

# Living Guide to Making Unruly Screen Reader Accessible

Begun by Annabelle Platt

## Intro

This document is an analysis of best practices and recommendations to make the Unruly Splats interface accessible to screen readers. It should be a living document, updated as Unruly discovers more and better ways to make their product accessible to Blind/Low Vision (BLV) students.

## Requirements

Because Unruly Splats is a product used by children in publicly funded schools, the website must conform to at least the second level of Web Content Accessibility Guidelines. This includes being accessible to screen readers. The goal of this project is to determine the best way to achieve compliance with that criteria, as well as to make the best possible experience for BLV students using Unruly Splats to learn to program.

## Screen Readers

Screen readers convert HTML into audio that allows blind or low vision users to navigate websites. It follows that a screen reader is only as good as the HTML it is provided. There are many different types of screen readers, but they largely operate with the same principles.

The most common screen readers in general are:

- ChromeBooks, where Unruly is primarily used, has a builtin screen reader, ChromeVox  
<https://support.google.com/chromebook/answer/7031755?hl=en>
  - This is a free browser extension
- JAWS (Job Access With Speech)  
<https://www.freedomscientific.com/products/software/jaws/>
  - JAWS costs money
- NVDA (NonVisual Desktop Access) <https://www.nvaccess.org/about-nvda/>
  - NVDA is free to download
- MacOS has a screen reader called VoiceOver which I've heard is good  
<https://support.apple.com/guide/voiceover/welcome/mac>
  - This comes installed on Macs and can be turned on in settings
- Windows 10 has a builtin screen reader called Narrator  
<https://support.microsoft.com/en-us/windows/complete-guide-to-narrator-e4397a0d-ef4f-b386-d8ae-c172f109bdb1>
  - Narrator comes installed on Windows
  - According to a review I read (<https://www.afb.org/aw/18/10/15273>) , Narrator has been very bad for most of its history and only recently improved, it's now passable but most people won't bother when they already know how to use JAWS or NVDA

Screen readers rely primarily on keyboard commands, although users who have some sight may also be able to use a mouse for input. For the purposes of universality, we must assume that the user would not be able to use a mouse. However, zoom capabilities should be available for all areas of the site so low vision users are able to zoom in and read content. The specific key commands vary from screen reader to screen reader.

Screen readers read content linearly, one item at a time, from left to right and top to bottom. However, using navigation menus and ARIA landmarks can help direct the screen reader elsewhere. ARIA landmarks are required for WCAG 3, which we don't need, but they will provide a lot more flexibility for navigating between areas of the site. They allow for more specific attributes that make the HTML more informative for screen readers. Read more about them, including some rules for development at <https://webaim.org/techniques/aria/#landmarks>. This will be extra important for Unruly since we have a non-linear content flow that the user needs to constantly navigate around.

### **Proposed Solution:**

#### Macro Navigation:

There are four different sections to the main screen:

- Splats pairing
- Block menu
- Workspace
- Sidebar navigation

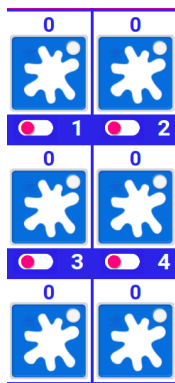
The user must be able to navigate between these four areas. In addition, the user must be able to easily navigate between the block menu and the workspace, since moving between these two is how the bulk of the programming will actually happen.

I propose that the user can use tab to move through the macro elements. For the shortcuts between the block menu and workspace, I propose two options which should be user tested. First, simply have shift + tab navigate in the reverse direction, so the user can use tab and shift + tab to move back and forth. Secondly, have A toggle back and forth between the menu and the workspace (so hitting A repeatedly will alternately open the menu switch to the workspace).

#### Micro Navigation:

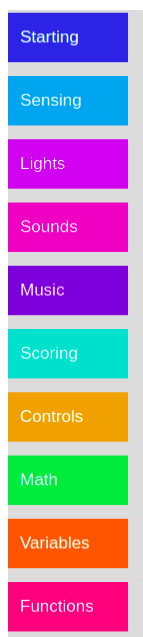
In order not to interfere with regular screen reader shortcuts, I propose using the WASD keys for most micro navigation, as used by Ludi and Spencer [1]. However, they use the D key for down and the S key for next. I think this may be confusing as most kids are used to the layout of the arrow keys, and swapping the functions of the S and D keys will likely be confusing. I therefore recommend simply using the WASD keys as one would use the arrow keys. The micro navigation of each section is discussed in greater detail below.

### Splats Pairing:



If the user selects the pairing area on macro navigation, they should now be able to use the WASD keys to navigate between the Splats. The screen reader should read the Splat ID and status, e.g. “Splat 1 is connected”, “Splat 2 is disconnected”. The user should be able to hit a key to toggle the Splat connection. For consistency with later shortcuts and ease of access I recommend E, but this should be verified via user testing. The screen reader should read this key if the user remains on a Splat until the screen reader is done reading, or possibly if they select the Splat (again, verify via testing). It should say something like “Press E to connect Splat 1”. The exact verbiage of “connect” or “activate” or something along those lines can be decided at the discretion of the team.

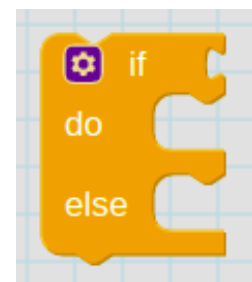
### Block Menu:



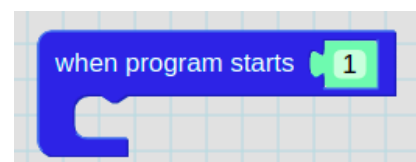
The block menu itself is relatively straightforward for the screen reader to navigate. As long as it’s tagged as a navigation menu, the user should be able to navigate through the headings with WASD. The screen reader should be able to read the headings, e.g. “Starting”, “Sensing”, “Lights”, etc. I would recommend reevaluating the clarity of each heading name, as it should be clear to the user what type of code blocks live in each section just from the heading name. In particular, I would consider renaming “Controls” to “Logic”.

Once the user selects a heading, they should be able to use the WASD keys to navigate through the blocks inside. The screen reader should read the currently selected block. As the blocks are also arranged linearly, navigating through them should be straightforward for the screen reader. What is more complicated is reading the actual blocks. Ludi and Spencer’s approach was to create a function that takes a block and generates the text to be read by the screen reader [1]. This seems like the best approach to me. The description of the block to be read can be generated by a combination of Blockly’s alternative representation, plus some manual tweaking.

Ludi and Spencer replaced empty connections/places where other blocks would go with letters to help with clarity when hearing blocks read aloud [1]. For example, the if block shown here would be read as “If A, do block B, else block C”. Some blocks, such as the starting block shown on the right, may require slightly more explanation, as I’m not even sure what “when program starts 1” means. Without adjustment, this block would be read as “When program starts 1, block A” which doesn’t make much sense.



A question I have that should be determined through user testing is whether talking about abstract blocks such as “Block



A” and “Block B” will make sense to young children. The idea of variables is a more advanced one, so this way of reading blocks should be tested to see if it makes sense to our young users. It may be that saying “If something is true, do something, else do something else”, or “If a block is true, do a block, else do another block” makes more sense to them. Here we see the intersection between programming concepts being challenging and screen reader clarity being challenging. This will only be able to be resolved through user testing. The flip side of this is that, as pointed out by Raquel, many young children may not be able to read yet, so having a screen reader read the blocks to them may help sighted children understand better as well. Thus, it is very important that the language used makes sense to the young children who use our product.

#### Navigating the Workspace:

Users should be able to use the WASD keys to jump between code blocks on the workspace. A and D should move forward and back to the block with the top left corner at the next highest X coordinate. If there is no block with a higher X coordinate, it should wrap to the first block on the next line. If it is the furthest bottom and rightmost block on the workspace, pressing D should wrap back to the first block. W and S should navigate up and down in the same manner.

The big question here is how much of the code block the screen reader should read as it navigates between blocks on the workspace. And, the bigger question is, how do we help BLV children understand the overall layout of their code? The problem of how blind programmers can skim code is a longstanding one. Some use a braille display, but we cannot assume that any of the children using our product have a braille display. Milne and Ladner in [2] found that giving children a spatial sense of the code was effective. In their program, there were essentially filler blocks that the children could count that would give them an understanding of how nested a block was. They later modified the program to read the innermost block, followed by “inside”, followed by the next outer block, etc. I did not find this source early enough in my research to properly consider how this might apply to Unruly, but perhaps the inside-out reading approach is something to consider to convey the nested structure of the program. I recommend investigating this further and potentially contacting the study authors. The authors found that aurally communicating the structure was not preferred so I will not go into it here. It did not match the mental model of the children, as the children retained information spatially.

I propose the following:

For the shorter term, the screen reader can simply read the entire block of code, although when the user navigates to the next block of code it should interrupt itself and start reading the new block. This should be tested against the same method of reading the whole block, but from the inside out as described above. For Unruly, this might mean the program would read “Play snare drum on Splat 1 inside if  $A < B$  inside when program starts 1”. I’m unsure which would make more sense in the context of Unruly.

In the longer term, whichever reading style is chosen can then be honed through user testing to determine whether perhaps the screen reader should read only the first line or the first few lines on navigation, and only read the rest of the block if the user selects it. Further, it can be tested

to see whether or not it is useful to have a greater degree of granularity for the user to control what code is read. For example, reading all the parent blocks could be the first degree of granularity, then all their immediate children, then all of the children's children? This would need to be rather carefully controlled for the code to make any sense.

### Connecting Blocks:

Drawing on both Ludi and Spencer [1] and Milne and Ladner [2], it seems that the best way to connect blocks is by selecting a connection point and selecting a block to go in that connection. In [1], the authors describe selecting a connection and then a block from a menu. However, the authors of [2] found through user testing that users prefer to select a block first and then a connection.

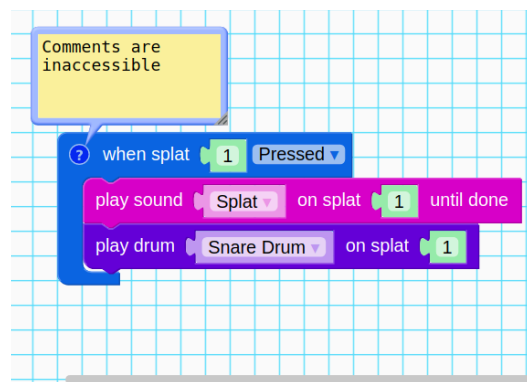
For Unruly, I recommend using the method found via testing in [2]. In the method, the user selects a block from the menu, as described previously. "Edit" or "Selection" mode then activates, allowing the user to select a block with WASD and then use WASD to select a connection on that block at which to add the new block.

### Reading Output:

A hot key should be bound to run the code so users don't have to navigate to the button. R would make sense and be easy to remember.

A simple solution to reading output is to have the screen reader reading updates that occur, such as numbers incrementing or lights flashing. Audio outputs don't need reading. However, we have the advantage of a physical form factor in the Splats, and we should leverage this as much as possible. Some of this may be environment dependent, such as teachers not directing blind students to do visual-based tasks. However, if the Splat can make sounds and vibrations, perhaps we could have it vibrate each time it increments in BLV mode. I think there's more work to be done on figuring out more ways for this to work, but the fact that there are already sound effect outputs is great.

### Comments:

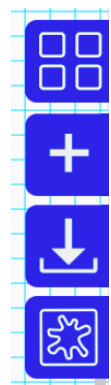


Currently, comments are in a callout box that is inaccessible to screen readers, as shown on the left. I propose moving comments to, or at least adding, a sidebar like in Google. All comments get added to the sidebar. As this panel will have a linear layout, it can be navigated easily via screen reader. There should be a short key for leaving a comment - C might be appropriate. There should also be a hot key for jumping between the comment and the code the comment refers to, in the same way that when a comment in the Google sidebar is clicked, the

document jumps to the location of the comment. This will be a useful tool for teachers working with BLV children, who can leave comments to help them navigate more easily or direct them to

a particular location. However, this can also improve the experience for sighted children, with all the comments in the same location. Teachers can similarly use the comments to direct all students, sighted or not, to particular pieces of code, or to leave feedback. Adding this sidebar will not only make comments screen reader accessible, but improve the whole experience of commenting.

#### Sidebar Navigation:



The final piece here is the sidebar to the right of the screen. I did not have time in this study to get into the contents of each button, so that should be looked at in the future. The important thing about this sidebar is that it be tagged as a navigation menu, with appropriate and informative headers for each button. It's not important that it physically appear as a navigation menu, but critical that in HTML it is represented as such. This will allow a screen reader to navigate it easily. I also recommend that the new comment tab described above be added as the first item in this sidebar - this sidebar for consistency and to minimize the number of macro navigation jumps necessary, and first as it will likely be the most accessed.

#### **Appendix**

1. Design Considerations to Increase Block-based Language Accessibility for Blind Programmers Via Blockly:  
[http://ksiresearchorg.ipage.com/vlss/journal/VLSS2017/vlss17paper\\_13.pdf](http://ksiresearchorg.ipage.com/vlss/journal/VLSS2017/vlss17paper_13.pdf)
  - This was the primary study I relied upon for this research, as it most closely addressed the problems I was trying to solve
  - A demo from this team: [https://www.youtube.com/watch?v=\\_22OhbIL3xA](https://www.youtube.com/watch?v=_22OhbIL3xA)
2. Blocks4All: Overcoming Accessibility Barriers to Blocks Programming for Children with Visual Impairments: <https://milnel2.github.io/files/Blocks4All.pdf>
  - I found this only at the end of my research but it has some really great user testing findings in it and should be explored more thoroughly
3. Accessible Blockly:  
<https://developers.google.com/blockly/guides/configure/web/keyboard-nav>
  - Not great in some ways but has some very useful specificity that may be helpful in answering questions when coding an actual implementation. It also details the keyboard navigation already available in Blockly that we can leverage
4. Demo: Expanding Blocks4All with Variables and Functions:  
<https://dl.acm.org/doi/10.1145/3308561.3354588>
5. Nonvisual Tool for Navigating Hierarchical Structures:  
<https://cs.winona.edu/cscap/papers/assets04.pdf>
6. StructJumper: A Tool to Help Blind Programmers Navigate and Understand the Structure of Code: <https://dl.acm.org/doi/10.1145/2702123.2702589>
7. Investigating Challenges Faced by Learners with Visual Impairments using Block-Based Programming/Hybrid Environments: <https://dl.acm.org/doi/10.1145/3373625.3417998>

8. Accessibility of Block-Based Introductory Programming Languages and a Tangible Programming Tool Prototype:  
[https://link.springer.com/chapter/10.1007/978-3-030-58796-3\\_4](https://link.springer.com/chapter/10.1007/978-3-030-58796-3_4)
  - Contains user interview data but more on the “what’s wrong” side than “how does our solution work” side
9. CodeMirror a code editor embedded in a website more easily navigable by screen readers: <https://codemirror.net/doc/manual.html>
10. AudioHighlight: Code Skimming for Blind Programmers:  
<https://ieeexplore.ieee.org/document/8530030>
11. Exploration of the Use of Auditory Cues in Code Comprehension and Navigation for Individuals with Visual Impairments in a Visual Programming Environment:  
<https://dl.acm.org/doi/10.1145/2982142.2982206>
  - Most likely not very applicable for actually coding but may have some relevance at some point
12. ARIA landmarks development guidelines:  
<https://webaim.org/techniques/aria/#landmarks>.