An Extension-Based Accessibility Framework for Making Blockly Accessible to Blind and Low-Vision Users

Rubel Hassan Mollik
The Department of Computer Science and Engineering
University of North Texas
Denton, USA
rubelhassanmollik@my.unt.edu

Vamsi Krishna Kosuri
The Department of Computer Science and Engineering
University of North Texas
Denton, USA
vamsikrishnakosuri@my.unt.edu

Hans Djalali
The Department of Computer Science and Engineering
University of North Texas
Denton, USA
HansDjalali@my.unt.edu

Stephanie Ludi
The Department of Computer Science and Engineering
University of North Texas
Denton, USA
stephanie.ludi@unt.edu

Aboubakar Mountapmbeme
The Department of Computer Science and Engineering
University of North Texas
Denton, USA
aboubakarmountapmbeme@my.unt.edu

Abstract

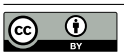
Block-based programming environments (BBPEs) such as Scratch and Code.org are now widely used in K-12 computer science classes, but they remain mostly inaccessible to blind or visually impaired (BVI) learners. A major problem is that prior accessibility solutions have relied on modifications to the Blockly library, making them difficult to apply in existing BBPEs and thereby limiting adoption. We present an Extension-based Accessibility Framework (EAF) to make BBPEs accessible for BVI students. The framework uses a modular architecture that enables seamless integration with existing Blockly-based BBPEs. We present an innovative three-dimensional (3D) hierarchical navigation model featuring stack labeling and block numbering, mode-based editing to prevent accidental modifications, and WAI-ARIA implementation to ensure compatibility with external screen readers. We evaluated our approach by integrating the EAF framework into two BBPEs (covering 177 test cases) and conducting semi-structured interviews with four participants using VoiceOver, JAWS, and NVDA. Participants reported clearer spatial orientation and easier mental model formation compared to default Blockly keyboard navigation. EAF shows that modular architecture can provide comprehensive accessibility while ensuring compatibility with existing BBPEs.

CCS Concepts

• **Human-centered computing** → **Accessibility systems and tools**; *Accessibility technologies*; *Empirical studies in accessibility*; • **Software and its engineering** → Software prototyping.

Keywords

Block-based programming, Keyboard Navigation, Screen Reader, Accessibility, Blockly



UISE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2401-5/26/04
https://doi.org/10.1145/3786169.3788398

ACM Reference Format:
Rubel Hassan Mollik, Vamsi Krishna Kosuri, Hans Djalali, Stephanie Ludi, and Aboubakar Mountapmbeme. 2026. An Extension-Based Accessibility Framework for Making Blockly Accessible to Blind and Low-Vision Users. In *1st International Workshop on User Interface and Experience for Software Engineering (UISE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3786169.3788398

1 Introduction

The Computer Science for All (CSForAll) [26] initiative has boosted participation in computer science (CS) and computational thinking (CT) education among K-12 students. Through this initiative, block-based programming has become widespread in K-12 CS courses. Block-based programming [32] replaces the complex syntax of text-based programming with visual, drag-and-drop blocks, making learning programming more accessible for novices. Students find visual programming more engaging and learn fundamental programming logic and computational thinking faster [33]. Platforms such as Scratch [25], Microsoft MakeCode [4], and Code.org [6] have emerged as block-based programming environments (BBPEs), which have become an integral part of K-12 CS curricula [15].

However, the BBPEs are not accessible to learners who are blind or visually impaired (BVI) due to their over-reliance on visual elements and animations [27]. Moreover, drag-and-drop operations like moving blocks involve mouse-based interaction and visual feedback, which is challenging for BVI learners [20]. As a result, BVI learners face barriers to participating in CS courses, which leaves them behind their sighted peers [21]. Since many popular BBPEs, such as Scratch, MakeCode, and Code.org, are built on the Blockly library, making it accessible can benefit all Blockly-based platforms.

Google Blockly recently added a default keyboard navigation [11] that offers limited support for keyboard interaction. However, it remains in an experimental stage and lacks screen reader compatibility, making it inaccessible to BVI learners. While promising prototypes such as Accessible Blockly [22] and GridWorld [30] have shown potential for keyboard-based alternative navigation for BVI learners, these efforts primarily focus on small-scale programs and

specific navigation tasks, with limited code editing support. Furthermore, these approaches inherit architectural constraints by modifying the Blockly codebase, which impedes interoperability and prevents integration into existing Blockly-based environments. Several implementations also rely on internal speech synthesis [8, 30], which conflicts with screen readers, creating compatibility issues.

This work is motivated by the need to develop an accessible solution for BBPEs that supports structured navigation and editing of complex programs while maintaining architectural modularity. We propose a novel accessibility framework that can be seamlessly integrated into Blockly-based systems without modifying the core codebase. This approach provides a scalable, interoperable, and screen reader–compatible solution, improving accessibility of BBPEs.

The main contributions of this paper are as follows:

• We present an Extension-based Accessibility Framework (EAF) that integrates accessibility features into Blockly-based environments. Unlike previous methods that require forking the Blockly repository, EAF extends across platforms and Blockly versions while maintaining compatibility and structural stability.

• We introduce a three-dimensional (3D) navigation and editing model with stack labels and sequential block numbers. The design establishes a predictable structural navigation and editing experience for BVI learners.

• We validate *EAF*'s viability through integration testing with Blockly-based platforms, WCAG 2.1 accessibility compliance assessment, and expert evaluation. Our results demonstrate that EAF enables accessible block-based programming while working with external screen readers (NVDA, JAWS, VoiceOver).

2 Related Work

Extensive research has been conducted to improve the accessibility of BBPEs for BVI individuals. Previous studies explored approaches that require Blockly code modification [8, 22, 30], and extension-based architectures remain underexplored even though they have potential for modularity and maintainability [17].

Google initially offered a text-based Blockly separately for BVI users, replacing the visual drag-and-drop block UI with a text-only interface [16]. Ludi et al. [17] then proposed an alternative to the visual drag-and-drop user interface, adding keyboard input and screen reader compatibility. This effort suggested adding functionalities via files that could be applied to Blockly-based projects. Accessible Blockly [22] was developed following this approach, augmenting the Blockly codebase, which demonstrated success in navigating block-based code using keyboard and screen reader support. Google later added an experimental keyboard navigation in Blockly [11], which makes blocks navigable using a keyboard but has no support for audio feedback or screen readers. Das et al. [8] an accessible BBPE by modifying Blockly with synthetic speech, which does not provide audio feedback for all interactions and has no support for program editing. Moumita et al. [30] took a similar approach to introduce a web application that implements Code.org's Maze Game puzzles with a minimal set of blocks.

Other researchers explored creating entirely new systems or using fundamentally different design principles to achieve accessibility [5, 18, 28]. Blocks4All [18] introduced a successful touchscreen-based BBPE specifically designed for children with visual impairments that requires specific hardware (iPad) and a Dash robot for tangible output. However, the application is not suitable for sighted learners and is incompatible with mainstream BBPEs used by K-12 learners. Similarly, OctoStudio [5] was designed with tinkerability for the Android platform, while it remains in progress. In contrast, Quorum Blocks [28] proposes a new accessible block language incorporating a hardware-accelerated graphics pipeline, but it is not purely block-based.

As shown in the Table 1, prior studies primarily employ either Blockly library modifications or new systems. These approaches face fundamental trade-offs between architectural integration and full accessibility. Blockly modification creates isolated silos and version dependencies, blocking integration with mainstream BBPEs, while new systems differ from K-12 BBPE practices. Moreover, current implementations lack structured navigation models and provide only partial support for keyboard-only interaction and screen reader feedback. These gaps underscore the need for a modular architecture with comprehensive accessibility support with full keyboard navigation and external screen readers support that integrates with existing BBPEs. Our work addresses this need.

3 Design Goals

Based on the identified key limitations of accessibility solutions for BBPEs discussed in Section 2, we have established five key design goals for the EAF that are outlined below.

Structured Navigation. The design should provide structured navigation enabling users to traverse program structures efficiently, aligning with mental models of program structure and block hierarchy. All blocks and elements should be keyboard accessible.

Screen Reader Compatibility. All visual information and interactive states should be conveyed through external screen readers, avoiding built-in synthetic speech that conflicts with users' configured assistive technologies. The design must follow WAI-ARIA [7] roles, states, and live-region policies to prevent duplicate announcements and ensure focus stays intact.

Fully Accessible Editing. Keyboard interactions should provide equivalents for all visual drag-and-drop operations, ensuring mouse-keyboard functional parity.

Inclusive Design. The design should maintain parity between sighted and BVI learners, ensuring that features integrate seamlessly without degrading the experience of any user group.

Architectural Modularity. The framework should seamlessly integrate with existing Blockly-based platforms through a modular architecture, avoiding Blockly library modifications or platform-specific forks. This approach ensures compatibility across Blockly versions and diverse platforms (e.g., Scratch, Code.org).

4 Extension-Based Accessibility Framework

4.1 Architecture

Based on the design goals stated in Section 3, we propose an Extension-Based Accessibility Framework (EAF) for Blockly-based

Table 1: Comparison of Accessible Block-Based Programming Approaches

System	Architecture	Navigation	Editing	Testing	AT Strategy	Platform	Evaluation
Google Text Blockly [16]	Blockly Modification	Text-based	N/A	N/A	Screen Reader	Web	Not reported
Ludi et al. [17]	Design Proposal	Full Keyboard	Limited	N/A	Screen Reader	Web	No
Accessible Blockly [22]	Blockly Modification	Partial Keyboard	Mostly	Limited	Screen Reader	Web	User study (n=12)[1]
Das et al. [8]	Blockly Modification	Partial Keyboard	Limited	N/A	Synth Speech	Web	User study (n=4)[2]
Tabassum et al. [30]	Blockly Modification	Partial Keyboard	Limited	Limited	Synth Speech	Web	User study (n=18)[3]
Blocks4All [18]	New System	Touch	Mostly	Yes	iOS VoiceOver	iPad + Robot	User study (n=6)[4]
OctoStudio [5]	New System	Touch	No	Yes	Mobile SRs	Mobile	In progress
Quorum Blocks [28]	New Language	Full	Full	Yes	Screen Reader	Cross-platform	User study (n=26)[5]
EAF (This Work)	**Extension**	**Full Keyboard**	**Full**	**Yes**	**Screen Reader**	**Web**	**Expert eval (n=4)**

[1] One VI student and 17 sighted students. [2] 4 BVI participants. [3] 12 blind/low-vision programmers. [4] 5 children with visual impairments (ages 5–10) and 1 TVI. [5] Focus group 1: 5 adults; focus group 2: 21 high school students.

programming environments. The architecture follows four principles: (1) The framework integrates seamlessly with existing BBPEs without library modifications, ensuring stability. (2) The framework provides WAI-ARIA compliant audio feedback for screen reader compatibility. (3) The framework provides configurable extension points allowing adopting platforms to customize keyboard shortcuts, screen reader verbosity, and color schemes through programmatic API. (4) The framework supports both pointing devices and keyboard interaction, enabling users to choose their preferred input method without compromising functionality.

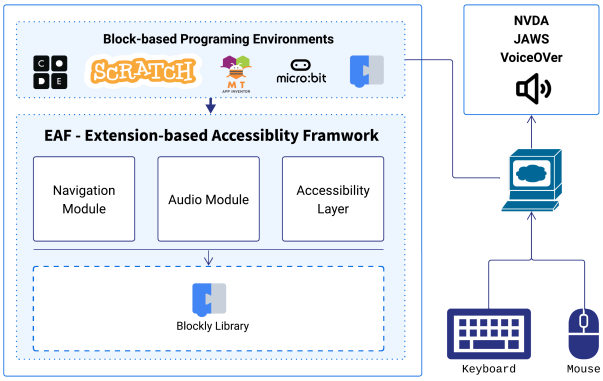


Figure 1: Extension-based Accessibility Framework (EAF) Architecture

The Figure 1 illustrates the architecture of our framework. The framework is composed of four primary, interconnected components that leverage the browser environment.

Blockly Library: Blockly library [12] is used as an association, instead of using it as a dependency of the framework. Our framework shares the same Blockly core semantics and components such as Block, Connection, Input, Field, Toolbox, and Workspace. This approach ensures compatibility with existing block-based platforms that are built on Blockly.

Navigation Module: This module implements an alternative navigational model to support drag-and-drop-like operations via keystrokes and provides comprehensive support for keyboard-only interaction. It also defines standardized keyboard shortcuts for common operations, including block insertion, movement, and traversal. It includes a programmatic API such as custom cursors, markers, and event listeners for extending navigational capabilities.

Audio Module: The audio module generates voice feedback and auditory cues as an alternative output for each keyboard interaction during programming. It monitors user interactions via event bus and provides context-based audio feedback. It applies WAI-ARIA roles, states, and properties for seamless integration with VoiceOver, JAWS, and NVDA, providing semantic information about workspace elements.

Accessibility Layer: The framework introduces a suite of accessibility features through its accessibility layer. This layer implements features such as stack labeling, block numbering, and stack jumping functionality to improve accessibility of block-based programming understanding to BVI users to match their sighted peers. These features will help to improve code comprehension and enable faster navigation for BVI users. The accessibility layer provides well-defined extension points, allowing adopting platforms to implement additional accessibility features into Blockly tailored to their specific user needs and domain contexts.

4.2 Implementation

The EAF is designed following the JavaScript plugin architectural pattern [9, 29], which enables shippable artifacts to host Block-based platforms. The framework is implemented using standard web technologies to ensure broad compatibility and ease of adoption. The core technology stack leverages JavaScript (ES6+) as the programming language, SVG for rendering accessible visual elements within the Blockly workspace, HTML5 for structural markup, and CSS3 for styling and visual feedback. This technology stack aligns with Blockly's native implementation, facilitating seamless integration without introducing additional runtime dependencies. The framework utilizes npm (Node Package Manager) [1] as the

build and distribution system, enabling developers to integrate EAF into projects through standard package management workflows.

Screen reader support is implemented through comprehensive WAI-ARIA [7] annotations. The framework dynamically generates and updates *aria-label* attributes based on the current state and context of workspace elements such as blocks, fields, inputs, and connections. ARIA live regions are employed to announce dynamic changes, such as block insertions, deletions, and connection changes, ensuring that screen reader users receive immediate feedback for their actions. The framework has been tested and validated with major screen readers, including VoiceOver (macOS) [3], JAWS (Windows) [10], and NVDA (Windows) [23].

5 EAF-based Blockly Design

5.1 Code Navigation and Editing Model

The navigational models proposed in earlier works, shown in Table 1 are semi-structured or incompletely designed. A structured navigational model is essential to support predictive navigation and better code comprehension, editing, and debugging.

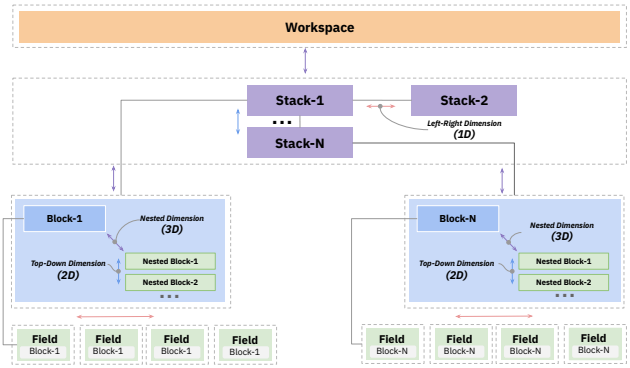


Figure 2: 3D Navigation Model Abstraction and Layers

3D Navigational Model. We have designed a novel Three-Dimensional (3D) navigation model for BVI learners based on programming abstractions such as loops, conditions, variables, etc, and layers such as workspace, stack, blocks, and fields in the Blockly workspace shown in Figure 2. The layer-based traversal and abstractions create contextual boundaries during navigation, which help users orient and maintain context while interacting non-visually with complex code structures [2]. This navigation approach ensures less cognitive load while aligning with the cognitive principles of traditional programming [13].

The 3D model defines three navigation dimensions: [1D] left-right (for horizontal movement within the same hierarchical level); [2D] up-down (for vertical movement within the same hierarchical level); and [3D] in-out (for traversing nesting levels). To implement this model through keyboard interactions, we have adopted the *WASDFQ* key configuration, where *WASD* keys control spatial navigation left, right, up, and down, respectively, and *F* and *Q* keys control nesting in and out, respectively. We selected *WASD* keys over traditional arrow keys because they frequently conflict with default screen reader shortcuts [17] and are consistent with previous attempts on accessible programming solutions [22, 24, 30]. Moreover, the spatial proximity of WASD and FQ keys enables one-handed operation, which may benefit learners with motor impairments by reducing the physical effort for navigation [28]. Figure 3 illustrates the 3D navigation model.

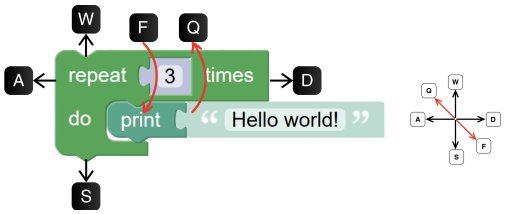


Figure 3: 3D Navigation Model with Keys

Editing Mode. Our design uses a mode-based editing approach in which users can switch between editing and navigation modes explicitly. Editing mode activates when a user selects a specific block, at which point modifications or insertions can be performed. This "select first, then edit" mechanism reduces cognitive load by clearly separating navigation and editing tasks [18], which keeps BVI learners from making unintentional modifications [22]. The editing mode provides keyboard-only alternatives for drag-and-drop operations (attach, detach, cut, copy, paste, delete). In order to reduce distractions, the model uses context-aware editing to show only compatible blocks in the toolbox and hide incompatible ones (Figure 4). Editing mode can be toggled using the *E* key.

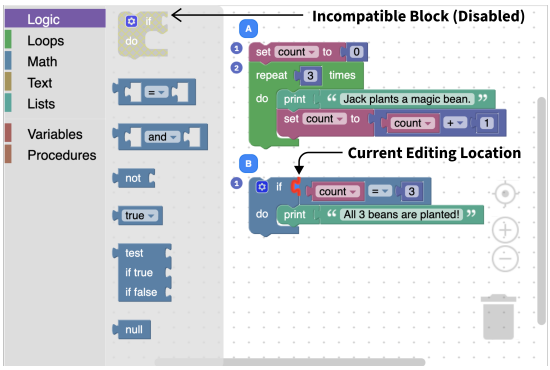


Figure 4: Context-aware Edit Mode

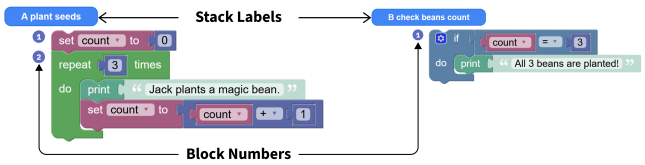


Figure 5: EAF Blockly Workspace with stack labels and block numbers

5.2 Accessible Features

We have designed a few features to improve the accessibility of block-based programs through our proposed framework. These implementations reside within the accessibility layer of the EAF framework. These features serve as exemplars that developers and researchers of Blockly-based platforms can adapt and extend their systems to improve accessibility according to the needs.

Stack Labeling and Block Numbers. The framework introduces stack labeling, assigning unique identifiers to each stack (a group of connected blocks separated from other blocks). By default, stacks are labeled alphabetically (A, B, C, etc.), with customizable labels reflecting semantic purpose (e.g., "main loop"). Block numbers within stacks enable unique block identification. These labels and numbers allow BVI users to reference specific blocks or stacks during navigation. Additionally, stack jumping allows direct navigation to any labeled stack from anywhere in the workspace. For example, if a user is on Stack A and presses *Alt+D*, the cursor jumps from Stack A to Stack D, significantly improving navigation efficiency in complex programs.

Navigational Assistant. We developed a navigational assistant that provides predictive movement information. When activated with *Shift+H*, the assistant informs users where their cursor will move if they press specific keys. This feature is helpful when users are unsure of their current context or have forgotten shortcuts, helping them learn shortcuts and navigate with confidence.

Cursor Locator A common frustration for BVI learners is losing track of cursor position after accidentally pressing keys or touching the mouse [22, 31]. To address this, we have implemented a cursor locator feature. When users press the *C* key, it provides voice feedback of the current position with the current and surrounding blocks' information.

Zoom Controls Zoom controls are not currently accessible via keyboard in Blockly [8], which presents a significant barrier for low-vision users. Our framework implements keyboard interactions for zoom controls: the *+* key zooms in, the *-* key zooms out, and the *0* key resets the view to the default zoom level.

5.3 Standard Keyboard Shortcuts

Prior implementations used custom keyboard shortcuts that vary across platforms, creating confusion when users transition between environments and increasing the learning curve. This lack of standardization limits interoperability. Our framework establishes a standardized keyboard shortcut schema serving as a baseline for Blockly-based environments (Table 2)

To evaluate the proposed accessibility features and demonstrate the framework's integration capabilities, we developed a custom BBPE built on the Blockly library, integrated with the EAF framework as a reference implementation, which is available on GitHub [19].

Table 2: Keyboard shortcuts for EAF

Scope	Shortcut	Action
Navigation	`W`	↑ Move up to the previous block
Navigation	`A`	← Move left to the previous block
Navigation	`S`	↓ Move down to the next block
Navigation	`D`	→ Move right to the next block
Navigation	`F`	Move to the first nested block or to the first nested connection
Navigation	`Q`	Move out to the parent block or the outer layer
Navigation	`Alt+[A–Z]`	Jump to a stack labeled with that letter
Mode	`E`	Toggle Edit mode (enter or exit)
Workspace	`Shift+W`	↑ Move cursor up
Workspace	`Shift+S`	↓ Move cursor down
Workspace	`Shift+A`	← Move cursor left
Workspace	`Shift+D`	→ Move cursor right
Toolbox	`T`	Open toolbox
Toolbox	`Esc`	Close toolbox; focus workspace
Announce	`C`	Speak/announce cursor location
Edit ops	`Ctrl+X`	Cut selected block (Nav)
Edit ops	`Ctrl+C`	Copy
Edit ops	`Ctrl+V`	Paste detached block to current connection (Edit)
Edit ops	`Delete`	Delete selected block
Edit ops	`Ctrl+/`	Add/Hide comment on selected block (Edit)
Edit ops	`Shift+X`	Disconnect at the current cursor location (Edit)
Assist	`Shift+H`	Toggle navigational assistant
Assist	`Shift+K`	Toggle keyboard shortcuts list
Assist	`Shift+I`	Customize stack label
Execution	`Shift+R`	Run the program
Execution	`Shift+O`	Access the output
Settings	`Ctrl+Shift+K`	Enable or disable keyboard accessibility

6 Evaluation

6.1 Methodology

We conducted a two-phase evaluation process combining the technical verification and expert assessment. The methodology involved integration testing of the EAF followed by a semi-structured interview with 4 participants. The goal is to validate system integration, functional stability, and compatibility with screen readers across heterogeneous systems to test and evaluate accessibility features.

Integration Testing. We followed an integration testing [14] method to evaluate both system-level and feature-level integration of EAF with Blockly-based programming environments. For system-level integration, we employed EAF on two Blockly-based systems: EAF Blockly and an open-source Blockly-based environment to verify the interoperability across different implementations. For feature-level testing, we prepared a comprehensive test suite with 177 test cases targeting all accessibility features with various navigational and editing scenarios. Each test case represents a user action, such as navigating cursors, moving, or inserting blocks. Our testing process covered screen readers such as VoiceOver, JAWS, and NVDA in macOS and Windows operating systems. Then the test cases were assigned among three experienced software developers who participated as testers in the whole testing process.

Semi-structured Interview. We conducted a task-based evaluation comparing the default keyboard navigation with EAF. The objective was to assess the improvements in the navigation flow, screen-reader feedback, and overall user experience within the BBPEs. Sessions were conducted remotely via Zoom. Participants were provided with supporting materials, including setup documentation, video tutorials, and keyboard shortcuts list to familiarize themselves with the environment before the interview.

Each session lasted 120 minutes and was divided into two segments. The first 30 minutes focused on exploring the Blockly default

Table 3: Summary of Participant demographics

ID	Level	Prog. Exp.	SR Fam.	Device / OS	SR Used
P1	Accessibility Expert	Expert	Moderate	Mac (macOS)	VoiceOver
P2	Undergrad	Beginner	None	Laptop (Win 11)	NVDA
P3	Undergrad	Intermediate	None	PC (Win 11)	JAWS
P4	Accessibility Expert	Expert	Minimal	Laptop (Win 11)	JAWS

keyboard navigation. Because this version lacks screen reader support, participants mainly focused on keyboard navigation and orientation. Participants performed three tasks: Navigation (exploring the workspace and locating blocks using keyboard), Create Program (building simple logic sequences to assess structural comprehension), and Edit Program (modifying values, rearranging blocks, and verifying feedback). The remaining 90 minutes were used for EAF with structured 3D navigation, stack labels, block numbers, and editing features. During this phase, participants performed all the tasks using their preferred system with screen reader and keyboard-only interactions and screen readers as stated in Table 3. Feedback for this phase was gathered following completion of all activities.

6.2 Participants and Demographics

We recruited four participants (P1-P4) for human evaluation through accessibility research communities. All participants provided informed consent, and the study was approved by our institutional review board (IRB). The group consists of two accessibility experts and two undergraduate computer science students, representing the range of programming experience and familiarity with screen readers. The expert participants had extensive familiarity with the accessibility tools and practices, while the undergraduate participants represented novice to intermediate users of block-based programming. All participants were sighted, as the evaluation focuses on technical validation of the EAF framework and accessibility design. Table 3 summarizes participants.

Each participant used their personal computers running either a Mac with VoiceOver or a Windows 11 laptop/PC with JAWS or NVDA. All participants received the preparatory documents beforehand to ensure familiarity with the tools and interview process.

6.3 Results

6.3.1 Integration Test Result. We tested the EAF with two open-source Blockly-based environments. The system-level integration was successful, and all EAF modules were functional. System-level validation confirmed full module compatibility without any functional conflicts. Feature-level testing consisted of n=177 individual test cases (see Table 4), covering navigation and movement, mode controls, block editing, stack labeling, accessibility, and general interface functions. Among these 139 tests (78.5 %) passed successfully, while 38 tests (21.5 %) failed initially due to minor issues like label misalignment, shortcut conflicts, and missing ARIA attributes. Subsequent debugging verified 13 valid defects, which were corrected during post-testing refinement, raising the overall pass rate to 152 of 177 tests (85.9%). All core accessibility features, such as 3D navigation, stack jumping, stack labeling, and mode-based editing, remained fully functional across both macOS and Windows environments. Testing with screen readers confirmed reliable keyboard-only support and consistent screen reader compatibility, showing that EAF achieved stable cross-platform integration and satisfied all accessibility performance goals.

Table 4: Categorized feature-level testing result

Category	Tests	Pass	Fail	Bugs Fixed
Navigation & Movement	38	35	3	3
Mode Controls & Interface Management	28	19	9	4
Block Editing & Operations	28	28	0	0
Stack Labeling, Search & Numbering	54	30	24	6
Accessibility & Screen Reader	10	8	2	0
Zoom & View Controls	9	9	0	0
Help & System Features	10	10	0	0
Total Tests	**177**	**139 (78.5%)**	**38 (21.5%)**	**13 (85.9%)**

6.3.2 Participants' Feedback. Participant feedback and behavioral observations were analyzed thematically across four dimensions, namely navigation, creating/editing a program, screen-reader interaction, and recommendations for enhancement.

Navigation Experience Across all sessions, participants reported clearer spatial orientation and less confusion when using EAF compared to the default keyboard navigation. During the default keyboard navigation, P1 struggled to move within the nested blocks, describing it as "*trial and error, sometimes impossible to know what is going on,*" while P4 mentioned it as *"very confusing and feels inconsistent."* and said it lacks feedback when actions fail. In the EAF, P1 successfully moved between the nested print block and a repeat block and observed that "*the navigation was a lot better with F and Q it was clearer and easy".* P4, while traversing multiple stacks, highlighted that the framework *"is more tied to program scope,"* and *"build a mental model that you can move in and out of easily."* Among the undergraduates, P2 mentioned that the new shortcut layout was *"easy to remember,"* and P3 described the stack-jumping feature was *"very helpful, much easier than the other one."*

Editing and Program interaction During editing tasks, participants shared their experience with Blockly's default keyboard navigation and EAF's keyboard navigation. In the default keyboard navigation editing task, P1 struggled with feedback uncertainty while attempting to cut and paste blocks, stating that *"the system status is completely unknown for me. It's not telling me what I can or cannot do."* In the EAF, P1 completed edits seamlessly, reflecting that *"it was good, not hard."* Similarly, P2 faced challenges with editing tasks in the default keyboard navigation but was able to add and edit blocks in the EAF environment, commented *"by exploring more, I can do better."* P3 used Edit mode to modify variable blocks, describing the process as *"pretty easy"*. P4 performed multiple actions, such as rearranging, cutting, and reattaching blocks within the edit mode, and summarized the workflow as *"a lot easier to navigate"*.

Screen Reader Feedback Participants reported that EAF integration with screen readers like JAWS, NVDA, and VoiceOver produced detailed and reliable auditory feedback. P4 observed that *"the screen reader modifies what's being presented based on what you're actually doing. I really like that. It is very specific as to what block or stack you're connected to"*. P2 described the output as *"helpful for navigation"*. With VoiceOver, P1 noted that *"the text was too much, the reading was a bit slow, I wish the voice was faster"* and P3 found JAWS *"too fast and jumbled"*.

Participant Recommendations Participants suggested shorter speech output, clearer indicators for mode changes, and more simplified onboarding materials. During the editing phase, P1 noted that *"each time, I didn't know whether I was in edit mode; it wasn't completely clear even after using it for a few times,"* further adding to that *"students without a screen reader cannot understand it clearly"*. P2 stated that, *"If I practice again, I can remember"*. The experts emphasized feedback improvement, with P1 noting that *"the text was too much, the reading was bit slow, I wish the voice was faster"* also added *"the system status is completely unknown, it is not telling me what I can do or cannot do"* and P4 suggested *"a quick corrective noise or something to tell you that you are doing something wrong"*.

7 Discussion

7.1 Findings

Our evaluation indicates that the EAF successfully addresses the fundamental challenges of making Google Blockly accessible to BVI learners without modifying the library code. The integration testing confirmed the structural stability across different Blockly-based environments, and participants' feedback offered profound insight for improving keyboard navigation with screen readers.

Navigation Model. The positive feedback regarding EAF's keyboard-only navigation indicated that 3D navigation model aligns with the conceptual structure of the programming rather than solely its visual spatial arrangement. P4's observation *("build a mental model that you can move in and out of easily")* suggests that the hierarchical navigation model successfully mapped to participants' understanding of program structure (loops, conditions, nesting). This differs from default navigation, which participants characterized as *"trial and error, sometimes impossible to know what is going on."* The predictability of EAF's directional model seems to make it easier for participants to understand the spatial orientation and reduce cognitive load. This addresses a major limitation found in previous research on accessible block-based - programming. [22, 30]. The fact that undergraduate participants found the shortcuts "easy to remember" while expert participants found them "tied to program scope" shows that the navigation model works for users with different levels of expertise.

Screen Reader Integration. The experiences of participants with VoiceOver, JAWS, and NVDA demonstrate that EAF's WAI-ARIA implementation effectively facilitates compatibility with external screen readers. P4 stated that the feedback was "*very specific as to what block or stack you're connected to,*" and P2 stated it was "*helpful for navigation.*" This demonstrates voice-based audio feedback are an effective way to inform users about the program structure and navigation status. However, the evaluation also showed trade-offs that were specific to screen readers. VoiceOver gave P1 "*too much*" information, and P1 stated that the reading was "a bit slow." P3 noted that JAWS was "*too fast and jumbled.*" These differences show that the primary issue is not that the two systems are fundamentally incompatible, but that they need to be adjusted to fit the user's speech rate and verbosity. Future improvements could prioritize user-configurable verbosity settings over adaptations tailored for specific screen readers. This method avoids the differences between internal synthetic speech and external screen readers documented in previous research [8].

Extension-Based Architecture. Our architectural approach is proven by the fact that EAF works well with two Blockly-based environments without having to modify the core library. Previous accessibility solutions either required forking Blockly's codebase [8] or creating new systems [30]. EAF's extension-based design adds accessibility while being compatible with new Blockly releases. Existing BBPEs such as Scratch, MakeCode, and Code.org can potentially adopt EAF without restructuring their current implementations, lowering the barrier to widespread accessibility adoption. The framework's ability to run on browsers makes it cross-platform compatible and accessible from anywhere.

Comparison to Prior Approaches. EAF's design directly addresses key limitations identified in related work. Where Accessible Blockly [8] required core modifications and lacked screen reader compatibility, EAF maintains architectural separation while supporting external screen readers. Where Blocks4All [18] focused on simplified tasks, EAF supports full Blockly functionality including nested structures. Where default keyboard navigation lacked structure, EAF's hierarchical approach provides predictability. The combination of modularity, screen reader compatibility, and structured navigation represents a synthesis of best practices from prior accessibility research applied to the Blockly ecosystem.

The success of EAF's extension architecture demonstrates that accessible BBPEs need not sacrifice either functionality or modularity. The rapid task completion and positive feedback suggest that keyboard-based block programming, when properly structured, can be viable for BVI learners despite the inherently visual nature of traditional block-based environments. The findings presented here are subject to the limitations discussed in Section 8.

7.2 Takeaways

We have derived the following takeaways for researchers and developers working with BVI users to make BBPEs inclusive.

Accessibility Extensions. Extension-based approaches like EAF can offer a practical pathway to improving accessibility in existing BBPEs. By providing a shippable, modular solution, it can enable accessibility without requiring platforms to implement accessibility support from scratch. This approach can lower the barrier to the adoption of accessible BBPEs in classrooms.

Standard Keyboard Shortcuts. Our design establishes a standard set of keyboard shortcuts for BBPEs. These standards can help BVI users to develop transferable skills and switch between platforms without needing to relearn shortcuts. Broad adoption of proposed standards could foster a more inclusive and interoperable ecosystem of BBPEs.

Accessible UI Elements. Sighted users intuitively perceive visual cues in UI components such as blocks and stacks to comprehend code, which are inaccessible to BVI users. Enhancements to Blockly UI components, such as stack labels and block numbers, can help BVI learners better understand program structure and comprehend code. We recommend that researchers and developers explore designing block-based UI elements with accessibility in mind that compensate for visual cues for BVI users.

8 Limitations

Our study demonstrates the viability of EAF, but we acknowledge several limitations that should be addressed in future studies.

Evaluation Scope. We evaluated the framework involving experts. The accessibility experts offered key insights into screen reader compatibility, interaction patterns and overall accessibility design, we lack direct feedback from BVI participants. Also, all participants were adults, while most BBPEs targets K-12 learners. Testing with BVI students in real classroom settings would provide critical understandings into the EAF's effectiveness and usability.

Platform Validation. We designed EAF to work with any Blockly-based platform. We tested the integration using open-source Blockly distributions. However, EAF compatibility with public BBPEs such as MakeCode or Code.org remains unknown, as the system is proprietary.

Assistive Technology Coverage. We successfully validated EAF with three major screen readers, but our testing did not include refreshable braille displays, which some BVI programmers prefer. We tested keyboard-based zoom controls, however, how it renders with screen magnification software for low-vision users needs further evaluation.

Despite these limitations, our results indicate that EAF architecture can make BBPEs accessible for BVI users while maintaining platform compatibility and integrity for widespread adoption, a strategy that has been largely overlooked in previous studies.

9 Conclusion and Future Work

This study presents the Extension-Based Accessibility Framework (EAF) for Google Blockly. The frameworks successfully integrate with Blockly-based environments without requiring modifications to Blockly library. The results indicate that it makes Blockly-based BBPEs fully accessible to BVI learners through keyboard-only navigation and support for external screen readers. Our design introduces a 3D navigation model with accessible block UI elements and mode-based context-aware program editing. Our evaluation and feedback from accessibility experts and computer science students validate that these efforts help to improve code comprehension, understanding program structure, accessibility of code navigation, and editing.

In our future work, we will conduct comprehensive user studies engaging BVI learners to assess the EAF's effectiveness in block-based programming learning in classroom settings. Additionally, our current work focuses on the EAF architecture and accessible code navigation and editing models. In future work, we will explore accessible debugging methods and techniques for making graphical outputs, such as animations and interactive visualizations, accessible to BVI users.

Acknowledgments

We thank Google for funding this work through the Blockly Accessibility Fund initiative.

References

[1] GitHub, Inc. 2025. *Node Package Manger.* GitHub, Inc. https://www.npmjs.com/ Accessed: October 22, 2025.

[2] Khaled Albusays, Stephanie Ludi, and Matt Huenerfauth. 2017. Interviews and observation of blind software developers at work to understand code navigation challenges. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility*. 91–100.

[3] Apple Inc. 2025. *VoiceOver User Guide for Mac.* https://support.apple.com/guide/voiceover/welcome/mac Apple Support.

[4] Thomas Ball, Abhijith Chatra, Peli de Halleux, Steve Hodges, Michał Moskal, and Jacqueline Russell. 2019. Microsoft MakeCode: embedded programming for education, in blocks and TypeScript. In *Proceedings of the 2019 ACM SIGPLAN symposium on SPLASH-E*. 7–12.

[5] Katarina Bulovic, Zoë Bentley, and Natalie Rusk. 2024. Designing for Tinkerability and Accessibility: Developing the OctoStudio mobile app to engage blind and visually impaired learners in creating with code. In *Proceedings of the 23rd Annual ACM Interaction Design and Children Conference*. 882–886.

[6] Code.org. 2024. Why does Code.org use Blockly, a visual programming language, for its elementary-level courses? https://support.code.org/hc/en-us/articles/202518363. Accessed: October 22, 2025.

[7] James Craig, Michael Cooper, L Pappas, R Schwerdtfeger, and L Seeman. 2009. Accessible rich internet applications (WAI-ARIA) 1.0. *W3C Working Draft* (2009).

[8] Meenakshi Das, Daniela Marghitu, Mahender Mandala, and Ayanna Howard. 2021. Accessible block-based programming for k-12 students who are blind or low vision. In *International Conference on Human-Computer Interaction*. Springer, 52–61.

[9] Vlad Filippov. 2023. *Building Your Own JavaScript Framework: Architect Extensible and Reusable Framework Systems.* Packt Publishing Ltd.

[10] Freedom Scientific. 2025. *JAWS Screen Reader Documentation.* https://support.freedomscientific.com/products/blindness/jawsdocumentation Freedom Scientific Support.

[11] Google Blockly Team. 2024. Keyboard Navigation. https://developers.google.com/blockly/guides/configure/web/keyboard-nav. Official documentation for Blockly's experimental keyboard navigation plugin. Accessed: October 22, 2025.

[12] Google Developers. 2025. *Google Blockly Library.* https://developers.google.com/blockly Official documentation. Accessed: October 22, 2025.

[13] Mohammed Hassan, Kathryn Cunningham, and Craig Zilles. 2023. Evaluating Beacons, the Role of Variables, Tracing, and Abstract Tracing for Teaching Novices to Understand Program Intent. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*. 329–343.

[14] Florentin Ipate and Mike Holcombe. 1997. An integration testing method that is proved to find all faults. *International journal of computer mathematics* 63, 3-4 (1997), 159–178.

[15] Filiz Kalelioğlu. 2015. A new way of teaching programming skills to K-12 students: Code. org. *Computers in Human Behavior* 52 (2015), 200–210.

[16] Varsha Koushik and Clayton Lewis. 2016. An accessible blocks language: work in progress. In *Proceedings of the 18th international ACM SIGACCESS conference on computers and accessibility*. 317–318.

[17] Stephanie Ludi and Mary Spencer. 2017. Design Considerations to Increase Block-based Language Accessibility for Blind Programmers Via Blockly. *J. Vis. Lang. Sentient Syst.* 3, 1 (2017), 119–124.

[18] Lauren R Milne and Richard E Ladner. 2018. Blocks4All: overcoming accessibility barriers to blocks programming for children with visual impairments. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–10.

[19] Rubel Hassan Mollik and Vamsi Krishna Kosuri. 2025. *EAF Blockly Environment.* https://rubelhassan.github.io/eaf-blockly Accessed: October 27, 2025.

[20] Aboubakar Mountapmbeme and Stephanie Ludi. 2020. Investigating challenges faced by learners with visual impairments using block-based programming/hybrid environments. In *Proceedings of the 22nd International ACM SIGACCESS Conference on Computers and Accessibility*. 1–4.

[21] Aboubakar Mountapmbeme and Stephanie Ludi. 2021. How teachers of the visually impaired compensate with the absence of accessible block-based languages. In *Proceedings of the 23rd International ACM SIGACCESS Conference on Computers and Accessibility*. 1–10.

[22] 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*. 1–15.

[23] NV Access. 2025. *NVDA 2025.3 User Guide.* https://download.nvaccess.org/releases/2025.3/documentation/userGuide.html NV Access.

[24] 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*. 1–5.

[25] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[26] Rafi Santo, Leigh Ann DeLyser, June Ahn, Anthony Pellicone, Julia Aguiar, and Stephanie Wortel-London. 2019. Equity in the who, how and what of computer science education: K12 school district conceptualizations of equity in 'cs for all'initiatives. In *2019 research on equity and sustained participation in engineering,*

computing, and technology (RESPECT). IEEE, 1–8.

[27] Misbahu Sharfuddeen Zubair, David Brown, Matthew Bates, and Thomas Hughes-Roberts. 2020. Are visual programming tools for children designed with accessibility in mind?. In *Proceedings of the 12th International Conference on Education Technology and Computers*. 37–40.

[28] 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*. 1286–1292.

[29] MM Mahbubul Syeed, Alexander Lokhman, Tommi Mikkonen, and Imed Hammouda. 2015. Pluggable systems as architectural pattern: An ecosystemability perspective. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*. 1–6.

[30] Moumita Tabassum, Jeff Gray, and Derrick Smith. 2024. An Accessible Blocks Language for Students with and without Visual Impairments. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. 1300–1306.

[31] Emmanuel Utreras and Enrico Pontelli. 2020. Accessibility of block-based introductory programming languages and a tangible programming tool prototype. In *International Conference on Computers Helping People with Special Needs*. Springer, 27–34.

[32] David Weintrop. 2019. Block-based programming in computer science education. *Commun. ACM* 62, 8 (2019), 22–25.

[33] David Weintrop and Uri Wilensky. 2017. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education (TOCE)* 18, 1 (2017), 1–25.