Accessible to Whom? Bringing Accessibility to Blocks

Andreas Stefik
andreas.stefik@unlv.edu
University of Nevada, Las Vegas
Las Vegas, NV, U.S.A.

Willliam Allee
william.allee@unlv.edu
University of Nevada, Las Vegas
Las Vegas, NV, U.S.A.

Gabriel Contreras
gabriel.contreras@unlv.edu
University of Nevada, Las Vegas
Las Vegas, NV, U.S.A.

Timothy Kluthe
kluthe@unlv.nevada.edu
University of Nevada, Las Vegas
Las Vegas, NV, U.S.A.

Alex Hoffman
alex.hoffman@unlv.edu
University of Nevada, Las Vegas
Las Vegas, NV, U.S.A.

Brianna Blaser
blaser@uw.edu
University of Washington
Seattle, WA, U.S.A.

Richard Ladner
ladner@cs.washington.edu
University of Washington
Seattle, WA, U.S.A.

ABSTRACT

The introduction of block-based programming has gradually changed the landscape of programming education, particularly for school children. Block languages today, however, have serious technical barriers to students with disabilities. For example, block languages are generally not screen reader accessible, incompatible with braille, and contain serious problems for users with motor impairments. No student with a disability should ever be denied access to learning computer science and they do not have to be. To help rectify this, we present a new approach to the design of block languages called Quorum Blocks. Quorum Blocks uses a custom hardware accelerated graphical rendering pipeline that takes into account how screen readers and other devices work under the hood. We discuss these technical details and demonstrate that accessibility support can be fully achieved without meaningfully losing either the look of modern blocks or their visual output. We present the results from focus groups that highlight the barriers students faced with a variety of disabilities when using the first version of Quorum Blocks. We focus especially on challenges with low vision users, screen reader users, or those using no mouse and only one hand to type. Block languages built using either our techniques, or on top of our libraries, would become accessible out of the box.

CCS CONCEPTS

• **Social and professional topics → People with disabilities**; • **Software and its engineering → General programming languages**; • **Human-centered computing → Accessibility**.

KEYWORDS

Accessibility, Block Languages, Computer Graphics, Human Factors

ACM Reference Format:
Andreas Stefik, Willliam Allee, Gabriel Contreras, Timothy Kluthe, Alex Hoffman, Brianna Blaser, and Richard Ladner. 2024. Accessible to Whom? Bringing Accessibility to Blocks. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024), March 20–23, 2024, Portland, OR, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3626252.3630770

1 INTRODUCTION

Traditional development environments using text-based languages often start users with a blank page where they need to type. When learning, students memorize passages of text and may make a mental model of the code over time. In professional environments there exists many visualizations and helpers (e.g., code completion, editor hints), but they arguably require learning to use effectively. In contrast, block languages, while there is still memorization, mental models, and learning, often have visualizations and palettes that provide help (e.g., scope, available commands). Evidence in the literature has largely supported the hypothesis that such affordances help students in practice and impact transfer of learning [37].

While block languages have documented benefits, it is important to acknowledge that there have been negative consequences of such technologies. Today's block environments are not really "for all" and often exclude people with disabilities [12]. There has been significant recent interest in improving accessibility support due to overlapping interest of ethics (like CSAccess [6]) and changes to laws (like Maryland's SB0617 [13]). Generally, block languages require one to be able to see, have enough motor control to use drag and drop without a keyboard, and are mostly used for small programs in a first programming course. In fact, since the inception of Scratch, SNAP!, Alice, and others, small benefits to transfer of learning have been documented [37], but the needs of students with disabilities have been minimally studied and most block languages remain either inaccessible or only partially accessible. That said, exactly what it means for block languages to be accessible and universal is deceptively difficult [5, 19, 21, 29, 30]. We need to ask:

accessible to whom? For example, students with different kinds of disabilities may have different needs for both the input (e.g., the blocks or text) and the output (e.g., cats, 3D items, charts).

We make three primary contributions. First, we contribute an accessible rendering pipeline for graphics, which fixes an issue that can delay the user experience we call *Accessible Starvation.* We discuss technical details here because it is a key issue for the accessibility of tools for computer science education. Any programming language can use similar techniques for accessibility of graphics. Second, we have re-imagined what block languages could be from the ground up through the lens of both being born accessible and general purpose. Our design includes a large spectrum of features like Find/Replace, Copy/Paste, version control through Git, structural navigation, code completion, editor hints, 2D and 3D visual editors similar to Unity3D, in addition to accessibility affordances. All graphical features are accessible through screen readers and braille, use of the mouse is optional, and other accessibility considerations have been reconsidered from first principles. Finally, we conducted an evaluation of our tool in one focus group with teachers of the blind and visually impaired and two more with students in high school with various disabilities. We examine the successes and failures of our first attempt, highlighting student experiences.

The rest of the paper is as follows. First, we discuss related work. Then, we report how our accessible block language technology works on a technical and user level. We then follow-up with our formative focus groups and conclude.

2 REVIEW OF RELEVANT SCHOLARSHIP

There is an abundance of literature available on block programming. Some claims [24] centered around the idea that mastering programming is difficult because it often lacks an interesting context, does not provide enough guidance, and does not provide encouragement to dive deeper when things go awry. The literature often uses words like tinkerable, meaningful, or social.

In practice today, block-based programming may make it easier for learners. Possible causal mechanisms include features like visual cues, some mitigation of syntax errors, presentation of available commands, removing typing challenges, visual representations of lexical scope, and the use of natural language. The exact causal mechanisms and their effect sizes are less clear. Importantly, however, evidence suggests there is no significant difference in learners' capabilities with a text-based programming language after 10 weeks of learning, regardless of whether they transitioned from blocks or started with text [37].

In terms of how people use block languages, we often consider the "breadth," or range of distinct features used, and "depth," the amount of features used [28]. Unfortunately, in addition to high dropout rates, in practice learners do not pursue increases to breadth or depth without curriculum guiding them to do so [16].

While most studies in the literature are on students at a young age, several have addressed adult novices learning how to program [25, 26] with block-based programming environments. Generally, the focus in this kind of work has been more on making programming tasks easier for adult novices than on educating them in traditional computer science principles. Just as an exemplar, one case study used block languages to improve usability for programming industrial robots [31].

The research literature contains many block programming studies toward education and beyond. An abridged list includes curriculum considerations like grading and rubrics [18], parallel programming [32], notebooks [36], and data science [3, 4]. In a sense, instead of making general purpose block languages, the community has made lots of little ones. Most common block languages are not accessible to people with disabilities in practice.

Finally, while the use of hardware accelerated graphics and screen readers is poorly understood, it is necessary for block and text-based languages because the input and output needs to be accessible. Making blocks accessible without the output is like solving half the problem. Unfortunately, there is little work in the literature and the closest that inspired us was work on gaming and accessibility. For example, Roden and Parberry created a game engine which was capable of making audio only games set in 3D environments [27]. Blind Adventure is a game engine designed for blind game designers that is capable of making audio-based games [33] and Torrente *et al.*'s E-Adventure, a game engine for accessible web-based games, was built for both fun and more serious settings like education [35]. While accessible gaming is commonly evaluated in the literature, our general purpose accessible graphics architecture is novel to this work.

3 A BRIEF INTRODUCTION TO ACCESSIBILITY PROGRAMMING

While information on Web Content Accessibility Guidelines (WCAG) 2.1 AA is likely the most well known resource about accessibility, it is not prescriptive. It does not tell you how to achieve accessibility. Implementation details are thus hard to find in the literature and generally left to operating system manufacturers like Microsoft, Apple, or Google. Unfortunately, these manufacturers provide somewhat minimal documentation and, in our experience, what is online is imperfect. Further, block languages, especially, make serious errors in how they connect to the accessibility systems, which causes them to be inaccessible.

Thus, we discuss here implementation details on how and why accessibility works. The purpose is two-fold. First accessible graphics cannot be understood without understanding how accessibility is programmed under the hood. Second, implementation details in block languages are a key reason why block languages are not accessible—dancing cats, dragons, 3D editors, and such are all fine even for students that cannot see the screen, but only if the operating system receives the right messages, at the right time, in the right way. Otherwise, accessibility devices cannot translate this information into sound, touch, or other alternatives. Thus, by *accessible graphics*, we mean the graphics system, the operating system, and the accessibility system have to coordinate.

In colloquial terms, an application is accessible if people with disabilities can meaningfully interact with and use it. What this means is application dependent and reasonable people will disagree on the details. For example, for an application to be accessible to people who are blind or visually impaired, it must include keyboard support and auditory (such as speech) or tactile (such as braille) feedback. When that feedback appears is typically controlled by accessibility technologies, not host applications, and this is crucial to understand. Technologies for the blind should not, with rare exceptions, trigger their own speech. This can cause double speaking and will not work with braille [1]. Accessibility systems must communicate back and forth with the operating system.

Consider a button, which are not just pixels on the screen. Buttons are a declaration to the operating system that those pixels mean a "button." Each operating system implements the semantics of buttons through APIs (e.g., dimensions, screen location, enabled, color). To name some, these include NSAccessibility on Mac OS X [9] and UIAccessibility for iOS [10]. On Android, the model is integrated into the view API [7]. On Windows, the API is User Interface Automation (UIA) [17]. The web is different, but in practice maps down to the operating system analogously to what desktop applications do under the hood. Each of these models differs in important ways that are outside the scope of this paper. Incorrect or missing implementations of these APIs automatically means an application does not communicate with accessibility technologies. Thus, using the APIs correctly is the minimum bar, but hardly means an environment is inviting or easy to use.

While there are many semantic models, we discuss only the one for Windows for simplicity—Microsoft User Interface Automation (UIA). UIA is a semantic model used by assistive technologies to know what is on the screen and what those things can do. While this is conceptually true, on a technical level UIA is an interface for interprocess communication. Specifically, "server" processes expose the available interface elements and "providers" allow "client" processes to query information. For example, a host application tells the operating system it exists and there is a button in it. A screen reader reads the operating system, finds the button, and messages can be sent back and forth between the intermediary and host application (e.g., Where are you? Can I click you?).

Elements in accessibility libraries like UIA are typically organized in a tree structure, with the window as the root and interface components as the structure. Each element has an associated "provider," which exposes information to external processes. Information a provider offers includes a control type, a set of UIA patterns, and a set of properties. UIA supports 38 different control types representing common UI elements. We draw attention to the control types because they dictate which patterns and properties must be supported by a provider. These control types provide common controls like buttons or textboxes. There is no "accessible blocks," "dragon," or "3D camera" pattern. Other platforms have similarities, but many quirky differences outside of this paper's scope.

To manage graphics, the UIA documentation includes these details (emphasis added): "The UI Automation extensibility features enable third parties to introduce custom, *mutually agreed-upon* properties, events, and control patterns to support new UI elements and application scenarios." We added emphasis here to the phrase "mutually agreed upon." This essentially means manufacturers and creators of accessibility products must coordinate and agree, which is a social problem that is hard to achieve in practice. One of the larger issues that needs resolution for block languages and their corresponding output to be accessible is what we term *accessible graphics* and its associated *accessible starvation.*

4 CONTRIBUTION 1: ACCESSIBLE GRAPHICS AND ACCESSIBLE STARVATION

We created a multi-platform graphics pipeline that manages accessibility support and *accessible starvation.* Our engine is free and open-source and can be used by any team using graphics for any purpose, including block rendering or learning. It uses OpenGL for hardware accelerated 2D and 3D graphics, with WebGL for the web. The same ideas could be applied to other graphics APIs. In a sense, our system abstracts the graphics and combines it with an operating system neutral accessibility model (e.g., the web with SHADOW-DOM and ARIA, Windows desktop with UIA). The reason we did this is because accessibility programming is extremely complex and it should not be presumed or expected that a typical programmer will have the required expertise.

Recall that at a low level, graphics, user interaction, and accessibility messages enter a shared message queue (sometimes called a pump). Messages from any aspect of the application are shared in this queue and thus interleaving issues like frames of animation can interfere in non-intuitive ways. While an imperfect analogy, one might think of this as a fan effect. For example, a particular call in OpenGL might be sandwiched between a mouse event and a request for information from any accessibility or automation technology.

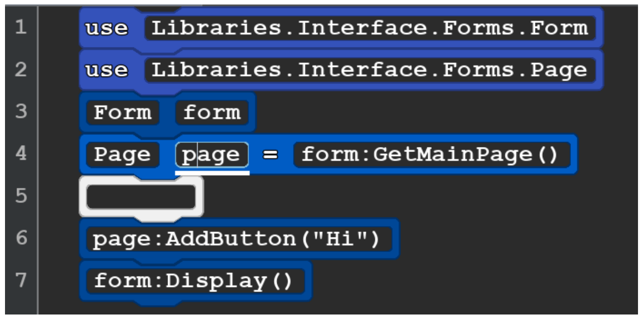
Accessibility technologies were not always designed for graphics and can react in problematic ways, especially the JAWS screen reader because of how we suspect (it is closed source) it manages threading. Users may experience serious delays between interacting with an application and receiving a response from a screen reader. In our testing, we found that each keystroke could take up to 10 seconds before the screen reader speaks in response if OpenGL is used out of the box. This delay, which is complicated in practice, is the heart of what we call *accessible starvation.*

Consider this example to help explain the complexity of the challenge. Screen readers frequently follow-up with requests to gather more information. If a user changes the focus, a screen reader might send a request asking what was changed, which gets interleaved in the queue. If, for example, it identifies the focused element is a tree, it could send further requests regarding the structure of the tree, the number of items, or other properties. This can lead to hundreds or thousands of requests that are *interleaved with frames of animation.* Interleaving is a big issue, but not the only. Some calls are blocking or conflict with others. The only reliable way we have found to locate them, since we have observed no online documentation on this issue, is using professional grade profilers and correlating graphics/accessibility callbacks by hand.

Fortunately, *accessible starvation* can be mitigated. The strategy is to "pump" the queue within frames of animation and to optimize as described above on every platform. We use a configurable target frame rate to set the upper bound and set a minimum delay time per frame as a lower bound. This dedicated polling period allows many requests, responses, and follow-ups to be processed within a single frame and can be turned on or off. Thus, one downside of our approach is that there must be a capped frame rate. Fortunately, after extensive testing with multiple platforms, we found that a

[1]A good example of this mistake is that Zoom on Mac has a second voice speak on top of the screen reader when meeting participants enter or exit.

Figure 1: Blocks for a mobile phone application in dark mode. The application generated is also accessible.



```
use Libraries.Interface.Forms.Form
use Libraries.Interface.Forms.Page
Form form
Page page = form:GetMainPage()

page:AddButton("Hi")
form:Display()
```

time dedicated to message pumping of 0.001 seconds per frame was sufficient to make applications responsive and highly usable.

So far as we can tell, our system works for arbitrary 2D or 3D graphics and allows one to custom inject arbitrary accessibility properties in a universal way. This is very convenient if you need graphics to be accessible. With this technology as a basis, we then set out to make all aspects of blocks, from the blocks themselves to their graphical output pipeline, accessible. This is in contrast to previous work, which predominately, but not exclusively, focuses on issues with input (block interactions) [5, 19, 20, 29, 30].

5 CONTRIBUTION 2: ACCESSIBLE BLOCKS AND ACCESSIBLE OUTPUT

After completing our accessible graphics pipeline, we engaged in a two-year project to think through the design of block languages from the lens of being general purpose and accessible. We looked carefully at issues we suspected excluded students with disabilities at any time from from high school to professional practice. While important, we did not consider below high school in this design. Thus, in addition to building the rendering of the blocks on top of our engine so they are accessible, we identified three problems that were exclusionary of students with disabilities in mainstream block languages today: 1) spatial layout, 2) the editing experience, and 3) the visual output.

5.1 Spatial Layout

First, block languages typically have a spatial and non-linear layout system. A student might drag and drop a block from a left palette to the right hand side of a page. Drag and drop itself is fine for accessibility, so long as alternatives like keyboard support are available. However, when blocks can be at arbitrary spatial positions on the page, focus traversal ordering can be unclear. There may also be organization issues for neuro-divergent people. While one can easily dream up a system for focus traversal spatially (e.g., y coordinate mappings, tabbing between regions) we made all blocks in our system display, top to bottom, like in a normal text editor with line numbers. Figure 1 shows an example of a block program that generates an accessible (with Talkback support) mobile phone application on an Android device in one of the color modes (dark). Code.org similarly uses line numbers in its App Lab editor.

5.2 Editing Experience

Second, block designs vary in how a student edits or creates code. For example, almost all block languages, with notable exceptions for those that have attempted accessibility [5, 19, 20, 29, 30], generally use the mouse. We adapted ideas from all of these approaches for the editor, but especially two. First, Mountapmbeme and Ludi found evidence that the horizontal blocks in blockly were difficult to use for blind students [20], so we allowed expressions to be typed. More evidence on the impact of this decision is needed. Second, we adopted an approach similar to Stride [11], which uses horizontal and vertical cursors for block and character level edits, respectively.

Unique to our version, when landing on a block, we send a UIA Custom call, which triggers braille and speech for the line of code. To trigger this, users press common keys from a text editor (e.g., arrows). Alternatively, they can also explore expansion concepts with hotkeys (e.g., navigation, go to line, Git, hints). This design lets us take into account the fact that students with disabilities vary significantly in how well they know their own accessibility technologies, especially for young children. It also makes the blocks feel like existing technologies they may already know (e.g., text editors). While that is the default experience, many advanced options can be explored and learned. For example, if you are on a block and press tab, we trigger UIA for a text field or box, which prevents students from making syntax errors very analogously to current block languages.

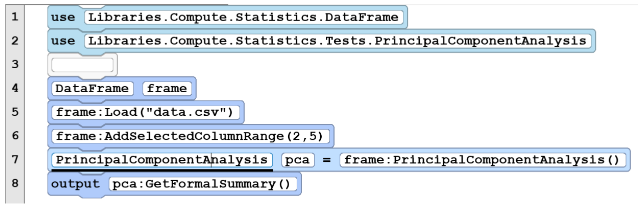
For students that can see, our system would look and feel like a block. For students that are blind, when either of the cursors moves, they hear or feel it. For those with difficulty using the mouse, the mouse is never required. For the keyboard, there are key differences we made with previous work. First, instead of using keys like Stride, we adapted empirical work by Baker *et al.* [2] and Ameer *et al.* [1] that studied navigation and comprehension. In broad strokes, from a keyboard our system feels more like a text editor with optional features like StructJumper (e.g., jump up and down a scope, go to the previous function). We also included navigation features common to many text editors (e.g., go to line, find). Again, our broad design goal was to make the common keys feel like a text editor, but allow more advanced features to be explored and tinkered with.

Finally, when a user deletes a block, we created an algorithm we call *Minimum Semantic Difference.* This approach calculates the region of code to delete that would keep the code syntactically, and with some limitations semantically, correct. While the algorithm is complex and out of scope of this paper, consider an example if statement. If a user deletes an if, what is inside that region is retained and the surrounding scope is adjusted based on the syntax of the language.

In terms of being general purpose, Figure 2 shows blocks for a principal component analysis. We mention this because we have at least tried to think through complex programs as well. A statistician can use our block system to conduct factor analysis in about 7 blocks, using the mouse or keyboard. We now use the block system for professional grade statistical analysis in other academic work.

For those beyond high school age, we created a variety of visual affordances to allow for modification of code. In tools like Scratch, one often has to drag a block to the trash, grab the new one, and drag it back. We offer keyboard affordances for similar operations

Figure 2: Blocks for a Principal Component Analysis. Charts and graphs are available as well and generate accessibly



```
use Libraries.Compute.Statistics.DataFrame
use Libraries.Compute.Statistics.Tests.PrincipalComponentAnalysis

DataFrame frame
frame:Load("data.csv")
frame:AddSelectedColumnRange(2,5)
PrincipalComponentAnalysis pca = frame:PrincipalComponentAnalysis()
output pca:GetFormalSummary()
```

philosophically in range of Typing Blocks [14] and later adapted by Switch blocks [15], except that in our case the use of the vertical cursor provides temporary and small switches to raw text for expressions that we call freeform blocks. Our blocks use graphics shaders for visual rendering of hints and error and these are injected into the accessibility system as well. Note that a student in high school may not benefit from hints or code completion, but Peterson *et al.* conducted a randomized controlled trial testing whether such features were helpful for college students, with positive results [23]. The point is we hide more advanced features by default, but they can be learned and tinkered with as a student learns.

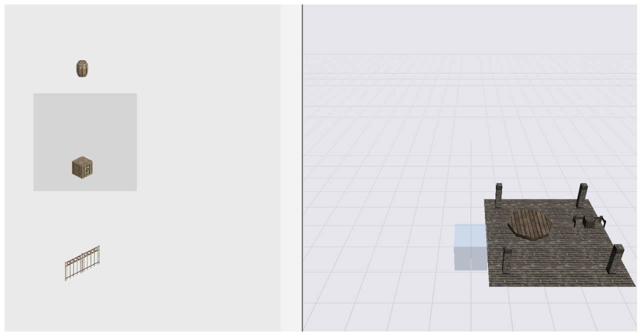
5.3 Visual Output

Finally, one major challenge is not just the blocks, but all the visuals that exist around them, like dancing cats in Scratch or 3D graphics in Alice [22]. Figure 3 shows our 2D/3D drag and drop and canvas palette that can be used visually or with a screen reader/braille. We used what we coin as the *two-dummy solution* for drag and drop editing of visuals in the canvas. In 2-dummy style, we switch the focus between two fake objects that are not on the screen and do not exist, but that are injected into the accessibility tree on a per platform basis. In effect, we lie to the accessibility system in a controlled manner. Figure 3 shows an image of this editor. A student using braille would feel the x, y, and z coordinate if the box cursor was over negative space. If, instead, the cursor was an item, the name would be mentioned first. Like the block editor, we emphasized using common keys to manipulate the 3D transparent box (which is the "cursor"), like the arrows, while providing advanced navigation features that students can again tinker with.

The purpose is that if we inject fake properties at each point, we can trigger screen reader responses without a mutual agreement—accessible graphics becomes a technical problem and not a social one. For example, if one dummy is actually a 3D camera, we can strategically inject properties that tell a blind user what the camera is showing. If the dummy is an ogre, we can express what it is. Focus changes happen through switching the dummy on focus change events but keeping the entire visual scene to only 2 items. This is important to consider in graphics because the number of items on screen can get very large (e.g., every polygon in a 3D terrain). We suspect 2-dummy is sufficient for accessibility needs for many, but not all, kinds of visual editors and works identically in 2D and 3D.

Since our solution provides control of the graphics and accessibility pipeline, we can inject some fun into a blind user's experience. We do not have to be dry and say "ogre." We can instead say, "blood thirsty ogre at x, y, and z. Watch out or it will get ya!" while also

Figure 3: This image shows a mouse/keyboard accessible palette along with the canvas showing a small medieval scene. Screen reader and braille users hear or feel information through a 3D selection cursor, shown visually as a transparent box. On empty space, only x, y, and z coordinates are stated by default.



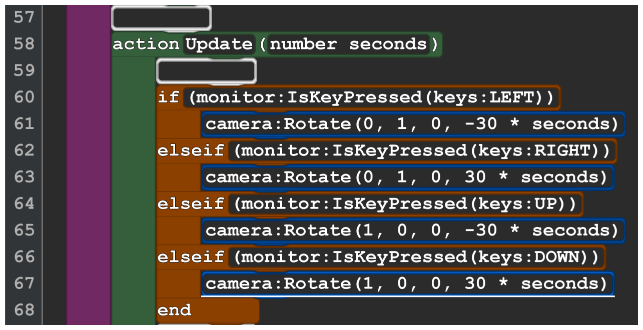
rendering it visually in any creative way we wish. Or, if we simply want to, we could provide context clues, location information, camera information, Easter eggs, or really anything we want. When our application boots, for example, users on our loading screen hear, see, or feel, phrases like, "Hiding in the corner. Eating all your breakfast," or "Suddenly ... kitties." There is no need to make loading screen messages serious and we want especially screen reader users listening to know they were our privilege, not our burden. As a community, we can and should do better than legal compliance: *we should try to bring some real magic to students.*

6 CONTRIBUTION 3: FOCUS GROUPS ON ACCESSIBLE BLOCKS

We conducted three formative focus groups in July of 2023 to help identify our successes and failures. The first was online with 5 adults (4 high school teachers and one adult braille user). The goal here was to help find bugs and fix problems before talking to students. With the second, we engaged with 21 students with a variety of disabilities in two separate focus groups in person. Disabilities varied and included ADHD (11), autism (18), brain (1), health (8), hearing (5), learning (12), mental health (8), mobility (17), and vision (6). Many students in our sample had more than one disability. The goal there was direct observation and to learn from students' lived experience. One interesting finding from the first focus group with teachers was in regard to braille. There was only one braille user in the group, but he noticed JAWS would speak the horizontal cursor on a block, but not issue braille. The NVDA screen reader would do both. After an investigation, we found this was a bug in the JAWS screen reader. We contacted the company that makes JAWS and informed them of the bug. We focus the remainder of this section on student feedback.

Next, we ran two focus group sessions with high school aged participants in person. For each, there was about an hour of observation while students used blocks in a data science lesson and then a half hour for discussion. Several students noted low contrast in dark mode, but none objected to light mode. We had copied our

Figure 4: Blocks in dark and thin mode to ease magnification.



```
action Update(number seconds)

    if (monitor:IsKeyPressed(keys:LEFT))
        camera:Rotate(0, 1, 0, -30 * seconds)
    elseif (monitor:IsKeyPressed(keys:RIGHT))
        camera:Rotate(0, 1, 0, 30 * seconds)
    elseif (monitor:IsKeyPressed(keys:UP))
        camera:Rotate(1, 0, 0, -30 * seconds)
    elseif (monitor:IsKeyPressed(keys:DOWN))
        camera:Rotate(1, 0, 0, 30 * seconds)
    end
```

dark mode colors from Scratch's new system as a first attempt. After the focus groups, while we thought we had, we re-tested the colors finding all of the contrasts out of accessibility compliance. While an easy fix, the realization was humbling and reminded us just how easy it is to make accessibility mistakes.

Students also made suggestions around errors. Since our environment involves typing, it means mistakes are constrained but possible. By default, our environment pulsed a block red (or through sound/touch). However, students observed that it was sometimes difficult to determine exactly where in a line an error was without additional locality. We suspect in-block highlighting, or something similar, may have helped. One participant used an example of spellchecking in a document (e.g., red underlines in a code editor). Another option could be an informational popup.

Another issue was case sensitivity in code completion. One student with a mobility-related disability heavily favored one hand, making simultaneous key striking difficult. Removing case sensitivity for all typing features would have made it easier for one hand typing and we have since made adjustments.

For low vision students, the most crucial problem we observed was the height of the blocks. Traditional blocks are quite wide and tall and we made ours look similar to Scratch. However, low vision students scaled the size of the blocks so large that, because of padding, they could only see 2-3 lines without scrolling. We suspect block languages may benefit from a "thin" mode for students with low vision and have since added the feature.

While the student focus groups overwhelmingly expressed praise for Quorum Blocks, we have highlighted our failures here to provide the community feedback on the kinds of issues that come up for various disabilities. All told, we made 45 unique changes to the system, which came from teachers or students. As a final example, Figure 4 shows a version with thin mode turned on. This figure highlights a more complex example with scoping and functions. The bottom-line is that users with, or without, disabilities can use our tool in practice.

7 DISCUSSION

We created a new design for accessible blocks, built on an accessible graphics platform. While blocks are the input, they often render graphics or visualization, have user interactions, and let students drag and drop images or 3D models in a canvas. Such a technology could make it possible, unlike previous work that adapted high school curriculum like Computer Science Principles to text-based languages [34], to retain all graphics and the blocks. Students also create applications and what they invent is accessible by default.

We imagine that one reasonable criticism will be that it is difficult to boil down accessibility work to an obvious end-point, like "how accessible and general is this approach?" We used focus groups with teachers and students as a formative gut check, but this is hardly a direct answer. That said, users with a variety of disabilities, including screen reader users, were able to use the blocks to make charts in data science and could interact with them without difficulty. We also found the focus groups to be useful and were surprised to find a bug in a professional grade screen reader, to learn our blocks would need a "thin" mode, or how specifically to improve the system for one handed typing.

We also need to be clear that making block languages general purpose is not easy. Any kind of general purpose design is hard and the impact of programming language design on people with different demographics is still poorly understood in the literature (e.g., gender, age, natural language [8], disability). However, environments made only for people with disabilities, we observe through experience, sometimes are not maintained. Similarly, we were hesitant to focus only on k-12 students. After all, if we did so, knowing full well that many languages in college and beyond are also not accessible, students would still be excluded. Our approach for general purpose thus has obvious challenges, but we argue it needs to be the goal. A student with a disability cannot "transfer learning" if the next step is also not accessible.

8 CONCLUSIONS

We have created a novel block language designed for accessibility. We found that graphics pipelines can meaningfully coordinate with accessibility technologies through the operating system, so long as issues like accessible starvation are strategically mitigated. This allows the input (the blocks) and the output (the editor) to be accessible and we designed our system on top of this technology for 2D and 3D, data science, and other general purpose areas. Focus groups on our design were positive and provided insight into block-related issues especially for those using one hand to type, screen reader and braille users, and those using large magnification.

9 ACKNOWLEDGMENT

This work is supported by the National Science Foundation under Grant Numbers NSF# 2121993, 2122189, 2106392 and 2048356.

REFERENCES

[1] Ameer Armaly, Paige Rodeghero, and Collin McMillan. 2018. A Comparison of Program Comprehension Strategies by Blind and Sighted Programmers. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 788. https://doi.org/10.1145/3180155.3182544

[2] Catherine M. Baker, Lauren R. Milne, and Richard E. Ladner. 2015. StructJumper: A Tool to Help Blind Programmers Navigate and Understand the Structure of Code. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems* (Seoul, Republic of Korea) *(CHI '15)*. Association for Computing Machinery, New York, NY, USA, 3043–3052. https://doi.org/10.1145/2702123.2702589

[3] Luiz Barboza, Rafael Mello, Micah Modell, and Erico Souza Teixeira. 2023. Blockly-DS: Blocks Programming for Data Science with Visual, Statistical, Descriptive and Predictive Analysis. In *LAK23: 13th International Learning Analytics and Knowledge Conference (LAK2023)*. Association for Computing Machinery, New York, NY, USA, 644–649. https://doi.org/10.1145/3576050.3576097

[4] Austin Cory Bart, Javier Tibau, Dennis Kafura, Clifford A. Shaffer, and Eli Tilevich. 2020. Design and Evaluation of a Block-based Environment with a Data Science Context. *IEEE Transactions on Emerging Topics in Computing* 8, 1 (Jan. 2020), 182–192. https://doi.org/10.1109/TETC.2017.2729585 Conference Name: IEEE Transactions on Emerging Topics in Computing.

[5] Logan B. Caraco, Sebastian Deibel, Yufan Ma, and Lauren R. Milne. 2019. Making the Blockly Library Accessible via Touchscreen. In *Proceedings of the 21st International ACM SIGACCESS Conference on Computers and Accessibility* (Pittsburgh, PA, USA) *(ASSETS '19)*. Association for Computing Machinery, New York, NY, USA, 648–650. https://doi.org/10.1145/3308561.3354589

[6] CSTA. 2023. CSTA CSAccess Working Group. Accessed on August 7th, 2023 from https://csteachers.org/the-csaccess-working-group/.

[7] Google. 2021. Android Accessibility Package. Accessed on January 22nd, 2022 from https://developer.android.com/reference/android/view/accessibility/package-summary.

[8] Felienne Hermans. 2020. Hedy: A Gradual Language for Programming Education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research* (Virtual Event, New Zealand) *(ICER '20)*. Association for Computing Machinery, New York, NY, USA, 259–270. https://doi.org/10.1145/3372782.3406262

[9] Apple Inc. 2018. Accessibility Programming Guide for OS X: The OS X Accessibility Model. Accessed on January 22nd, 2022 from https://developer.apple.com/library/archive/documentation/Accessibility/Conceptual/AccessibilityMacOSX/OSXAXmodel.html.

[10] Apple Inc. 2022. Apple UI Accessibility Developer Documentation. Accessed on January 22nd, 2022 from https://developer.apple.com/documentation/objectivec/nsobject/uiaccessibility.

[11] Michael Kölling, Neil C. C. Brown, Hamza Hamza, and Davin McCall. 2019. Stride in BlueJ – Computing for All in an Educational IDE. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 63–69. https://doi.org/10.1145/3287324.3287462

[12] Richard E. Ladner and Maya Israel. 2016. "For All" in "Computer Science for All". *Commun. ACM* 59, 9 (aug 2016), 26–28. https://doi.org/10.1145/2971329

[13] Maryland Legislature. 2022. Local School Systems - Equivalent Access Standards - Digital Tools (Equivalent and Nonvisual Access Accountability Act for K-12 Education). Accessed on August 7th, 2023 from https://mgaleg.maryland.gov/mgawebsite/Legislation/Details/SB0617?ys=2022RS.

[14] Terrance Liang. 2019. *Typeblocking : keyboard integration with block programming in StarLogo Nova.* Master's thesis. MIT.

[15] Yuhan Lin, David Weintrop, and Jason McKenna. 2023. Switch Mode: A Visual Programming Approach for Transitioning from Block-Based to Text-Based Programming. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2* (Toronto ON, Canada) *(SIGCSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1262. https://doi.org/10.1145/3545947.3573235

[16] J. Nathan Matias, Sayamindu Dasgupta, and Benjamin Mako Hill. 2016. Skill Progression in Scratch Revisited. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. Association for Computing Machinery, New York, NY, USA, 1486–1490. https://doi.org/10.1145/2858036.2858349

[17] Microsoft. 2020. UI Automation - Win32 apps | Microsoft Docs. Accessed on January 22nd, 2022 from https://docs.microsoft.com/en-us/windows/win32/winauto/entry-uiauto-win32.

[18] Alexandra Milliken, Veronica Cateté, Ally Limke, Isabella Gransbury, Hannah Chipman, Yihuan Dong, and Tiffany Barnes. 2021. Exploring and Influencing Teacher Grading for Block-based Programs through Rubrics and the GradeSnap Tool. In *Proceedings of the 17th ACM Conference on International Computing Education Research (ICER 2021)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3446871.3469762

[19] Lauren R. Milne and Richard E. Ladner. 2019. Blocks4All: Making Blocks-Based Programming Languages Accessible for Children with Visual Impairments. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 1290. https://doi.org/10.1145/3287324.3293755

[20] Aboubakar Mountapmbeme, Obianuju Okafor, and Stephanie Ludi. 2022. Accessible Blockly: An Accessible Block-Based Programming Library for People with Visual Impairments. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '22)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3517428.3544806

[21] Obianuju Okafor and Stephanie Ludi. 2022. Voice-Enabled Blockly: Usability Impressions of a Speech-Driven Block-Based Programming System. In *Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility* (Athens, Greece) *(ASSETS '22)*. Association for Computing Machinery, New York, NY, USA, Article 64, 5 pages. https://doi.org/10.1145/3517428.3550382

[22] Randy Pausch. 2008. Alice: A Dying Man's Passion. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (Portland, OR, USA) *(SIGCSE '08)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/1352135.1352137

[23] Pujan Petersen, Stefan Hanenberg, and Romain Robbes. 2014. An Empirical Comparison of Static and Dynamic Type Systems on API Usage in the Presence of an IDE: Java vs. Groovy with Eclipse. In *Proceedings of the 22nd International Conference on Program Comprehension* (Hyderabad, India) *(ICPC 2014)*. Association for Computing Machinery, New York, NY, USA, 212–222. https://doi.org/10.1145/2597008.2597152

[24] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. *Commun. ACM* 52, 11 (nov 2009), 60–67. https://doi.org/10.1145/1592761.1592779

[25] Nico Ritschel, Felipe Fronchetti, Reid Holmes, Ronald Garcia, and David C. Shepherd. 2022. Enabling end-users to implement larger block-based programs. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings (ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 347–349. https://doi.org/10.1145/3510454.3528644

[26] Nico Ritschel, Vladimir Kovalenko, Reid Holmes, Ronald Garcia, and David C. Shepherd. 2022. Comparing Block-Based Programming Models for Two-Armed Robots. *IEEE Transactions on Software Engineering* 48, 5 (May 2022), 1630–1643. https://doi.org/10.1109/TSE.2020.3027255 Conference Name: IEEE Transactions on Software Engineering.

[27] Timothy Roden and Ian Parberry. 2005. Designing a Narrative-Based Audio Only 3D Game Engine. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology* (Valencia, Spain) *(ACE '05)*. Association for Computing Machinery, New York, NY, USA, 274–277. https://doi.org/10.1145/1178477.1178525

[28] Christopher Scaffidi and Christopher Chambers. 2012. Skill Progression Demonstrated by Users in the Scratch Animation Environment. *International Journal of Human–Computer Interaction* 28, 6 (2012), 383–398. https://doi.org/10.1080/10447318.2011.595621 arXiv:https://doi.org/10.1080/10447318.2011.595621

[29] Emmanuel Schanzer, Sina Bahram, and Shriram Krishnamurthi. 2019. Accessible AST-Based Programming for Visually-Impaired Programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 773–779. https://doi.org/10.1145/3287324.3287499

[30] Emmanuel Schanzer, Sina Bahram, and Shriram Krishnamurthi. 2020. Adapting Student IDEs for Blind Programmers. In *Proceedings of the 20th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '20)*. Association for Computing Machinery, New York, NY, USA, Article 23, 5 pages. https://doi.org/10.1145/3428029.3428051

[31] David Shepherd, Patrick Francis, David Weintrop, Diana Franklin, Boyang Li, and Afsoon Afzal. 2018. [Engineering Paper] An IDE for Easy Programming of Simple Robotics Tasks. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 209–214. https://doi.org/10.1109/SCAM.2018.00032

[32] Ana Luisa Veroneze Solórzano and Andrea Schwertner Charão. 2021. BlocklyPar: from sequential to parallel with block-based visual programming. In *2021 IEEE Frontiers in Education Conference (FIE)*. 1–8. https://doi.org/10.1109/FIE49875.2021.9637261 ISSN: 2377-634X.

[33] Viktor Stadler and Helmut Hlavacs. 2018. Blind Adventure - A Game Engine for Blind Game Designers. In *Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play* (Melbourne, VIC, Australia) *(CHI PLAY '18)*. Association for Computing Machinery, New York, NY, USA, 503–509. https://doi.org/10.1145/3242671.3242703

[34] Andreas Stefik, Richard E. Ladner, William Allee, and Sean Mealin. 2019. Computer Science Principles for Teachers of Blind and Visually Impaired Students. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (Minneapolis, MN, USA) *(SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 766–772. https://doi.org/10.1145/3287324.3287453

[35] Javier Torrente, Ángel Serrano-Laguna, Ángel Aguado, Pablo Moreno Ger, and Baltasar Fernández-Manjón. 2014. Development of a Game Engine for Accessible Web-Based Games. 107–115. https://doi.org/10.1007/978-3-319-12157-4_9

[36] Mauricio Verano Merino, Juan Pablo Sáenz, and Ana María Díaz Castillo. 2022. Suppose You Had Blocks within a Notebook. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT 2022)*. Association for Computing Machinery, New York, NY, USA, 57–62. https://doi.org/10.1145/3563836.3568728

[37] David Weintrop. 2019. Block-Based Programming in Computer Science Education. *Commun. ACM* 62, 8 (jul 2019), 22–25. https://doi.org/10.1145/3341221