways by our team. Some mutators were turned into individual blocks such as the **`else if`** and **`else`** blocks. Other mutators, namely the ones that changed the number of outputs on a block, were given a drop-down menu that would dynamically add and remove outputs as necessary.

4.4. Additional Features to Enhance Access

Additional changes were made to enhance the overall experience for visually impaired users, which may also appeal to sighted users. These changes include adding the ability to change the text size and color of the workspace, an accessible custom help guide, and a comment display window.

Themes

Three themes were added to the workspace: high contrast, off-white, and matte blue. These colors were added after a participant in an early study commented on the eyestrain caused by the original bright white workspace. All of the colors were tested with a color blindness simulator and a member of the team who is color blind to ensure that the blocks were distinguishable from each other and the background.

Accessible Help

An accessible custom help site was created with references for each block. When a block is selected, a hotkey can be used to open that particular block's help information on the site. The default help pages for the Blockly library led to a page that is not fully accessible and provided limited information. The new site was specifically designed by our team to navigate with a screen reader and has detailed information on each block.

5. Next Steps

The initial version of accessible Blockly should be completed during Winter 2017. The results of a prior audio feedback study provided early feedback on the user interface, as well as assessed the impact of various types of audio feedback modalities for code navigation and the understanding of nesting. The feedback will be used in implementing code-based audio feedback during code navigation, in conjunction with the option for screen reader use, if needed. The accessibility features will also be studied in order to compare the usability impact for users with and without sight in order to ascertain what value may be found for sighted users as well as those who are visually impaired. Concerns include the verbosity of the information being presented, as well as the issues that young users may have as they are often users of Block-based systems. The formal studies will allow the team to refine the system and provide greater access for users, while providing a model for developers of other Blockly-based systems or block-based systems in general.

In addition to block-based languages, hybrid text and block-based languages such as Pencil Code [11] also have the potential for increased accessibility. Our team is also looking at applying our strategies to Pencil Code, but the teams behind Pencil Code, Code.org's App Lab, and other block languages can integrate accessibility into their system's designs by following the ideas we have discussed here. Some changes are at the user interface level while other accommodations are deeper in terms of new or redesigned features. An example of this is the Stride frame-based editor used in tools such as Greenfoot [12]. This editor enables users to work with operations and constructs at an abstract level. This innovation, and others like it, may help increase access to computer science for students with disabilities, as well as students overall.

Acknowledgments

Thanks to all the hard work on the RIT Blockly team, as well as the participants who gave initial feedback. This project is supported by the National Science Foundation (CNS-1240856, CNS-1240809).

References

[1] Blockly Development Website. [Online]. https://developers.google.com/blockly

[2] J. Goode. "Exploring computer science: An equity-based reform program for 21st century computing education," *Journal for Computing Teachers*. Summer, 2011.

[3] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, (2010). "The Scratch programming language and environment." *ACM Transactions on Computing Education*, vol. 10, no. 4, pp. 16:1–16:15, Nov. 2010.

[4] Demo of Accessible Blockly from Google. [Online]. https://blockly-demo.appspot.com/static/demos/accessible/index.html

[5] A. Kirkpatrick. (2010) Adobe Accessibility / Flash Player and Flex Support for IAccessible2 and WAI-ARIA. [Online]. http://blogs.adobe.com/accessibility/2010/03/flash_player_and_flex_support.html

[6] W3C-WAI (2014). WAI-ARIA Overview. [Online]. http://www.w3.org/WAI/intro/aria

[7] A. Wagner and J. Gray. "An empirical evaluation of a vocal user interface for programming by voice," *International Journal of Information Technologies and System. Approach,* vol. 8, no. 2, pp. 47-63, Jul. 2015.

[8] T. Dingler, J. Lindsay, and B. N. Walker, "Learnability of sound cues for environmental features: Auditory icons, earcons, spearcons, and speech," *Proceedings of the International Conference on Auditory Display (ICAD 2008)*, Paris, France, 24-27, Jun. 2008.

[9] A. Stefik, C. Hundhausen, and R. Patterson. "An empirical investigation into the design of auditory cues to enhance computer program comprehension," *The International Journal of Human-Computer Studies*, vol. 69, no. 12, pp. 820-838, 2011.

[10] N. Fraser, "Ten things we've learned from Blockly", *IEEE Blocks and Beyond Workshop*, 2015, pp. 49-50.

[11] D. Bau, D. A. Bau, M. Dawson, and C. S. Pickens, "Pencil code: Block code for a text world," in *Proceedings of the 14th International Conference on Interaction Design and Children (IDC '15),* ACM, 2015, pp. 445–448.

[12] T. W. Price, N, C.C. Brown, D. Lipovac, T. Barnes, and M. Kölling, "Evaluation of a frame-based programming editor." In *Proceedings of the 2016 ACM Conference on International Computing Education Research (ICER '16),* ACM, 2016, pp. 33-42.